

# 1 APC的本质

## 1.1 了解APC

线程是不能被结束、暂停、恢复的，线程在执行的时候自己占据着CPU。设想一种极端的例子：如果不调用API、屏蔽中断，并保证代码不出现异常，线程将永久占用CPU，线程只有自己执行代码将自己结束。

如果我们想要控制线程的行为，就需要提供给它一个函数，让它自己去调用，**这个函数就是APC (Asynchroneus Procedure Call，即异步过程调用)**。

## 1.2 APC队列

线程结构体0x34偏移位成员指向了一个结构体\_KAPC\_STATE，该结构体有5个成员：

```

1  0: kd> dt _KAPC_STATE
2  nt!_KAPC_STATE
3      +0x000 ApcListHead      : [2] _LIST_ENTRY // 2个APC队列（2个双向链表，单表
        占用8字节，4字节指向链表头，4字节指向链表尾），分别是用户APC、内核APC
4      +0x010 Process          : Ptr32 _KPROCESS // 线程所属或者所挂靠的进程
5      +0x014 KernelApcInProgress : UChar // 内核APC是否正在执行
6      +0x015 KernelApcPending  : UChar // 是否有正在等待执行的内核APC
7      +0x016 UserApcPending    : UChar // 是否有正在等待执行的用户APC

```

成员ApcListHead有2个APC队列，本质上就是双向链表，在双向链表内存储的就是给线程提供的APC函数，你想让线程做什么事情，就给给它提供一个APC函数挂进这个双向链表内，挂进去之后线程在某一时刻会去检查这个链表，当发现链表内有函数就会去调用，因此你可以在提供的APC函数中来控制线程的行为，当线程去检查列表且执行就会进行设定的行为。


成员Process表示当前线程所属或者所挂靠的进程，如果当前线程没有所挂靠的进程，那么该成员的值与ETHREAD结构体0x220偏移位成员的值一致。

```

0: kd> dt _KTHREAD
nt!_KTHREAD
+0x000 Header          : _DISPATCHER_HEADER
+0x010 MutantListHead  : _LIST_ENTRY
+0x018 InitialStack    : Ptr32 Void
+0x01c StackLimit      : Ptr32 Void
+0x020 Teb             : Ptr32 Void
+0x024 TlsArray         : Ptr32 Void
+0x028 KernelStack     : Ptr32 Void
+0x02c DebugActive      : UChar
+0x02d State           : UChar
+0x02e Alerted         : [2] UChar
+0x030 Iopl            : UChar
+0x031 NpxState        : UChar
+0x032 Saturation       : Char
+0x033 Priority         : Char
+0x034 ApcState        : _KAPC_STATE

```

0: kd> dt \_KAPC\_STATE  
 nt!\_KAPC\_STATE  
 +0x000 ApcListHead : [2] \_LIST\_ENTRY  
 +0x010 Process : Ptr32 \_KPROCESS  
 +0x014 KernelApcInProgress : UChar  
 +0x015 KernelApcPending : UChar  
 +0x016 UserApcPending : UChar



### 1.3 APC结构

我们向APC队列中挂入的函数，准确来说不能称之为函数，而是一个**APC结构体**。这个结构看着有些复杂，在本章节我们只需要知道0x1c偏移位成员NormalRoutine，**该成员的作用就是可以帮助我们找到提供的APC函数地址。**

```
0: kd> dt _KAPC
nt!_KAPC
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 Spare0         : Uint4B
+0x008 Thread         : Ptr32 _KTHREAD
+0x00c ApcListEntry   : _LIST_ENTRY
+0x014 KernelRoutine  : Ptr32      void
+0x018 RundownRoutine : Ptr32      void
+0x01c NormalRoutine  : Ptr32      void
+0x020 NormalContext  : Ptr32 Void
+0x024 SystemArgument1 : Ptr32 Void
+0x028 SystemArgument2 : Ptr32 Void
+0x02c ApcStateIndex  : Char
+0x02d ApcMode        : Char
+0x02e Inserted       : UChar
```

### 1.4 APC执行

KiServiceExit函数是**系统调用、异常或中断**返回用户空间的必经之路，因此当线程调用API、程序出现异常或中断时就会**调用KiServiceExit函数**，该函数会检查当前KTHREAD.KAPC\_STATE.UserApcPending是否为0，即表示是否有用户空间的APC请求，如果有的话就会向下继续走，**最后KiDeliverApc函数专门用于处理APC**。（只判断用户空间的APC请求是因为在KiDeliverApc函数中会优先处理内核空间的APC然后处理用户空间的APC，因此我们不需要再多余进行判断）

```
mov     ebx, large fs:124h          ; 获取KTHREAD
mov     byte ptr [ebx+2Eh], 0       ; KTHREAD+0x2e -> Alerted
cmp     byte ptr [ebx+4Ah], 0       ; KTHREAD+0x4a -> KTHREAD -> KAPC_STATE -> UserApcPending
; 检查UserApcPendings是否为0，即表示是否有用户空间的APC请求
jz      short loc_406F87

mov     ebx, ebp
mov     [ebx+44h], eax
mov     dword ptr [ebx+50h], 38h ; ';'
mov     dword ptr [ebx+38h], 23h ; '#'
mov     dword ptr [ebx+34h], 23h ; '#'
mov     dword ptr [ebx+30h], 0
mov     ecx, 1                     ; NewIrql
call    ds:__imp_@KfRaiseIrql@4    ; KfRaiseIrql(x)

push    eax
sti
push    ebx
push    0
push    1
call    _KiDeliverApc@12            ; 执行内核APC，并为用户空间的APC执行进行准备
```

