

1 异常

如果你希望在软件调试上有所突破，或者想了解如何通过异常进行反调试，或者想自己写一个调试器，那么就必须深入了解异常，异常与调试是紧密相连的，异常是调试的基础。

异常产生后，首先是要记录异常信息（异常的类型、异常发生的位置等），然后要寻找异常的处理函数，我们称为异常的分发，最后找到异常处理函数并调用，我们称为异常处理。我们后续的学习，也是围绕异常的、分发、处理。

1.1 异常记录

异常可以简单分为2类，即**CPU产生的异常**和**软件模拟产生的异常**，如下两张图所示，我们可以看见第一张图中进行了除法运算，CPU检测到除数为0，就产生了异常；第二张图中使用了throw关键词，通过软件模拟主动产生了异常。

```
int main(int argc, char* argv[])
{
    // CPU
    int a = 10;
    int b = 0;
    printf("%d", a / b);
}
```



```
int main(int argc, char* argv[])
{
    // CPU
    // int a = 10;
    // int b = 0;
    // printf("%d", a / b);

    // Software
    throw 1;
}
```



1.1.1 CPU的异常记录

我们先了解一个结构体_EXCEPTION_RECORD，它的格式及每个成员的意义如下：

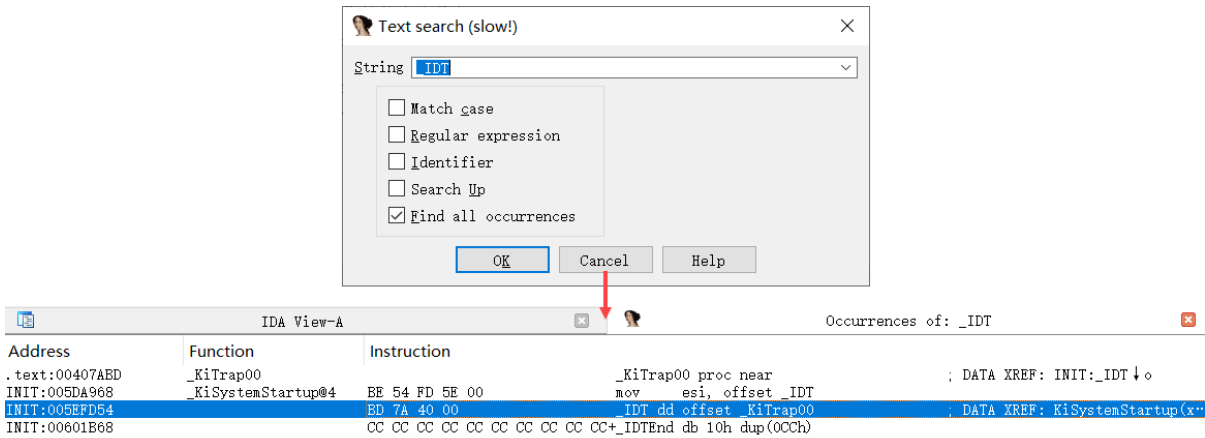
```
1 typedef struct _EXCEPTION_RECORD {
2     DWORD ExceptionCode; // 异常状态码，在Windows中每一种状态（包括异常）都有一个
    状态码
3     DWORD ExceptionFlags; // 异常状态，0表示CPU异常，1表示软件模拟异常，8表示堆栈
    异常
4     struct _EXCEPTION_RECORD *ExceptionRecord; // 通常情况下该值为空，如果发生
    嵌套异常（即处理异常时又出现了异常）则指向下一个异常
5     PVOID ExceptionAddress; // 异常发生地址，表示异常发生时的位置
```

```
6     DWORD NumberParameters; // 附加参数个数
7     ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS]; // 附加参
    数指针
8     } EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Windows状态码及其对应含义，我们可以在在线文档中获取：https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebc55，如下图所示我们可以看见，整数除0时的异常状态码为0xC0000094。

| | |
|-------------------------------|--|
| <u>0xC0000094</u> | <u>{EXCEPTION} Integer division by zero.</u> |
| STATUS_INTEGER_DIVIDE_BY_ZERO | |
| 0xC0000095 | {EXCEPTION} Integer overflow. |
| STATUS_INTEGER_OVERFLOW | |
| 0xC0000096 | {EXCEPTION} Privileged instruction. |
| STATUS_PRIVILEGED_INSTRUCTION | |

接着我们可以通过除0的例子来看一下CPU的异常记录过程，它的大致过程就是：**CPU指令检测到异常**→**查IDT表执行中断处理函数**→**执行CommonDispatchException**→**执行KiDispatchException**。我们可以通过IDA打开Ntoskrnl.exe先找到IDT表，通过ALT+T快捷键全局搜索_IDT，找到IDT表。



根据下图所示的中断描述符表我们可以知道除0异常对应的0号中断处理函数，因此我们就可以在IDA中进入对应的处理函数：

| INT_NUM | Short Description | PM [clarification needed] |
|---------|---------------------------------------|---------------------------|
| 0x00 | Division by zero | |
| 0x01 | Single-step interrupt (see trap flag) | |
| 0x02 | NMI | |

```

INIT:005EFD53 90 align 4
INIT:005EFD54 BD 7A 40 00 _IDT dd offset _KiTrap00 ; DATA XREF: KiSystemStartup(x)+1D51o
INIT:005EFD58 00 8E db 0, 8Eh
INIT:005EFD5A 08 00 word_SEFD5A dw 8 ; DATA XREF: KiSwapIDT()!o
INIT:005EFD5C 3E 7C 40 00 dd offset _KiTrap01

.text:00407ABD 6A 00 push 0
.text:00407ABF 66 C7 44 24 02 00 00 mov [esp+4+var_2], 0
.text:00407AC6 55 push ebp
.text:00407AC7 53 push ebx
.text:00407AC8 56 push esi
.text:00407AC9 57 push edi
.text:00407ACA 0F A0 push fs
.text:00407ACC BB 30 00 00 00 mov ebx, 30h ; '0'
.text:00407AD1 66 8E E3 mov fs, bx
.text:00407AD4 assume fs:nothing
.text:00407AD4 64 8B 1D 00 00 00 00 mov ebx, large fs:0
.text:00407ADB 53 push ebx
.text:00407ADC 83 EC 04 sub esp, 4
.text:00407ADF 50 push eax
.text:00407AE0 51 push ecx
.text:00407AE1 52 push edx
.text:00407AE2 1E push ds
.text:00407AE3 06 push es
.text:00407AE4 0F A8 push gs
.text:00407AE6 66 B8 23 00 mov ax, 23h ; '#'
.text:00407AEA 83 EC 30 sub esp, 30h
.text:00407AED 66 8E D8 mov ds, ax
.text:00407AF0 assume ds:nothing

```

该函数的前面一部分代码和KiSystemService函数（系统调用API进0环）的代码一样，都是用来保存现场的：

```

kd> u KiSystemService L10
nt!KiSystemService:
80541441 6a00 push 0
80541443 55 push ebp
80541444 53 push ebx
80541445 56 push esi
80541446 57 push edi
80541447 0fa0 push fs
80541449 bb30000000 mov ebx,30h
8054144e 668ee3 mov fs,bx
80541451 64fff35000000000 push dword ptr fs:[0]
80541458 64c70500000000ffff mov dword ptr fs:[0],0FFFFFFFFh
80541463 648b3524010000 mov esi,dword ptr fs:[124h]
8054146a ffb640010000 push dword ptr [esi+140h]
80541470 83ec48 sub esp,48h
80541473 8b5c246c mov ebx,dword ptr [esp+6Ch]
80541477 83e301 and ebx,1
8054147a 889e40010000 mov byte ptr [esi+140h],bl

```

接着向下可以看到，该中断处理函数直至执行结束都没有对异常进行处理（微软在设计时，希望程序员自己能够对异常进行处理，因此在中断处理函数中并没有对异常进行处理），反而是有多处的跳转，调用了另一个函数CommonDispatchException。

```

loc_407B49:                                ; CODE XREF: _KiTrap00+AB↓j
                                           ; _KiTrap00+B6↓j
sti
mov     ebx, [ebp+68h]
mov     eax, 0C0000094h                    ; 异常状态码
jmp     loc_407925

; -----
loc_407B57:
mov     ebx, large fs:124h
mov     ebx, [ebx+44h]
cmp     dword ptr [ebx+158h], 0
jz      short loc_407B49

loc_407B6A:
push    0
call    _Ki386VdmReflectException_A@4      ; Ki386VdmReflectException_A(x)

or      al, al
jz      short loc_407B49

jmp     Kei386EoiHelper@0                  ; Kei386EoiHelper()

_KiTrap00 endp

```



```

loc_407925:
xor     ecx, ecx
call    CommonDispatchException

```

跟进CommonDispatchException函数，我们可以看见它开辟了一块大小为0x50的空间，用于存放_EXCEPTION_RECORD结构体，并且给结构体的每个成员赋值，最终执行KiDispatchException函数，该函数通常用来分发异常，目的是找到异常的处理函数。

```

sub     esp, 50h                ; 开辟一块空间，这里大小是0x50
                                   ; 即_EXCEPTION_RECORD结构体的大小
mov     [esp+50h+var_50], eax    ; EAX是异常状态码，给到ExceptionCode
xor     eax, eax                ; 将EAX置0
mov     [esp+50h+var_4C], eax    ; ExceptionFlags设为0
mov     [esp+50h+var_48], eax    ; ExceptionRecord设为0
mov     [esp+50h+var_44], ebx    ; EBX是上层传递过来的[EBP+0x68]
                                   ; 赋值给了结构体的ExceptionAddress
mov     [esp+50h+var_40], ecx    ; 将ECX给到NumberParameters
cmp     ecx, 0
jz      short loc_407974

lea     ebx, [esp+50h+var_3C]
mov     [ebx], edx
mov     [ebx+4], esi
mov     [ebx+8], edi

loc_407974:                      ; CODE XREF: CommonDispatchException+1B↑j
mov     ecx, esp
test    dword ptr [ebp+70h], 20000h
jz      short loc_407986

mov     eax, 0FFFFh
jmp     short loc_407989

; -----
loc_407986:                      ; CODE XREF: CommonDispatchException+32↑j
mov     eax, [ebp+6Ch]

loc_407989:                      ; CODE XREF: CommonDispatchException+39↑j
and     eax, 1
push    1                       ; char
push    eax                     ; int
push    ebp                     ; BugCheckParameter3
push    0                       ; int
push    ecx                     ; ExceptionRecord
call    KiDispatchException@20 ; KiDispatchException(x,x,x,x,x)

mov     esp, ebp
jmp     Kei386EoiHelper@0        ; Kei386EoiHelper()

CommonDispatchException endp

```

其中异常状态码是来自上层的EAX，这点我们通过之前的流程就可以知道，接着我们来看异常发生地址（即ExceptionAddress）是来自上层的EBX，而EBX又来自[EBP+0x68]，这里实际上指的是Trap_Frame结构体0x68偏移位Eip成员，它是用来记录中断发生地址的，这是因为在保存现场结束之后，**ESP指向Trap_Frame的顶部，而EBP也与ESP一样。**

