

# 1 软件调试

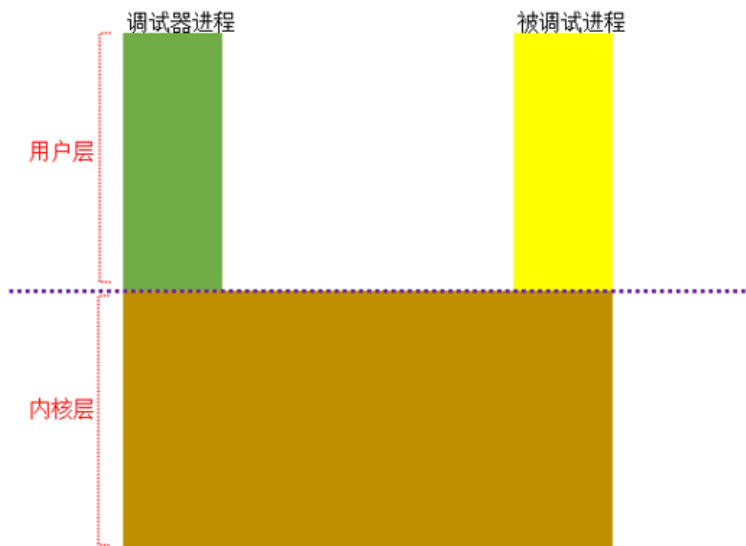
软件调试实际上涉及的内容并不多。如果你想开发一个调试器，掌握十几个API的使用就足够了。然而，如果你希望在调试与反调试的对抗中保持主动地位，对细节的了解就变得尤为重要。

软件调试的学习主要涉及到的文件有kernel32.dll、ntdll.dll和ntoskrnl.exe。对于一些未公开的信息，我们可以通过ReactOS (<https://master.dl.sourceforge.net/project/reactos/ReactOS/0.3.15/ReactOS-0.3.15-REL-src.zip?viasf=1>) 来帮助我们分析代码。

## 1.1 调试对象

### 1.1.1 调试器与被调试程序

调试器与被调试程序在用户层都是独立的进程，我们要想将两者之间相互关联，就需要通过一个媒介，即进程空间的高2G，也就是我们常说的内核层，每个进程的高2G都是共享的，因此调试器可以通过它来与被调试器进行通信。



通过调试器去调试一个程序有两个办法，第一种方法是通过**CreateProcess**函数打开一个未运行的程序，第二种方法是附加进程的方式，通过**DebugActiveProcess**函数附加一个正在运行的进程。

这两种方法其实在建立进程间联系的方式是一样的，唯一不同的是CreateProcess函数多了一个步骤，即创建进程，因此我们只需要来分析**DebugActiveProcess**函数就能知道建立联系的方法。

### 1.1.2 DebugActiveProcess执行流程

#### 关联调试对象与调试器

我们通过IDA来分析DebugActiveProcess函数的执行流程，首先，我们进入kernel32.dll中的DebugActiveProcess函数，在函数的最前面，我们可以注意到一个值得关注的函数，名为**DbgUiConnectToDbg**。

当我们进入这个函数后，我们发现它调用了另一个同名函数，位于ntdll.dll模块中。

```

; BOOL __stdcall DebugActiveProcess(DWORD dwProcessId)
public _DebugActiveProcess@4
_DebugActiveProcess@4 proc near          ; DATA XREF: .text:off_7C8026541o

    dwProcessId= dword ptr 8

    mov     edi, edi
    push    ebp
    mov     ebp, esp
    call    _DbgUiConnectToDbg@0          ; DbgUiConnectToDbg()

; NTSTATUS __stdcall DbgUiConnectToDbg()
_DbgUiConnectToDbg@0 proc near
    jmp     ds:__imp__DbgUiConnectToDbg@0

```

Address	Ordinal	Name	Library
7C801118		DbgUiConnectToDbg	ntdll

我们继续跟进就会发现ntdll.dll模块中的DbgUiConnectToDbg函数，我们可以看见它先获取FS:[0x18]，也就是\_TEB.NtTib.Self，其实就是\_TEB的地址本身，然后通过它来获取FS:[0xF24]，即\_TEB.DbgSSReserved[1]（该成员是一个指针数组，可以存储2个，它是一个保留成员，专门给调试器使用，调试器可以使用该字段来存储自己），用它作为参数给到了\_ZwCreateDebugObject函数，最终该函数执行的返回结果就是EAX，也就表示将返回信息存储到了\_TEB.DbgSSReserved[1]中。

```

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 18h
xor     ecx, ecx
mov     eax, large fs:18h
cmp     [eax+0F24h], ecx
jnz     short loc_7C9706D5

```

```

mov     [ebp+var_18], 18h
mov     [ebp+var_14], ecx
mov     [ebp+var_C], ecx
mov     [ebp+var_10], ecx
mov     [ebp+var_8], ecx
mov     [ebp+var_4], ecx
mov     eax, large fs:18h

```

```

push    1
lea     ecx, [ebp+var_18]
push    ecx
push    1F000Fh
add     eax, 0F24h
push    eax

```

```

call    _ZwCreateDebugObject@16

```

```

mov     ecx, eax

```

```

kd> dt _TEB
nt!_TEB
+0x000 NtTib           : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void

kd> dt _NT_TIB
nt!_NT_TIB
+0x000 ExceptionList   : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase       : Ptr32 Void
+0x008 StackLimit      : Ptr32 Void
+0x00c SubSystemTib    : Ptr32 Void
+0x010 FiberData       : Ptr32 Void
+0x010 Version         : Uint4B
+0x014 ArbitraryUserPointer : Ptr32 Void
+0x018 Self            : Ptr32 NT_TIB

```

```

kd> dt _TEB
nt!_TEB
+0x000 NtTib           : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
...
+0xf1c ReservedForNtRpc : Ptr32 Void
+0xf20 DbgSSReserved   : [2] Ptr32 Void
+0xf28 HardErrorsAreDisabled : Uint4B

```

```

; ZwCreateDebugObject(x,x,x,x)

```

\_ZwCreateDebugObject函数其实什么也没干，就是通过系统调用号进入0环执行对应的0环函数来做具体的事情，根据函数的命名，我们可以推断，这个函数的作用是用于创建调试对象\_DEBUG\_OBJECT（以下简称为调试对象）。调试对象充当了我们之前提到的媒介，目前调试器与调试对象已经关联起来了。

因此，这意味着该函数的返回结果即为调试对象，它不会直接返回调试对象在0环的地址。**所以在3环**  
**\_TEB.DebugSSReserved[1]中存储了该调试对象的句柄。**这个句柄用于在3环中引用和操作调试对象。

```

; __stdcall ZwCreateDebugObject(x, x, x, x)
public _ZwCreateDebugObject@16
_ZwCreateDebugObject@16 proc near          ; CODE XREF: DbgUiConnectToDbg()+45↓p
                                           ; DATA XREF: .text:off_7C92395C↑o
                                           ; NtCreateDebugObject
mov     eax, 21h ; '!'
mov     edx, 7FFE0300h
call    dword ptr [edx]

retn    10h

```

### 关联调试对象与被调试进程

在以上流程都结束之后，再回到kernel32.dll，如下图所示，会比较返回结果，如果调试对象创建成功就进行跳转。

```

mov     edi, edi
push    ebp
mov     ebp, esp
call    _DbgUiConnectToDbg@0              ; DbgUiConnectToDbg()

test     eax, eax
jge     short loc_7C859F23                ; 比较返回结果，如果调试对象创建成功就跳转

push     eax                             ; Status
call     _BaseSetLastNTErr@4              ; BaseSetLastNTErr(x)

```

继续跟进代码，如下图所示我们可以看见首先通过\_ProcessIdToHandle函数，根据进程ID获取被调试进程的句柄（返回结果EAX给到ESI），其次将被调试进程句柄作为参数带入到\_DbgUiDebugActiveProcess函数。

```

loc_7C859F23:                             ; CODE XREF: DebugActiveProcess(x)+C↑j
push     esi
push     [ebp+dwProcessId]                ; ProcessHandle
call     _ProcessIdToHandle@4              ; 根据进程ID获取进程句柄（被调试进程）

mov     esi, eax
test     esi, esi
jz       short loc_7C859F54

push     edi
push     esi                             ; 将被调试句柄作为参数传入
call     _DbgUiDebugActiveProcess@4        ; DbgUiDebugActiveProcess(x)

```

\_DbgUiDebugActiveProcess函数位于ntdll.dll模块中，所以需要切过去进行查看。在该函数内先获取当前线程的\_TEB，然后获取\_TEB的0xF24偏移位成员，即调试对象，将其与被调试进程句柄一并作为参数带入到\_NtDebugActiveProcess函数。而这个函数仍然是一个系统调用，最终进入0环。

```

; int __stdcall DbgUiDebugActiveProcess(HANDLE Handle)
public _DbgUiDebugActiveProcess@4
_DbgUiDebugActiveProcess@4 proc near    ; DATA XREF: .text:off_7C92395C↑o

Handle= dword ptr 8

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     eax, large fs:18h                ; 获取当前线程的TEB
push    dword ptr [eax+0F24h]            ; 压入TEB+0xF24, 即调试对象
push    [ebp+Handle]                    ; 压入被调试进程句柄
call    _NtDebugActiveProcess@8         ; NtDebugActiveProcess(x,x)

; __stdcall NtDebugActiveProcess(x, x)
public _NtDebugActiveProcess@8
_NtDebugActiveProcess@8 proc near        ; CODE XREF: DbgUiDebugActiveProcess(x)+15↓p
                                          ; DATA XREF: .text:off_7C92395C↑o
                                          ; NtDebugActiveProcess
mov     eax, 39h ; '9'
mov     edx, 7FFE0300h
call    dword ptr [edx]

retn    8

```

我们要继续跟进0环，也就是ntoskrnl.exe模块，找同名函数\_NtDebugActiveProcess即可。如下图所示，我们可以看见最开始部分就是调用了\_ObReferenceObjectByHandle函数，**该函数的作用就是通过句柄来获取对象**，它有6个参数，第5个参数就是用于存储函数获取的进程对象地址，在这里是通过被调试进程的句柄（参数1）获取\_PsProcessType（参数3）类型的对象，也就是\_EPROCESS。

```

; NTSTATUS __stdcall NtDebugActiveProcess(HANDLE Process, HANDLE DebugObject)
_NtDebugActiveProcess@8 proc near        ; DATA XREF: .text:0040E024↑o

AccessMode= byte ptr -4
Process= dword ptr 8
DebugObject= dword ptr 0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
mov     eax, large fs:124h
mov     al, [eax+140h]
push    0                                ; 参数6
mov     [ebp+AccessMode], al
lea     eax, [ebp+Process]
push    eax                              ; 参数5: OUT类型, 用于存储函数获取的_EPROCESS
push    dword ptr [ebp+AccessMode]       ; 参数4
push    _PsProcessType                  ; 参数3: ObjectType
push    800h                            ; 参数2
push    [ebp+Process]                   ; 参数1: 被调试进程句柄
call    _ObReferenceObjectByHandle@24    ; 通过句柄获取对象, 此处用于获得被调试进程的_EPROCESS

```

往下继续看，这里就是获取当前进程结构体地址与被调试进程结构体地址进行对比，即判断是否存在自己调试自己的情况，如果存在则跳转走。以及判断了被调试进程是否是初始化系统进程，如果是的话也跳转走。这里其实也就说明了两种无法继续调试的条件。

```

push    ebx
push    esi
mov     eax, large fs:124h          ; 获取当前线程结构体_KTHREAD
                                           ; FS:[0] -> _KPCR
                                           ; FS:[0x124] -> _KPCR._KPRCB.CurrentThread
mov     esi, [ebp+Process]         ; ebp+Process已不再是被调试进程句柄了
                                           ; 在 _ObReferenceObjectByHandle函数执行之后
                                           ; 这里已经变成了被调试进程的 _EPROCESS结构体
cmp     esi, [eax+44h]             ; 取当前进程(_KTHREAD.ApcState.Process)与被调试进程进行对比
                                           ; 也就是判断当前是否属于自己调试自己
jz      short loc_58B227           ; 如果是自己调试自己就跳转

cmp     esi, _PsInitialSystemProcess ; 接着再判断被调试进程是否是初始化系统进程
jz      short loc_58B227           ; 如果是则跳转

```

然后又来到了通过句柄获取对象的环节，这次是通过调试对象的句柄来获取调试对象的地址，并且将调试对象的地址和被调试进程的\_EPROCESS结构体地址作为参数带入\_DbgkpSetProcessDebugObject函数执行。

```

push    0                          ; HandleInformation
lea     eax, [ebp+Process]          ; 存放的位置
push    eax                        ; Object
push    dword ptr [ebp+AccessMode]  ; AccessMode
push    _DbgkDebugObjectType       ; ObjectType
push    2                          ; DesiredAccess
push    [ebp+DebugObject]           ; 调试对象的句柄
call    _ObReferenceObjectByHandle@24 ; 通过调试对象的句柄获取调试对象结构体的地址
...
push    [ebp+DebugObject]           ; 调试对象句柄
push    eax                        ; int
push    [ebp+Process]              ; 调试对象的地址
push    esi                        ; 被调试进程的 _EPROCESS结构体
call    _DbgkpSetProcessDebugObject@16 ; DbgkpSetProcessDebugObject(x,x,x,x)