## 2 备用APC队列

在上一章节的学习中，我们知道如果想控制线程的行为，就需要给它的APC队列里面挂一个APC，也就是在线程结构体0x34偏移位成员ApcState的成员ApcListHead其中一个链表中挂入，除了ApcState成员以外，在线程结构体KTHREAD中存着与APC有关的字段不止这一处。

如下所示KTHREAD结构体中，除了ApcState，还有4处与APC有关的成员，本章将会依次介绍这几个字段的含义，并学习一个新的知识，即备用APC队列。

```

1  0: kd> dt _KTHREAD
2  nt!_KTHREAD
3      ...
4      +0x034 ApcState           : _KAPC_STATE
5      ...
6      +0x138 ApcStatePointer   : [2] Ptr32 _KAPC_STATE
7      ...
8      +0x14c SavedApcState     : _KAPC_STATE
9      ...
10     +0x165 ApcStateIndex      : Uchar
11     +0x166 ApcQueueable      : Uchar
12     ...

```

### 2.1 SavedApcState

线程APC队列中的APC函数都是与进程相关联的，如A进程的1线程中的所有APC函数要访问的内存地址都是A进程的。但线程是可以挂靠到其他的进程，如A进程的线程1通过修改Cr3，即改为B进程的页目录基址，就可以访问B进程地址空间，即所谓的进程挂靠。

当A进程的1线程挂靠B进程后，APC队列中存储的却仍然是原来的APC，那么如果某个APC函数要读取一个地址为0x12345678的数据，读到的将是B进程的地址空间，这样逻辑就错误了。

为了避免混乱，在1线程挂靠B进程时，会将ApcState中的值暂时存储到SavedApcState中，等回到原进程A时，再将APC队列恢复。所以，SavedApcState又称为备用APC队列。

那么在这种进程挂靠的场景下，也是可以向线程APC队列中插入APC的，ApcState内所存储的就是B进程相关的APC信息（挂靠进程），而SavedApcState所存储的就是A进程相关的APC信息（所属进程）。

### 2.2 ApcStatePointer

为了方便寻址，在KTHREAD结构体中定义了一个指针数组ApcStatePointer，数组内有两个成员。

该成员存储的信息也分为两种场景，分别是正常场景和挂靠场景：

```

1  // 正常场景
2  ApcStatePointer[0] // 指向 ApcState
3  ApcStatePointer[1] // 指向 SavedApcState
4
5  // 挂靠场景
6  ApcStatePointer[0] // 指向 SavedApcState
7  ApcStatePointer[1] // 指向 ApcState

```

## 2.3 ApcStateIndex

ApcStateIndex用来标识当前线程处于什么状态，**正常状态下该值为0，挂靠状态下该值为1**。那么我们将该成员结合ApcStatePointer，就会发现一个设计细节的地方。

无论是正常场景，还是挂靠场景，我们使用**ApcStatePointer[ApcStateIndex]**的组合方式，都可以获取到ApcState，即线程当前使用的APC信息。

## 2.4 ApcQueueable

ApcQueueable用于表示**是否可以向线程的APC队列中挂入APC**。当线程正在执行退出的代码时，会将这个值设置为0，如果此时执行插入APC的代码（在内核中会使用KeInsertQueueApc插入APC），在插入函数中会判断这个值的状态，如果为0则插入失败。

## 3 APC挂入过程

无论是正常状态还是挂靠状态，都有两个APC队列，一个是内核队列，一个是用户队列。每当要挂入一个APC时，不管是内核空间还是用户空间的函数，内核都要准备一个KAPC的数据结构，并将这个结构挂到相应的APC队列中。