```

push    0
mov     [esp+4+var_2], 0
push    ebp
push    ebx
push    esi
push    edi
push    fs
mov     ebx, 30h ; '0'
mov     fs, bx
assume fs:nothing
mov     ebx, large fs:0
push    ebx
sub     esp, 4
push    eax
push    ecx
push    edx
push    ds
push    es
push    gs
mov     ax, 23h ; '#'
sub     esp, 30h
mov     ds, ax
assume ds:nothing
mov     es, ax
assume es:nothing
mov     ebp, esp
test    [esp+68h+arg_4], 20000h
jnz     short V86_kit0_a

```

1.1.2 软件模拟的异常记录

我们接着来看一下throw关键词触发的软件模拟异常记录过程，在关键词处下断点，然后运行通过反汇编代码我们可以知道它调用的就是CxxThrowException函数：

```

6:      throw 1;
0040BCE8  mov     dword ptr [ebp-4],1
0040BCEF  push    offset __TI1H (00421580)
0040BCF4  lea     eax,[ebp-4]
0040BCF7  push    eax
0040BCF8  call    __CxxThrowException@8 (00401290)
7:
8:      getchar();
9:      return 0;
10:   }

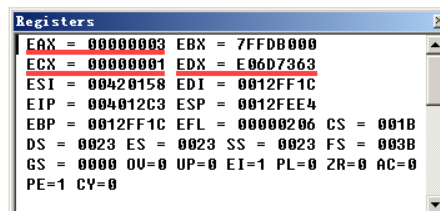
```

跟进该函数，会发现它也是调用了另外一个函数RaiseException，并通过栈的方式压入了几个传参：

```

_CxxThrowException@8:
00401290 push     ebp
00401291 mov      ebp,esp
00401293 sub      esp,20h
00401296 push     esi
00401297 push     edi
00401298 mov      ecx,8
0040129D mov      esi,offset string "The value of ESP was not proper!"+0E0h (00420138)
004012A2 lea      edi,[ebp-20h]
004012A5 rep movs dword ptr [edi],dword ptr [esi]
004012A7 mov      eax,dword ptr [ebp+8]
004012AA mov      dword ptr [ebp-8],eax
004012AD mov      ecx,dword ptr [ebp+0Ch]
004012B0 mov      dword ptr [ebp-4],ecx
004012B3 lea      edx,[ebp-0Ch]
004012B6 push     edx
004012B7 mov      eax,dword ptr [ebp-10h]
004012BA push     eax
004012BB mov      ecx,dword ptr [ebp-1Ch]
004012BE push     ecx
004012BF mov      edx,dword ptr [ebp-20h]
004012C2 push     edx
004012C3 call     dword ptr [ imp_RaiseException@16 (00426148) ]
004012C9 pop      edi
004012CA pop      esi
004012CB mov      esp,ebp
004012CD pop      ebp
004012CE ret     8
004012D1 int     3

```



接着我们跟进RaiseException函数发现，正是这几个传参填充了_EXCEPTION_RECORD结构体，在这里我们会发现两个比较关键成员的值都与CPU异常记录时的赋值内容不一致，首先是ExceptionCode，它的值很明显是一个Windows异常状态码中没有的（即0xE06D7363），这是因为在软件模拟产生的异常场景下，**ExceptionCode的值是根据不同的编译环境而生成的**；其次是ExceptionAddress，如下图中我们可以看见，它的值是**RaiseException函数的首地址**，而并不是真正产生异常的那段地址。

```

➡ 7C81EAE1  mov     edi,edi
   7C81EAE3  push    ebp
   7C81EAE4  mov     ebp,esp
   7C81EAE6  sub     esp,50h
   7C81EAE9  mov     eax,dword ptr [ebp+8]
   7C81EAEc  and     dword ptr [ebp-48h],0    // ExceptionRecord
   7C81EAF0  mov     dword ptr [ebp-50h],eax  // ExceptionCode
   7C81EAF3  mov     eax,dword ptr [ebp+0Ch]
   7C81EAF6  push    esi
   7C81EAF7  mov     esi,dword ptr [ebp+14h]
   7C81EAFa  and     eax,1
   7C81EAFD  test    esi,esi
   7C81EAFF  mov     dword ptr [ebp-4Ch],eax  // ExceptionFlags
   7C81EB02  mov     dword ptr [ebp-44h],7C81EAE1h // ExceptionAddress
   7C81EB09  je      7C81EBA9
   7C81EB0F  mov     ecx,dword ptr [ebp+10h]
   7C81EB12  cmp     ecx,0Fh
   7C81EB15  ja      7C844584
   7C81EB1B  test    ecx,ecx
   7C81EB1D  mov     dword ptr [ebp-40h],ecx  // NumberParameters
   7C81EB20  je      7C81EB29
   7C81EB22  push    edi
   7C81EB23  lea     edi,[ebp-3Ch]
   7C81EB26  rep movs dword ptr [edi],dword ptr [esi]
   7C81EB28  pop     edi
   7C81EB29  lea     eax,[ebp-50h]
   7C81EB2C  push    eax
   7C81EB2D  call    dword ptr ds:[7C801504h]
   7C81EB33  pop     esi
   7C81EB34  leave   10h
   7C81EB35  ret

```

RaiseException函数之后的流程是这样的：RtlRaiseException→NtRaiseException→KiRaiseException，在最后执行到KiRaiseException函数时，会将ExceptionCode的最高位清0，便于区分CPU/软件模拟异常：

```

push    [ebp+BugCheckParameter3]
call    KeContextToKframes@20 ; KeContextToKframes(x,x,x,x,x)
and     byte ptr [ebx+3], 0EFh
push    dword ptr [ebp+arg_10] ; char
push    [ebp+var_300] ; int
push    [ebp+BugCheckParameter3] ; BugCheckParameter3
push    [ebp+var_2F0] ; int
push    ebx ; ExceptionRecord
call    _KiDispatchException@20 ; KiDispatchException(x,x,x,x,x)
xor     eax,eax

```

虽然模拟异常与CPU异常有一定的差异，但是在最后，两者都会去调用KiDispatchException函数，用于异常分发。

CPU异常

模拟异常



1.2 异常分发与处理

异常可以发生在用户空间，也可以发生在内核空间。无论是CPU异常还是模拟异常，无论是用户空间异常还是内核空间异常，**最终都要通过KiDispatchException函数进行分发**，理解这个函数是学好异常的关键，这个函数比较复杂，我们以内核、用户两个角度来分析学习。

1.2.1 内核异常

本章我们主要分析内核异常是如何分发，如何处理的。首先我们需要来了解一下KiDispatchException函数的格式，及每个参数的作用：

```
1  VOID KiDispatchException (
2      PEXCEPTION_RECORD ExceptionRecord, // 异常记录结构体
3      PKEXCEPTION_FRAME ExceptionFrame, // X86系统下, 该值为NULL
4      PKTRAP_FRAME TrapFrame, // 3环进0环保存现场所用的结构体
5      KPROCESSOR_MODE PreviousMode, // 先前模式, 表示调用来自什么模式, 0表示
    内核模式, 1表示用户模式
6      BOOLEAN FirstChance // 判断是否是第一次分发这个异常, 对于同一
    个异常, Windows最多分发两次
7      // 该值为1表示第一次分发, 为0表示第二次分
    发
8  )
```

通过IDA直接打开Ntoskrnl.exe模块，找到KiDispatchException函数，该函数最开始就是先调用_KeContextFromKframes函数，将Trap_Frame备份到_Context中，为返回3环做准备（这里与用户APC执行过程一样，**因为该函数支持内核、用户空间的异常分发和处理**，因此我们不知道异常处理函数到底是在用户空间还是内核空间，所以第一件事情就是备份Trap_Frame，**便于中途回到3环**）。

```
loc_426378:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+55↑j
                                           ; KiDispatchException(x,x,x,x,x)+68↑j
lea     eax, [ebp+Context]
push    eax
push    ecx
push    ebx
call    KeContextFromKframes@12           ; KeContextFromKframes(x,x,x)
```

接着我们可以看见它会去判断先前模式，如果先前模式为1则进入用户空间的异常处理逻辑，为0则接着向下判断FirstChance，即当前异常是否为第一次分发，如果是第一次则继续向下判断当前是否有内核调试器，如果没有内核调试器则跳转，有内核调试器的话优先调用内核调试器函数。

```
loc_426398:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+2541A↓j
cmp     byte ptr [ebp+PreviousMode], 0
jnz     loc_42B6C3

cmp     [ebp+FirstChance], 1
jnz     loc_44B72B

mov     eax, ds:_KiDebugRoutine
cmp     eax, edi
jz      loc_438EB6

push    edi
push    edi
lea     ecx, [ebp+Context]
push    ecx
push    esi
push    [ebp+var_2F0]
push    ebx
call    eax ; _KiDebugRoutine
```

接着我们向下看，会发现没有内核调试器，或者内核调试器函数返回结果为0的情况下，都会跳转至同一代码段；这里需要说明下内核调试器函数返回结果为0则表示异常未被处理，为1则表示异常被处理了，然后会将_Context转换成Trap_Frame返还，异常处理过程结束，退出KiDispatchException函数。

```

mov     eax, ds:_KiDebugRoutine
cmp     eax, edi
jz      loc_438EB6

push    edi
push    edi
lea     ecx, [ebp+Context]
push    ecx
push    esi
push    [ebp+var_2F0]
push    ebx
call    eax ; _KiDebugRoutine

test    al, al
jz      loc_438EB6

push    [ebp+PreviousMode]
push    [ebp+Context.ContextFlags]
lea     eax, [ebp+Context]
push    eax
push    [ebp+var_2F0]
push    ebx
call    _KeContextToKframes@20 ; KeContextToKframes(x,x,x,x,x)

```

然后我们跟进它们都跳转进的代码段会发现这里调用了RtlDispatchException函数，这个函数专门负责调用异常处理函数来处理异常，我们可以看见该函数调用时候传递了两个参数，即Context和ExceptionRecord。

```

; __unwind { // __SEH_prolog
lea     eax, [ebp+Context]
push    eax ; Context
push    esi ; ExceptionRecord
call    _RtlDispatchException@8 ; RtlDispatchException(x,x)

```

跟进RtlDispatchException函数（该函数是一个库函数，内核和用户异常都会使用它来进行分发，在本章简单了解一下，后续用户异常分发和处理我们详细进行分析），我们会发现其调用了RtlpGetRegistrationHead函数，它的作用就是获取FS:[0]。

```

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 64h
push    ebx
lea     eax, [ebp+HighLimit]
push    eax ; HighLimit
lea     eax, [ebp+LowLimit]
push    eax ; LowLimit
mov     [ebp+var_1], 0
call    _RtlpGetStackLimits@8 ; RtlpGetStackLimits(x,x)

call    _RtlpGetRegistrationHead@0 ; RtlpGetRegistrationHead()
; _DWORD __stdcall RtlpGetRegistrationHead()
; CODE XREF: RtlUnwind(x,x,x,x)+904p
; RtlDispatchException(x,x)+1A4p
mov     eax, large fs:0
ret

```