```

\_DbgkpSetProcessDebugObject函数的作用就是将调试对象与被调试进程关联，如下图所示，我们可以看见在获取了被调试进程结构体地址之后判断被调试进程是否已经属于被调试状态，即\_EPROCESS的0xBC偏移位成员DebugPort是否非0，如果为0则表示没有被调试继续向下执行。接着就是最关键的，将调试对象放进被调试进程结构体的DebugPort成员当中，至此就将两者成功关联起来。

```

; int __stdcall DbgkpsSetProcessDebugObject(PVOID, PVOID, int, PVOID Object)
_DbgkpsSetProcessDebugObject@16 proc near
    ; CODE XREF: NtDebugActiveProcess(x,x)+97↓p

P= dword ptr -18h
var_14= dword ptr -14h
FastMutex= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_2= byte ptr -2
var_1= byte ptr -1
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h
Object= dword ptr 14h
...
mov     eax, [ebp+arg_8]
xor     ebx, ebx                ; ebx为0
cmp     eax, ebx
...
loc_58AF74:                    ; CODE XREF: DbgkpsSetProcessDebugObject(x,x,x,x)+34↑j
cmp     [ebp+arg_8], ebx
mov     edi, [ebp+arg_0]        ; 获取_EPROCESS
mov     esi, ds:__imp__ExAcquireFastMutex@4 ; ExAcquireFastMutex(x)
jnl     loc_58B019

mov     ecx, offset _DbgkpProcessDebugPortMutex ; FastMutex
mov     [ebp+var_1], 1
call    esi ; ExAcquireFastMutex(x) ; ExAcquireFastMutex(x)

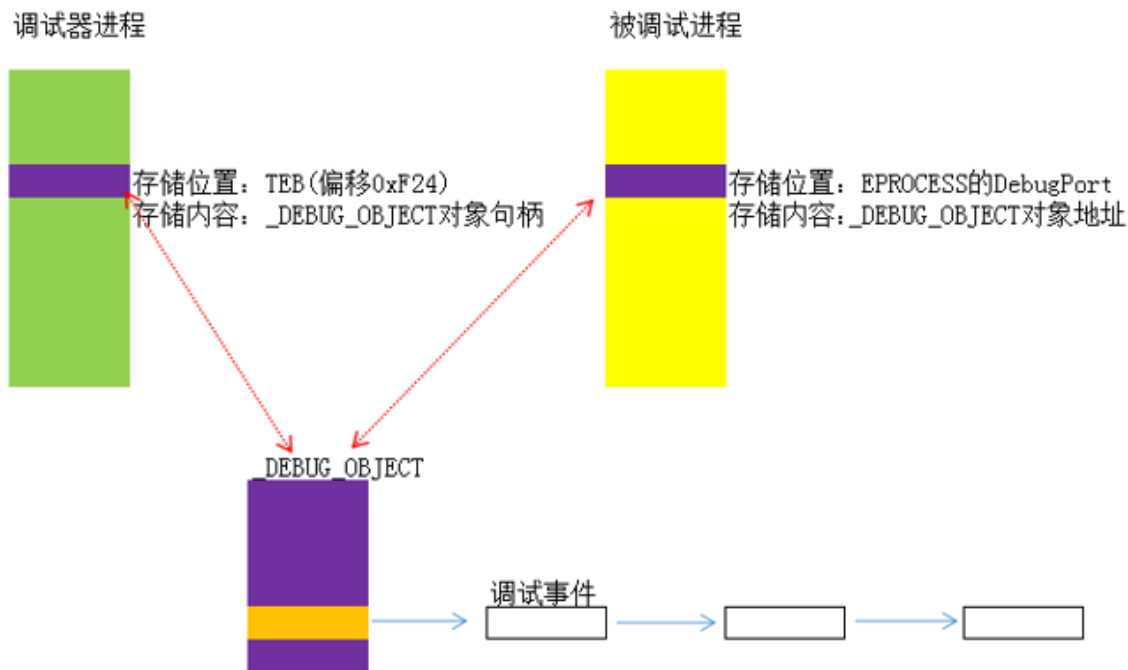
cmp     [edi+0BCh], ebx        ; 判断DebugPort ( _EPROCESS+0xBC) 是否为0
jnz     short loc_58B00C        ; 如果不是0, 则表示已经被调试了, 则跳转

loc_58AF99:                    ; CODE XREF: DbgkpsSetProcessDebugObject(x,x,x,x)+CF↑j
mov     eax, [ebp+arg_4]        ; 获取调试对象
mov     ecx, [ebp+Object]       ; Object
mov     [edi+0BCh], eax        ; 将调试对象存储到被调试进程的DebugPort
call    @ObfReferenceObject@4   ; ObfReferenceObject(x)

```

### 1.1.3 \_DEBUG\_OBJECT结构体

通过上述的学习，我们了解到了调试器与被调试进程，通过调试对象结构体\_DEBUG\_OBJECT成功的关联起来：



以下为\_DEBUG\_OBJECT结构体的组成：

```
1 typedef struct _DEBUG_OBJECT {
2     KEVENT EventsPresent;
3     FAST_MUTEX Mutex;
4     LIST_ENTRY EventList;
5     ULONG Flags;
6 } DEBUG_OBJECT, *PDEBUG_OBJECT;
```

1.1.4 调试对抗

在了解了调试原理之后，我们可以总结出一些反调试的方法：

- 1. **DebugPort检查与置0**：创建一个线程，不断检查当前进程的DebugPort值，一旦有值就退出程序或将其置0，中断调试对象与被调试进程之间的联系。
- 2. **遍历所有进程的TEB+0xF24处**：检查该位置是否有值，如果有值，说明存在调试器，可以选择退出程序。
- 3. **Hook NtCreateDebugObject**：阻止NtCreateDebugObject函数创建调试对象，从而防止调试器的附加。

反反调试的方法：

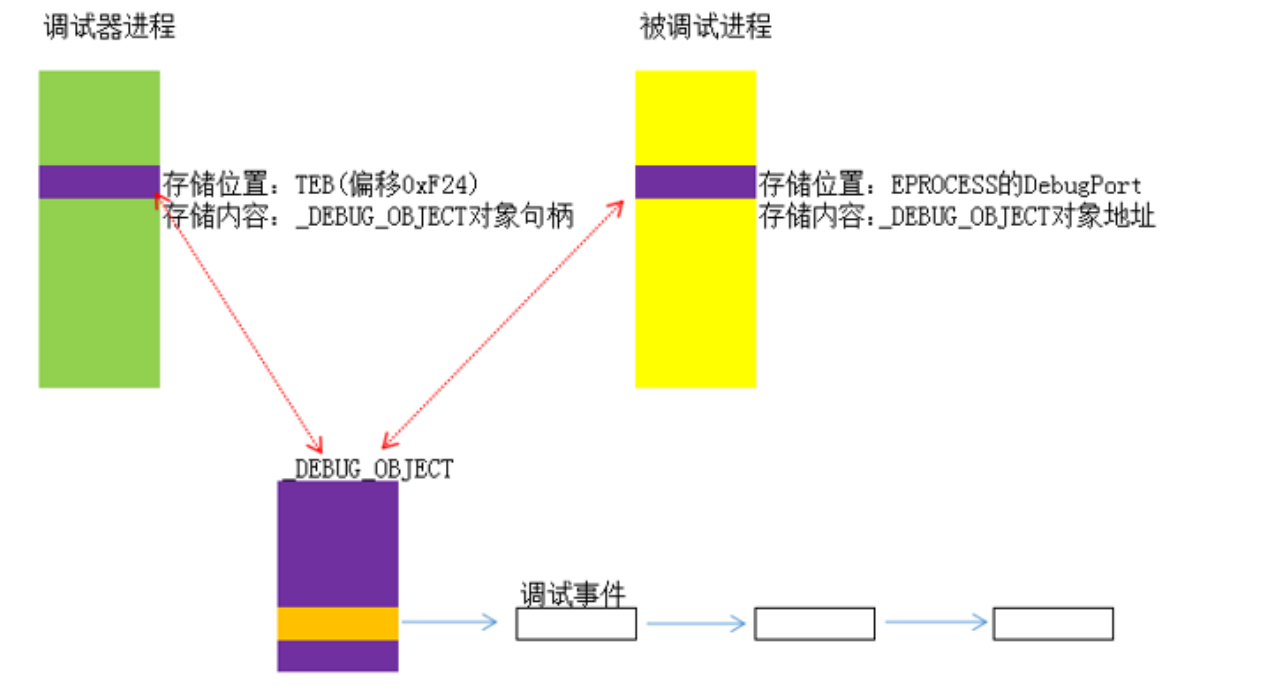
- 1. **针对DebugPort检查与置0**：不使用DebugPort，而是在进程结构体中另外选择一个空闲成员来存放\_DEBUG\_OBJECT的地址。
- 2. **针对Hook NtCreateDebugObject**：自行分配内存给\_DEBUG\_OBJECT结构体，并为其成员赋值，绕过原有的Hook。
- 3. **重写DebugActiveProcess函数**：重新实现DebugActiveProcess函数，以增加对反调试技术绕过能力。

## 1.2 调试事件的采集

### 1.2.1 调试事件

调试器与被调试进程之间通过\_DEBUG\_OBJECT（调试对象）来建立起联系，调试器通过调试对象的EventList成员来指导被调试进程到底发生了什么事情，该成员是一个链表，其中存储的是从被调试进程发送过来的各种类型的调试事件。

```
1 typedef struct _DEBUG_OBJECT {
2     KEVENT EventsPresent; // 用于指示有调试事件发生的事件对象
3     FAST_MUTEX Mutex; // 用于同步的互斥对象
4     LIST_ENTRY EventList; // 保存调试事件的链表
5     ULONG Flags; // 标志位，调试消息是否已读取
6 } DEBUG_OBJECT, *PDEBUG_OBJECT;
```



### 种类

不是所有的进程行为都会产生调试事件的，调试事件也分种类，如下所示一共有8种，但有一个已经废弃，因此实际上只有7种调试事件：

```
1 typedef enum _DBGKM_APINUMBER
2 {
3     DbgKmExceptionApi = 0, // 异常
4     DbgKmCreateThreadApi = 1, // 创建线程
```



```

5      DbgKmCreateProcessApi = 2, // 创建进程
6      DbgKmExitThreadApi = 3, // 线程退出
7      DbgKmExitProcessApi = 4, // 进程退出
8      DbgKmLoadDllApi = 5, // 加载DLL
9      DbgKmUnloadDllApi = 6, // 卸载DLL
10     DbgKmErrorReportApi = 7, // 已废弃
11     DbgKmMaxApiNumber = 8, // 最大值
12 } DBGKM_APINUMBER;

```

## 采集函数

在Windows系统中提供了一些以Dbgk开头的函数，可用于采集调试事件，并生成相应事件的结构体，我们称之为**调试事件采集函数**。如下图所示，我们会发现**这些事件采集函数都是在API调用的必经之路上加入的**。图中黑色字体的函数就是每个事件API最终都会经过的地方，紫色字体的函数用于采集调试事件，红色字体的函数用于发送（写入）调试事件。

<1> 创建进程、线程必经之路：

```

PspUserThreadStartup
    DbgkCreateThread
        DbgkpSendApiMessage(x, x)

```

<2> 退出线程、进程必经之路：

```

PspExitThread
    DbgkExitThread/DbgkExitProcess
        DbgkpSendApiMessage(x, x)

```

<3> 加载模块的必经之路：

```

NtMapViewOfSection
    DbgkMapViewOfSection
        DbgkpSendApiMessage(x, x)

```

<4> 卸载模块的必经之路：

```

NtUnMapViewOfSection
    DbgkUnMapViewOfSection
        DbgkpSendApiMessage(x, x)

```

<5> 异常的必经之路：

```

KiDispatchException
    DbgkForwardException
        DbgkpSendApiMessage(x, x)

```

## 1.2.2 采集流程

我们要想知道采集函数做了什么就需要跟进流程，来看它的代码逻辑。这里我们以创建进程、线程，和退出线程、进程为例来跟进分析一下。

### 创建进程、线程事件采集

创建进程的本质就是创建线程，其中第一次创建线程时即为创建进程。因此，无论是创建进程还是线程，底层调用的函数是相同的，都是PspUserThreadStartup函数。

在PspUserThreadStartup函数内，我们可以很快找到对应的调试事件采集函数\_DbgkCreateThread。

```

004AFED8      loc_4AFED8:                ; CODE XREF: PspUserThreadStartup(x,x)+44↑j
004AFED8 32 C9      xor     cl, cl                ; NewIrql
004AFEDA FF 15 04 10 40 00    call    ds:__imp_@KfLowerIrql@4 ; KfLowerIrql(x)
004AFEDA
004AFEE0 F6 86 48 02 00 00 06    test    byte ptr [esi+248h], 6
004AFEE7 75 08      jnz     short loc_4AFEF1
004AFEE7
004AFEE9 FF 75 0C      push    [ebp+arg_4]
004AFEEC E8 E0 FE FF FF    call    DbgkCreateThread@4 ; DbgkCreateThread(x)
004AFEEC
004AFEF1      loc_4AFEF1:                ; CODE XREF: PspUserThreadStartup(x,x)+80↑j
004AFEF1 80 7D E7 00      cmp     [ebp+var_19], 0
004AFEF5 0F 85 83 72 00 00    jnz     loc_48717E
004AFEF5
004AFEFB 83 3D E8 92 48 00 00    cmp     _CcPfEnablePrefetcher, 0
004AFF02 74 13      jz      short loc_4AFF17
004AFF02
004AFF04 8D 97 48 02 00 00    lea     edx, [edi+248h]
0009B4EC 004AFEEC: PspUserThreadStartup(x,x)+85 (Synchronized with Hex View-1)

```

继续跟进\_DbgkCreateThread函数，我们会发现它判断了当前进程的DebugPort的值是否为空，这个判断是每个调试事件采集函数都会走的逻辑，如果判断值不为空则表示当前进程正在被调试就跳转进去。

```

; __unwind { // __SEH_prolog
push    0D4h
push    offset stru_4248A0
call    __SEH_prolog

mov     eax, large fs:124h
mov     [ebp+var_3C], eax
mov     esi, [eax+44h]
mov     [ebp+var_30], esi
xor     ebx, ebx
cmp     ds:_PsImageNotifyEnabled, bl
jnz     loc_4EDAB4

```

```

loc_4AFDFD:                ; CODE XREF: DbgkCreateThread(x)+3DCE6↓j
                        ; DbgkCreateThread(x)+3DDF0↓j
cmp     [esi+0BCh], ebx
jnz     loc_4EDBC6

```

```

loc_4AFE09:                ; CODE XREF: DbgkCreateThread(x)+3E01B↓j
                        ; DbgkCreateThread(x)+3E02F↓j
                        ; DbgkCreateThread(x)+3E07A↓j
call    __SEH_epilog

retn    4
; } // starts at 4AFDD1

