

# 1 消息机制

我们在初级班对于Win32 API的学习中，知道可以通过向窗口发送消息实现交互，但是我们始终没有从本质上了解什么是消息机制，从而也就无法回答如下这些问题。

<1> 什么是窗口句柄?在哪里?有什么用?

<2> 什么是消息?什么是消息队列?消息队列在哪?

<3> 什么是窗口过程?窗口过程是谁调用的?没有消息循环窗口过程会执行吗?

<4> 为什么要有w32k.sys这个模块?

<5> 为什么只有使用图形界面的程序才可以访问KeServiceDescriptorTableShadow?

<6> 界面"卡死"的时候为什么鼠标还可以动?

因此为了弄清楚这些问题，我们就必须进入0环，从底层理解消息机制的本质。

## 1.1 消息队列

### 1.1.1 什么是消息队列

首先我们要理解什么是消息队列，我们可以编写运行如下代码，在桌面的左上角画上一个窗口，接着我们可以通过发送消息来与窗口进行互动：

```
1  #include <Windows.h>
2
3  typedef struct _Color
4  {
5      DWORD red;
6      DWORD green;
7      DWORD blue;
8  } Color;
9
10 typedef struct _WindowClass
11 {
12     DWORD x;
13     DWORD y;
14     DWORD width;
15     DWORD height;
16     Color color;
17 } WindowClass;
18
19 void PaintWindow(HDC hdc, WindowClass* window)
20 {
21     HBRUSH hBrush = (HBRUSH)GetStockObject(DC_BRUSH);
```

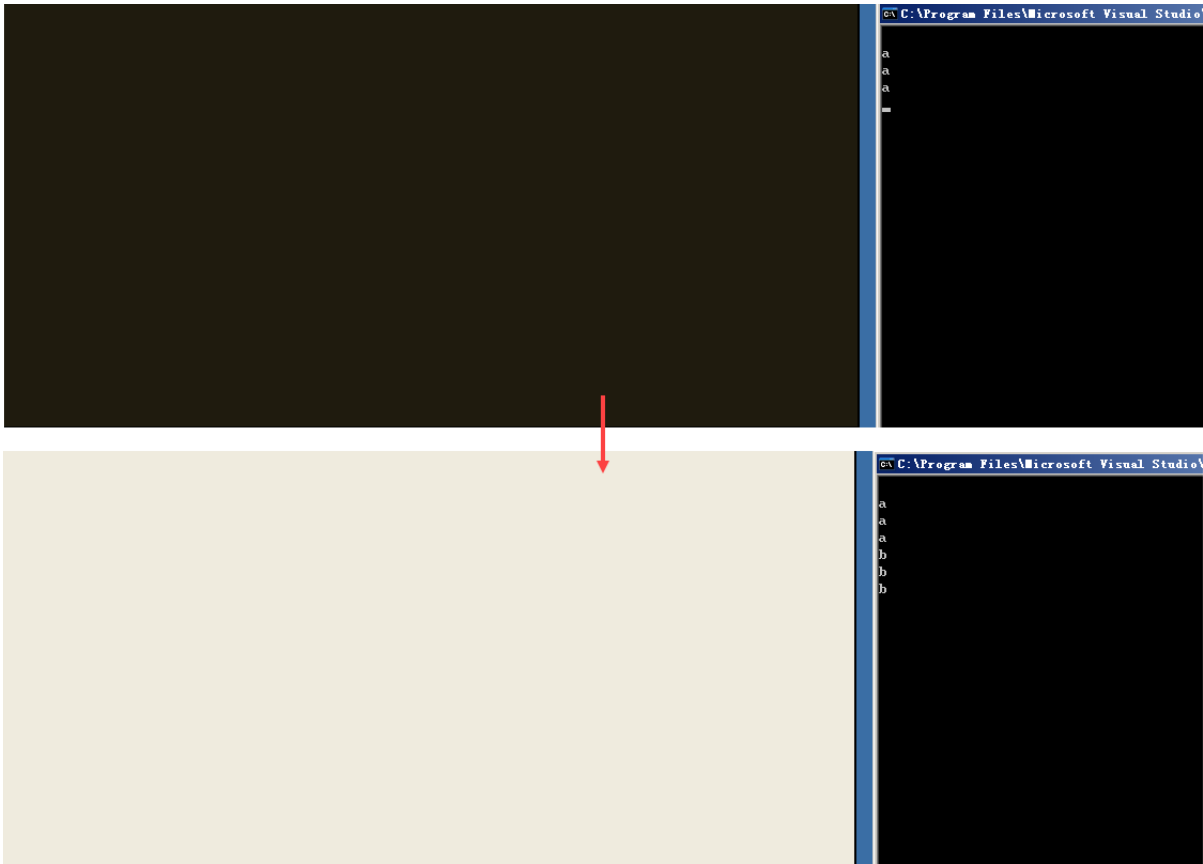
```

22     SelectObject(hdc, hBrush);
23     SetDCBrushColor(hdc, RGB(window->color.red, window->color.green,
window->color.blue));
24
25     MoveToEx(hdc, window->x, window->y, NULL);
26     LineTo(hdc, window->x + window->width, window->y);
27     LineTo(hdc, window->x + window->width, window->y + window->height);
28     LineTo(hdc, window->x, window->y + window->height);
29     LineTo(hdc, window->x, window->y);
30     Rectangle(hdc, window->x, window->y, window->x + window->width,
window->y + window->height + 1);
31
32     DeleteObject(hBrush);
33 }
34
35 int main()
36 {
37     char cMessage;
38     HWND hwnd;
39     HDC hdc;
40
41     WindowClass windowClass;
42     windowClass.x = 0;
43     windowClass.y = 0;
44     windowClass.width = 800;
45     windowClass.height = 400;
46     windowClass.color.red = 0xEF;
47     windowClass.color.green = 0xEB;
48     windowClass.color.blue = 0xDE;
49
50     hwnd = GetDesktopWindow();
51     hdc = GetWindowDC(hwnd);
52
53     for (;;)
54     {
55         PaintWindow(hdc, &windowClass);
56
57         cMessage = getchar();
58         switch (cMessage)
59         {
60             case 'a':
61                 windowClass.color.red += 0x10;
62                 windowClass.color.green += 0x10;
63                 windowClass.color.blue += 0x10;
64                 break;
65
66             case 'b':
67                 windowClass.color.red -= 0x10;
68                 windowClass.color.green -= 0x10;
69                 windowClass.color.blue -= 0x10;
70                 break;
71         }
72     }

