

1 Win32课程介绍

很多人对Win32的认识是错误的，他们认为Win32就是画界面，都已经学MFC了还学什么Win32？

Win32不是用来画界面的，如果你以后要在Windows写好程序，是必须要学Win32的；摆正学习态度。

2 字符编码

我们会经常接触到各种各样的字符编码，本章节就来讲解一下常见的编码。

2.1 原始的ASCII编码

计算机是由美国人发明的，所以一开始设计编码的时候只会考虑到自身的元素，采用ASCII编码完全可以满足其需求，但是计算机普及之后，很多国家的文字是象形文字，所以我们使用ASCII编码是无法满足需求的。

ASCII表																					
		ASCII控制字符								ASCII打印字符											
		0000				0001				0010		0011		0100		0101		0110		0111	
高四位 低四位	十进制 字符 Ctrl 代码 转义 字符 字符解释	0000				0001				十进制 字符 Ctrl	十进制 字符 Ctrl 代码 转义 字符 字符解释	0010	0011	0100	0101	0110	0111	0111			
		0	1	2	3	4	5	6	7												
0000	0	0	^@ NUL \0	空字符	16 ▶ ^P DEL	数据链路转义	32 48 0 @	64 P	96 `	112 p											
0001	1	1	^A SOH	标题开始	17 ◀ ^Q DC1	设备控制 1	33 !	49 1 A	81 Q	97 a	113 q										
0010	2	2	^B STX	正文开始	18 ↴ ^R DC2	设备控制 2	34 "!	50 2 B	82 R	98 b	114 r										
0011	3	3	^C ETX	正文结束	19 !! ^S DC3	设备控制 3	35 #	51 3 C	83 S	99 c	115 s										
0100	4	4	^D EOT	传输结束	20 ¶ ^T DC4	设备控制 4	36 \$	52 4 D	84 T	100 d	116 t										
0101	5	5	^E ENQ	询问	21 § ^U NAK	否定应答	37 %	53 5 E	85 U	101 e	117 u										
0110	6	6	^F ACK	肯定应答	22 — ^V SYN	同步空闲	38 &	54 6 F	86 V	102 f	118 v										
0111	7	7	^G BEL \a	响铃	23 ↑ ^W ETB	传输块结束	39 ^	55 7 G	87 W	103 g	119 w										
1000	8	8	^H BS \b	退格	24 ↑ ^X CAN	取消	40 (56 8 H	88 X	104 h	120 x										
1001	9	9	^I HT \t	横向制表	25 ↓ ^Y EM	分离结束	41)	57 9 I	89 Y	105 i	121 y										
1010	A	10	^J LF \n	换行	26 → ^Z SUB	替代	42 *	58 :	74 J	90 Z	106 j	122 z									
1011	B	11	^K VT \v	纵向制表	27 ← ^[ESC [退出	43 +	59 ;	75 K	91 [107 k	123 {									
1100	C	12	^L FF \f	换页	28 ↳ ^] RS	文件分隔符	44 ,	60 <	76 L	92 \	108 I	124									
1101	D	13	^M CR \r	回车	29 ↔ ^[CS	组分隔符	45 -	61 =	77 M	93]	109 m	125 }									
1110	E	14	^N SO	移出	30 ▲ ^A ES	记录分隔符	46 .	62 >	78 N	94 ^	110 n	125 ~									
1111	F	15	^O SI	移入	31 ▼ ^- US	单元分隔符	47 /	63 ?	79 O	95 _	111 o	127 □									
注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。																					
2013/08/08																					

2.2 ASCII编码的拓展：GB2312或GB2312-80

由于ASCII编码无法满足需求，所以在其基础上进行扩展，大家都知道ASCII编码只有0-127，用十六进制表示是0x00-0x7F，而之后的0x80-0xFF在标准的ASCII编码中是不存在的，所以就出现了我们所说的ASCII编码的扩展。

但是这样能满足中文、韩文这种象形文字吗？其实并不可以，如上这张表实际上使用频率很低，而这时候
GB2312编码（该编码与GBK没有什么本质区别，无非就是收录的汉字和图形符号的区别：GB2312标准共收录
6763个汉字，GBK共收入21886个汉字和图形符号）考虑到这个因素就占用了这张表，那么是怎么占用的呢？
其本质就是创建两张如上图所示的表，然后进行拼接，两个字节加起来就组成一个新的中文文字。

例如：中国的“中”这个字，就是0xD0和0xD6拼接起来的。这种编码是否存在问题是必然的，我们已经知道了该编码的设计原理，假设我们将“中国”这两个字发给国外的朋友，他的电脑上并没有该编码表，所以解析出来的则不会是汉字，而会出现大家所熟知的“乱码”。

2.3 Unicode编码

为了解决ASCII的缺陷，Unicode编码就诞生了，那么Unicode是如何解决这一问题的呢？

其实很简单，Unicode编码创建了一张包含世界上所有文字的编码表，只要世界上存在的文字符号，都会赋予一个唯一的编码。

Unicode编码的范围是：0x0-0x10FFFF，其可以容纳100多万个符号，但是Unicode本身也存在问题，因为Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何去存储。

假设中这个字以Unicode方式表示占2个字节，而国这个字却占4个，这个时候你该如何去存储？

2.4 Unicode存储的实现方式

2.4.1 UTF-16

UTF-16/UTF-8是Unicode存储的实现方式；UTF-16编码是以16个无符号整数位单位，注意是16位为一个单位，但不表示一个字符就只有16位，具体的要看字符的Unicode编码所在范围，有可能是2字节，有可能是4字节，现在机器上的Unicode编码一般指的就是UTF-16。

我们举个例子（虚构）：

中（Unicode编码）：0x1234

国（Unicode编码）：0x12345

UTF-16存储的时候，“中”这个字肯定是存储的0x1234，但是“国”这个字就不一样，我们知道UTF-16是16位（2字节）为一个单位，所以国这个字拆下来存储应该是0x00 0x01 0x23 0x45。

UTF-16的优点一看便知：计算、拆分、解析非常方便，2个字节为一个单位，一个一个来。

UTF-16是否是最优解呢？其实不然，我们通过如上的例子中可以看到一个很明显的缺点，那就是UTF-16会存在浪费空间的情况，因为其16位（2字节）为一个单位，它需要字节对齐，例如字母A只需要一个字节就可以表示，而使用UTF-16时就会变成2个字节，所以很浪费，而这时候UTF-8横空出世。（UTF-16在本地存储是没有啥问题的，顶多就是浪费一点硬盘空间，但是如果在网络中传输，那就太过于浪费了）

2.4.2 UTF-8

UTF-8称之为可变长存储方案，其存储根据字符大小来分配，例如字母A就分配一个字节，汉字“中”就分配两个字节。

优点：节省空间；缺点：解析很麻烦。

UTF-8存储的方式是有对应表的：

Unicode编码（16进制）	UTF-8字节流（二进制）
0x000000 - 0x00007F	0XXXXXXX
0x000080 - 0x0007FF	110XXXXX 10XXXXXX
0x000800 - 0x00FFFF	1110XXXX 10XXXXXX 10XXXXXX
0x010000 - 0x10FFFF	11110XXX 10XXXXXX 10XXXXXX 10XXXXXX

例如字母A，在0x000000 - 0x00007F范围之间，则采用0XXXXXXX的方式进行存储，也就是按照一个字节的方式来不会改变什么，而汉字“中”则不一样了。

中（Unicode编码）：0x4E 0x2D，它属于0x000800 - 0x00FFFF范围之间。

$0x4E\ 0x2D = 0100\ 1110\ 0010\ 1101$ ，其以UTF-8的方式存储就是1110 (0100) 10(11 1000) 10(10 1101)，括号包裹起来的就是汉字“中”的Unicode编码。

最后一个问题，假设我们把UTF-8的文本格式发给对方，那对方如果按照UTF-16的方式去解析该怎么办？如何让对方只采用UTF-8的方式去解析呢？

2.5 BOM (Byte Order Mark)

BOM中文为字节顺序标记，其就是用来插入到文本文件起始位置开头的，用于识别Unicode文件的编码类型。

对应关系如下：

UTF-8	EF EB BF
UTF-16LE（小端存储）	FF FE
UTF-16BE（大端存储）	FE FF

3 C语言中的宽字符

本章主要是讲解在C语言中如何使用上一章所述的编码格式表示字符串。

ASCII码：char strBuff[] = "中国";

Unicode编码（UTF-16）：wchar_t strBuff[] = L"中国"; // 这里需要在双引号之前加上L是因为如果你不加的话，编译器会默认使用当前文件的编码格式去存储，所以我们需要加上。（注意使用这个的时候需要包含stdio.h这个头文件）

The screenshot illustrates the difference between ASCII and Unicode encoding for the string "中国".

Source Code:

```
#include <stdio.h>

void main() {
    char strBuff[] = "中国";
    wchar_t strBuff1[] = L"中国";
    return;
}
```

Memory Dump (Address 12ff78): Shows the ASCII representation of the string "中国" as D6 D0 B9 FA 00. The character '中' is represented by the bytes D6 D0, and '国' by B9 FA.

地址	12ff78	值	内容
0012FF78	D6 D0 B9 FA 00		中国
0012FF7D	CC CC CC C0 FF		烫汤
0012FF82	12 00 59 11 40		..Y@
0012FF87	00 01 00 00 00	
0012FF8C	20 0E 37 00 B8		.7...
0012FF91	0E 37 00 00 00		.7...
0012FF96	00 00 00 00 00	
0012FF9B	00 00 40 FD 7F		..@?
0012FFA0	01 00 00 00 06	
0012FFA5	00 00 00 94 FF	
0012FFAA	12 00 08 1D C6	
0012FFAF	B1 E0 FF 12 00 26		编辑...&

Memory Dump (Address 12ff70): Shows the Unicode representation of the string "中国" as 2D 4E FD 56 00 00 CC. The character '中' is represented by 2D 4E, and '国' by FD 56.

地址	12ff70	值	内容
0012FF70	2D 4E FD 56 00 00 CC		-N... 汉
0012FF77	CC D6 D0 B9 FA 00 CC		讨论...
0012FF7E	CC CC C0 FF 12 00 59		烫....Y
0012FF85	11 40 00 01 00 00 00		@.....
0012FF8C	20 0E 37 00 B8 0E 37		.7...7
0012FF93	00 00 00 00 00 00 00	
0012FF9A	00 00 00 40 FD 7F 01		..@?.
0012FFA1	00 00 00 06 00 00 00	
0012FFA8	94 FF 12 00 08 1D C6	
0012FFAF	B1 E0 FF 12 00 26		编辑...&
0012FFB6	40 00 30 20 42 00 00		@.0 B..
0012FFBD	00 00 00 F0 FF 12 00	

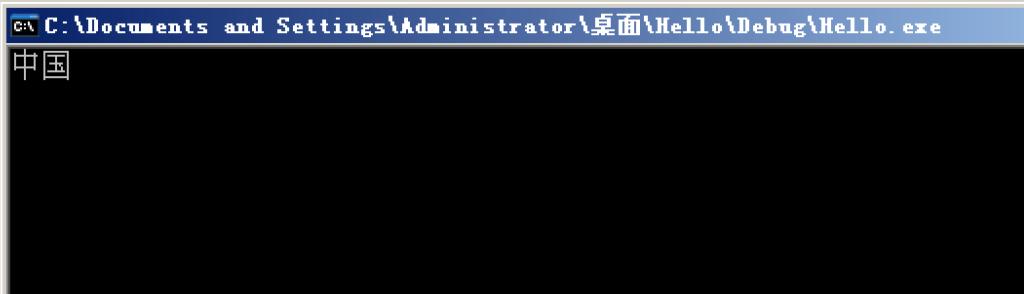
Unicode编码这种表现形式实际上就是宽字符，所以在提起宽字符的时候我们就应该想到这种方式。

ASCII编码和Unicode编码在内存中的存储方式不一样，所以我们使用相关函数的时候也要注意，如下图所示，ASCII编码使用左边的，而Unicode则是右边的：

char	wchar_t	//多字节字符类型 宽字符类型
printf	wprintf	//打印到控制台函数
strlen	wcslen	//获取长度
strcpy	wcsncpy	//字符串复制
strcat	wcsncat	//字符串拼接
strcmp	wcsncmp	//字符串比较

例如我们想要在控制台中打印一个宽字符的字符串：

```
#include <locale.h>
#include <stdio.h>
void main() {
    setlocale(LC_ALL, ""); // 设置控制台的默认编码
    wchar_t strBuff[] = L"中国";
    wprintf(L"%s \n", strBuff);
    return;
}
```



再一个例子就是字符串的长度：

```
(i) char strBuff[] = "China";
wchar_t strBuff1[] = L"China";
strlen(strBuff); //取得多字节字符串中字符长度，不包含 00
wcslen(strBuff1); //取得多字节字符串中字符长度，不包含 00 00
```

4 Win32 API中的宽字符

4.1 了解什么是Win32 API

Win32 API就是Windows操作系统提供给我们的函数（应用程序接口），其主要存放在C:\Windows\System32（存储的DLL是64位）、C:\Windows\SysWOW64（存储的DLL是32位）下面的所有DLL文件（几千个）。

重要的DLL文件：

1. Kernel32.dll：最核心的功能模块，例如内存管理、进程线程相关的函数等；
2. User32.dll：Windows用户界面相关的应用程序接口，例如创建窗口、发送信息等；
3. GDI32.dll：全称是Graphical Device Interface（图形设备接口），包含用于画图和显示文本的函数。

在C语言中我们想要使用Win32 API的话直接在代码中包含windows.h这个头文件即可。

比如我们想要弹出一个提示窗口，Win32 API文档中弹窗API的格式如下：

```

1 int MessageBox(
2     HWND hWnd,           // handle to owner window
3     LPCTSTR lpText,      // text in message box
4     LPCTSTR lpCaption,   // message box title
5     UINT uType          // message box style
6 );

```

这个代码可能看起来非常可怕，好像我们都没有接触过，但实际上其不是什么新的类型，所谓的新的类型无非就是给原有的类型重新起了一个名字，这样做是为了将所有类型统一化，便于读写，如果涉及到跨平台的话将原来的类型修改一下就好了，无需对代码进行重写。

例如以上代码中的类型LPCTSTR，实际上我们跟进一下代码（选中F12）会发现其本质就是const char *这个类型，只不过是换了一个名字罢了。

常用的数据类型在Win32中都重新起了名字：

汇编：

byte	BYTE	PBYTE
word	WORD	PWORD
dword	DWORD	PDWORD

C语言：

char	CHAR	PCHAR
unsigned char	UCHAR	PUCHAR
short	SHORT	PSHORT
unsigned short	USHORT	PUSHORT
int	INT	PINT
unsigned int	UINT	PUINT

C++语言：

bool	BOOL
------	------

4.2 在Win32中使用字符串

字符类型：

① CHAR strBuff[] = "中国"; // char
 WCHAR strBuff[] = L"中国"; // wchar_t
 TCHAR strBuff[] = TEXT("中国"); // TCHAR 根据当前项目的编码自动选择char还是wchar_t，在Win32中推荐使用这种方式

字符串指针：

① PSTR strPoint = "中国"; // char*
 PWSTR strPoint = L"中国"; // wchar_t*
 PTSTR strPoint = TEXT("中国"); // PTSTR 根据当前项目的编码自动选择如char*还是wchar_t*，在Win32中推荐使用这种方式

4.3 使用Win32 API弹框

之前我们了解到Win32 API中的弹框，其名称为MessageBox，其实上本质就是MessageBoxW和MessageBoxA：

```
#ifdef UNICODE
#define MessageBox MessageBoxW
#else
#define MessageBox MessageBoxA
#endif // !UNICODE
```

MessageBoxA只接受ASCII编码的参数，而MessageBoxW则只接受Unicode编码的参数。

从本质上讲，Windows字符串都是宽字符的，所以使用**MessageBoxW这种方式性能会更好一些**，因为当你使用MessageBoxA的时候，在到内核的时候（系统底层）其会转化Unicode，所以性能相对差一些。

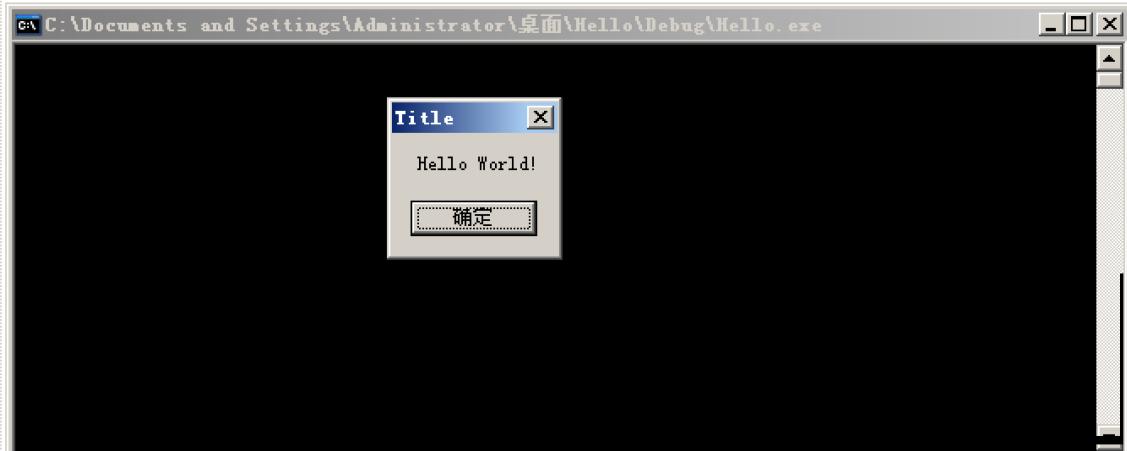
弹框调用如下：

```
1 CHAR strTitle[] = "Title";
2 CHAR strContent[] = "Hello World!";
3 MessageBox(0, strContent, strTitle, MB_OK);
4
5 WCHAR strTitle[] = L"Title";
6 WCHAR strContent[] = L"Hello World!";
7 MessageBoxW(0, strContent, strTitle, MB_OK);
8
9 TCHAR strTitle[] = TEXT("Title");
10 TCHAR strContent[] = TEXT("Hello World!");
11 MessageBox(0, strContent, strTitle, MB_OK);
```

```
#include <windows.h>

void main() {
    WCHAR strTitle[] = L"Title";
    WCHAR strContent[] = L"Hello World!";
    MessageBoxW(0, strContent, strTitle, MB_OK);

    return;
}
```



5 进程的创建过程

5.1 什么是进程

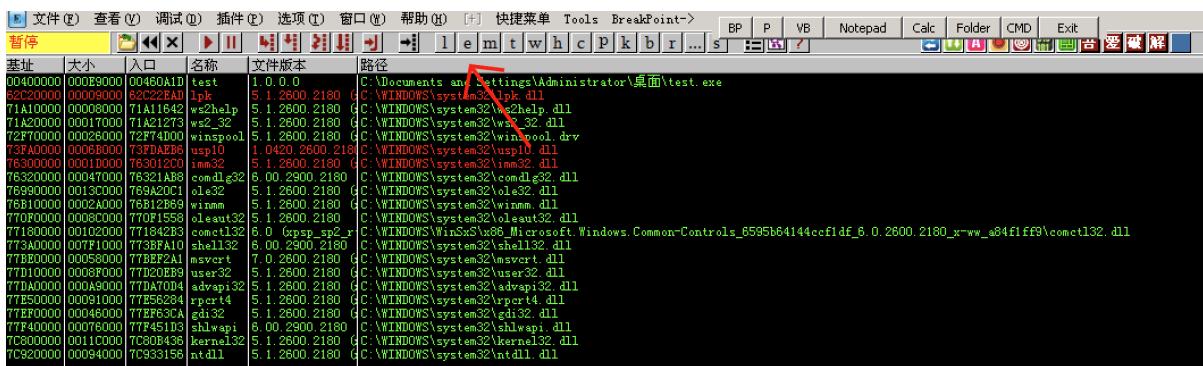
程序所需要的资源（数据、代码...）是由进程提供的；进程是一种空间上的概念，它的责任就是提供资源，至于资源如何使用，与它无关。

每一个进程都有自己的一个4GB大小的虚拟空间，也就是从0x0-0xFFFFFFFF这个范围。

进程内存空间的地址划分如下，每个进程的内核是同一份（高2G），只有其他三个分区是进程独有的（低2G），而只有用户模式区是我们使用的范围：



进程也可以理解为是一对模块组成的，我们可以使用OD打开一个进程看一下：



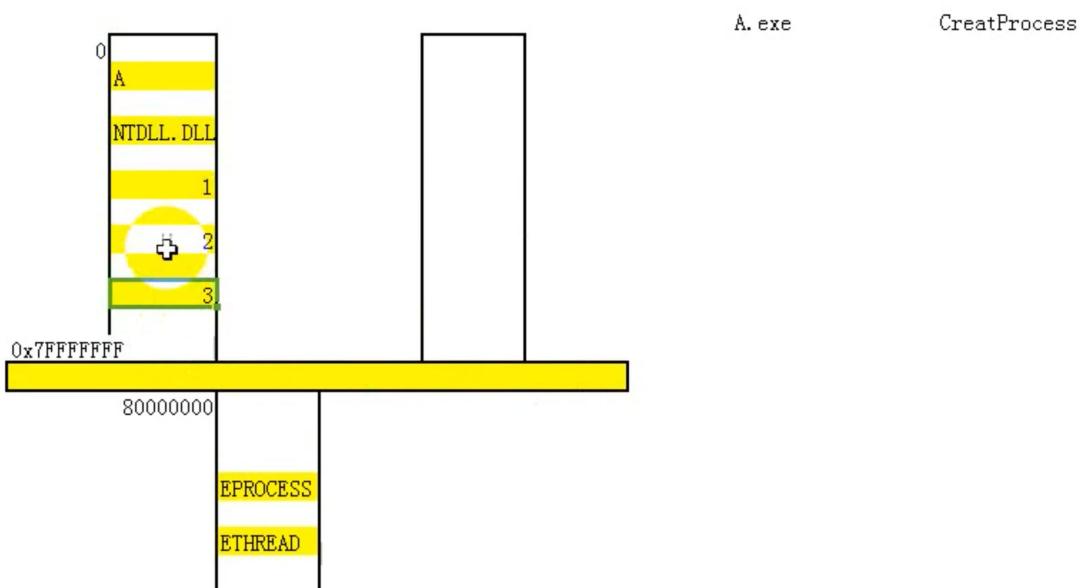
这里面有很多的模块，每个模块都是一个可执行文件，它们遵守相同的格式，即PE结构，所以我们也可以说进程就是一堆PE组合。

5.2 进程的创建

我们需要知道任何进程都是别的进程创建的，当我们在Windows下双击打开一个文件，实际上就是explore.exe这个进程创建的我们打开文件的进程，其使用的方法就是：CreateProcess()

进程创建的过程也就是**CreateProcess**函数：

1. 映射EXE文件（低2G）
2. 创建内核对象EPROCESS（高2G）
3. 映射系统DLL（ntdll.dll）
4. 创建线程内核对象RTHREAD（高2G）
5. 系统启动线程：
 - a. 映射DLL（ntdll.LdrInitializeThunk）
 - b. 线程开始执行



如上图就是打开A.exe的创建过程图，进程是空间上的概念，只用于提供代码和数据资源等等...而想要使用这些资源的是线程，每个进程至少需要一个线程。

6 创建进程

创建进程的函数是CreateProcess()，这个函数的使用方法如下：

```

1  BOOL CreateProcess(
2      LPCTSTR lpApplicationName,           // name of executable module 进程名 (完整文件路径)
3      LPTSTR lpCommandLine,              // command line string 命令行传参
4      LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD 进程句柄
5      LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD 线程句柄
6      BOOL bInheritHandles,             // handle inheritance option 句柄
7      DWORD dwCreationFlags,            // creation flags 标志
8      LPVOID lpEnvironment,            // new environment block 父进程环境变量
9      LPCTSTR lpCurrentDirectory,       // current directory name 父进程目录作为当前目录, 设置目
录
10     LPSTARTUPINFO lpStartupInfo,        // startup information 结构体详细信息 (启动进程相关信息)
11     LPPROCESS_INFORMATION lpProcessInformation // process information 结构体详细信息 (进程ID、线程ID、
进程句柄、线程句柄)
12 );

```

本章节对**CreateProcess函数**的了解就是前2个参数和后2个参数，前两个参数：**lpApplicationName**、**lpCommandLine**，第一个是需要启动的进程文件路径，第二个是命令行参数，如果你启动的进程有参数的可以可以传入。

命令行参数是指在CMD命令行下运行程序所需要提供的参数，例如我们的main入口函数：

```

1 int main(int argc, char* argv[])
2 {
3     printf("%s - %s", argv[0], argv[1]);
4     return 0;
5 }

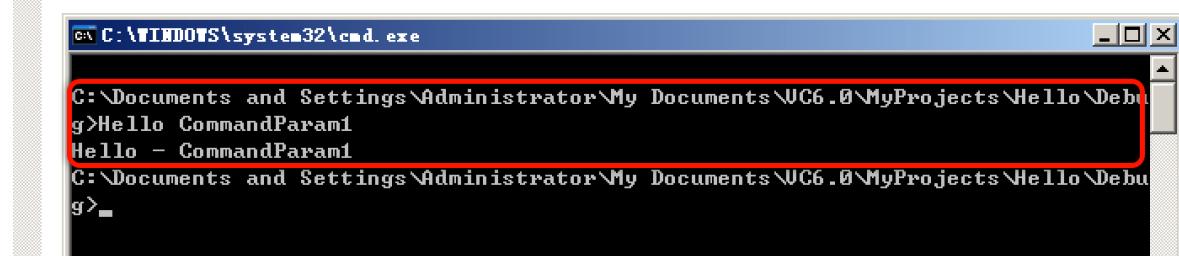
```

其函数传参char* argv[]就是命令行参数，要使用的话就是argv[0]则表示程序本身，其余往后则是参数，argv[1]、argv[2]...等等：

```

int main(int argc, char* argv[])
{
    printf("%s - %s", argv[0], argv[1]);
    return 0;
}

```



所以我们要使用**CreateProcess函数**创建进程的话，如果需要提供命令行参数则需要填写第二个参数**lpCommandLine**：

```

#include <windows.h>
int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:/WINDOWS/system32/cmd.exe"); // process
    TCHAR childProcessCommandLine[] = TEXT(" /c ping www.baidu.com"); // command line

    // LPSTARTUPINFO lpStartupInfo
    STARTUPINFO si;
    // LPPROCESS_INFORMATION lpProcessInformation
    PROCESS_INFORMATION pi;

    // 用0来填充si pi
    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    // LPSTARTUPINFO lpStartupInfo 是一个结构体，其他成员都可以不写，但是cb这个成员是用来表示结构体本身大小的，所以必须要写入值
    si.cb = sizeof(si);

    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        printf("CreateProcess Successfully!");
    } else {
        printf("CreateProcess Error: %d \n", GetLastError());
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    return 0;
}

```

```

1 #include <windows.h>
2 #include <stdlib.h>
3
4 int main(int argc, char* argv[])
5 {
6     TCHAR childProcessName[] = TEXT("C:/WINDOWS/system32/cmd.exe");
7     TCHAR childProcessCommandLine[] = TEXT(" /c ping 127.0.0.1");
8
9     STARTUPINFO si;
10    PROCESS_INFORMATION pi;
11
12    ZeroMemory(&si, sizeof(si));
13    ZeroMemory(&pi, sizeof(pi));
14
15    si.cb = sizeof(si);
16
17    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, 0, NULL, NULL,
&si, &pi)) {
18        printf("CreateProcess Successfully! \n");
19    } else {
20        printf("CreateProcess Error: %d \n", GetLastError());
21    }
22
23    CloseHandle(pi.hProcess);
24    CloseHandle(pi.hThread);
25
26    system("pause");
27    return 0;
28}

```

如上图所示代码，首先我定义了进程路径、进程命令行参数，其次**创建了si、pi两个结构体**，然后使用**ZeroMemory函数用0填充结构体数据**，再给**si.cb**成员赋值当前结构体大小（为什么需要？这是因为Windows会有很多个版本，便于未来更新换代）；最后**CreateProcess函数创建进程**，由于CreateProcess函数本身返回值是布尔类型的，所以使用if来判断，如果出问题则使用**GetLastError函数来获取问题编号**，具体编号对应什么内容可以参考百度百科：<https://baike.baidu.com/item/GetLastError/4278820?fr=aladdin>

在创建完进程之后需要关闭进程，但是我们所知道，每个进程至少有一个线程，所以我们也要关闭线程，使用**CloseHandle函数**来关闭。

```
C:\Documents and Settings\Administrator\My Documents\UC6.0\MyProjects>Hello\Debug>Hello
CreateProcess Successfully!
C:\Documents and Settings\Administrator\My Documents\UC6.0\MyProjects>Hello\Debug>
Pinging www.a.shifen.com [180.101.49.12] with 32 bytes of data:

Reply from 180.101.49.12: bytes=32 time=10ms TTL=52
Reply from 180.101.49.12: bytes=32 time=12ms TTL=52
Reply from 180.101.49.12: bytes=32 time=12ms TTL=52
Reply from 180.101.49.12: bytes=32 time=10ms TTL=52

Ping statistics for 180.101.49.12:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 10ms, Maximum = 12ms, Average = 11ms
```

6.1 课外扩展-反调试(STARTUPINFO结构体)

CreateProcess()函数创建进程，其有一个参数是**STARTUPINFO结构体**，这个参数是进程启动的一些信息，我们一开始会将其ZeroMemory()函数处理，填充0，那么在运行的时候是否还都是0呢？或者说在载入调试工具的时候所有成员是否都是0呢？

首先我们来看一下**STARTUPINFO结构体**的成员：

1	typedef struct _STARTUPINFOA {
2	DWORD cb;
3	LPSTR lpReserved;
4	LPSTR lpDesktop;
5	LPSTR lpTitle;
6	DWORD dwX;
7	DWORD dwY;
8	DWORD dwXSize;
9	DWORD dwYSize;
10	DWORD dwXCountChars;
11	DWORD dwYCountChars;
12	DWORD dwFillAttribute;
13	DWORD dwFlags;
14	WORD wShowWindow;
15	WORD cbReserved2;
16	LPBYTE lpReserved2;
17	HANDLE hStdInput;
18	HANDLE hStdOutput;
19	HANDLE hStdError;
20	} STARTUPINFOA, *LPSTARTUPINFOA;

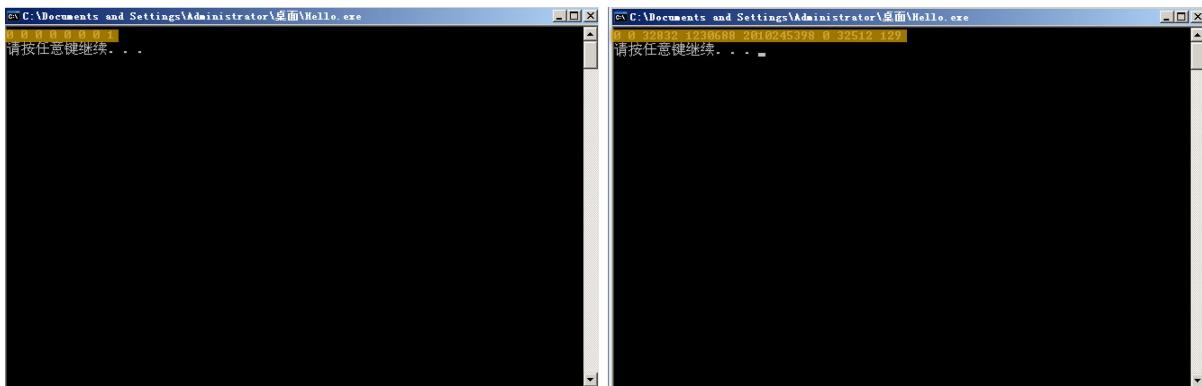
将这几个**DWORD**类型的成员打印一下看看，通过**GetStartupInfo**函数来获取信息：

```

1 #include "stdafx.h"
2 #include <windows.h>
3 #include <stdlib.h>
4
5 int main(int argc, char* argv[])
6 {
7     STARTUPINFO si;
8     ZeroMemory(&si, sizeof(si));
9     si.cb = sizeof(si);
10
11     GetStartupInfo(&si);
12
13     printf("%d %d %d %d %d %d %d\n", si.dwX, si.dwY, si.dwXSize, si.dwYSize, si.dwXCountChars,
14     si.dwYCountChars, si.dwFillAttribute, si.dwFlags);
15     system("pause");
16     return 0;
}

```

正常打开（P1）和在DTDebug调试工具（P2）中打开：



我们可以很清楚的看见了几个值在调试工具中打开发生变化：**si.dwXSize**, **si.dwYSize**, **si.dwXCountChars**, **si.dwFillAttribute**, **si.dwFlags**

所以我们可以根据这几个值来判断从而进行反调试：

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    STARTUPINFO si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);

    GetStartupInfo(&si);

    if ((int)si.dwXSize != 0 || (int)si.dwYSize != 0 || (int)si.dwXCountChars != 0 || (int)si.dwFillAttribute != 0 || (int)si.dwFlags != 0)
        exit(0);
    else {
        printf("%d %d %d %d %d %d %d\n", si.dwX, si.dwY, si.dwXSize, si.dwYSize, si.dwXCountChars, si.dwYCountChars, si.dwFillAttribute);
    }

    system("pause");
    return 0;
}

```

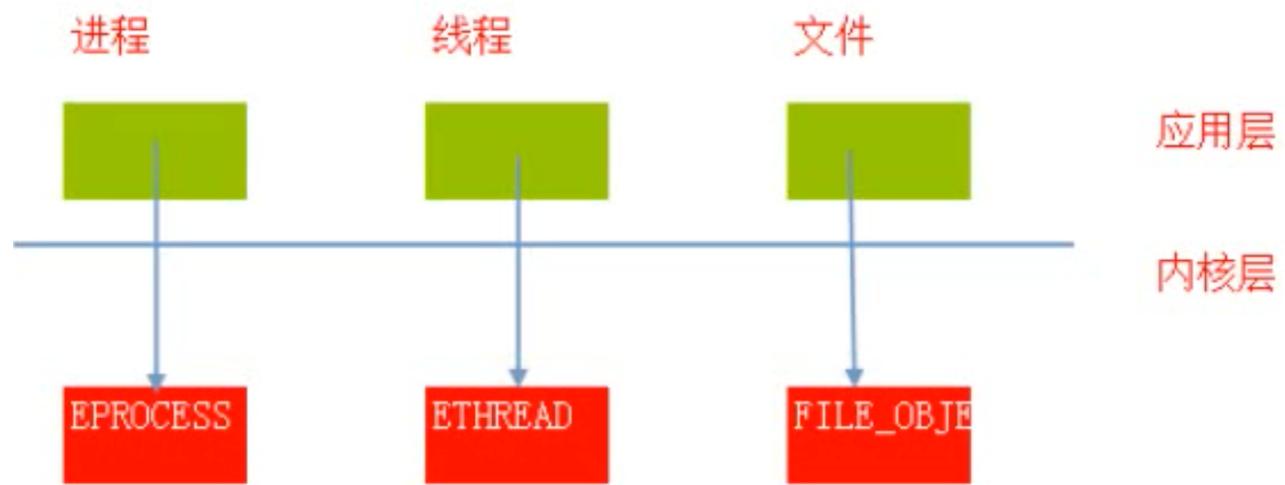
7 句柄表

在上一章节中，我们了解到了CreateProcess()函数创建进程会有一个结构体**LPPROCESS_INFORMATION** **lpProcessInformation**，这个结构体会有进程和线程的ID、句柄信息，那么什么是ID？什么是句柄？

7.1 内核对象

7.1.1 什么是内核对象

首先我们来了解一下内核对象，以后会经常与内核对象打交道，例如进程、线程、文件、互斥体、事件等等在内核都有一个对应的结构体，这些结构体都由内核负责管理，所以我们可以称之为内核对象（当我们创建一个进程，在内核层（高2G）就会创建一个结构体**EPROCESS...**）。



记不住没关系，我们可以在[MSDN Library](#)中搜索**CloseHandle**这个函数，它是用来关闭句柄的，暂时先不用管其原理，我们只要知道它所支持关闭就都是内核对象：

Platform SDK: Helper Libraries

CloseHandle

The [CloseHandle](#) function closes an open object handle.

```
BOOL CloseHandle(
    HANDLE hObject // handle to object
);
```

Parameters

hObject
[in/out] Handle to an open object.

Return Values

If the function succeeds, the return value is nonzero.
 If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).
Windows NT/2000/XP: Closing an invalid handle raises an exception when the application is running under a debugger. This includes closing a handle twice, and using [CloseHandle](#) on a handle returned by the [FindFirstFile](#) function.

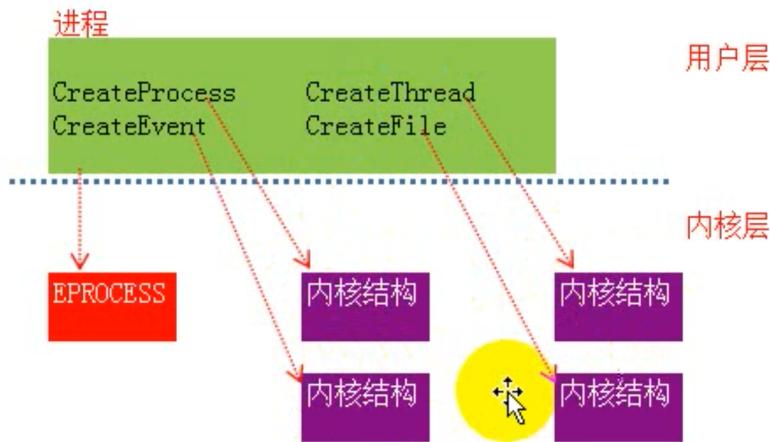
Remarks

The [CloseHandle](#) function closes handles to the following objects:

- Access token
- Communications device
- Console input
- Console screen buffer
- Event
- File
- File mapping
- Job
- Mailslot
- Mutex
- Named pipe
- Process
- Semaphore
- Socket
- Thread

7.1.2 管理内核对象

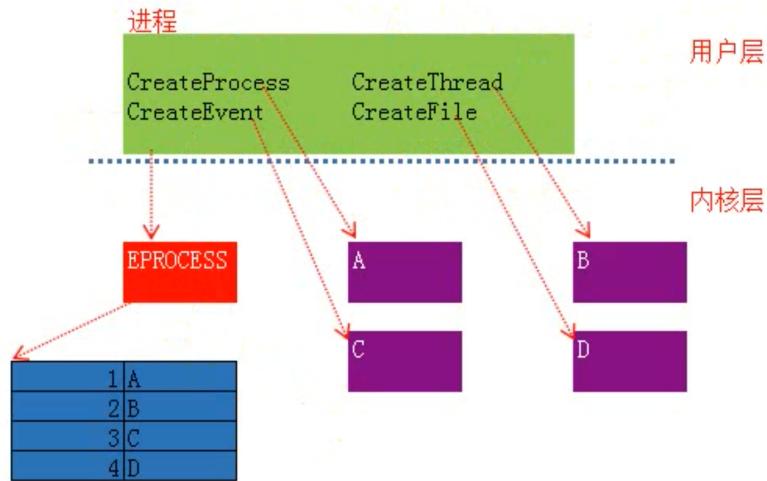
当我们使用如下图所示的函数创建时，会在内核层创建一个结构体，而我们该如何管理这些结构体呢？或者说如何使用这些结构体呢？其实很好解决，我们可以通过内核结构体地址来管理，但是这样做存在问题：**应用层很有可能操作不当导致修改啦内核结构体的地址**，我们写应用层代码都知道访问到一个不存在的内存地址就会报错，而如果访问到一个内核地址是错误的，微软系统下则直接会蓝屏。



微软为了避免这种情况的发生，所以其不会讲内核结构体的地址暴露给应用层，也就是说没法通过这种方式来直接管理。

7.2 进程句柄表

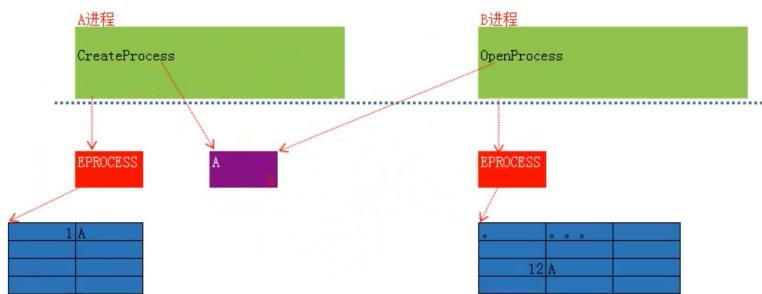
没法直接管理内核对象，这时候句柄表就诞生了，但是需要注意的是，只有进程才会有句柄表，并且每一个进程都会有一个句柄表。



句柄本质上就是一个防火墙，将应用层、内核层隔离开来，通过句柄就可以控制进程内核结构体，我们得到所谓句柄的值实际上就是句柄表里的一个索引。

7.3 多进程共享一个内核对象

如下图所示，A进程通过**CreateProcess**函数创建了一个内核对象；B进程通过**OpenProcess**函数可以打开别人创建好的一个进程，也就是可以操作其的内核对象；A进程想要操作内核对象就通过其对应的句柄表的句柄（索引）来操作；B进程操作这个内核对象也是通过它自己的句柄表的句柄（索引）来操作内核对象。（需要注意的是：句柄表是一个私有的，句柄值就是进程自己句柄表的索引）



在之前的例子中我们提到了**CloseHandle**这个函数是用来关闭进程、线程的，其实它的本质就是释放句柄，但是并不代表执行了这个函数，创建的内核对象就会彻底消失；如上图中所示内核对象存在一个计数器，目前是2，它的值是根据调用A的次数来决定的，如果我们只是在A进程中执行了**CloseHandle**函数，内核对象并不会消失，因为进程B还在使用，而只有进程B也执行了**CloseHandle**函数，这个内核对象的计数器为0，就会关闭消失了。

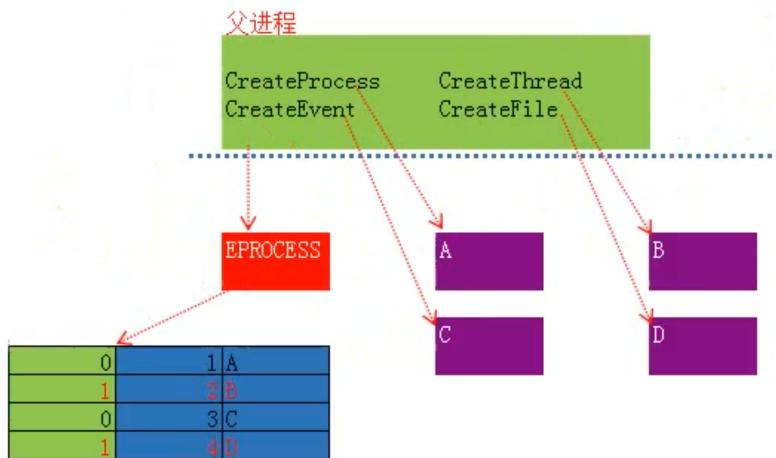
最后：注意，以上所述特性适合于除了线程以外的所有内核对象，创建进程，同时也会创建线程，如果你想把线程关闭，首先需要**CloseHandle**函数要让其计数器为0，其次需要有人将其关闭，所以假设我们创建了一个IE

进程打开了一个网站，如果我们只是在代码中使用了**CloseHandle**函数，这样IE浏览器并不会关闭，需要我们手动点击窗口的关闭按钮才行（只有线程关闭了，进程才会关闭）。

7.4 句柄是否"可以"被继承

除了我们上述的方式可以进行共享内核对象以外，Windows还设计了一种方式来提供我们共享内核对象，我们先来了解一下句柄是否"可以"被继承。

如下图所示（句柄表是有三列的，分别是句柄值、内核结构体地址、句柄是否可以被继承），比如说我们在A进程（父进程）创建了4个内核对象：



这四个函数都有一个参数**LPSECURITY_ATTRIBUTES lpThreadAttributes**，通过这个参数我们可以判断函数是否创建的是内核对象。

CreateThread

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.
To create a thread that runs in the virtual address space of another process, use the [CreateRemoteThread](#) function.