```

跟进跳转的片段代码，我们可以看见它一开始就做了一件事情，即判断当前线程是否为第一个线程，以此来判断生成的调试事件是创建进程还是创建线程。

```

loc_4EDBC6:                                ; CODE XREF: DbgkCreateThread(x)+32↑j
cmp     [esi+3Ch], ebx
jz      short loc_4EDBD7

xor     eax, eax
inc     eax
lea     ecx, [esi+248h]
lock or [ecx], eax

loc_4EDBD7:                                ; CODE XREF: DbgkCreateThread(x)+3DDF8↑j
xor     ecx, ecx
inc     ecx
lea     edx, [esi+248h]
mov     eax, [edx]

loc_4EDBE2:                                ; CODE XREF: DbgkCreateThread(x)+3DE19↓j
mov     edi, eax
or      edi, ecx
lock cmpxchg [edx], edi
jnz     short loc_4EDBE2

mov     [ebp+FileHandle], ebx
test    cl, al
jnz     loc_4EDE18
mov     [ebp+var_A8], ebx
push    dword ptr [esi+138h]                ; P
call    _DbgkpSectionToFileHandle@4        ; DbgkpSectionToFileHandle(x)

mov     [ebp+Handle], eax
mov     eax, [esi+13Ch]
mov     [ebp+var_B4], eax
mov     [ebp+var_A4], ebx
mov     [ebp+var_B0], ebx
mov     [ebp+var_AC], ebx

; __try { // __except at loc_4EDC76
mov     [ebp+ms_exc.registration.TryLevel], 2
push    dword ptr [esi+13Ch]                ; BaseAddress
call    _RtlImageNtHeader@4                ; RtlImageNtHeader(x)

mov     [ebp+var_44], eax
cmp     eax, ebx
jz      short loc_4EDC8D

mov     ecx, [eax+34h]
add     ecx, [eax+28h]
mov     [ebp+var_A4], ecx
mov     ecx, [eax+0Ch]
mov     [ebp+var_B0], ecx
mov     eax, [eax+10h]
mov     [ebp+var_AC], eax
jmp     short loc_4EDC8D

```

接着它做了第二件事情，将对应的调试事件打包成一个结构体，跳转到如下片段代码，最终会调用 `_DbgkpSendApiMessage` 函数。这个函数的第一个参数就是之前打包好的调试事件的结构体。

```

loc_4EDC8D:                                ; CODE XREF: DbgkCreateThread(x)+3DE75↑j
                                           ; DbgkCreateThread(x)+3DE95↑j
or      [ebp+ms_exc.registration.TryLevel], 0FFFFFFFh
mov     [ebp+var_DC], 780024h
mov     [ebp+var_D8], 8
mov     [ebp+var_C4], 2
push    ebx
lea     eax, [ebp+var_DC]
push    eax
call    _DbgkSendApiMessage@8              ; DbgkSendApiMessage(x,x)

```

## 退出线程、进程事件采集

退出进程的本质也是退出线程，其中退出线程为最后一个时即为退出进程。因此，无论是退出进程还是线程，底层调用的函数是相同的，都是PspExitThread函数。

在PspExitThread函数内，我们可以看见它判断了DebugPort是否为0，如果不为0表示当前正在调试，就进行跳转。

```

loc_49F1A6:                                ; CODE XREF: PspExitThread(x)+D1↑j
                                           ; PspExitThread(x)+9023A↓j
cmp     [ebp+Object], 0
jnz     loc_4B723F

```

```

loc_49F1B0:                                ; CODE XREF: PspExitThread(x)+1817F↓j
cmp     dword ptr [edi+0BCh], 0
jnz     loc_52F307

```

```

loc_49F1BD:                                ; CODE XREF: PspExitThread(x)+90246↓j
                                           ; PspExitThread(x)+9025D↓j
                                           ; PspExitThread(x)+9026A↓j
cmp     _KdDebuggerEnabled, 0
jz      short loc_49F1D3

```

跟进跳转的片段代码，这里判断了当前退出的线程是不是最后一个，也就表示判断是否是退出的进程，如果是则调用函数\_DbgkExitProcess，反之如果不是则表示当前退出的是线程，则调用函数\_DbgkExitThread。**这里是根据退出事件选择对应的采集函数。**

```

loc_52F307:                                ; CODE XREF: PspExitThread(x)+EF↑j
test     byte ptr [esi+248h], 10h
jnz      loc_49F1BD

cmp      [ebp+var_19], 0
jz       short loc_52F32A

push     dword ptr [edi+24Ch]
call     _DbgkExitProcess@4                ; DbgkExitProcess(x)

jmp      loc_49F1BD

; -----

loc_52F32A:                                ; CODE XREF: PspExitThread(x)+90250↑j
push     [ebp+arg_0]
call     _DbgkExitThread@4                ; DbgkExitThread(x)

jmp      loc_49F1BD

```

而我们每个都跟进去看，会发现**最开始都在填充对应的调试事件结构体**，最终调用\_DbgkpSendApiMessage函数。

<pre> _DbgkExitProcess mov     eax, [ebp+arg_0] mov     [ebp+var_58], eax push    0 lea     eax, [ebp+var_78] push    eax mov     [ebp+var_78], 78000Ch mov     [ebp+var_74], 8 mov     [ebp+var_60], 4 call    _DbgkpSendApiMessage@8      ; DbgkpSendApiMessage(x,x) </pre>	<pre> _DbgkExitThread mov     eax, [ebp+arg_0] push    ebx mov     [ebp+var_58], eax mov     [ebp+var_78], 78000Ch mov     [ebp+var_74], 8 mov     [ebp+var_60], 3 call    _DbgkpSuspendProcess@0      ; DbgkpSuspendProcess()  mov     bl, al push    0 lea     eax, [ebp+var_78] push    eax call    _DbgkpSendApiMessage@8      ; DbgkpSendApiMessage(x,x) </pre>
---	--

### 1.2.3 DbgkpSendApiMessage函数

#### 执行流程

通过上面几个案例的分析，我们知道DbgkpSendApiMessage函数的作用是将已创建的调试事件发送到调试对象的事件链表中。

进入函数内，发现它调用了\_DbgkpQueueMessage函数，该函数有两个参数，一个是Event，它是由调试事件采集函数创建的结构体，另一个是FastMutex，这是一个互斥体参数，与调试对象的第一个成员相同。

```

loc_58B438:                                ; CODE XREF: DbgkpSendApiMessage(x,x)+C↑j
mov     edx, [ebp+arg_0]
mov     dword ptr [edx+1Ch], 103h
mov     eax, large fs:124h
mov     ecx, [eax+44h]
xor     eax, eax
inc     eax
lea     esi, [ecx+248h]
lock or [esi], eax
mov     eax, large fs:124h
push    ebx                                ; FastMutex
push    ebx                                ; Event
push    edx                                ; int
push    eax                                ; PVOID
push    ecx                                ; Object
call    _DbgkpQueueMessage@20              ; DbgkpQueueMessage(x,x,x,x,x)

```

我们跟进\_DbgkpQueueMessage函数，在该函数内部发现代码逻辑执行到一半时，它先从自身进程的\_EPROCESS结构体中获取调试对象（调试对象不为空）。然后从调试对象中取出EventList成员的首地址（0x30偏移位），并将保存在EBX中的节点插入到EventList的第一个位置，也就表示这里的EBX为处理好的调试事件。至此，我们的调试事件就挂入了链表中，可以使得调试器进行处理了。

```

loc_58A1E4:                                ; CODE XREF: DbgkpQueueMessage(x,x,x,x,x)+17↑j
mov     ecx, offset _DbgkpProcessDebugPortMutex ; FastMutex
lea     ebx, [ebp+var_B8]
mov     [ebp+var_8C], esi
call    ds:imp_@ExAcquireFastMutex@4 ; ExAcquireFastMutex(x)

mov     eax, [ebp+Object]
mov     eax, [eax+08Ch] ; DebugPort -> 调试对象地址
mov     [ebp+Event], eax
mov     eax, [ebp+arg_8]
mov     eax, [eax+18h]
cmp     eax, 1
jz      short loc_58A217

loc_58A296:                                ; CODE XREF: DbgkpQueueMessage(x,x,x,x,x)+108↑j
add     ecx, 10h                                ; FastMutex
mov     [ebp+FastMutex], ecx
call    ds:imp_@ExAcquireFastMutex@4 ; ExAcquireFastMutex(x)

mov     edx, [ebp+Event]
test    byte ptr [edx+38h], 1
jnz     short loc_58A2CD

cmp     [ebp+var_4], edi
lea     eax, [edx+30h]
mov     ecx, [eax+4]
mov     [ebx], eax
mov     [ebx+4], ecx
mov     [ecx], ebx
mov     [eax+4], ebx
jnz     short loc_58A2C8

```

## 函数参数

该函数有两个参数，这里简要说明一下：

**参数1：**消息结构。总共有7种类型的消息结构，每种消息都有自己的消息结构，这些结构是由不同的调试事件采集函数创建的。

**参数2：**挂起线程标志。这个参数用于指示是否需要挂起除了当前进程之外的其他线程。有些调试事件需要挂起其他线程，比如针对int3断点的事件；而有些调试事件则不需要挂起线程，比如模块加载事件。

DbgkpSendApiMessage函数是整个调试事件采集的主要入口。如果在这个函数中设置了钩子，调试器将无法正常进行调试操作。

### 1.2.4 再看\_DEBUG\_OBJECT

我们现在再看一下调试对象的每个成员，应该能更好理解一些，在调用DbgkpQueueMessage函数将调试事件添加到链表后，**\_DEBUG\_OBJECT.EventsPresent状态会被修改**，调试器会判断这个状态，当状态改变后从EventList链表中提取调试事件。

为了避免并发修改EventList链表，**需要使用一个Mutex进行互斥操作**。这样，当调试器从EventList链表中提取调试事件时，同时DbgkpSendApiMessage函数向EventList链表写入数据时也会被正确地互斥。另外，**Flags参数**用于标识该事件是否已被调试器读取。

```

1  typedef struct _DEBUG_OBJECT {
2      KEVENT EventsPresent; // 用于指示有调试事件发生的事件对象
3      FAST_MUTEX Mutex; // 用于同步的互斥对象
4      LIST_ENTRY EventList; // 保存调试事件的链表
5      ULONG Flags; // 标志位，调试消息是否已读取
6  } DEBUG_OBJECT, *PDEBUG_OBJECT;

```

## 1.3 调试事件的处理

调试事件采集之后就需要对不同的事件进行处理，接下来我们通程序模拟两种建立调试关系的方式（即创建进程、附加进程），分析调试事件的处理过程和不同调试事件的结构细节。

### 1.3.1 建立调试关系：创建进程

如下代码所示，我们创建了一个简易调试器，首先创建一个调试进程（CreateProcess函数第6个参数表示创建进程的标志，用于指定创建进程的行为和属性，这里的值为DEBUG\_PROCESS | DEBUG\_ONLY\_THIS\_PROCESS，表示创建一个调试进程，并限制只有当前调试器可以附加到该进程进行调试，确保调试过程的独占性和安全性），接着进入一个调试循环。

在循环中，等待调试事件的发生，如异常、线程创建、进程创建等，然后根据具体的调试事件类型进行相应的处理。处理完事件后，告诉被调试程序继续执行。

```

1  #include "windows.h"
2  #include "stdio.h"
3
4  int main()
5  {
6      // 定义调试进程路径
7      const char* dbgProcessName = "C:\\notepad.exe";
8
9      // 创建调试进程
10     STARTUPINFO startupInfo = {0};
11     PROCESS_INFORMATION processInfo = {0};
12     GetStartupInfo(&startupInfo);
13
14     BOOL createProcessSuccess = CreateProcess(
15         dbgProcessName, NULL, NULL, NULL, TRUE,
16         DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS,
17         NULL, NULL, &startupInfo, &processInfo
18     );
19
20     if (!createProcessSuccess)
21     {
22         printf("CreateProcess error: %d\n", GetLastError());
23         getchar();
24         return 0;
25     }
26
27     // 调试循环

```

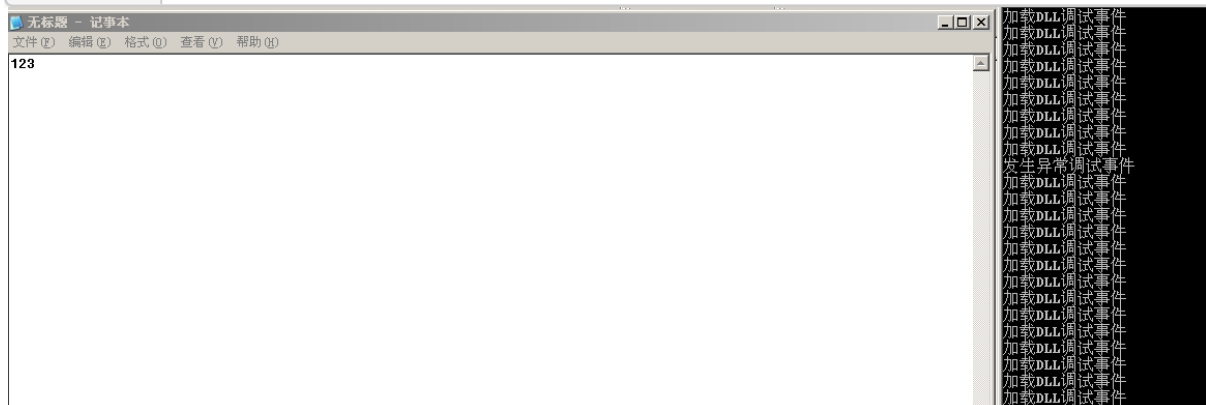
```
28     while (TRUE)
29     {
30         // 等待调试事件
31         DEBUG_EVENT debugEvent = {0};
32         BOOL waitForDebugEventSuccess = WaitForDebugEvent(&debugEvent,
INFINITE);
33         if (!waitForDebugEventSuccess)
34         {
35             printf("WaitForDebugEvent error: %d\n", GetLastError());
36             return 0;
37         }
38
39         // 处理调试事件
40         switch (debugEvent.dwDebugEventCode)
41         {
42             case EXCEPTION_DEBUG_EVENT:
43                 printf("发生异常调试事件\n");
44                 break;
45
46             case CREATE_THREAD_DEBUG_EVENT:
47                 printf("创建线程调试事件\n");
48                 break;
49
50             case CREATE_PROCESS_DEBUG_EVENT:
51                 printf("创建进程调试事件\n");
52                 break;
53
54             case EXIT_THREAD_DEBUG_EVENT:
55                 printf("退出线程调试事件\n");
56                 break;
57
58             case EXIT_PROCESS_DEBUG_EVENT:
59                 printf("退出进程调试事件\n");
60                 break;
61
62             case LOAD_DLL_DEBUG_EVENT:
63                 printf("加载DLL调试事件\n");
64                 break;
65
66             case UNLOAD_DLL_DEBUG_EVENT:
67                 printf("卸载DLL调试事件\n");
68                 break;
69
70             default:
71                 break;
72         }
73
74         // 告诉被调试程序让其继续执行
75         /*
76         第三个参数可以有两个值：
77         1. DBG_CONTINUE：表示调试器已处理该异常
78         2. DBG_EXCEPTION_NOT_HANDLED：表示调试器没有处理该异常，转回到用户态
           中执行，寻找可以处理该异常的异常处理器
```



```

79         */
80         ContinueDebugEvent(debugEvent.dwProcessId, debugEvent.dwThreadId,
DBG_CONTINUE);
81     }
82
83     return 0;
84 }