```

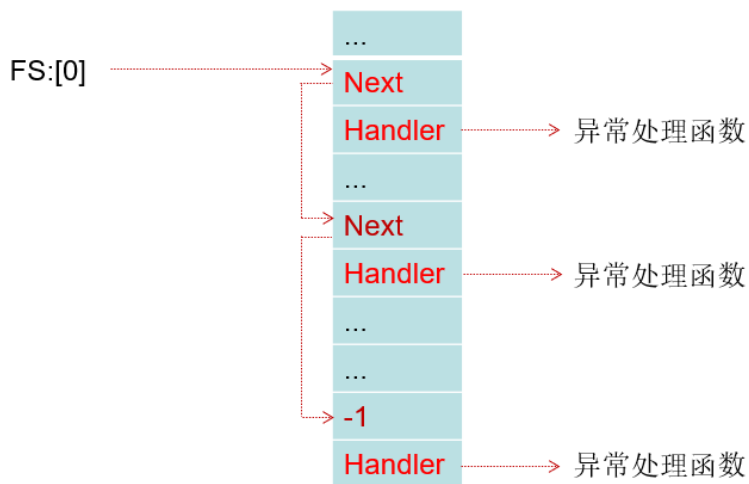
kd> dt _KPCR
nt!_KPCR
+0x000 NtTib : _NT_TIB
+0x01c SelfPcr : Ptr32 _KPCR
+0x020 Prcb : Ptr32 _KPRCB
...
kd> dt _NT_TIB
nt!_NT_TIB
+0x000 ExceptionList : Ptr32 EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase : Ptr32 Void

```

根据之前的学习，我们知道在0环的FS:[0]指向的是_KPCR结构体，_KPCR的第一个成员是NtTib¹，而NtTib的第一个字段是ExceptionList，ExceptionList这个字段是一个指针，它指向了一个结构体_EXCEPTION_REGISTRATION_RECORD，该结构体有2个成员，第一个成员Next指向下一个_EXCEPTION_REGISTRATION_RECORD结构体地址（如果没有下一个结构体，则该值为-1），第二个成员Handler指向了异常处理函数。

```
1 kd> dt _EXCEPTION_REGISTRATION_RECORD
2 nt!_EXCEPTION_REGISTRATION_RECORD
3     +0x000 Next           : Ptr32 _EXCEPTION_REGISTRATION_RECORD
4     +0x004 Handler        : Ptr32 _EXCEPTION_DISPOSITION
```

所以RtlDispatchException的作用就是遍历异常链表，调用异常处理函数，如果异常被正确处理了，该函数返回1；如果当前异常处理函数不能处理该异常，就调用下一个，以此类推到最后也没有异常处理函数处理这个异常，则该函数返回0。



如果RtlDispatchException函数返回0，我们可以看见它又会去判断当前是否有内核调试器，和上面的流程一样，发现没有内核调试器，或者内核调试器函数返回结果为0的情况下，跳转至同一代码段，该代码段的作用就是蓝屏（KeBugCheckEx就是Windows执行崩溃的函数，作用就是使计算机蓝屏）。

¹ <https://cataloc.gitee.io/blog/2020/03/30/KPCR/#0x000-NtTib>

```

; __unwind { // __SEH_prolog
lea    eax, [ebp+Context]
push   eax                                ; Context
push   esi                                ; ExceptionRecord
call   _RtlDispatchException@8           ; RtlDispatchException(x,x)

jmp     loc_44B723

loc_44B723:                               ; CODE XREF: KiDispatchException(x,x,x,x,x)+12BBF↑j
cmp     al, 1
jz      loc_4263D7

loc_44B72B:                               ; CODE XREF: KiDispatchException(x,x,x,x,x)+A5↑j
mov     eax, ds:_KiDebugRoutine
cmp     eax, edi
jz      loc_44B828

push    1
push    edi
lea     ecx, [ebp+Context]
push    ecx
push    esi
push    [ebp+var_2F0]
push    ebx
call    eax ; _KiDebugRoutine

test    al, al
jnz     loc_4263D7

jmp     loc_44B828

push    edi                                ; BugCheckParameter4
push    ebx                                ; BugCheckParameter3
push    dword ptr [esi+0Ch]                ; BugCheckParameter2
push    dword ptr [esi]                    ; BugCheckParameter1
push    8Eh                                ; BugCheckCode
call    _KeBugCheckEx@20                   ; KeBugCheckEx(x,x,x,x,x)

```

↓

蓝屏

1.2.2 用户异常

异常如果发生在内核层，处理起来比较简单，因为异常处理函数也在0环，不用切换堆栈，但是如果异常发生在3环，就意味着必须要切换堆栈，回到3环执行异常处理函数。这个堆栈切换的处理方式与用户APC的执行过程几乎是一样的，惟一的区别就是执行用户APC时返回3环后执行的函数是KiUserApcDispatcher，而异常处理时返回3环后执行的函数是KiUserExceptionDispatcher。因此本章节不再对堆栈的切换进行了解，可以自行复习一下用户APC执行过程的笔记。

我们接着来看一下KiDispatchException函数是如何对用户空间的异常进行分发处理的，还是开头的那几步：保存Trap_Frame至_Context，判断先前模式如果为1则表示来当前异常来自用户空间，跳转到对应代码段，接着判断是否是第一次分发，判断内核调试器，调用内核调试器...

```

loc_426378:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+55↑j
                                           ; KiDispatchException(x,x,x,x,x)+68↑j
lea     eax, [ebp+Context]
push    eax
push    ecx
push    ebx
call    _KeContextFromKframes@12          ; KeContextFromKframes(x,x,x)
...
loc_42639B:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+2541A↓j
cmp     byte ptr [ebp+PreviousMode], 0
jnz     loc_42B6C3
↓
loc_42B6C3:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+9B↑j
; _unwind { // _SEH_prolog
cmp     [ebp+FirstChance], 1
jnz     loc_44B7FC

cmp     ds:_KiDebugRoutine, edi
jz      short loc_42B70B

mov     eax, large fs:124h
mov     eax, [eax+44h]
cmp     [eax+0BCh], edi
jnz     loc_44B759

loc_42B6EA:                                ; CODE XREF: KiDispatchException(x,x,x,x,x)+12BCC↓j
push    edi
push    [ebp+PreviousMode]
lea     eax, [ebp+Context]
push    eax
push    esi
push    [ebp+var_2F0]
push    ebx
call    ds:_KiDebugRoutine

test    al, al
jnz     loc_4263D7

```

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，都会跳转或向下执行到同一处代码段，这里调用了函数 `_DbgkForwardException`，它的作用是将异常发送给3环的调试器，如果返回非0则表示有3环调试器且处理了异常，就会跳转。

```

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(x,x,x)

test    al, al
jnz     loc_4263F3

```

反之，如果返回为0则表示没有处理该异常，继续向下执行就是为返回3环做准备，将 `Trap_Frame` 的值修改为返回3环时候的环境，详细的不再多说，其中最为重要的就是返回3环时的地址，即 `Trap_Frame.Eip (0x68偏移位)`，如下图代码所示，将 `_Eip` 修改为了 `KeUserExceptionDispatcher` 函数的地址，也就表示返回3环后就执行该函数。但是这里修改完 `Trap_Frame` 的值之后并没有返回3环，而是直接跳转，`KiDispatchException` 函数执行结束。

```

mov     [ebx+50h], eax
and     dword ptr [ebx+30h], 0
mov     eax, ds: _KeUserExceptionDispatcher
mov     [ebx+68h], eax
or      [ebp+ms_exc.registration.TryLevel], 0FFFFFFFFh
jmp     loc_4263F3

```

由于不同类型的异常，调用KiDispatcherException的函数不同，所以会当KiDispatcherException执行完后，会返回当相应的函数继续执行。如下图所示，如果是CPU异常，KiDispatcherException函数执行完成之后会返回到CommonDispatcherException函数中，并通过IRETD返回3环（CPU是通过中断门进的0环，因此用中断返回）；如果是模拟异常，KiDispatcherException执行完成之后会返回到KiRaiseException函数中，并通过系统调用（KiServiceExit）返回3环。

CPU异常：CPU检测到异常 → 查IDT执行处理函数 → CommonDispatchException

→ KiDispatchException 通过IRETD返回3环

模拟异常：CxxThrowException → RaiseException → RtlRaiseException

NT!NtRaiseException → NT!KiRaiseException

→ KiDispatchException 通过系统调用返回3环

虽然返回3环的方式不同，但只要是用户空间的异常，当线程再次回到3环时，执行的都是KiUserExceptionDispatcher。（KiUserExceptionDispatcher函数会在后续章节中了解）

1.3 VEH

KiUserExceptionDispatcher函数存在与Ntdll.dll模块中，我们可以通过IDA找到并分析，该函数的作用很简单，首先调用RtlDispatchException函数（该函数主要用于查找并执行异常处理函数）：如果RtlDispatchException返回真则表示异常处理成功，那么原代码就需要从新的地方或原来的地方开始执行，因此需要调用ZwContinue函数再次进入0环，将修正后的_Context结构体给到Trap_Frame，这样就可以在线程再次返回3环时，从修正后的位置开始执行；如果RtlDispatchException返回假则表示异常没有被处理或没有异常处理函数，则调用ZwRaiseException函数进行二次异常分发。

```

mov     ecx, [esp+arg_0]
mov     ebx, [esp+0]
push    ecx
push    ebx
call    _RtlDispatchException@8          ; RtlDispatchException(x,x)

or      al, al
jz      short loc_7C92EB0A

pop     ebx
pop     ecx
push    0
push    ecx
call    _ZwContinue@8                  ; ZwContinue(x,x)

loc_7C92EB0A:                          ; CODE XREF: KiUserExceptionDispatcher(x,x)+10↑j
pop     ebx
pop     ecx
push    0
push    ecx
push    ebx
call    _ZwRaiseException@12           ; ZwRaiseException(x,x,x)

```

RtlDispatchException是个库函数，在内核异常分发处理时用的也是这个函数，虽然在用户异常也是如此，但实际过程是不一样的。

我们在之前内核异常分发学习中知道RtlDispatchException函数调用了RtlpGetRegistrationHead函数来查找一个异常链表，这个异常链表也可以称之为SEH链表，而在用户异常分发的RtlDispatchException函数中首先调用了RtlpExecuteHandlerForException函数来寻找VEH链表，如果有的话则遍历该链表找到对应的异常处理函数，如果没有则继续使用RtlpGetRegistrationHead函数来寻找SEH链表。

| 用户空间 | | 内核空间 | |
|-------------------|--|-------------------|---|
| 8B FF | mov edi, edi | 8B FF | mov edi, edi |
| 55 | push ebp | 55 | push ebp |
| 8B EC | mov ebp, esp | 8B EC | mov ebp, esp |
| 83 EC 64 | sub esp, 64h | 83 EC 64 | sub esp, 64h |
| 56 | push esi | 53 | push ebx |
| FF 75 0C | push [ebp+arg_4] | 8D 45 F4 | lea eax, [ebp+HighLimit] |
| 8B 75 08 | mov esi, [ebp+arg_0] | 50 | push eax |
| 56 | push esi | 8D 45 F0 | lea eax, [ebp+LowLimit] |
| C6 45 FF 00 | mov [ebp+var_1], 0 | 50 | push eax |
| E8 C2 FF FF FF | call _RtlCallVectoredExceptionHandlers@8 ; RtlCallVectoredExceptionHandlers(x,x) | C6 45 FF 00 | mov [ebp+var_1], 0 |
| 84 C0 | test al, al | E8 88 2E FE FF | call _RtlpGetStackLimits@8 ; RtlpGetStackLimits(x,x) |
| 0F 85 84 72 01 00 | jnz loc_7C96EA66 | E8 A4 2E FE FF | call _RtlpGetRegistrationHead@0 ; RtlpGetRegistrationHead() |
| 53 | push ebx | 83 65 F8 00 | and [ebp+var_8], 0 |
| 8D 45 F4 | lea eax, [ebp+var_C] | 8B D8 | mov ebx, eax |
| 50 | push eax | 83 FB FF | cmp ebx, 0FFFFFFFFh |
| 8D 45 F8 | lea eax, [ebp+var_8] | 0F 84 7A 2B 03 00 | jz loc_455450 |
| 50 | push eax | 56 | push esi |
| E8 1C C1 FC FF | call _RtlpGetStackLimits@8 ; RtlpGetStackLimits(x,x) | 8B 75 08 | mov esi, [ebp+ExceptionRecord] |
| E8 38 C1 FC FF | call _RtlpGetRegistrationHead@0 ; RtlpGetRegistrationHead() | 57 | push edi |
| 83 65 08 00 | and [ebp+arg_0], 0 | loc_422808: | |
| 8B D8 | mov ebx, eax | 3B 5D F0 | cmp ebx, [ebp+LowLimit] |
| 83 FB FF | cmp ebx, 0FFFFFFFFh | 8D 7B 08 | lea edi, [ebx+8] |
| 0F 84 8F 00 00 00 | jz loc_7C957893 | 72 76 | short loc_422959 |
| 57 | push edi | 3B 7D F4 | cmp edi, [ebp+HighLimit] |
| | | 77 71 | ja short loc_422959 |

SEH链表是一种存在堆栈中的局部链表，本章要学习的VEH链表结构与之相似，只不过VEH链表是全局链表。我们要想通过VEH来处理异常，要先创建如下回调函数来接收和处理异常：