### 3.1 KAPC结构体

我们要了解APC挂入过程，先要了解KAPC这个数据结构，知道它每个成员的作用。如下所示，我们可以从注释中了解每个成员的作用和含义。

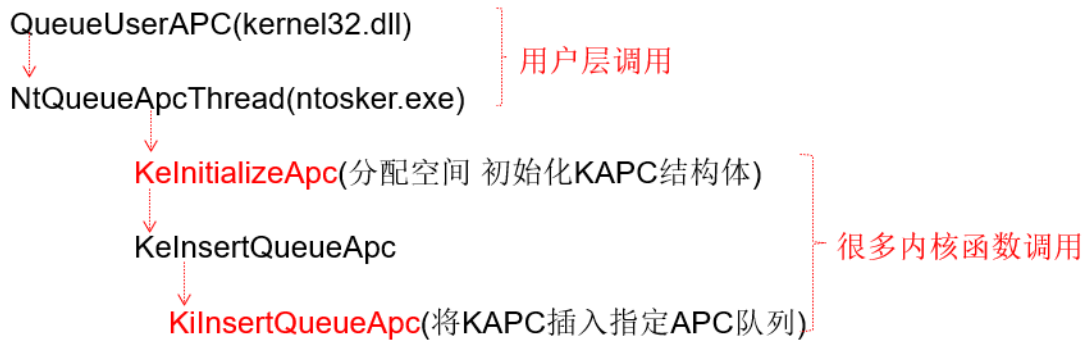
```

1  0: kd> dt _KAPC
2  nt!_KAPC
3      +0x000 Type           : Int2B           // APC的类型：0x12
4      +0x002 Size           : Int2B           // 当前结构体的大小：0x30
5      +0x004 Spare0         : Uint4B          // 该成员未使用
6      +0x008 Thread         : Ptr32 _KTHREAD  // APC所属的线程
7      +0x00c ApcListEntry   : _LIST_ENTRY     // APC所挂入的队列，存放的是双向
链表，2个前后成员的地址
8      +0x014 KernelRoutine  : Ptr32 void      // 指向一个函数，在对应函数内调用
了ExFreePoolWithTag来释放APC，当APC执行完毕之后，内核程序就会调用该成员指定的函数进
行释放
9      +0x018 RundownRoutine : Ptr32 void      // 该成员未使用
10     +0x01c NormalRoutine  : Ptr32 void      // 如果当前APC为用户空间的，则为
用户APC的总入口，反之为真正的内核APC函数，因此通过它可以帮助我们找到提供的APC函数地址
11     +0x020 NormalContext  : Ptr32 Void      // 如果当前APC为内核APC，则该值
没有意义，如果为用户APC，则该值为真正的APC函数
12     +0x024 SystemArgument1 : Ptr32 Void     // APC函数的参数
13     +0x028 SystemArgument2 : Ptr32 Void     // APC函数的参数
14     +0x02c ApcStateIndex  : Char            // 这个值与KTHREAD结构体的
0x165偏移位成员一样，但不同的是这里的这个成员有4个值：0、1、2、3
15     +0x02d ApcMode        : Char            // 当前APC的模式：内核、用户
16     +0x02e Inserted       : UChar           // 表示本APC是否已挂入队列，挂入
前：0，挂入后：1

```

### 3.2 挂入流程

APC挂入流程大致如下，如果是用户层的话会先通过QueueUserAPC → NtQueueApcThread，然后到内核层 KeInitializeApc → KeInsertQueueApc → KiInsertQueueApc。



但也不完全如此，很多内核函数调用时会直接的去使用KeInitializeApc、KiInsertQueueApc，也就不完全是这么一个挂入流程了，因此我们想要真正了解挂入流程，就需要了解这两个函数。

### 3.2.1 KeInitializeApc

KeInitializeApc函数的使用格式如下，我们可以看见传递的每个参数都是用来给KAPC结构体成员赋值用的，也就表示KeInitializeApc函数就是用于创建并初始化一个APC的。

```

1  VOID KeInitializeApc(
2      IN PRKAPC
   Apc,                                // KAPC结构体指针
3      IN PKTHREAD
   Thread,                            // 目标线程，对应KAPC.Thread
4      IN KAPC_ENVIRONMENT Environment, // 四种状态，对应
   KAPC.ApcStateIndex
5      IN PKKERNEL_ROUTINE KernelRoutine, // 销毁KAPC的函数地址，对应
   KAPC.KernelRoutine
6      IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL, // 未使用
7      IN PKNORMAL_ROUTINE NormalRoutine,           // 对应
   KAPC.NormalRoutine
8      IN KPROCESSOR_MODE ApcMode,                  // 挂入的APC模式：内核、用
   户，对应KAPC.ApcMode
9      IN PVOID
   NormalContext                        // 对应KAPC.NormalContext
10 )
  
```

KeInitializeApc本身的功能很简单，但是对Kapc.ApcStateIndex的赋值，我们需要了解一下，虽然它与KTHREAD.ApcStateIndex同名，但含义不完全一样，因为当前该成员有4个值：0、1、2、3，它们的含义如下：

```

1  0：原始环境
2  1：挂靠环境
3  2：当前环境
4  3：插入APC时的当前环境
  