```

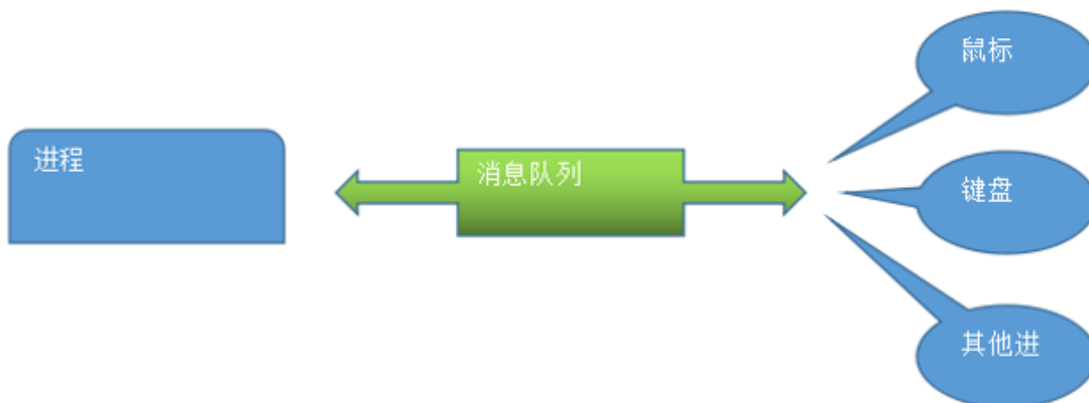
```
73  
74     getchar();  
75     return 0;  
76 }
```

编译运行这段代码，我们输入a或者b进行回车，一开始创建的窗口就会随着不同的指令进行颜色的切换。



这段代码是一个简单的交互程序，它通过接收键盘消息与窗口进行交互。然而，它有一个限制，即只能处理键盘消息，而无法处理鼠标或其他进程发来的消息。

因此我们希望接收并处理所有类型的消息，**就需要提供一个容器，即消息队列**。将所有消息都存放在消息队列中，进程再从消息队列中获取。

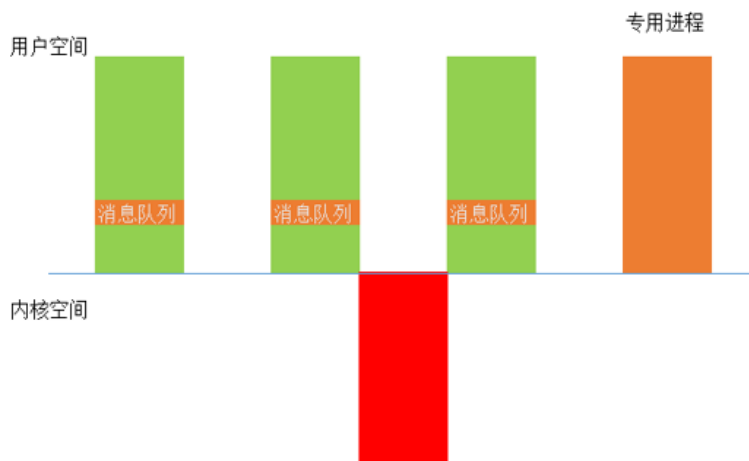


### 1.1.2 消息队列存放位置

#### 用户空间

如果消息队列存在于用户空间，也就表示每个进程有一个独属于自己的消息队列，那么就需要有一个专用进程来对不同类型的消息进行分发。

Linux操作系统就采用了一种类似的机制，通过单独的进程专门负责接收消息，并将消息发送给不同的进程进行处理。但这种方法需要进行频繁的跨进程通信，可能导致效率下降。



#### 内核空间

在Windows操作系统上微软采用了一种不同的策略。由于在0环（内核空间）中，不同进程的地址空间往往是相同的（即我们常说的高2G为共享内存），因此可以将消息队列存储在内核空间。

在线程结构体的0x130偏移位有个成员Win32Thread，该成员在线程调用图形界面相关函数时，会指向一个名为\_THREADINFO的结构体，该结构体中包含了消息队列。

```
kd> dt _KTHREAD
nt!_KTHREAD
+0x000 Header          : _DISPATCHER_HEADER
+0x010 MutantListHead  : _LIST_ENTRY
+0x018 InitialStack    : Ptr32 Void
+0x01c StackLimit      : Ptr32 Void
+0x020 Teb             : Ptr32 Void
+0x024 TlsArray        : Ptr32 Void
+0x028 KernelStack     : Ptr32 Void
...
+0x12b NextProcessor   : UChar
+0x12c CallbackStack   : Ptr32 Void
+0x130 Win32Thread     : Ptr32 Void
+0x134 TrapFrame       : Ptr32 _KTRAP_FRAME
+0x138 ApcStatePointer : [2] Ptr32 _KAPC_STATE
...
```

由于该结构体是未公开的，所以我们也只能在ReactOS代码 (<https://sourceforge.net/projects/reactos/>) 中找到，如下代码中的成员MessageQueue就是消息队列：

```

1  #ifdef __cplusplus
2  typedef struct _THREADINFO : _W32THREAD
3  {
4  #else
5  typedef struct _THREADINFO
6  {
7      W32THREAD;
8  #endif
9      PTL                                ptl;
10     PPROCESSINFO                       ppi;
11     struct _USER_MESSAGE_QUEUE* MessageQueue;
12     struct tagKL*                      KeyboardLayout;
13     struct _CLIENTTHREADINFO * pcli;
14     struct _DESKTOP*                  rpdesk;
15     struct _DESKTOPINFO * pDeskInfo;
16     struct _CLIENTINFO * pClientInfo;
17     FLONG                             TIF_flags;
18     PUNICODE_STRING                   pstrAppName;
19     struct _USER_SENT_MESSAGE * psmSent;
20     struct _USER_SENT_MESSAGE * psmCurrent;
21     /* Queue of messages sent to the queue. */
22     LIST_ENTRY                        SentMessagesListHead;    // psmsReceiveList
23     /* Last message time and ID */
24     LONG                             timeLast;
25     ULONG_PTR                        idLast;
26     /* True if a WM_QUIT message is pending. */
27     BOOLEAN                          QuitPosted;
28     /* The quit exit code. */
29     INT                              exitCode;
30     HDESK                            hdesk;
31     UINT                             cPaintsReady; /* Count of paints pending. */
32     UINT                             cTimersReady; /* Count of timers pending. */
33     struct tagMENUSTATE* pMenuState;
34     DWORD                           dwExpWinVer;
35     DWORD                           dwCompatFlags;
36     DWORD                           dwCompatFlags2;
37     struct _USER_MESSAGE_QUEUE* pqAttach;
38     PTHREADINFO                      ptiSibling;
39     ULONG                            fsHooks;
40     struct tagHOOK * sphkCurrent;
41     LPARAM                           lParamHkCurrent;
42     WPARAM                           wParamHkCurrent;
43     struct tagSBTRACK* pSBTrack;
44     /* Set if there are new messages specified by WakeMask in any of the
45     queues. */
45     HANDLE                           hEventQueueClient;
46     /* Handle for the above event (in the context of the process owning
47     the queue). */
47     PKEVENT                           pEventQueueServer;