```



## 调试事件结构

在调试循环事件等待中，我们是通过结构体DEBUG\_EVENT在WaitForDebugEvent函数中获取调试事件的，该结构体的定义如下。

前三个成员很好理解，最后一个成员是一个联合体，里面包含了各种结构体，这是因为调试事件类型的多样性，所以对应的结构也不同，通过这样的方式就可以根据不同类型的调试事件存放对应的结构体信息（这也是为什么不同的调试事件有不同的调试采集函数）。

```

1  typedef struct _DEBUG_EVENT {
2      DWORD dwDebugEventCode; // 调试事件类型
3      DWORD dwProcessId; // 触发调试事件的进程ID
4      DWORD dwThreadId; // 触发调试事件的线程ID
5      union { // 联合体
6          EXCEPTION_DEBUG_INFO Exception;
7          CREATE_THREAD_DEBUG_INFO CreateThread;
8          CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
9          EXIT_THREAD_DEBUG_INFO ExitThread;
10         EXIT_PROCESS_DEBUG_INFO ExitProcess;
11         LOAD_DLL_DEBUG_INFO LoadDll;
12         UNLOAD_DLL_DEBUG_INFO UnloadDll;
13         OUTPUT_DEBUG_STRING_INFO DebugString;
14         RIP_INFO RipInfo;
15     } u;
16 } DEBUG_EVENT, *LPDEBUG_EVENT;

```

各类调试事件对应的结构体如下：

```

1  // 异常类型信息

```

```

2  typedef struct _EXCEPTION_DEBUG_INFO {
3      EXCEPTION_RECORD ExceptionRecord;
4      DWORD dwFirstChance;
5  } EXCEPTION_DEBUG_INFO, *LPEXCEPTION_DEBUG_INFO;
6
7  // 线程创建类型信息
8  typedef struct _CREATE_THREAD_DEBUG_INFO {
9      HANDLE hThread;
10     LPVOID lpThreadLocalBase;
11     LPTHREAD_START_ROUTINE lpStartAddress;
12 } CREATE_THREAD_DEBUG_INFO, *LPCREATE_THREAD_DEBUG_INFO;
13
14 // 进程创建类型信息
15 typedef struct _CREATE_PROCESS_DEBUG_INFO {
16     HANDLE hFile;
17     HANDLE hProcess;
18     HANDLE hThread;
19     LPVOID lpBaseOfImage;
20     DWORD dwDebugInfoFileOffset;
21     DWORD nDebugInfoSize;
22     LPVOID lpThreadLocalBase;
23     LPTHREAD_START_ROUTINE lpStartAddress;
24     LPVOID lpImageName;
25     WORD fUnicode;
26 } CREATE_PROCESS_DEBUG_INFO, *LPCREATE_PROCESS_DEBUG_INFO;
27
28 // 线程退出类型信息
29 typedef struct _EXIT_THREAD_DEBUG_INFO {
30     DWORD dwExitCode;
31 } EXIT_THREAD_DEBUG_INFO, *LPEXIT_THREAD_DEBUG_INFO;
32
33 // 进程退出类型信息
34 typedef struct _EXIT_PROCESS_DEBUG_INFO {
35     DWORD dwExitCode;
36 } EXIT_PROCESS_DEBUG_INFO, *LPEXIT_PROCESS_DEBUG_INFO;
37
38 // 模块加载类型信息
39 typedef struct _LOAD_DLL_DEBUG_INFO {
40     HANDLE hFile;
41     LPVOID lpBaseOfDll;
42     DWORD dwDebugInfoFileOffset;
43     DWORD nDebugInfoSize;
44     LPVOID lpImageName;
45     WORD fUnicode;
46 } LOAD_DLL_DEBUG_INFO, *LPLOAD_DLL_DEBUG_INFO;
47
48 // 模块卸载类型信息
49 typedef struct _UNLOAD_DLL_DEBUG_INFO {
50     LPVOID lpBaseOfDll;
51 } UNLOAD_DLL_DEBUG_INFO, *LPUNLOAD_DLL_DEBUG_INFO;
52
53 typedef struct _OUTPUT_DEBUG_STRING_INFO {
54     LPSTR lpDebugStringData;

```

```

55     WORD fUnicode;
56     WORD nDebugStringLength;
57 } OUTPUT_DEBUG_STRING_INFO, *LPOUTPUT_DEBUG_STRING_INFO;
58
59 typedef struct _RIP_INFO {
60     DWORD dwError;
61     DWORD dwType;
62 } RIP_INFO, *LPRIP_INFO;

```

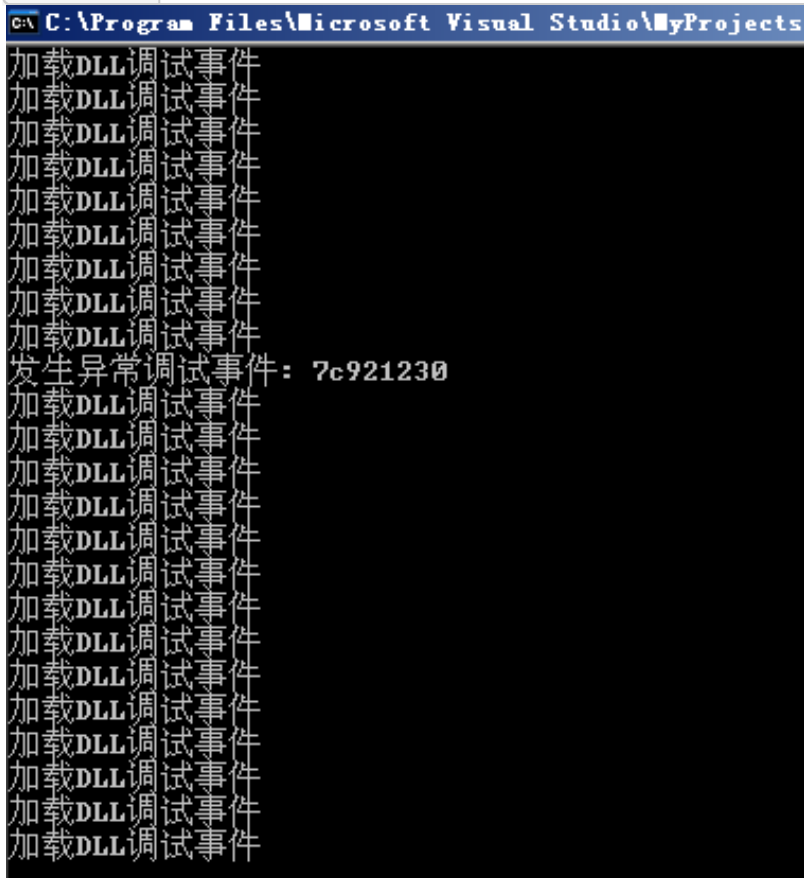
## 额外的异常调试事件

我们可以在上述的简易调试器调试循环判断异常类型的调试事件，将异常发生的地址打印，然后运行代码，就会发现一个被调试进程一切都正常运行，但是产生了一个异常调试事件。

```

1    printf("发生异常调试事件: %x\n",
      debugEvent.u.Exception.ExceptionRecord.ExceptionAddress);

```

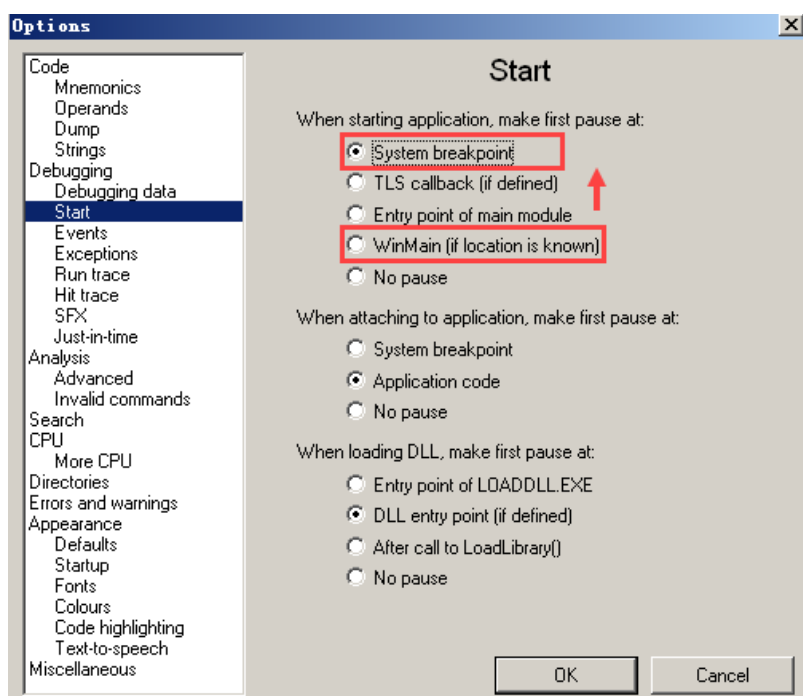


```

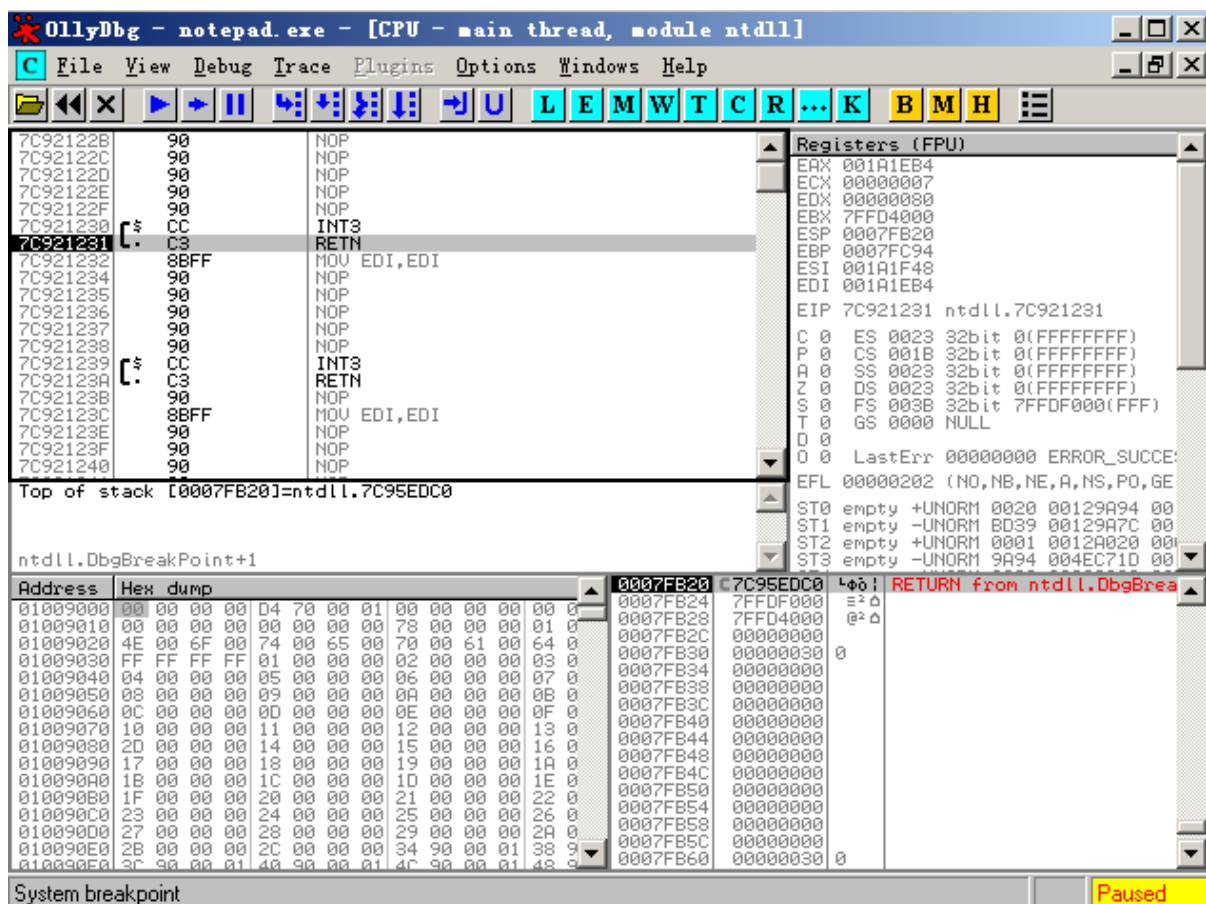
C:\Program Files\Microsoft Visual Studio\MyProjects
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
发生异常调试事件: 7c921230
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件
加载DLL调试事件

```

想要知道这是为什么，我们可以通过OD来打开被调试进程，在这之前我们可以先设置一下打开时的暂停断点，由WinMain变成System breakpoint，这样就不会在进程一打开时就断下来：



然后我们重新打开被调试进程，就会发现程序在我们之前编写的简易调试器所输出的发生异常调试事件地址加一的地址（0x7c921231）进行了断点。



简易调试器输出的地址处的指令刚好是INT 3，也证明了异常调试事件是来源于此。至于为什么会突然出现这个断点，我们需要回顾一下进程的创建过程：

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：
  - 映射DLL（ntdll.LdrInitializeThunk）
  - 线程开始执行