```

值为0和1我们好理解，但是2和3我们就需要来具体分析代码了，如下图所示因为不确定此时的环境和先前的环境是否一致（可能发生挂靠），所以这里会进行判断，如果Environment值为2就会跳转到一处代码，将当前线程的ApcStateIndex赋值给DL寄存器，并跳回去之后将DL寄存器的值赋给KAPC.ApcStateIndex。

```

mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]          ; KAPC结构体指针
mov     edx, [ebp+arg_8]          ; Environment
cmp     edx, 2                    ; 判断值是否为2
mov     ecx, [ebp+arg_4]          ; Thread
mov     word ptr [eax], 12h        ; KAPC.Type -> 0x12
mov     word ptr [eax+2], 30h      ; KAPC.Size -> 0x30
jz      loc_4101B0                ; 如果挂入APC时的环境 (Environment) 为2则跳转

loc_4101B0:
mov     dl, [ecx+165h]             ; CODE XREF: KeInitializeApc(x,x,x,x,x,x,x,x)+1C1j
jmp     loc_40F23D                ; 将KTHREAD.ApcStateIndex给到DL寄存器

loc_40F23D:
mov     [eax+8], ecx              ; CODE XREF: KeInitializeApc(x,x,x,x,x,x,x,x)+F9B4j
mov     ecx, [ebp+arg_C]
mov     [eax+14h], ecx
mov     ecx, [ebp+arg_10]
mov     [eax+2Ch], dl
mov     [eax+18h], ecx            ; 将DL给到KAPC.ApcStateIndex

```

这里就是确保在初始化APC时，挂入的APC队列一定是线程当前环境对应的。

### 3.2.2 KiInsertQueueApc

我们知道了Environment值为2时的作用，当该值为3表示插入APC时的当前环境，我们就需要来分析插入APC的函数，即KiInsertQueueApc。该函数使用方法很简单，只有2个参数：

```

1  VOID FASTCALL KiInsertQueueApc
2  (
3      IN PKAPC Apc,                // KAPC指针（指向一个已经初始化完成的APC）
4      IN KPRIORITY PriorityBoost
5  )

```

我们可以通过IDA来看一下，该函数最开始有两个判断，第一个判断是KAPC.Inserted的值是否为0，如果不为0就说明已经挂入到队列中，那就进行跳转，最终RET；第二个判断是KAPC.ApcStateIndex的值是否为3，如果为3则表示当前在插入APC时的环境（也就是执行KiInsertQueueApc时的环境），就会跳转进一块代码，将当前线程的ApcStateIndex赋值给DL，然后再通过DL赋值给KAPC.ApcStateIndex。

```

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx                       ; KAPC指针
mov     eax, ecx
cmp     byte ptr [eax+2Eh], 0      ; 判断KAPC+0x2E(Inserted)的值是否为0
mov     ecx, [eax+8]
jnz     loc_44BA25                ; KAPC.Inserted不为0，说明已经挂入APC队列，就跳转

loc_44BA25:
xor     al, al
leave   ret                       ; CODE XREF: KiInsertQueueApc(x,x)+12fj

cmp     byte ptr [eax+2Ch], 3      ; 判断KAPC+0x2C(ApcStateIndex)的值是否为3
jz      loc_44BA29                ; 如果为3则跳转，也就表示“插入APC时的当前环境”

loc_44BA29:
mov     dl, [ecx+165h]             ; CODE XREF: KiInsertQueueApc(x,x)+1C1j
mov     [eax+2Ch], dl             ; 将KTHREAD+0x165(ApcStateIndex)的值给到DL
jmp     loc_402C94                ; 在通过DL给到KAPC.ApcStateIndex

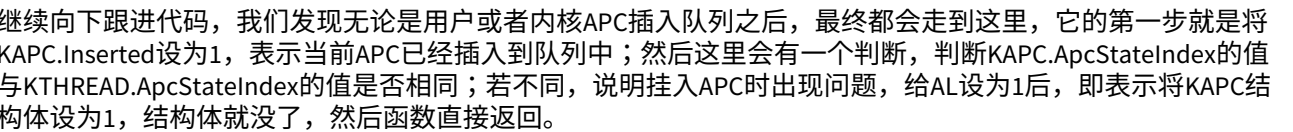
```

接着我们向下看代码流程，首先是判断KAPC.NormalRoutine是否为0，不为0则跳转，跳转之后又做了一个判断，判断DL的值，也就KAPC.ApcNode的值是否为0，如果为0也就表示当前要插入的为内核APC，然后进行跳转，将APC插入到对应的内核APC队列中，最后跳回主代码；

如果DL的值不为0，则会判断该KAPC.KernelRoutine是不是PsExitSpecialApc，如果是，则会把对应的APC挂入到队列中，之后跳回主代码，这里有个细节需要注意，再挂入队列之前会将UserApcPending的值设为1；如果不是，那么也会走下面的代码，将用户APC插入带对应的用户APC队列中，最后跳回主代码。

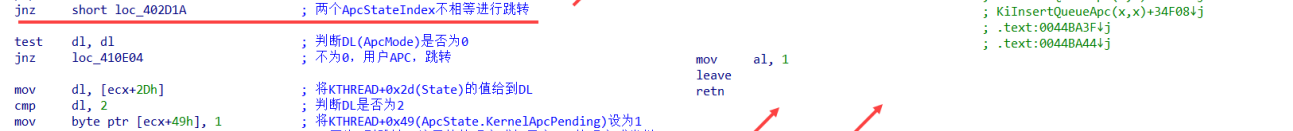
接着我们看Kapc.NormalRoutine的值为0，以及链表为空的情况下的情况，向下走就直接将当前APC挂入到对应的队列中。