```
HANDLE CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
    _In_opt_ SIZE_T dwStackSize, // initial stack size
    _In_opt_ LPTHREAD_START_ROUTINE lpStartAddress, // thread function
    _In_opt_ LPVOID lpParameter, // thread argument
    _In_opt_ DWORD dwCreationFlags, // creation option
    _Out_opt_ LPDWORD lpThreadId // thread identifier
);
```

我们可以跟进看一下这个参数，它就是一个结构体：

```
SECURITY_ATTRIBUTES
The SECURITY_ATTRIBUTES structure contains the security descriptor for an object and specifies whether the handle retrieved by specifying this structure is inheritable.

typedef struct _SECURITY_ATTRIBUTES {
    DWORD   nLength;
    LPVOID lpSecurityDescriptor;
    BOOL    bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

结构体成员分别是：**1.结构体长度；2.安全描述符；3.句柄是否被继承。**

第一个成员我们见怪不怪了，在Windows设计下都会有这样一个成员；第二个安全描述符，这个对我们来说实际上没有任何意义，一般留空就行，默认它会遵循父进程的来，其主要作用就是**描述谁创建了该对象，谁有访问、使用该对象的权限。**

第三个成员是我们重点需要关注的，因为其决定了句柄是否可以被继承，如下图所示，我们让CreateProcess函数创建的进程、线程句柄可以被继承：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,           // name of executable module
    LPCTSTR lpCommandLine,              // command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
    BOOL bInheritHandles,                // handle inheritance option
    DWORD dwCreationFlags,              // creation flags
    LPVOID lpEnvironment,               // new environment block
    LPCTSTR lpCurrentDirectory,         // current directory name
    LPSTARTUPINFO lpStartupInfo,        // startup information
    LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

CreateProcess有两个
SECURITY_ATTRIBUTE
S结构体，一个是进
程的，一个是线程的。

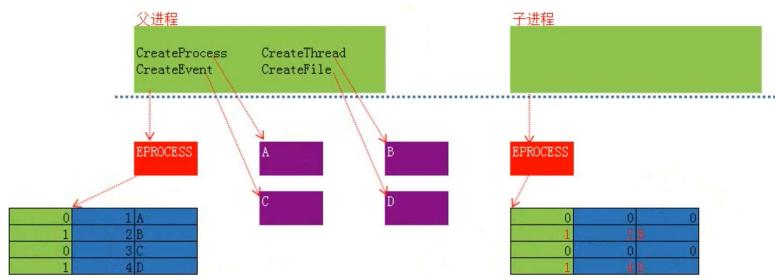
自定义结构体，成员
bInheritHandle赋值为
TRUE表示进程句柄、线
程句柄都可以被继承。

```
CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
```

```
CreateProcess(childProcessName, childProcessCommandLine, &sa, &sa, FALSE, 0, NULL, NULL, &si, &pi);
```

7.5 句柄是否"允许"被继承

我们可以让句柄被继承，但也仅仅是可以，要真正完成继承，或者说我们允许子进程继承父进程的句柄，这时候就需要另外一个参数了。



我们还是以CreateProcess函数举例，其有一个参数**BOOL bInheritHandles**，这个参数决定了是否允许创建的子进程继承句柄：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,           // name of executable module
    LPCTSTR lpCommandLine,              // command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
    BOOL bInheritHandles,                // handle inheritance option
    DWORD dwCreationFlags,              // creation flags
    LPVOID lpEnvironment,               // new environment block
    LPCTSTR lpCurrentDirectory,         // current directory name
    LPSTARTUPINFO lpStartupInfo,        // startup information
    LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

只有这个参数设置为TRUE时，我们创建的子进程才允许继承父进程的句柄。

8 进程相关API

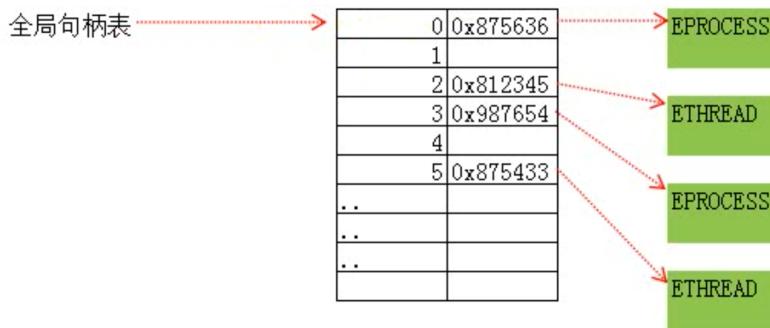
8.1 ID与句柄

如果我们成功创建了一个进程，CreateProcess函数会给我们返回一个结构体，包含四个数据：进程编号（ID）、进程句柄、线程编号（ID）、线程句柄。

进程ID其实我们早就见过了，通常我们称之为PID，在任务管理器的进程栏下就可以很清楚的看见：



每个进程都有一份私有的句柄表，而操作系统也有一份句柄表，我们称之为全局句柄表，这张表里包含了所有正在运行的进程、线程：



PID我们可以理解为是全局句柄表中的一个索引，那么PID和句柄的区别就很容易看出来了，**PID是全局的，在任何进程中都有意义，都可以使用，而句柄则是局部的、私有的；PID是唯一的，绝对不可能出现重复的存在，但是当进程消失，那么这个PID就有可能会分给另外一个进程。（PID不是句柄，但是可以通过PID获得全局句柄表中其对应的句柄）**

8.2 TerminateProcess函数

我们可以来论证一下如上所述的概念，首先A进程打开IE浏览器，然后获取进程ID和句柄：

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:\\Program Files\\Internet Explorer\\IEXPLORE.exe"); // process
    TCHAR childProcessCommandLine[] = TEXT(" http://www.baidu.com/"); // command line

    // LPSTARTUPINFO lpStartupInfo
    STARTUPINFO si;
    // LPPROCESS_INFORMATION lpProcessInformation
    PROCESS_INFORMATION pi;

    // 用0来填充si pi
    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    // LPSTARTUPINFO lpStartupInfo 是一个结构体，其他成员都可以不写，但是cb这个成员是用来表示结构体本身大小的，所以必须要写入值
    si.cb = sizeof(si);

    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        printf("CreateProcess Succesfully! \nPID:%x \n句柄:%x \n", pi.dwProcessId, pi.hProcess);
    } else {
        printf("CreateProcess Error: %d \n", GetLastError());
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    system("pause");

    return 0;
}

```

其次B进程使用TerminateProcess函数来终止A进程，首先使用句柄信息终止：

1	// TerminateProcess函数
2	BOOL TerminateProcess(
3	HANDLE hProcess, // handle to the process 句柄
4	UINT uExitCode // exit code for the process 退出代码
5);

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    HANDLE hProcess;
    hProcess = (HANDLE)0x34;

    if(!TerminateProcess(hProcess, 1)) {
        printf("终止进程失败: %d \n", GetLastError());
    }

    system("pause");

    return 0;
}

```



TerminateProcess函数是用来终止进程的，具体的可以参考MSDN Library，在这里我们很清楚的可以看见终止进程失败了，这个错误编号的意思就是句柄无效，那么就论证了句柄是私有的，其他进程无法根据这个句柄来终止进程，但是我们想要真正的关闭这个进程，那就需要借助PID来获取句柄了，具体细节如下。

8.3 OpenProcess函数

了解了TerminateProcess函数后，我们想要真正的去关闭一个进程，需要借助OpenProcess函数，这个函数是用来打开进程对象的：

<pre> 1 HANDLE OpenProcess(2 DWORD dwDesiredAccess, // access flag 你希望的访问权限 3 BOOL bInheritHandle, // handle inheritance option 是否可以被继承 4 DWORD dwProcessId // process identifier 进程ID 5); </pre>
--

OpenProcess

The **OpenProcess** function opens an existing process object.

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess, // access flag
    BOOL bInheritHandle, // handle inheritance option
    DWORD dwProcessId // process identifier
);
```

Parameters

dwDesiredAccess
 [in] Specifies the access to the process object. For operating systems that support security checking, this access is checked against any security descriptor for the target process. This parameter can be STANDARD_RIGHTS_REQUIRED or one or more of the following values.

Value	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the CreateRemoteThread function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the DuplicateHandle function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the GetExitCodeProcess and GetPriorityClass functions to read information from the process object.
PROCESS_SET_QUOTA	Enables using the process handle in the AssignProcessToJobObject and SetProcessWorkingSetSize functions to set memory limits.
PROCESS_SET_INFORMATION	Enables using the process handle in the SetPriorityClass function to set the priority class of the process.
PROCESS_TERMINATE	Enables using the process handle in the TerminateProcess function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the VirtualProtectEx and WriteProcessMemory functions to modify the virtual memory of the process.
PROCESS_VM_READ	Enables using the process handle in the ReadProcessMemory function to read from the virtual memory of the process.
PROCESS_VM_WRITE	Enables using the process handle in the WriteProcessMemory function to write to the virtual memory of the process.
SYNCHRONIZE	Windows NT/2000/XP: Enables using the process handle in any of the wait functions to wait for the process to terminate.

如下代码所示，我通过PID打开进程（OpenProcess函数），拥有所有权，不继承句柄表，当OpenProcess函数执行完成之后，我就获得一个句柄，通过这个句柄我就可以终止进程：

```

1 HANDLE hProcess;
2 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 0x524);
3
4 if(!TerminateProcess(hProcess, 0)) {
5     printf("终止进程失败 :%d \n", GetLastError());
6 }
```

8.4 以挂起的形式创建进程

CreateProcess函数的所有参数都需要了解一下，现在我们来看一下第六个参数**DWORD dwCreationFlags**：

```

1  BOOL CreateProcess(
2      LPCTSTR lpApplicationName,           // name of executable module
3      LPTSTR lpCommandLine,              // command line string
4      LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
5      LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
6      BOOL bInheritHandles,             // handle inheritance option
7      DWORD dwCreationFlags,            // creation flags <--这个参数
8      LPVOID lpEnvironment,             // new environment block
9      LPCTSTR lpCurrentDirectory,       // current directory name
10     LPSTARTUPINFO lpStartupInfo,        // startup information
11     LPPROCESS_INFORMATION lpProcessInformation // process information
12 );

```

当我们创建一个控制台进程时，会发现子进程和父进程都在同一个命令行控制台中：

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:/WINDOWS/system32/cmd.exe");
    TCHAR childProcessCommandLine[] = TEXT(" /c ping 127.0.0.1 -t");

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    si.cb = sizeof(si);

    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
        printf("CreateProcess Successfully! \n");
    } else {
        printf("CreateProcess Error: %d \n", GetLastError());
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    system("pause");
    return 0;
}

```

而如果我们想要区分的话就需要借助**dwCreationFlags**这个参数，将其修改为**CREATE_NEW_CONSOLE**即可：

DWORD dwCreationFlags // creation flags	
dwCreationFlags [in] Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted.	
Value	Meaning
CREATE_BREAKAWAY_FROM_JOB	Windows 2000/XP: The child processes of a process associated with a job are not associated with the job. If the calling process is not associated with a job, this flag has no effect. If the calling process is associated with a job, the job must set the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit or CreateProcess will fail.
CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, CreateProcess gives the new process the current default error mode. An application sets the current default error mode by calling SetErrorMode . This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for CreateProcess is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_FORCEDOS	Windows NT/2000/XP: This flag is valid only when starting a 16-bit bound application. If set, the system will force the application to run as an MS-DOS-based application rather than as an OS/2-based application.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag.

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:/WINDOWS/system32/cmd.exe");
    TCHAR childProcessCommandLine[] = TEXT(" /c ping 127.0.0.1 -t");

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

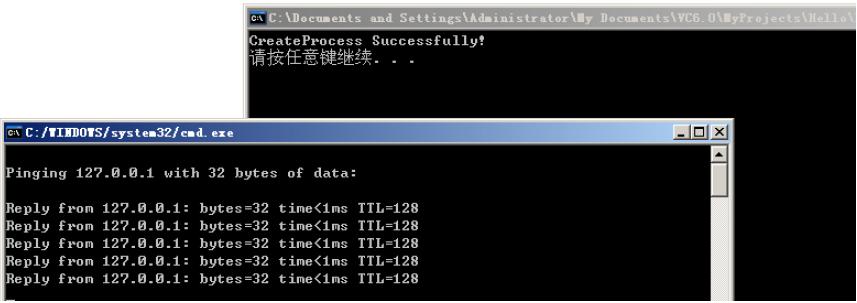
    si.cb = sizeof(si);

    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi)) {
        printf("CreateProcess Successfully! \n");
    } else {
        printf("CreateProcess Error: %d \n", GetLastError());
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

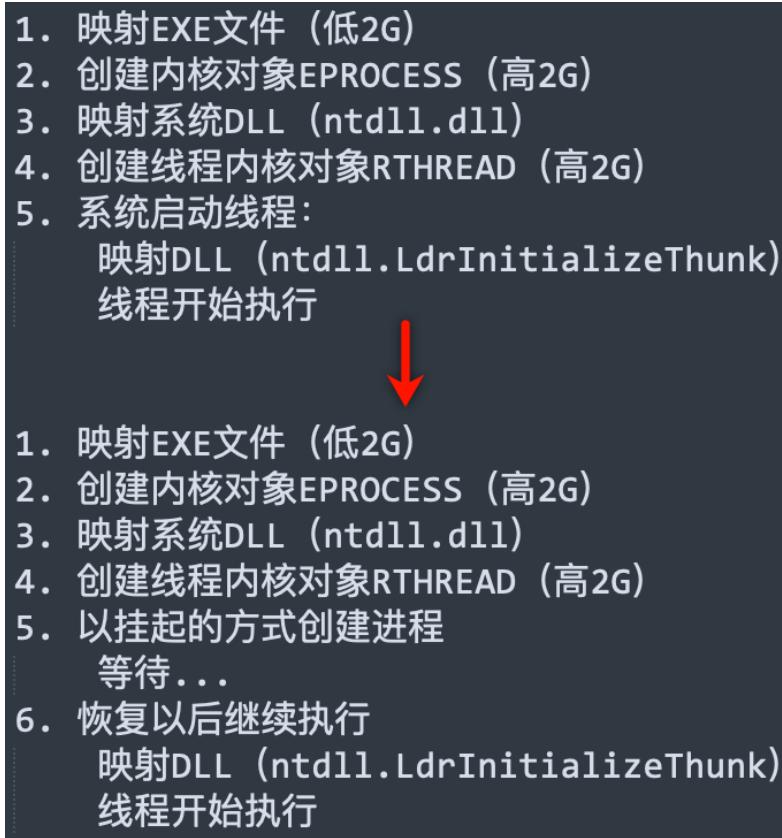
    system("pause");
    return 0;
}

```



但是这个并不是我们最重要的，或者说不是其真正有意义的参数，有意义的是参数值为**CREATE_SUSPENDED**，也就是以挂起的形式创建进程。

而如果是以挂起的方式创建进程，那么进程的创建过程就会发生变化：



那就说明了一点，挂起本质上挂起的是线程，进程还是会创建的，所以，最终如果想恢复的话也是恢复线程：

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:/WINDOWS/system32/cmd.exe");
    TCHAR childProcessCommandLine[] = TEXT(" /c ping 127.0.0.1 -t");

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

    si.cb = sizeof(si);

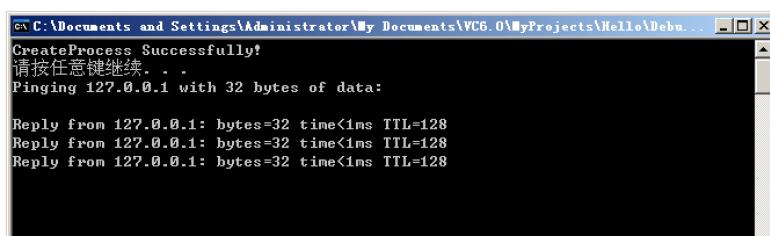
    if(CreateProcess(childProcessName, childProcessCommandLine, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi)) {
        printf("CreateProcess Successfully!\n");
    } else {
        printf("CreateProcess Error: %d\n", GetLastError());
    }

    // 恢复
    ResumeThread(pi.hThread);

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    system("pause");
    return 0;
}

```



8.5 模块目录与工作目录

通过**GetModuleFileName**和**GetCurrentDirectory**函数可以分别获得当前模块目录和当前工作目录：

```

1  char strModule[256];
2  GetModuleFileName(NULL, strModule, 256); // 得到当前模块目录，当前exe所在的路径，包含exe文件名
3
4  char strWork[1000];
5  GetCurrentDirectory(1000, strWork); // 获取当前工作目录
6
7  printf("模块目录：%s \n工作目录：%s \n", strModule, strWork);

```

需要注意的是工作目录是可以修改的，我们可以通过**CreateProcess**函数来创建一个进程，并且修改其工作目录，这是CreateProcess函数的第八个参数**LPCTSTR lpCurrentDirectory**。

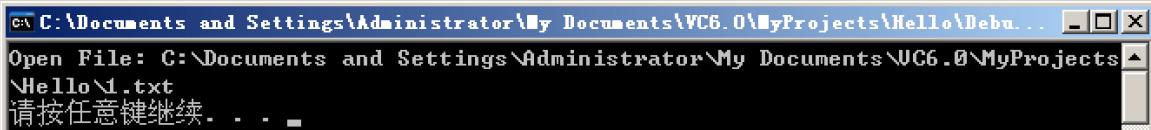
假设我们有这样一个需求：打开当前工作目录下的1.txt文件：

```

#include <stdlib.h>

int main(int argc, char* argv[])
{
    char strWork[1000];
    GetCurrentDirectory(1000, strWork);
    char fileName[] = "\\1.txt";
    strcat(strWork, fileName);
    FILE* f = fopen(strWork, "r");
    printf("Open File: %s \n", strWork);
    system("pause");
    return 0;
}

```



而这时候我们可以通过CreateProcess函数修改工作路径，让其读取我们指定工作目录的文件：

```

#include <windows.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    TCHAR childProcessName[] = TEXT("C:\\Documents and Settings\\Administrator\\桌面\\Hello.exe");
    TCHAR workPath[] = TEXT("C:\\\\");

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));

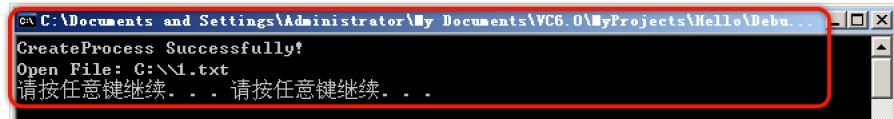
    si.cb = sizeof(si);

    if(CreateProcess(childProcessName, NULL, NULL, NULL, FALSE, 0, NULL, workPath, &si, &pi)) {
        printf("CreateProcess Successfully!\n");
    } else {
        printf("CreateProcess Error: %d\n", GetLastError());
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    system("pause");
    return 0;
}

```



8.6 其他进程相关API

获取当前进程ID（PID）：GetCurrentProcessId

获取当前进程句柄：GetCurrentProcess

获取命令行：GetCommandLine

获取启动信息：GetStartupInfo

遍历进程ID：EnumProcesses

快照：CreateToolhelp32Snapshot

9 创建线程

9.1 什么是线程

1. 线程是附属在进程上的执行实体，是代码的执行流程；
2. 一个进程可以包含多个线程（一个进程至少要包含一个线程，进程是空间上的概念，线程是时间上的概念）。

通过Windows任务管理器我们也可以很清晰的看见每个进程当前的线程数量：

映像名称	PID	用户名	CPU	内存使用	线程数
Hello.exe	3996	Administrator	00	1,012 K	1
Hello.exe	3396	Administrator	00	1,084 K	1
MSDEV.EXE	2652	Administrator	00	6,044 K	10
taskmgr.exe	2580	Administrator	02	5,036 K	3
hh.exe	2304	Administrator	00	15,540 K	8
vmtoolsd.exe	2040	Administrator	00	15,096 K	5
cmd.exe	2004	Administrator	00	2,388 K	1
vmtoolsd.exe	1912	SYSTEM	00	14,352 K	8
VGAuthService...	1748	SYSTEM	00	9,044 K	2
ctfmon.exe	1684	Administrator	00	3,848 K	1
svchost.exe	1688	LOCAL SERVICE	00	3,312 K	8
cmd.exe	1544	Administrator	00	2,396 K	1
cmd.exe	1524	Administrator	00	2,512 K	1
spoolsv.exe	1484	SYSTEM	00	6,656 K	11
svchost.exe	1300	LOCAL SERVICE	00	4,600 K	13
svchost.exe	1228	NETWORK SERVICE	00	3,308 K	4
svchost.exe	1120	SYSTEM	00	19,228 K	60
rundll32.exe	1036	Administrator	00	3,488 K	4
winsmolt.exe	1024	Administrator	00	5,288 K	3

有几个线程就表示着有几个代码在执行，但是它们并不一定是同时执行，例如单核的CPU情况下是不存在多线程的，线程的执行是有时间顺序的，但是CPU切换的非常快，所以给我们的感觉和多核CPU没什么区别。

9.2 创建线程

创建线程使用CreateThread函数，其语法格式如下：

```

1 HANDLE CreateThread( // 返回值是线程句柄
2     LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD 安全属性，包含安全描述符
3     SIZE_T dwStackSize, // initial stack size 初始堆栈
4     LPTHREAD_START_ROUTINE lpStartAddress, // thread function 线程执行的函数代码
5     LPVOID lpParameter, // thread argument 线程需要的参数
6     DWORD dwCreationFlags, // creation option 标识，也可以以挂起形式创建线程
7     LPDWORD lpThreadId // thread identifier 返回当前线程ID
8 );

```

线程执行函数的语法要求如下：

ThreadProc

The **ThreadProc** function is an application-defined function that serves as the starting address for a thread. Specify this address when calling the [CreateThread](#) or [CreateRemoteThread](#) function. The **LPTHREAD_START_ROUTINE** type defines a pointer to this callback function. **ThreadProc** is a placeholder for the application-defined function name.

```

DWORD WINAPI ThreadProc(
    LPVOID lpParameter // thread data
);

```

我们尝试创建一个线程执行for循环，如下图：

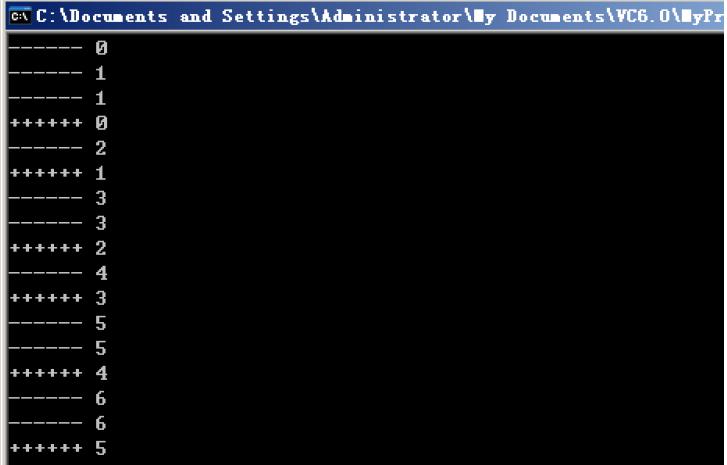
```
#include <windows.h>

// 线程执行的函数有语法要求，参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    // 要执行的代码
    for(int i = 0; i < 100; i++) {
        Sleep(500);
        printf("++++++ %d \n", i);
    }

    return 0;
}

int main(int argc, char* argv[])
{
    // 创建线程
    CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

    // 要执行的代码
    for(int i = 0; i < 100; i++) {
        Sleep(500);
        printf("----- %d \n", i);
    }
    return 0;
}
```



```
0
1
1
0
2
2
1
3
3
2
4
4
3
5
5
4
6
6
5
6
6
5
```

```

1 #include <windows.h>
2
3 // 线程执行的函数有语法要求，参考MSDN Library
4 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
5     // 要执行的代码
6     for(int i = 0; i < 100; i++) {
7         Sleep(500);
8         printf("++++++ %d \n", i);
9     }
10
11     return 0;
12 }
13
14 int main(int argc, char* argv[])
15 {
16     // 创建线程
17     CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
18
19     // 要执行的代码
20     for(int i = 0; i < 100; i++) {
21         Sleep(500);
22         printf("----- %d \n", i);
23     }
24     return 0;
25 }
26 }
```

线程间不会相互配合，而是各自执行自己的，如果想要配合就需要了解线程通信，这个后面会学习到。

9.3 向线程函数传递参数

向线程传递参数，如下图所示，我们想要自定义线程执行for循环的次数，将n传递进去，这时候需要注意参数传递到线程参数时在堆栈中存在，并且传递的时候需要强制转换一下：

```
#include <windows.h>

// 线程执行的函数有语法要求, 参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    int* p = (int*) lpParameter;

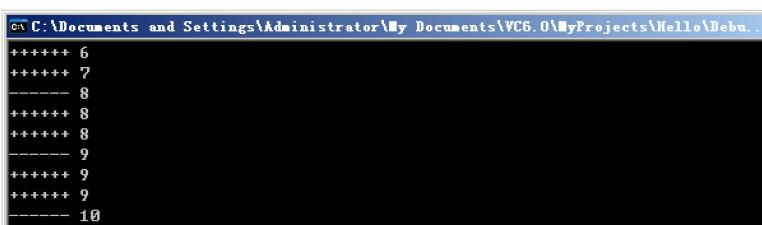
    // 要执行的代码
    for(int i = 0; i < *p; i++) {
        Sleep(500);
        printf("++++++ %d \n", i);
    }

    return 0;
}

int main(int argc, char* argv[])
{
    int n = 10;
    // 创建线程
    CreateThread(NULL, NULL, ThreadProc, (LPVOID)&n, 0, NULL);

    // 要执行的代码
    for(int i = 0; i < 100; i++) {
        Sleep(500);
        printf("----- %d \n", i);
    }

    return 0;
}
```



为了保证参数的生命周期，我们也可以将参数放在全局变量区：

```
#include <windows.h>

// 线程执行的函数有语法要求, 参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    int* p = (int*) lpParameter;

    // 要执行的代码
    for(int i = 0; i < *p; i++) {
        Sleep(500);
        printf("++++++ %d \n", i);
    }

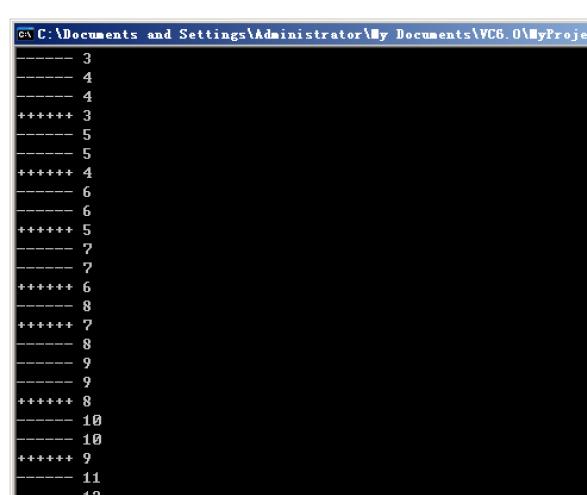
    return 0;
}

int n = 10;

int main(int argc, char* argv[])
{
    CreateThread(NULL, NULL, ThreadProc, (LPVOID)&n, 0, NULL);

    // 要执行的代码
    for(int i = 0; i < 100; i++) {
        Sleep(500);
        printf("----- %d \n", i);
    }

    return 0;
}
```



10 线程控制

10.1 让线程停下来

10.1.1 Sleep函数

Sleep函数是让当前执行到本函数时延迟指定的毫秒之后再向下走，例如：

```

1  for(int i = 0; i < 100; i++) {
2      Sleep(500);
3      printf("----- %d \n", i);
4 }
```

10.1.2 SuspendThread函数

SuspendThread函数用于暂停（挂起）某个线程，当暂停后该线程不会占用CPU，其语法格式很简单，只需要传入一个线程句柄即可：

```

1  DWORD SuspendThread(
2      HANDLE hThread // handle to thread
3 );
```

10.1.3 ResumeThread函数

ResumeThread函数用于恢复被暂停（挂起）的线程，其语法格式也很简单，只需要传入一个线程句柄即可：

```

1  DWORD ResumeThread(
2      HANDLE hThread // handle to thread
3 );
```

需要注意的是，挂起几次就要恢复几次。

```

1  SuspendThread(hThread);
2  SuspendThread(hThread);
3
4  ResumeThread(hThread);
5  ResumeThread(hThread);
```

10.2 等待线程结束

10.2.1 WaitForSingleObject函数

WaitForSingleObject函数用于等待一个内核对象状态发生变更，那也就是执行结束之后，才会继续向下执行，其语法格式如下：

```

1  DWORD WaitForSingleObject(
2      HANDLE hHandle,          // handle to object 句柄
3      DWORD dwMilliseconds    // time-out interval 等待超时时间 (毫秒)
4 );

```

Parameters

hHandle
 [in] Handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section.
 If this handle is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000/XP: The handle must have SYNCHRONIZE access. For more information, see [Standard Access Rights](#).

dwMilliseconds
 [in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. **If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.**

如果你想一直等待的话，可以将第二参数的值设置为**INFINITE**。

```

1  HANDLE hThread;
2  hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
3  WaitForSingleObject(hThread, INFINITE);
4  printf("OK...");

```

10.2.2 WaitForMultipleObjects函数

WaitForMultipleObjects函数与WaitForSingleObject函数作用是一样的，只不过它可以等待多个内核对象的状态发生变更，其语法格式如下：

```

1  DWORD WaitForMultipleObjects(
2      DWORD nCount,           // number of handles in array 内核对象的数量
3      CONST HANDLE *lpHandles, // object-handle array 内核对象的句柄数组
4      BOOL bWaitAll,          // wait option 等待模式
5      DWORD dwMilliseconds    // time-out interval 等待超时时间 (毫秒)
6 );

```

Parameters

nCount
 [in] Specifies the number of object handles in the array pointed to by *lpHandles*. The maximum number of object handles is MAXIMUM_WAIT_OBJECTS.

lpHandles
 [in] Pointer to an array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles to objects of different types. It may not contain the multiple copies of the same handle.
 If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000/XP: The handles must have SYNCHRONIZE access. For more information, see [Standard Access Rights](#).

Windows 95/98/Me: No handle may be a duplicate of another handle created using [DuplicateHandle](#).

bWaitAll
 [in] Specifies the wait type. If TRUE, the function returns when the state of all objects in the *lpHandles* array is signaled. If FALSE, the function returns when the state of any one of the objects is set to signaled. In the latter case, the return value indicates the object whose state caused the function to return.

dwMilliseconds
 [in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the conditions specified by the *bWaitAll* parameter are not met. If *dwMilliseconds* is zero, the function tests the states of the specified objects and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

等待模式的值是布尔类型，一个是TRUE，一个是FALSE，TRUE就是等待所有对象的所有状态发生变更，FALSE则是等待任意一个对象的状态发生变更。

```

1 HANDLE hThread[2];
2 hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
3 hThread[1] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
4 WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
```

10.2.3 GetExitCodeThread函数

线程函数会有一个返回值（DWORD），这个返回值可以根据你的需求进行返回，而我们需要如何获取这个返回结果呢？这时候就可以使用**GetExitCodeThread**函数，其语法格式如下：

```

1 BOOL GetExitCodeThread(
2     HANDLE hThread,           // handle to the thread
3     LPDWORD lpExitCode      // termination status
4 );
```

Parameters

hThread
 [in] Handle to the thread.

Windows NT/2000/XP: The handle must have THREAD_QUERY_INFORMATION access. For more information, see [Thread Security and Access Rights](#).

lpExitCode
 [out] Pointer to a variable to receive the thread termination status.

根据MSDN Library我们可以知道该函数的参数分别是线程句柄，而另一个则是out类型参数，这种类型则可以理解为GetExitCodeThread函数的返回结果。

```

1 HANDLE hThread;
2 hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
3
4 WaitForSingleObject(hThread, INFINITE);
5
6 DWORD exitCode;
7 GetExitCodeThread(hThread, &exitCode);
8
9 printf("Exit Code: %d \n", exitCode);
```

```

#include <windows.h>

// 线程执行的函数有语法要求, 参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    // 要执行的代码
    for(int i = 0; i < 2; i++) {
        Sleep(500);
        printf("+++++ %d \n", i);
    }

    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hThread;
    hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

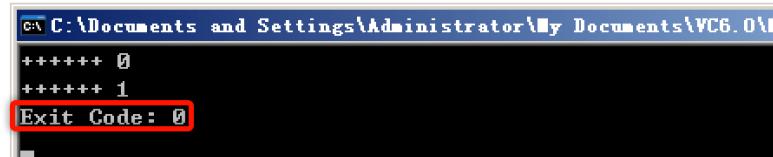
    WaitForSingleObject(hThread, INFINITE);

    DWORD exitCode;
    GetExitCodeThread(hThread, &exitCode);

    printf("Exit Code: %d \n", exitCode);

    getchar();
    return 0;
}

```



需要注意的是这个函数应该搭配着如上所学的2个等待函数一起使用，不然获取到的值就不会是线程函数返回的值。

10.3 设置、获取线程上下文

线程上下文是指某一时间点CPU寄存器和程序计数器的内容，如果想要设置、获取线程上下文就需要先将线程挂起。

10.3.1 GetThreadContext函数

GetThreadContext函数用于获取线程上下文，其语法格式如下：

1	BOOL GetThreadContext(
2	HANDLE hThread, // handle to thread with context 句柄
3	LPCONTEXT lpContext // context structure
4);

Parameters***hThread***

[in] Handle to the thread whose context is to be set.

Windows NT/2000/XP: The handle must have the THREAD_SET_CONTEXT access right to the thread. For more information, see [Thread Security and Access Rights](#).

lpContext

[in] Pointer to the **CONTEXT** structure that contains the context to be set in the specified thread. The value of the **ContextFlags** member of this structure specifies which portions of a thread's context to set. Some values in the **CONTEXT** structure that cannot be specified are silently set to the correct value. This includes bits in the CPU status register that specify the privileged processor mode, global enabling bits in the debugging register, and other states that must be controlled by the operating system.

第一个参数就是线程句柄，这个很好理解，重点是第二个参数，其是一个CONTEXT结构体，该结构体包含指定线程的上下文，其ContextFlags成员的值指定了要设置线程上下文的哪些部分。

当我们将CONTEXT结构体的ContextFlags成员的值设置为CONTEXT_INTEGER时则可以获取edi、esi、ebx、edx、ecx、eax这些寄存器的值：

```

// 
// This section is specified/returned if the
// ContextFlags word contains the flag CONTEXT_INTEGER.
//

DWORD Edi;
DWORD Esi;
DWORD Ebx;
DWORD Edx;
DWORD Ecx;
DWORD Eax;

```

如下代码尝试获取：

1	HANDLE hThread;
2	hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
3	SuspendThread(hThread);
4	CONTEXT c;
5	c.ContextFlags = CONTEXT_INTEGER;
6	GetThreadContext(hThread, &c);
7	
8	
9	
10	printf("%x %x \n", c.Eax, c.Ecx);

```
#include <windows.h>

// 线程执行的函数有语法要求, 参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    // 要执行的代码
    for(int i = 0; i < 2; i++) {
        Sleep(500);
        printf("++++++ %d \n", i);
    }

    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hThread;
    hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
    SuspendThread(hThread);
    CONTEXT c;
    c.ContextFlags = CONTEXT_INTEGER;
    GetThreadContext(hThread, &c);

    printf("%x %x \n", c.Eax, c.Ecx);
    getchar();
    return 0;
}
```

C:\Documents and Settings\Administrator\My Documents\VC6.0\MyProject

40100a 40

10.3.2 SetThreadContext函数

GetThreadContext函数是个设置修改线程上下文，其语法格式如下：

1	BOOL SetThreadContext(
2	HANDLE hThread, // handle to thread
3	CONST CONTEXT *lpContext // context structure
4);

我们可以尝试修改Eax，然后再获取：

```
1 HANDLE hThread;
2 hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
3
4 SuspendThread(hThread);
5
6 CONTEXT c;
7 c.ContextFlags = CONTEXT_INTEGER;
8 c.Eax = 0x123;
9 SetThreadContext(hThread, &c);
10
11 CONTEXT c1;
12 c1.ContextFlags = CONTEXT_INTEGER;
13 GetThreadContext(hThread, &c1);
14
15 printf("%x \n", c1.Eax);
```

```

#include <windows.h>

// 线程执行的函数有语法要求, 参考MSDN Library
DWORD WINAPI ThreadProc(LPVOID lpParameter) {

    // 要执行的代码
    for(int i = 0; i < 2; i++) {
        Sleep(500);
        printf("++++++ %d \n", i);
    }

    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hThread;
    hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

    SuspendThread(hThread);

    CONTEXT c;
    c.ContextFlags = CONTEXT_INTEGER;
    c.Eax = 0x123;
    SetThreadContext(hThread, &c);

    CONTEXT c1;
    c1.ContextFlags = CONTEXT_INTEGER;
    GetThreadContext(hThread, &c1);

    printf("%x \n", c1.Eax);
    getchar();
    return 0;
}

```



11 临界区

11.1 线程安全问题

每个线程都有自己的栈，局部变量是存储在栈中的，这就意味着每个进程都会有一份自己的“句柄变量”（栈），如果线程仅仅使用自己的“局部变量”那就不存在线程安全问题，反之，**如果多个线程共用一个全局变量呢？那么在什么情况下会有问题呢？那就是当多线程共用一个全局变量并对其进行修改时则存在安全问题**，如果仅仅是读的话没有问题。

如下所示代码，我们写了一个线程函数，该函数的作用就是使用全局变量，模拟的功能就是售卖物品，全局变量countNumber表示该物品的总是，其值是10，而如果有多个地方（线程）去卖（使用）这个物品（全局变量），则会出现差错：

```

1 #include <windows.h>
2
3 int countNumber = 10;
4
5 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
6     while (countNumber > 0) {
7         printf("Sell num: %d\n", countNumber);
8         // 售出-1
9         countNumber--;
10        printf("Count: %d\n", countNumber);
11    }
12    return 0;
13 }
14
15 int main(int argc, char* argv[])
16 {
17     HANDLE hThread;
18     hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
19
20     HANDLE hThread1;
21     hThread1 = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
22
23     CloseHandle(hThread);
24
25     getchar();
26     return 0;
27 }
```

如图，我们运行了代码，发现会出现重复售卖，并且到最后总数竟变成了-1：

```
#include <windows.h>

int countNumber = 10;

DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    while (countNumber > 0) {
        printf("Sell num: %d\n", countNumber);
        // 售出-1
        countNumber--;
        printf("Count: %d\n", countNumber);
    }
    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE hThread;
    hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

    HANDLE hThread1;
    hThread1 = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

    CloseHandle(hThread);

    getchar();
    return 0;
}
```

```
C:\Documents and Settings\Administrator
Sell num: 10
Count: 9
Count: 9
Sell num: 9
Sell num: 9
Count: 8
Count: 7
Sell num: 7
Count: 6
Count: 5
Sell num: 5
Sell num: 5
Count: 4
Count: 3
Sell num: 3
Sell num: 3
Count: 2
Count: 1
Sell num: 1
Sell num: 1
Count: 0
Count: -1
```

出现这样的问题其本质原因是什么呢？因为多线程在执行的时候是同步进行的，并不是按照顺序来，所以就都会窒息，自然就会出现这种情况。

11.1.1 解决问题

想要解决线程安全问题，就需要引伸出一个概念：临界资源，临界资源表示对该资源的访问一次只能有一个线程；访问临界资源的那一段程序，我们称之为临界区。

那么我们如何实现临界区呢？第一，我们可以自己来写，但是这需要一定门槛，先不过多的去了解；第二，可以使用Windows提供的API来实现。

11.2 实现临界区

首先会有一个令牌，假设线程1获取了这个令牌，那么这时候令牌则只为线程1所有，然后线程1会执行代码去访问全局变量，最后归还令牌；如果其他线程想要去访问这个全局变量就需要获取这个令牌，但当令牌已经被取走时则无法访问。

假设你自己来实现临界区，可能在判断令牌有没有被拿走的时候就又会出现问题，所以自己实现临界区还是有一定的门槛的。

线程1：

获取令牌

... 代码
归还令牌

线程2：

获取令牌

... 代码
归还令牌

线程3：

获取令牌

... 代码

令牌
0 或者 1

全局变量X

11.3 线程锁

线程锁就是临界区的实现方式，通过线程锁我们可以完美解决如上所述的问题，其步骤如下所示：

1. 创建全局变量：CRITICAL_SECTION cs;
2. 初始化全局变量：InitializeCriticalSection(&cs);
3. 实现临界区：进入 → EnterCriticalSection(&cs); 离开 → LeaveCriticalSection(&cs);

我们就可以这样改写之前的售卖物品的代码：

在使用全局变量开始前构建并进入临界区，使用完之后离开临界区：

```

1 #include <windows.h>
2
3 CRITICAL_SECTION cs; // 创建全局变量
4 int countNumber = 10;
5
6 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
7     while (1) {
8         EnterCriticalSection(&cs); // 构建临界区，获取令牌
9         if (countNumber > 0) {
10             printf("Thread: %d\n", *((int*)lpParameter));
11             printf("Sell num: %d\n", countNumber);
12             // 售出-1
13             countNumber--;
14             printf("Count: %d\n", countNumber);
15         } else {
16             LeaveCriticalSection(&cs); // 离开临界区，归还令牌
17             break;
18         }
19         LeaveCriticalSection(&cs); // 离开临界区，归还令牌
20     }
21
22     return 0;
23 }
24
25 int main(int argc, char* argv[])
26 {
27
28     InitializeCriticalSection(&cs); // 使用之前进行初始化
29
30     int a = 1;
31     HANDLE hThread;
32     hThread = CreateThread(NULL, NULL, ThreadProc, (LPVOID)&a, 0, NULL);
33
34     int b = 2;
35     HANDLE hThread1;
36     hThread1 = CreateThread(NULL, NULL, ThreadProc, (LPVOID)&b, 0, NULL);
37
38     CloseHandle(hThread);
39
40     getchar();
41     return 0;
42 }
```

```

#include <windows.h>

CRITICAL_SECTION cs; // 创建全局变量
int countNumber = 10;

DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    while (1) {
        EnterCriticalSection(&cs); // 构建临界区，获取令牌
        if (countNumber > 0) {
            printf("Thread: %d\n", *(int*)lpParameter);
            printf("Sell num: %d\n", countNumber);
            // 售出-1
            countNumber--;
            printf("Count: %d\n", countNumber);
        } else {
            LeaveCriticalSection(&cs); // 离开临界区，归还令牌
            break;
        }
        LeaveCriticalSection(&cs); // 离开临界区，归还令牌
    }
    return 0;
}

int main(int argc, char* argv[])
{
    InitializeCriticalSection(&cs); // 使用之前进行初始化

    int a = 1;
    HANDLE hThread;
    hThread = CreateThread(NULL, NULL, ThreadProc, (LPVOID)&a, 0, NULL);

    int b = 2;
    HANDLE hThread1;
    hThread1 = CreateThread(NULL, NULL, ThreadProc, (LPVOID)&b, 0, NULL);

    CloseHandle(hThread);

    getchar();
    return 0;
}

```

```

C:\Documents and Settings\Administrator\

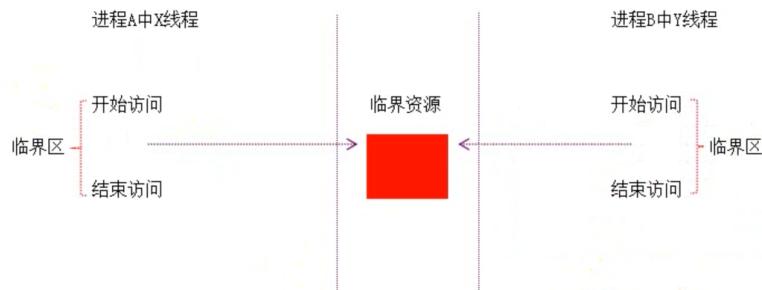
Thread: 1
Sell num: 10
Count: 9
Thread: 2
Sell num: 9
Count: 8
Thread: 1
Sell num: 8
Count: 7
Thread: 2
Sell num: 7
Count: 6
Thread: 1
Sell num: 6
Count: 5
Thread: 2
Sell num: 5
Count: 4
Thread: 1
Sell num: 4
Count: 3
Thread: 2
Sell num: 3
Count: 2
Thread: 1
Sell num: 2
Count: 1
Thread: 2
Sell num: 1
Count: 0
-