在第5步，系统启动线程进行DLL的映射时会调用ntdll模块中的LdrInitializeThunk函数，我们跟进该函数看一下，在LdrInitializeThunk函数内部会调用\_LdrpInitialize函数，在\_LdrpInitialize函数内部会判断当前创建的是否为第一个线程，如果是就会调用\_LdrpInitializeProcess函数。

```
; __stdcall LdrInitializeThunk(x, x, x, x)
public _LdrInitializeThunk@16
_LdrInitializeThunk@16 proc near          ; DATA XREF: .text:off_7C92395C↓o

arg_0= dword ptr 4
arg_C= byte ptr 10h

lea     eax, [esp+arg_C]
mov     [esp+arg_0], eax
xor     ebp, ebp
jmp     _LdrpInitialize@12              ; LdrpInitialize(x,x,x)

↓

loc_7C938DBF:                          ; CODE XREF: _LdrpInitialize(x,x,x)+26AF6↓j
mov     [ebp+var_20], ebx
; __try { // __finally(loc_7C95F905)
mov     [ebp+ms_exc.registration.TryLevel], ebx
cmp     [esi+0Ch], ebx
jz      loc_7C941616

↓

loc_7C941616:                          ; CODE XREF: _LdrpInitialize(x,x,x)+62↑j
; __unwind { // __SEH_prolog
mov     _LdrpInLdrInit, 1

; __try { // __finally(loc_7C95F905)
; __try { // __except at loc_7C95F8B9
mov     [ebp+ms_exc.registration.TryLevel], 1
push    [ebp+var_28]
push    [ebp+var_2C]
lea     eax, [ebp+var_3C]
push    eax
push    [ebp+arg_4]
push    [ebp+arg_0]
call    _LdrpInitializeProcess@20      ; LdrpInitializeProcess(x,x,x,x,x)
```

接着我们进入 `LdrpInitializeProcess` 函数，这个函数的内容太多，我们只看关键的点。首先，通过将PEB的地址存储到EBX寄存器中。接下来检查PEB结构体中偏移为0x2的位置，即BeingDebugged的值是否为0。如果不等于0，将跳转并调用函数 `DbgBreakPoint`。**DbgBreakPoint函数就是执行了INT 3断点。**

```

mov     _NtDllBase, eax
mov     eax, large fs:18h
mov     ebx, [eax+30h]

loc_7C94165C:                                ; CODE XREF: LdrpInitializeProcess(x,x,x,x,x)+BCD↓j
cmp     byte ptr [ebx+2], 0
jnz     loc_7C95EDBB

loc_7C95EDBB:                                ; CODE XREF: LdrpInitializeProcess(x,x,x,x,x)-45C↑j
call    _DbgBreakPoint@0                    ; DbgBreakPoint()

int     3                                    ; Trap to Debugger
retn

```

那么我们也得出结论，在初始化进程时，会通过检查PEB中的BeingDebugged字段来判断当前进程是否正在被调试。如果当前进程正在被调试，就会为它添加一个INT 3断点。

### 1.3.2 建立调试关系：附加进程

我们可以通过修改建议调试器代码中的部分代码，即 `CreateProcess` 函数的逻辑修改为 `DebugActiveProcess` 函数的逻辑，这样就可以实现以附加进程的方式来建立调试关系。

```

1  DWORD PID;
2  scanf("%d", &PID);
3  if (!DebugActiveProcess(PID))
4  {
5      printf("AttachProcess error:%d\n", GetLastError());
6      getchar();
7      return 0;
8  }

```

在这边我们需要通过手动查看任务管理器的方式找到进程ID，然后在调试器中输入，以达到附加进程调试的目的。

这里我们仍然以 `notepad.exe` 为例进行附加调试，从简易调试器的输出结果来看，使用附加进程方式与创建进程方式并没有什么区别。

[illegible]

但是我们会发现使用创建进程调试时，触发DLL加载类型的调试事件。使用附加进程调试时，也同样的会出现DLL加载类型的调试事件。

按照我们正常对于进程创建的理解，在创建进程后就已经完成了DLL加载，为什么在附加时又进行了DLL的加载呢？为了解决这个问题，我们需要来看一下ntoskrnl模块中的NtDebugActiveProcess函数。

## 虚构的假消息

当我们进入NtDebugActiveProcess函数时，我们可以观察到在将被调试进程与调试对象关联之前，会调用一个名为 DbgkpPostFakeProcessCreateMessages的函数。

从字面意思理解，这个函数的目的很明显，就是向调试器发送一些虚构的假消息。并且此时，被调试进程还未与调试对象关联，因此调试器无法接收来自被调试线程的任何消息。

进入\_DbgkpPostFakeProcessCreateMessages函数后，我们可以发现与线程相关的虚假消息以及与模块相关的虚假消息都会被发送给调试器。显然，当我们通过附加进程的方式建立调试关系时，看到的DLL加载调试事件的情况实际上是由 NtDebugActiveProcess函数发送给调试器的虚构的假消息。



```

lea     eax, [ebp+DebugObject]
push    eax                                ; int
push    [ebp+Process]
push    esi
call    _DbgkpPostFakeProcessCreateMessages@12 ; DbgkpPostFakeProcessCreateMessages(x,x,x)

push    [ebp+DebugObject]                ; 调试对象句柄
push    eax                                ; int
push    [ebp+Process]                    ; 调试对象的地址
push    esi                                ; 被调试进程的 _EPROCESS结构体
call    _DbgkpSetProcessDebugObject@16    ; 将调试对象与被调试进程进行关联

lea     ↓ eax, [ebp+Object]
push    eax                                ; int
lea     eax, [ebp+var_8]
push    eax                                ; int
xor     edi, edi
push    edi                                ; Object
push    [ebp+FastMutex]                  ; FastMutex
push    [ebp+PROCESS]                    ; PVOID
call    _DbgkpPostFakeThreadMessages@20 ; DbgkpPostFakeThreadMessages(x,x,x,x,x)

mov     esi, eax
cmp     esi, edi
jl      short loc_58A760

push    [ebp+FastMutex]                  ; FastMutex
push    [ebp+var_8]                      ; PVOID
push    [ebp+PROCESS]                    ; Object
call    _DbgkpPostFakeModuleMessages@12 ; DbgkpPostFakeModuleMessages(x,x,x)

```

调试器之所以能够观察到被调试进程加载的DLL模块，主要是因为在每个模块加载时，调试事件采集函数会先采集这些事件，并将其发送给调试器。

通过附加进程的方式建立调试关系时，由于无法观察到进程初始化时加载的DLL模块。因此，在执行 NtDebugActiveProcess 函数时，加入了这些虚假消息，希望为调试器提供必要的信息。

但很显然这些虚假消息并不可靠，因为它们主要依赖于 PEB.Ldr 中的三个链表（很容易被修改），这些链表以不同的顺序存储了该进程加载的所有模块。

kd> dt \_PEB\_LDR\_DATA

nt!\_PEB\_LDR\_DATA

```

+0x000 Length           : Uint4B
+0x004 Initialized      : UChar
+0x008 SsHandle         : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress   : Ptr32 Void

```



## 1.4 异常的处理流程

在之前异常（用户异常的分发与内核异常的分发过程）章节的学习中，我们分析了KiDispatchException函数的执行流程。因此，我们知道在异常分发时，首先会检查调试器是否存。那么本文将在存在调试器和不存在调试器的情况下，验证异常分发的流程。

### 1.4.1 调试器下的异常分发

在异常章节的学习中，我们通过分析KiDispatchException函数知道在异常分发时会检查一下是否有内核调试器，但在当时我们并没有学习软件调试的，因此在这里我们可以加入调试来分析一下有无调试器时异常的分发流程。

如下代码所示我们构建一个除零异常，直接执行的话，正常流程就会进入\_except分支执行代码：

```
1  #include "stdio.h"
2
3  int main()
4  {
5      int x = 100;
6      int y = 0;
7
8      _try
9      {
10         int a = x / y;
11         printf("Error");
12     }
13     _except(1)
14     {
15         printf("Except");
16     }
17
18     getchar();
19
20     return 0;
21 }
```

```

#include "windows.h"
#include "stdio.h"

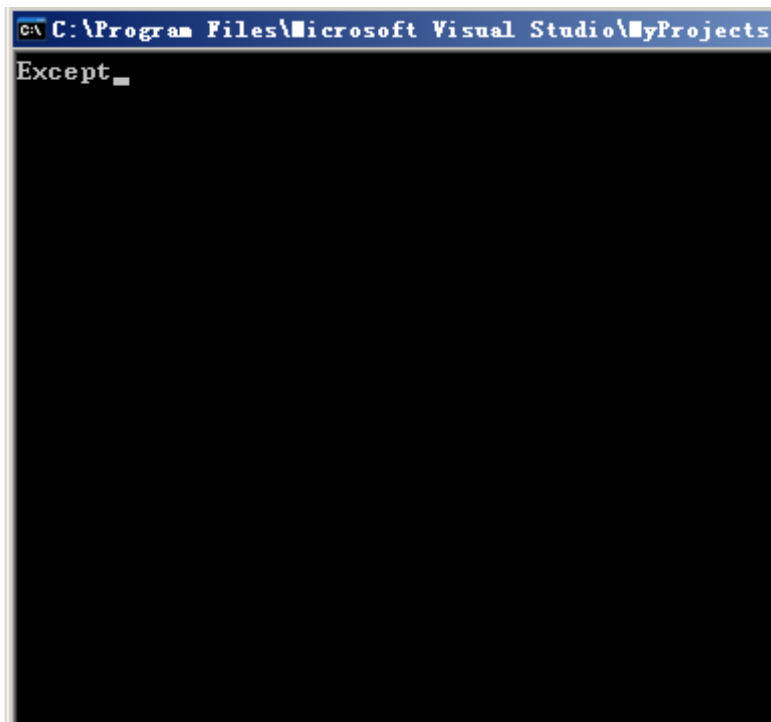
int main()
{
    int x = 100;
    int y = 0;

    _try
    {
        int a = x / y;
        printf("Error");
    }
    _except(1)
    {
        printf("Except");
    }

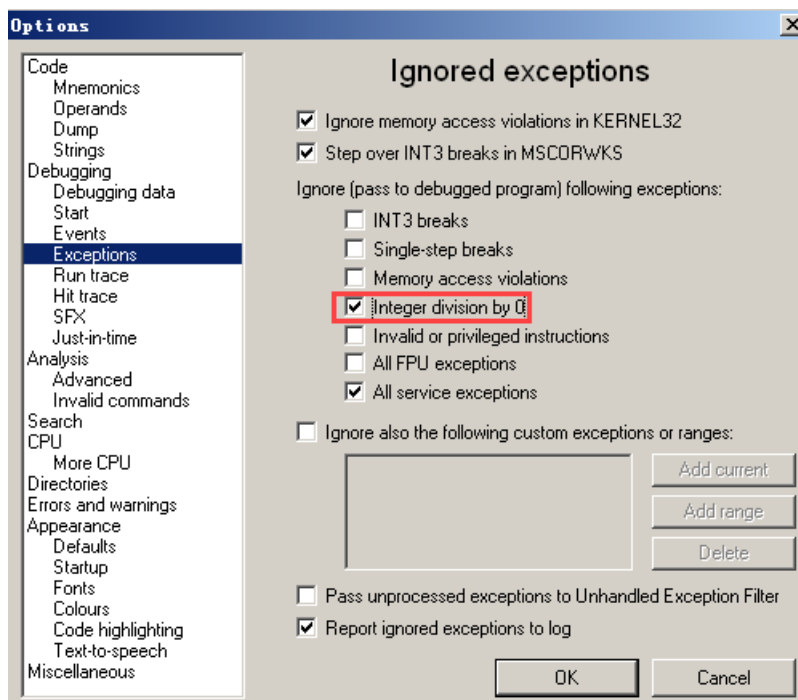
    getchar();

    return 0;
}

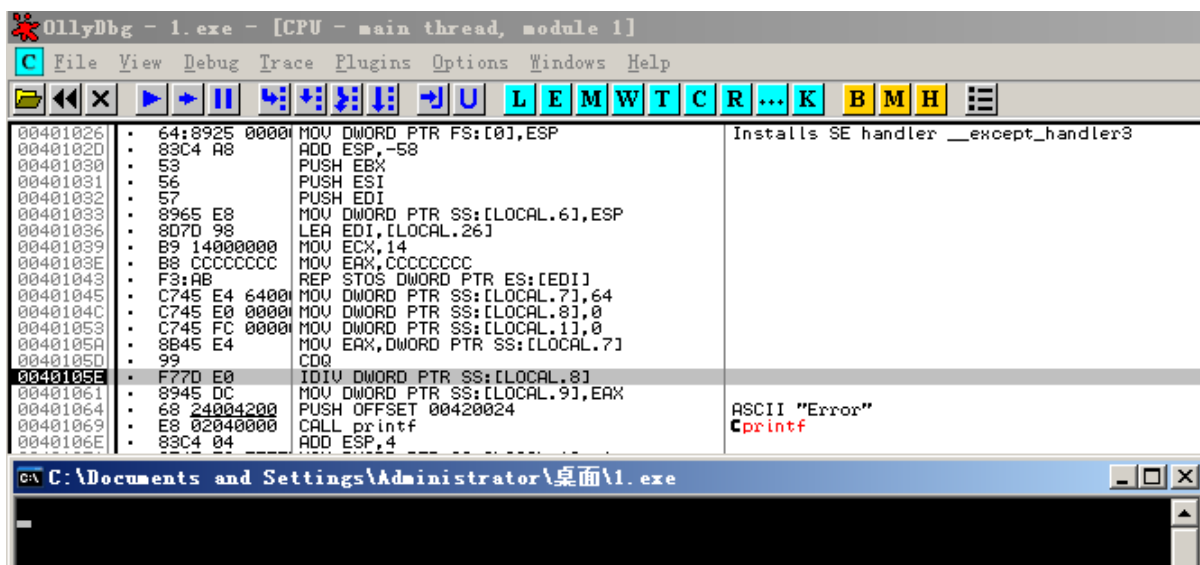
```