若以上条件均满足，则会执行KiUnwaitThread函数，将当前线程从等待链表里取出，挂到就绪链表中，也就是把当前线程唤醒，才有机会执行APC函数；若条件不满足，那么APC函数就无法得到执行，当然还有一种情况，如果当前APC因条件不满足而没法执行，但是它已经位于APC队列中，如果下一个APC插入时，满足唤醒线程的条件，就有可能出现两个APC依次执行的情况。

loc_402CFC:		loc_402D1A:
	; CODE XREF: KiInsertQueueApc(x,x)+DFB64j	; CODE XREF: KiInsertQueueApc(x,x)+711j
	KiInsertQueueApc(x,x)+2B0704j	KiInsertQueueApc(x,x)+C8B44j
movsx edi, byte ptr [eax+2Ch]	将KACPC+0x2c (ApcStateIndex) 给到EDI	KiInsertQueueApc(x,x)+C8B64j
mov ptr [eax+2Eh], 1	将KACPC.Inserted设为1. 表示APC已经插入到队列中	KiInsertQueueApc(x,x)+C8B84j
movzx esi, byte ptr [ecx+165h]	将当前线程的ApcStateIndex给到ESI	KiInsertQueueApc(x,x)+E1964j
cmp edi, esi	两个ApcStateIndex进行比较	KiInsertQueueApc(x,x)+E1A04j
pop edi	恢复现场	KiInsertQueueApc(x,x)+128D34j
pop esi		KiInsertQueueApc(x,x)+146FB4j





```

loc_410E04:                                ; CODE XREF: KiInsertQueueApc(x,x)+75↑j
cmp     byte ptr [ecx+2Dh], 5              ; 判断KTHREAD.State的值是否为5
jnz     loc_402D1A                        ; 不为5跳转

cmp     byte ptr [ecx+59h], 1              ; 判断KTHREAD.WaitMode的值是否为1
jnz     loc_402D1A                        ; 不为1跳转

cmp     byte ptr [ecx+164h], 0             ; 判断KTHREAD.Alertable的值是否为0
jz      loc_437B76                        ; 为0跳转

loc_410E25:                                ; CODE XREF: KiInsertQueueApc(x,x)+34F0E↓j
mov     byte ptr [ecx+4Ah], 1              ; 当Alertable的值不为0, 则将KTHREAD.ApcState.UserApcPending设为1
push    0
mov     edx, 0C0h
jmp     loc_40F51F

                                ↓
loc_40F51F:                                ; CODE XREF: KiInsertQueueApc(x,x)+E1BE↓j
push    [ebp+var_4]
call    @KiUnwaitThread@16                ; KiUnwaitThread(x,x,x,x)

```

这里我们提到了以后新的东西，即KTHREAD.Alertable，该成员表示当前线程是否可以被唤醒，这个值可以被SleepEx、WaitForSingleObjectEx等能让线程进入等待状态的函数修改。

```

DWORD SleepEx(
    DWORD dwMilliseconds, // time-out interval
    BOOL bAlertable        // early completion option
);

DWORD WaitForSingleObjectEx(
    HANDLE hHandle,        // handle to object
    DWORD dwMilliseconds, // time-out interval
    BOOL bAlertable        // alertable option
);

```

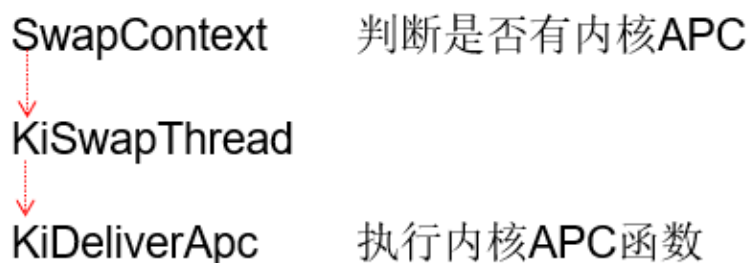
## 4 内核APC执行过程

APC函数的执行与插入并不是同一个线程，在A线程中向B线程插入一个APC，插入的动作是在A线程中完成的，但什么时候执行则由B线程决定，所以称之为**异步过程调用**。内核APC函数与用户APC函数的执行时间和执行方式也有区别，我们本章节主要学习内核APC的执行过程。

### 4.1 执行点

#### 4.1.1 线程切换

我们要知道内核APC函数在哪个执行点执行，**第一个执行点就是线程切换**，在SwapContext函数内判断是否有内核APC，在KiDeliverApc函数内执行内核APC函数。