```

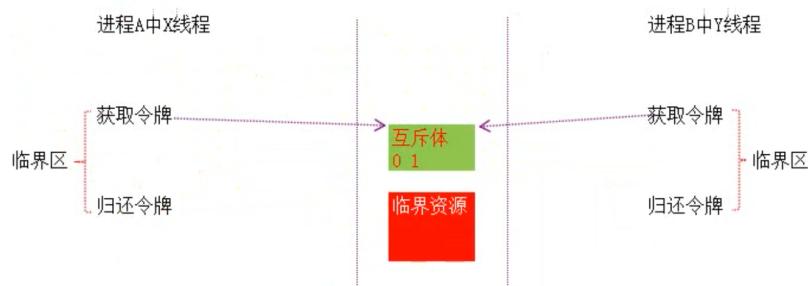
12 互斥体

12.1 内核级临界资源怎么办？

上一章中我们了解了使用线程锁来解决多个线程共用一个全局变量的线程安全问题；那么假设A进程的B线程和C进程的D线程，同时使用的是内核级的临界资源（内核对象：线程、文件、进程...）该怎么让这个访问是安全的？使用线程锁的方式明显不行，因为线程锁仅能控制同进程中的多线程。



那么这时候我们就需要一个能够放在内核中的令牌来控制，而实现这个作用的，我们称之为互斥体。



12.1.1 互斥体的使用

创建互斥体的函数为**CreateMutex**，该函数的语法格式如下：

<pre> 1 HANDLE CreateMutex(2 LPSECURITY_ATTRIBUTES lpMutexAttributes, // SD 安全属性，包含安全描述符 3 BOOL bInitialOwner, // initial owner 是否希望互斥体创建出来就有信号，或者说就可以 4 LPCTSTR lpName // object name 互斥体的名字 5); </pre>
--

我们可以模拟一下操作资源然后创建：

```
1 #include <windows.h>
2
3 int main(int argc, char* argv[])
4 {
5     // 创建互斥体
6     HANDLE cm = CreateMutex(NULL, FALSE, "XYZ");
7     // 等待互斥体状态发生变化，也就是有信号或为互斥体拥有者，获取令牌
8     WaitForSingleObject(cm, INFINITE);
9
10    // 操作资源
11    for (int i = 0; i < 5; i++) {
12        printf("Process: A Thread: B -- %d \n", i);
13        Sleep(1000);
14    }
15    // 释放令牌
16    ReleaseMutex(cm);
17    return 0;
18 }
```

我们可以运行两个进程来看一下互斥体的作用：

```

Hello - Microsoft Visual C++ - [Hello.cpp]
[文件] [编辑] [查看] [插入] [工程] [组建] [工具] [窗口] [帮助]
[Globals] [All global members] main
工作区 'Hello': 1 工程
Hello files
- Source Files
  - Hello.cpp
  - StdAfx.cpp
- Header Files
  - StdAfx.h
- Resource Files
- ReadMe.txt
- External Depend

#include "stdafx.h"
#include <windows.h>

int main(int argc, char* argv[])
{
    // 创建互斥体
    HANDLE cm = CreateMutex(NULL, FALSE, "XYZ");
    // 等待互斥体状态发生变化, 也就是有信号或为互斥体拥有者, 获得令牌
    WaitForSingleObject(cm, INFINITE);

    // 操作资源
    for (int i = 0; i < 5; i++) {
        printf("Process: A Thread: B -- %d \n", i);
        Sleep(1000);
    }

    // 释放令牌
    ReleaseMutex(cm);
    return 0;
}

Test - Microsoft Visual C++ - [Test.cpp]
[文件] [编辑] [查看] [插入] [工程] [组建] [工具] [窗口] [帮助]
[Globals] [All global members] main
工作区 'Test': 1 工程
Test files
- Source Files
  - StdAfx.cpp
  - Test.cpp
- Header Files
- Resource Files
- ReadMe.txt
- External Depend

#include "stdafx.h"
#include <windows.h>

int main(int argc, char* argv[])
{
    // 创建互斥体
    HANDLE cm = CreateMutex(NULL, FALSE, "XYZ");
    // 等待互斥体状态发生变化, 也就是有信号或为互斥体拥有者, 获得令牌
    WaitForSingleObject(cm, INFINITE);

    // 操作资源
    for (int i = 0; i < 5; i++) {
        printf("Process: A Thread: B -- %d \n", i);
        Sleep(1000);
    }

    // 释放令牌
    ReleaseMutex(cm);
    return 0;
}

```

12.2 互斥体和线程锁的区别

1. 线程锁只能用于单个进程间的线程控制
2. 互斥体可以设定等待超时, 但线程锁不能
3. 线程意外结束时, 互斥体可以避免无限等待
4. 互斥体效率没有线程锁高

12.3 课外扩展-互斥体防止程序多开

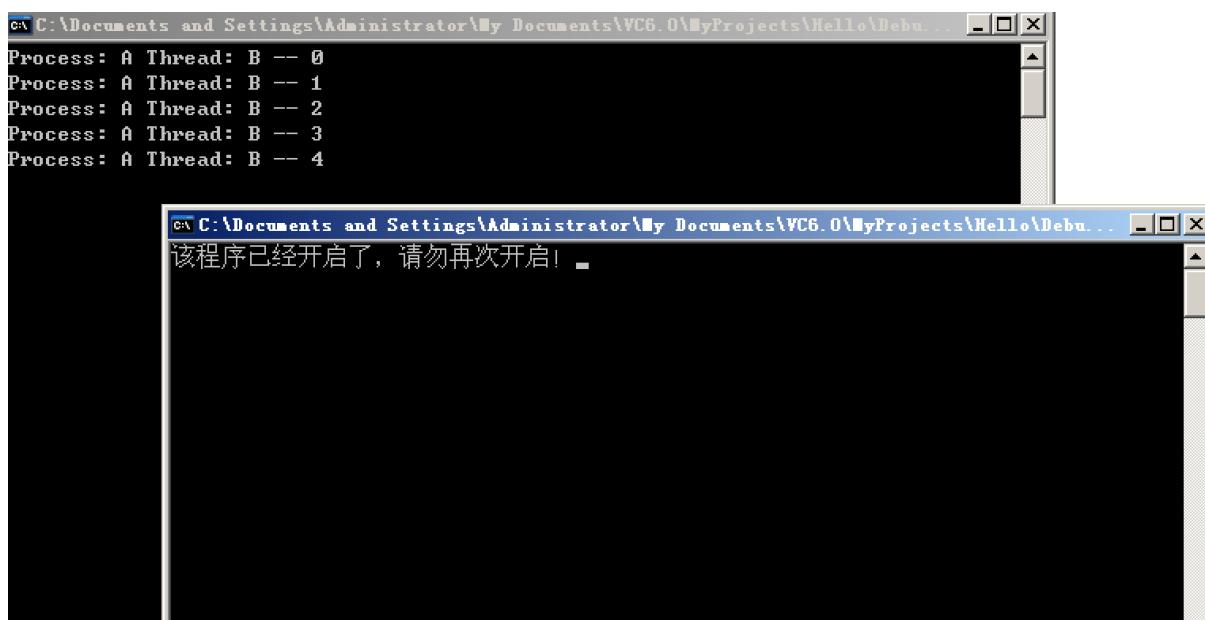
CreateMutex函数的返回值MSDN Library的介绍是这样的：如果函数成功，返回值是一个指向mutex对象的句柄；如果命名的mutex对象在函数调用前已经存在，函数返回现有对象的句柄，GetLastError返回ERROR_ALREADY_EXISTS（表示互斥体以及存在）；否则，调用者创建该mutex对象；如果函数失败，返回值为NULL，要获得扩展的错误信息，请调用GetLastError获取。

Return Values
If the function succeeds, the return value is a handle to the mutex object. If the named mutex object existed before the function call, the function returns a handle to the existing object and GetLast Error returns ERROR_ALREADY_EXISTS. Otherwise, the caller created the mutex.
If the function fails, the return value is NULL. To get extended error information, call GetLastError.

所以我们可以利用互斥体来防止程序进行多开：

```

1 #include <windows.h>
2
3 int main(int argc, char* argv[])
4 {
5     // 创建互斥体
6     HANDLE cm = CreateMutex(NULL, TRUE, "XYZ");
7     // 判断互斥体是否创建失败
8     if (cm != NULL) {
9         // 判断互斥体是否已经存在, 如果存在则表示程序被多次打开
10        if (GetLastError() == ERROR_ALREADY_EXISTS) {
11            printf("该程序已经开启了, 请勿再次开启!");
12            getchar();
13        } else {
14            // 等待互斥体状态发生变化, 也就是有信号或为互斥体拥有者, 获得令牌
15            WaitForSingleObject(cm, INFINITE);
16            // 操作资源
17            for (int i = 0; i < 5; i++) {
18                printf("Process: A Thread: B -- %d \n", i);
19                Sleep(1000);
20            }
21            // 释放令牌
22            ReleaseMutex(cm);
23        }
24    } else {
25        printf("CreateMutex 创建失败! 错误代码: %d\n", GetLastError());
26    }
27
28    return 0;
29 }
```



13 事件

事件本身也是一种内核对象，其也是用来控制线程的。

13.1 通知类型

事件本身可以做为通知类型来使用，创建事件使用函数**CreateEvent**，其语法格式如下：

```
1 HANDLE CreateEvent(
2     LPSECURITY_ATTRIBUTES lpEventAttributes, // SD 安全属性，包含安全描述符
3     BOOL bManualReset,                   // reset type 如果你希望当前事件类型是通知类型则写TRUE，反之
4     FALSE                                // initial state 初始状态，决定创建出来时候是否有信号，有为
5     TRUE, 没有为FALSE                  // object name 事件名字
6 );
```

那么通知类型到底是什么？我们可以写一段代码来看一下：

```

1 #include <windows.h>
2
3 HANDLE e_event;
4
5 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
6     // 等待事件
7     WaitForSingleObject(e_event, INFINITE);
8     printf("ThreadProc - running ...\\n");
9     getchar();
10    return 0;
11 }
12
13 DWORD WINAPI ThreadProcB(LPVOID lpParameter) {
14     // 等待事件
15     WaitForSingleObject(e_event, INFINITE);
16     printf("ThreadProcB - running ...\\n");
17     getchar();
18     return 0;
19 }
20
21 int main(int argc, char* argv[])
22 {
23
24     // 创建事件
25     // 第二个参数, FALSE表示非通知类型通知, 也就是互斥; TRUE则表示为通知类型
26     // 第三个参数表示初始状态没有信号
27     e_event = CreateEvent(NULL, TRUE, FALSE, NULL);
28
29     // 创建2个线程
30     HANDLE hThread[2];
31     hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
32     hThread[1] = CreateThread(NULL, NULL, ThreadProcB, NULL, 0, NULL);
33
34     // 设置事件为已通知, 也就是设置为有信号
35     SetEvent(e_event);
36
37     // 等待线程执行结束, 销毁内核对象
38     WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
39     CloseHandle(hThread[0]);
40     CloseHandle(hThread[1]);
41     // 事件类型也是内核对象, 所以也需要关闭句柄
42     CloseHandle(e_event);
43
44     return 0;
45 }
```

如下图所示，我们运行了代码，会发现两个线程都执行了，而如果是之前我们使用互斥体的话则线程A先执行然后线程B等待线程A归还令牌（执行结束）才会执行，这里我们在线程函数的最后使用了getchar()阻止了线程执行结束，但是两个线程还是都执行了：

```

#include "stdafx.h"
#include <windows.h>

HANDLE e_event;

DWORD WINAPI ThreadProc(LPUUID lpParameter) {
    // 等待事件
    WaitForSingleObject(e_event, INFINITE);
    printf("ThreadProc - running ...\n");
    getchar();
    return 0;
}

DWORD WINAPI ThreadProcB(LPUUID lpParameter) {
    // 等待事件
    WaitForSingleObject(e_event, INFINITE);
    printf("ThreadProcB - running ...\n");
    getchar();
    return 0;
}

int main(int argc, char* argv[])
{
    // 创建事件
    // 第二个参数, FALSE表示非通知类型通知, 也就是互斥; TRUE则表示为通知类型
    // 第三个参数表示初始状态没有信号
    e_event = CreateEvent(NULL, TRUE, FALSE, NULL);

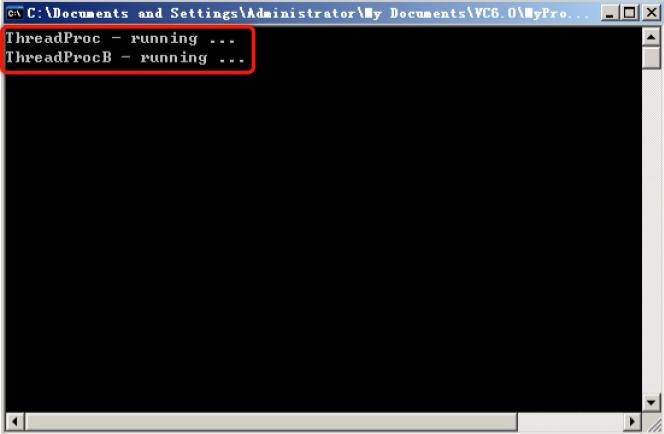
    HANDLE hThread[2];
    hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
    hThread[1] = CreateThread(NULL, NULL, ThreadProcB, NULL, 0, NULL);

    // 设置事件为已通知, 也就是设置为有信号
    SetEvent(e_event);

    // 等待线程执行结束, 销毁内核对象
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
    // 事件类型也是内核对象, 所以也需要关闭句柄
    CloseHandle(e_event);

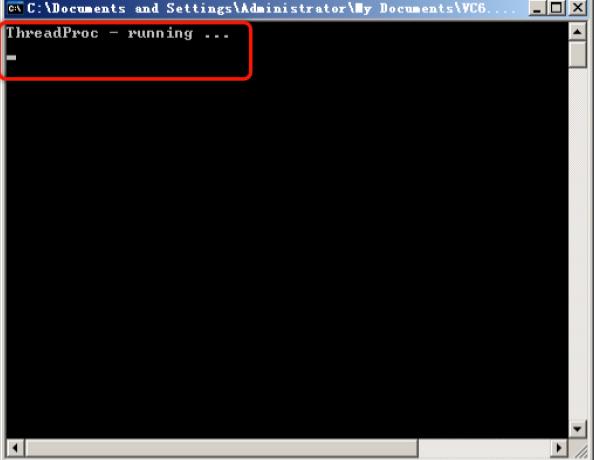
    return 0;
}

```



The screenshot shows a Windows command prompt window titled 'C:\Documents and Settings\Administrator\My Documents\VC6.0\MyPro...'. Inside the window, two lines of text are displayed: 'ThreadProc - running ...' and 'ThreadProcB - running ...'. Both lines are highlighted with a red rectangular box, indicating they are being output at the same time, which demonstrates that the threads are running concurrently.

我们修改下创建事件函数的参数为互斥，来看一下，那么互斥和通知类型的区别一下就很明显了展示出来了：



```

#include "stdafx.h"
#include <windows.h>

HANDLE e_event;

DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    // 等待事件
    WaitForSingleObject(e_event, INFINITE);
    printf("ThreadProc - running ...\n");
    getchar();
    return 0;
}

DWORD WINAPI ThreadProcB(LPVOID lpParameter) {
    // 等待事件
    WaitForSingleObject(e_event, INFINITE);
    printf("ThreadProcB - running ...\n");
    getchar();
    return 0;
}

int main(int argc, char* argv[])
{
    // 创建事件
    // 第三个参数, FALSE表示非通知类型通知, 也就是互斥, TRUE则表示为通知类型
    // 第三个参数表示初始状态没有信号
    e_event = CreateEvent(NULL, FALSE, FALSE, NULL);

    HANDLE hThread[2];
    hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
    hThread[1] = CreateThread(NULL, NULL, ThreadProcB, NULL, 0, NULL);

    // 设置事件为已通知, 也就是设置为有信号
    SetEvent(e_event);

    // 等待线程执行结束, 销毁内核对象
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
    // 事件类型也是内核对象, 所以也需要关闭句柄
    CloseHandle(e_event);

    return 0;
}

```

那么通知类型的原理是什么呢？实际上这个跟WaitForSingleObject函数有关，我们可以看下MSDN Library对该函数的介绍：

Remarks

The WaitForSingleObject function checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters the wait state. It uses no processor time while waiting for the object state to become signaled or the time-out interval to elapse.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one.

可以很清晰的看见最后说到，该函数会修改内核对象的状态，所以通知类型的原理就很简单了，就是当事件对象为通知类型时该函数就不会去修改对象的状态，这个状态我们可以理解成是占用，当WaitForSingleObject函数判断为非占用时就修改内核对象的状态为占用然后向下执行，而其他线程想使用就需要等待，这就是互斥的概念。

13.2 线程同步

线程互斥：线程互斥是指对于共享的进程系统资源，在各单个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；同步的前提是互斥，其次就是有序，互斥并不代表A线程访问临界资源后就一定是B线程再去访问，也有可能是A线程，这就是属于无序的状态，所以同步就是互斥加上有序。

13.2.1 生产者与消费者

想要证明事件和互斥体最本质的区别，我们可以使用生产者与消费者模型来举例子，那么这个模型是什么意思呢？

- ① 生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合（依赖性）问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

我们就可以理解为生产者生产一个物品，将其放进容器里，然后消费者从容器中取物品进行消费，就这样“按部就班”下去...

互斥体

首先我们来写一段互斥体下的生产者与消费者的代码：

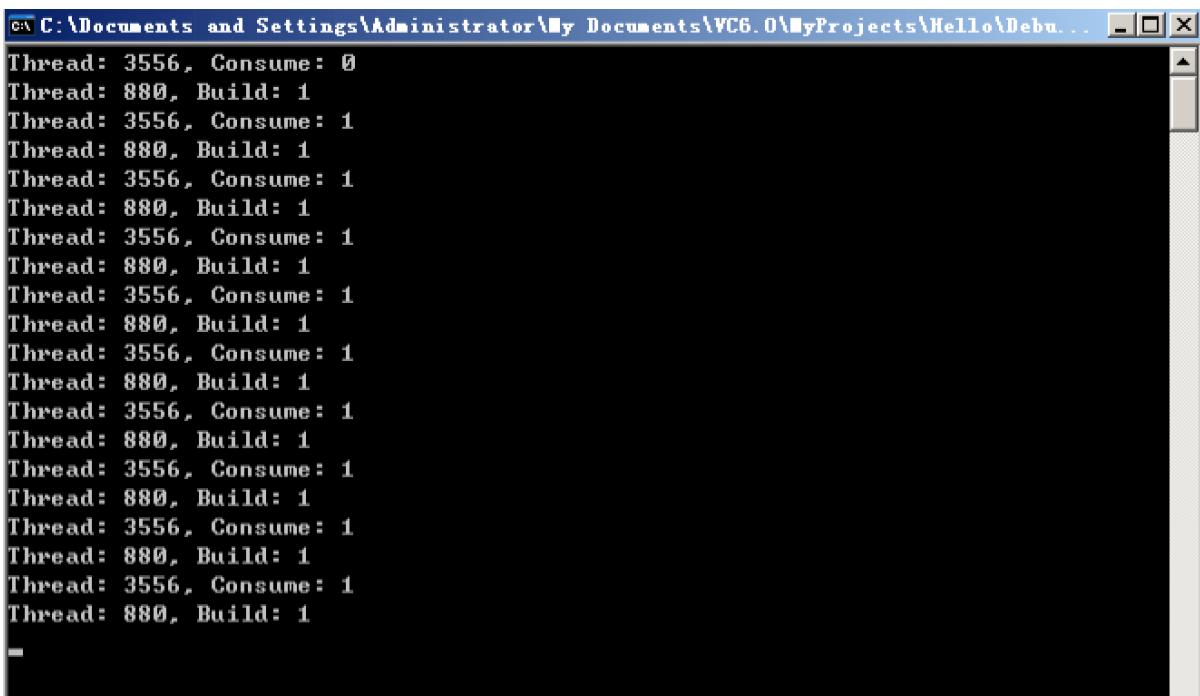
```

1 #include "stdafx.h"
2 #include <windows.h>
3
4 // 容器
5 int container;
6
7 // 次数
8 int count = 10;
9
10 // 互斥体
11 HANDLE hMutex;
12
13 // 生产者
14 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
15     for (int i = 0; i < count; i++) {
16         // 等待互斥体，获取令牌
17         WaitForSingleObject(hMutex, INFINITE);
18         // 获取当前进程ID
19         int threadId = GetCurrentThreadId();
20         // 生产存放进容器
21         container = 1;
22         printf("Thread: %d, Build: %d \n", threadId, container);
23         // 释放令牌
24         ReleaseMutex(hMutex);
25     }
26     return 0;
27 }
28
29 // 消费者
30 DWORD WINAPI ThreadProcB(LPVOID lpParameter) {
31     for (int i = 0; i < count; i++) {
32         // 等待互斥体，获取令牌
33         WaitForSingleObject(hMutex, INFINITE);
34         // 获取当前进程ID
35         int threadId = GetCurrentThreadId();
36         printf("Thread: %d, Consume: %d \n", threadId, container);
37         // 消费
38         container = 0;
39         // 释放令牌
40         ReleaseMutex(hMutex);
41     }
42     return 0;
43 }
44
45 int main(int argc, char* argv[])
46 {
47     // 创建互斥体
48     hMutex = CreateMutex(NULL, FALSE, NULL);
49
50     // 创建2个线程
51     HANDLE hThread[2];
52     hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
53     hThread[1] = CreateThread(NULL, NULL, ThreadProcB, NULL, 0, NULL);
54
55     WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

```

```
56     CloseHandle(hThread[0]);
57     CloseHandle(hThread[1]);
58     CloseHandle(hMutex);
59
60     return 0;
61 }
```

运行结果如下图所示：



我们可以清晰的看见结果并不是我们想要的，生产一次消费一次的有序进行，甚至还出现了先消费后生产的情况，这个问题我们可以去修改代码解决：

```

// 生产者
DWORD WINAPI ThreadProcA(LPVOID lpParameter) {
    for (int i = 0; i < count; i++) {
        // 判断容器
        if (container == 0) {
            // 等待互斥体，获取令牌
            WaitForSingleObject(hMutex, INFINITE);
            // 获取当前进程ID
            int threadId = GetCurrentThreadId();
            // 生产存放进容器
            container = 1;
            printf("Thread: %d, Build: %d \n", threadId, container);
        } else {
            // 保证次数
            i--;
        }
        // 释放令牌
        ReleaseMutex(hMutex);
    }
    return 0;
}

// 消费者
DWORD WINAPI ThreadProcB(LPVOID lpParameter) {
    for (int i = 0; i < count; i++) {
        // 判断容器
        if (container == 1) {
            // 等待互斥体，获取令牌
            WaitForSingleObject(hMutex, INFINITE);
            // 获取当前进程ID
            int threadId = GetCurrentThreadId();
            printf("Thread: %d, Consume: %d \n", threadId, container);
            // 消费
            container = 0;
        } else {
            // 保证次数
            i--;
        }
        // 释放令牌
        ReleaseMutex(hMutex);
    }
    return 0;
}

```

```

C:\Documents and Settings\Administrator\My Documents
Thread: 2076, Build: 1
Thread: 1140, Consume: 1

```

这样虽然看似解决了问题，但是实际上也同样会出现一种问题，那就是for循环执行了不止10次，这样会倒是过分的占用计算资源。

事件

我们使用事件的方式就可以更加完美的解决这一需求：

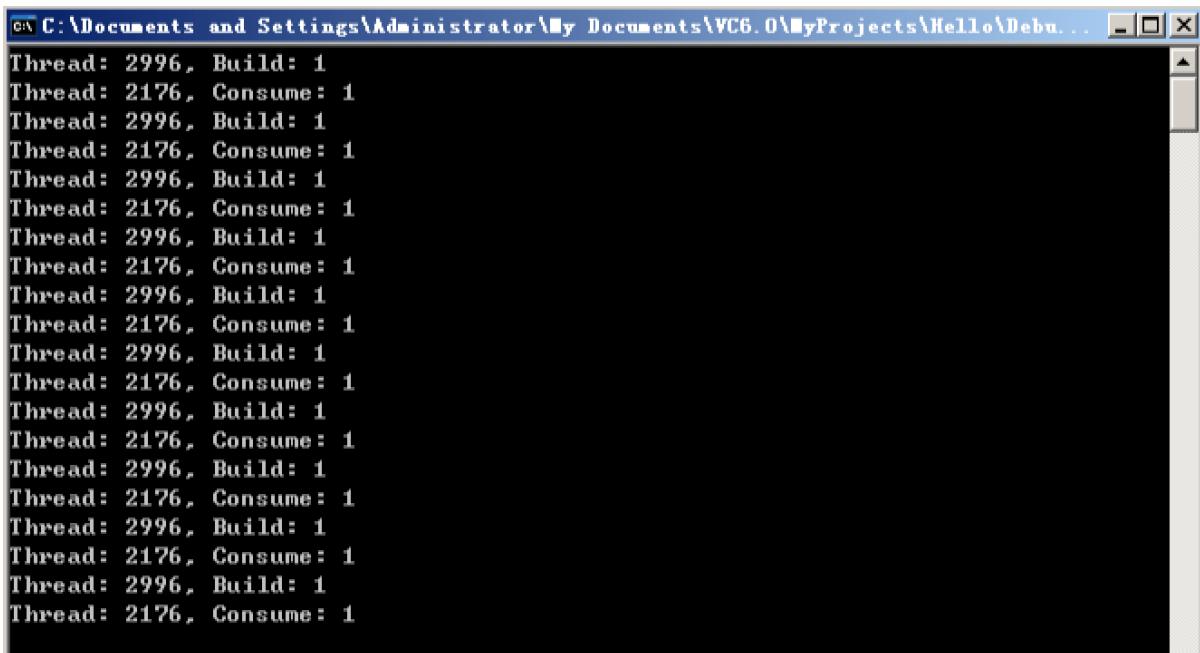
```

1 #include <windows.h>
2
3 // 容器
4 int container = 0;
5
6 // 次数
7 int count = 10;
8
9 // 事件
10 HANDLE eventA;
11 HANDLE eventB;
12
13 // 生产者
14 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
15     for (int i = 0; i < count; i++) {
16         // 等待事件，修改事件A状态
17         WaitForSingleObject(eventA, INFINITE);
18         // 获取当前进程ID
19         int threadId = GetCurrentThreadId();
20         // 生产存放进容器
21         container = 1;
22         printf("Thread: %d, Build: %d \n", threadId, container);
23         // 给eventB设置信号
24         SetEvent(eventB);
25     }
26     return 0;
27 }
28
29 // 消费者
30 DWORD WINAPI ThreadProcB(LPVOID lpParameter) {
31     for (int i = 0; i < count; i++) {
32         // 等待事件，修改事件B状态
33         WaitForSingleObject(eventB, INFINITE);
34         // 获取当前进程ID
35         int threadId = GetCurrentThreadId();
36         printf("Thread: %d, Consume: %d \n", threadId, container);
37         // 消费
38         container = 0;
39         // 给eventA设置信号
40         SetEvent(eventA);
41     }
42     return 0;
43 }
44
45 int main(int argc, char* argv[])
46 {
47     // 创建事件
48     // 线程同步的前提是互斥
49     // 顺序按照先生产后消费，所以事件A设置信号，事件B需要通过生产者线程来设置信号
50     eventA = CreateEvent(NULL, FALSE, TRUE, NULL);
51     eventB = CreateEvent(NULL, FALSE, FALSE, NULL);
52
53     // 创建2个线程
54     HANDLE hThread[2];
55     hThread[0] = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);

```

```
56     hThread[1] = CreateThread(NULL, NULL, ThreadProcB, NULL, 0, NULL);
57
58     WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
59     CloseHandle(hThread[0]);
60     CloseHandle(hThread[1]);
61     // 事件类型也是内核对象，所以也需要关闭句柄
62     CloseHandle(eventA);
63     CloseHandle(eventB);
64
65     return 0;
66 }
```

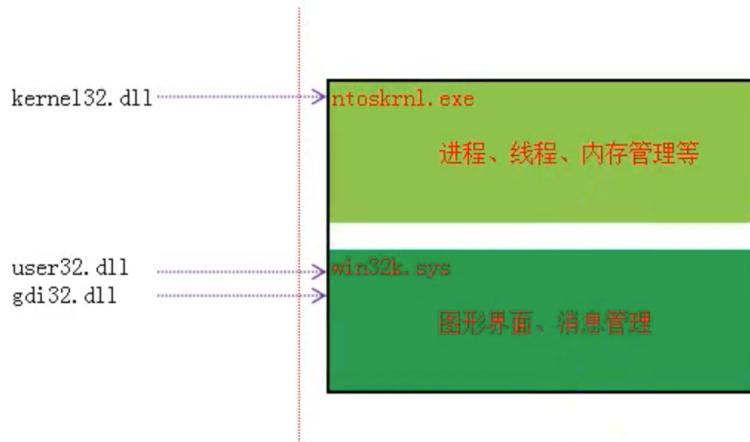
运行结果如下图：



14 窗口的本质

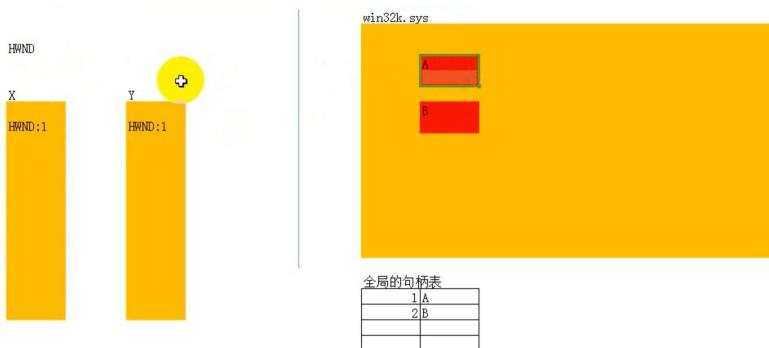
之前我们学习写的程序都是基于控制台的，而从本章开始学习图形化界面相关的知识。

之前我们所学习的进程、线程之类的函数，其接口来源于kernel32.dll → ntoskrnl.exe；而我们要学习的图形化界面的接口，它就来源于user32.dll、gdi32.dll → win32k.sys。



user32.dll和gdi32.dll的区别在哪呢？前者是你想使用Windows已经画好的界面就用它，我们称之为**GUI编程**；后者是你想自己画一个界面，例如你要画一朵花，那么就使用后者，因为这涉及到绘图相关的内容，我们称之为**GDI编程**。

之前我们了解过HANDLE句柄，其都是私有的，而在图形界面中有一个新的句柄，其叫HWND，win32k.sys提供在内核层创建图形化界面，我们想要在应用层调用就需要对应的句柄HWND，而这个句柄表是全局的，并且只有一个。



14.1 GDI - 图形设备接口

GDI是Graphics Device Interface的缩写，其中文为图形设备接口。

本章主要是学习如何进行GDI编程，但是我们在日常的工作中是不需要用到的，并且没有什么实际意义（需要的都有现成的），我们学习它就是为了来了解窗口的本质、**消息机制**的本质。

关于GDI有这么几个概念：

1. 设备对象：画的位置
2. DC (Device Contexts)：设备上下文对象（内存）
3. 图像（图形）对象：决定你要画的东西的属性

图像对象	作用
画笔(Pen)	影响线条，包括颜色、粗细、虚实、箭头形状等
画刷(Brushes)	影响对形状、区域等操作，如使用的颜色、是否有阴影等
字体(Fonts)	影响文字输出的字体
位图(Bitmap)	影响位图创建、位图操作和保存等

14.2 进行简单的绘画

如下代码就是在桌面中进行绘画，具体代码意思都在注释中了，不了解的可以在MSDN Library中查询：

```

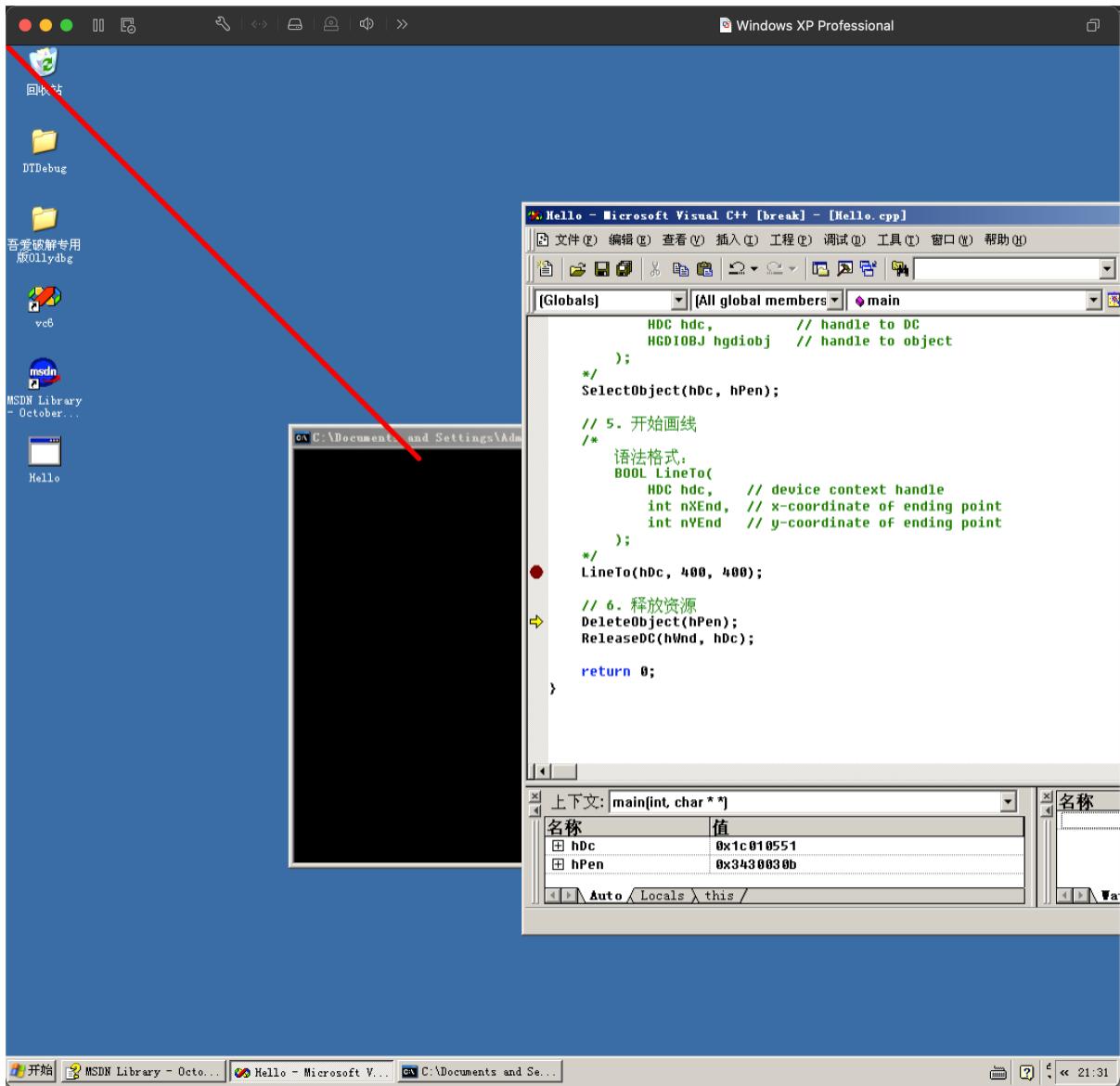
1 #include <windows.h>
2
3 int main(int argc, char* argv[])
4 {
5     HWND hWnd; // 窗口句柄
6     HDC hDc; // 设备上下文对象
7     HPEN hPen; // 画笔
8     // 1. 设备对象，要绘画的位置
9     // 设置为NULL则表示在桌面中绘画
10    hWnd = (HWND)NULL;
11
12    // 2. 获取设备的上下文对象 (DC)
13    /*
14        语法格式：
15        HDC GetDC(
16            HWND hWnd      // handle to window
17        );
18    */
19    hDc = GetDC(hWnd);
20
21    // 3. 创建画笔，设置线条的属性
22    /*
23        语法格式：
24        HPEN CreatePen(
25            int fnPenStyle,    // pen style
26            int nWidth,       // pen width
27            COLORREF crColor // pen color
28        );
29    */
30    hPen = CreatePen(PS_SOLID, 5, RGB(0xFF,00,00)); // RGB表示红绿蓝，红绿蓝的组合就可以组成新的一种颜色。
31
32    // 4. 关联
33    /*
34        语法格式：
35        HGDIOBJ SelectObject(
36            HDC hdc,           // handle to DC
37            HGDIOBJ hgdiobj // handle to object
38        );
39    */
40    SelectObject(hDc, hPen);
41
42    // 5. 开始画线
43    /*
44        语法格式：
45        BOOL LineTo(
46            HDC hdc,      // device context handle
47            int nXEnd,   // x-coordinate of ending point
48            int nYEnd    // y-coordinate of ending point
49        );
50    */
51    LineTo(hDc, 400, 400);
52
53    // 6. 释放资源
54    DeleteObject(hPen);
55    ReleaseDC(hWnd, hDc);

```

```

56
57     return 0;
58 }

```



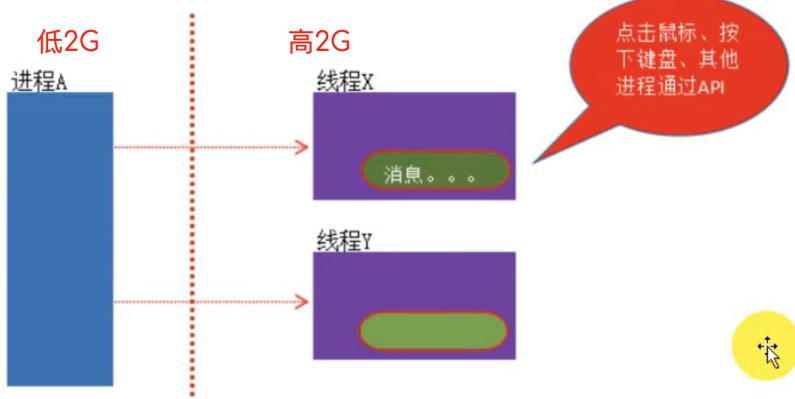
15 消息队列

15.1 什么是消息

当我们点击鼠标的时候，或者当我们按下键盘的时候，操作系统都要把这些动作记录下来，存储到一个结构体中，这个**结构体**就是消息。

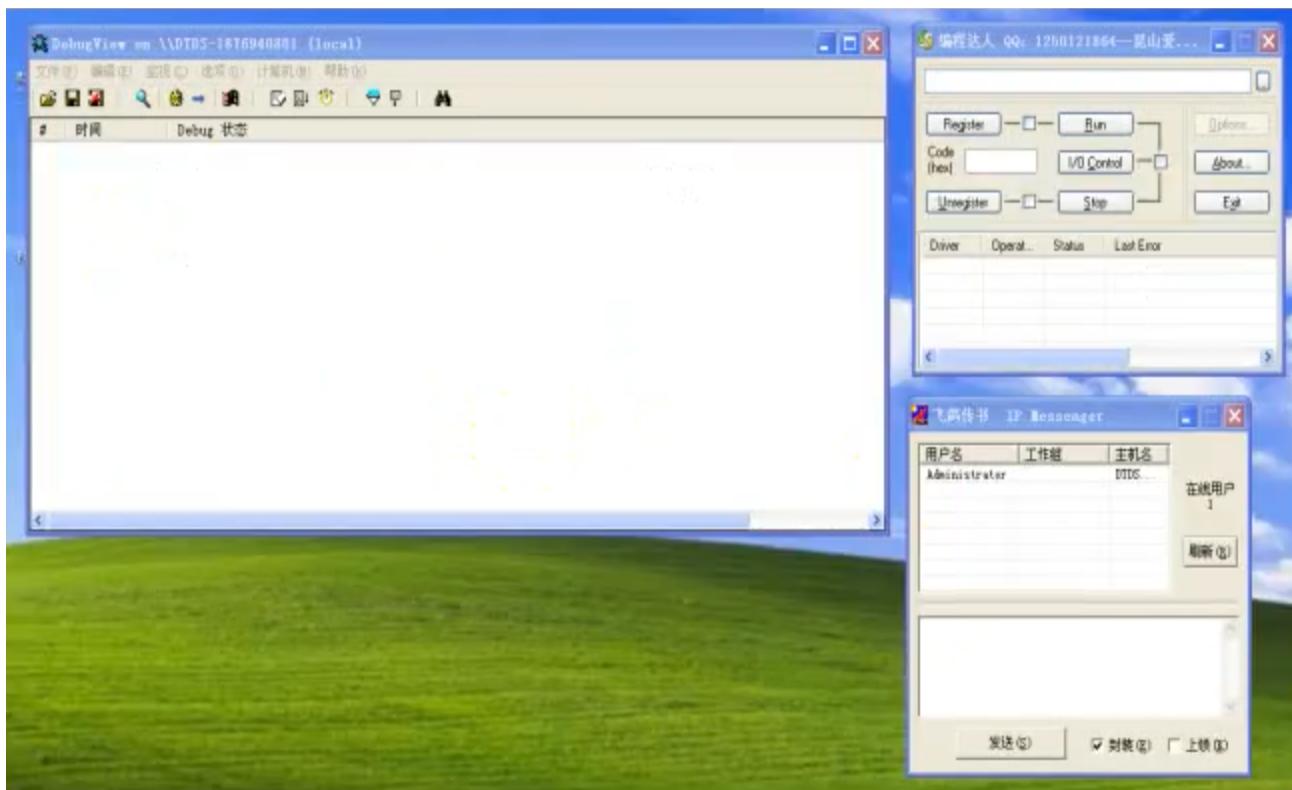
15.2 消息队列

每个线程只有一个消息队列。

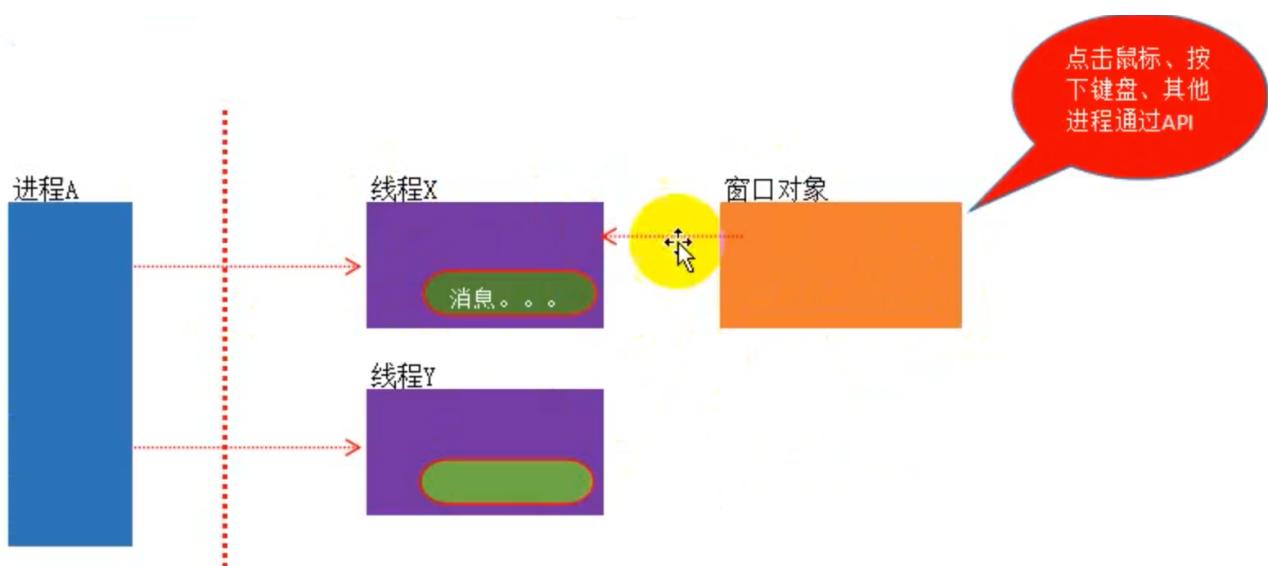


15.3 窗口与线程

当我们把鼠标点击左边窗口关闭按钮，为什么它会关闭，这个关闭（坐标、左右键...）操作系统会封装到结构体里（消息），那么这个消息如何精确的传递给对应进程的线程呢？



那是因为操作系统可以将坐标之类的作为索引，去找到对应的窗口，窗口在内核中是有窗口对象的，而这个窗口对象就会包含一个成员，这个成员就是线程对象的指针，线程又包含了消息，所以这样一个顺序就很容易理解了。