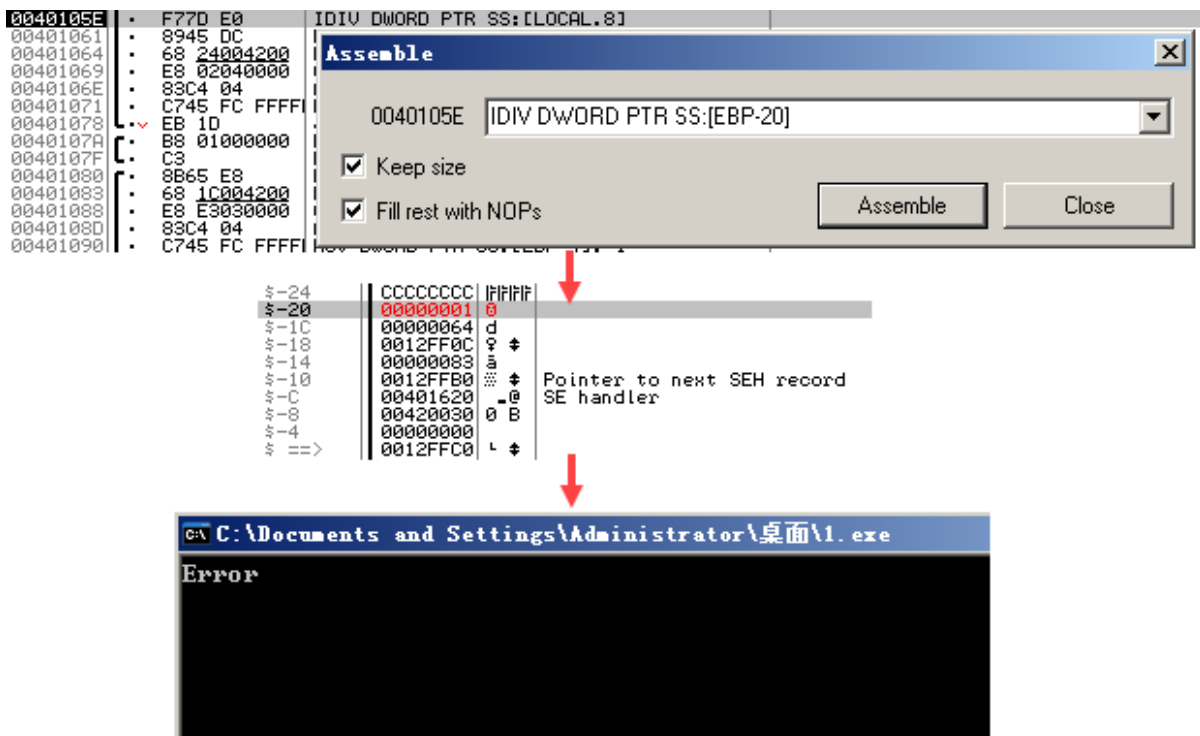
接着我们可以通过OD来调试该可执行文件，这里需要注意的是我们需要将OD设置中的忽略除零异常给取消勾选，如果勾选上，OD调试器就会在遇到除零异常时执行ContinueDebugEvent，使得被调试进程继续执行。



接着我们来调试，当调试器断点到除零异常时，就会发现无论我们怎么F9，断点依然停留在这里，这是因为调试器并没有处理这个异常。



那我们就要在这里手动修改被除数为1，然后再次F9，就会顺利的执行\_try分支内的代码。



### 1.4.2 最后一道防线与二次分发

我们知道无论是进程的入口线程还是另起的线程，都会被添加的异常处理函数给处理掉，这是因为操作系统添加了一道最后的防线，其伪代码如下：

1	__try
2	{

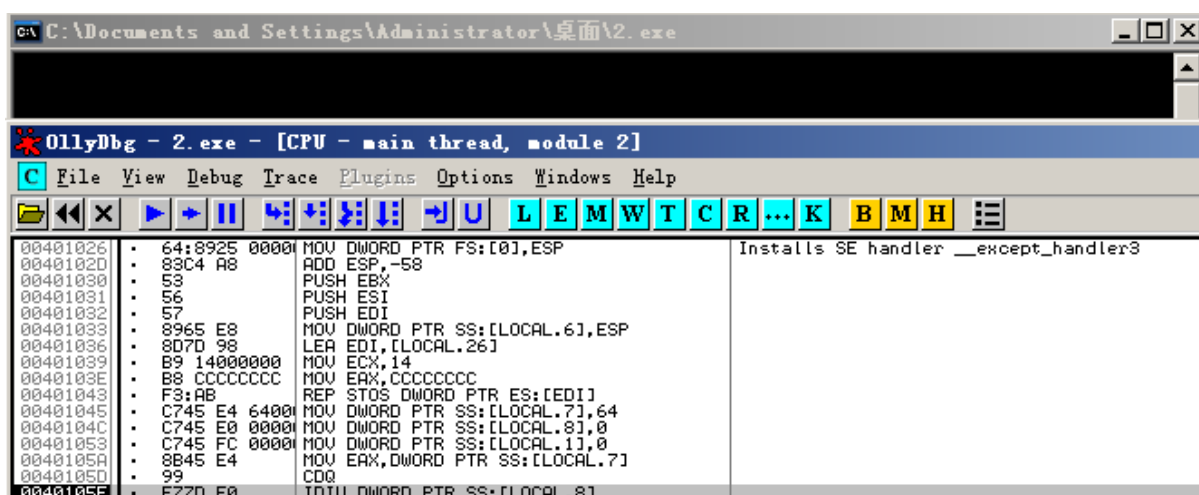
```

3
4 }
5 __except(UnhandledExceptionFilter(GetExceptionInformation()))
6 {
7     //终止线程
8     //终止进程
9 }

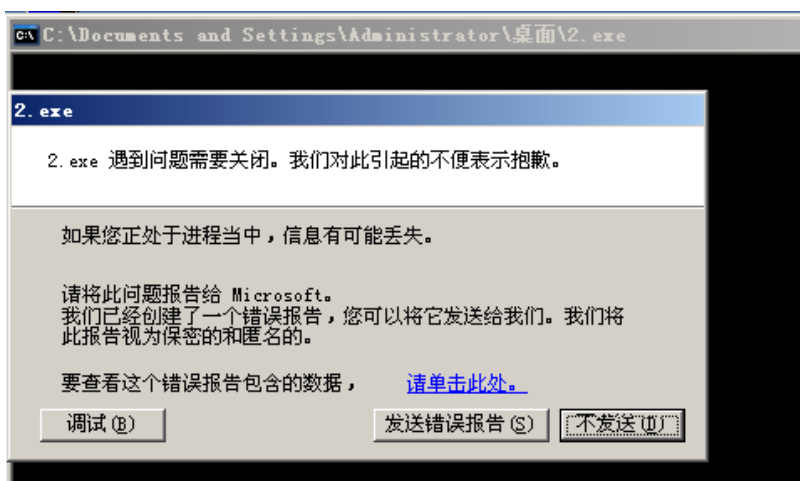
```

如果UnhandledExceptionFilter函数返回为0，即EXCEPTION\_CONTINUE\_SEARCH，那就真的是找不到对应的异常处理程序了，这种情况下只有在程序被调试时才会存在。

因此我可以将除零异常代码中的\_except(1)修改为\_except(0)，表示没有对应的异常处理程序，也不去注册。我们再通过OD来调试会发现，即使我们忽略了除零异常，也仍然会在异常处断点。这是因为在第一次的异常分发时VEH、SEH、调试器都没有处理异常，最后一道防线中进入UnhandledExceptionFilter函数，在该函数内检测到此时存在调试器，就会进行第二次的异常分发，也就是给到调试器。

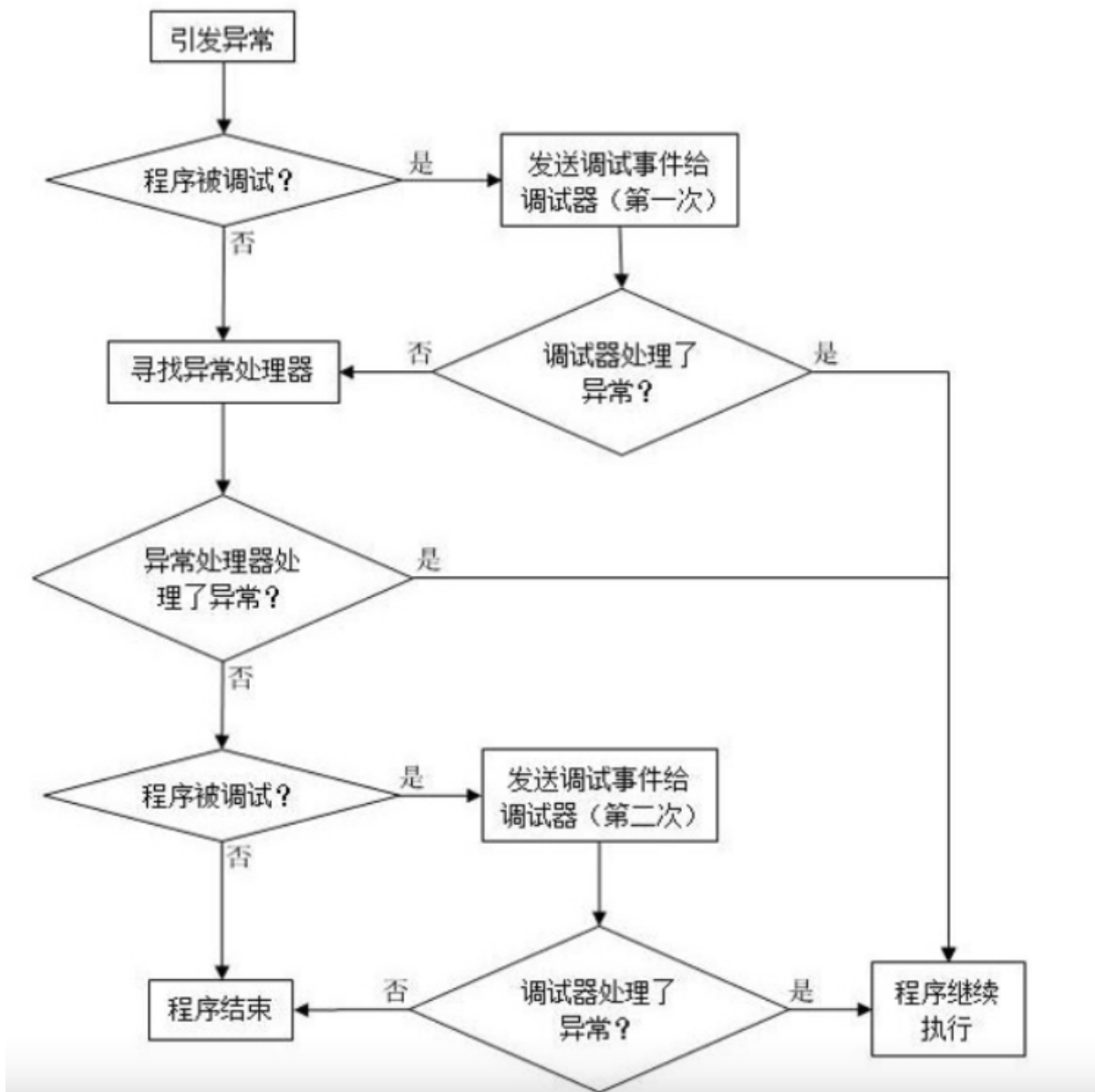


而如果没有调试器，正常打开该程序时，由于没有被调试，最后一道防线会查询当前是否有通过SetUnhandledExceptionFilter函数去注册异常处理函数，如果有就调用，没有的话Windows就会弹出窗口让用户选择终止程序还是启动即时调试器。