```

1 LONG NTAPI VectoredHandler(
2     PEXCEPTION_POINTERS ExceptionInfo
3 );

```

该函数的参数ExceptionInfo为EXCEPTION_POINTERS结构体的指针，该结构体及其成员如下所示：


```

1  typedef struct _EXCEPTION_POINTERS {
2      PEXCEPTION_RECORD ExceptionRecord;    // 异常记录
3      PCONTEXT ContextRecord;    // 异常发生时的线程上下文环境
4  };

```

回调函数VectoredHandler有两个返回值：

```

1  #define EXCEPTION_CONTINUE_SEARCH    0    // 异常未被处理，继续搜索
2  #define EXCEPTION_CONTINUE_EXECUTION -1    // 异常处理完毕，恢复执行

```

回调函数VectoredHandler创建之后，就可以将其进行注册，也就是添加到全局链表中，当遇到对应的用户异常时，就会被查找并调用。注册回调函数的格式如下：

```

1  PVOID AddVectoredExceptionHandler(
2      ULONG FirstHandler, // 指定注册的回调函数被调用的顺序，该值为0表示希望最后被调用，为1表示希望最先被调用，若注册了多个回调函数，且所有的FirstHandler值都为1，那么最后注册的回调函数会被最先调用
3      PVECTORED_EXCEPTION_HANDLER VectorerHandler // 需要注册的回调函数
4  );

```

在AddVectoredExceptionHandler函数内部，会为每个VEH准备如下一个结构体：

```

1  typedef struct _VEH_REGISTRATION{
2      _VEH_REGISTRATION* next; // 指向下一个VEH
3      _VEH_REGISTRATION* prev; // 指向上一个VEH
4      PVECTORED_EXCEPTION_HANDLER pfnVeh; // 指向当前VEH的回调函数
5  }VEH_REGISTRATION, *PVEH_REGISTRATION;

```

当有多个VEH时，**这些VEH的_VEH_REGISTRATION结构体串联组成一个双向链表**。在Ntdll.dll模块中，全局变量RtlpCalloutEntryList指向该链表的链表头（在0环中，则是通过FS:[0]找到ExceptionList再找到SEH的链表头）。

当VEH处理异常结束之后，我们可以注销VEH，即如下这个函数：

```

1  PVOID RemoveVectoredExceptionHandler(
2      PVECTORED_EXCEPTION_HANDLER VectorerHandler // 注册的回调函数
3  );

```

在有了以上基础的铺垫之后，我们可以编写如下代码来使用VEH处理异常，这段代码大致分为3部分：

1. 定义了指向AddVectoredExceptionHandler函数的函数指针，**因为VEH异常处理在XP之前的系统中并没有，也就不存在这个函数**，因此为了代码兼容性，我们不直接通过库调用，而是以动态加载DLL的方式来获取函数地址，然后用定义的函数指针指向它，再调用函数指针就可以使用相应功能了；
2. 定义异常处理函数VectExcepHandler，根据其参数ExceptionInfo.ExceptionRecord.ExceptionCode来获取异常状态码，如果是除0异常则通过两种方式来处理异常，**一是通过ExceptionInfo.ContextRecord.Eip来修改返回3环时的地址**，以此来跳过异常处理的代码（EIP+2是因为idiv ecx这个指令长度就是2），**二是通过ExceptionInfo.ContextRecord.Ecx来修改除数**，修复异常的代码；
3. 注册异常处理函数，并通过汇编的方式构造除0异常，这里也就将EAX作为了除数，这样就可以触发异常，从而进入异常的处理。

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  // 定义指向AddVectoredExceptionHandler函数的函数指针
5  typedef PVOID (NTAPI *FnAddVectoredExceptionHandler) (ULONG,
6  _EXCEPTION_POINTERS*);
7  FnAddVectoredExceptionHandler MyAddVectoredExceptionHandler;
8
9  // 定义异常处理函数
10 LONG NTAPI VectExcepHandler(PEXCEPTION_POINTERS pExcepInfo)
11 {
12     MessageBox(NULL, "VEH异常处理函数执行了...", "VEH异常", MB_OK);
13
14     // 除0异常
15     if (pExcepInfo->ExceptionRecord->ExceptionCode == 0xC0000094)
16     {
17         // 修改发生异常时的EIP
18         // pExcepInfo->ContextRecord->Eip = pExcepInfo->ContextRecord->Eip
19         + 2;
20
21         // 修改发生异常时的ECX
22         pExcepInfo->ContextRecord->Ecx = 1;
23
24         // 此处返回表示异常已处理
25         return EXCEPTION_CONTINUE_EXECUTION;
26     }
27     // 此处返回表示异常未处理
28     return EXCEPTION_CONTINUE_SEARCH;
29 }
30
31 // 主函数
32 int main()
33 {
34     // 动态获取AddVectoredExceptionHandler函数地址，并将异常处理函数挂入VEH链表
35     HMODULE hModule = GetModuleHandle("Kernel32.dll");
36     MyAddVectoredExceptionHandler =
37     (FnAddVectoredExceptionHandler)::GetProcAddress(hModule,
38     "AddVectoredExceptionHandler");
39
40     // 注册
41     MyAddVectoredExceptionHandler(0, (_EXCEPTION_POINTERS
42     *)&VectExcepHandler);

```

```

38
39     // 构造除0异常
40     _asm
41     {
42         xor edx, edx
43         xor ecx, ecx
44         mov eax, 100
45         idiv ecx
46     }
47
48     printf("Running ... ");
49
50     getchar();
51     return 0;
52 }

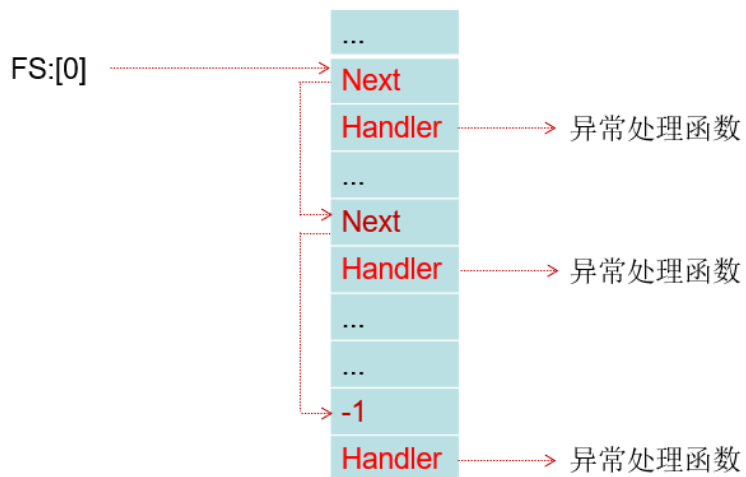
```

1.4 SEH

1.4.1 原生SEH

如果调用RtlDispatchException函数找不到VEH链表，则会去寻找SEH链表，SEH链表是一个局部链表，存储在当前线程的栈中，**因此不同的线程要通过SEH来处理异常都需要在自己的堆栈中存放。**

SEH链表的格式如下所示，它是一个单向链表，每个成员中包含了两个成员，即Next（下一个SEH）和Handler（异常处理函数）。无论是在内核还是用户空间，我们都可以**通过FS:[0]来找到该链表头。**



```

kd> dt _TEB
nt!_TEB
+0x000 NtTib          : NT TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId       : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
...
kd> dt _NT_TIB
nt!_NT_TIB
+0x000 ExceptionList  : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase      : Ptr32 Void
+0x008 StackLimit     : Ptr32 Void

```

SEH与VEH不同的是，前者没法使用函数的方式去注册异常处理函数，需要我们手动的向栈中填充，然后将FS:[0]指向这块栈地址。这里我们可以自己定义一个SEH结构体，但一定需要包含2个成员，即Next、Handler，如下所示：

```

1  struct _MY_EXCEPTION
2  {
3      struct _MY_EXCEPTION *Next;
4      DWORD Handler;
5  };

```

我们接着来分析一下RtlDispatchException函数，以验证上述的一些观点。如下图所示在该函数内找不到VEH链表后，会去调用两个函数。

第一个函数RtlpGetStackLimits取了FS:[8]和FS:[4]的值，即_TEB._NT_TIB.StackBase和_TEB._NT_TIB.StackLimit，这两个值就是栈的基址和栈的大小，取这两个值的目的是为了检查SEH链表是否属于当前线程的栈中。

第二个函数RtlpGetRegistrationHead取FS:[0]的值，即_TEB._NT_TIB.ExceptionList，也就是SEH链表头地址。

```

push    ebx
lea     eax, [ebp+var_C]
push    eax
lea     eax, [ebp+var_8]
push    eax
call    _RtlpGetStackLimits@8      ; RtlpGetStackLimits(x,x)
call    _RtlpGetRegistrationHead@0 ; RtlpGetRegistrationHead()

; _DWORD __stdcall RtlpGetRegistrationHead()
; CODE XREF: RtlDispatchException(x,x)+2F4p
; RtlUnwind(x,x,x,x)+904p
mov     eax, large fs:0
retn

; __stdcall RtlpGetStackLimits(x, x)
; CODE XREF: RtlDispatchException(x,x)+2A4p
; RtlUnwind(x,x,x,x)+264p
arg_0= dword ptr 4
arg_4= dword ptr 8

mov     eax, large fs:8
mov     ecx, [esp+arg_0]
mov     [ecx], eax
mov     eax, large fs:4
mov     ecx, [esp+arg_4]
mov     [ecx], eax
retn    8

_RtlpGetStackLimits@8 endp