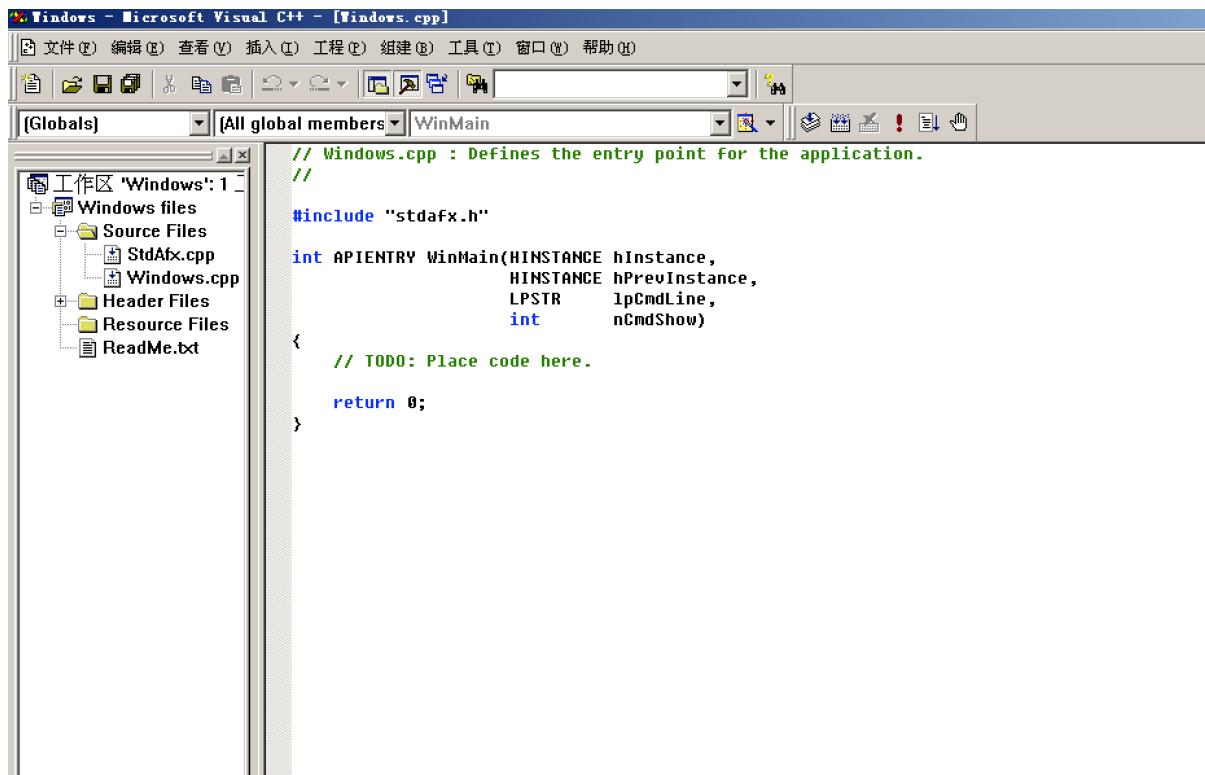


注意：一个线程可以有多个窗口，但是一个窗口只属于一个线程。

16 第一个Windwos程序

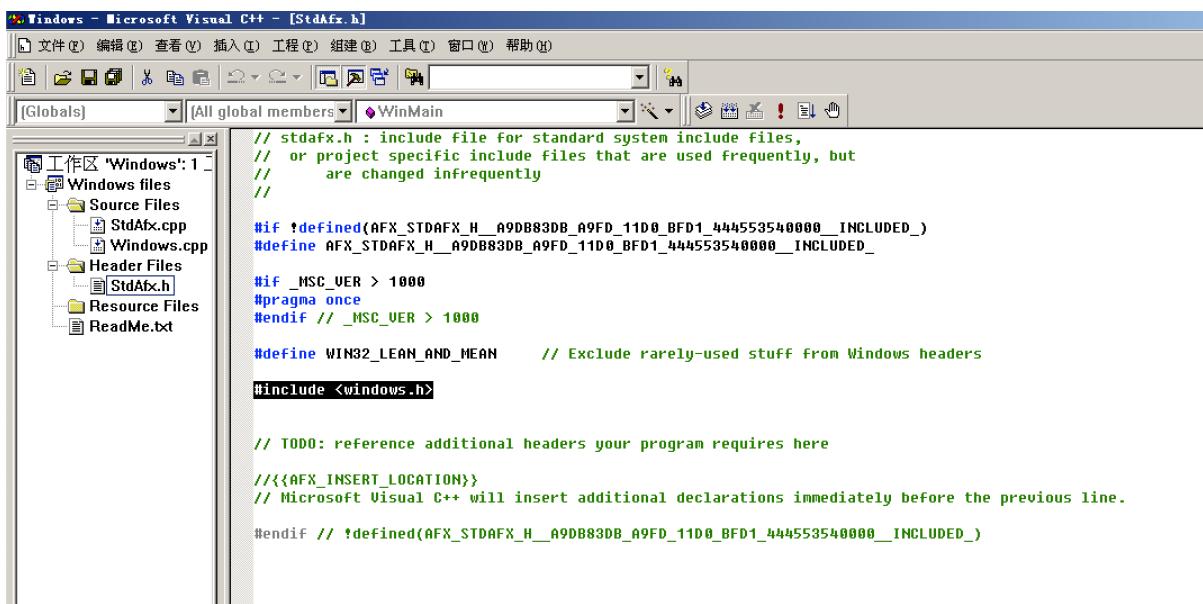
16.1 新建Windows窗口程序项目

VC6新建工程，选择Win32 Application，下一步选择一个简单的Win32的程序。



控制台程序是从Main函数为入口开始执行的，而Win32窗口程序是从WinMain函数开始执行的。

新建的项目里的头文件已经把需要用到的Windows.h头文件包含了：



16.2 WinMain函数

WinMain函数作为Win32窗口程序的入口函数，我们需要了解一下其函数的参数，语法格式如下：

1	<code>int WINAPI WinMain(</code>
2	<code>HINSTANCE hInstance, // handle to current instance</code>
3	<code>HINSTANCE hPrevInstance, // handle to previous instance</code>
4	<code>LPSTR lpCmdLine, // command line</code>
5	<code>int nCmdShow // show state</code>
6	<code>);</code>

参数解释：

1. HINSTANCE hInstance，这是一个句柄，在Win32中H开头的通常都是句柄，这里的HINSTANCE是指向模块的句柄，实际上这个值就是模块在进程空间内的内存地址；
2. HINSTANCE hPrevInstance，该参数永远为空NULL，无需理解；
3. 第三、第四个参数（LPSTR lpCmdLine、int nCmdShow）是由CreateProcess的LPTSTR lpCommandLine、LPSTARTUPINFO lpStartupInfo参数传递的。

16.3 调试信息输出

我们在窗口程序中想要输出信息就不可以使用printf了，我们可以使用另外一个函数OutputDebugString，其语法格式如下：

1	<code>void OutputDebugString(</code>
2	<code>LPCTSTR lpOutputString</code>
3	<code>);</code>

传参就是一个LPCTSTR类型（字符串），但是需要注意的是这个函数只能打印固定字符串，不能打印格式化的字符串，所以如果需要格式化输出，需要在这之前使用sprintf函数进行格式化（自行查阅），这里我们可以尝试输出当前模块的句柄：

```
1 #include "stdafx.h"
2
3 int APIENTRY WinMain(HINSTANCE hInstance,
4                     HINSTANCE hPrevInstance,
5                     LPSTR     lpCmdLine,
6                     int       nCmdShow)
7 {
8     // TODO: Place code here.
9     DWORD dwAddr = (DWORD)hInstance;
10
11    char szOutBuff[0x80];
12    sprintf(szOutBuff, "hInstance address: %x \n", dwAddr); // 该函数需要包含stdio.h头文件
13    OutputDebugString(szOutBuff);
14
15    return 0;
16 }
```

运行该代码就会在Debug输出框中发现打印的字符串，这就是一个内存地址：

```

Windows - Microsoft Visual C++ - [Windows.cpp]
File Edit View Insert Project Build Tools Window Help
[Globals] [All global members] WinMain
WinMain : int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // TODO: Place code here.
    DWORD dwAddr = (DWORD)hInstance;

    char szOutBuff[0x80];
    sprintf(szOutBuff, "hInstance address: %x \n", dwAddr);
    OutputDebugString(szOutBuff);

    return 0;
}

Loaded 'ntdll.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\kernel32.dll', no matching symbolic information found.
hInstance address: 400000
The thread 0xA94 has exited with code 0 (0x0).

```

16.4 创建窗口程序

如下代码创建了一个简单的窗口程序：

```

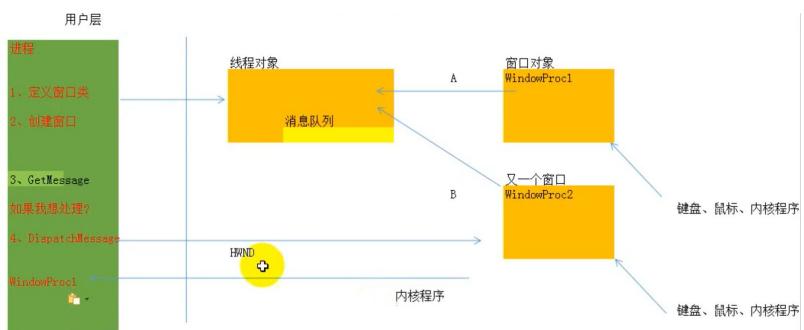
1 // Windows.cpp : Defines the entry point for the application.
2 //
3
4 #include "stdafx.h"
5
6 // 窗口函数定义
7 LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
8     // 必须要调用一个默认的消息处理函数，关闭、最小化、最大化都是由默认消息处理函数处理的
9     return DefWindowProc(hwnd, uMsg, wParam, lParam);
10 }
11
12 int APIENTRY WinMain(HINSTANCE hInstance,
13                     HINSTANCE hPrevInstance,
14                     LPSTR     lpCmdLine,
15                     int       nCmdShow)
16 {
17     char szOutBuff[0x80];
18
19     // 1. 定义创建的窗口(创建注册窗口类)
20     TCHAR className[] = TEXT("My First Window");
21     WNDCLASS wndClass = {0};
22     // 设置窗口背景色
23     wndClass.hbrBackground = (HBRUSH)COLOR_BACKGROUND;
24     // 设置类名字
25     wndClass.lpszClassName = className;
26     // 设置模块地址
27     wndClass.hInstance = hInstance;
28     // 处理消息的窗口函数
29     wndClass.lpfWndProc = WindowProc; // 不是调用函数，只是告诉操作系统，当前窗口对应的窗口回调函数是什么
30     // 注册窗口类
31     RegisterClass(&wndClass);
32
33     // 2. 创建并显示窗口
34     // 创建窗口
35     /*
36     CreateWindow 语法格式：
37     HWND CreateWindow(
38         LPCTSTR lpClassName, // registered class name 类名字
39         LPCTSTR lpWindowName, // window name 窗口名字
40         DWORD dwStyle, // window style 窗口外观的样式
41         int x, // horizontal position of window 相对于父窗口x坐标
42         int y, // vertical position of window 相对于父窗口y坐标
43         int nWidth, // window width 窗口宽度：像素
44         int nHeight, // window height 窗口长度：像素
45         HWND hWndParent, // handle to parent or owner window 父窗口句柄
46         HMENU hMenu, // menu handle or child identifier 菜单句柄
47         HINSTANCE hInstance, // handle to application instance 模块
48         LPVOID lpParam // window-creation data 附加数据
49     );
50     */
51     HWND hWnd = CreateWindow(className, TEXT("窗口"), WS_OVERLAPPEDWINDOW, 10, 10, 600, 300, NULL,
52     NULL, hInstance, NULL);
53
54     if (hWnd == NULL) {
55         // 如果为NULL则窗口创建失败，输出错误信息
56     }

```

```

55         sprintf(szOutBuff, "Error: %d", GetLastError());
56         OutputDebugString(szOutBuff);
57         return 0;
58     }
59
60     // 显示窗口
61     /*
62     ShowWindow 语法格式：
63     BOOL ShowWindow(
64         HWND hWnd,          // handle to window 窗口句柄
65         int nCmdShow      // show state 显示的形式
66     );
67     */
68     ShowWindow(hWnd, SW_SHOW);
69
70     // 3. 接收消息并处理
71     /*
72     GetMessage 语法格式：
73     BOOL GetMessage(
74         LPMMSG lpMsg,           // message information OUT类型参数，这是一个指针
75         // 后三个参数都是过滤条件
76         HWND hWnd,            // handle to window 窗口句柄，如果为NULL则表示该线程中的所有消息都要
77         UINT wMsgFilterMin,   // first message 第一条信息
78         UINT wMsgFilterMax    // last message 最后一条信息
79     );
80     */
81     MSG msg;
82     BOOL bRet;
83     while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0 )
84     {
85         if (bRet == -1)
86         {
87             // handle the error and possibly exit
88             sprintf(szOutBuff, "Error: %d", GetLastError());
89             OutputDebugString(szOutBuff);
90             return 0;
91         }
92         else
93         {
94             // 转换消息
95             TranslateMessage(&msg);
96             // 分发消息：就是给系统调用窗口处理函数
97             DispatchMessage(&msg);
98         }
99     }
100
101     return 0;
102 }
```

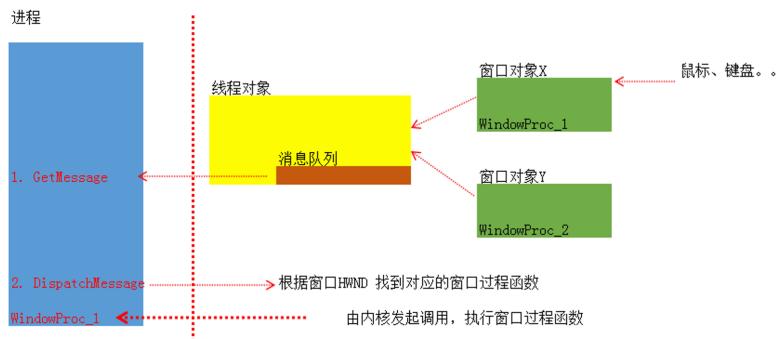
如下图是窗口程序创建执行流程：



17 消息类型

17.1 消息的产生与处理流程

消息的产生与处理流程，从消息发起这个点开始说，假设我们点击了某个窗口时就会产生一个消息，操作系统得到这个消息后先判断当前点击的是哪个窗口，找到对应的窗口对象，再根据窗口对象里的某一个成员找到对应线程，一旦找到了对应线程，操作系统就会把封装好的消息（这是一个结构体，包含了你鼠标点击的坐标等等消息）存到对应的消息队列里，应用程序就会通过GetMessage不停的从消息队列中取消息。



17.2 消息结构体

我们是通过GetMessage函数接收消息的，其第一个参数就是接收的消息（结构体），所以可以在之前的代码中选中MSG然后F12跟进看一下消息结构体的定义：

```

/*
 * Message structure
 */
typedef struct tagMSG {
    HWND         hwnd;
    UINT         message;
    WPARAM       wParam;
    LPARAM       lParam;
    DWORD        time;
    POINT        pt;
#ifndef _MAC
    DWORD        lPrivate;
#endif
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMMSG;

```

1	typedef struct tagMSG {
2	HWND hwnd; // 所属窗口句柄
3	UINT message; // 消息类型：编号
4	WPARAM wParam; // 附加数据，进一步描述消息的
5	LPARAM lParam; // 附加数据，进一步描述消息的
6	DWORD time; // 消息产生的时间
7	POINT pt; // 在哪里产生的
8	} MSG, *PMSG;

能产生消息的情况有四种情况：**1. 键盘 2. 鼠标 3. 其他应用程序 4. 操作系统内核程序**，有这么多消息要处理，所以操作系统会将所有消息区分类别，每个消息都有独一无二的编号。

消息这个结构体存储的信息也不多，只能知道消息属于哪个窗口，根本不知道对应窗口函数是什么，所以我们不得不在之后对消息进行分发（DispatchMessage函数），而后由内核发起调用来执行窗口函数。

换而言之，我们这个消息的结构体实际上就是传递给了窗口函数，其四个参数对应着消息结构体的前四个成员。

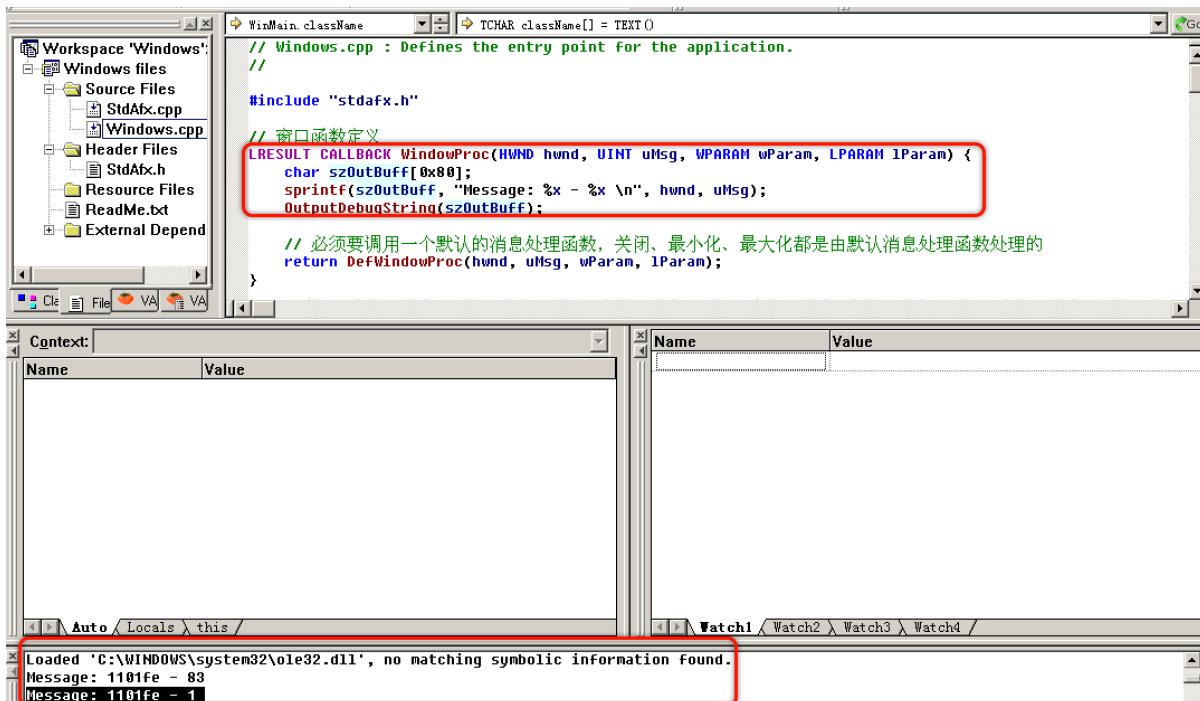
17.3 消息类型

我们想要关注自己想要关注的消息类型，首先可以在窗口函数中打印消息类型来看看都有什么消息类型：

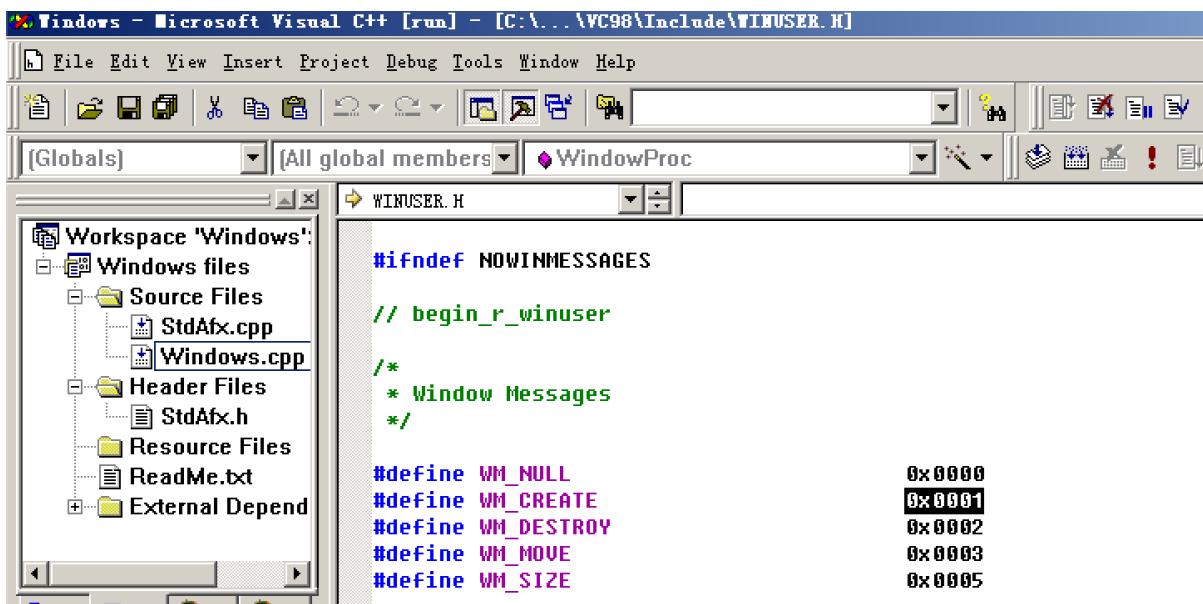
```

1 // 窗口函数定义
2 LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3     char szOutBuff[0x80];
4     sprintf(szOutBuff, "Message: %x - %x \n", hwnd, uMsg);
5     OutputDebugString(szOutBuff);
6
7     // 必须要调用一个默认的消息处理函数，关闭、最小化、最大化都是由默认消息处理函数处理的
8     return DefWindowProc(hwnd, uMsg, wParam, lParam);
9 }

```



可以看见这边输出了一个0x1，想要知道这个对应着什么，我们可以在**C:\Program Files\Microsoft Visual Studio\VC98\Include**目录中找到**WINUSER.H**这个文件来查看，搜索0x0001就可以找到：



那么我们可以看见对应的宏就是**WM_CREATE**，这个消息的意思就是窗口创建，所以我们有很多消息是不需要关注的，而且消息时刻都在产生，非常非常多。

17.3.1 处理窗口关闭

在窗口关闭时，实际上进程并不会关闭，所以我们需要在窗口函数中筛选条件，当窗口关闭了就退出进程。

```

1 // 窗口函数定义
2 LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3     switch(uMsg) {
4         // 当窗口关闭则退出进程
5         case WM_DESTROY:
6             {
7                 PostQuitMessage(0);
8                 break;
9             }
10    }
11
12    // 必须要调用一个默认的消息处理函数，关闭、最小化、最大化都是由默认消息处理函数处理的
13    return DefWindowProc(hwnd, uMsg, wParam, lParam);
14 }

```

17.3.2 处理键盘按下

我们除了可以处理窗口关闭，处理键盘按下也是没问题的，键盘按下的宏是**WM_KEYDOWN**，但是我们想要按下a这个键之后才处理该怎么办？首先我们需要查阅一下MSDN Library：

```

1 LRESULT CALLBACK WindowProc(
2     HWND hwnd,           // handle to window
3     UINT uMsg,          // WM_KEYDOWN
4     WPARAM wParam,      // virtual-key code
5     LPARAM lParam       // key data
6 );

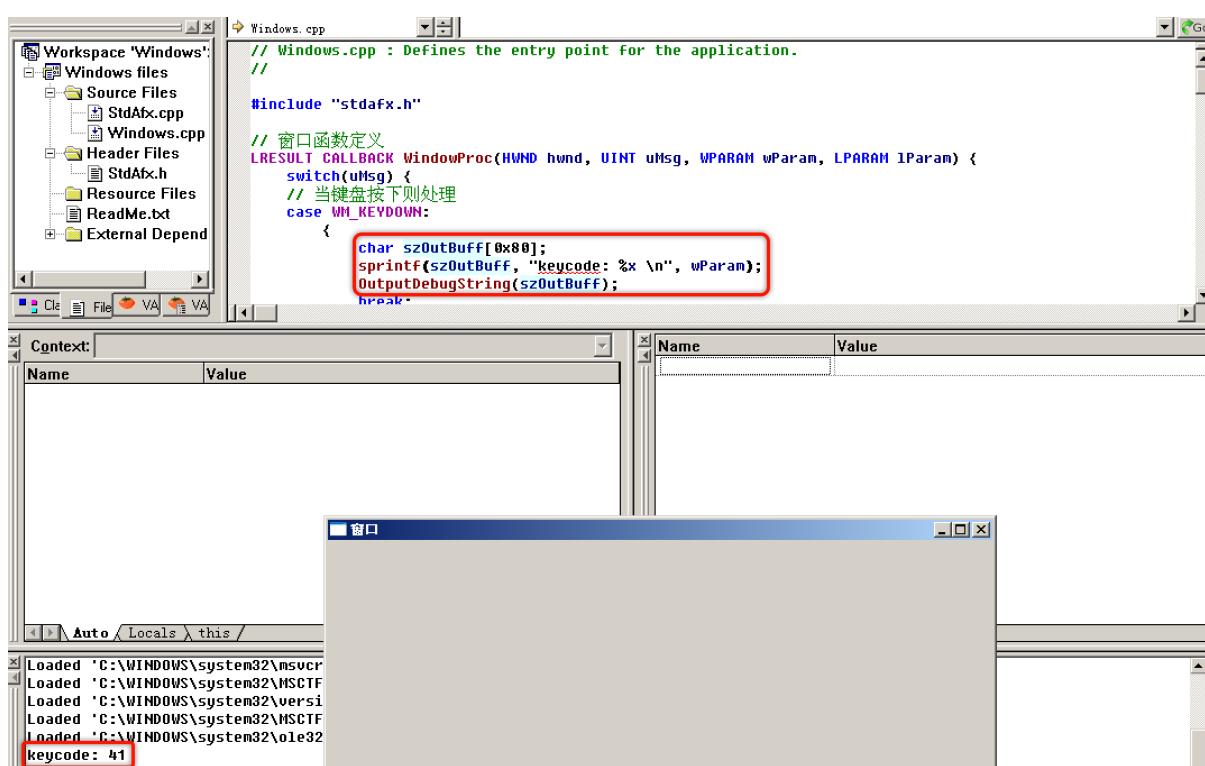
```

可以很清楚的看见窗口函数的第三个参数就是虚拟键码（键盘上每个键都对应一个虚拟键码），我们可以输出下按下a，其对应虚拟键码是什么：

```

1 // 窗口函数定义
2 LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
3     switch(uMsg) {
4         // 当键盘按下则处理
5         case WM_KEYDOWN:
6             {
7                 char szOutBuff[0x80];
8                 sprintf(szOutBuff, "keycode: %x \n", wParam);
9                 OutputDebugString(szOutBuff);
10                break;
11            }
12        }
13
14     // 必须要调用一个默认的消息处理函数，关闭、最小化、最大化都是由默认消息处理函数处理的
15     return DefWindowProc(hwnd, uMsg, wParam, lParam);
16 }

```



如上图所示，按下a之后输出的虚拟键码是0x41，所以我们可以根据这个来进行判断。

17.4 转换消息

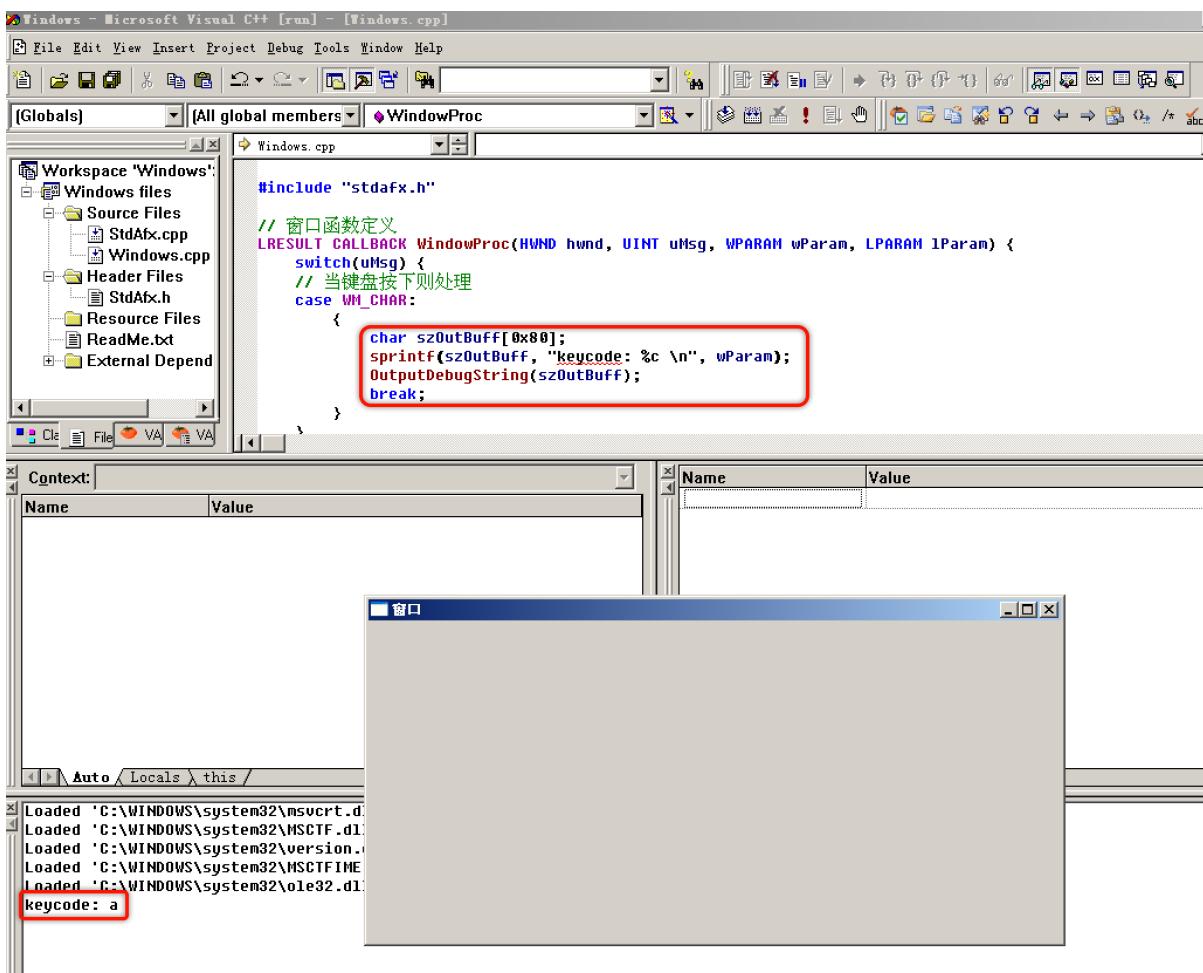
之前我们举例可以处理键盘按下的消息，但是我们想要直观的看到底输入了什么而不是虚拟键码该怎么办？这时候我们就需要使用WM_CHAR这个宏了，但是在这之前，我们的消息是必须要经过转换的，只有其转换了，我们的虚拟键码才能变成具体的字符。

```
else
{
    // 转换消息
TranslateMessage(&msg);
    // 分发消息：就是给系统调用窗口处理函数
DispatchMessage(&msg);
}
```

WM_CHAR宏对应的窗口函数参数作用如下：

1	LRESULT CALLBACK WindowProc(
2	HWND hwnd, // handle to window
3	UINT uMsg, // WM_CHAR
4	WPARAM wParam, // character code (TCHAR)
5	LPARAM lParam // key data
6);

第三个参数就是字符所以我们直接输出这个即可：



18 子窗口控件

18.1 关于子窗口控件

1. Windows提供了几个预定义的窗口类以方便我们的使用，我们一般叫它们为子窗口控件，简称控件；
2. 控件会自己处理消息，并在自己状态发生改变时通知父窗口；
3. 预定义的控件有：按钮、复选框、编辑框、静态字符串标签和滚动条等。

18.2 创建编辑框和按钮

我们想使用子窗口控件可以使用CreateWindow函数来创建，创建位置我们可以选在窗口函数中，当窗口创建则开始创建子窗口控件。

```

1 // Windows.cpp : Defines the entry point for the application.
2 //
3
4 #include "stdafx.h"
5 // 定义子窗口标识
6 #define CWA_EDIT 0x100
7 #define CWA_BUTTON_0 0x101
8 #define CWA_BUTTON_1 0x102
9
10 // 定义全局模块
11 HINSTANCE gHinstance;
12
13
14 // 窗口函数定义
15 LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
16     switch (uMsg) {
17         // 当键盘按下则处理
18         case WM_CHAR:
19             {
20                 char szOutBuff[0x80];
21                 sprintf(szOutBuff, "keycode: %c \n", wParam);
22                 OutputDebugString(szOutBuff);
23                 break;
24             }
25         // 当窗口创建则开始创建子窗口控件
26         case WM_CREATE:
27             {
28                 // 创建编辑框
29                 CreateWindow(
30                     TEXT("EDIT"), // registered class name 注册的类名, 使用EDIT则为编辑框
31                     TEXT(""), // window name 窗口名称
32                     WS_CHILD | WS_VISIBLE | WS_VSCROLL | ES_MULTILINE, // window style 子窗口控件样式：子窗口、创建后可以看到、滚动条、自动换行
33                     0, // horizontal position of window 在父窗口上的x坐标
34                     0, // vertical position of window 在父窗口上的y坐标
35                     400, // window width 控件宽度
36                     300, // window height 控件高度
37                     hwnd, // menu handle or child identifier 父窗口句柄
38                     (HMENU)CWA_EDIT, // menu handle or child identifier 子窗口标识
39                     gHinstance, // handle to application instance 模块
40                     NULL // window-creation data 附加数据
41                 );
42
43                 // 创建"设置"按钮
44                 CreateWindow(
45                     TEXT("BUTTON"), // registered class name 注册的类名, 使用BUTTON则为按钮
46                     TEXT("设置"), // window name 按钮名称
47                     WS_CHILD | WS_VISIBLE, // window style 子窗口控件样式：子窗口、创建后可以看到
48                     450, // horizontal position of window 在父窗口上的x坐标
49                     150, // vertical position of window 在父窗口上的y坐标
50                     80, // window width 控件宽度
51                     20, // window height 控件高度
52                     hwnd, // menu handle or child identifier 父窗口句柄
53                     (HMENU)CWA_BUTTON_0, // menu handle or child identifier 子窗口标识
54                     gHinstance, // handle to application instance 模块

```

```

55             NULL          // window-creation data 附加数据
56         );
57
58     // 创建"获取"按钮
59     CreateWindow(
60         TEXT("BUTTON"),    // registered class name 注册的类名，使用BUTTON则为按钮
61         TEXT("获取"),    // window name 按钮名称
62         WS_CHILD | WS_VISIBLE,      // window style 子窗口控件样式：子窗口、创建后可以看到
63         450,                // horizontal position of window 在父窗口上的x坐标
64         100,                // vertical position of window 在父窗口上的y坐标
65         80,                 // window width 控件宽度
66         20,                 // window height 控件高度
67         hwnd,               // menu handle or child identifier 父窗口句柄
68         (HMENU)CWA_BUTTON_1,        // menu handle or child identifier 子窗口标识
69         gInstance,            // handle to application instance 模块
70         NULL                 // window-creation data 附加数据
71     );
72
73     break;
74 }
75 // 当按钮点击则处理
76 case WM_COMMAND:
77 {
78     // 宏WM_COMMAND中，wParam参数的低16位中有标识，根据标识我们才能判断哪个按钮和编辑框，使用LOWORD()
可以获取低16位
79     switch (LOWORD(wParam)) {
80         // 当按钮为设置
81         case CWA_BUTTON_0:
82         {
83             // SetDlgItemText函数修改编辑框内容
84             SetDlgItemText(hwnd, (int)CWA_EDIT, TEXT("HACK THE WORLD"));
85             break;
86         }
87         // 当按钮为获取
88         case CWA_BUTTON_1:
89         {
90             // MessageBox弹框输出编辑框内容
91             TCHAR szEditBuffer[0x80];
92             GetDlgItemText(hwnd, (int)CWA_EDIT, szEditBuffer, 0x80);
93             MessageBox(NULL, szEditBuffer, NULL, NULL);
94             break;
95         }
96     }
97     break;
98 }
99 }
100 // 必须要调用一个默认的消息处理函数，关闭、最小化、最大化都是由默认消息处理函数处理的
101 return DefWindowProc(hwnd, uMsg, wParam, lParam);
102 }
103 }
104 int APIENTRY WinMain(HINSTANCE hInstance,
105                     HINSTANCE hPrevInstance,
106                     LPSTR     lpCmdLine,
107                     int       nCmdShow)
108 {
109 }
```

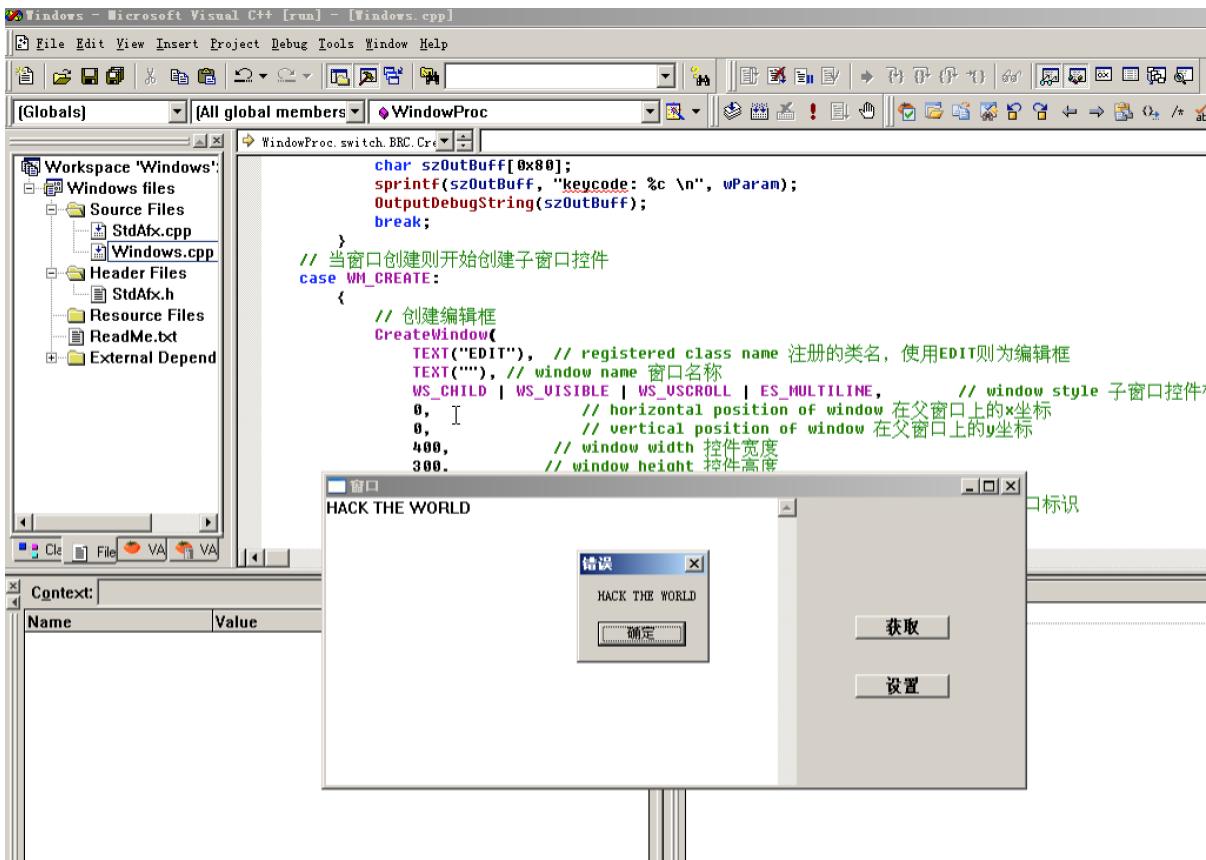
```

110     char szOutBuff[0x80];
111
112     // 1. 定义创建的窗口(创建注册窗口类)
113     TCHAR className[] = TEXT("My First Window");
114     WNDCLASS wndClass = {0};
115     // 设置窗口背景色
116     wndClass.hbrBackground = (HBRUSH)COLOR_BACKGROUND;
117     // 设置类名字
118     wndClass.lpszClassName = className;
119     // 设置模块地址
120     gHinstance = hInstance;
121     wndClass.hInstance = hInstance;
122     // 处理消息的窗口函数
123     wndClass.lpfWndProc = WindowProc; // 不是调用函数，只是告诉操作系统，当前窗口对应的窗口回调函数是什么
124     // 注册窗口类
125     RegisterClass(&wndClass);
126
127     // 2. 创建并显示窗口
128     // 创建窗口
129     /*
130     CreateWindow 语法格式：
131     HWND CreateWindow(
132         LPCTSTR lpClassName, // registered class name 类名字
133         LPCTSTR lpWindowName, // window name 窗口名字
134         DWORD dwStyle, // window style 窗口外观的样式
135         int x, // horizontal position of window 相对于父窗口x坐标
136         int y, // vertical position of window 相对于父窗口y坐标
137         int nWidth, // window width 窗口宽度：像素
138         int nHeight, // window height 窗口长度：像素
139         HWND hWndParent, // handle to parent or owner window 父窗口句柄
140         HMENU hMenu, // menu handle or child identifier 菜单句柄
141         HINSTANCE hInstance, // handle to application instance 模块
142         LPVOID lpParam // window-creation data 附加数据
143     );
144     */
145     hWnd = CreateWindow(className, TEXT("窗口"), WS_OVERLAPPEDWINDOW, 10, 10, 600, 300, NULL,
146     NULL, hInstance, NULL);
147
148     if (hWnd == NULL) {
149         // 如果为NULL则窗口创建失败，输出错误信息
150         sprintf(szOutBuff, "Error: %d", GetLastError());
151         OutputDebugString(szOutBuff);
152         return 0;
153     }
154
155     // 显示窗口
156     /*
157     ShowWindow 语法格式：
158     BOOL ShowWindow(
159         HWND hWnd, // handle to window 窗口句柄
160         int nCmdShow // show state 显示的形式
161     );
162     */
163     ShowWindow(hWnd, SW_SHOW);
164
// 3. 接收消息并处理

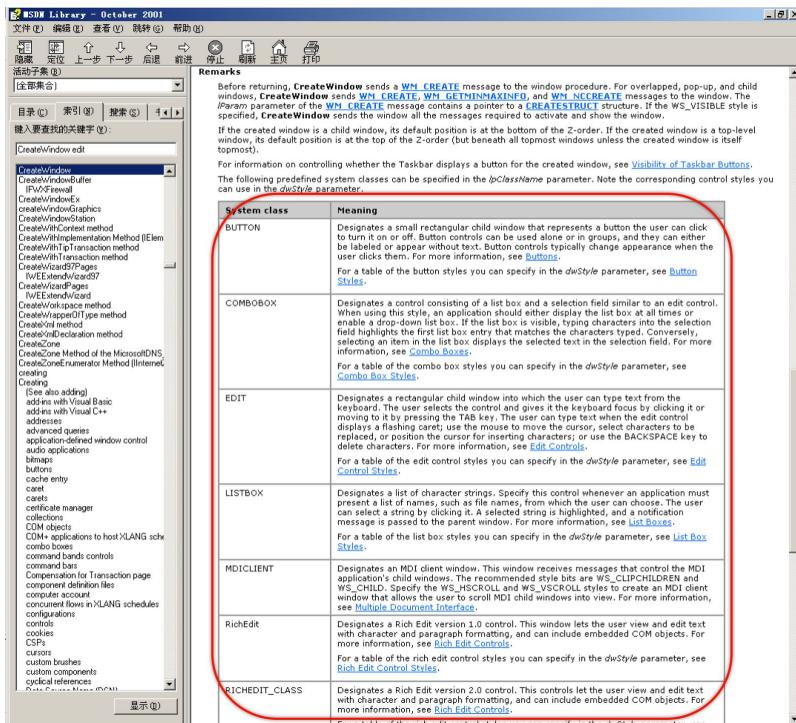
```

```
165  /*
166  GetMessage 语法格式：
167  BOOL GetMessage(
168      LPMSG lpMsg,           // message information OUT类型参数，这是一个指针
169      // 后三个参数都是过滤条件
170      HWND hWnd,            // handle to window 窗口句柄，如果为NULL则表示该线程中的所有消息都要
171      UINT wMsgFilterMin,   // first message 第一条信息
172      UINT wMsgFilterMax); // last message 最后一条信息
173 );
174 */
175 MSG msg;
176 BOOL bRet;
177 while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
178 {
179     if (bRet == -1)
180     {
181         // handle the error and possibly exit
182         sprintf(szOutBuff, "Error: %d", GetLastError());
183         OutputDebugString(szOutBuff);
184         return 0;
185     }
186     else
187     {
188         // 转换消息
189         TranslateMessage(&msg);
190         // 分发消息：就是给系统调用窗口处理函数
191         DispatchMessage(&msg);
192     }
193 }
194
195 return 0;
196 }
```