```

```

48     LIST_ENTRY      PtiLink;
49     INT             iCursorLevel;
50     /* Last message cursor position */
51     POINT           ptLast;
52
53     INT             cEnterCount;
54     /* Queue of messages posted to the queue. */
55     LIST_ENTRY      PostedMessagesListHead; // m!Post
56     WORD            fsChangeBitsRemoved;
57     WCHAR           wchInjected;
58     UINT            cWindows;
59     UINT            cVisWindows;
60 #ifndef __cplusplus /// FIXME!
61     LIST_ENTRY      aphkStart[NB_HOOKS];
62     CLIENTTHREADINFO cti; // Used only when no Desktop or pcti NULL.
63
64     /* ReactOS */
65
66     /* Thread Queue state tracking */
67     // Send list QS_SENDMESSAGE
68     // Post list QS_POSTMESSAGE|QS_HOTKEY|QS_PAINT|QS_TIMER|QS_KEY
69     // Hard list QS_MOUSE|QS_KEY only
70     // Accounting of queue bit sets, the rest are flags. QS_TIMER QS_PAINT
71     counts are handled in thread information.
72     DWORD nCntsQBits[QSIDCOUNTS]; // QS_KEY QS_MOUSEMOVE QS_MOUSEBUTTON
73     QS_POSTMESSAGE QS_SENDMESSAGE QS_HOTKEY
74
75     LIST_ENTRY WindowListHead;
76     LIST_ENTRY W32CallbackListHead;
77     SINGLE_LIST_ENTRY ReferencesList;
78     ULONG cExclusiveLocks;
79 #if DBG
80     USHORT acExclusiveLockCount[GDIObjTypeTotal + 1];
81 #endif
82 #endif // __cplusplus
83 } THREADINFO;

```

在所有线程刚创建时，它们都是普通线程，可以通过使用\_KTHREAD.ServiceTable，可以找到一张表KeServiceDescriptorTable。然而，当线程首次调用Win32k.sys（在0环中实现图形界面API）时，将调用PsConvertToGuiThread函数，该函数执行以下几个主要步骤：

1. 扩展内核栈大小为64KB，因为普通内核栈只有12KB大小；
2. 创建一个带有消息队列的结构体，并将其挂接到\_KTHREAD结构体Win32Thread成员上；
3. 将Thread.ServiceTable指向KeServiceDescriptorTableShadow，此时两个表都可见；
4. 将所需的内存数据映射到当前进程的地址空间中。

### 1.1.3 总结

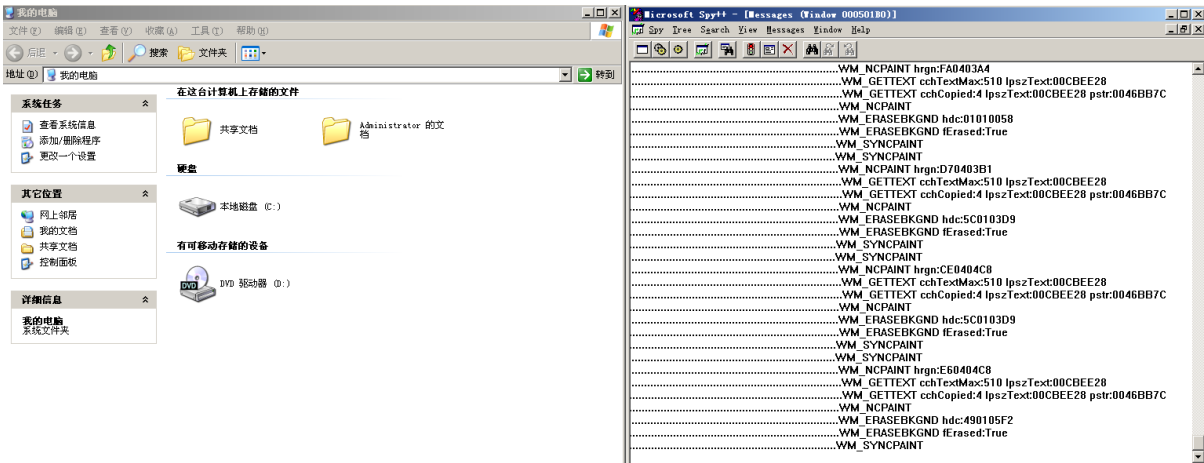
1. 在Windows操作系统中，消息队列存储在0环（内核空间）中，并可以通过KTHREAD.Win32Thread找到。
2. 并非所有线程都需要消息队列，只有GUI线程才会拥有消息队列。
3. 每个GUI线程有且只有一个消息队列。

## 1.2 窗口与线程

了解消息队列与线程关系后，我们需要知道消息是从哪里来，又到哪里去，是谁来做这些消息传递的。

### 1.2.1 消息来源

如下图所示，我们可以通过VC++6.0的工具Spy++用于捕捉窗口接收到的消息。当鼠标在窗口上移动、点击或键盘敲击时，就会产生消息。

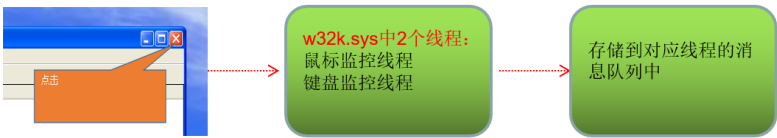


除了键盘、鼠标外，消息还来自于其他进程，假设A进程使用了CreateWindow创建窗口，就会获得一个窗口句柄，窗口句柄具有全局特性（所有窗口对象都存在一张公共表中），这意味着一旦其他进程获取到窗口句柄，都可以利用SendMessage或PostMessage函数向A进程创建的窗口发送消息以进行交互。

### 1.2.2 消息去处

当一个消息产生时，肯定是需要通过消息监控来知道由消息产生了，然后再由监控的程序存储到窗口对应线程的消息队列中。

在Windows上0环（内核空间）Win32k.sys的2个线程分别对鼠标、键盘进行消息的监控，具体的我们可以看下如下这个函数InitInputImpl，在初始化Win32k.sys的服务时，会调用它来创建2个线程：KeyboardThreadMain、MouseThreadMain，也就是鼠标、键盘的监控线程。



```
1 NTSTATUS FASTCALL InitInputImpl(VOID)
2 {
3     NTSTATUS Status;
4     KeInitializeEvent(&InputThreadsStart, NotificationEvent, FALSE);
5     MasterTimer = ExAllocatePoolWithTag(NonPagedPool, sizeof(KTIMER),
6     TAG_INPUT);
7     KeInitializeTimer(MasterTimer);
```