```

RtlDispatchException函数再往下走，我们看见它调用了RtlIsValidHandler函数，用于验证异常处理函数是否有效，接着调用RtlpExecuteHandlerForException函数，用于调用异常处理函数，因此我们的异常处理函数并不能完全自定义，而是需要遵循其所能执行的格式。

```

loc_7C957834:                                ; CODE XREF: RtlDispatchException(x,x)+68↑j
push     eax
call     _RtlIsValidHandler@4                ; RtlIsValidHandler(x)

test     al, al
jz       loc_7C94AA2B

test     byte_7C99C35A, 80h
jnz     loc_7C96EA6F

loc_7C95784F:                                ; CODE XREF: RtlDispatchException(x,x)+172BF↓j
push     dword ptr [ebx+4]
lea      eax, [ebp+var_14]
push     eax
push     [ebp+arg_4]
push     ebx
push     esi
call     _RtlpExecuteHandlerForException@20 ; RtlpExecuteHandlerForException(x,x,x,x,x)

```

SEH异常处理函数的格式如下，它一共有4个参数，我们只需要了解前三个参数的含义即可：

```

1  EXCEPTION_DISPOSITION _cdecl MyExceptionHandler
2  (
3      struct _EXCEPTION_RECORD *ExceptionRecord, // 异常记录结构体
4      PVOID EstabliherFrame, // SEH结构体地址
5      struct _CONTEXT *ContextRecord, // 存储异常发生时的上下文环境
6      PVOID DispatcherContext
7  )

```

那么有了这些基础的铺垫之后我们可以通过编写代码来实现SEH异常处理，这段代码与VEH异常处理差不多，唯一的区别在于在我们将FS:[0]指向SEH结构体之前**需要先保存原FS:[0]的值**，然后在结构体的Next赋值时将**原FS:[0]的值写入**，因为可能在原SEH链表中是有内容的，最后在SEH异常处理函数结束，**将FS:[0]的值还原**。

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  struct _MY_EXCEPTION
5  {
6      struct _MY_EXCEPTION *Next;
7      DWORD Handler;
8  };
9
10 EXCEPTION_DISPOSITION _cdecl MyExceptionHandler
11 (
12     struct _EXCEPTION_RECORD *ExceptionRecord, // 异常记录结构体
13     PVOID EstabliherFrame, // SEH结构体地址
14     struct _CONTEXT *ContextRecord, // 存储异常发生时的上下文环境
15     PVOID DispatcherContext
16 )
17 {

```

```
18     MessageBox(NULL, "SEH异常处理函数执行了...", "SEH", MB_OK);
19
20     if (ExceptionRecord->ExceptionCode == 0xC0000094)
21     {
22         // ContextRecord->Eip = ContextRecord->Eip + 2;
23
24         ContextRecord->EcX = 1;
25
26         return ExceptionContinueExecution;
27     }
28     return ExceptionContinueSearch;
29 }
30
31
32 int main()
33 {
34     DWORD tmpData;
35     // 必须在当前线程栈中
36     _MY_EXCEPTION Exception;
37
38     // FS:[0] -> Exception
39     _asm
40     {
41         mov eax, fs:[0]
42         mov tmpData, eax // 保存原FS:[0]
43         lea ecx, Exception
44         mov fs:[0], ecx
45     }
46
47     // 成员赋值
48     Exception.Next = ( _MY_EXCEPTION *)tmpData;
49     Exception.Handler = (DWORD)&MyExceptionHandler;
50
51     // 构造除0异常
52     _asm
53     {
54         xor edx, edx
55         xor ecx, ecx
56         mov eax, 1
57         idiv ecx
58     }
59
60     // 摘除刚插入的SEH, 还原FS:[0]
61     _asm
62     {
63         mov eax, tmpData
64         mov fs:[0], eax
65     }
66
67     printf("Running ... ");
68
69     getchar();
70     return 0;
```

```
71 }

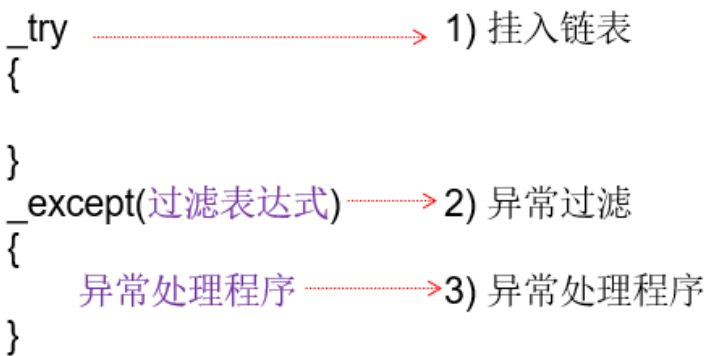
```

1.4.2 编译器扩展SEH

`_try_except`

我们在上一章学习的是Windows自带的原生SEH，除此之外实际上编译器还另外扩展了SEH，这是因为使用原生SEH实际上非常麻烦，需要自己构建结构体、异常处理函数、写入FS:[0]等等操作。

在VC6编译器下，我们可以通过如下格式的代码来处理SEH异常，即 `_try{...}except(...){...}` 格式，使用这种格式编写代码后，编译器会在编译时帮我们转换，使得最终代码能够实现挂入链表、异常过滤、异常处理。



在过滤表达式部分，我们可以直接写常量值，也可以写表达式或调用的函数，但无论什么方式，这里最终的值只能为0、1、-1，它们的含义如下：

```

1  #define EXCEPTION_EXECUTE_HANDLER 1 // 执行except内的代码
2  #define EXCEPTION_CONTINUE_SEARCH 0 // 寻找下一个异常处理函数
3  #define EXCEPTION_CONTINUE_EXECUTION -1 // 返回至出错位置重新执行

```

三种不同风格的表达式示例代码如下，我们可以看见当简单的异常过滤可以直接写常量，复杂一些的使用三元表达式，再复杂一些的可以使用函数的方式，并且我们可以使用 `GetExceptionCode`、`GetExceptionInformation` 函数来获取异常状态码、异常相关信息：

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  int getFilterCode(PEXCEPTION_POINTERS pExcepInfo)
5  {
6      pExcepInfo->ContextRecord->Ecx = 1;
7      return EXCEPTION_CONTINUE_EXECUTION;
8  }
9
10 int main()
11 {
12     _try
13     {
14         _asm

```

```

15      {
16          xor edx, edx
17          xor ecx, ecx
18          mov eax, 1
19          idiv ecx
20      }
21  }
22  // _except(EXCEPTION_EXECUTE_HANDLER)
23  // _except(GetExceptionCode() == 0xC0000094 ?
EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
24  _except(getFilterCode(GetExceptionInformation()))
25  {
26      printf("Processing ...");
27  }
28
29  printf("Running ...");
30
31  getchar();
32  return 0;
33  }

```

我们可以通过反汇编来看一下使用 `_try_except` 的背后到底干了什么，如下图所示，我们可以看见与我们手动挂入链表类似，保存原 `FS:[0]`，将其作为下一 SEH 结构体地址，然后将 `FS:[0]` 指向当前 SEH 结构体的地址，并且在这里的异常处理函数为 `_except_handler3`。

```

00401060  push     ebp
00401061  mov      ebp,esp
00401063  push     0FFh
00401065  push     offset string "Processing ..." + 14h (00420040)
0040106A  push     offset except_handler3 (00401670)
0040106F  mov      eax,fs:[00000000]
00401075  push     eax
00401076  mov      dword ptr fs:[0],esp
0040107D  add      esp,0B4h
00401080  push     ebx
00401081  push     esi
00401082  push     edi
00401083  mov      dword ptr [ebp-18h],esp
00401086  lea      edi,[ebp-5Ch]
00401089  mov      ecx,11h
0040108E  mov      eax,0CCCCCCCCh
00401093  rep stos dword ptr [edi]
12:  _try
00401095  mov      dword ptr [ebp-4],0

```

那么出现嵌套 `_try_except` 使用时会怎么样呢，按照正常逻辑来想，肯定会出现重复挂入链表的操作（设置链表头），但是我们通过反汇编看见，编译器处理实际上只挂入了一次，并且仍然只有一个异常处理函数 `_except_handler3`。


```

10:  int main()
11:  {
00401060  push        ebp
00401061  mov         ebp,esp
00401063  push        0FFh
00401065  push        offset string "stream != NULL"+28h (00420fb0)
0040106A  push        offset __except_handler3 (00401670)
0040106F  mov         eax,fs:[00000000]
00401075  push        eax
00401076  mov         dword ptr fs:[0],esp
0040107D  add         esp,0B4h
00401080  push        ebx
00401081  push        esi
00401082  push        edi
00401083  mov         dword ptr [ebp-18h],esp
00401086  lea         edi,[ebp-5Ch]
00401089  mov         ecx,11h
0040108E  mov         eax,0CCCCCCCCh
00401093  rep stos    dword ptr [edi]
12:  _try
00401095  mov         dword ptr [ebp-4],0
13:  {
14:      _try
0040109C  mov         dword ptr [ebp-4],1
15:  {
16:      _try
004010A3  mov         dword ptr [ebp-4],2
17:  {
18:

```

能这样实现是因为编译器扩展了SEH结构体，从原先只有2个成员的结构体变成了5个成员：

```

1  struct _EXCEPTION_REGISTRATION{
2      struct _EXCEPTION_REGISTRATION *prev;
3      void (*handler)(PEXCEPTION_RECORD, PEXCEPTION_REGISTRATION, PCONTEXT,
4      PEXCEPTION_RECORD);
5      struct scopetable_entry *scopetable;
6      int trylevel;
7      int _ebp;
8  };

```

堆栈对应也就发生了变化，如下图所示为堆栈对应结构体成员：

```

0040BED0  push     ebp
0040BED1  mov      ebp,esp
0040BED3  push     0FFh
0040BED5  push     offset string "stream != NULL"+28h (00420fb0)
0040BEDA  push     offset _except_handler3 (00401670)
0040BEDF  mov      eax,fs:[00000000]
0040BEE5  push     eax
0040BEE6  mov      dword ptr fs:[0],esp
0040BEE8  add      esp,0B8h
0040BEF0  push     ebx
0040BEF1  push     esi
0040BEF2  push     edi
0040BEF3  mov      dword ptr [ebp-18h],esp
0040BEF6  lea      edi,[ebp-58h]
0040BEF9  mov      ecx,10h
0040BEFE  mov      eax,0CCCCCCCCh
0040BF03  rep stos dword ptr [edi]
6:      _try
0040BF05  mov      dword ptr [ebp-4],0
7:      {
8:
9:      }
0040BF0C  mov      dword ptr [ebp-4],0FFFFFFFh
0040BF13  jmp      $L42207+0Ah (0040bf25)
10:     _except(1)
0040BF15  mov      eax,1
$L42208:
0040BF1A  ret

```

我们来看一下再原有SEH结构体上新增的3个成员，首先是_ebp这个成员就是栈底，其次是scopetable，它是一个结构体指针，指向了scopetable_entry结构体，该结构体及成员含义如下：

```

1  struct scopetable_entry
2  {
3      DWORD previousTryLevel // 上一个_try_except程序块编号
4      PDWORD lpfnFilter // 过滤函数的起始地址
5      PDWORD lpfnHandler // 异常处理程序的地址
6  }

```

我们可以具体来调试看一下该结构体，如下所示我们使用了1个单独的_try_except代码块和1个嵌套的_try_except代码块：

```

1  void test()
2  {
3      _try
4      {
5
6      }
7      _except(1)
8      {
9          printf(123);
10     }
11
12     _try
13     {
14         _try
15         {
16
17         }
18         _except(1)
19         {
20             printf(456);