首先我们来看一下SwapContext，在该函数内有一处判断，判断ESI+0x49（即KTHREAD.ApcState.KernelApcPending）是否为0，也就表示判断当前内核APC队列是否有等待执行的APC函数，如果不为0的情况下就进行跳转，最终RET。我们可以看见整个链是SwapContext ← KiSwapContext ← KiSwapThread。在SwapContext函数内，实际上是将判断的结果给到了AL，而后再在KiSwapThread调用完KiSwapContext之后，将AL值进行了比较，当AL的值不为0时则进行跳转。

```

loc_405F69:                                ; CODE XREF: SwapContext+E2↑j
mov     eax, [ebx+18h]
mov     ecx, [ebx+3Ch]
mov     [ecx+3Ah], ax
shr     eax, 10h
mov     [ecx+3Ch], al
mov     [ecx+3Fh], ah
inc     dword ptr [esi+4Ch]
inc     dword ptr [ebx+61Ch]
pop     ecx
mov     [ebx], ecx
cmp     byte ptr [esi+49h], 0              ; 判断KTHREAD+0x49(ApcState.KernelApcPending)是否为0, 不为0则进行跳转
jnz     short loc_405F92

loc_405F92:                                ; CODE XREF: SwapContext+115↑j
popf
jnz     short loc_405F98

mov     al, 1
retn

call     SwapContext

mov     ebp, [esp+10h+var_10]
mov     edi, [esp+10h+var_C]
mov     esi, [esp+10h+var_8]
mov     ebx, [esp+10h+var_4]
add     esp, 10h

loc_40B1EC:                                ; CODE XREF: KiSwapThread()+5F1F↓j
mov     ecx, eax
call     @KiSwapContext@4                ; KiSwapContext(x)

test     al, al
mov     cl, [edi+58h]                     ; NewIrql
mov     edi, [edi+54h]
mov     esi, ds:__imp_@KfLowerIrql@4      ; KfLowerIrql(x)
jnz     loc_4164CB

```

接着我们向下看当AL值为1，跟随跳转，就会去将eax的值清0，然后传递给KiDeliverApc函数，该函数的作用就是根据参数1的值，来判断执行哪个空间的APC函数。

```

loc_4164CB:                                ; CODE XREF: KiSwapThread()+46↑j
mov     cl, 1                             ; NewIrql
call     esi ; KfLowerIrql(x)             ; KfLowerIrql(x)

xor     eax, eax                          ; 清0操作
push    eax                              ; 参数3
push    eax                              ; 参数2
push    eax                              ; 参数1, 当值为0时处理内核APC, 当值为1时候处理内核APC和用户APC
call    _KiDeliverApc@12                  ; KiDeliverApc(x,x,x)

```

#### 4.1.2 系统调用、中断或者异常

第二个执行点是系统调用、中断或者异常，因此当线程调用API、程序出现异常或中断时返回用户空间调用**KiServiceExit**函数，该函数会检查当前KTHREAD.ApcState.UserApcPending是否为0，即表示是否有用户空间的APC请求，如果有的话就会向下继续走，通过**KiDeliverApc**函数执行APC，因为第一个参数为1，所以会先执行**内核APC**，然后执行**用户APC**。因此我们也知道在此处如果没有用户APC，也就不会执行内核APC。

```

mov     ebx, large fs:124h           ; 获取KTHREAD
mov     byte ptr [ebx+2Eh], 0       ; KTHREAD+0x2e -> Alerted
cmp     byte ptr [ebx+4Ah], 0       ; KTHREAD+0x4a -> KTHREAD -> KAPC_STATE -> UserApcPending
                                           ; 检查UserApcPendings是否为0，即表示是否有用户空间的APC请求
jz      short loc_406F87

mov     ebx, ebp
mov     [ebx+44h], eax
mov     dword ptr [ebx+50h], 3Bh ; ';'
mov     dword ptr [ebx+38h], 23h ; '#'
mov     dword ptr [ebx+34h], 23h ; '#'
mov     dword ptr [ebx+30h], 0
mov     ecx, 1                     ; NewIrql
call    ds:__imp_@KfRaiseIrql@4    ; KfRaiseIrql(x)

push    eax
sti
push    ebx
push    0
push    1
call    _KiDeliverApc@12           ; 执行内核APC，并为用户空间的APC执行进行准备

```

## 4.2 KiDeliverApc函数

我们知道了无论什么点触发APC的执行，最终都会通过KiDeliverApc函数来执行APC，因此我们分析该函数就可以了解执行的过程。

通过IDA打开内核文件（Ntoskrnl.exe）找到KiDeliverApc函数开始分析，首先我们可以看见会将KernelApcPending设为0，获取内核APC队列地址，接着**判断队列（链表）是否为空**，如果不为空则表示队列中有APC，就会进行跳转。

```

mov     eax, large fs:124h           ; 将FS:0x124(CurrentThread[KTHREAD])给到EAX
mov     esi, eax                   ; EAX给到ESI
mov     eax, [esi+134h]
mov     [ebp+var_1C], eax
mov     eax, [esi+44h]
mov     [esi+134h], ecx
lea     ecx, [esi+0E8h]           ; SpinLock
lea     edx, [ebp+LockHandle]     ; LockHandle
mov     [ebp+BugCheckParameter1], eax
call    ds:__imp_@KeAcquireInStackQueuedSpinLock@8 ; KeAcquireInStackQueuedSpinLock(x,x)

mov     byte ptr [esi+49h], 0       ; 将ESI+0x49(KTHREAD.ApcState.KernelApcPending)设为0
lea     ebx, [esi+34h]           ; 将ESI+0x34(KTHREAD.ApcState.ApcListHead)，即内核APC队列地址给到EBX

loc_4063A9:
                                           ; CODE XREF: IopCompleteRequest(x,x,x,x,x)+21D↓j
                                           ; IopCompleteRequest(x,x,x,x,x)+7156↓j
                                           ; .text:0044B9F5↓j
cmp     [ebx], ebx                ; 微软链表设计时，内容为空不会存空值，而会存当前链表的地址，即[ebx] == ebx
                                           ; 因此使用CMP判断两者是否相等，如果不相等则表示队列中有APC
jnz     loc_416444                ; 不相等即跳转