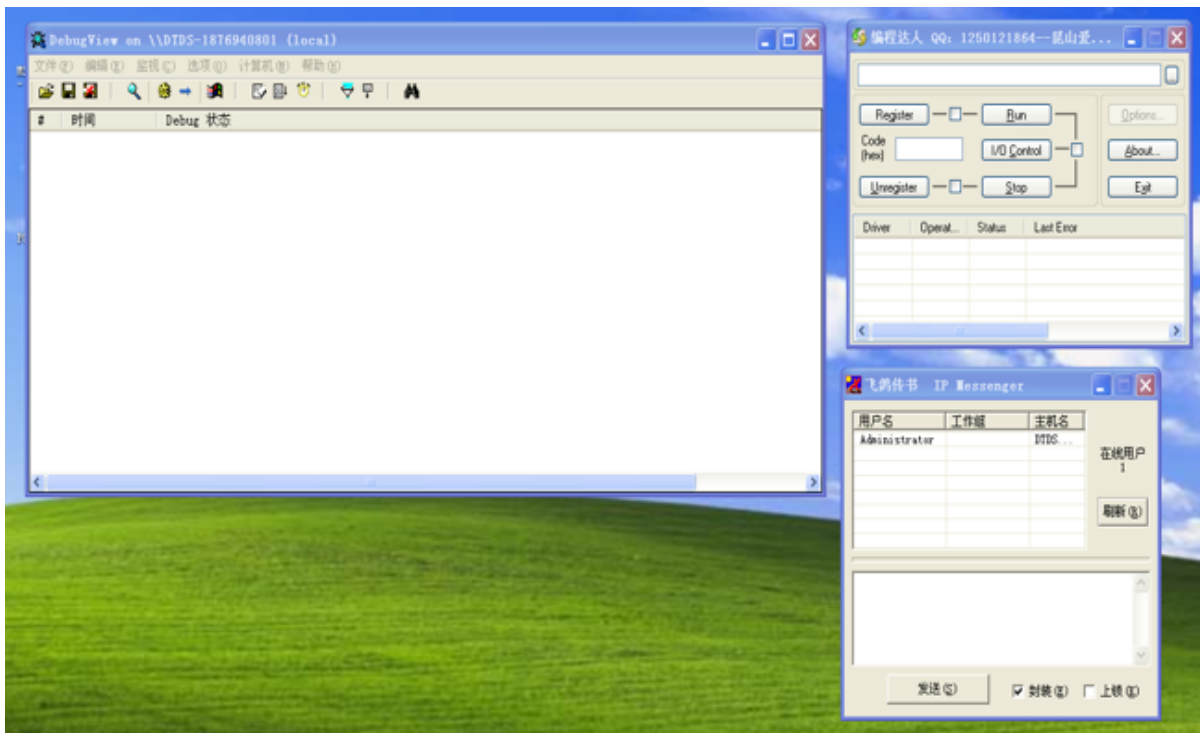
```
7     Status =  
8     PsCreateSystemThread(&RawInputThreadHandle, THREAD_ALL_ACCESS, NULL, NULL,  
9  
10    &RawInputThreadId, RawInputThreadMain, NULL);  
11  
12    // 键盘输入线程: KeyboardThreadMain  
13    Status =  
14    PsCreateSystemThread(&KeyboardThreadHandle, THREAD_ALL_ACCESS, NULL, NULL,  
15  
16    &KeyboardThreadId, KeyboardThreadMain, NULL);  
17  
18    // 鼠标输入线程: MouseThreadMain  
19    Status =  
20    PsCreateSystemThread(&MouseThreadHandle, THREAD_ALL_ACCESS, NULL, NULL,  
21  
22    &MouseThreadId, MouseThreadMain, NULL);  
  
23    InputThreadsRunning = TRUE; // TRUE表示现在可以开始读取键盘鼠标输入  
24    KeSetEvent(&InputThreadsStart, IO_NO_INCREMENT, FALSE);  
25  
26    return STATUS_SUCCESS;  
27 }
```

如上代码也就解释了为什么有的时候程序卡死，但鼠标仍然可以移动，这是因为鼠标操作运行在独立的线程中。

### 1.2.3 消息队列的寻找

如下图所示，在打开了三个窗口的情况下，当鼠标进行点击和移动操作时，操作系统是如何准确地将消息发送给不同窗口所对应的消息队列呢？





首先，图中不仅有三个窗口，每个进程的窗口内部的按钮、表格等也都是窗口的一部分。因此，一个进程可以拥有多个窗口，但这些窗口只能属于同一个进程。

接着，我们可以看一下创建窗口的API到底是怎么调用的：CreateWindow - CreateWindowA/W - CreateWindowEx - VerNtUserCreateWindowEx - NtUserCreateWindow - NtUserCreateWindow。

我们会发现创建窗口最终还是会通过系统调用到0环去，根据系统调用号（从0位开始，第12位为1）我们就知道它最终调用的就是Win32k.sys提供的服务。

```
.text:77D2E430      push    [ebp+arg_10]
.text:77D2E433      push    [ebp+arg_C]
.text:77D2E436      push    [ebp+Str]
.text:77D2E439      push    edi
.text:77D2E43A      push    dword ptr [ebp+8]
.text:77D2E43D      call    VerNtUserCreateWindowEx@52 ; VerNtUserCreateWindowEx(x,x,x,x,x,x,x,x,x,x,x,x,x,x,x)
.text:77D2E442      mov     esi, eax
.text:77D2E444      test    esi, esi
.text:77D2E446      jz      short loc_77D2E452
.text:77D2E448      cmp     [ebp+var_8], 0
.text:77D2E44C      jnz     loc_77D3F924

↓

.text:77D2E346      push    esi
.text:77D2E347      push    [ebp+arg_0]
.text:77D2E34A      call    NtUserCreateWindowEx@60 ; NtUserCreateWindowEx(x,x,x,x,x,x,x,x,x,x,x,x,x,x,x)
.text:77D2E34F      mov     ebx, eax
.text:77D2E351      xor     edi, edi
.text:77D2E353      cmp     ebx, edi
.text:77D2E355      jz      loc_77D4A659

↓

.text:77D2E370      ; __stdcall NtUserCreateWindowEx(x, x, x, x, x, x, x, x, x, x, x, x, x, x, x)
.text:77D2E370      _NtUserCreateWindowEx@60 proc near ; CODE XREF: VerNtUserCreateWindowEx(x,x,x,x,x,x,x,x,x,x,x,x,x)+1581p
.text:77D2E370      mov     eax, 1157h
.text:77D2E382      mov     edx, 7FFE0300h
.text:77D2E387      call    dword ptr [edx]
.text:77D2E389      retn    3Ch
.text:77D2E389      NtUserCreateWindowEx@60 endp
```

因此，窗口就像进程和线程一样，实际上是一个处于0环的结构。窗口也有与之对应的内核结构体，即 **\_WINDOW\_OBJECT**。（该结构体没有通过符号表导出，可以查看ReactOS的代码来查看该结构体）

在窗口对象 **\_WINDOW\_OBJECT** 中，存在一个名为 **pti** 的成员，其类型为 **\_PTHREADINFO**，指向 **\_THREADINFO** 结构体。这个 **\_THREADINFO** 结构体正是前文提到的 **\_KTHREAD.Win32Thread** 相关联的结构体。**通过这种方式，就可以将线程与窗口联系在一起。**

在初始状态下，**\_HREAD.Win32Thread** 指向的值为空。然而，当线程调用 **Win32k.sys** 中的函数创建一个窗口时，**\_KTHREAD.Win32Thread** 将指向 **\_THREADINFO** 结构体，从而将该线程由普通线程转变为 GUI 线程。此时，窗口对象对应的内核结构体 **\_WINDOW\_OBJECT** 中的 **pti** 成员也会指向这个 **\_THREADINFO** 结构体。**而消息队列则位于 **\_THREADINFO** 结构体中，这就使得窗口可以访问所属线程的消息队列。**