```

```

21     }
22     }
23     _except(1)
24     {
25         printf(789);
26     }
27 }

```

通过反汇编我们可以看见，当有3个_try_except程序块时就会有三个scopetable_entry结构体，它们是对应关系。成员previousTryLevel指向上一个_try_except程序块的编号，不在嵌套里就没有上一层，因此值为-1，这里我们可以看见有两个结构体的成员previousTryLevel值均为-1，第三个结构体的previousTryLevel值为1，是因为它是在嵌套_try_except程序块中，上面确实有一层_try_except程序块，因此我们可以通过这个值来判断当前在第几层嵌套中；成员lpfnFilter指向过滤函数的首地址，如图所示也就是我们的_except(过滤表达式)那个地址，通过反汇编我们可以看见这是有RET返回结果的，所以当异常发生时，代码会经过多次跳转和返回；成员lpfnHandler指向异常处理函数的首地址，也就是_except程序块内的异常处理代码，即下图所示中的printf函数部分。

```

0040BED0  push     ebp
0040BED1  mov      ebp,esp
0040BED3  push     0FFh
0040BED5  push     offset string "stream != NULL"+28h (00420fb0)
0040BEDA  push     offset __except_handler3 (00401670)
0040BEDF  mov      eax,fs:[00000000]
0040BEE5  push     eax
0040BEE6  mov      dword ptr fs:[0],esp
0040BEE8  add      esp,088h
0040BEF0  push     ebx
0040BEF1  push     esi
0040BEF2  push     edi
0040BEF3  mov      dword ptr [ebp-18h],esp
0040BEF6  lea      edi,[ebp-58h]
0040BEF9  mov      ecx,10h
0040BEFE  mov      eax,0CCCCCCCCh
0040BF03  rep stos dword ptr [edi]
6:      _try
0040BF05  mov      dword ptr [ebp-4],0
7:      {
8:
9:      }
0040BF0C  mov      dword ptr [ebp-4],0FFFFFFFFh
0040BF13  jmp      $L42215+17h (0040bf32)
10:     _except(1)
0040BF15  mov      eax,1
$L42216:
0040BF1A  ret
$L42215:
0040BF1B  mov      esp,dword ptr [ebp-18h]
11:     {
12:         printf("123");
0040BF1E  push     offset string "123" (00420024)
0040BF23  call     printf (004014c0)
0040BF28  add      esp,4
13:     }
0040BF2B  mov      dword ptr [ebp-4],0FFFFFFFFh
14:
15:     _try
0040BF32  mov      dword ptr [ebp-4],1
16:     {
17:         _try
0040BF39  mov      dword ptr [ebp-4],2
18:         {
19:
20:         }
0040BF40  mov      dword ptr [ebp-4],1
0040BF47  jmp      $L42223+17h (0040bf66)
21:     _except(1)
0040BF49  mov      eax,1
$L42224:
0040BF4F  ret

```

| Address | Hex | ASCII |
|----------|-------------|-------|
| 00420fb0 | FF FF FF FF | |
| 00420fb4 | 15 BF 40 00 | 繖 |
| 00420fb8 | 1B BF 40 00 | 繖 |
| 00420fbc | FF FF FF FF | |
| 00420fc0 | 6F BF 40 00 | 繖 |
| 00420fc4 | 75 BF 40 00 | 繖 |
| 00420fc8 | 01 00 00 00 | |
| 00420fcc | 49 BF 40 00 | 繖 |
| 00420fd0 | 4F BF 40 00 | 繖 |
| 00420fd4 | 00 00 00 00 | |
| 00420fd8 | 00 00 00 00 | |
| 00420fdc | 00 00 00 00 | |
| 00420fe0 | 00 00 00 00 | |
| 00420fe4 | 00 00 00 00 | |
| 00420fe8 | 00 00 00 00 | |
| 00420fec | 00 00 00 00 | |
| 00420ff0 | 00 00 00 00 | |
| 00420ff4 | 00 00 00 00 | |
| 00420ff8 | 00 00 00 00 | |
| 00420ffc | 00 00 00 00 | |
| 00421000 | 00 00 00 00 | |

了解了scopetable及其对应结构体的成员之后，我们再来看一下trylevel成员，该成员表示当前在哪个_try_except程序块，我们可以通过如下这段代码来看一下该值的变化：

```
1  void test()
2  {
3      _try
4      {
5          _try
6          {
7
8          }
9          _except(1)
10         {
11             printf("123");
12         }
13     }
14     _except(1)
15     {
16         printf("456");
17     }
18
19     _try
20     {
21         _try
22         {
23
24         }
25         _except(1)
26         {
27             printf("789");
28         }
29     }
30     _except(1)
31     {
32         printf("012");
33     }
34 }
```

我们接着来看这段反汇编，首先将EBP压入扩展结构体，然后将EBP提升至ESP位置，再压入trylevel，**初始值就是-1，我们也可以通过[EBP-4]来找到该值**，接着我们会发现每个_try都会有一次的trylevel的修改，根据数值来看这就像是一共索引，第一个_try_except块trylevel值为0，第二个则值为1，以此类推，如果是嵌套的_try_except块则内嵌的块指向结束之后会将该trylevel值修改为上层的索引值，如果当上层没有_try_except块时，执行完毕就会将trylevel值修改为-1。因此我们可以看见它是动态变化的，并不是固定值。

```

5:      {
0040BED0  push      ebp
0040BED1  mov       ebp,esp
0040BED3  push      0FFh
0040BED5  push      offset string "stream != NULL"+28h (00420fb0)
0040BEDA  push      offset __except_handler3 (00401670)
0040BEDF  mov       eax,fs:[00000000]
0040BEE5  push      eax
0040BEE6  mov       dword ptr fs:[0],esp
...
6:      _try
0040BF05  mov       dword ptr [ebp-4],0
7:      {
8:          _try
0040BF0C  mov       dword ptr [ebp-4],1
9:          {
10:         }
11:         mov       dword ptr [ebp-4],0
0040BF13  jmp       $L42222+17h (0040bf39)
12:         _except(1)
0040BF1C  mov       eax,1
$L42223:  ret
0040BF21  ret
$L42222:
0040BF22  mov       esp,dword ptr [ebp-18h]
13:      {
14:          printf("123");
0040BF25  push      offset string "123" (00420028)
0040BF2A  call      printf (004014c0)
0040BF2F  add       esp,4
15:      }
0040BF32  mov       dword ptr [ebp-4],0
16:      }
0040BF39  mov       dword ptr [ebp-4],0FFFFFFFFh
0040BF40  jmp       $L42218+17h (0040bf5f)
17:      _except(1)
0040BF42  mov       eax,1
$L42219:  ret
0040BF47  ret
$L42218:
0040BF48  mov       esp,dword ptr [ebp-18h]
18:      {
19:          printf("456");
0040BF4B  push      offset string "456" (00420024)
0040BF50  call      printf (004014c0)
0040BF55  add       esp,4
20:      }
0040BF58  mov       dword ptr [ebp-4],0FFFFFFFFh
21:
22:      _try
0040BF5F  mov       dword ptr [ebp-4],2
23:      {
24:          _try
0040BF66  mov       dword ptr [ebp-4],3
25:          {
26:          }
27:          mov       dword ptr [ebp-4],2

```

那么当前编译器环境中，异常处理函数_except_handler3就会根据trylevel选择scopetable数组中的结构体，然后找到结构体的lpfnFilter成员，即异常过滤函数地址。

_try_finally

除了_try_except程序块之外，编译器还提供了_try_finally，与前者不同的，在_finally块内的代码一定会去执行，无论你_try块内是否出错、中断、正常，都可以得到执行。

如下图所示，无论使用continue、break控制，还是return返回，亦或是触发异常，_finally块内的代码始终会被执行。

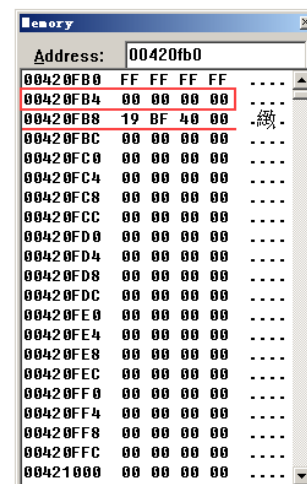


我们以return的代码为例，来看一下反汇编，如下图所示，我们可以看见_try_finally程序块时，扩展SEH结构体中的scopetable成员与_try_except不一样，首先是它的lpfnFilter过滤函数地址是空的，因此我们可以通过这个成员来判断当前是否是_finally块，其次是它的lpfnHandler异常处理函数地址指向的是_finally块中的代码地址。

```

6: {
0040BED0 55          push     ebp
0040BED1 8B EC       mov     ebp,esp
0040BED3 6A FF       push     0FFh
0040BED5 68 B0 0F 42 00 push   offset string "stream != NULL"+28h (00420fb0)
0040BEDA 68 70 16 40 00 push   offset __except_handler3 (00401670)
0040BEDF 64 A1 00 00 00 00 mov     eax,fs:[00000000]
0040BEE5 50          push     eax
0040BEE6 64 89 25 00 00 00 00 mov     dword ptr fs:[0],esp
0040BEED 83 C4 B4     add     esp,0B4h
0040BEF0 53          push     ebx
0040BEF1 56          push     esi
0040BEF2 57          push     edi
0040BEF3 8D 7D A4     lea     edi,[ebp-5Ch]
0040BEF6 B9 11 00 00 00 mov     ecx,11h
0040BEFB B8 CC CC CC CC mov     eax,0CCCCCCCCh
0040BF00 F3 AB       rep stos dword ptr [edi]
7:
8:      __try
0040BF02 C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
0040BF09 6A FF       push     0FFh
9: {
0040BF0B 8D 45 F0     lea     eax,[ebp-10h]
0040BF0E 50          push     eax
0040BF0F E8 A6 56 FF FF call    __local_unwind2 (004015ba)
0040BF14 83 C4 08     add     esp,8
10:      return;
0040BF17 EB 58     jmp     $L42210+4Ah (0040bf71)
11:      printf("Try ... \n");
12:  }
13:      __finally
14:  {
15:      printf("Finally ... \n");
0040BF19 68 2C 00 42 00 push   offset string "Except ... \n" (0042002c)
0040BF1E E8 9D 55 FF FF call    printf (004014c0)
0040BF23 83 C4 04     add     esp,4
$L42210:
0040BF26 C3          ret