```

接着跟进跳转，我们可以看见获取到KAPC的首地址，将几个成员KernelRoutine、NormalRoutine、NormalContext、SystemArgument1、SystemArgument1取出，分别压入栈中，便于后续的使用，在这操作之间有一层判断，**判断NormalRoutine是否为0**，即表示当前内核APC函数是否为空，不为空则表示存在APC函数，就会进行跳转。

```

loc_416444:                                ; CODE XREF: KiDeliverApc(x,x,x)+57↑j
mov     eax, [ebx]                        ; 将KACP结构体地址给到EAX, 需要注意的是, 这里给的地址并不是结构体首地址
; 因为在插入APC时给的地址就是KACP.ApcListEntry成员位的地址
lea     edi, [eax-0Ch]                    ; 通过EAX-0xc的方式将KACP结构体首地址给到EDI
mov     ecx, [edi+14h]                    ; 如下几行指令操作就是将KernelRoutine、NormalRoutine
; NormalContext、SystemArgument1、SystemArgument2取出, 存放之内存中, 便于后续使用

mov     [ebp-14h], ecx
mov     ecx, [edi+1Ch]
test    ecx, ecx                          ; 判断NormalRoutine是否为0
mov     [ebp-4], ecx
mov     edx, [edi+20h]
mov     [ebp-10h], edx
mov     edx, [edi+24h]
mov     [ebp-0Ch], edx
mov     edx, [edi+28h]
mov     [ebp-8], edx
jnz     loc_41D364                        ; 不为0则进行跳转

```

继续跟进跳转, 我们可以看见会进行两个判断, 即判断当前是否有内核APC正在运行以及当前是否禁用了内核APC, 如果都没有的话继续向下走, 将当前APC从队列中进行摘除, 调用KernelRoutine对应的函数, 该成员指向的是释放APC的函数, 接着会再判断一次NormalRoutine是否为0, 不为0则会先将KernelApcInProgress设为1, 然后执行NormalRoutine对应的函数, 即内核APC函数, 并传入三个值: NormalContext、SystemArgument1、SystemArgument2。

```

loc_41D364:                                ; CODE XREF: IopCompleteRequest(x,x,x,x,x)+1DD↑j
; __unwind { // __SEH_prolog
cmp     byte ptr [esi+48h], 0              ; 判断ESI+0x48(KAPC.KernelApcInProgress)是否为0
jnz     loc_4063BE                        ; 非0, 表示当前有内核APC正在运行, 跳转

cmp     dword ptr [esi+0D4h], 0            ; 判断ESI+0xD4(KTHREAD.KernelApcDisable)是否为0
jnz     loc_4063BE                        ; 非0, 表示当前内核APC被禁用了, 跳转

mov     ecx, [eax]                        ; 从APC队列中摘除当前APC
mov     eax, [eax+4]
mov     [eax], ecx
mov     [ecx+4], eax
lea     ecx, [ebp+LockHandle]              ; LockHandle
mov     byte ptr [edi+2Eh], 0              ; 将EDI+0x2e(KACP.Inserted)设为0
call    ds:__imp_@KeReleaseInStackQueuedSpinLock@4 ; KeReleaseInStackQueuedSpinLock(x)

lea     eax, [ebp-8]                      ; 依次将SystemArgument2、SystemArgument1
; NormalContext、NormalRoutine压入栈内
push    eax
lea     eax, [ebp-0Ch]
push    eax
lea     eax, [ebp-10h]
push    eax
lea     eax, [ebp-4]
push    eax
push    edi
call    dword ptr [ebp-14h]                ; 调用KernelRoutine, 即释放APC

cmp     dword ptr [ebp-4], 0               ; 判断NormalRoutine的值是否为0
jz      short loc_41D3CF                  ; 为0则跳转

xor     cl, cl                             ; NewIrql
mov     byte ptr [esi+48h], 1              ; 将ESI+0x48(KAPC.KernelApcInProgress)设为1
call    ds:__imp_@KfLowerIrql@4           ; KfLowerIrql(x)

push    dword ptr [ebp-8]                  ; 压入SystemArgument2、SystemArgument1、NormalContext
push    dword ptr [ebp-0Ch]
push    dword ptr [ebp-10h]
call    dword ptr [ebp-4]                  ; 执行NormalRoutine, 即内核APC函数