## 1.3 消息的接收

### 1.3.1 窗口的创建过程

在3环创建窗口时，首先需要创建和注册一个窗口类对象，并注册和设置窗口的样式和过程函数。然后，通过调用 **CreateWindow** 函数来创建窗口。

```
//创建窗口类的对象
WNDCLASS wndclass = {0};
//窗口的背景色
wndclass.hbrBackground = (HBRUSH)COLOR_MENU;
wndclass.hCursor = LoadCursor(NULL, IDC_APPSTARTING);
//窗口过程函数
wndclass.lpfnWndProc = WindowProc;
...

// 创建窗口
CreateWindow(
    className,           //类名
    "我的第一个窗口",    //窗口标题
    WS_OVERLAPPEDWINDOW, //窗口外观样式
    10,                  //相对于父窗口的X坐标
    10,                  //相对于父窗口的Y坐标
    600,                 //窗口的宽度
    300,                 //窗口的高度
    NULL,                //父窗口句柄，为NULL
    NULL,                //菜单句柄，为NULL
    hInstance,           //当前应用程序的句柄
    NULL);               //附加数据一般为NULL
```

```
_WINDOW_OBJECT结构
PTHREADINFO pti;
PWND Wnd;

...
DWORD ExStyle;
//扩展样式
DWORD style;
//标准样式

...
//窗口过程
WNDPROC lpfnWndProc;
```

实质上，**CreateWindow** 只是一个3环的接口，最终调用的是位于 **Win32k.sys** 中的0环函数。在0环中，会为窗口创建一个名为 **\_WINDOW\_OBJECT** 的结构体，每个窗口都有一个这样的结构体。

### 1.3.2 消息队列的结构

当线程调用 **Win32k.sys** 提供的图形界面函数时，线程结构体 **\_KTHREAD** 中的成员 **Win32Thread** 会指向一个名为 **\_THREADINFO** 的结构体。在该结构体中有一个成员 **MessageQueue**，即消息队列，其中包含7组队列（仅适用于旧版ReactOS），用于处理不同类型的消息。如下3个是比较常见的消息队列：

1. **SentMessagesListHead**：接到 **SendMessage** 发来的消息。
2. **PostedMessagesListHead**：接到 **PostMessage** 发来的消息。
3. **HardwareMessagesListHead**：接到鼠标、键盘的消息。

### 1.3.3 GetMessage

我们在窗口创建后需要使用GetMessage、TranslateMessage和DispatchMessage来获取、转换和分发消息。如下代码所示，我们通常会这样去写：

```

1  MSG msg;
2  BOOL bRet;
3  while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
4  {
5      if (bRet == -1)
6      {
7          // handle the error and possibly exit
8          sprintf(szOutBuff, "Error: %d", GetLastError());
9          OutputDebugString(szOutBuff);
10         return 0;
11     }
12     else
13     {
14         // 转换消息
15         TranslateMessage(&msg);
16         // 分发消息：就是给系统调用窗口处理函数
17         DispatchMessage(&msg);
18     }
19 }

```

GetMessage表面上的意思是获取消息，但实际功能不仅限于此，我们首先来看下该函数的语法：

```

1  BOOL GetMessage(
2      LPMSG lpMsg,           // 返回从队列中获取的消息
3      HWND hWnd,            // 过滤条件：指定接收消息的窗口
4      UINT wMsgFilterMin,    // 过滤条件
5      UINT wMsgFilterMax     // 过滤条件
6  );

```

GetMessage函数有4个参数，其后3个参数是过滤条件，第一个条件是用于指定接收消息的窗口。而第一个参数则是从消息队列中获取的消息。

GetMessage函数通过循环判断是否存在该窗口的消息，如果有，将消息存储到MSG结构体中，并从原始消息队列中删除该消息。然后，将消息传递给TranslateMessage和DispatchMessage函数进行处理。

我们可以在代码中，将TranslateMessage和DispatchMessage注释掉，来看一下没有这两个函数进行消息的转换和分发，我们的窗口过程函数WindowProc是否仍然可以执行，接着另外一个程序发送消息：

```

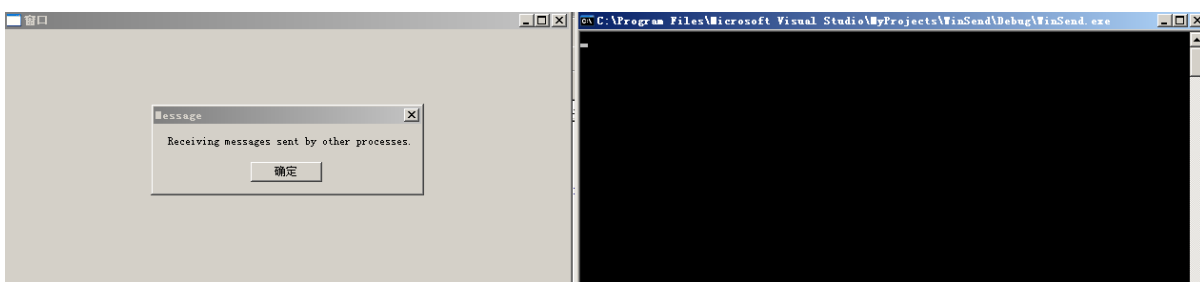
MSG msg;
BOOL bRet;
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        sprintf(szOutBuff, "Error: %d", GetLastError());
        OutputDebugString(szOutBuff);
        return 0;
    }
    else
    {
        // TranslateMessage(&msg);
        // DispatchMessage(&msg);
    }
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch(uMsg) {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            return 0;
        }
        case 0x401:
        {
            ::MessageBox(NULL, "Receiving messages sent by other processes.", "Message", 0);
            return 0;
        }
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

void main()
{
    HWND hWnd = ::FindWindow(TEXT("My First Window"), TEXT("窗口"));
    if (hWnd != NULL)
    {
        ::SendMessage(hWnd, 0x401, 0, 0);
    }
}

```

我们运行这个窗口程序后再运行发送消息的程序，会发现窗口成功接收到了消息，并执行了对应的处理函数：



DispatchMessage函数用于将消息转发到窗口过程函数，以触发相应的处理逻辑。但是在这里我们通过实验发现GetMessage函数也会对消息进行处理。

GetMessage函数调用的是Win32k.sys中的NtUserGetMessage函数，在该函数内部有如下的大致逻辑：

```

1  do
2  {
3      // 先判断SentMessagesListHead是否有消息，如果有就处理掉
4      do
5      {
6          ....
7          KeUserModeCallback(USER32_CALLBACK_WINDOWPROC,
8                              Arguments,
9                              ArgumentLength,
10                             &ResultPointer,
11                             &ResultLength);
12         ....
13     } while (SentMessagesListHead != NULL)
14     // 依次判断其他的6个队列，里面如果有消息就返回，没有则继续
15 } while (其他队列 != NULL)

```

在一个内部的do...while循环中，NtUserGetMessage首先会判断SentMessagesListHead中是否存在消息，如果有，则调用窗口回调函数处理它。然后，在处理完SentMessagesListHead中的所有消息之后，才会考虑其他六个队列中的消息。在这种情况下，将不会处理这些消息，而是直接将它们返回。因此，**GetMessage也会对消息进行处理，但只会处理SentMessagesListHead中的消息。**