运行结果如下：



Windows预定义的窗口类可以在MSDN Library的CreateWindow函数下面找到：

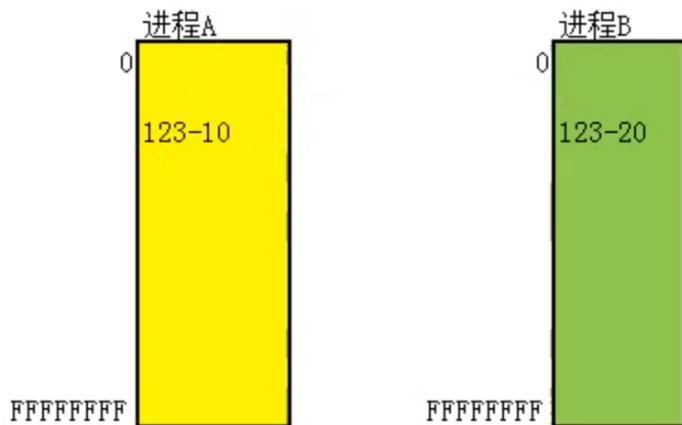


19 虚拟内存与物理内存

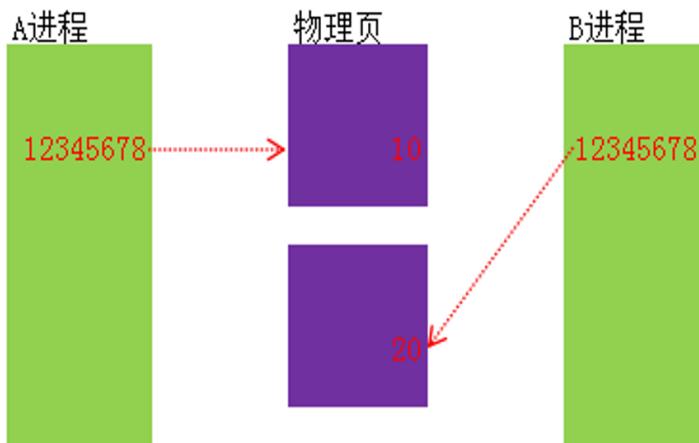
19.1 虚拟内存与物理内存的关系

每个进程都有自己的4GB内存，但是这个4GB内存并不是真实存在的，而是一块虚拟内存。

在进程A的0x12345678内存地址中存入一个值，在进程B的0x12345678内存地址中也存入一个值，两者并不会冲突，而是各自存放各自的。



但是存放的这个值是存放在物理内存上的，所以这里的虚拟内存和物理内存就有一个对应关系，当你真正使用的时候才会给分配物理内存，不使用的时候则就只有虚拟内存（空头支票）。



每一个物理内存的大小是4KB，按照4KB大小来分页（Page），所以如上图所示，就有物理页这个概念。

19.2 虚拟内存地址划分

每个进程都有4GB的虚拟内存，虚拟内存的地址是如何划分的？首先，我们需要知道一个虚拟内存分为高2G、低2G。

如下图所示，用户空间是低2G，内核空间是高2G，对我们来说只能使用低2G的用户空间，高2G内核空间是所有进程共用的。

但是需要注意的是低2G的用户空间使用还有前64KB的空指针赋值区和后64KB的用户禁入区是我们目前不能使用的。



分区	x86 32位Windows
空指针赋值区	0x00000000 - 0x0000FFFF
用户模式区	0x00010000 - 0x7FFEFFFF
64KB禁入区	0x7FFF0000 - 0x7FFFFFFF
内核	0x80000000 - 0xFFFFFFFF

术语：线性地址就是虚拟内存的地址

特别说明：线性地址有4G，但未必都能访问，所以需要记录哪些地方分配了。

19.3 物理内存

19.3.1 可使用的物理内存

为了管理方便，物理内存以4KB大小来分页，那么在系统里面这个物理页的数量是多少呢？我使用的虚拟机是可以设置内存大小的（从物理上可以理解为这就是一个内存条）：



比如我现在的是是2GB（2048MB），我们可以在任务管理器清晰的看见物理内存的总数是接近 2048×1024 的：

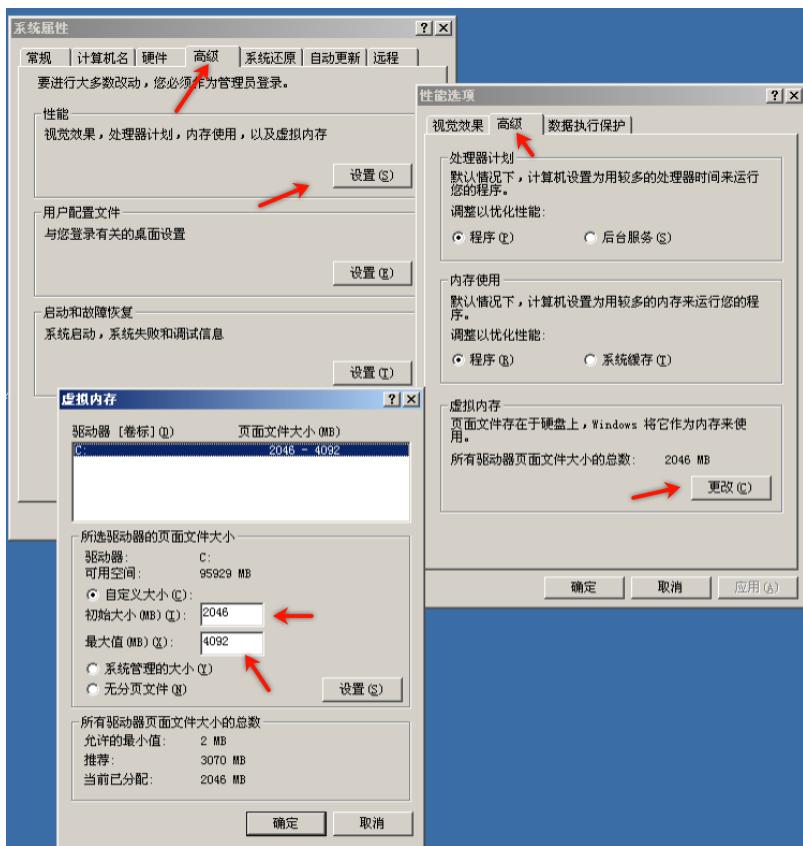


那么这一块物理内存能有多少物理页呢？我们可以将总数/4：



也就是有524138个物理页（十进制），转为十六进制就是0x7FF6A

那么物理页面只有这些不够用该怎么办？这时候操作系统会分配硬盘空间来做虚拟内存。我们可以通过系统属性来查看、更改当前分配的虚拟内存大小：



可以看见当前初始大小是2046MB，那么这个是存放在哪的呢？我们可以在C盘下查看（需要显示系统隐藏文件）pagefile.sys这个文件，它刚好是2046MB这个大小，这个文件就是用来做虚拟内存的：



19.3.2 可识别的物理内存

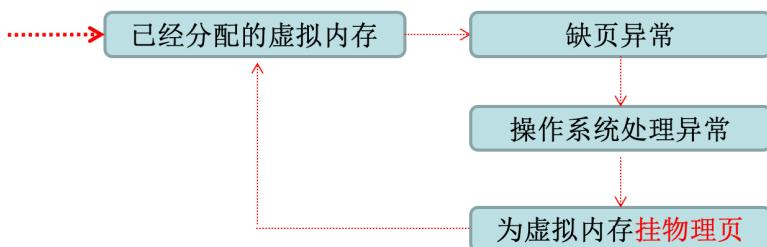
32位操作系统最多可以识别物理内存为64G，但是操作系统会进行限制，例如XP这个系统只能识别4G的物理内存（Windows Server 2003服务器版本可以识别4G以上）。

但是我们可以通过HOOK系统函数来突破XP操作系统的4GB限制。

19.4 物理页的使用

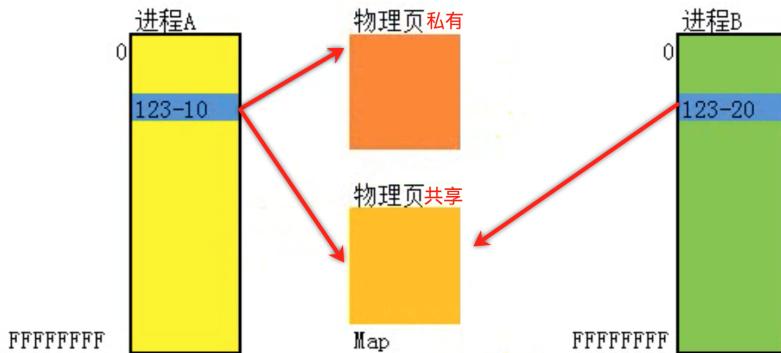
我们知道了进程在使用虚拟内存时，就会分配一块物理内存（物理页），但是有那么多程序，很快就会占满物理页，操作系统不会这样设计，而是会去看你的程序是否需要频繁的使用物理页，如果不是很频繁就会将你存储在物理页的内容放在pagefile.sys文件中，然后将这个物理页分配给其他需要的进程；

如果你的程序再次访问物理页的话，就会重新给你分配物理页，然后把数据从pagefile.sys文件中拿出来放到新的物理页中，这都是操作系统在操作的，写程序是感受不到这样的细节的。



20 私有内存的申请释放

物理内存分为两类，一个是私有内存（Private）一个是共享内存（Mapped），私有内存的意思是这块物理内存（物理页）只有你使用，而共享内存则是多个进程一起用。



20.1 申请内存的两种方式

1. 私有内存通过**VirtualAlloc/VirtualAllocEx函数**申请，这两个函数在底层实现是没有区别的，但是后者是可以在其他进程中申请内存。
2. 共享内存通过**CreateFileMapping函数**映射

20.2 内存申请与释放

申请内存的函数是**VirtualAlloc**，其语法格式如下：

1	LPVOID VirtualAlloc(
2	LPVOID lpAddress, // region to reserve or commit 要分配的内存区域的地址，没有特殊需求通常不指定
3	SIZE_T dwSize, // size of region 分配的大小，一个物理页大小是0x1000 (4KB)，看你需要申请多少个物理页就乘以多少
4	DWORD flAllocationType, // type of allocation 分配的类型，常用的是MEM_COMMIT (占用线性地址，也需要物理内存) 和MEM_RESERVE (占用线性地址，但不需要物理内存)
5	DWORD flProtect // type of access protection 该内存的初始保护属性
6);

第三、第四参数可以根据MSDN Library查看系统定义的：

#AllocationType

[in] Specifies the type of allocation. This parameter must contain one of the following values: MEM_COMMIT, MEM_RESET, or MEM_RESERVE. All other values can be used as indicated in the following table.

Value	Meaning
MEM_COMMIT	Allocates physical storage in memory or in the paging file on disk for the specified region of memory pages. The function initializes the memory to zero. An attempt to commit a memory page that is already committed does not cause the function to fail. This means that you can commit a range of pages without determining the current commitment state of each page. If a memory page is not yet reserved, setting this value causes the function to both reserve and commit the memory page.
MEM_PHYSICAL	Allocates physical memory with read-write access. This value is solely for use with Address Windowing Extensions (AWE) memory. This value must be used with MEM_RESERVE and no other values.
MEM_RESERVE	Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk. Other memory allocation functions such as malloc and LocalAlloc , cannot use a reserved range of memory until it is released. You can commit reserved memory pages in subsequent calls to the VirtualAlloc function.
MEM_RESET	Windows NT/2000/XP: Specifies that the data in the memory range specified by lpAddress and dwSize is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be deallocated. This value cannot be used with any other value. Using this value does not guarantee that the range operated on with MEM_RESET will contain zeroes. If you want the range to contain zeroes, decommit the memory and then recommit it. When you specify MEM_RESET, the VirtualAlloc function ignores the value of lpProtect. However, you must still set lpProtect to a valid protection value, such as PAGE_NOACCESS. VirtualAlloc returns an error if you use MEM_RESET and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.
MEM_TOP_DOWN	Windows NT/2000/XP: Allocates memory at the highest possible address.
MEM_WRITE_WATCH	Windows 98/Me: Causes the system to track pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE. To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the GetWriteWatch function. To reset the write-tracking state, call GetWriteWatch or ResetWriteWatch . The write-tracking feature remains enabled for the memory region until the region is freed.

#Protect

[in] Specifies the type of access protection. If the pages are being committed, you can specify any one of the following value, along with PAGE_GUARD and PAGE_NOCACHE as needed.

Value	Meaning
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_GUARD	Windows NT/2000/XP: Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages thus act as a one-shot access alarm. PAGE_GUARD is a page protection modifier. An application uses it with one of the other page protection modifiers, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. Windows 95/98/Me: To simulate this behavior, use PAGE_NOACCESS.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This value is a page protection modifier, and it is only valid when used with one of the page protections other than PAGE_NOACCESS.

如下代码则表示申请2个物理页，占用线性地址并分配物理内存，该内存可读写：

```
1 LPVOID pm = VirtualAlloc(NULL, 0x1000*2, MEM_COMMIT, PAGE_READWRITE);
```

那么内存申请好了我们不想要了，这时候就需要释放，释放函数为**VirtualFree**，其语法格式如下：

```

1  BOOL VirtualFree(
2      LPVOID lpAddress,    // address of region 内存区域的地址
3      SIZE_T dwSize,       // size of region 内存大小
4      DWORD dwFreeType     // operation type 如何释放，释放的类型，一共有两个类型：MEM_DECOMMIT（释放物理内存，但线性地址保留）、MEM_RELEASE（释放物理内存，释放线性地址，使用这个设置的时候内存大小就必须为0）
5  );

```

所以我们想要释放物理内存，释放线性地址就写如下代码：

```
1  VirtualFree(pm, 0, MEM_RELEASE);
```

20.3 堆与栈

之前我们学习过的**malloc**或者**new**申请内存，它们是申请的什么内存呢？其实通过它们申请的内存是假申请，因为它们是从已经申请好的内存中申请给自己用的，通过它们申请的内存称为堆内存，局部变量称为栈内存。

无论堆内存还是栈内存，都是操作系统启动时操作系统使用**VirtualAlloc函数**替我们申请好的。

所以堆、栈的本质就是私有内存，也就是通过**VirtualAlloc函数**申请的。

```

1  int main(int argc, char* argv[])
2  {
3      int x = 0x12345678; // 栈
4
5      int* y = (int*)malloc(sizeof(int)*128); // 堆
6
7      return 0;
8  }

```

21 共享内存的申请释放

21.1 共享内存

共享内存通过**CreateFileMapping**函数映射，该函数语法格式如下：

```

1 HANDLE CreateFileMapping( // 内核对象，这个对象可以为我们准备物理内存，还可以将文件映射到物理页
2     HANDLE hFile,           // handle to file 文件句柄，如果不想将文件映射到物理页，则不指定该参数
3     LPSECURITY_ATTRIBUTES lpAttributes, // security 安全属性，包含安全描述符
4     DWORD  flProtect,        // protection 保护模式，物理页的属性
5     DWORD  dwMaximumSizeHigh, // high-order DWORD of size 高32位，在32位计算机里通常设置为空
6     DWORD  dwMaximumSizeLow, // low-order DWORD of size 低32位，指定物理内存的大小
7     LPCTSTR  lpName         // object name 对象名字，公用时写，自己使用则可以不指定
8 );

```

该函数的作用就是为我们准备好物理内存（物理页），但是创建好了并不代表就可以使用了，我们还需要通过**MapViewOfFile**函数将物理页与线性地址进行映射，**MapViewOfFile**函数语法格式如下：

```

1 LPVOID MapViewOfFile(
2     HANDLE hFileMappingObject, // handle to file-mapping object file-mapping对象的句柄
3     DWORD  dwDesiredAccess,   // access mode 访问模式(虚拟内存的限制必须比物理地址更加严格)
4     DWORD  dwFileOffsetHigh, // high-order DWORD of offset 高32位，在32位计算机里通常设置为空
5     DWORD  dwFileOffsetLow,  // low-order DWORD of offset 低32位，指定从哪里开始映射
6     SIZE_T dwNumberOfBytesToMap // number of bytes to map 共享内存的大小，一般与物理页大小一致
7 );

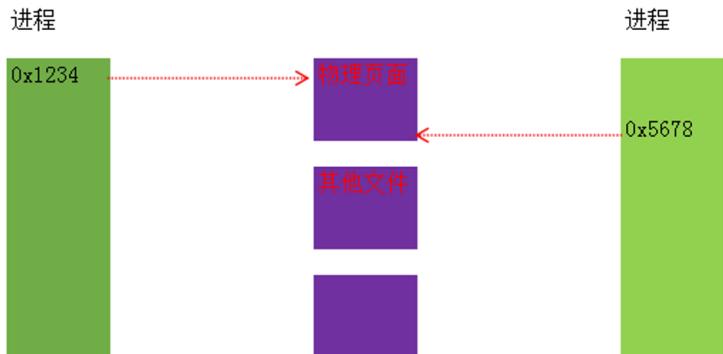
```

示例代码如下：

```

1 #include <windows.h>
2
3 #define MapFileName "共享内存"
4 #define BUF_SIZE 0x1000
5 HANDLE g_hMapFile;
6 LPTSTR g_lpBuff;
7
8 int main(int argc, char* argv[])
9 {
10     // 内核对象：准备好物理页，无效句柄值-1、物理页可读写、申请一个物理页
11     g_hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, -1, BUF_SIZE,
12     MapFileName);
13     // 将物理页与线性地址进行映射
14     g_lpBuff = (LPTSTR)MapViewOfFile(g_hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, BUF_SIZE);
15     // 向物理内存中存储
16     *(PDWORD)g_lpBuff = 0x12345678;
17
18     // 关闭映射，关闭映射则表示释放了线形地址，但是物理页还存在
19     UnmapViewOfFile(g_lpBuff);
20     // 关闭句柄，这样才能释放物理页，但需要等待物理页使用完毕才会真正的释放，这里只是告诉系统我们当前进程不使用该句
21     // 柄（物理页）罢了
22     CloseHandle(g_hMapFile);
23     return 0;
}

```



22 文件系统

文件系统是操作系统用于管理磁盘上文件的方法和数据结构；简单点说就是在磁盘上如何组织文件的方法。

	NTFS	FAT32
磁盘分区容量	2T(2048G)	32G
单个文件容量	4G以上	最大4G
EFS加密	支持	不支持
磁盘配额	支持	不支持

在Windows下有NTFS、FAT32这两种文件系统，我们可以通过查看本地磁盘的属性查看：



22.1 卷相关API

卷可以理解为就是我们的本地磁盘（逻辑驱动器），我们可以把一块2GB的内存条分为两个卷，卷里头的就是文件和目录。

22.1.1 获取卷（逻辑驱动器）

函数**GetLogicalDrives**用于获取当前计算机所有逻辑驱动器，语法格式为：

1	DWORD GetLogicalDrives(); // 返回值是一个DWORD，没有参数
---	---

如下图所示代码，我们最终获取到的就是一个十六进制的d，在MSDN Library中明确说明了这个返回值表示的结果：

二进制位标志着存在哪些驱动器，**位0为1则表示存在驱动器A，位1为1则表示存在驱动器B**，以此类推，这里我们获取的0xd二进制是1101，位1为1、位2为0、位3为1、位4为1，那么就表示我们存在驱动器A、C、D。

```

1 // 获取卷 (逻辑驱动器)
2 DWORD gLd = GetLogicalDrives();
3 printf("GetLogicalDrives: %x", gLd);

```

```

int main(int argc, char* argv[])
{
    // 获取卷 (逻辑驱动器)
    DWORD gLd = GetLogicalDrives();
    printf("GetLogicalDrives: %x", gLd);
    return 0;
}

```

The screenshot shows the assembly code for the `main` function. It includes comments in Chinese and calls to `GetLogicalDrives`. The output window shows the result of the printf call: `GetLogicalDrives: d`.

硬盘



本地磁盘 (C:)

有可移动存储的设备



3.5 软盘 (A:)



DVD 驱动器 (D:)

22.1.2 获取所有逻辑驱动器的字符串

函数**GetLogicalDriveStrings**用于获取所有逻辑驱动器的字符串，语法格式为：

```

1  DWORD GetLogicalDriveStrings(
2      DWORD nBufferLength, // size of buffer 输入类型, 要获取字符串的大小
3      LPTSTR lpBuffer     // drive strings buffer 输出类型, 将获取的字符串放到该参数中
4  );

```

如下图所示我可以获取所有逻辑驱动器的字符串，那么很清晰的可以看见逻辑驱动器的字符串就是盘符加上冒号和反斜杠：

```

1  // 获取一个逻辑驱动器的字符串
2  DWORD nBufferLength = 100;
3  char szOutBuffer[100];
4  GetLogicalDriveStrings(nBufferLength, szOutBuffer);

```

Name	Value
nBufferLength	100
szOutBuffer	0x0012FF18 "A:\\"
[0]	65 'A'
[1]	58 ':'
[2]	92 '\\'
[3]	0 ''
[4]	67 'C'
[5]	58 ':'
[6]	92 '\\'
[7]	0 ''
[8]	68 'D'
[9]	58 ':'
[10]	92 '\\'
[11]	0 ''
[12]	0 ''

22.1.3 获取卷（逻辑驱动器）的类型

函数**GetLogicalDriveStrings**用于获取卷的类型，语法格式为：

```

1  UINT GetDriveType(
2      LPCTSTR lpRootPathName // root directory 根目录, 这里我们可以使用驱动器字符串
3 );

```

如下图所示，我获取了逻辑驱动器C的类型：

```

1 // 获取卷的类型
2
3     type = GetDriveType(TEXT("C:\\\\"));
4
5     if (type == DRIVE_UNKNOWN) {
6         printf("无法确定驱动器的类型 \\n");
7     } else if (type == DRIVE_NO_ROOT_DIR) {
8         printf("根路径是无效的, 例如: 在该路径上没有安装任何卷 \\n");
9     } else if (type == DRIVE_REMOVABLE) {
10        printf("磁盘可以从驱动器中取出 \\n");
11    } else if (type == DRIVE_FIXED) {
12        printf("磁盘不能从驱动器中取出 \\n");
13    } else if (type == DRIVE_REMOTE) {
14        printf("该驱动器是一个远程 (网络) 驱动器 \\n");
15    } else if (type == DRIVE_CDROM) {
16        printf("该驱动器是一个CD-ROM驱动器 \\n");
17    } else if (type == DRIVE_RAMDISK) {
18        printf("该驱动器是一个RAM磁盘 \\n");
19    }

```

```

// 获取卷的类型
UINT type;
type = GetDriveType(TEXT("C:\\\\"));

if (type == DRIVE_UNKNOWN) {
    printf("无法确定驱动器的类型 \\n");
} else if (type == DRIVE_NO_ROOT_DIR) {
    printf("根路径是无效的, 例如: 在该路径上没有安装任何卷 \\n");
} else if (type == DRIVE_REMOVABLE) {
    printf("磁盘可以从驱动器中取出 \\n");
} else if (type == DRIVE_FIXED) {
    printf("磁盘不能从驱动器中取出 \\n");
} else if (type == DRIVE_REMOTE) {
    printf("该驱动器是一个远程 (网络) 驱动器 \\n");
} else if (type == DRIVE_CDROM) {
    printf("该驱动器是一个CD-ROM驱动器 \\n");
} else if (type == DRIVE_RAMDISK) {
    printf("该驱动器是一个RAM磁盘 \\n");
}

return 0;

```



22.1.4 获取卷的信息

函数**GetVolumeInformation**用于获取卷的信息，语法格式为：

```

1  BOOL GetVolumeInformation(
2      LPCTSTR lpRootPathName,           // root directory 输入类型, 驱动器字符串
3      LPTSTR lpVolumeNameBuffer,       // volume name buffer 输出类型, 返回卷名
4      DWORD nVolumeNameSize,          // length of name buffer 输入类型, 卷名长度
5      LPDWORD lpVolumeSerialNumber,    // volume serial number 输出类型, 卷宗序列号
6      LPDWORD lpMaximumComponentLength, // maximum file name length 输出类型, 指定文件系统支持的文件名组件的最大长度
7      LPDWORD lpFileSystemFlags,       // file system options 输出类型, 与指定文件系统相关的标志
8      LPTSTR lpFileSystemNameBuffer,   // file system name buffer 输出类型, 文件系统 (如FAT或NTFS) 名称
9      DWORD nFileSystemNameSize       // length of file system name buffer 输入类型, 文件系统名称的长度
10 );

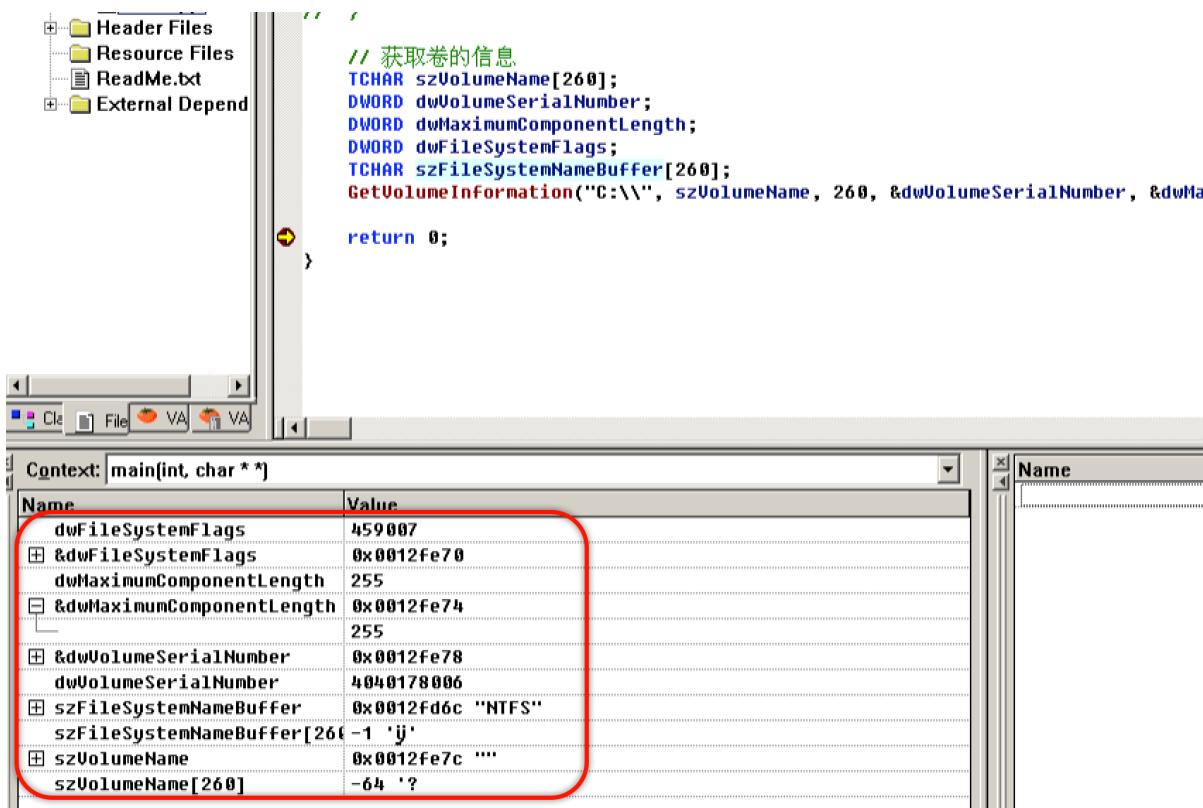
```

如下图所示，我获取了逻辑驱动器C的相关信息：

```

1 // 获取卷的信息
2 TCHAR szVolumeName[260];
3 DWORD dwVolumeSerialNumber;
4 DWORD dwMaximumComponentLength;
5 DWORD dwFileSystemFlags;
6 TCHAR szFileSystemNameBuffer[260];
7 GetVolumeInformation("C:\\", szVolumeName, 260, &dwVolumeSerialNumber, &dwMaximumComponentLength,
&dwFileSystemFlags, szFileSystemNameBuffer, 260);

```



22.2 目录相关API

22.2.1 创建目录

函数**CreateDirectory**用于创建目录，其语法格式如下：

```

1  BOOL CreateDirectory(
2      LPCTSTR lpPathName,           // directory name 目录名称, 需要指定完整路径包含盘符的
3      LPSECURITY_ATTRIBUTES lpSecurityAttributes // SD 安全属性, 包含安全描述符
4 );

```

在C盘下创建test目录：

```

1 // 创建目录, 如果不指定绝对路径, 则默认会在程序当前目录下
2 CreateDirectory(TEXT("C:\\\\test"), NULL);

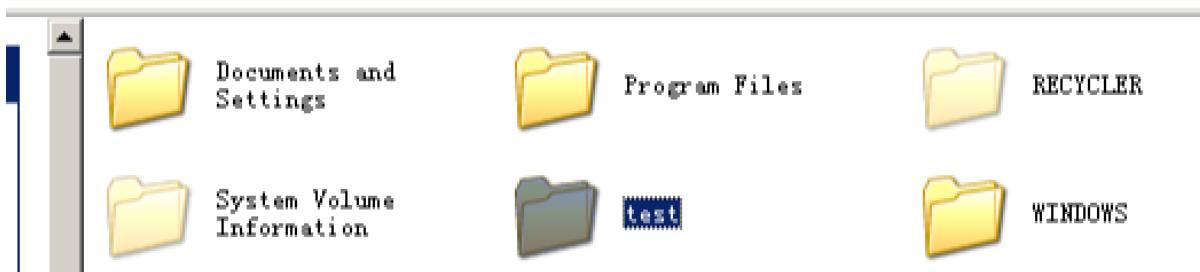
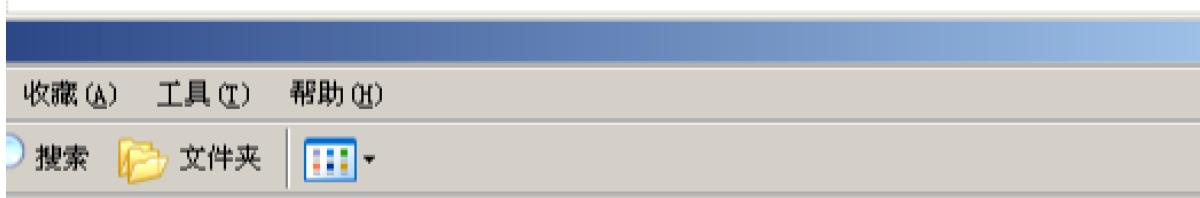
```

```

int main(int argc, char* argv[])
{
    // 创建目录
    CreateDirectory(TEXT("C:\\\\test"), NULL);

    return 0;
}

```



22.2.2 删除目录

函数**RemoveDirectory**用于删除目录，其语法格式如下：

```
1 BOOL RemoveDirectory(  
2     LPCTSTR lpPathName    // directory name 目录名称，需要指定完整路径包含盘符的  
3 );
```

删除C盘下的test目录：

```
1 // 删除目录  
2 RemoveDirectory(TEXT("C:\\\\test"));
```

22.2.3 修改目录名称（移动）

函数**MoveFile**用于修改目录名称（移动），其语法格式如下：

```
1 BOOL MoveFile(  
2     LPCTSTR lpExistingFileName, // file name 目录名  
3     LPCTSTR lpNewFileName      // new file name 新目录名  
4 );
```

将C盘下的test文件夹重命名为test1，也可以理解为以新的名称移动到新的目录下：

```
1 // 修改目录名称（移动）  
2 MoveFile(TEXT("C:\\\\test"), TEXT("C:\\\\test1"));
```

```

int main(int argc, char* argv[])
{
    // 创建目录
    CreateDirectory(TEXT("C:\\\\test"), NULL);

    // 删除目录
    // RemoveDirectory(TEXT("C:\\\\test"));

    // 修改目录名称(移动)
    MoveFile(TEXT("C:\\\\test"), TEXT("C:\\\\test1"));
    return 0;
}

```



22.2.4 获取程序当前目录

函数**GetCurrentDirectory**用于获取程序当前目录，其语法格式如下：

```

1  DWORD GetCurrentDirectory(
2      DWORD nBufferLength, // size of directory buffer 输入类型, 获取当前目录名的大小
3      LPTSTR lpBuffer     // directory buffer 输出类型, 当前目录名称
4 );

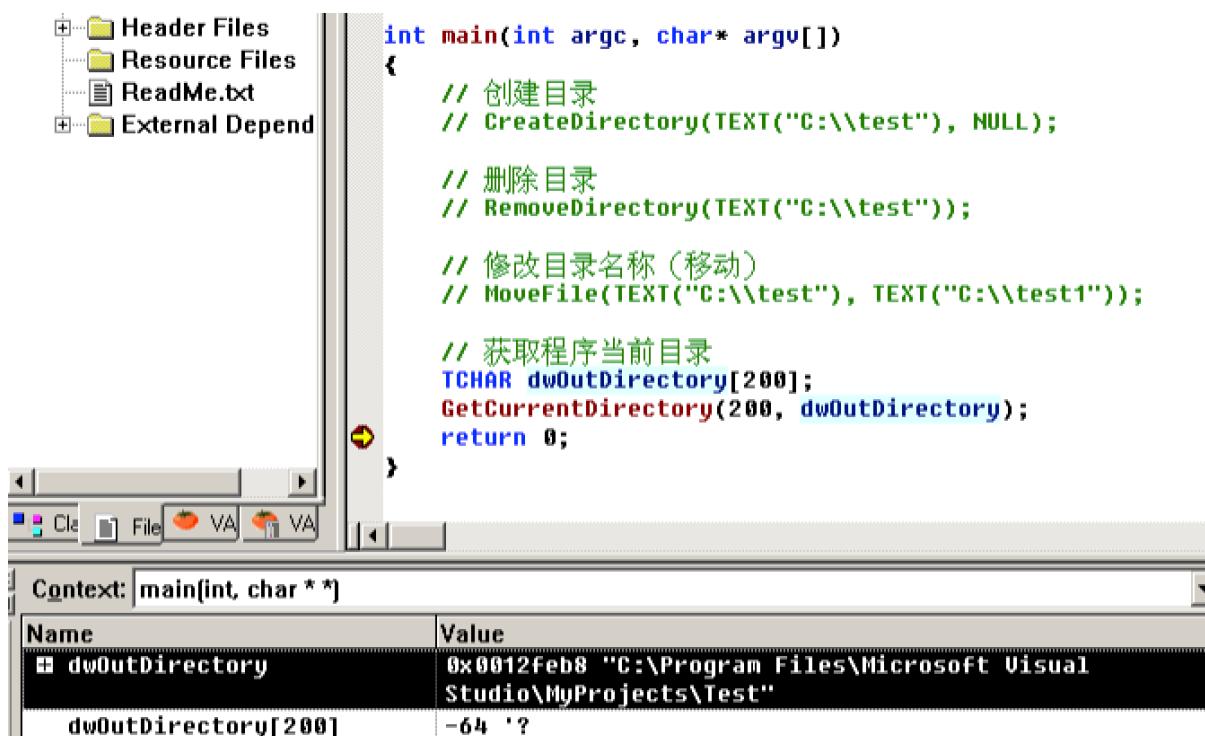
```

示例代码：

```

1 // 获取程序当前目录
2 TCHAR dwOutDirectory[200];
3 GetCurrentDirectory(200, dwOutDirectory);

```



22.2.5 设置程序当前目录

函数**SetCurrentDirectory**用于设置程序当前目录，其语法格式如下：

```

1  BOOL SetCurrentDirectory(
2      LPCTSTR lpPathName // new directory name 新的目录名称
3 );

```

示例代码：

```

1 // 设置程序当前目录
2 SetCurrentDirectory(TEXT("C:\\\\test"));

```

22.3 文件相关API

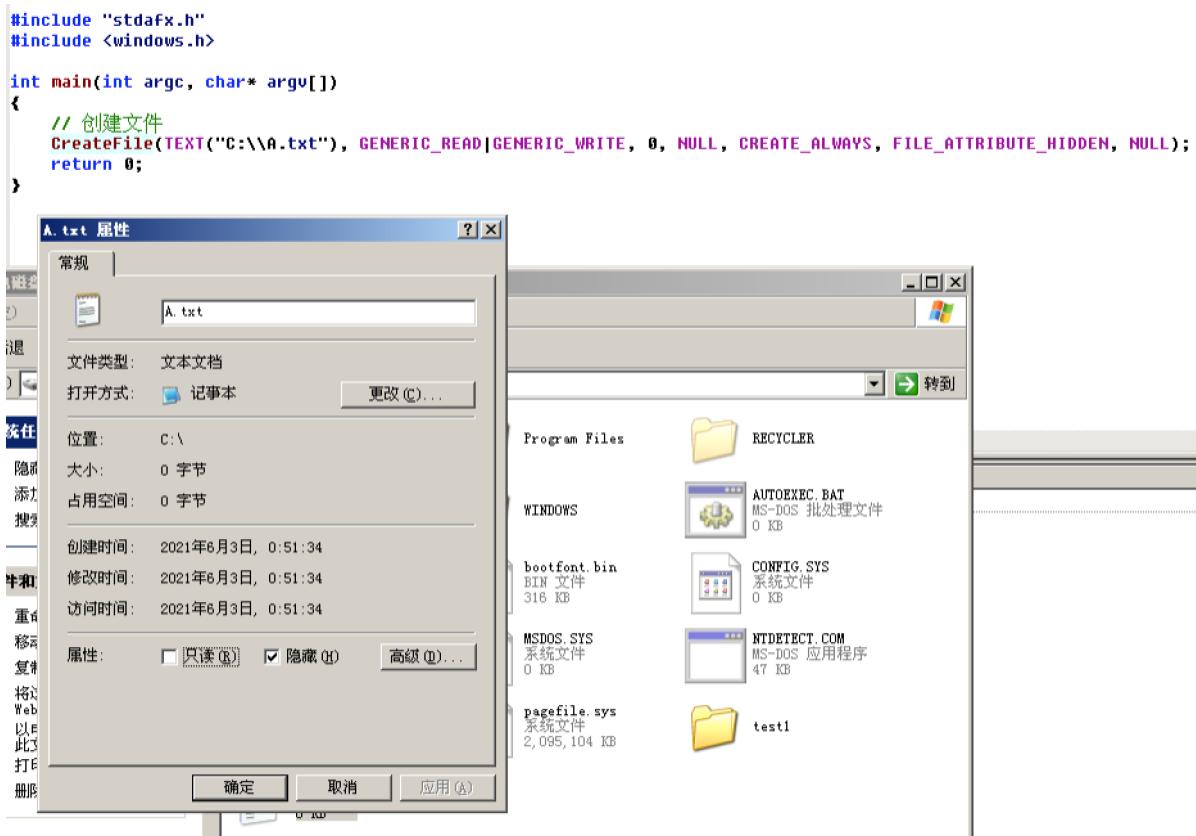
22.3.1 创建文件

函数**CreateFile**用于创建文件，其语法格式如下：

1	HANDLE CreateFile(
2	LPCTSTR lpFileName,
3	DWORD dwDesiredAccess,
4	DWORD dwShareMode,
	// file name 文件名 // access mode 访问模式 // share mode 共享模式，如果为0则是排他性，就是目前在使用时 其他人是无法使用的
5	LPSECURITY_ATTRIBUTES lpSecurityAttributes, // SD 安全属性，包含安全描述符
6	DWORD dwCreationDisposition,
7	DWORD dwFlagsAndAttributes,
8	HANDLE hTemplateFile
9);

以可读可写方式不管有没有，有就覆盖没有就新建的方式创建一个隐藏文件：

1	// 创建文件
2	CreateFile(TEXT("C:\\A.txt"), GENERIC_READ GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_HIDDEN, NULL);



22.3.2 关闭文件

函数**CloseHandle**用于关闭文件，其语法格式如下：

```

1  BOOL CloseHandle(
2      HANDLE hObject // handle to object 文件句柄
3 );

```

```

int main(int argc, char* argv[])
{
    // 创建文件
    HANDLE hFile = CreateFile(TEXT("C:\\\\A.txt"), GENERIC_READ|G
    // 关闭文件
    CloseHandle(hFile);
    return 0;
}

```

22.3.3 获取文件大小

函数**GetFileSize**用于获取文件大小，其语法格式如下：

```

1  DWORD GetFileSize(
2      HANDLE hFile,           // handle to file 输入类型, 文件句柄
3      LPDWORD lpFileSizeHigh // high-order word of file size, 输出类型, 高32位的文件大小, 这个没有用, 长度一
般在低32位中, 也就是当前函数的返回值
4  );

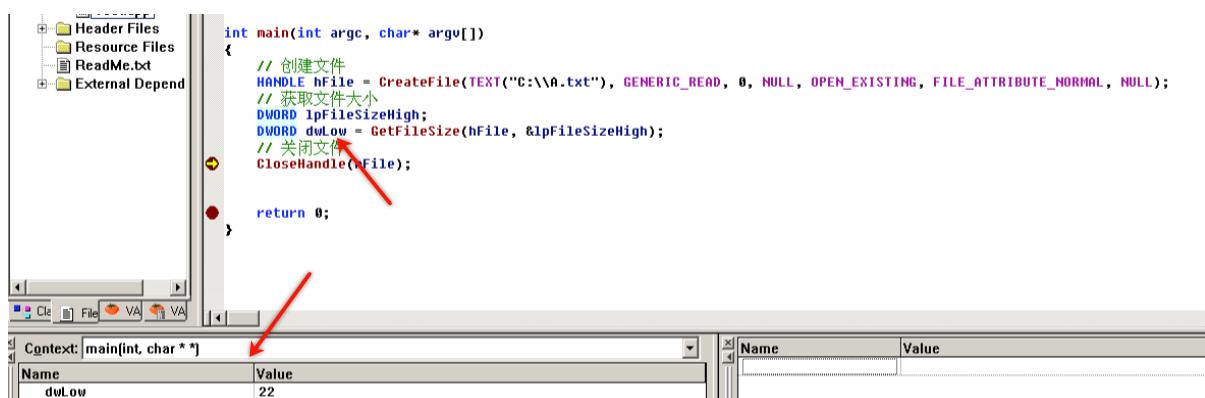
```

示例代码如下：

```

1 // 创建文件
2 HANDLE hFile = CreateFile(TEXT("C:\\\\A.txt"), GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
3 // 获取文件大小, 单位是字节
4 DWORD lpFileSizeHigh;
5 DWORD dwLow = GetFileSize(hFile, &lpFileSizeHigh);
6 // 关闭文件
7 CloseHandle(hFile);