### 1.4.3 总结

一张图来总结包含了调试器的异常处理流程。



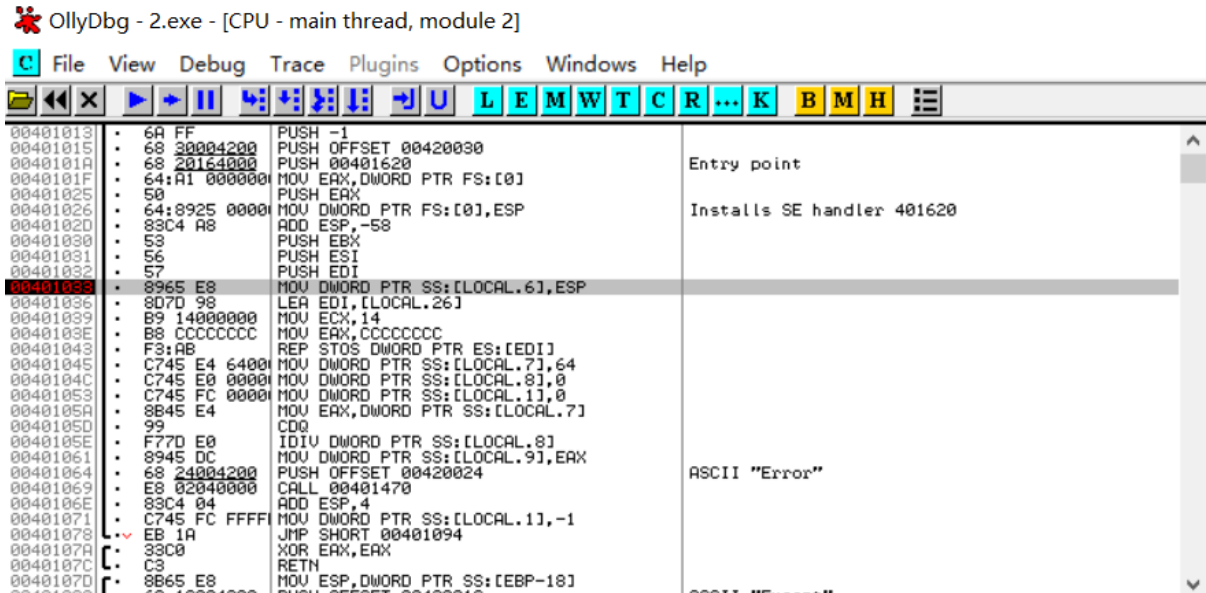
### 1.5 三大断点

所谓调试，其实就是在被调试进程中想法设法的去触发异常，当异常产生后，由调试器来接管异常。**有三种触发异常的方法：软件断点、内存断点、硬件断点。**

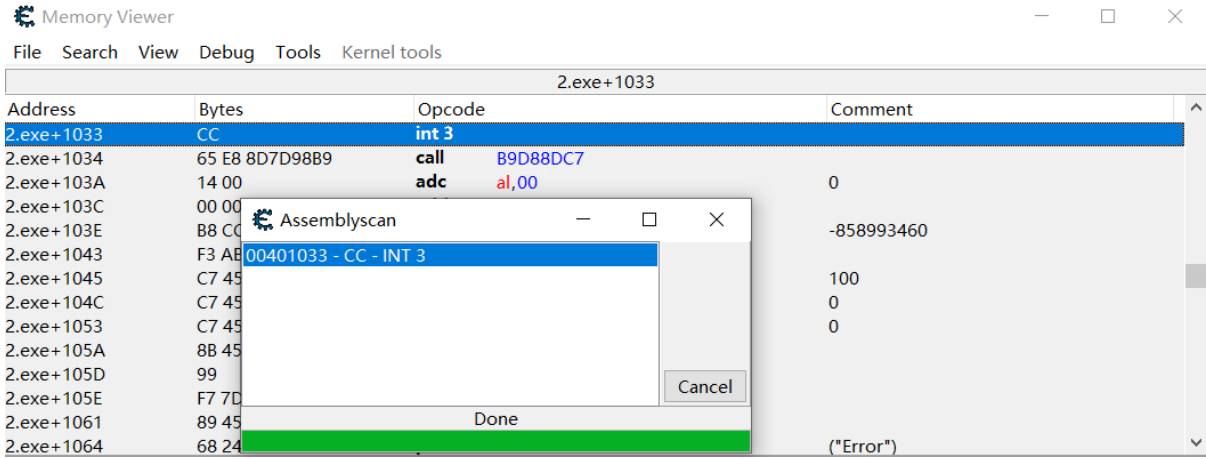
1.5.1 软件断点

软件断点就是我们所熟悉的INT 3指令，软件断点的实现就是将下断点的地址处硬编码指令修改为0xCC。

这里我们可以先使用OD随便一处下断点，然后等程序停在对应位置，如下图所示指令并没有被修改为INT 3，这是因为OD为了更好的用户体验展示的时候保留了原指令。



实际上我们通过其他根据来看，例如下图所示的CE，对应地址处的指令确实变成了INT 3。



执行流程

触发软件断点的过程，实际上就是CPU异常分发的过程：CPU检测到异常（INT 3指令），查询中断描述符表，找到对应的中断处理函数（3号），中断处理函数内部会调用CommonDispatchException函数，在CommonDispatchException函数内部又会调用KiDispatchException函数。

这个流程我们在异常章节的学习中就已经了解了，那现在我们进入KiDispatchException函数内，之前在异常章节的学习时分析过这个函数，这里我们仅文字描述一下流程，不再重复的截图解释。

由于当前异常来自用户空间，所以我们直接看用户异常的处理流程，**这里不管有没有内核调试器，或内核调试器函数返回结果为0，都会跳转到如下图所示的代码片段**，然后调用\_DbgkForwardException函数，将异常发送给3环调试器。

```
loc_42B70B:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+53CF↑j
                                           ; KiDispatchException(x,x,x,x,x)+12BC6↓j
push     edi
push     1
push     esi
call     _DbgkForwardException@12          ; _DbgkForwardException有三个参数：
                                           ; 1. 异常结构体
                                           ; 2. 调试端口：TRUE，异常端口：FALSE
                                           ; 3. 是否进行第二次分发：TRUE/FALSE

test     al, al
jnz      loc_4263F3                        ; 如果3环调试器处理结束，则跳转走

mov      [ebp+var_3A0], edi                ; 如果没有3环或0环调试，或都未处理，则执行到这
```

在\_DbgkForwardException函数内，最终会调用\_DbgkpSendApiMessage函数，这个函数我们在“调试事件的采集”篇的学习中已经了解，它的作用就是将已创建的调试事件发送到调试对象的事件链表中。

```
loc_58B47C:                                ; CODE XREF: DbgkpSendApiMessage(x,x)+53↑j
mov      eax, esi
pop      esi
pop      ebx
pop      ebp
retn     8

_DbgkpSendApiMessage@8 endp
```

进入\_DbgkpSendApiMessage函数，我们会发现它立刻就判断了传递过来的第二个参数是否为0，如果不为0，则会调用\_DbgkpSuspendProcess函数将当前进程（被调试进程）内除当前线程外的其他线程挂起，当前例子中的INT 3引起的异常就会被挂起。

```

; __stdcall DbgkpSendApiMessage(x, x)
_DbgkpSendApiMessage@8 proc near
; CODE XREF: DbgkCreateThread(x)+3DEE6↑p
; DbgkCreateThread(x)+3E000↑p
; DbgkCreateThread(x)+3E075↑p
; DbgkForwardException(x,x,x)+889A8↑p
; DbgkMapViewOfSection(x,x,x,x)+94553↑p
; DbgkUnMapViewOfSection(x)+9ECA9↑p
; DbgkExitThread(x)+62↓p
; DbgkExitProcess(x)+6C↓p

arg_0= dword ptr 8
arg_4= byte ptr 0Ch

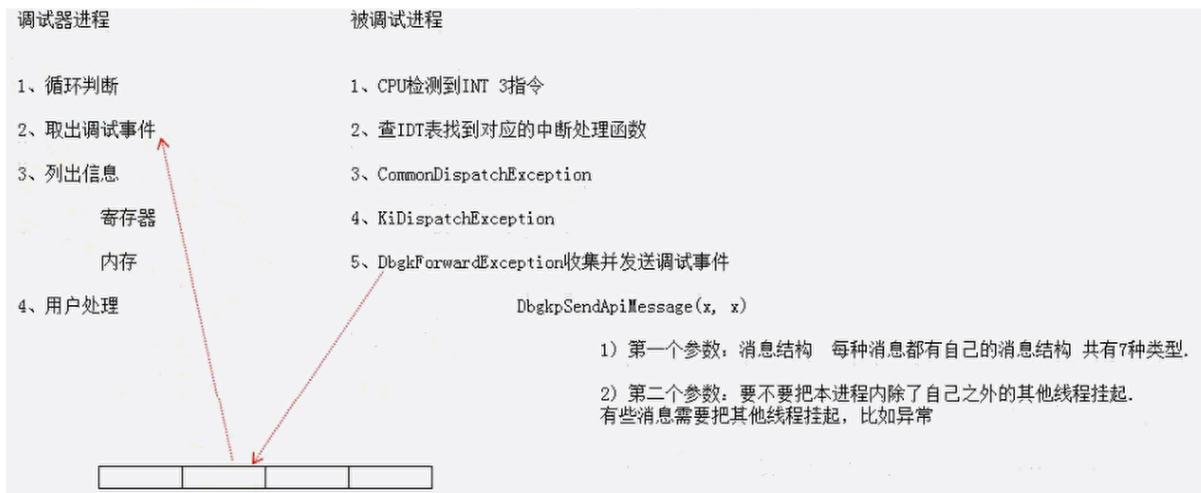
mov     edi, edi
push    ebp
mov     ebp, esp
push    ebx
xor     ebx, ebx
cmp     [ebp+arg_4], bl           ; 判断第二个参数值是否为0
push    esi                     ; 为0就跳转
jz      short loc_58B438         ; 不为0就挂起

call    _DbgkpSuspendProcess@@0

```

在挂起后，调试事件会被发送给调试对象，并在调试循环中被调试器提取。调试器将使用异常调试事件的结构体，列出相关信息，例如当前寄存器的值和内存情况。然后，调试器的用户可以对这些信息进行处理。

总体流程可以参考下图：



## 实验代码

刚刚我们是从一个被调试器角度了解软件断点的执行流程，那么从调试器角度又如何去实现软件断点，这值得我们去关注。首先在这里，为了实现软件断点功能，需要手动编写一个SetInt3BreakPoint函数，具体实现如下代码所示，我们可以在处理创建进程调试事件时，下该断点：



```

1  VOID SetInt3BreakPoint(PCHAR pAddress)
2  {
3      CHAR cInt3 = 0xCC;
4
5      // 1. 备份原始代码
6      ReadProcessMemory(hDebuggeeProcess, pAddress, &OriginalCode, 1, NULL);
7
8      // 2. 设置软件断点
9      WriteProcessMemory(hDebuggeeProcess, pAddress, &cInt3, 1, NULL);
10 }

```

接着我们需要一个INT 3软件断点的处理函数，下面仅列出相关代码，其每一步的意义如下：

1. 由于软件断点会修改原指令，因此在重新执行之前需要恢复原硬编码指令。在这里判断了当前的INT 3指令是否为系统断点，如果是系统断点，则无需修复。IsSystemInt3函数需要自行实现。如果不是系统断点则通过调试事件来获取对应的异常地址，然后将原指令写入回去。
2. 显示断点地址，便于使用者更清晰的知道当前行为所在位置。
3. 获取线程的上下文环境，一旦获得了线程的上下文环境，就可以获取当前状态下各个寄存器的值。
4. 修复EIP寄存器的值，是对于不同类型的断点，在断点触发后，EIP寄存器的位置会有所不同。对于软件断点INT 3来说，断点触发后，EIP位于原始地址的下一个字节位置，因此需要将EIP减去1来修复它。
5. 显示反汇编代码，在常规调试器中，能够实时查看程序的反汇编代码至关重要。因此，在触发断点后，至少要显示断点周围的反汇编代码。用户知道的信息越多自然对调试就越有帮助。
6. 等待用户命令，调试器的最主要的作用就是可以对代码进行调试，包括但不限于单步执行、逐行执行、继续执行等操作。在这里，通过一个循环来等待用户执行命令，如果用户没有执行命令，就一直等待下去。WaitForUserCommand函数需要自行实现，在后续单步学习时候会了解到。

```

1  BOOL Int3ExceptionProc(EXCEPTION_DEBUG_INFO* pExceptionInfo)
2  {
3      BOOL bRet = FALSE;
4      CONTEXT Context;
5
6      // 1. 如果是系统断点，不需要修复INT3
7      if (IsSystemInt3(pExceptionInfo))
8      {
9          return TRUE;
10     }
11     else
12     {
13         WriteProcessMemory(hDebuggeeProcess, pExceptionInfo->ExceptionRecord.ExceptionAddress, &OriginalCode, 1, NULL);
14     }
15
16     // 2. 显示断点位置
17     printf("INT3断点地址: 0x%p \n", pExceptionInfo->ExceptionRecord.ExceptionAddress);
18
19     // 3. 获取线程上下文
20     Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
21     GetThreadContext(hDebuggeeThread, &Context);
22
23     // 4. 修复EIP

```

```

24     Context.Eip--;
25     SetThreadContext(hDebuggeeThread, &Context);
26
27     // 5. 显示反汇编
28
29     // 6. 等待用户命令
30     while (!bRet)
31     {
32         bRet = WaitForUserCommand();
33     }
34
35     return bRet;
36 }

```

## 1.5.2 内存断点

软件断点修改的是指令，内存断点修改的是物理页的属性。例如当某一地址，下了内存断点，当该地址被读取、写入时都会断下来。调试器进程通过调用VirtualProtectEx函数来跨进程对被调试进程物理页属性进行修改。

VirtualProtectEx函数的语法格式如下，其中需要关注的是flNewProtect参数，该参数取决定了修改的物理页的新属性：

```

1  BOOL VirtualProtectEx(
2      HANDLE hProcess,           // 要修改内存的进程句柄
3      LPVOID lpAddress,         // 要修改内存的起始地址
4      SIZE_T dwSize,            // 页区域大小
5      DWORD flNewProtect,       // 新物理页属性
6      PDWORD lpflOldProtect     // 原物理页属性，用于保存改变前的属性
7  )

```

这边我们需要注意的是，由于VirtualProtectEx函数修改的并不是某一地址的属性，而是地址所在物理页的属性，因此在对应物理页内的地址都有可能出现异常，所以需要调试器去判断接收到的调试事件中的对应地址是否是下内存断点的地址，如果不是的话就应该放行。

## 执行流程

下完内存断点后，当被调试进程试图访问或写入被修改属性后的物理页时，就会触发页异常。然后就进行了异常的处理流程：

1. CPU访问或写入访问或写入被修改属性后的物理页，触发页异常；
2. 查IDT表找到对应的中断处理函数（这里页异常是0x0E号中断）；
3. 调用CommonDispatchException函数；
4. 调用KiDispatchException函数；
5. 进入DbgkForwardException函数，收集调试事件并发送给调试对象；
6. 最后交给调试器来处理。

这个流程其实跟软件断点的流程差不多，因此就不必再重复的去跟进代码了。

## 实验代码

从调试器角度，我们第一步仍然是要下断点，在处理创建进程调试事件时，选择要下的断点类型，以下只是示例代码，具体的要根据不同的需求进行分支的判断和调整：

```

1  VOID SetMemBreakPoint(PCHAR pAddress)
2  {
3      DWORD dwOriginalProtect; // 保存原物理页属性
4      SIZE_T dwSize = 1;
5
6      // 设置访问断点
7      VirtualProtectEx(hDebuggeeProcess, pAddress, dwSize, PAGE_NOACCESS,
8                      &dwOriginalProtect);
9
10     // 设置写入断点
11     VirtualProtectEx(hDebuggeeProcess, pAddress, dwSize, PAGE_EXECUTE_READ,
12                     &dwOriginalProtect);
13 }

```

当异常触发之后，调试器会收到异常调试事件，因此就需要判断调试事件的类型，异常类型的调试事件结构体为\_EXCEPTION\_DEBUG\_INFO（调试循环时即可获取：\_DEBUG\_EVENT.u.Exception），该结构体的第一个成员为ExceptionRecord，其也是一个结构体EXCEPTION\_RECORD：

```

1  typedef struct _EXCEPTION_RECORD {
2      DWORD ExceptionCode;
3      DWORD ExceptionFlags;
4      struct _EXCEPTION_RECORD *ExceptionRecord;
5      PVOID ExceptionAddress;
6      DWORD NumberParameters;
7      UINT_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
8  } EXCEPTION_RECORD;