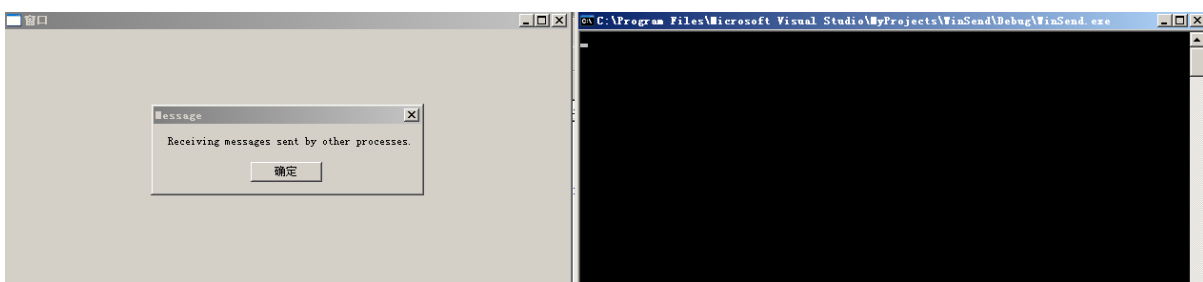
那么也就表示如果我们刚刚发送消息的代码使用的函数从SendMessage变成PostMessage，在没有TranslateMessage和DispatchMessage函数的情况下，GetMessage函数并不会处理这个消息。

### 1.3.4 SendMessage与PostMessage

接着我们来看一下这两个函数：SendMessage与PostMessage，这两者都是用于发送消息的，但不同之处在于前者是同步发送消息，后者是异步发送消息。

使用SendMessage函数发送消息，当GetMessage函数接收消息时，它会进入0环遍历SentMessagesListHead消息队列以检查是否有消息。如果有消息，则会进行处理；如果没有消息，则会立即返回。在有消息的情况下，必须完全处理完消息才能返回，否则SendMessage函数会一直阻塞在这里，直到接收到对方的执行结果并返回。

如下图所示，我们的消息发送程序使用SendMessage函数发送消息，当窗口程序的处理没有结束，即弹窗没有关闭，则消息发送程序一直停留在那里，也不会关闭。

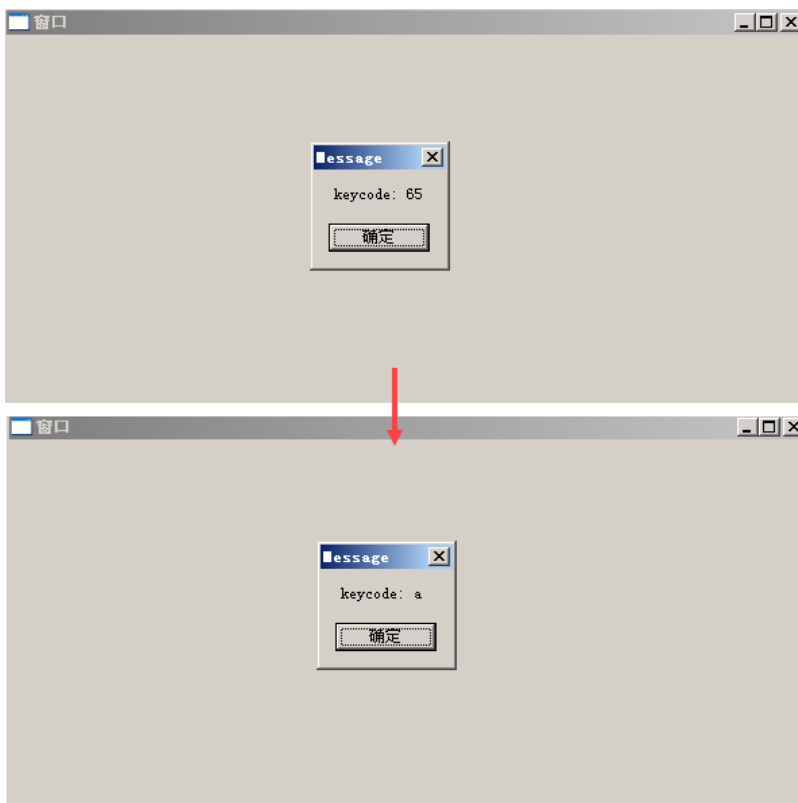


相比之下，当我们使用PostMessage函数发送消息时，GetMessage函数只是接收该消息，而不会进行处理。消息的处理由TranslateMessage和DispatchMessage函数负责。**PostMessage函数不会等待对方返回处理结果，一旦发送完成就立即结束自身程序。**

## 1.4 消息的转换

我们再来看一下消息的转换，即TranslateMessage函数，该函数是针对键盘类消息的一种优化。如果没有使用它，键盘消息将属于WM\_KEYDOWN类型，并以ASCII对应的十进制值打印出来。但是，通过使用它，键盘消息将被转换为WM\_CHAR类型，并打印出按下的键盘符号。因此，TranslateMessage函数的使用与否对结果影响不大，它只是对消息类型进行转换。

我们可以添加对WM\_KEYDOWN、WM\_CHAR消息的处理，并且注释掉TranslateMessage函数和不注释掉进行对比，就会发现这一特征，需要主要的是在实际使用过程中**如果你使用了TranslateMessage函数就只需要添加对WM\_KEYDOWN消息的处理**，不需要再对WM\_CHAR消息进行处理，否则就会造成重复处理：



## 1.5 消息的分发

所谓消息的分发，核心点在于DispatchMessage函数，它会根据窗口句柄调用相关的窗口过程函数。在上一节的学习中我们了解到GetMessage函数除了接收消息还会处理SendMessage发送过来的消息，也就是SentMessagesListHead消息队列中的消息。

也就表示其他消息是交由DispatchMessage函数来处理的，DispatchMessage函数最终调用的是Win32k.sys中的NtUserDispatchMessage函数。该函数主要做了以下两件事情：

1. 根据窗口句柄找到窗口对象\_WINDOW\_OBJECT；（UserGetWindowObject函数）
2. 根据窗口对象获取窗口过程函数，并通过0环发起调用。