```



22.3.4 获取文件的属性和信息

函数**GetFileAttributes**、**GetFileAttributesEx**用于获取文件的属性和信息，其语法格式如下：

```

1  DWORD GetFileAttributes( // 这个仅能获取属性
2      LPCTSTR lpFileName    // name of file or directory 文件或目录的名称
3  );
4
5  BOOL GetFileAttributesEx( // 这个可以获取属性、信息
6      LPCTSTR lpFileName,          // file or directory name 输入类型, 文件或目录的名称
7      GET_FILEEX_INFO_LEVELS fInfoLevelId, // attribute class 输入类型, 这个只有GetFileExInfoStandard一
个值
8      LPVOID lpFileInformation     // attribute information 输出类型, 文件属性和信息的结果
9  );

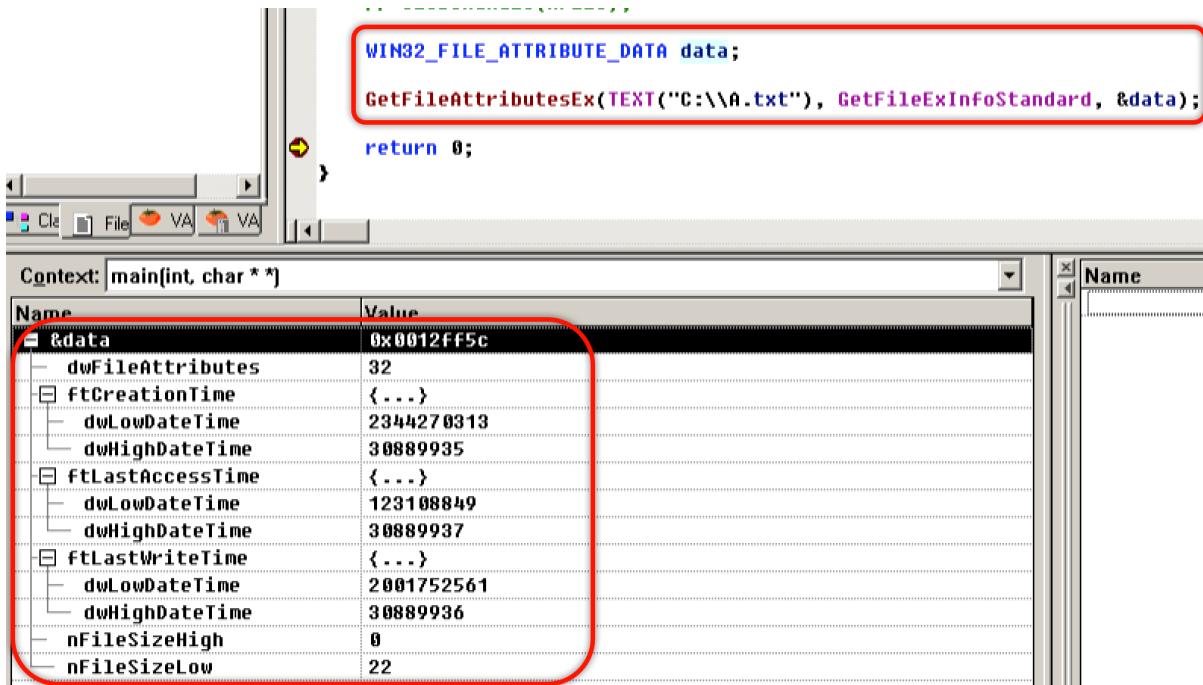
```

示例代码如下：

```

1 WIN32_FILE_ATTRIBUTE_DATA data; // 定义一个结构体
2
3 GetFileAttributesEx(TEXT("C:\\\\A.txt"), GetFileExInfoStandard, &data); // 传递结构体指针

```



22.3.5 读/写/拷贝/删除文件

函数**ReadFile**、**WriteFile**、**CopyFile**、**DeleteFile**用于读/写/拷贝/删除文件，其语法格式如下：

```

1  BOOL ReadFile( // 读取文件
2      HANDLE hFile,           // handle to file 文件句柄
3      LPVOID lpBuffer,        // data buffer 输出类型，数据放哪
4      DWORD nNumberOfBytesToRead, // number of bytes to read 要读多少字节
5      LPDWORD lpNumberOfBytesRead, // number of bytes read 真正读多少字节
6      LPOVERLAPPED lpOverlapped // overlapped buffer
7  );
8
9  BOOL WriteFile( // 写入文件
10     HANDLE hFile,          // handle to file 文件句柄
11     LPCVOID lpBuffer,       // data buffer 要写入的数据在哪
12     DWORD nNumberOfBytesToWrite, // number of bytes to write 要写多少字节
13     LPDWORD lpNumberOfBytesWritten, // number of bytes written 真正写多少字节
14     LPOVERLAPPED lpOverlapped // overlapped buffer
15  );
16
17  BOOL CopyFile( // 拷贝文件
18      LPCTSTR lpExistingFileName, // name of an existing file 已经存在的文件
19      LPCTSTR lpNewFileName,    // name of new file 复制的文件
20      BOOL bFailIfExists      // operation if file exists FALSE则复制位置的文件已经存在就覆盖，TRUE反之
21  );
22
23  BOOL DeleteFile( // 删除文件
24      LPCTSTR lpFileName     // file name 文件名
25  );

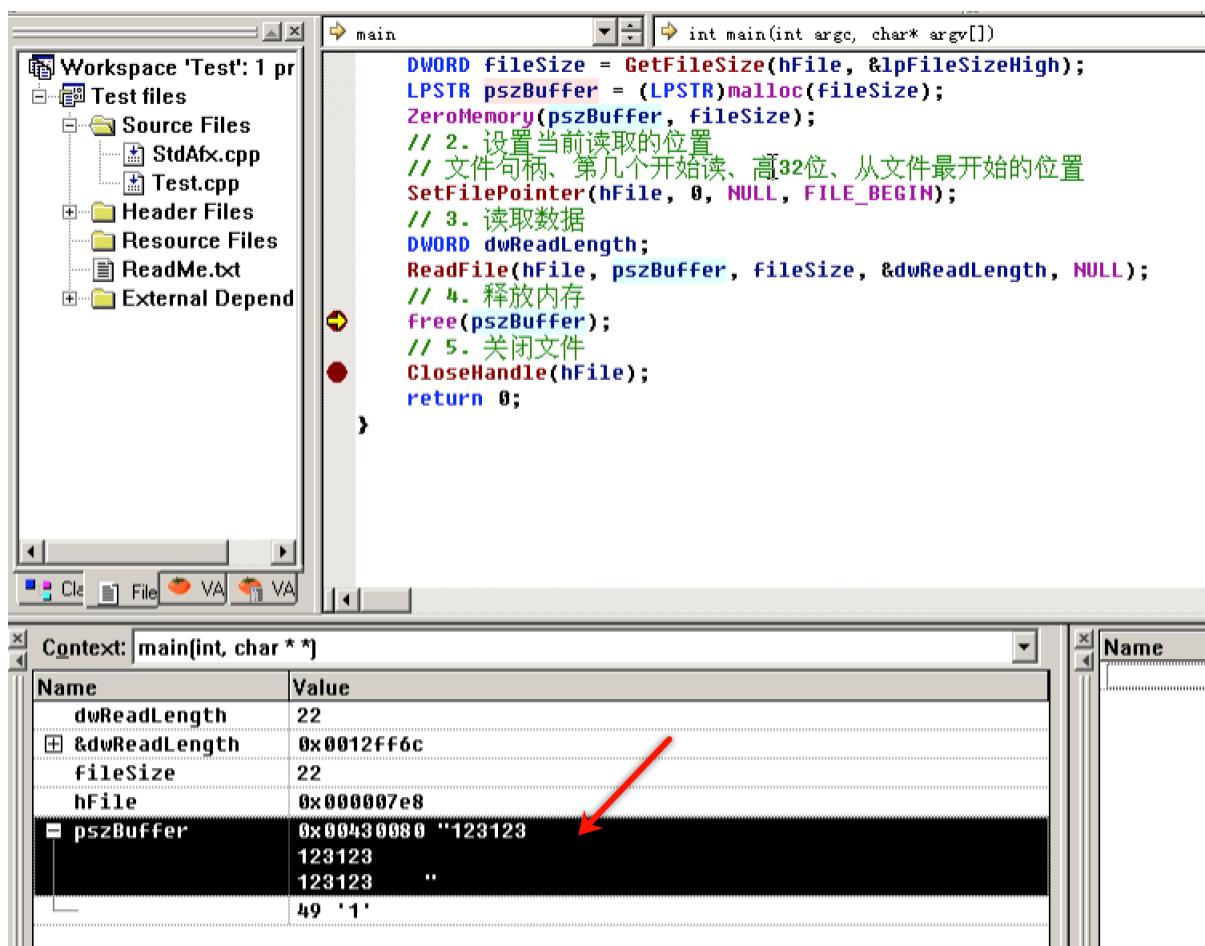
```

示例代码如下（举一反三）：

```

1 #include <windows.h>
2 #include <stdlib.h>
3
4 int main(int argc, char* argv[])
5 {
6     HANDLE hFile = CreateFile(TEXT("C:\\A.txt"), GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
7     // 读取文件
8     // 1. 分配空间
9     DWORD lpFileSizeHigh;
10    DWORD fileSize = GetFileSize(hFile, &lpFileSizeHigh);
11    LPSTR pszBuffer = (LPSTR)malloc(fileSize);
12    ZeroMemory(pszBuffer, fileSize);
13    // 2. 设置当前读取的位置
14    // 文件句柄、第几个开始读、高32位、从文件最开始的位置
15    SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
16    // 3. 读取数据
17    DWORD dwReadLength;
18    ReadFile(hFile, pszBuffer, fileSize, &dwReadLength, NULL);
19    // 4. 释放内存
20    free(pszBuffer);
21    // 5. 关闭文件
22    CloseHandle(hFile);
23    return 0;
24 }

```



22.3.6 查找文件

函数**FindFirstFile**、**FindNextFile**用于查找文件，其语法格式如下：

```

1  HANDLE FindFirstFile(
2      LPCTSTR lpFileName,           // file name 输入类型, 文件名
3      LPWIN32_FIND_DATA lpFindFileData // data buffer 输出类型, WIN32_FIND_DATA结构体指针, 找到的文件数据
4  );
5
6  BOOL FindNextFile(
7      HANDLE hFindFile,            // search handle 输入类型, 搜索句柄
8      LPWIN32_FIND_DATA lpFindFileData // data buffer 输出类型, WIN32_FIND_DATA结构体指针, 存放找到的文件数
9  );

```

示例代码如下：

```

1 WIN32_FIND_DATA firstFile;
2 WIN32_FIND_DATA nextFile;
3 // 在C盘下搜索.txt后缀的文件
4 HANDLE hFile = FindFirstFile(TEXT("C:\\*.txt"), &firstFile);
5 printf("第一个文件名: %s 文件大小: %d\\n", firstFile.cFileName, firstFile.nFileSizeLow);
6 if (hFile != INVALID_HANDLE_VALUE) {
7     // 有搜索到, 就使用FindNextFile寻找下一个文件, FindNextFile函数返回为真则表示搜索到了
8     while (FindNextFile(hFile, &nextFile)) {
9         printf("文件名: %s 文件大小: %d\\n", nextFile.cFileName, nextFile.nFileSizeLow);
10    }
11 }

```

```

int main(int argc, char* argv[])
{
    WIN32_FIND_DATA firstFile;
    WIN32_FIND_DATA nextFile;
    // 在C盘下搜索.txt后缀的文件
    HANDLE hFile = FindFirstFile(TEXT("C:\\*.txt"), &firstFile);
    printf("第一个文件名: %s 文件大小: %d\\n", firstFile.cFileName, firstFile.nFileSizeLow);
    if (hFile != INVALID_HANDLE_VALUE) {
        // 有搜索到, 就使用FindNextFile寻找下一个文件, FindNextFile函数返回为真则表示搜索到了
        while (FindNextFile(hFile, &nextFile)) {
            printf("文件名: %s 文件大小: %d\\n", nextFile.cFileName, nextFile.nFileSizeLow);
        }
    }
    return 0;
}

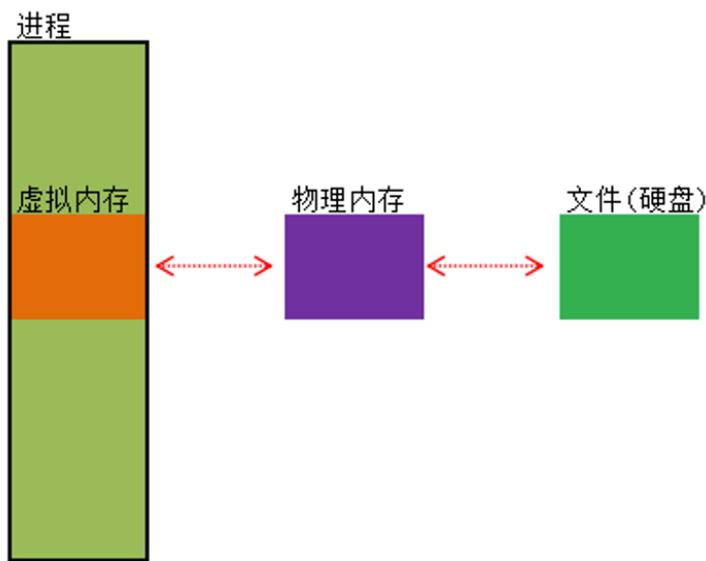
```



23 内存映射文件

23.1 什么是内存映射文件

内存映射文件就如下图，将硬盘某个文件映射到物理页上，然后再将物理页映射到虚拟内存中。



优点：

1. 访问文件就像访问内存一样简单，想读就读，想怎么样就怎么样，不用那么繁杂；
2. 当文件过大时，使用内存映射文件的方式，性能相对于普通I/O的访问要好很多。

23.2 内存映射文件读写

之前我们学习过用CreateFileMapping函数来创建共享内存，这个函数同样也可以将文件映射到物理页，只不过在这之前我们需要传递一个文件句柄。

如下代码我们写了一个读取文件最开始第一个字节的值：

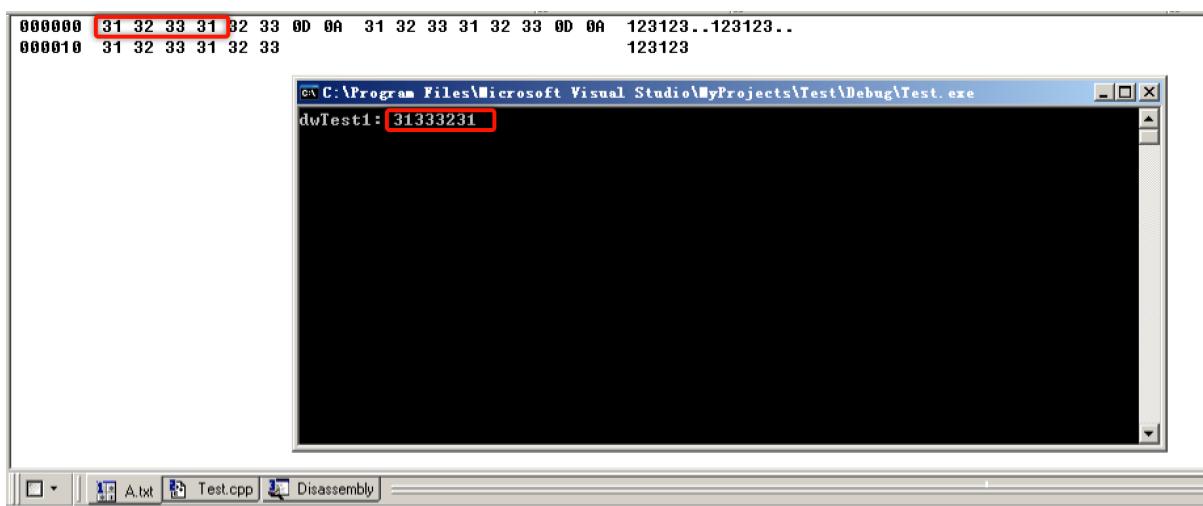
```

1  DWORD MappingFile(LPSTR lpcFile) {
2      HANDLE hFile;
3      HANDLE hMapFile;
4      LPVOID lpAddr;
5
6      // 1. 创建文件 (获取文件句柄)
7      hFile = CreateFile(lpcFile, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
8      FILE_ATTRIBUTE_NORMAL, NULL);
9
10     // 判断CreateFile是否执行成功
11     if(hFile == NULL) {
12         printf("CreateFile failed: %d \n", GetLastError());
13         return 0;
14     }
15
16     // 2. 创建FileMapping对象
17     hMapFile = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
18
19     // 判断CreateFileMapping是否执行成功
20     if(hMapFile == NULL) {
21         printf("CreateFileMapping failed: %d \n", GetLastError());
22         return 0;
23     }
24
25     // 3. 物理页映射到虚拟内存
26     lpAddr = MapViewOfFile(hMapFile, FILE_MAP_COPY, 0, 0, 0);
27
28     // 4. 读取文件
29     DWORD dwTest1 = *(LPDWORD)lpAddr; // 读取最开始的4字节
30     printf("dwTest1: %x \n", dwTest1);
31     // 5. 写文件
32     // *(LPDWORD)lpAddr = 0x12345678;
33
34     // 6. 关闭资源
35     UnmapViewOfFile(lpAddr);
36     CloseHandle(hFile);
37     CloseHandle(hMapFile);
38     return 0;
}

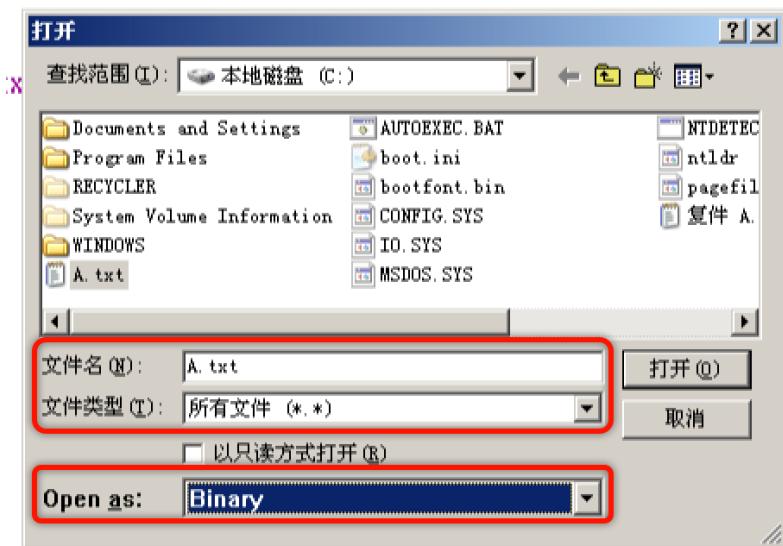
```

调用函数运行之后成功输出，并获取到对应内容：

1	MappingFile(TEXT("C:\\A.txt"));
---	---------------------------------



小技巧 → 在VC6中想要以HEX的形式查看某个文件的话可以在打开文件的时候这样设置：



举一反三，写文件也很简单，但是需要注意的是写文件不是立即生效的，而是先将写入的存放到缓存中，只有等到你释放资源了才会把缓存里的值真正的写入到文件。

如果你希望修改可以立即生效，我们可以通过**FlushViewOfFile**函数来强制更新缓存，其语法格式如下：

```

1  BOOL FlushViewOfFile(
2      LPCVOID lpBaseAddress,           // starting address 你要刷新的地址
3      SIZE_T dwNumberOfBytesToFlush // number of bytes in range 要刷新的大小 (字节)
4 );

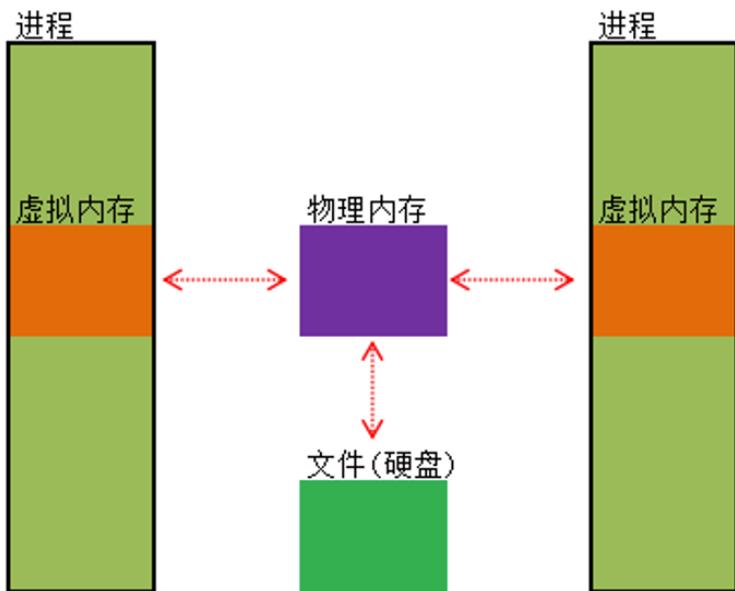
```

示例代码：

```
1  FlushViewOfFile(((LPDWORD)lpAddr), 4);
```

23.3 内存映射文件之共享

内存映射文件可以让2个进程同时共享一个文件：



其实本质很简单，我们只需要在创建FileMapping对象时候给其一个对象名称即可。

```

#define MAPPINGNAME "Share File"

DWORD MappingFile(LPSTR lpcFile) {
    HANDLE hFile;
    HANDLE hMapFile;
    LPVOID lpAddr;

    // 1. 创建文件 (获取文件句柄)
    hFile = CreateFile(lpcFile, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    // 判断CreateFile是否执行成功
    if(hFile == NULL) {
        printf("CreateFile Failed: %d \n", GetLastError());
        return 0;
    }

    // 2. 创建FileMapping对象
    hMapFile = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, MAPPINGNAME);

```

现在我们来A进程写入，B进程读取看看到底能不能跨进程恭喜，写入代码：

```

1 #define MAPPINGNAME "Share File"
2
3 DWORD MappingFile(LPSTR lpcFile) {
4     HANDLE hFile;
5     HANDLE hMapFile;
6     LPVOID lpAddr;
7
8     // 1. 创建文件 (获取文件句柄)
9     hFile = CreateFile(lpcFile, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
10
11    // 判断CreateFile是否执行成功
12    if(hFile == NULL) {
13        printf("CreateFile failed: %d \n", GetLastError());
14        return 0;
15    }
16
17    // 2. 创建FileMapping对象
18    hMapFile = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, MAPPINGNAME);
19
20    // 判断CreateFileMapping是否执行成功
21    if(hMapFile == NULL) {
22        printf("CreateFileMapping failed: %d \n", GetLastError());
23        return 0;
24    }
25
26    // 3. 物理页映射到虚拟内存
27    lpAddr = MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
28
29    // 4. 读取文件
30    // DWORD dwTest1 = *(LPDWORD)lpAddr; // 读取最开始的4字节
31    // printf("dwTest1: %x \n", dwTest1);
32    // 5. 写文件
33    *(LPDWORD)lpAddr = 0x41414142;
34    FlushViewOfFile(((LPDWORD)lpAddr), 4);
35    printf("Process A Write");
36    getchar();
37    // 6. 关闭资源
38    UnmapViewOfFile(lpAddr);
39    CloseHandle(hFile);
40    CloseHandle(hMapFile);
41    return 0;
42}

```

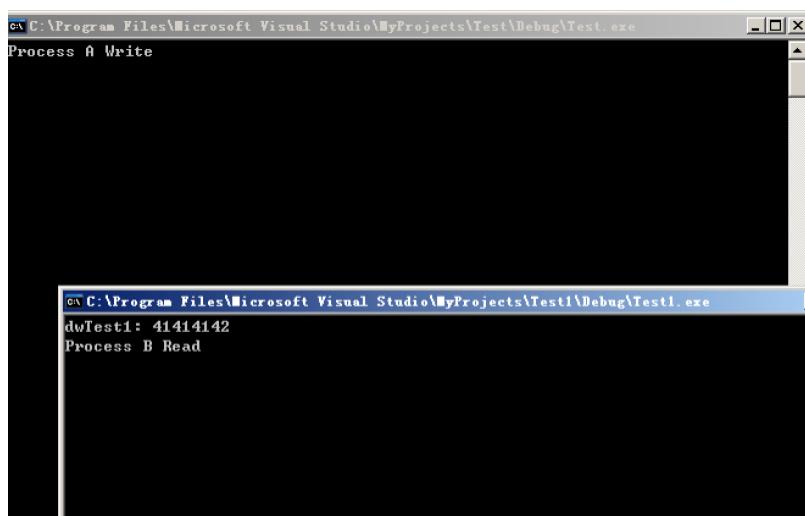
读取代码：

```

1 #define MAPPINGNAME "Share File"
2
3 DWORD MappingFile(LPSTR lpcFile) {
4     HANDLE hMapFile;
5     LPVOID lpAddr;
6
7     // 1. 打开FileMapping对象
8     /*
9      OpenFileMapping 函数语法格式：
10     HANDLE OpenFileMapping(
11         DWORD dwDesiredAccess, // access mode 访问模式
12         BOOL bInheritHandle, // inherit flag 继承标识，为真则表示这个可以被新进程继承，为假反之
13         LPCTSTR lpName // object name 对象名称
14     );
15     */
16     hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, MAPPINGNAME);
17
18     // 2. 物理页映射到虚拟内存
19     lpAddr = MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
20
21     // 3. 读取文件
22     DWORD dwTest1 = *(LPDWORD)lpAddr; // 读取最开始的4字节
23     printf("dwTest1: %x \n", dwTest1);
24     // 4. 写文件
25     // *(LPDWORD)lpAddr = 0x41414142;
26     printf("Process B Read");
27     getchar();
28     // 5. 关闭资源
29     UnmapViewOfFile(lpAddr);
30     CloseHandle(hMapFile);
31     return 0;
32 }

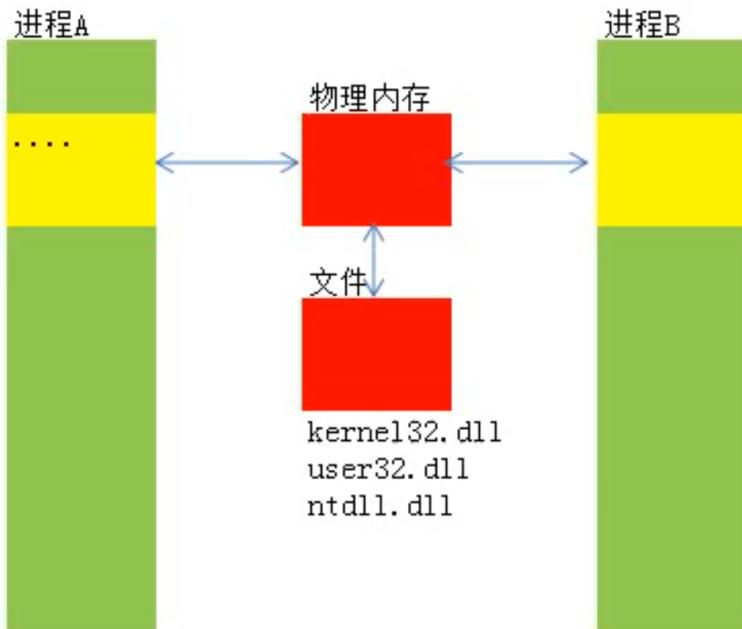
```

分别使用getchar函数挂住了运行，A进程写入0x41414142，B进程也成功读取到了这个值：



23.4 内存映射文件之写拷贝

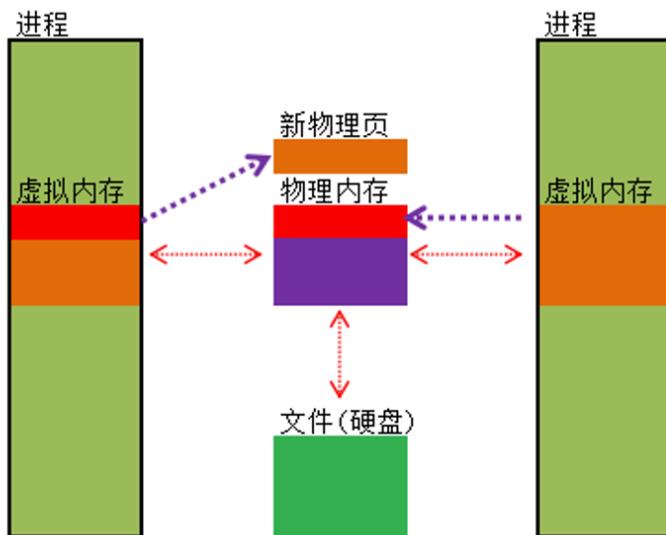
我们知道了内存映射文件可以共享，但是这样也有弊端，如下图所示，实际上我们程序调用的user32.dll这类dll文件，也是通过这种方式进行调用的，如果我们进场A修改了某个DLL，就会导致进程B出问题。



为了解决这种隐患，我们可以使用写拷贝的方式来处理。

写拷贝的实现就是MapViewOfFile函数中的第二个参数值为FILE_MAP_COPY，它的意思表示当你在写的时候进行拷贝。

当我们设置为该熟悉时，则多进程之间读取的是同一块物理页，但是当进程A写入的时候则会将这份物理页拷贝为一个新的物理页进行写：



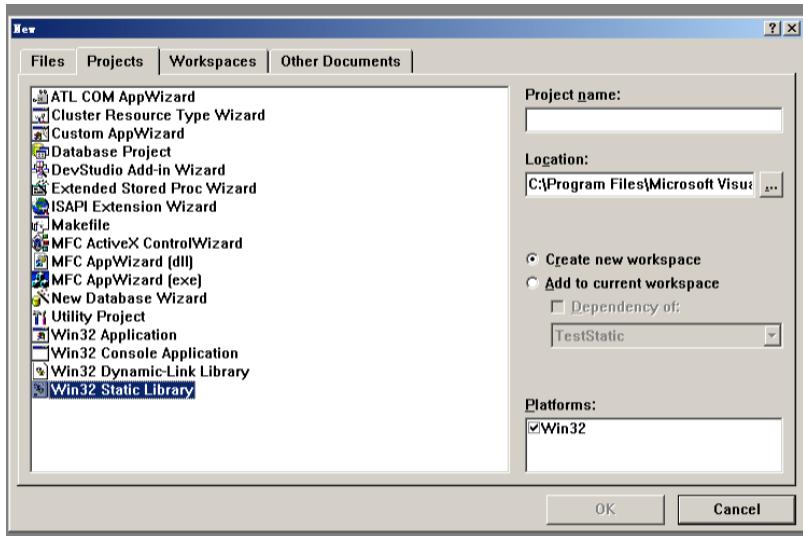
写拷贝属性时候，写入时**并不会影响**原本的文件内容。

24 静态链接库

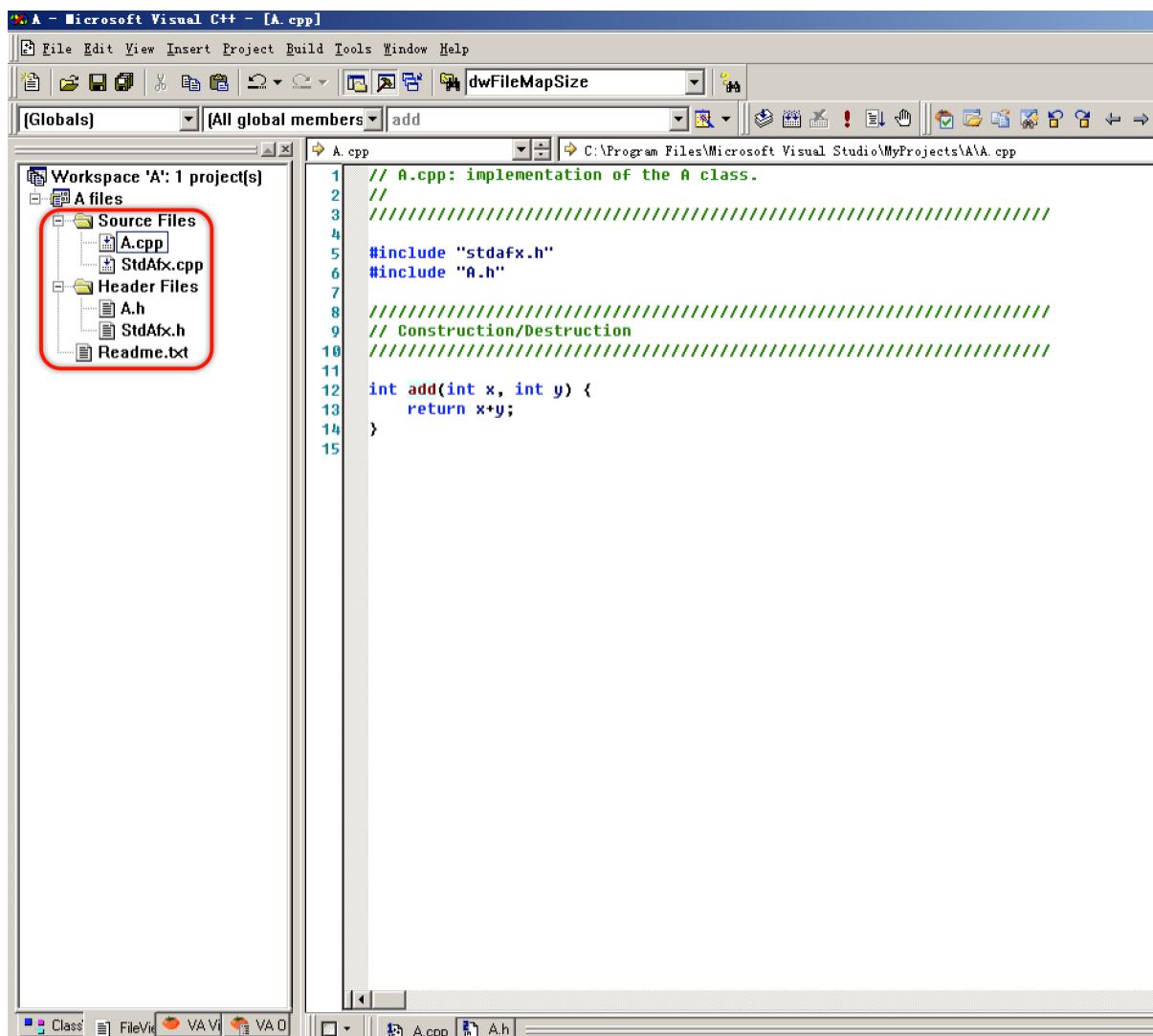
软件随着更迭会越来越复杂，包含的功能会很多，一个大型软件参与开发的人会非常多，因为不可能一个把所有事情干了，这样就会把软件分为多个模块，每个模块有对应的人去写，静态链接库就是软件模块化的一种解决方案。

24.1 编写静态链接库文件

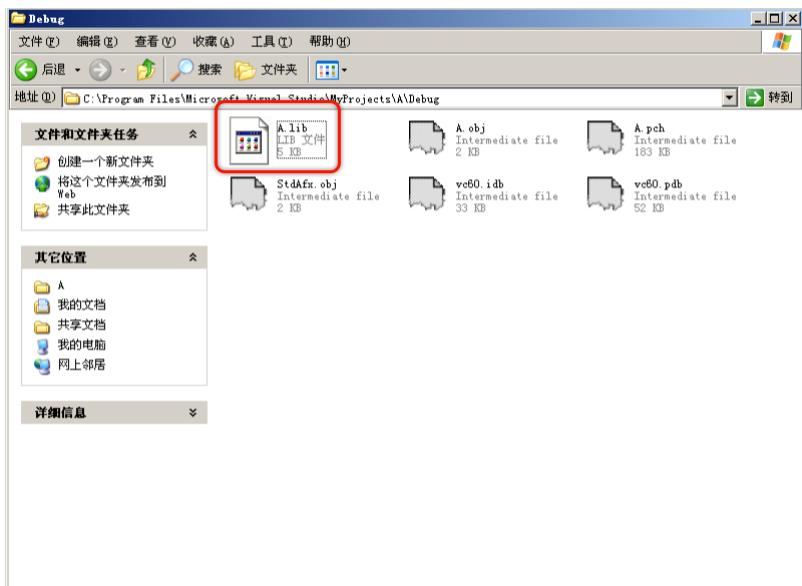
在VC6中创建静态链接库项目：



创建项目A，然后新建A.cpp和A.h，在A.h中声明一个add方法，在A.cpp中实现该方法：



编译一下，在项目目录的Debug目录下会有一个A.lib文件，这就是我们的静态链接库：



如果我们要给别人用的话那就需要A.lib + A.h这个头文件一起给别人。

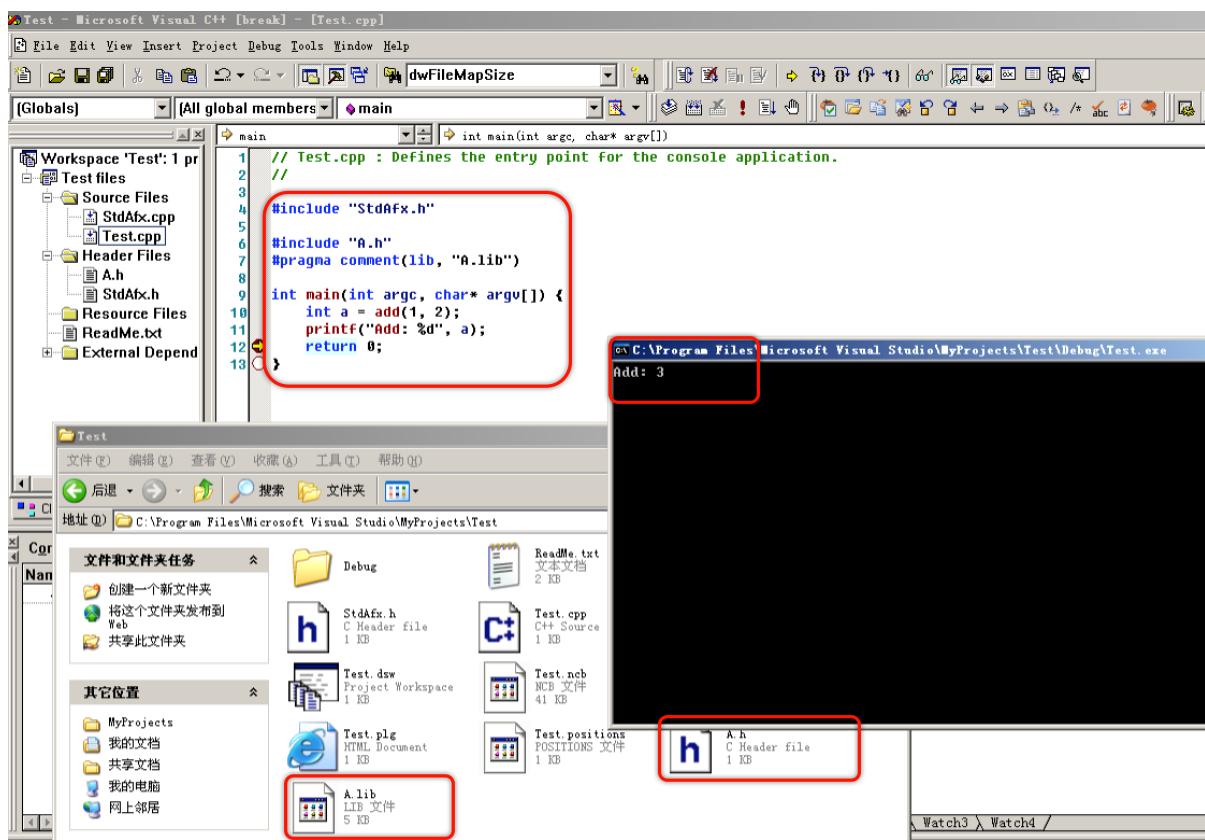
24.2 静态链接库的使用

静态链接库的使用有两种方法：

24.2.1 项目根目录

第一种方法：将生成的.h与.lib文件复制到项目根目录，然后在代码中引用：

	<pre> 1 #include "xxxx.h" 2 #pragma comment(lib, "xxxx.lib") </pre>
--	---



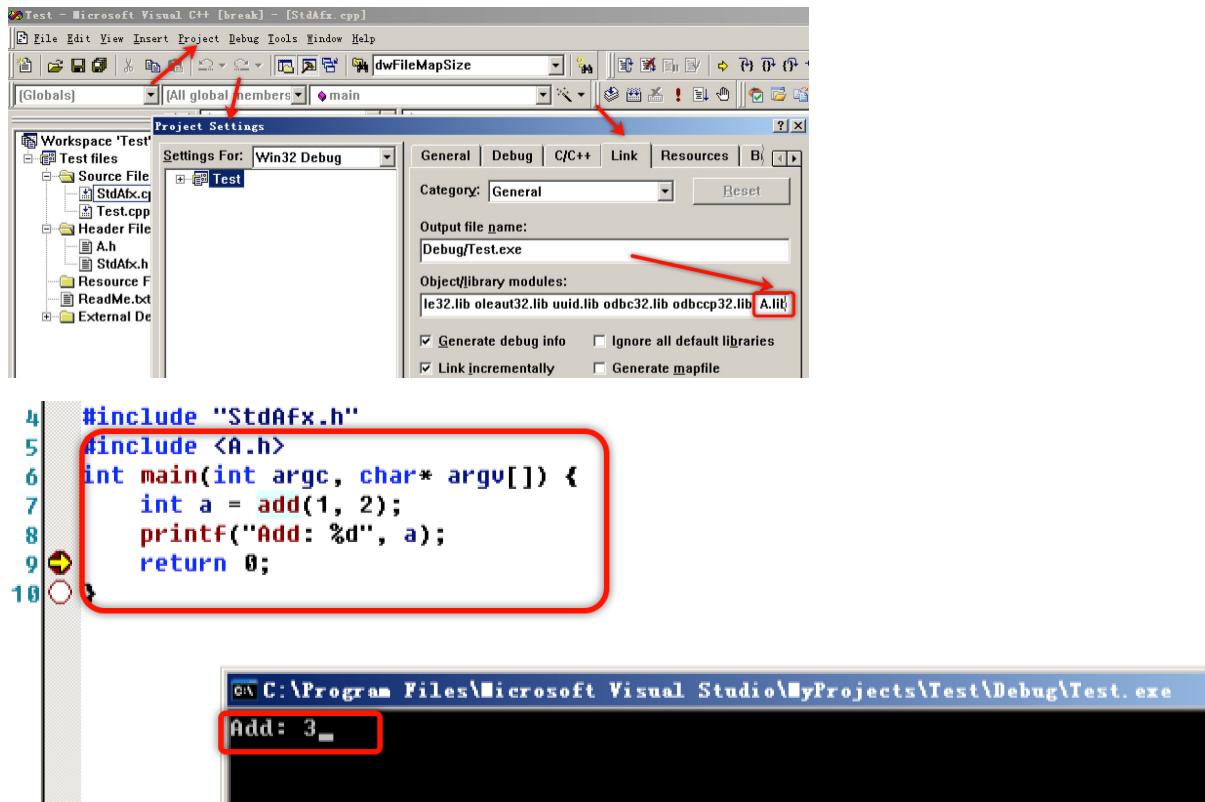
24.2.2 VC6安装目录

第二种方法：将xxxx.h与xxxx.lib文件复制到VC6安装目录，与库文件放在一起，然后在工程->设置->连接->对象/库模块中添加xxxx.lib，最后就可以像使用C语言库一样使用它了

- 头文件路径：C:\Program Files\Microsoft Visual Studio\VC98\Include
- 静态链接库路径：C:\Program Files\Microsoft Visual Studio\VC98\Lib



在编辑框中写入A.lib，多个lib文件以空格隔开：



24.3 静态链接库的缺点

第一个：使用静态链接生成的可执行文件体积较大，例如我们从汇编层面来看，是根本无法区分哪个是静态库中的代码的：

```

7:         int a = add(1, 2);
00401028    push      2
0040102A    push      1
0040102C    call      add (00401070)
00401031    add       esp,8
00401034    mov       dword ptr [ebp-4],eax
8:         printf("Add: %d", a);
00401037    mov       eax,dword ptr [ebp-4]
0040103A    push      eax
0040103B    push      offset string "Add: %d" (0042001c)
00401040    call      printf (004010a0)
00401045    add       esp,8
9:         return 0;