```

在该结构体中ExceptionCode成员表示异常的类型，类型有很多种，如下图所示，STATUS\_ACCESS\_VIOLATION（0xC0000005，表示访问违例）这就是内存断点引发的异常类型。

```

#define STATUS_WAIT_0 ((DWORD) 0x00000000L)
#define STATUS_ABANDONED_WAIT_0 ((DWORD) 0x00000080L)
#define STATUS_USER_APC ((DWORD) 0x000000C0L)
#define STATUS_TIMEOUT ((DWORD) 0x00000102L)
#define STATUS_PENDING ((DWORD) 0x00000103L)
#define STATUS_SEGMENT_NOTIFICATION ((DWORD) 0x40000005L)
#define STATUS_GUARD_PAGE_VIOLATION ((DWORD) 0x80000001L)
#define STATUS_DATATYPE_MISALIGNMENT ((DWORD) 0x80000002L)
#define STATUS_BREAKPOINT ((DWORD) 0x80000003L)
#define STATUS_SINGLE_STEP ((DWORD) 0x80000004L)
#define STATUS_ACCESS_VIOLATION ((DWORD) 0xC0000005L)
#define STATUS_IN_PAGE_ERROR ((DWORD) 0xC0000006L)
#define STATUS_INVALID_HANDLE ((DWORD) 0xC0000008L)
#define STATUS_NO_MEMORY ((DWORD) 0xC0000017L)
#define STATUS_ILLEGAL_INSTRUCTION ((DWORD) 0xC000001DL)
#define STATUS_NONCONTINUABLE_EXCEPTION ((DWORD) 0xC0000025L)
#define STATUS_INVALID_DISPOSITION ((DWORD) 0xC0000026L)
#define STATUS_ARRAY_BOUNDS_EXCEEDED ((DWORD) 0xC000008CL)
#define STATUS_FLOAT_DENORMAL_OPERAND ((DWORD) 0xC000008DL)
#define STATUS_FLOAT_DIVIDE_BY_ZERO ((DWORD) 0xC000008EL)

```

因此我们可以通过switch分支来判断异常类型，从而进一步的处理相关异常引起的断点。

```

1  swtich(pExceptionInfo->ExceptionRecord.ExceptionCode)
2  {
3      // 软件断点 (异常断点)
4      case STATUS_BREAKPOINT:
5          bRet = Int3ExceptionProc(pExceptionInfo);
6          break;
7
8      // 内存断点 (访问违例)
9      case STATUS_ACCESS_VIOLATION:
10         bRet = AccessExceptionProc(pExceptionInfo);
11         break;
12 }

```

接着我们就需要处理内存断点了，如下代码所示，这里的逻辑跟软件断点的处理逻辑差不多，唯一的区别就是第一步获取异常信息，在ExceptionRecord结构体中，有一个数组ExceptionInformation，根据MSDN Library的说明，该数组第一个成员用于表示异常的原因（值为0表示有线程试图读取，值为1表示有线程试图写入）。该数组第二个成员用于标识导致异常的虚拟地址。由于内存断点是针对整个物理页设置的，因此触发异常的位置可能不是我们设置断点的实际位置。因此，在此处通过第二个成员对地址进行检查，以确定是否为我们设置断点的位置。如果不是，则直接继续执行，如果是，则进行相应处理。

```

1  BOOL AccessExceptionProc(EXCEPTION_DEBUG_INFO* pExceptionInfo)
2  {
3      BOOL bRet = FALSE;
4      CONTEXT Context;
5      DWORD dwAccessFlag;

```

```

6     DWORD dwAccessAddr;
7     DWORD dwUnused;
8
9     // 1. 获取异常信息, 修改内存属性
10    dwAccessFlag = pExceptionInfo->ExceptionRecord.ExceptionInformation[0]
11    ;
12    dwAccessAddr = pExceptionInfo->ExceptionRecord.ExceptionInformation[1]
13    ;
14    printf("内存断点 %x 0x%p\n", dwAccessFlag, dwAccessAddr);
15    VirtualProtectEx(hDebuggeeProcess, (PVOID)dwAccessAddr, 1,
16    dwOriginalProtect, &dwUnused);
17
18    // 2. 获取线程上下文
19    Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
20    GetThreadContext(hDebuggeeThread, &Context);
21
22    // 3. 修复EIP, 内存断点不需要修复EIP, 软件断点需要
23
24    // 4. 显示反汇编
25
26    // 5. 等待用户命令
27    while (bRet == FALSE)
28    {
29        bRet = WaitForUserCommand();
30    }
31    return bRet;
32 }

```

### 1.5.3 硬件断点

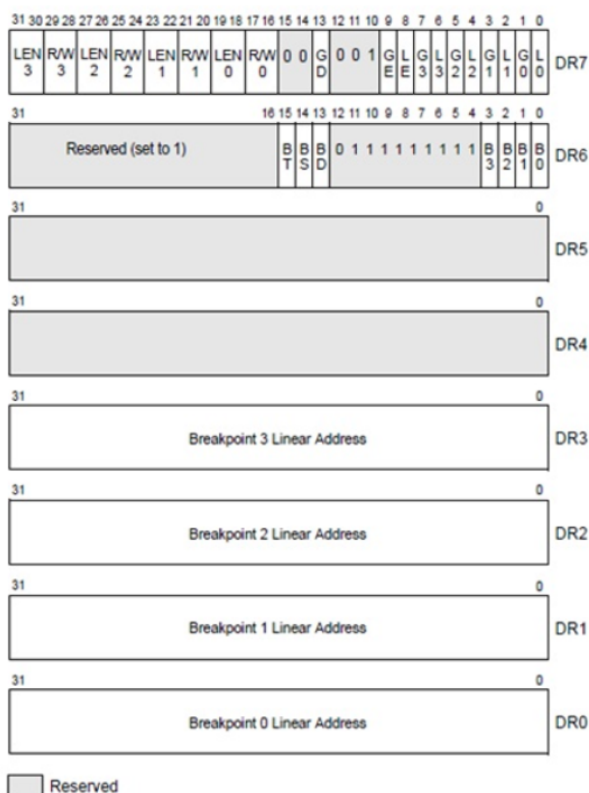
对于软件断点, 可以使用CRC校验来进行检测; 对于内存断点, 可以创建一个线程来不断刷新PTE的属性, 以防止其被修改。而本篇要介绍的硬件断点比较难以防范, 因为不依赖于修改被调试进程的数据, 所以也值得我们深入学习。

硬件断点的实现需要使用调试寄存器DR0到DR7, 它们的结构如下:

Dr0-3用于设置硬件断点, 也就是存储对应的线性地址, 由于只有4个断点寄存器, 所以最多只能设置4个硬件调试断点。Dr4-5是保留的, 我们可以不用看。

Dr7是最重要的寄存器, 它控制着断点的各类属性, 有很多个位, 每位的意思如下 (**0-3都是与Dr0-3为对应关系, 如Dr0对应L0、G0、LEN0、R/W0**) :

1. L0/G0 - L3/G3: 控制Dr0-3寄存器是否有效, 确定是局部 (Lx) 还是全局 (Gx) 断点。每次异常触发后, Lx都被清零, Gx不清零。当你向Dr0-3存储线性地址后, Lx或Gx必须有一个为1, 才会触发。
2. 断点长度 (LENx) : 00 (1字节), 01 (2字节), 11 (4字节)。
3. 断点类型 (R/Wx) : 00 (执行断点), 01 (写入断点), 11 (访问断点)。



硬件调试断点产生的异常是STATUS\_SINGLE\_STEP（单步异常）。除了硬件断点外，当Eflags的TF标志位置为1时，也会产生单步异常。Dr6寄存器的作用是**确定产生的是哪种类型的单步异常**。当B0-3中有值时，可以确定是某个硬件断点触发了单步异常。**如果B0-3的值都为空，那么说明是由Eflags的TF标志位为1引起的单步异常。**

在设置断点时，我们需要将要断点的线性地址写入Dr0-3中的任意一个寄存器。当CPU执行到该线性地址时，如果发现与调试寄存器中的值相同，就会触发断点异常，进而暂停执行。需要注意的是，设置断点是修改当前线程Context中记录的调试寄存器的值，不同线程之间是相互隔离的，因此设置硬件断点不会影响其他线程的执行。

## 执行流程

下完硬件断点后，当CPU检测调试寄存器（Dr0-3）到执行到对应的线性地址，就会触发单步异常。然后就进行了异常的处理流程：

1. CPU检测调试寄存器（Dr0-3）到执行到对应的线性地址，就会触发单步异常；
2. 查IDT表找到对应的中断处理函数（这里是0x01号中断）；
3. 调用CommonDispatchException函数；
4. 调用KiDispatchException函数；
5. 进入DbgkForwardException函数，收集调试事件并发送给调试对象；
6. 最后交给调试器来处理。

这个流程其实跟软件断点的流程差不多，因此就不必再重复的去跟进代码了。

## 实验代码

硬件断点与软件断点或内存断点有所不同。软件断点和内存断点可以在OEP（原始执行点）处设置断点，但是硬件断点不能在OEP处设置断点，因为此时主线程还未创建出来（参考使用CreateProcess函数创建调试关系时所产生的调试事件）。

硬件断点是基于线程的，没有线程的情况下无法触发硬件断点。因此，可以采用另一种方法，在OEP处设置一个软件断点。当软件断点被触发时，将进入软件断点处理函数，这样就会创建一个线程，因此我们可以在软件断点处理函数中来设置硬件断点（可以在OEP+1处设置），这样就可以触发硬件断点。硬件断点的实现如下所示（省略了软件断点处的代码）：

```

1  VOID SetHardBreakPoint(PVOID pAddress)
2  {
3      CONTEXT Context;
4      // 获取线程上下文
5      Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
6      GetThreadContext(hDebuggeeThread, &Context);
7
8      // 设置断点位置（使用Dr0作为断点寄存器）
9      Context.Dr0 = (DWORD)pAddress;
10     Context.Dr7 |= 1;
11
12     // 设置断点长度
13     Context.Dr7 &= 0xffff0fff;
14
15     // 设置线程上下文
16     SetThreadContext(hDebuggeeThread, &Context);
17 }

```

由于单步异常存在两种情况，因此在处理函数内部需要进行判断，以确定是否是由硬件断点引起的异常。

```

1  BOOL SingleStepExceptionProc(EXCEPTION_DEBUG_INFO* pExceptionInfo)
2  {
3      CONTEXT Context;
4      BOOL bRet = FALSE;
5
6      // 1. 获取线程上下文
7      Context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
8      GetThreadContext(hDebuggeeThread, &Context);
9
10     // 2. 判断是否是硬件断点导致的异常
11     if (Context.Dr6 & 0xF) // B0-3不为空
12     {
13         // 显示硬件断点信息
14         printf("硬件断点 %d 0x%p\n", Context.Dr7 & 0x00030000,
Context.Dr0);
15
16         // 移除断点
17         Context.Dr0 = 0;
18         Context.Dr7 &= 0xfffffff;

```

```

19     }
20     else
21     {
22         // 显示单步异常信息
23         printf("单步异常 0x%p\n", Context.Eip);
24
25         // 清除单步标志
26         Context.EFlags &= 0xffffeff;
27     }
28
29     // 3. 等待用户命令
30     while (!bRet)
31     {
32         bRet = WaitForUserCommand();
33     }
34
35     return bRet;
36 }

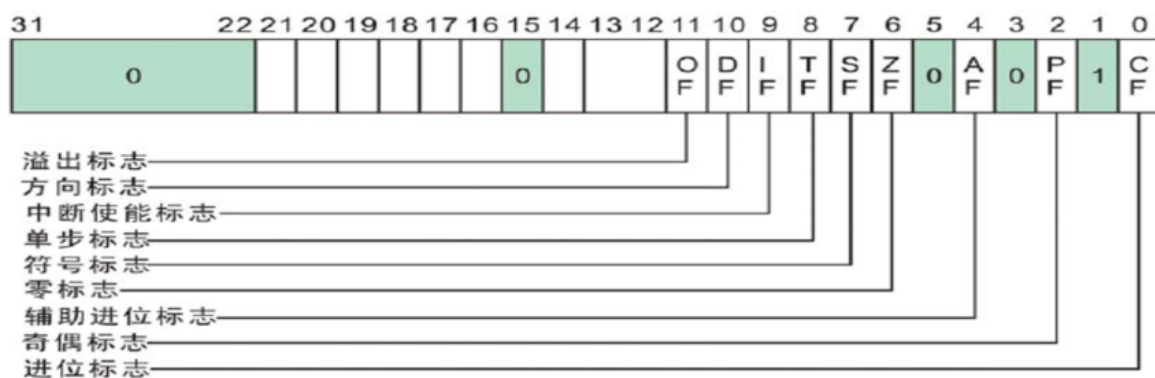
```

## 1.6 单步步入和步过

所谓单步步入和步过，实际上就是OD调试器的快捷键F7和F8，单步步入是可以每一行指令进行单步执行，单步步过也是如此，不同的是单步步过不会进入调用函数的内部（CALL指令）。

### 1.6.1 步入

想要实现单步步入有很多种方式，例如我们可以将每一行要执行的指令下一地址的指令中设为INT 3，然后单步执行后就恢复再以此类推去设置。但是这样的方式很笨拙，因此Intel在设计CPU时考虑到了调试程序的必要性，就在EFlags寄存器中设置了一个TF位。TF位我们可以称之为单步标志，当TF位被置1时，每执行完一行指令就会产生一个异常。



单步异常的处理流程与硬件断点一致，都是CPU检测到异常后通过IDT表找到0x01号中断函数进行处理等等，因此此处就不再赘述。



## 实验代码

单步步入（异常）的断点实现也很简单，就是获取线程上下文，就得到了相关的寄存器，接着将EFlags寄存器的TF位置1，然后再将上下文设置回去。

```
1  VOID SetSingleStep()
2  {
3      CONTEXT Context;
4      Context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
5      GetThreadContext(hDebuggeeThread, &Context);
6      // 将EFlags寄存器的TF位置1
7      Context.EFlags |= 0x100;
8      SetThreadContext(hDebuggeeThread, &Context);
9  }
```

由于单步步入与硬件断点触发的异常都属于单步异常，因此这两种异常可以使用同一个处理函数，只需对Dr6的值进行判断即可区分，这部分代码与硬件断点的实现代码一致，所以也不再赘述。

### 1.6.2 步过

想要实现单步步过，可以通过两种方式：硬件断点、软件断点。实现的原理是在遇到CALL指令（包括多种类型）后，计算当前指令的长度，并在当前EIP+当前指令长度的位置设置断点，即CALL指令的下一行，然后继续执行，从而实现单步步过的效果。