如下图所示，我们可以看见DispatchMessage函数与GetMessage函数（NtUserGetMessage函数）一样，最终都是调用KeUserModeCallback的回调函数进入3环，再去调用窗口过程函数。

```

LRESULT APIENTRY
NtUserDispatchMessage(PMSG UnsafeMsgInfo)
{
    LRESULT Res = 0;
    MSG SafeMsg;

    _SEH2_TRY
    {
        ProbeForRead(UnsafeMsgInfo, sizeof(MSG), 1);
        RtlCopyMemory(&SafeMsg, UnsafeMsgInfo, sizeof(MSG)); {
            _SEH2_EXCEPT(EXCEPTION_EXECUTE_HANDLER)
            {
                SetLastNtError(_SEH2_GetExceptionCode());
                _SEH2_YIELD(return FALSE);
            }
        }
        _SEH2_END;

        UserEnterExclusive();

        Res = IntDispatchMessage(&SafeMsg);

        UserLeave();
        return Res;
    } « end NtUserDispatchMessage »
}

LRESULT FASTCALL
IntDispatchMessage(PMSG pMsg)
{
    LARGE_INTEGER TickCount;
    LONG Time;
    LRESULT retval = 0;
    PTHREADINFO pti;
    PWND Window = NULL;
    BOOL DoCallBack = TRUE;

    if (pMsg->hwnd)
    {
        Window = UserGetWindowObject(pMsg->hwnd);
        if (!Window) return 0;
    }

    pti = PsGetCurrentThreadWin32Thread();

    if (Window && Window->head.pti != pti)
    {
        EngSetLastError( ERROR_MESSAGE_SYNC_ONLY );
        return 0;
    }

    if (DoCallBack)
    {
        retval = co_IntCallWindowProc( Window->lpfnWndProc,
            !Window->Unicode,
            pMsg->hwnd,
            pMsg->message,
            pMsg->wParam,
            pMsg->lParam,
            -1);
    }
}

LRESULT APIENTRY
co_IntCallWindowProc(WNDPROC Proc,
    BOOLEAN IsAnsiProc,
    HWND Wnd,
    UINT Message,
    WPARAM wParam,
    LPARAM lParam,
    INT lParamBufferSize)
{
    WINDOWPROC_CALLBACK_ARGUMENTS StackArguments;
    PWINDOWPROC_CALLBACK_ARGUMENTS Arguments;
    NTSTATUS Status;
    PVOID ResultPointer, pActCtx;
    PWND pWnd;
    ULONG ResultLength;
    ULONG ArgumentLength;
    LRESULT Result;
    ...
    IntSetTebWndCallback (&Wnd, &pWnd, &pActCtx);
    UserLeaveCo();

    Status = KeUserModeCallback(USER32_CALLBACK_WINDOWPROC,
        Arguments,
        ArgumentLength,
        &ResultPointer,
        &ResultLength);
}

```

并且我们会发现即使DispatchMessage函数的唯一参数MSG结构体，发挥了巨大的作用，该结构体里有窗口句柄、消息类型、消息参数等内容，具体的我们看如下定义。

```

1  typedef struct tagMSG {
2      HWND      hwnd;      // 窗口句柄
3      UINT      message;   // 消息类型
4      WPARAM    wParam;    // 消息参数
5      LPARAM    lParam;    // 消息参数
6      DWORD     time;      // 消息时间戳
7      POINT     pt;        // 鼠标坐标
8      #ifdef _MAC
9          DWORD     lPrivate;
10     #endif
11     } MSG, *PMSG, NEAR *NPMMSG, FAR *LPMSG;

```

### 1.5.1 默认的消息处理函数

消息无时无刻都在产生，我们在自定义窗口过程函数时候只需要对关注的消息进行处理，其他的我们都可以交给默认的消息处理函数，即DefWindowProc。

```
1  LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM  
2      lParam) {  
3      // 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  
4      return DefWindowProc(hwnd, uMsg, wParam, lParam);  
}
```

## 1.6 内核回调机制

### 1.6.1 窗口过程的调用者


我们知道 GetMessage 函数再处理 SentMessagesListHead 消息队列中的消息时，以及 DispatchMessage 在处理其他消息队列中的消息时，都会调用窗口过程函数。**除此之外内核代码也会调用窗口过程函数。**

我们可以注释如下图所示的代码，并且在窗口过程函数中加入一个输出，再编译运行窗口会发现，即使没有 GetMessage、DispatchMessage 函数，窗口过程函数仍然会被执行。



```

/*
MSG msg;
BOOL bRet;
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        sprintf(szOutBuff, "Error: %d", GetLastError());
        OutputDebugString(szOutBuff);
        return 0;
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
*/

LRESULT CALLBACK WindowProc(HWND hwnd,  UINT uMsg, WPARAM wParam, LPARAM lParam) {
    char szOutBuff[0x80];
    switch(uMsg) {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_KEYDOWN:
        {
            sprintf(szOutBuff, "keycode: %d \n", wParam);
            ::MessageBox(NULL, szOutBuff, "Message", 0);
            break;
        }
    }
    sprintf(szOutBuff, "Message: %d \n", uMsg);
    OutputDebugString(szOutBuff);
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}


```

```

Loaded 'C:\WINDOWS\system32\version.dll', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\MSCTIME.IME', no matching symbolic information found.
Loaded 'C:\WINDOWS\system32\ole32.dll', no matching symbolic information found.
Message: 131
Message: 1
Message: 24
Message: 70
Message: 70
Message: 28
Message: 134
Message: 13
Message: 6
Message: 641
Message: 642
Message: 7
Message: 133
Message: 13
Message: 20
Message: 71
Message: 5
Message: 3

```

那么为什么内核代码需要调用窗口过程函数呢？假设有一个需求，当我们想要在窗口创建时去做一些事情时，由于窗口创建时是肯定没有消息的，因此**GetMessage函数也没法获取到消息，DispatchMessage函数自然也就无法分发消息。**

所以只能借助于内核代码来调用回调函数，这里实际上就是CreateWindow发挥的作用，其进入0环调用的NtUserCreateWindowEx函数，该函数在窗口创建之前通过调用内核回调函数向窗口发送消息，这些消息不会进入消息队列，而是直接发送给窗口过程函数，**消息类型为WM\_CREATE。**

### 1.6.2 0到3跨环调用

从0环调用3环函数的有三种方法：

**用户APC的执行：**用户APC（Asynchronous Procedure Call）是一种用于异步执行用户模式代码的机制。在这种情况下，内核可以将用户模式函数作为一个APC请求提交给目标线程，并在目标线程处于用户模式执行时，将该函数插入到目标线程的执行流中。

**用户异常的处理：**当内核调试器和用户调试器均不存在或不处理时，如果发生用户模式异常，处理流程将从Ring 0（内核模式）转移到Ring 3（用户模式）。在这种情况下，操作系统会将异常传递给目标进程的异常处理程序，由用户模式代码处理该异常。

**内核回调：**内核回调是指在Ring 0的代码中调用窗口过程函数。这种调用是通过内核提供的机制实现的，允许Ring 0的代码向特定窗口的过程函数发送消息。通过这种方式，Ring 0的代码可以与用户模式的窗口过程进行交互，以实现特定的功能或处理特定事件。这种机制通常由操作系统提供，如SendMessage或PostMessage函数。