```

同时我们也了解了静态链接库的本质，那就是把你想要调用的接口（函数）直接写入到你的程序中。

第二个：包含相同的公共代码，造成浪费，假设我们在多个项目中使用同一个静态链接库，其实也就表示相同的代码复制多份。

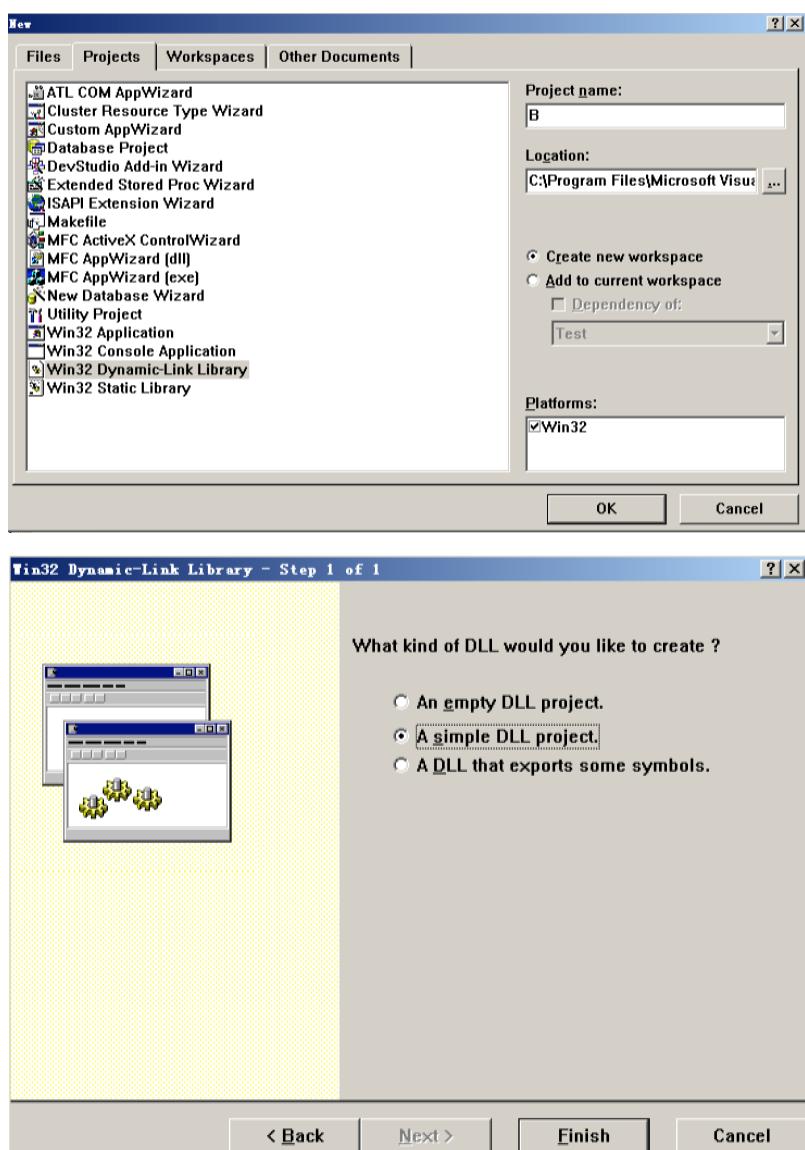
25 动态链接库

动态链接库弥补了静态链接库的两个缺点，动态链接库（Dynamic Link Library，缩写为DLL），是微软公司在微软Windows操作系统中对共享函数库概念的一种实现方式，这些库函数的文件扩展名称为：.dll、.ocx（包含ActiveX控制的库）。

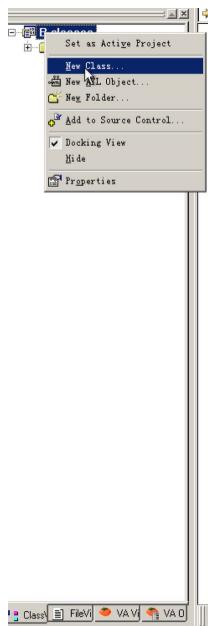
25.1 创建动态链接库

25.1.1 extern的方式

VC6创建项目：



与静态链接库的创建方式一样，我们创建一个新的类MyDLL，这样就会自动创建MyDLL.h和MyDLL.cpp：



在头文件MyDLL.h中我们要声明接口（函数），需要使用固定格式来声明而不能像静态链接库那样直接使用，格式如下：

```
1     extern "C" __declspec(dllexport) 调用约定 返回类型 函数名 (参数列表);
```

```
#if !defined(AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_)
#define AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

extern "C" __declspec(dllexport) __stdcall int add(int x, int y);

#endif // !defined(AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_)
```

在MyDLL.cpp中实现方法，需要在开头写上一致的调用约定：

```

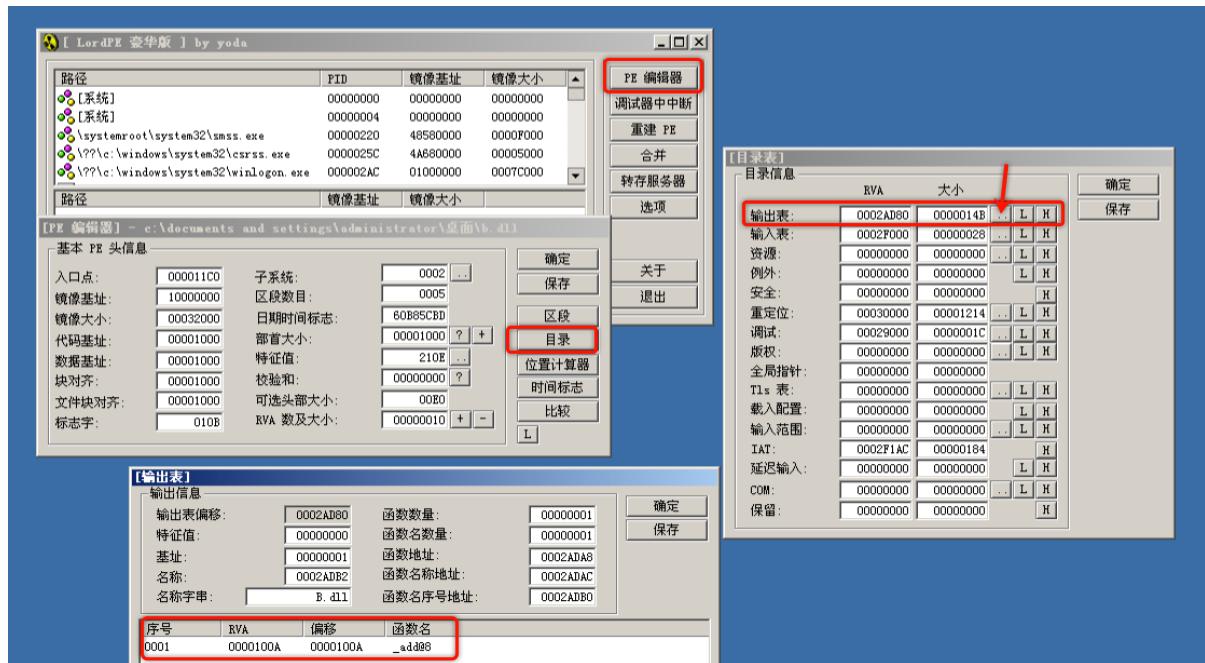
1 // MyDLL.cpp: implementation of the MyDLL class.
2 //
3 ///////////////////////////////////////////////////////////////////////////////
4
5 #include "stdafx.h"
6 #include "MyDLL.h"
7
8 ///////////////////////////////////////////////////////////////////////////////
9 // Construction/Destruction
10 ///////////////////////////////////////////////////////////////////////////////
11
12 _stdcall int add(int x, int y) {
13     return x+y;
14 }
15

```

编译后在Debug目录就会生成B.dll文件：



在这里我们可以使用LordPE来查看我们这个DLL文件的导出表（涉及中级班课程暂时可以略过），我们只要知道在这个导出表中有这个DLL声明的函数：



可以很清楚的看见我们的函数名称变成了 `_add@8`。

25.1.2 使用.DEF文件

我们可以在项目中创建一个文件扩展名为.def的文件，在该文件中使用如下格式来声明：

```

1  EXPORTS
2  函数名 @编号 // 有编号，也有名称
3  函数名 @编号      NONAME // 有编号，没有名称

```

按照这种方式修改如下：

头文件：

```

5  #if !defined(AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_)
6  #define AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_
7
8  #if _MSC_VER > 1000
9  #pragma once
10 #endif // _MSC_VER > 1000
11
12 int add(int x, int y);
13
14 #endif // !defined(AFX_MYDLL_H__76B660D2_1293_463C_8D06_17671F9B2403__INCLUDED_)
15

```

CPP文件：

```

5  #include "stdafx.h"
6  #include "MyDLL.h"
7
8  /////////////////////////////////
9  // Construction/Destruction
10 ///////////////////////////////
11
12 int add(int x, int y) {
13     return x+y;
14 }

```

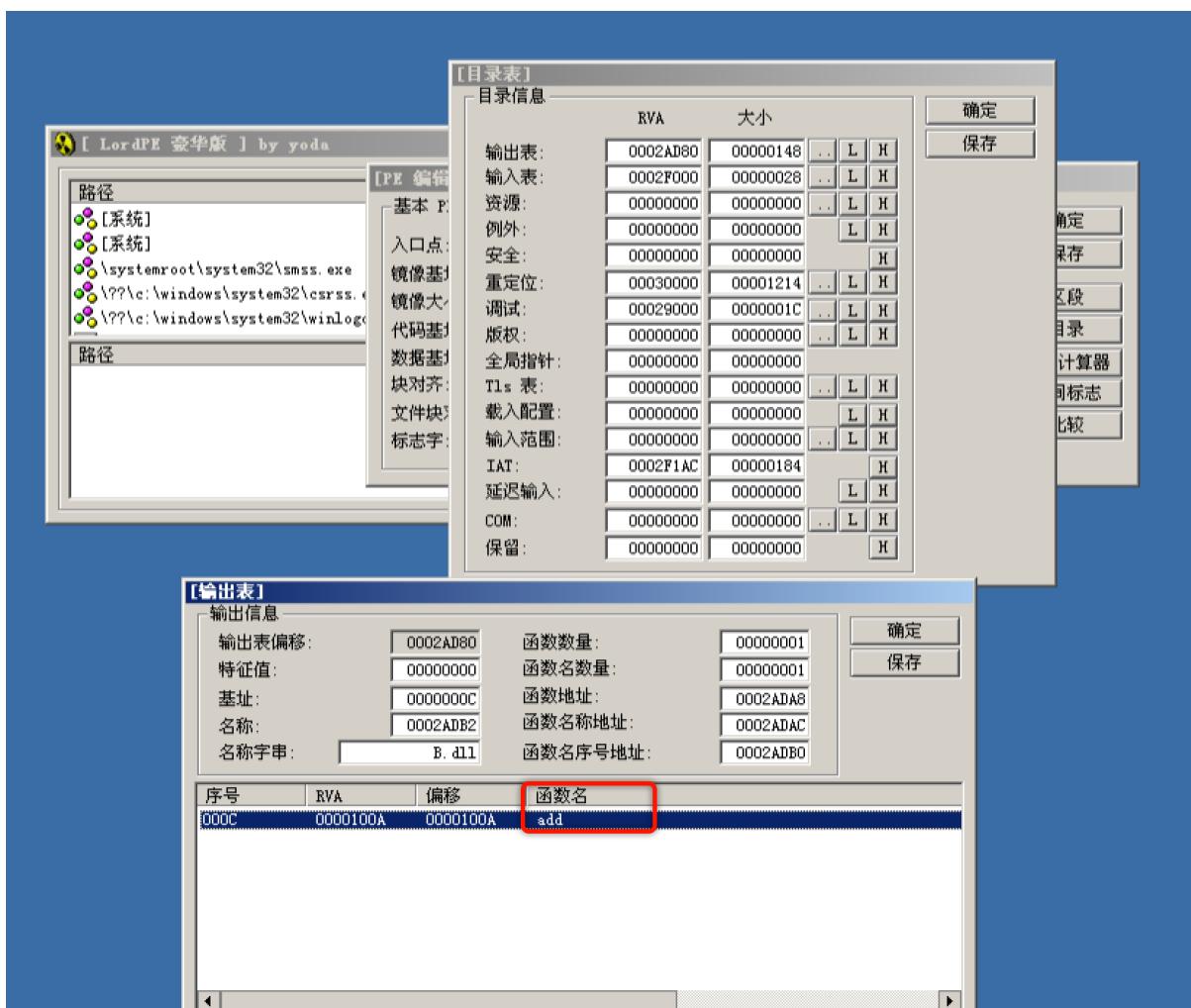
DEF文件：

```

1  EXPORTS
2
3  add @12

```

然后编译，用LordPE打开查看一下函数名称就会发现其没有了@xxx这样的格式而是我们定义什么样就是什么样：



这样做的好处就是：可以很直观的看见函数名，并且在应用层面可以达到隐藏的目的。

25.2 使用动态链接库

使用动态链接库的步骤比较繁琐，一共有如下几个步骤：

```

1 // 将DLL文件复制到项目目录下
2
3 // 步骤1：定义函数指针,如：
4 typedef int (*lpAdd)(int,int);
5
6 // 步骤2：声明函数指针变量,如：
7 lpAdd myAdd;
8
9 // 步骤3：动态加载dll到内存中,如：
10 // LoadLibrary函数会先从当前目录寻找,然后在系统目录寻找
11 HINSTANCE hModule = LoadLibrary("B.dll");
12
13 // 步骤4：获取函数地址,如：
14 myAdd = (lpAdd)GetProcAddress(hModule, "add");
15
16 // 步骤5：调用函数,如：
17 int a = myAdd(10,2);
18
19 // 步骤6：释放动态链接库,如：
20 FreeLibrary(hModule);

```

执行结果如下图：

```

// 步骤1： 定义函数指针,如:
typedef int (*lpAdd)(int,int);

// 步骤2： 声明函数指针变量,如:
lpAdd myAdd;

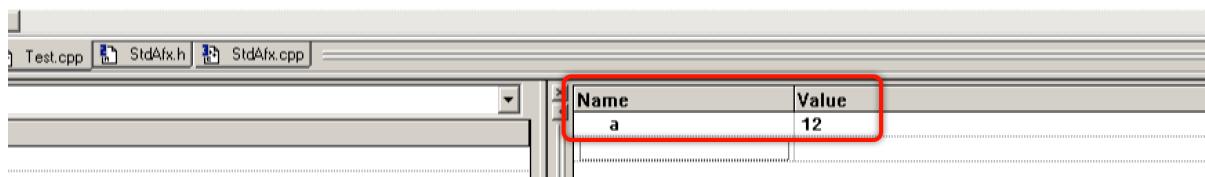
// 步骤3： 动态加载dll到内存中,如:
HINSTANCE hModule = LoadLibrary("B.dll");

// 步骤4： 获取函数地址,如:
myAdd = (lpAdd)GetProcAddress(hModule, "add");

// 步骤5： 调用函数,如:
int a = myAdd(10,2);

// 步骤6： 释放动态链接库,如:
FreeLibrary(hModule);
return 0;

```



26 隐式链接

之前我们调用动态链接库（DLL文件）使用的方式实际上是**显式链接**，它的优点是非常灵活，缺点就是使用起来非常麻烦，步骤很繁琐。

本章节我们来学习**隐式链接**，通过隐式链接我们只需要一次配置，之后就会非常的方便。

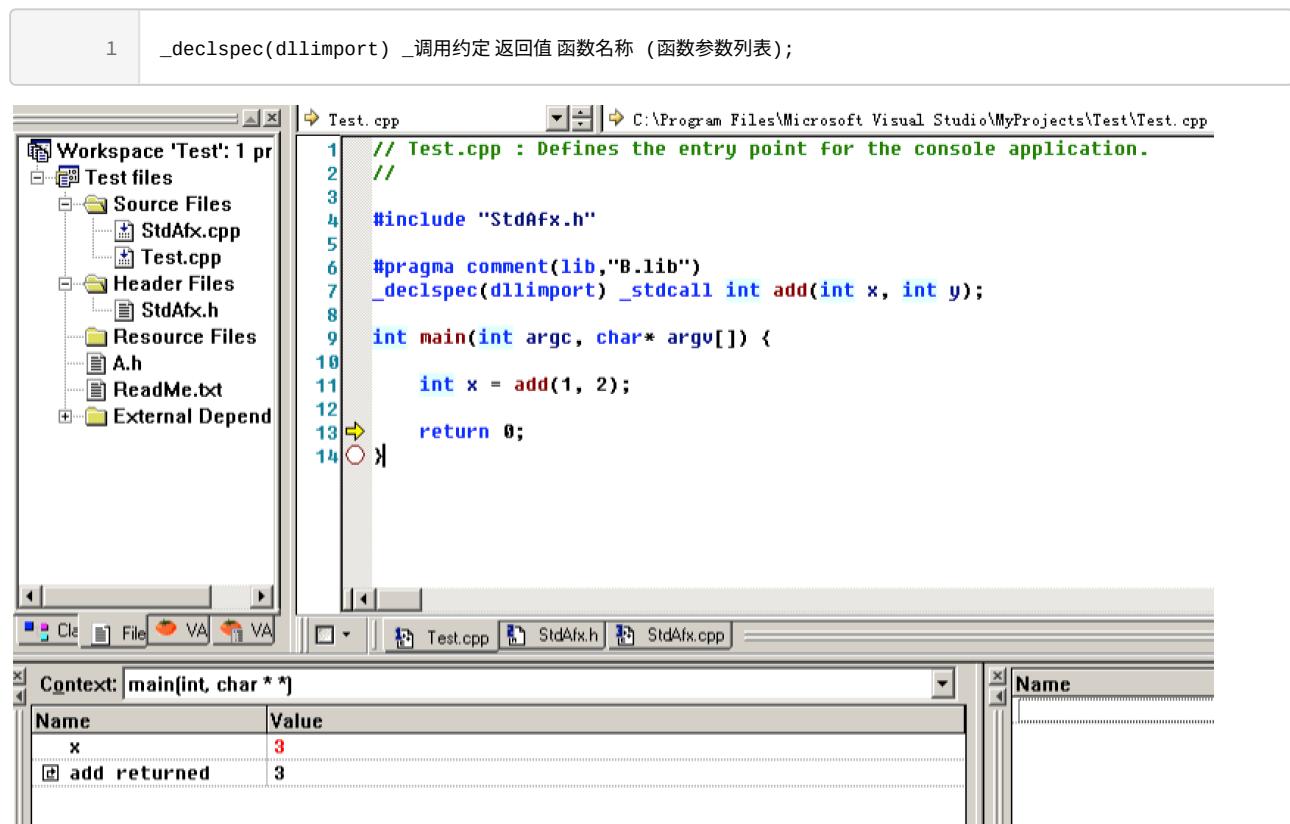
26.1 隐式链接

隐式链接有这几个步骤：

1. 将.dll和.lib放到项目目录下
2. 将`#pragma comment(lib, "DLL名.lib")`添加到调用文件
3. 加入函数声明

函数声明格式如下：

注意：在节目中给出的`_declspec`是有两个下划线的，经过查询之后实际上一个下划线和两个下划线是等价的。



注意，如果你创建动态链接库的方式是`extern`的方式，那么在第三步加入函数声明时就应该按照`extern`的格式来：

- ```

1 extern "C" _declspec(dllexport) 调用约定 返回类型 函数名 (参数列表);
2 extern "C" _declspec(dllimport) 调用约定 返回类型 函数名 (参数列表);

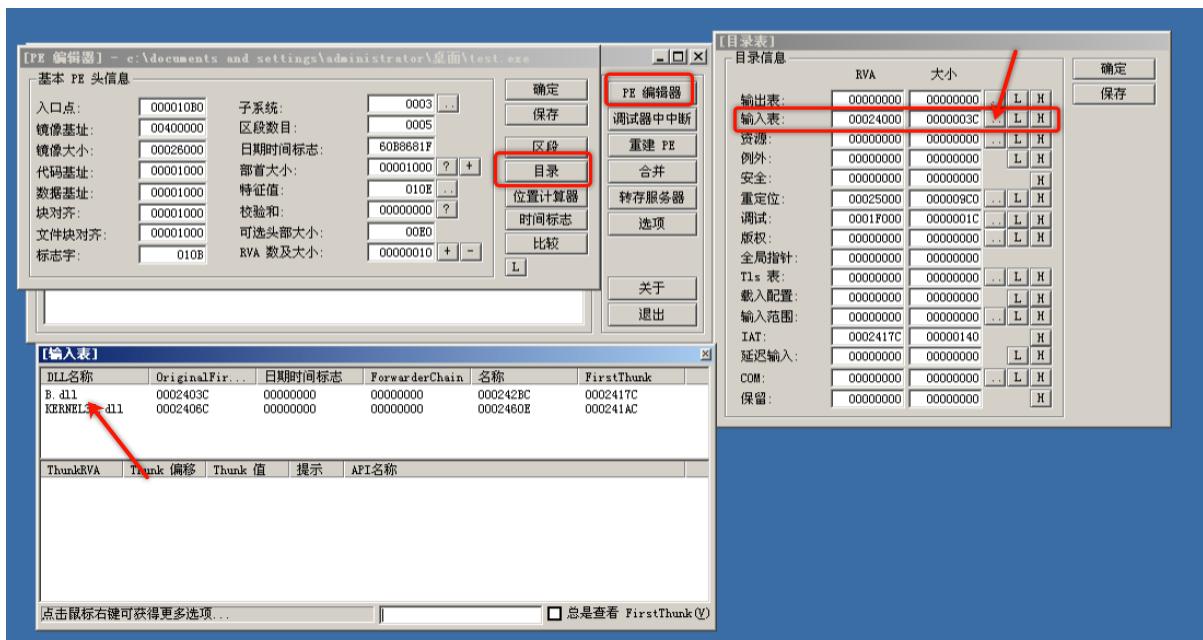
```

## 26.2 隐式链接的实现

使用隐式链接，编译器会将链接的DLL文件存放到导入表中：



我们可以使用LordPE来查看一下：



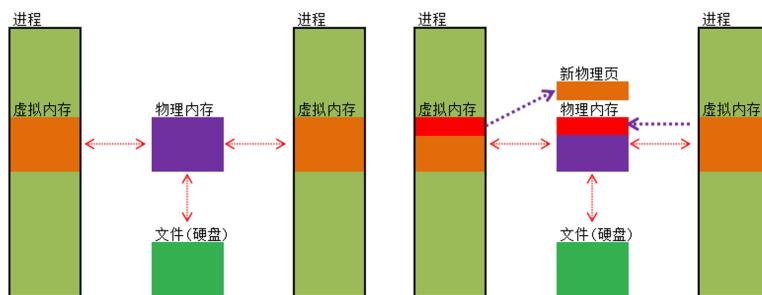
并且它可以详细的记录使用了DLL中的哪些函数：

| [输入表]        |                    |          |                |           |            |
|--------------|--------------------|----------|----------------|-----------|------------|
| DLL名称        | OriginalFirstThunk | 日期时间标志   | ForwarderChain | 名称        | FirstThunk |
| B.dll        | 0002403C           | 00000000 | 00000000       | 000242BC  | 0002417C   |
| KERNEL32.dll | 0002406C           | 00000000 | 00000000       | 0002460E  | 000241AC   |
| <hr/>        |                    |          |                |           |            |
| ThunkRVA     | Thunk 偏移           | Thunk 值  | 提示             | API名称     |            |
| 0002403C     | 0002303C           | 80000001 | -              | 序号: 1h 1d |            |

Thunk 数: 1h / 1d (OriginalFirstThunk chain)  总是查看 FirstThunk (V)

## 26.3 DLL的优点

DLL的优点如下图所示，DLL只在内存中加载一份，修改的时候就是写拷贝原理，不会影响别的进程使用DLL以及不会影响DLL本身：



## 26.4 DllMain函数

我们的控制台程序入口是Main函数，而DLL文件的入口函数是**DllMain函数**（**DllMain函数**可能会执行很多次，不像我们的Main函数只执行一次），其语法格式如下：

```

1 BOOL WINAPI DllMain(
2 HINSTANCE hinstDLL, // handle to the DLL module DLL模块的句柄, 当前DLL被加载到什么位置
3 DWORD fdwReason, // reason for calling function DLL被调用的原因, 有4种情况: DLL_PROCESS_ATTACH
4 (当某个进程第一次执行LoadLibrary)、DLL_PROCESS_DETACH (当某个进程释放了DLL)、DLL_THREAD_ATTACH (当某个进
5 程的其他线程再次执行LoadLibrary)、DLL_THREAD_DETACH (当某个进程的其他线程释放了DLL)
6 LPVOID lpvReserved // reserved
7);

```

## 27 远程线程

### 27.1 线程的概念

线程是附属在进程上的执行实体，是代码的执行流程；代码必须通过线程才能执行。

### 27.2 创建远程线程

创建远程线程的函数是**CreateRemoteThread**，其语法格式如下：

```

1 HANDLE CreateRemoteThread(
2 HANDLE hProcess, // handle to process 输入类型，进程句柄
3 LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD 输入类型，安全属性，包含安全描述符
4 SIZE_T dwStackSize, // initial stack size 输入类型，堆大小
5 LPTHREAD_START_ROUTINE lpStartAddress, // thread function 输入类型，线程函数，线程函数地址应该是在别的
6 //进程中存在的
7 LPVOID lpParameter, // thread argument 输入类型，线程参数
8 DWORD dwCreationFlags, // creation option 输入类型，创建设置
9 LPDWORD lpThreadId); // thread identifier 输出类型，线程id

```

**CreateThread**函数是在当前进程中创建线程，而**CreateRemoteThread**函数是允许在其他进程中创建线程，所以远程线程就可以理解为是非本进程中的线程。

首先创建A进程，代码如下：

```

1 void Fun() {
2 for(int i = 0; i <= 5; i++) {
3 printf("Fun running... \n");
4 Sleep(1000);
5 }
6 }
7
8 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
9 Fun();
10 return 0;
11 }
12
13 int main(int argc, char* argv[]) {
14
15 HANDLE hThread = CreateThread(NULL, NULL, ThreadProc, NULL, 0, NULL);
16
17 CloseHandle(hThread);
18
19 getchar();
20 return 0;
21 }

```

进程B写了一个远程线程创建的代码：

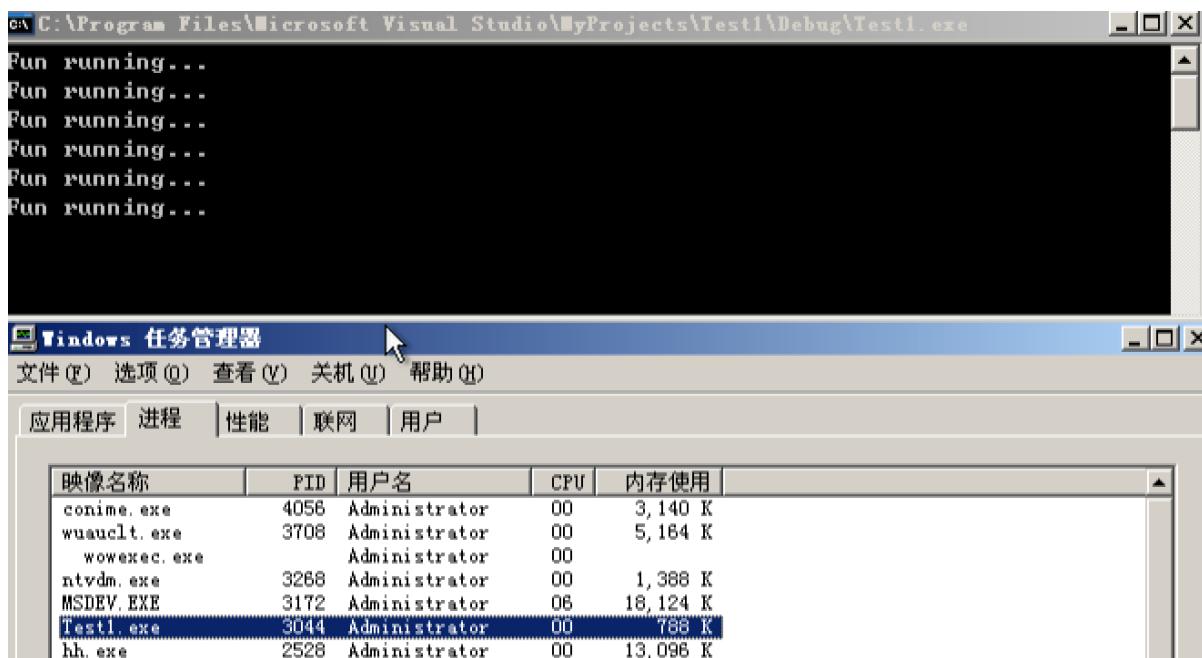
```

1 BOOL MyCreateRemoteThread(DWORD dwProcessId, DWORD dwProcessAddr) {
2 DWORD dwThreadId;
3 HANDLE hProcess;
4 HANDLE hThread;
5 // 1. 获取进程句柄
6 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessId);
7 // 判断OpenProcess是否执行成功
8 if(hProcess == NULL) {
9 OutputDebugString("OpenProcess failed! \n");
10 return FALSE;
11 }
12 // 2. 创建远程线程
13 hThread = CreateRemoteThread(
14 hProcess, // handle to process
15 NULL, // SD
16 0, // initial stack size
17 (LPTHREAD_START_ROUTINE)dwProcessAddr, // thread function
18 NULL, // thread argument
19 0, // creation option
20 &dwThreadId // thread identifier
21);
22 // 判断CreateRemoteThread是否执行成功
23 if(hThread == NULL) {
24 OutputDebugString("CreateRemoteThread failed! \n");
25 CloseHandle(hProcess);
26 return FALSE;
27 }
28 // 3. 关闭
29 CloseHandle(hThread);
30 CloseHandle(hProcess);
31
32 // 返回
33 return TRUE;
34 }

```

函数MyCreateRemoteThread传入2个参数，一个是进程ID，一个是线程函数地址。

进程ID通过任务管理器查看：



我们在进程A的代码下断点找到线程函数地址：

```

13: DWORD WINAPI ThreadProc(LPVOID lpParameter) {
 004010B0 push ebp
 004010B1 mov ebp,esp
 004010B3 sub esp,40h
 004010B6 push ebx
 004010B7 push esi
 004010B8 push edi
 004010B9 lea edi,[ebp-40h]
 004010BC mov ecx,10h
 004010C1 mov eax,0CCCCCCCCCh
 004010C6 rep stos dword ptr [edi]
14: Fun();
 004010C8 call @ILT+5(Fun) (0040100a)
15: return 0;
 004010CD xor eax,eax

```

然后将对应值填入即可远程创建线程：

The screenshot shows a Microsoft Visual Studio interface during a debug session. The top part is the code editor with the following C++ code:

```
10 // 1. 获取进程句柄
11 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessId);
12 // 判断OpenProcess是否执行成功
13 if(hProcess == NULL) {
14 OutputDebugString("OpenProcess failed! \n");
15 return FALSE;
16 }
17 // 2. 创建远程线程
18 hThread = CreateRemoteThread(
19 hProcess,
20 NULL, // SD
21 0, // dwStackSize
22 (LPTHREAD_START_ROUTINE)dwP,
23 NULL,
24 0, // cr
25 &dwThreadId
26);
27 // 判断CreateRemoteThread是否执行成功
28 if(hThread == NULL) {
29 }
```

The bottom part shows two windows: a command prompt window titled "C:\Program Files\Microsoft Visual Studio\MyProjects\Test\Debug\Test.exe" displaying the output "Fun running..." repeated multiple times, and a "Watch3 / Watch4" window which is currently empty.

## 28 远程线程注入

之前我们是远程创建线程，调用的也是人家自己的线程函数，而如果我们想要创建远程线程调用自己定义的线程函数就需要使用**远程线程注入**技术。

### 28.1 什么是注入

所谓注入就是在第三方进程不知道或者不允许的情况下将模块或者代码写入对方进程空间，并设法执行的技术。

在安全领域，“注入”是非常重要的一种技术手段，注入与反注入也一直处于不断变化的，而且也愈来愈激烈的对抗当中。

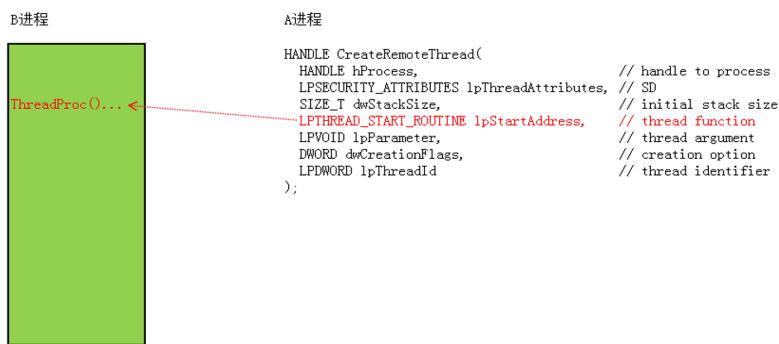
**已知的注入方式：**

远程线程注入、APC注入、消息钩子注入、注册表注入、导入表注入、输入法注入等等。

### 28.2 远程线程注入的流程

远程线程注入的思路就是在进程A中创建线程，将线程函数指向**LoadLibrary**函数。

那么为什么可以这样呢？这是因为我们执行远程线程函数满足返回值是4字节，一个参数是4字节即可（ThreadProc就是这样的条件）：



我们再来看一下**LoadLibrary**函数的语法格式：

|   |                                           |
|---|-------------------------------------------|
| 1 | HMODULE LoadLibrary(                      |
| 2 | LPCTSTR lpFileName // file name of module |
| 3 | );                                        |

我们可以跟进（F12）一下**HMODULE**和**LPCTSTR**这两个宏的定义，就会发现其实都是4字节宽度。

具体实现步骤如下图所示：

进程A



思路：

在进程A中创建线程，将线程函数指向为：LoadLibrary()

具体实现步骤：

`LoadLibrary("A.DLL")`

- 1、在进程A中分配空间，存储“`A.DLL`”
- 2、获取`LoadLibrary`函数的地址
- 3、创建远程线程，执行`LoadLibrary()`函数

## 28.3 如何执行代码

DLL文件，在DLL文件入口函数判断并创建线程：

```

1 // B.cpp : Defines the entry point for the DLL application.
2 //
3
4 #include "stdafx.h"
5
6 DWORD WINAPI ThreadProc(LPVOID lpParameter) {
7 for (;;) {
8 Sleep(1000);
9 printf("DLL RUNNING...");
10 }
11 }
12
13 BOOL APIENTRY DllMain(HANDLE hModule,
14 DWORD ul_reason_for_call,
15 LPVOID lpReserved
16)
17 { // 当进程执行LoadLibrary时创建一个线程，执行ThreadProc线程
18 switch (ul_reason_for_call) {
19 case DLL_PROCESS_ATTACH:
20 CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
21 break;
22 }
23 return TRUE;
24 }
```

文件我们用之前写的Test1.exe即可，将编译好的DLL和Test1.exe放在同一个目录并打开Test1.exe。

注入实现：

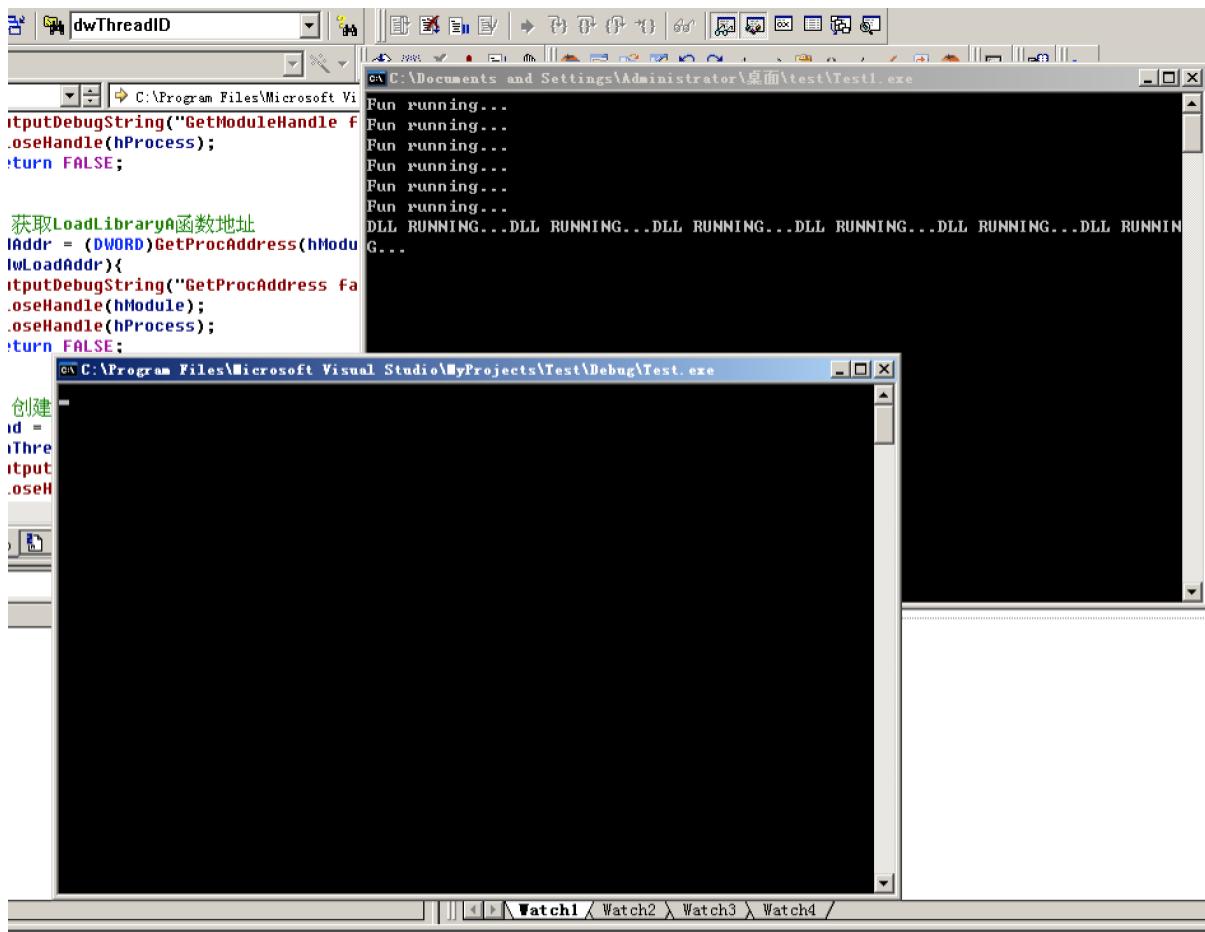
```

1 // Test.cpp : Defines the entry point for the console application.
2 //
3
4 #include "StdAfx.h"
5
6 // LoadDll需要两个参数一个参数是进程ID，一个是DLL文件的路径
7 BOOL LoadDll(DWORD dwProcessID, char* szDllPathName) {
8
9 BOOL bRet;
10 HANDLE hProcess;
11 HANDLE hThread;
12 DWORD dwLength;
13 DWORD dwLoadAddr;
14 LPVOID lpAllocAddr;
15 DWORD dwThreadID;
16 HMODULE hModule;
17
18 bRet = 0;
19 dwLoadAddr = 0;
20 hProcess = 0;
21
22 // 1. 获取进程句柄
23 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessID);
24 if (hProcess == NULL) {
25 OutputDebugString("OpenProcess failed! \n");
26 return FALSE;
27 }
28
29 // 2. 获取DLL文件路径的长度，并在最后+1，因为要加上0结尾的长度
30 dwLength = strlen(szDllPathName) + 1;
31
32 // 3. 在目标进程分配内存
33 lpAllocAddr = VirtualAllocEx(hProcess, NULL, dwLength, MEM_COMMIT, PAGE_READWRITE);
34 if (lpAllocAddr == NULL) {
35 OutputDebugString("VirtualAllocEx failed! \n");
36 CloseHandle(hProcess);
37 return FALSE;
38 }
39
40 // 4. 拷贝DLL路径名字到目标进程的内存
41 bRet = WriteProcessMemory(hProcess, lpAllocAddr, szDllPathName, dwLength, NULL);
42 if (!bRet) {
43 OutputDebugString("WriteProcessMemory failed! \n");
44 CloseHandle(hProcess);
45 return FALSE;
46 }
47
48 // 5. 获取模块句柄
49 // LoadLibrary这个函数是在kernel32.dll这个模块中的，所以需要现货区kernel32.dll这个模块的句柄
50 hModule = GetModuleHandle("kernel32.dll");
51 if (!hModule) {
52 OutputDebugString("GetModuleHandle failed! \n");
53 CloseHandle(hProcess);
54 return FALSE;
55 }

```

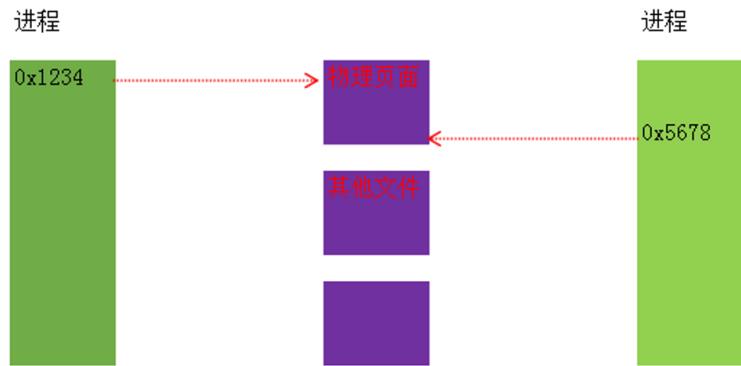
```
56
57 // 6. 获取LoadLibraryA函数地址
58 dwLoadAddr = (DWORD)GetProcAddress(hModule, "LoadLibraryA");
59 if (!dwLoadAddr){
60 OutputDebugString("GetProcAddress failed! \n");
61 CloseHandle(hModule);
62 CloseHandle(hProcess);
63 return FALSE;
64 }
65
66 // 7. 创建远程线程，加载DLL
67 hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)dwLoadAddr,
68 lpAllocAddr, 0, &dwThreadID);
69 if (!hThread){
70 OutputDebugString("CreateRemoteThread failed! \n");
71 CloseHandle(hModule);
72 CloseHandle(hProcess);
73 return FALSE;
74 }
75
76 // 8. 关闭进程句柄
77 CloseHandle(hThread);
78 CloseHandle(hProcess);
79
80 return TRUE;
81 }
82
83 int main(int argc, char* argv[]) {
84 LoadDll(384, "C:\\\\Documents and Settings\\\\Administrator\\\\桌面\\\\test\\\\B.dll");
85 getchar();
86 return 0;
87 }
```

注入成功：



## 29 进程间通信

同一台机器上进程之间的通信虽然有很多种方法，但其本质就是共享内存。



### 29.1 举例说明

假设现在我们**进程A**的代码是这样的：

```

1 void Attack()
2 {
3 printf("*****攻击***** \n");
4 return;
5 }
6
7 void Rest()
8 {
9 printf("*****打坐***** \n");
10 return;
11 }
12
13 void Blood()
14 {
15 printf("*****加血***** \n");
16 return;
17 }
18
19 int main(int argc, char* argv[]) {
20 char cGetchar;
21 printf("*****GAME BEGIN***** \n");
22 while(1) {
23 cGetchar = getchar();
24 switch(cGetchar) {
25 case 'A':
26 {
27 Attack();
28 break;
29 }
30 case 'R':
31 {
32 Rest();
33 break;
34 }
35 case 'B':
36 {
37 Blood();
38 break;
39 }
40 }
41 }
42 return 0;
43 }
```

这就是获取输入的字符来攻击、打坐、加血的小程序，我们想要自动化的控制这个程序而不是自己输入该怎么办？这时候就需要使用平时中大家常提的外挂技术，在这里实际上就是**远程线程注入**，通过**进程B控制进程A的执行流程**。