```

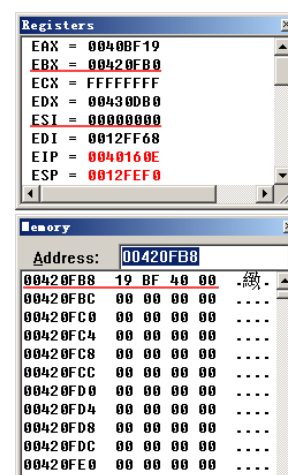


并且我们可以看见在执行return语句之前，调用了个名为_local_unwind2的函数（这个函数翻译成中文就是局部展开的意思，没有别的含义）。我们可以继续跟进_local_unwind2，会发现它调用了个函数，即[ebx+esi*4+8]，通过寄存器的值计算，我们查看对应地址为_finally块中的代码地址，这也就解释了为什么可以在return之前执行_finally块中的地址了（即_finally块中的代码一定得到执行）。

```

__local_unwind2:
004015B0 53          push     ebx
004015B1 56          push     esi
004015B2 57          push     edi
004015B3 8B 44 24 10 mov     eax,dword ptr [esp+10h]
004015B6 50          push     eax
004015B7 6A FE       push     0FEh
004015B9 68 98 15 40 00 push   offset __global_unwind2+20h (00401598)
004015BC 64 FF 35 00 00 00 00 push   dword ptr fs:[0]
004015C1 50          push     eax
004015C2 6A FE       push     0FEh
004015C4 68 98 15 40 00 push   offset __global_unwind2+20h (00401598)
004015C7 64 FF 35 00 00 00 00 push   dword ptr fs:[0]
004015CC 64 89 25 00 00 00 00 mov     dword ptr fs:[0],esp
004015D0 8B 44 24 20 mov     eax,dword ptr [esp+20h]
004015D3 8B 58 08     mov     ebx,dword ptr [eax+8]
004015D6 8B 70 0C     mov     esi,dword ptr [eax+0Ch]
004015D9 83 FE FF     cmp     esi,0FFh
004015DB 74 2E       je      __NLG_Return2+2 (00401614)
004015DE 8B 74 24     cmp     esi,dword ptr [esp+24h]
004015E1 74 28       je      __NLG_Return2+2 (00401614)
004015E4 8D 34 76     lea     esi,[esi+esi*2]
004015E7 8B 0C B3     mov     ecx,dword ptr [ebx+esi*4]
004015EA 89 4C 24 08 mov     dword ptr [esp+8],ecx
004015ED 89 48 0C     mov     dword ptr [eax+0Ch],ecx
004015F0 83 7C B3 04 00 cmp     dword ptr [ebx+esi*4+4],0
004015F3 75 12       jne     __NLG_Return2 (00401612)
004015F6 68 01 01 00 00 push   101h
004015F9 8B 44 B3 08 mov     eax,dword ptr [ebx+esi*4+8]
00401602 E8 40 00 00 00 call    __NLG_Notify (0040164e)
00401605 54 54 B3 08 call    dword ptr [ebx+esi*4+8]

```



我们接着来看下面这段代码，它有三层_try，外层是_try_except，第二、三层都是_try_finally，发生异常的代码处于第三层，按照我们之前所学习的内容来看，一旦触发异常，except_handler3函数会根据当前trylevel的值找到对应的结构体并寻找异常处理函数lpfnFilter，从内至外，从第三层开始找，通过previousTryLevel来逐层向上寻找，但由于三层、二层都是_finally块，因此过滤函数地址那一块的值是0，最终会找到第一层，也就是最外层的_except块，此时过滤表达式为1，_except块内的代码得到执行，然后就会返回。

1 void test()

```

2   {
3       _try
4       {
5           _try
6           {
7               _try
8               {
9                   *(int*)0 = 1;
10              }
11              _finally
12              {
13                  printf("Finally2 ... \n");
14              }
15          }
16          _finally
17          {
18              printf("Finally1 ... \n");
19          }
20      }
21      _except(1)
22      {
23          printf("Except ... \n");
24      }
25  }

```

此时就会有个问题，在内层的_finally块代码是否会得到执行呢？我们可以实际运行下代码，如下图所示我们可以看见，_finally块的代码都得到了执行。

```

6:      _try
0040BF05 C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
7:      {
8:          _try
0040BF0C C7 45 FC 01 00 00 00 mov     dword ptr [ebp-4],1
9:          {
10:             _try
0040BF13 C7 45 FC 02 00 00 00 mov     dword ptr [ebp-4],2
11:             {
12:                 *(int*)0 = 1;
0040BF1A C7 05 00 00 00 00 01 mov     dword ptr ds:[0],1
13:             }
0040BF24 C7 45 FC 01 00 00 00 mov     dword ptr [ebp-4],1
0040BF2B E8 02 00 00 00      call     $L42223 (0040bf32)
0040BF30 EB 0E              jmp     $L42226 (0040bf40)
14:         _finally
15:         {
16:             printf("Finally2 ... \n");
0040BF32 68 3C 00 42 00      push    offset string "Try ... \n" (0042003c)
0040BF37 E8 84 55 FF FF      call    printf (004014c0)
0040BF3C 83 C4 04              add     esp,4
$L42224:
0040BF3F C3              ret
17:         }
18:     }
0040BF40 C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
0040BF47 E8 02 00 00 00      call    $L42219 (0040bf4e)
0040BF4C EB 0E              jmp     $L42222 (0040bf5c)

```

```

C:\Program Files\
Finally2 ...
Finally1 ...
Except ...

```

那么在这里是为什么呢，实际在每个_except块内代码执行之前，都会调用一次全局展开函数，即_global_unwind2函数，该函数会从触发异常的那个try开始，依次调用局部展开，这样就可以保证finally块语句一定会得到执行。


```

24:      _except(1)
0040BF65 B8 01 00 00 00      mov     eax,1
$L42217:
0040BF6A C3                ret
↓
004016BB 5D                pop     ebp
004016BC 5E                pop     esi
004016BD 8B 5D 0C          mov     ebx,dword ptr [ebp+0Ch]
004016C0 0B C0            or      eax,eax
004016C2 74 33            je      __except_handler3+87h (004016F7)
004016C4 78 3C            js      __except_handler3+92h (00401702)
004016C6 8B 7B 08          mov     edi,dword ptr [ebx+8]
004016C9 53                push    ebx
004016CA E8 A9 FE FF FF    call    global unwind2 (00401578)
004016CF 83 C4 04          add     esp,4
004016D2 8D 6B 10          lea     ebp,[ebx+10h]

```

1.5 未处理异常

如果VEH与SEH都没有对异常进行处理，这种异常我们就称之为**未处理异常**。因为我们根据之前分析内核空间异常知道，当没有任何方法取处理异常时会调用KeBugCheckEx函数启用蓝屏，内核未处理异常的处理结果也很简单，因此本章节所学习的是用户空间的未处理异常。

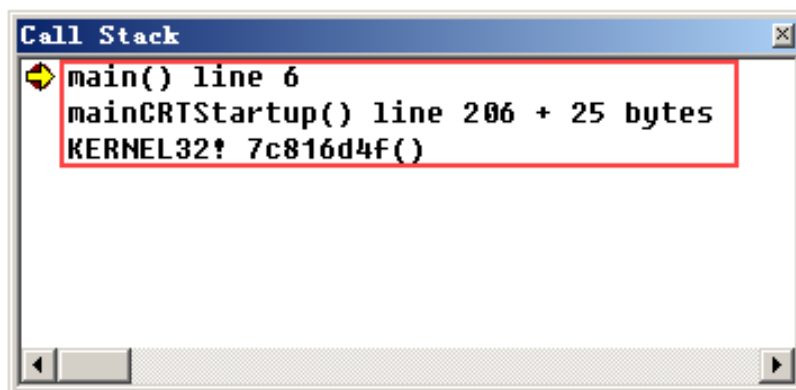
我们首先编写一个简单的代码，启动一下，看它的调用栈，我们可以看见，**程序启动时并不是直接从main函数开始执行的**，它的上层有mainCRTStartup、KERNEL32!7c816d4f()。

```


void main()
{
    int a = 1;

    getchar();
}

```



我们可以跟进**KERNEL32!7c816d4f**，最终就会发现它在这一层将SEH挂到链表上，也就是说实际上在入口程序部分，也给我们加了一道异常处理的防线。因此main函数如果发生异常，且在它的SEH链表中未能查找到能够处理异常的异常处理函数，那么_except_handler3函数则会通过previousTryLevel查找最外层的异常处理函数，也就是在这挂上去的SEH结构体中的异常处理函数。

| | | |
|---|------|------------------------------|
| 7C816D2C 6A 0C | push | 0Ch |
| 7C816D2E 68 58 6D 81 7C | push | 7C816D58h |
| 7C816D33 E8 93 B7 FE FF | call | 7C8024CB |
| 7C816D38 83 65 FC 00 | and | dword ptr [ebp-4],0 |
| 7C816D3C 6A 04 | push | 4 |
| 7C816D3E 8D 45 08 | lea | eax,[ebp+8] |
| 7C816D41 50 | push | eax |
| 7C816D42 6A 09 | push | 9 |
| 7C816D44 6A FE | push | 0FEh |
| 7C816D46 FF 15 A0 13 80 7C | call | dword ptr ds:[7C8013A0h] |
| 7C816D4C FF 55 08 | call | dword ptr [ebp+8] |
| 7C816D4F 50 | push | eax |
| 7C816D50 E8 54 5F FF FF | call | 7C80CCA9 |
|  | | |
| 7C8024CB 68 F3 99 83 7C | push | 7C8399F3h |
| 7C8024D0 64 A1 00 00 00 00 | mov | eax,fs:[00000000] |
| 7C8024D6 50 | push | eax |
| 7C8024D7 8B 44 24 10 | mov | eax,dword ptr [esp+10h] |
| 7C8024DB 89 6C 24 10 | mov | dword ptr [esp+10h],ebp |
| 7C8024DF 8D 6C 24 10 | lea | ebp,[esp+10h] |
| 7C8024E3 2B E0 | sub | esp,ebp |
| 7C8024E5 53 | push | ebx |
| 7C8024E6 56 | push | esi |
| 7C8024E7 57 | push | edi |
| 7C8024E8 8B 45 F8 | mov | eax,dword ptr [ebp-8] |
| 7C8024EB 89 65 E8 | mov | dword ptr [ebp-18h],esp |
| 7C8024EE 50 | push | eax |
| 7C8024EF 8B 45 FC | mov | eax,dword ptr [ebp-4] |
| 7C8024F2 C7 45 FC FF FF FF FF | mov | dword ptr [ebp-4],0FFFFFFFFh |
| 7C8024F9 89 45 F8 | mov | dword ptr [ebp-8],eax |
| 7C8024FC 8D 45 F0 | lea | eax,[ebp-10h] |
| 7C8024FF 64 A3 00 00 00 00 | mov | fs:[00000000],eax |
| 7C802505 C3 | ret | |

在这里，这个函数实际上就是**Kernel32.dll**模块中的**BaseProcessStart**函数，在该函数内调用了另外一个函数 **_SEH_prolog**，它的作用就是添加SEH到链表上。

```

; void __stdcall __noreturn BaseProcessStart(int ThreadInformation)
_BaseProcessStart@4 proc near          ; CODE XREF: BaseProcessStartThunk(x,x)+5↑j

uExitCode= dword ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h
ThreadInformation= dword ptr 8

; FUNCTION CHUNK AT .text:7C843612 SIZE 00000011 BYTES
; FUNCTION CHUNK AT .text:7C843628 SIZE 00000018 BYTES

; __unwind { // __SEH_prolog
push    0Ch
push    offset stru_7C816D58
call    __SEH_prolog
      ↓
push    offset __except_handler3
mov     eax, large fs:0
push    eax
mov     eax, [esp+8+arg_4]
mov     [esp+8+arg_4], ebp
lea     ebp, [esp+8+arg_4]
sub     esp, eax
push    ebx
push    esi
push    edi
mov     eax, [ebp-8]
mov     [ebp-18h], esp
push    eax
mov     eax, [ebp-4]
mov     dword ptr [ebp-4], 0FFFFFFFh
mov     [ebp-8], eax
lea     eax, [ebp-10h]
mov     large fs:0, eax
retn