### 1.6.3 KeUserModeCallback

内核回调机制中，0环是通过KeUserModeCallback函数来调用3环函数的。在之前我们的分析NtUserDispatchMessage函数的调用时（NtUserDispatchMessage->IntDispatchMessage->co\_IntCallWindowProc->KeUserModeCallback）也知道它会去调用KeUserModeCallback函数。

```

LRESULT APIENTRY
NtUserDispatchMessage(PMSG UnsafeMsgInfo)
{
    LRESULT Res = 0;
    MSG SafeMsg;

    _SEH2_TRY
    {
        ProbeForRead(UnsafeMsgInfo, sizeof(MSG), 1);
        RtlCopyMemory(&SafeMsg, UnsafeMsgInfo, sizeof(MSG)); {
    _SEH2_EXCEPT(EXCEPTION_EXECUTE_HANDLER)
    {
        SetLastNtError(_SEH2_GetExceptionCode());
        _SEH2_YIELD(return FALSE);
    }
    _SEH2_END;

    UserEnterExclusive();

    Res = IntDispatchMessage(&SafeMsg);

    UserLeave();
    return Res;
} « end NtUserDispatchMessage »

LRESULT FASTCALL
IntDispatchMessage(PMSG pMsg)
{
    LARGE_INTEGER TickCount;
    LONG Time;
    LRESULT retval = 0;
    PTHREADINFO pti;
    PWND Window = NULL;
    BOOL DoCallBack = TRUE;

    if (pMsg->hwnd)
    {
        Window = UserGetWindowObject(pMsg->hwnd);
        if (!Window) return 0;
    }

    pti = PsGetCurrentThreadWin32Thread();

    if (Window && Window->head.pti != pti)
    {
        EngSetLastError( ERROR_MESSAGE_SYNC_ONLY );
        return 0;
    }

    if (DoCallBack)
    {
        retval = co_IntCallWindowProc( Window->lpfnWndProc,
            !Window->Unicode,
            pMsg->hwnd,
            pMsg->message,
            pMsg->wParam,
            pMsg->lParam,
            -1);
    }
}

```

Diagram illustrating the flow of message dispatching:

- `NtUserDispatchMessage` calls `IntDispatchMessage` (indicated by a red arrow).
- `IntDispatchMessage` calls `co_IntCallWindowProc` (indicated by a red arrow).
- `co_IntCallWindowProc` calls `KeUserModeCallback` (indicated by a red arrow).

KeUserModeCallback函数的语法格式如下：

```

1  NTSTATUS NTAPI KeUserModeCallback(
2      IN ULONG      RoutineIndex,
3      IN PVOID      Argument,
4      IN ULONG      ArgumentLength,
5      OUT PVOID *    Result,
6      OUT PULONG     ResultLength
7  )

```

其有5个参数，其中Argument是提供参数的，其中包含窗口过程函数、窗口句柄、消息类型、消息参数等等内容，这些参数也就对上在3环窗口过程函数所需要的参数了。

```

Arguments->Proc = Proc;
Arguments->IsAnsiProc = IsAnsiProc;
Arguments->Wnd = Wnd;
Arguments->Msg = Message;
Arguments->wParam = wParam;
Arguments->lParam = lParam;
Arguments->lParamBufferSize = lParamBufferSize;
ResultPointer = NULL;
ResultLength = ArgumentLength;

```

RoutineIndex参数是一个索引，它与回到3环的落脚点有关，在当前代码里它是一个宏，点进去查看就会发现实际上是一个数字，还有其他许多别的索引值。

```

#define USER32_CALLBACK_WINDOWPROC (0)
#define USER32_CALLBACK_SENDASYNCPROC (1)
#define USER32_CALLBACK_LOADSYSTEMMENUTEMPLATE (2)
#define USER32_CALLBACK_LOADDEFAULTCURSORS (3)
#define USER32_CALLBACK_HOOKPROC (4)
#define USER32_CALLBACK_EVENTPROC (5)
#define USER32_CALLBACK_LOADMENU (6)
#define USER32_CALLBACK_CLIENTTHREADSTARTUP (7)
#define USER32_CALLBACK_CLIENTLOADLIBRARY (8)
#define USER32_CALLBACK_GETCHARSETINFO (9)
#define USER32_CALLBACK_MAXIMUM (9)

```

既然有索引，也就表示肯定有张表，可以通过索引在表中找到回到3环的落脚点。**这张表就叫做回调函数表**，其包含多个回调函数，根据不同的索引值RoutineIndex，KeUserModeCallback就可以调用表内不同的回调函数。

这些回调函数均由USER32.dll提供，回调函数表我们可以这样去寻找：FS:[0]->TEB->TEB.PEB(0x30偏移位)->PEB.KernelCallbackTable(0x2C偏移位)，其实也就是PEB结构体中的成员KernelCallbackTable。

如下图所示随便找个程序放大OD里都可以找到回调函数表，回调函数实现的功能我们也可以推测出个大概，从Argument中取窗口过程函数地址，再将其他几个参数作为传参调用窗口过程函数：

地址	数值	注释
7FFDF000	0012FFE0	(指向 SEH 链指针)
7FFDF004	00130000	(线程堆栈顶部)
7FFDF008	0012D000	(线程堆栈底部)
7FFDF00C	00000000	
7FFDF010	00001E00	
7FFDF014	00000000	
7FFDF018	7FFDF000	
7FFDF01C	00000000	
7FFDF020	00000D60	
7FFDF024	0000035C	(线程 ID)
7FFDF028	00000000	
7FFDF02C	00000000	(指向线程局部存储指针)
7FFDF030	7FFD6000	
7FFDF034	00000006	(上个错误 = ERROR_INVALID_HANDLE)

C	0	ES	0023	32位	0(FFFFFFFF)
P	1	CS	001B	32位	0(FFFFFFFF)
A	0	SS	0023	32位	0(FFFFFFFF)
Z	1	DS	0023	32位	0(FFFFFFFF)
S	0	FS	0020	32位	7FFDF000(FFF)
T	0	GS	0000	NULL	
D	0				

地址	数值	注释
77D12970	77D3F534	user32.77D3F534
77D12974	77D583AC	user32.77D583AC
77D12978	77D1B390	user32.77D1B390
77D1297C	77D1E613	user32.77D1E613
77D12980	77D58365	user32.77D58365
77D12984	77D58566	user32.77D58566
77D12988	77D30A9A	user32.77D30A9A
77D1298C	77D5883B	user32.77D5883B
77D12990	77D3F6E2	user32.77D3F6E2
77D12994	77D58708	user32.77D58708
77D12998	77D1E7E2	user32.77D1E7E2
77D1299C	77D58746	user32.77D58746
77D129A0	77D4FC0C	user32.77D4FC0C
77D129A4	77D58784	user32.77D58784
77D129A8	77D58784	user32.77D58784
77D129AC	77D585A6	user32.77D585A6
77D129B0	77D4F37C	user32.77D4F37C
77D129B4	77D1D2BC	user32.77D1D2BC
77D129B8	77D21A95	user32.77D21A95
77D129BC	77D1D1B3	user32.77D1D1B3
77D129C0	77D3DC44	user32.77D3DC44
77D129C4	77D1D65C	user32.77D1D65C

地址	数值	注释
7FFD6000	00000000	
7FFD6004	FFFFFFFF	
7FFD6008	00400000	LoadDll.00400000
7FFD600C	00251E90	UNICODE "("
7FFD6010	00020000	
7FFD6014	00000000	
7FFD6018	00150000	
7FFD601C	7C99E4C0	ntdll.7C99E4C0
7FFD6020	7C921005	ntdll.RtlEnterCriticalSection
7FFD6024	7C9210ED	ntdll.RtlLeaveCriticalSection
7FFD6028	00000001	
7FFD602C	77D12970	user32.77D12970