如下是DLL文件的代码：

```

1 // B.cpp : Defines the entry point for the DLL application.
2 //
3
4 #include "stdafx.h"
5
6 #define _MAP_ "共享内存"
7
8 // 首先需要获取函数的地址
9 #define ATTACK 0x00401030
10 #define REST 0x00401080
11 #define BLOOD 0x004010D0
12
13 HANDLE g_hModule;
14 HANDLE g_hMapFile;
15 LPTSTR lpBuffer;
16 DWORD dwType;
17
18 DWORD WINAPI ThreadProc(LPVOID lpParameter)
19 {
20 dwType = 0;
21 g_hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, _MAP_);
22
23 if (g_hMapFile == NULL)
24 {
25 printf("OpenFileMapping failed: %d", GetLastError());
26 return 0;
27 }
28
29 //映射内存
30 lpBuffer = (LPTSTR)MapViewOfFile(g_hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, BUFSIZE);
31
32 for (;;)
33 {
34 if (lpBuffer != NULL)
35 {
36 // 读取数据
37 CopyMemory(&dwType, lpBuffer, 4);
38 }
39
40 if (dwType == 1)
41 {
42 // 攻击
43 __asm {
44 mov eax, ATTACK
45 call eax
46 }
47 dwType == 0;
48 CopyMemory(&dwType, lpBuffer, 4);
49 }
50
51 if (dwType == 2)
52 {
53 // 打坐
54 __asm {
55 mov eax, REST

```

```

56 call eax
57 }
58 dwType == 0;
59 CopyMemory(&dwType, lpBuffer, 4);
60 }
61
62 if (dwType == 3)
63 {
64 // 加血
65 __asm {
66 mov eax, BLOOD
67 call eax
68 }
69 dwType == 0;
70 CopyMemory(&dwType, lpBuffer, 4);
71 }
72
73 if (dwType == 4)
74 {
75 //卸载自身并退出
76 FreeLibraryAndExitThread((HMODULE)g_hModule, 0);
77 }
78
79 Sleep(500);
80 }
81
82 return 0;
83 }
84
85 BOOL APIENTRY DllMain(HMODULE hModule,
86 DWORD ul_reason_for_call,
87 LPVOID lpReserved
88)
89 {
90 switch (ul_reason_for_call) {
91 case DLL_PROCESS_ATTACH:
92 {
93 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);
94 break;
95 }
96 }
97 return TRUE;
98 }
```

需要注意的是我们首先需要获取函数的地址，这个我们可以通过VC6反汇编来寻找：

The screenshot shows the Microsoft Visual Studio interface with the title bar "Test - Microsoft Visual C++ [run] - [Disassembly]". The menu bar includes File, Edit, View, Insert, Project, Debug, Tools, Window, Help. The toolbar has various icons for file operations. The left pane displays the project structure under "Workspace 'Test': 1 pr": "Test files" contains "Source Files" (StdAfx.cpp, Test.cpp) and "Header Files" (StdAfx.h); "Resource Files" contains "A.h"; "External Depend" contains "ReadMe.txt". The right pane is titled "dwFileMapSize" and shows assembly code for the "Rest()" function:

```
0040107B int 3
0040107C int 3
0040107D int 3
0040107E int 3
0040107F int 3
--- C:\Program Files\Microsoft Visual Studio\MyProjects\Test\Test.cpp -----
11:
12: void Rest()
13: {
00401080 push ebp
00401081 mov ebp,esp
00401083 sub esp,40h
00401086 push ebx
00401087 push esi
00401088 push edi
00401089 lea edi,[ebp-40h]
0040108C mov ecx,10h
00401091 mov eax,0CCCCCCCCCh
00401096 rep stos dword ptr [edi]
```

编译好DLL之后，我们需要一个进程B来控制进程A，代码如下：

```

1 #include <tlhelp32.h>
2 #include <stdio.h>
3 #include <windows.h>
4
5 #define _MAP_ "共享内存"
6
7 HANDLE g_hMapFile;
8 LPTSTR lpBuffer;
9
10 BOOL LoadDll(DWORD dwProcessID, char* szDllPathName) {
11
12 BOOL bRet;
13 HANDLE hProcess;
14 HANDLE hThread;
15 DWORD dwLength;
16 DWORD dwLoadAddr;
17 LPVOID lpAllocAddr;
18 DWORD dwThreadId;
19 HMODULE hModule;
20
21 bRet = 0;
22 dwLoadAddr = 0;
23 hProcess = 0;
24
25 // 1. 获取进程句柄
26 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessID);
27 if (hProcess == NULL) {
28 OutputDebugString("OpenProcess failed! \n");
29 return FALSE;
30 }
31
32 // 2. 获取DLL文件路径的长度，并在最后+1，因为要加上0结尾的长度
33 dwLength = strlen(szDllPathName) + 1;
34
35 // 3. 在目标进程分配内存
36 lpAllocAddr = VirtualAllocEx(hProcess, NULL, dwLength, MEM_COMMIT, PAGE_READWRITE);
37 if (lpAllocAddr == NULL) {
38 OutputDebugString("VirtualAllocEx failed! \n");
39 CloseHandle(hProcess);
40 return FALSE;
41 }
42
43 // 4. 拷贝DLL路径名字到目标进程的内存
44 bRet = WriteProcessMemory(hProcess, lpAllocAddr, szDllPathName, dwLength, NULL);
45 if (!bRet) {
46 OutputDebugString("WriteProcessMemory failed! \n");
47 CloseHandle(hProcess);
48 return FALSE;
49 }
50
51 // 5. 获取模块句柄
52 // LoadLibrary这个函数是在kernel32.dll这个模块中的，所以需要现货区kernel32.dll这个模块的句柄
53 hModule = GetModuleHandle("kernel32.dll");
54 if (!hModule) {
55 OutputDebugString("GetModuleHandle failed! \n");

```

```

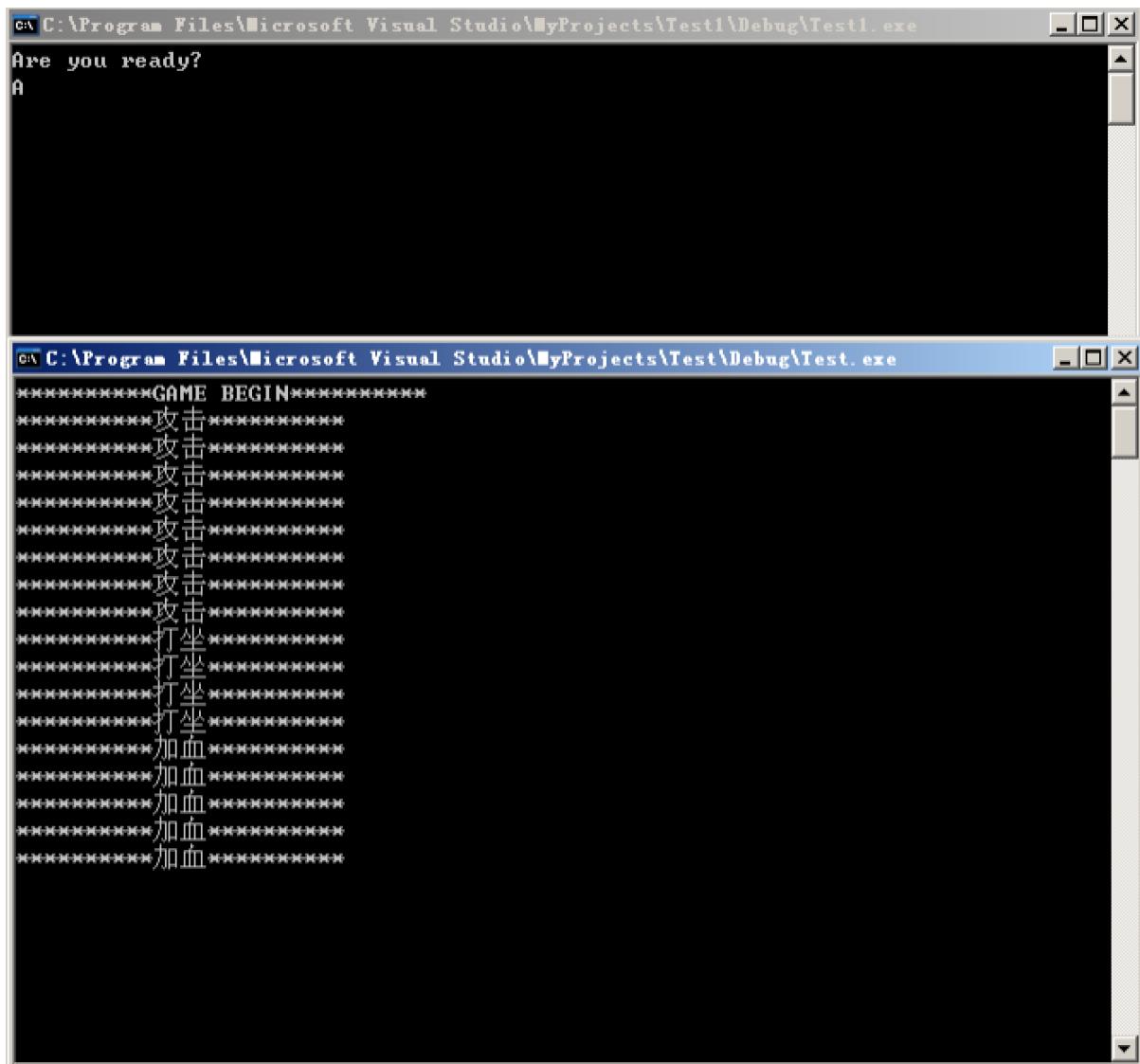
56 CloseHandle(hProcess);
57 return FALSE;
58 }
59
60 // 6. 获取LoadLibraryA函数地址
61 dwLoadAddr = (DWORD)GetProcAddress(hModule, "LoadLibraryA");
62 if (!dwLoadAddr){
63 OutputDebugString("GetProcAddress failed! \n");
64 CloseHandle(hModule);
65 CloseHandle(hProcess);
66 return FALSE;
67 }
68
69 // 7. 创建远程线程, 加载DLL
70 hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)dwLoadAddr,
71 lpAllocAddr, 0, &dwThreadID);
72 if (!hThread){
73 OutputDebugString("CreateRemoteThread failed! \n");
74 CloseHandle(hModule);
75 CloseHandle(hProcess);
76 return FALSE;
77 }
78
79 // 8. 关闭进程句柄
80 CloseHandle(hThread);
81 CloseHandle(hProcess);
82
83 return TRUE;
84 }
85
86 BOOL Init()
87 {
88 // 创建共享内存
89 g_hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, 0x1000, _MAP_);
90 if (g_hMapFile == NULL)
91 {
92 printf("CreateFileMapping failed! \n");
93 return FALSE;
94 }
95
96 // 映射内存
97 lpBuffer = (LPTSTR)MapViewOfFile(g_hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, BUFSIZ);
98 if (lpBuffer == NULL)
99 {
100 printf("MapViewOfFile failed! \n");
101 return FALSE;
102 }
103
104 return TRUE;
105 }
106
107 // 根据进程名称获取进程ID
108 DWORD GetPID(char *szName)
109 {
110 HANDLE hProcessSnapShot = NULL;
111 PROCESSENTRY32 pe32 = {0};

```

```

111
112 hProcessSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
113 if (hProcessSnapShot == (HANDLE)-1)
114 {
115 return 0;
116 }
117
118 pe32.dwSize = sizeof(PROCESSENTRY32);
119 if (Process32First(hProcessSnapShot, &pe32))
120 {
121 do {
122 if (!strcmp(szName, pe32.szExeFile)) {
123 return (int)pe32.th32ProcessID;
124 }
125 } while (Process32Next(hProcessSnapShot, &pe32));
126 }
127 else
128 {
129 CloseHandle(hProcessSnapShot);
130 }
131 return 0;
132 }
133
134 int main()
135 {
136 DWORD dwCtrlCode = 0;
137 // 指令队列
138 DWORD dwOrderList[10] = {1, 1, 2, 3, 3, 1, 2, 1, 3, 4};
139
140 printf("Are you ready? \n");
141
142 getchar();
143
144 if (Init()) {
145 LoadDll(GetPID("Test.exe"), (char*)"C:\\\\Documents and Settings\\\\Administrator\\\\桌面\\\\test\\\\B.dll");
146 }
147
148 for (int i = 0; i < 10; i++)
149 {
150 dwCtrlCode = dwOrderList[i];
151 CopyMemory(lpBuffer, &dwCtrlCode, 4);
152 Sleep(2000);
153 }
154
155 getchar();
156
157 return 0;
158 }
```

成功执行并控制了进程A：



## 30 模块隐藏

之前我们了解了直接注入一个DLL到进程中，但是这样实际上是很困难存活的，因为程序很容易就可以通过API来获取当前加载的DLL模块，所以我们需要使用模块隐藏技术来隐藏自己需要注入的DLL模块。

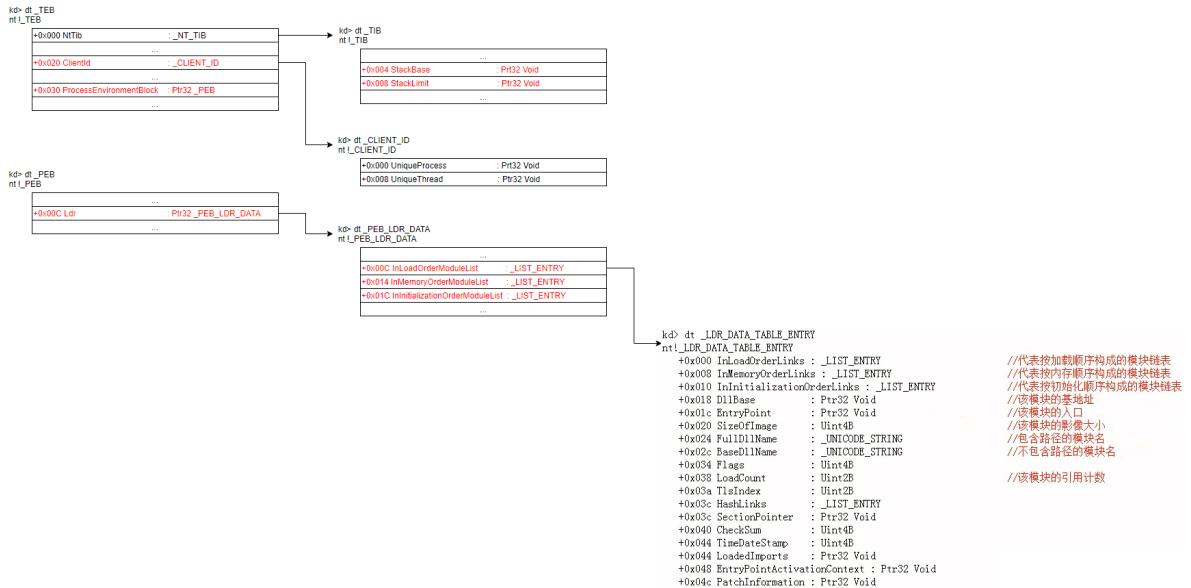
### 30.1 模块隐藏之断链

API是通过什么将模块查询出来的？其实API都是从这几个结构体（**结构体属于3环应用层**）中查询出来的：

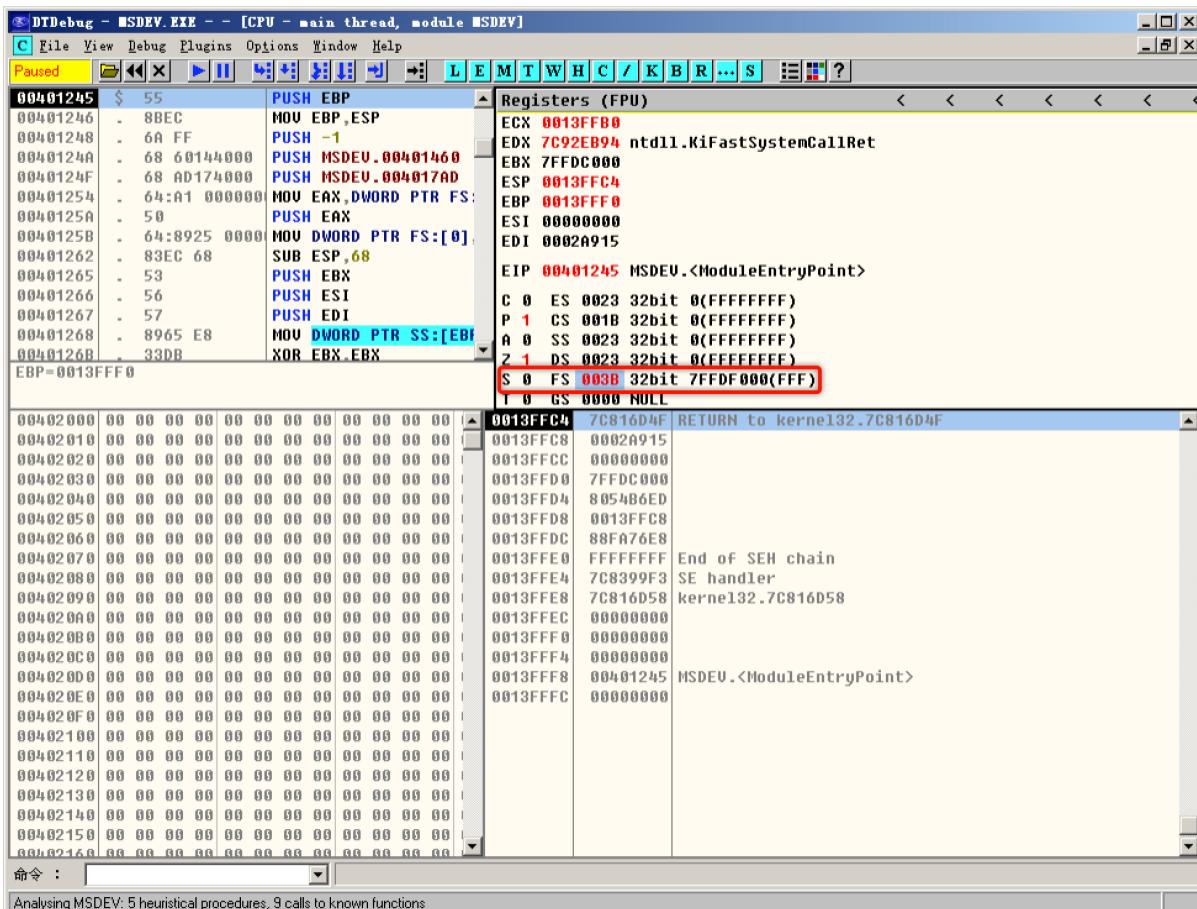
1. TEB(Thread Environment Block, 线程环境块)，它存放线程的相关信息，每一个线程都有自己的TEB信息，FS:[0]即是当前线程的TEB。
2. PEB(Process Environment Block, 进程环境块)，它存放进程的相关信息，每个进程都有自己的PEB信息，FS:[0x30]即当前进程的PEB。

如下图所示（只介绍与本章节相关的信息）

1. TEB第一个成员是一个结构体，这个结构体包含了当前线程栈底和当前线程栈的界限；TEB的**020**偏移是一个结构体，其包含了两个成员，一个是当前线程所在进程的PID和当前线程自己的线程ID；
2. PEB的**00c**偏移是一个结构体，这个结构体包括**模块链表**，API函数遍历模块就是查看这个链表。



我们如何去获取这个**TEB结构体**呢？我们可以随便找一个EXE拖进DDTDebug，然后来看一下**FS寄存器**（目前你只需要知道TEB的地址就存储在FS寄存器中即可，具体细节在中级课程中）：



我们可以在左下角使用**dd 7FFDF000**命令来查看TEB结构体：

|          |          |                                   |
|----------|----------|-----------------------------------|
| 7FFDF000 | 0013FFE0 | (Pointer to SEH chain)            |
| 7FFDF004 | 00140000 | (Top of thread's stack)           |
| 7FFDF008 | 0013D000 | (Bottom of thread's stack)        |
| 7FFDF00C | 00000000 |                                   |
| 7FFDF010 | 00001E00 |                                   |
| 7FFDF014 | 00000000 |                                   |
| 7FFDF018 | 7FFDF000 |                                   |
| 7FFDF01C | 00000000 |                                   |
| 7FFDF020 | 000008C4 |                                   |
| 7FFDF024 | 000000A0 | (Thread ID)                       |
| 7FFDF028 | 00000000 |                                   |
| 7FFDF02C | 00000000 | (Pointer to Thread Local Storage) |
| 7FFDF030 | 7FFDC000 |                                   |
| 7FFDF034 | 00000000 | (Last error = ERROR_SUCCESS)      |
| 7FFDF038 | 00000000 |                                   |
| 7FFDF03C | 00000000 |                                   |
| 7FFDF040 | E2E33218 |                                   |
| 7FFDF044 | 00000000 |                                   |
| 7FFDF048 | 00000000 |                                   |
| 7FFDF04C | 00000000 |                                   |
| 7FFDF050 | 00000000 |                                   |
| 7FFDF054 | 00000000 |                                   |
| 7FFDF058 | 00000000 |                                   |

命令 : dd 7FFDF000 DD [地址] -- 显示堆

FS寄存器中存储的就是目前正在使用的线程的TEB结构体的地址。

PEB结构体同理，我们只需要找到FS寄存器中存储地址的0x30偏移然后跟进即可：

|        |          |                              |
|--------|----------|------------------------------|
| \$ ==> | 0013FFE0 | (Pointer to SEH chain)       |
| \$+4   | 00140000 | (Top of thread's stack)      |
| \$+8   | 0013D000 | (Bottom of thread's stack)   |
| \$+C   | 00000000 |                              |
| \$+10  | 00001E00 |                              |
| \$+14  | 00000000 |                              |
| \$+18  | 7FFDF000 |                              |
| \$+1C  | 00000000 |                              |
| \$+20  | 000008C4 |                              |
| \$+24  | 000000A0 | (Thread ID)                  |
| \$+28  | 00000000 |                              |
| \$+2C  | 00000000 | (Pointer to Thread Local St  |
| \$+30  | 7FFDC000 |                              |
| \$+34  | 00000000 | (Last error = ERROR_SUCCESS) |
| \$+38  | 00000000 |                              |
| \$+3C  | 00000000 |                              |
| \$+40  | E2E33218 |                              |
| \$+44  | 00000000 |                              |
| \$+48  | 00000000 |                              |
| \$+4C  | 00000000 |                              |
| \$+50  | 00000000 |                              |
| \$+54  | 00000000 |                              |
| \$+58  | 00000000 |                              |

命令 : dd 7FFDF000

Analysing MSDEV: 5 heuristical procedures, 9 calls to known functions

|       |          |                                    |
|-------|----------|------------------------------------|
| \$+30 | 7FFDC000 |                                    |
| \$+34 | 0        | Backup > error = ERROR_SUCCESS     |
| \$+38 | 0        | Copy >                             |
| \$+3C | 0        | Binary >                           |
| \$+40 | E        | Modify >                           |
| \$+44 | 0        | Breakpoint >                       |
| \$+48 | 0        | Search for >                       |
| \$+4C | 0        | Follow in Dump >                   |
| \$+50 | 0        | Go to >                            |
| \$+54 | 0        | Hex >                              |
| \$+58 | 0        | Text >                             |
| \$+5C | 0        | Short > 9 calls to known functions |
| \$+60 | 0        | Long >                             |
| \$+64 | 0        | Float >                            |
| \$+68 | 0        | Disassemble >                      |
| \$+6C | 0        | Special >                          |

命令 : dd 7

Analysing MSDEV

|          |            |                               |
|----------|------------|-------------------------------|
| 7FFDC000 | 00010000   | UNICODE "=::=:\"              |
| 7FFDC004 | FFFFFFFFFF |                               |
| 7FFDC008 | 00400000   | MSDEV.00400000                |
| 7FFDC00C | 00250688   |                               |
| 7FFDC010 | 00020000   |                               |
| 7FFDC014 | 00000000   |                               |
| 7FFDC018 | 00150000   |                               |
| 7FFDC01C | 7C99E4C0   | ntdll.7C99E4C0                |
| 7FFDC020 | 7C921005   | ntdll.RtlEnterCriticalSection |
| 7FFDC024 | 7C9210ED   | ntdll.RtlLeaveCriticalSection |
| 7FFDC028 | 00000001   |                               |
| 7FFDC02C | 77D12970   | USER32.77D12970               |
| 7FFDC030 | 00000000   |                               |
| 7FFDC034 | 00000000   |                               |
| 7FFDC038 | 00000000   |                               |
| 7FFDC03C | 00000000   |                               |
| 7FFDC040 | 7C99E480   | ntdll.7C99E480                |
| 7FFDC044 | 00007FFF   |                               |
| 7FFDC048 | 00000000   |                               |
| 7FFDC04C | 7F6F0000   |                               |
| 7FFDC050 | 7F6F0000   |                               |
| 7FFDC054 | 7F6F0688   |                               |
| 7FFDC058 | 7FF00000   |                               |

命令 : dd 7FFDF000

Analysing MSDEV: 5 heuristical procedures, 9 calls to known functions

我们之前已经了解到了API函数遍历模块就是查看PEB那个链表，所以我们要想办法让它在查询的时候断链。

### 30.1.1 断链实现代码

如下我们通过断链的方式实现了一个隐藏模块的函数：

```

1 void HideModule(char* szModuleName) {
2 // 获取模块的句柄
3 HMODULE hMod = GetModuleHandle(szModuleName);
4 PLIST_ENTRY Head, Cur;
5 PPEB_LDR_DATA ldr;
6 PLDR_MODULE ldmod;
7
8 __asm {
9 mov eax, fs:[0x30] // 取PEB结构体
10 mov ecx, [eax + 0x0c] // 取PEB结构体的00c偏移的结构体，就是PPEB_LDR_DATA
11 mov ldr, ecx // 将ecx给到ldr
12 }
13 // 获取正在加载的模块列表
14 Head = &(ldr->InLoadOrderModuleList);
15 //
16 Cur = Head->Flink;
17 do {
18 // 宏CONTAINING_RECORD根据结构体中某成员的地址来推算出该结构体整体的地址
19 ldmod = CONTAINING_RECORD(Cur, LDR_MODULE, InLoadOrderModuleList);
20 // 循环遍历，如果地址一致则表示找到对应模块来，就进行断链
21 if(hMod == ldmod->BaseAddress) {
22 // 断链原理很简单就是将属性交错替换
23 ldmod->InLoadOrderModuleList.Blink->Flink = ldmod->InLoadOrderModuleList.Flink;
24 ldmod->InLoadOrderModuleList.Flink->Blink = ldmod->InLoadOrderModuleList.Blink;
25
26 ldmod->InInitializationOrderModuleList.Blink->Flink = ldmod->InInitializationOrderModuleList.Flink;
27 ldmod->InInitializationOrderModuleList.Flink->Blink = ldmod->InInitializationOrderModuleList.Blink;
28
29 ldmod->InMemoryOrderModuleList.Blink->Flink = ldmod->InMemoryOrderModuleList.Flink;
30 ldmod->InMemoryOrderModuleList.Flink->Blink = ldmod->InMemoryOrderModuleList.Blink;
31 }
32 Cur = Cur->Flink;
33 } while (Head != Cur);
34 }

```

我们可以调用隐藏kernel32.dll这个模块，然后用DTDebug来查看一下：

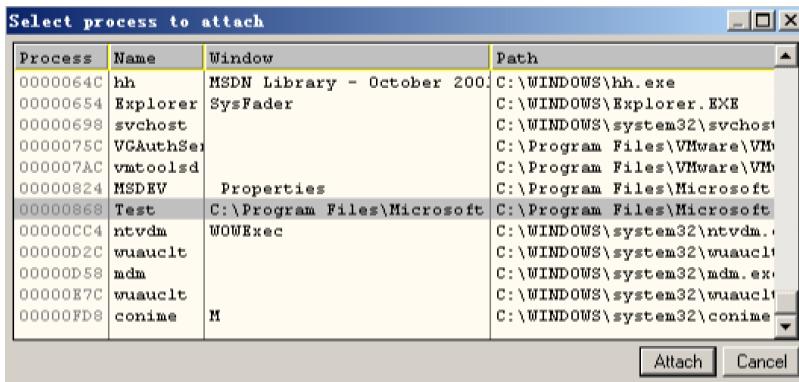
```

1 int main(int argc, char* argv[]) {
2 getchar();
3 HideModule("kernel32.dll");
4 getchar();
5 return 0;
6 }

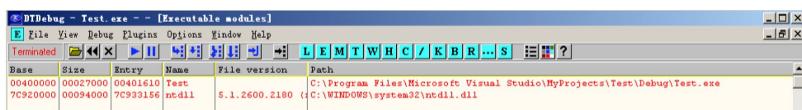
```

编译打开（不使用VC6打开）Test.exe然后使用DTDebug来Attach进程：





此刻我们是可以看见kernel32.dll模块的，但是当我们回车一下再来看就消失了：

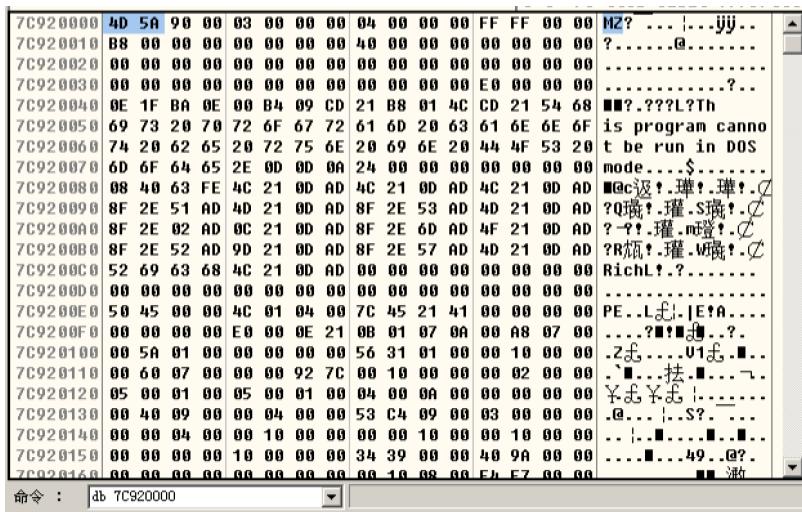


## 30.2 模块隐藏之PE指纹

首先我们来看某一个模块的PE指纹，这里就用ntdll.dll举例：



其地址是7c920000，我们在DTDebug中使用命令db 7c920000即可看到该模块的信息：



该模块开头两个字节是4D 5A，也就是MZ，当看见这两个字节后，在其位置向后找第64字节，发现是E0，那么就从模块起始位置0x7c920000加0xE0，这样就成了0x7c9200E0，然后我们找到对应地址的两个字节为50 45，也就是PE。

|          |              |       |       |       |       |           |       |       |                  |
|----------|--------------|-------|-------|-------|-------|-----------|-------|-------|------------------|
| 7C920000 | <b>4D 5A</b> | 90 00 | 03 00 | 00 00 | 04 00 | 00 00     | FF FF | 00 00 | MZ?... ...jj..   |
| 7C920010 | B8 00        | 00 00 | 00 00 | 00 00 | 40 00 | 00 00     | 00 00 | 00 00 | ?.....@.....     |
| 7C920020 | 00 00        | 00 00 | 00 00 | 00 00 | 00 00 | 00 00     | 00 00 | 00 00 | -----            |
| 7C920030 | 00 00        | 00 00 | 00 00 | 00 00 | 00 00 | <b>E0</b> | 00 00 | 00    | -----?..         |
| 7C920040 | 0E 1F        | BA 0E | 00 B4 | 09    | CD 21 | B8 01     | 4C CD | 21 54 | 68 M??.??!?!Th   |
| 7C920050 | 69 73        | 20 70 | 72 6F | 67 72 | 61 6D | 20 63     | 61 6E | 6E 6F | is program canno |
| 7C920060 | 74 20        | 62 65 | 20 72 | 75 6E | 20 69 | 6E 20     | 44 4F | 53 20 | t be run in DOS  |
| 7C920070 | 6D 6F        | 64 65 | 2E 0D | 0A 24 | 00 00 | 00 00     | 00 00 | 00 00 | mode....\$.....  |
| 7C920080 | 08 40        | 63 FE | 4C 21 | 0D AD | 4C 21 | 0D AD     | 4C 21 | 0D AD | Mac返!.瑾!.瑾!.Z    |
| 7C920090 | 8F 2E        | 51 AD | 4D 21 | 0D AD | 8F 2E | 53 AD     | 4D 21 | 0D AD | ?Q瑾!.瑾!.S瑾!.Z    |
| 7C9200A0 | 8F 2E        | 02 AD | 0C 21 | 0D AD | 8F 2E | 6D AD     | 4F 21 | 0D AD | ?-?!!.瑾!.W登!.Z   |
| 7C9200B0 | 8F 2E        | 52 AD | 9D 21 | 0D AD | 8F 2E | 57 AD     | 4D 21 | 0D AD | ?R桶!.瑾!.W瑾!.Z    |
| 7C9200C0 | 52 69        | 63 68 | 4C 21 | 0D AD | 00 00 | 00 00     | 00 00 | 00 00 | RichL!.?.....    |
| 7C9200D0 | 00 00        | 00 00 | 00 00 | 00 00 | 00 00 | 00 00     | 00 00 | 00 00 | -----            |
| 7C9200E0 | <b>50 45</b> | 00 00 | 4C 01 | 04 00 | 7C 45 | 21 41     | 00 00 | 00 00 | PE..L�!.IE!A.... |

这就是一个PE指纹，如果能满足这一套流程则表示这是一个模块。

### 30.3 模块隐藏之VAD树

这里涉及内核知识，建议观看视频简单讲解。

## 31 注入代码

最好的隐藏是无模块注入，也就是代码注入，将我们想要执行的代码注入进去。

### 31.1 注入代码的思路

我们可以将自定义函数复制到目标进程中，这样目标进程就可以执行我们想要执行的代码了，这就是注入代码的思路：



听起来很简单，但是其中有很多问题：

1. 你要将自定义函数复制到目标进程中，你复制的东西本质是什么？
2. 你复制过去就一定可以执行吗？前提条件是什么？

#### 31.1.1 机器码

首先我们来解决一下第一个问题，我们之前通过VC6是可以查看反汇编代码的，而实际上一个程序能看见具体的汇编代码吗？其实不可以，其表现形式应该是机器码，如下图所示左边是机器码，右边是机器码对应的汇编代码，我们能看见汇编代码是因为VC6的反汇编引擎将机器码转为汇编代码：

|                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 6: void Fun() { 7:     0040BF60 55     0040BF61 8B EC     0040BF63 83 EC 40     0040BF66 53     0040BF67 56     0040BF68 57     0040BF69 8D 7D C0     0040BF6C B9 10 00 00 00     0040BF71 B8 CC CC CC CC     0040BF76 F3 AB 7:     printf("Fun running...");     0040BF78 68 1C 00 42 00     0040BF7D E8 9E 52 FF FF     0040BF82 83 C4 04 8:     return; 9: } </pre> | <pre> push    ebp mov     ebp,esp sub    esp,40h push    ebx push    esi push    edi lea     edi,[ebp-40h] mov     ecx,10h mov     eax,0CCCCCCCCh rep    stos dword ptr [edi] push    offset string "*****\xb9\xA5\xBB\xF7***** \n" (00401220) call    printf (00401220) add    esp,4 </pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

所以我们拷贝过去的应该是机器码。

#### 31.1.2 前提条件

如下图所示，之间通过硬编码地址调用的机器码就没法注入执行，因为目标进程不可能会有目标地址内存给你进行使用：

```

8: void Fun() {
00401020 55 push ebp
00401021 8B EC mov ebp,esp
00401023 83 EC 40 sub esp,40h
00401026 53 push ebx
00401027 56 push esi
00401028 57 push edi
00401029 8D 7D C0 lea edi,[ebp-40h]
0040102C B9 10 00 00 00 mov ecx,10h
00401031 B8 CC CC CC CC mov eax,0CCCCCCCCCh
00401036 FE 3A rep stos dword ptr [edi]
9: g = 1;
00401038 C7 05 BC 35 42 00 01 mov dword ptr [g (004235bc)],1
10: printf("Fun running..."); offset string "Fun running..." (0042001c)
00401042 68 1C 00 42 00 push offset string "Fun running..." (0042001c)
00401047 E8 64 00 00 00 call printf (004010b0)
0040104C 83 C4 04 add esp,4
11: return;
12: }

```

## 31.2 复制代码的编写原则

1. 不能有全局变量
2. 不能使用常量字符串
3. 不能使用系统调用
4. 不能嵌套调用其他函数

## 31.3 传递参数

有这么多限制该怎么办？假设我们要将代码进程的代码拷贝过去，这段代码的作用就是创建文件，那么它得流程可以如下图所示：



首先将代码进程的ThreadProc复制过去，然后将复制过去之后目标进程的地址给到**CreateRemoteThread**函数，这样就解决了自定义函数的问题；

其次我们要创建文件的话必须要使用**CreateFile**函数，我们不能直接这样写，因为它依赖当前进程的导入表，当前进程和目标进程导入表的地址肯定是不一样的，所以不符合**复制代码的编写原则**；所以我们可以通过线程函数的参数来解决，我们先将所有用到的目标参数写到一个结构体中复制到目标进程，然后将目标进程结构体的地址作为线程函数的参数。

### 31.3.1 代码实现

如下是传递参数进行远程注入代码的实现：

```

1 #include <tlhelp32.h>
2 #include <stdio.h>
3 #include <windows.h>
4
5 typedef struct {
6 DWORD dwCreateAPIAddr; // Createfile函数的地址
7 LPCTSTR lpFileName; // 下面都是CreateFile所需要用到的参数
8 DWORD dwDesiredAccess;
9 DWORD dwShareMode;
10 LPSECURITY_ATTRIBUTES lpSecurityAttributes;
11 DWORD dwCreationDisposition;
12 DWORD dwFlagsAndAttributes;
13 HANDLE hTemplateFile;
14 } CREATEFILE_PARAM;
15
16 // 定义一个函数指针
17 typedef HANDLE(WINAPI* PFN_CreateFile) (
18 LPCTSTR lpFileName,
19 DWORD dwDesiredAccess,
20 DWORD dwShareMode,
21 LPSECURITY_ATTRIBUTES lpSecurityAttributes,
22 DWORD dwCreationDisposition,
23 DWORD dwFlagsAndAttributes,
24 HANDLE hTemplateFile
25);
26
27 // 编写要复制到目标进程的函数
28 DWORD __stdcall CreateFileThreadProc(LPVVOID lparam)
29 {
30 CREATEFILE_PARAM* Gcreate = (CREATEFILE_PARAM*)lparam;
31 PFN_CreateFile pfnCreateFile;
32 pfnCreateFile = (PFN_CreateFile)Gcreate->dwCreateAPIAddr;
33
34 // creatFile结构体全部参数
35 pfnCreateFile(
36 Gcreate->lpFileName,
37 Gcreate->dwDesiredAccess,
38 Gcreate->dwShareMode,
39 Gcreate->lpSecurityAttributes,
40 Gcreate->dwCreationDisposition,
41 Gcreate->dwFlagsAndAttributes,
42 Gcreate->hTemplateFile
43);
44
45 return 0;
46 }
47
48 // 远程创建文件
49 BOOL RemotCreateFile(DWORD dwProcessID, char* szFilePathName)
50 {
51 BOOL bRet;
52 DWORD dwThread;
53 HANDLE hProcess;
54 HANDLE hThread;
55 DWORD dwThreadFunSize;

```

```

56 CREATEFILE_PARAM GCreateFile;
57 LPVOID lpFilePathName;
58 LPVOID lpRemotThreadAddr;
59 LPVOID lpFileParamAddr;
60 DWORD dwFunAddr;
61 HMODULE hModule;
62
63
64 bRet = 0;
65 hProcess = 0;
66 dwThreadFunSize = 0x400;
67 // 1. 获取进程的句柄
68 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessID);
69 if (hProcess == NULL)
70 {
71 OutputDebugString("OpenProcessError! \n");
72 return FALSE;
73 }
74 // 2. 分配3段内存：存储参数，线程函数，文件名
75
76 // 2.1 用来存储文件名 +1是要计算到结尾处
77 lpFilePathName = VirtualAllocEx(hProcess, NULL, strlen(szFilePathName)+1, MEM_COMMIT,
78 PAGE_READWRITE); // 在指定的进程中分配内存
79
80 // 2.2 用来存储线程函数
81 lpRemotThreadAddr = VirtualAllocEx(hProcess, NULL, dwThreadFunSize, MEM_COMMIT,
82 PAGE_READWRITE); // 在指定的进程中分配内存
83
84 // 2.3 用来存储文件参数
85 lpFileParamAddr = VirtualAllocEx(hProcess, NULL, sizeof(CREATEFILE_PARAM), MEM_COMMIT,
86 PAGE_READWRITE); // 在指定的进程中分配内存
87
88
89 // 3. 初始化CreateFile参数
90 GCreateFile.dwDesiredAccess = GENERIC_READ | GENERIC_WRITE;
91 GCreateFile.dwShareMode = 0;
92 GCreateFile.lpSecurityAttributes = NULL;
93 GCreateFile.dwCreationDisposition = OPEN_ALWAYS;
94 GCreateFile.dwFlagsAndAttributes = FILE_ATTRIBUTE_NORMAL;
95 GCreateFile.hTemplateFile = NULL;
96
97 // 4. 获取CreateFile的地址
98 // 因为每个进程中的LoadLibrary函数都在Kernel32.dll中，而且此dll的物理页是共享的，所以我们进程中获得的
99 LoadLibrary地址和别的进程都是一样的
100 hModule = GetModuleHandle("kernel32.dll");
101 GCreateFile.dwCreateAPIAddr = (DWORD)GetProcAddress(hModule, "CreateFileA");
102 FreeLibrary(hModule);
103
104 // 5. 初始化CreateFile文件名
105 GCreateFile.lpFileName = (LPCTSTR)lpFilePathName;
106
107 // 6. 修改线程函数起始地址
108 dwFunAddr = (DWORD)CreateFileThreadProc;
109 // 间接跳
110 if ((*((BYTE*)dwFunAddr) == 0xE9)
111 {

```

```

108 dwFunAddr = dwFunAddr + 5 + *(DWORD*)(dwFunAddr + 1);
109 }
110
111 // 7. 开始复制
112 // 7.1 拷贝文件名
113 WriteProcessMemory(hProcess, lpFilePathName, szFilePathName, strlen(szFilePathName) + 1, 0);
114
115 // 7.2 拷贝线程函数
116 WriteProcessMemory(hProcess, lpRemotThreadAddr, (LPVOID)dwFunAddr, dwThreadFunSize, 0);
117
118 // 7.3 拷贝参数
119 WriteProcessMemory(hProcess, lpFileParamAddr, &GCreateFile, sizeof(CREATEFILE_PARAM), 0);
120
121 // 8. 创建远程线程
122 hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)lpRemotThreadAddr,
123 lpFileParamAddr, 0, &dwThread); // lpAllocAddr传给线程函数的参数.因为dll名字分配在内存中
124 if (hThread == NULL)
125 {
126 OutputDebugString("CreateRemoteThread Error! \n");
127 CloseHandle(hProcess);
128 CloseHandle(hModule);
129 return FALSE;
130 }
131
132 // 9. 关闭资源
133 CloseHandle(hProcess);
134 CloseHandle(hThread);
135 CloseHandle(hModule);
136 return TRUE;
137 }
138
139 // 根据进程名称获取进程ID
140 DWORD GetPID(char *szName)
141 {
142 HANDLE hProcessSnapShot = NULL;
143 PROCESSENTRY32 pe32 = {0};
144
145 hProcessSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
146 if (hProcessSnapShot == (HANDLE)-1)
147 {
148 return 0;
149 }
150
151 pe32.dwSize = sizeof(PROCESSENTRY32);
152 if (Process32First(hProcessSnapShot, &pe32))
153 {
154 do {
155 if (!strcmp(szName, pe32.szExeFile)) {
156 return (int)pe32.th32ProcessID;
157 }
158 } while (Process32Next(hProcessSnapShot, &pe32));
159 }
160 else
161 {
162 CloseHandle(hProcessSnapShot);
163 }
164 }
```

```
163 }
164 return 0;
165 }
166
167 int main()
168 {
169 RemotCreateFile(GetPID("进程名"), "文件名");
170 return 0;
171 }
```