```

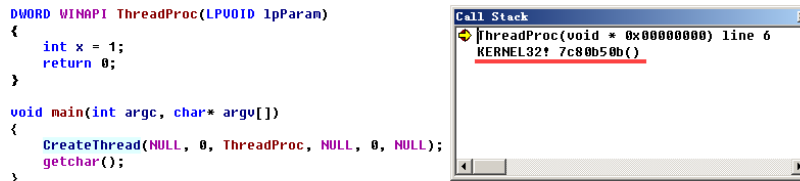
那么除了入口程序以外，如果我们另起一个线程，是否也会给我们提供一个异常处理的防线呢，我们可以编写一段代码来另起一个线程：

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  DWORD WINAPI ThreadProc(LPVOID lpParam)
5  {
6      int x = 1;
7      return 0;
8  }
9
10 void main(int argc, char* argv[])
11 {
12     CreateThread(NULL, 0, ThreadProc, NULL, 0, NULL);
13     getchar();
14 }

```

我们在新线程的代码中下断点，查看调用栈，回溯跟踪过去，会发现也会给它一个SEH挂入异常链表中：



| | | | | | |
|-------------------------------|------|---------------------------|-------------------------------|------|------------------------------|
| 7C80B4D4 6A 10 | push | 10h | 7C8024C8 68 F3 99 83 7C | push | 7C8399F3h |
| 7C80B4D6 68 18 85 80 7C | push | 7C80B518h | 7C8024D0 64 A1 00 00 00 00 | mov | eax,fs:[00000000] |
| 7C80B4D8 E8 EB 6F FF FF | call | 7C8024CB | 7C8024D6 50 | push | eax |
| 7C80B4E0 83 65 FC 00 | and | dword ptr [ebp-4],0 | 7C8024D7 8B 44 24 10 | mov | eax,dword ptr [esp+10h] |
| 7C80B4E4 64 A1 18 00 00 00 | mov | eax,fs:[00000018] | 7C8024D8 89 6C 24 10 | mov | dword ptr [esp+10h],ebp |
| 7C80B4E8 89 45 E0 | mov | dword ptr [ebp-20h],eax | 7C8024DF 8D 6C 24 10 | lea | ebp,[esp+10h] |
| 7C80B4ED 81 78 10 00 1E 00 00 | cmp | dword ptr [eax+10h],1E00h | 7C8024E3 2B E0 | sub | esp,esp |
| 7C80B4F4 75 0F | jne | 7C80B505 | 7C8024E5 53 | push | ebx |
| 7C80B4F6 80 3D 08 30 88 7C 00 | cmp | byte ptr ds:[7C803008h],0 | 7C8024E6 56 | push | esi |
| 7C80B4FD 75 06 | jne | 7C80B505 | 7C8024E7 57 | push | edi |
| 7C80B4FF FF 15 E8 12 80 7C | call | dword ptr ds:[7C8012E8h] | 7C8024E8 8B 45 F8 | mov | eax,dword ptr [ebp-8] |
| 7C80B505 FF 75 0C | push | dword ptr [ebp+0Ch] | 7C8024EB 89 65 E8 | mov | dword ptr [ebp-18h],esp |
| 7C80B508 FF 55 08 | call | dword ptr [ebp+8] | 7C8024ED 50 | push | eax |
| 7C80B50B 50 | push | eax | 7C8024EF 8B 45 FC | mov | eax,dword ptr [ebp-4] |
| 7C80B50C E8 98 17 00 00 | call | 7C80CCA9 | 7C8024F2 C7 45 FC FF FF FF FF | mov | dword ptr [ebp-4],0FFFFFFFFh |
| | | | 7C8024F9 89 45 F8 | mov | dword ptr [ebp-8],eax |
| | | | 7C8024FC 8D 45 F0 | lea | eax,[ebp-10h] |
| | | | 7C8024FF 64 A3 00 00 00 00 | mov | fs:[00000000],eax |
| | | | 7C802505 C3 | ret | |

我们也可以通过IDA来看这个地址对应的函数，即BaseThreadStart，它确实也调用了个_seh_prolog函数用于添加SEH至链表上。

```

; __stdcall BaseThreadStart(x, x)
_BaseThreadStart@8 proc near
; CODE XREF: BaseThreadStartThunk(x,x)+6↓j
; BaseFiberStart()+12↓p

var_20= dword ptr -20h
uExitCode= dword ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h
arg_0= dword ptr 8
arg_4= dword ptr 0Ch

; FUNCTION CHUNK AT .text:7C83AA5B SIZE 00000011 BYTES
; FUNCTION CHUNK AT .text:7C83AA71 SIZE 00000018 BYTES

; __unwind { // __seh_prolog
push    10h
push    offset stru_7C80B518
call    __seh_prolog

```

综上所述，我们可以得出结论大部分情况下，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，所以未处理异常的情况一般是不存在的。

我们可以将这个最后一道防线执行过程总结为如下的伪代码，当程序有异常发生时，若原先堆栈的SEH均未处理，那么这个函数一定会执行：

```

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，但这种情况如果你的程序在没有被调试的情况下是不会发生的，**只有程序被调试时，才会存在未处理异常。**

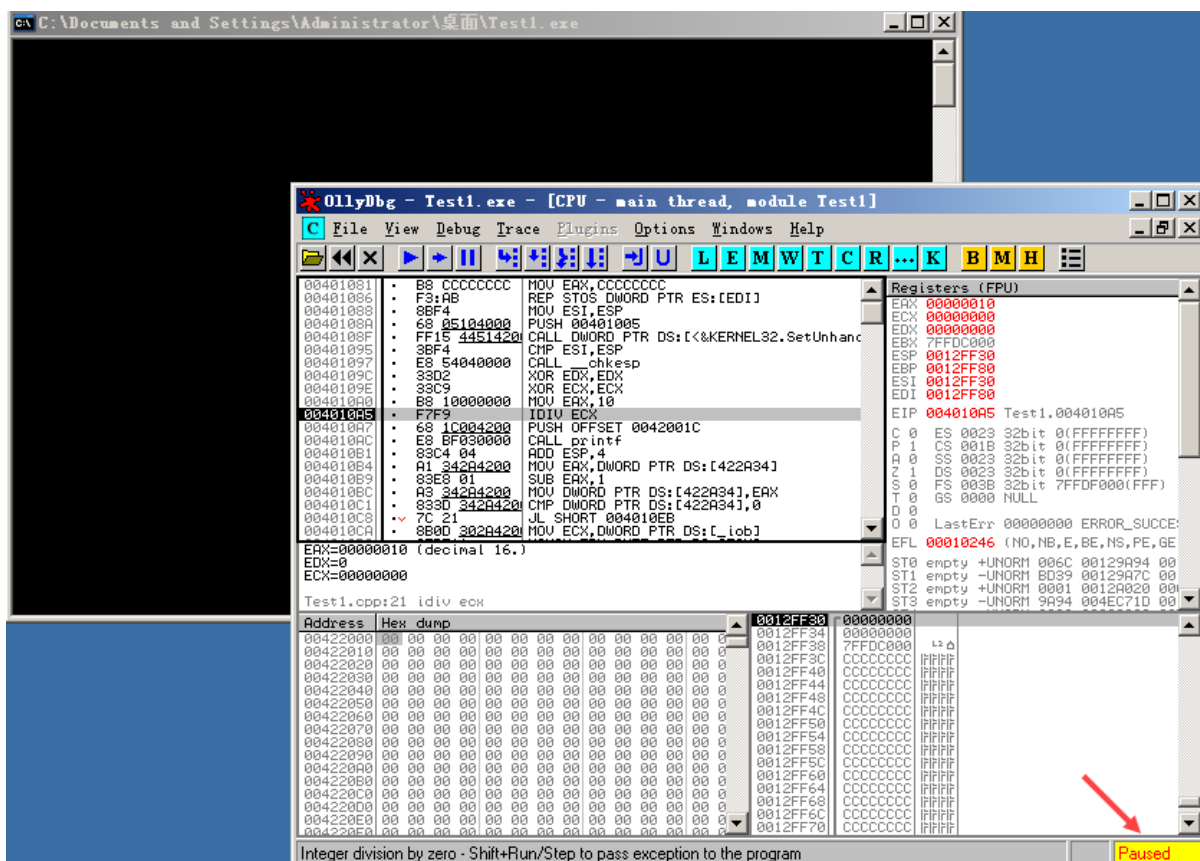
UnhandledExceptionFilter函数的执行流程如下：

1. 通过NtQueryInformationProcess函数查询当前进程是否正在被调试。
2. 如果被调试，返回EXCEPTION_CONTINUE_SEARCH，此时会进入第二轮分发。
3. 如果没有被调试：
 - a. 查询是否通过SetUnhandledExceptionFilter注册处理函数，如果有就调用；
 - b. 如果没有通过SetUnhandledExceptionFilter注册处理函数，就会弹出窗口让用户选择终止程序还是启动即时调试器；
 - c. 如果用户没有启用即时调试器，那么该函数返回EXCEPTION_EXECUTE_HANDLER。

因此我们可以通过如下的代码来进行反调试，编译这个程序，正常打开是会执行printf的，但如果通过OD打开则会停止运行。

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  long _stdcall callback(_EXCEPTION_POINTERS* excp)
5  {
6      excp->ContextRecord->EcX = 1;
7      return EXCEPTION_CONTINUE_EXECUTION;
8  }
9
10 void main(int argc, char* argv[])
11 {
12     // 注册一个最顶层异常处理函数
13     SetUnhandledExceptionFilter(callback);
14
15     // 除0异常
16     _asm
17     {
18         xor edx,edx
19         xor ecx,ecx
20         mov eax,0x10
21         idiv ecx
22     }
23
24     // 程序正常执行
25     printf("Running ... ");
26     getchar();
27 }
```

如下图所示我们使用纯净版的Ollydbg调试该程序就没法执行正常的代码，如果我们使用带插件版的OD，如吾爱破解的OD则可以直接绕过反调试：



绕过反调试的原理很简单，我们知道UnhandledExceptionFilter函数是通过NtQueryInformationProcess来判断是否被调试的，而NtQueryInformationProcess是通过DebugPort的值来判断程序是否正在被调试，因此我们只需要Hook了NtQueryInformationProcess，修改DebugPort的值就可以绕过反调试了。

最后我们再来看一下KiUserExceptionDispatcher函数，它首先会调用RtlDispatchException，这个函数包括了对VEH、SEH的查找，以及查看是否存在顶层函数，以及是否被调试。全部都判断完了以后，返回一个布尔值，返回为真，调用ZwContinue再进入0环，返回为假，调用ZwRaiseException进行第二轮异常分发。

```

mov     ecx, [esp+arg_0]
mov     ebx, [esp+0]
push    ecx
push    ebx
call    _RtlDispatchException@8          ; RtlDispatchException(x,x)

or      al, al
jz      short loc_7C92EB0A

pop     ebx
pop     ecx
push    0
push    ecx
call    _ZwContinue@8                   ; ZwContinue(x,x)

jmp     short loc_7C92EB15

; -----

loc_7C92EB0A:                          ; CODE XREF: KiUserExceptionDispatcher(x,x)+10↑j
pop     ebx
pop     ecx
push    0
push    ecx
push    ebx
call    _ZwRaiseException@12            ; ZwRaiseException(x,x,x)

```

最后我们就可以了解整个异常分发、处理的过程了，如下图所示：