```

当执行完内核APC函数时，则会将KernelApcInProgress设为0，然后再进行判断是否有下一个需要内核APC，如果有的话就重复上面的步骤。

```

mov     byte ptr [esi+48h], 0      ; 将ESI+0x48(KAPC.KernelApcInProgress)设为0
jmp     loc_4063A9

loc_4063A9:
; CODE XREF: IopCompleteRequest(x,x,x,x,x)+21D↓j
; IopCompleteRequest(x,x,x,x,x)+7156↓j
; .text:0044B9F5↓j
cmp     [ebx], ebx                ; 微软链表设计时，内容为空不会存空值，而会存当前链表的地址，即[ebx] == ebx
jnz     loc_416444                ; 因此使用CMP判断两者是否相等，如果不相等则表示队列中有APC
; 不相等即跳转

lea     ecx, [esi+3Ch]            ; 取第二个链表
mov     eax, [ecx]
cmp     eax, ecx
jnz     loc_410C2D

```



最后，我们可以简单总结一下：

1. 内核APC在线程切换的时候就会执行，这也就意味着，只要插入内核APC很快就会执行；
2. 在执行用户APC之前会先执行内核APC；
3. 内核APC在内核空间执行，不需要换栈，一个循环全部执行完毕。

## 5 用户APC执行过程

处理用户APC要比内核APC复杂的多，因为用户APC函数要在用户空间执行的，这里涉及到大量栈的切换操作：当线程从用户层进入内核层时，要保留原来的运行环境，比如各种寄存器，栈的位置等等（Trap\_Frame），然后切换成内核的栈，如果正常返回，恢复栈环境即可。

但如果有用户APC要执行的话，就意味着线程要**提前返回到用户空间**去执行，而且返回的位置不是线程进入内核时的位置，而是**返回到其他的位置**，每处理一个用户APC都会涉及到内核空间与用户空间的切换，因此栈的操作比较复杂，如果我们需要足够了解栈的操作细节才能理解用户APC是如何执行的。

### 5.1 KiDeliverApc函数

我们接着来看KiDeliverApc函数，该函数处理用户APC时，会先处理内核APC，然后回来取用户APC队列，判断队列是否为空，不为空则进行跳转处理。

```

mov     eax, large fs:124h          ; 将FS:0x124(CurrentThread[KTHREAD])给到EAX
mov     esi, eax                   ; EAX给到ESI
mov     eax, [esi+134h]
mov     [ebp+var_1C], eax
mov     eax, [esi+44h]
mov     [esi+134h], ecx
lea     ecx, [esi+0E8h]            ; SpinLock
lea     edx, [ebp+LockHandle]      ; LockHandle
mov     [ebp+BugCheckParameter1], eax
call    ds:___imp_@KeAcquireInStackQueuedSpinLock@8 ; KeAcquireInStackQueuedSpinLock(x,x)

mov     byte ptr [esi+49h], 0       ; 将ESI+0x49(KTHREAD.ApcState.KernelApcPending)设为0
lea     ebx, [esi+34h]             ; 将ESI+0x34(KTHREAD.ApcState.ApcListHead)，即内核APC队列地址给到EBX

loc_4063A9:                        ; CODE XREF: IopCompleteRequest(x,x,x,x,x)+21D↓j
                                        ; IopCompleteRequest(x,x,x,x,x)+7156↓j
                                        ; .text:0044B9F5↓j
cmp     [ebx], ebx                 ; 微软链表设计时，内容为空不会存空值，而会存当前链表的地址，即[ebx] == ebx
                                        ; 因此使用CMP判断两者是否相等，如果不相等则表示队列中有APC
jnz     loc_416444                 ; 不相等即跳转

lea     ecx, [esi+3Ch]             ; 取第二个链表，ESI+0x3c(KTHREAD.ApcState.ApcListHead + 0x8)，即用户内核队列
mov     eax, [ecx]
cmp     eax, ecx                   ; 判断队列（链表）是否为空
jnz     loc_410C2D                 ; 不为空则跳转

```

跳转过来之后，会进行两个判断，**第一个判断是KiDeliverApc函数的第一个参数是否为1**，为1则表示要处理用户APC，就不会跳转继续向下走；**第二个判断是UserApcPending是否为0**，不为0则表示当前有等待执行的用户APC，就不会跳转继续向下走；将UserApcPending设为0，然后接下来的操作就跟内核APC执行过程差不多了，**取KAPC结构体首地址**，将几个成员压入栈内便于后续使用，**再从链表中将当前APC摘除**，接着调用KernelRoutine来**释放当前APC所占用的内存空间**；在这之后就与内核APC执行过程不一样了，先判断NormalRoutine是否为0，不为0则表示用户APC的总入口是存在的，然后将SystemArgument2、SystemArgument1、NormalContext、NormalRoutine分别压入栈内压入栈内，**用于后续的KiInitializeUserApc函数的执行**。



```

loc_410C2D:                                ; CODE XREF: KiDeliverApc(x,x,x)+64↑j
cmp     [ebp+arg_0], 1                     ; 判断第一个参数是不是1
jnz     loc_4063BE                          ; 不是1则跳转，是1的话就会处理用户APC

cmp     byte ptr [esi+4Ah], 0               ; 判断ESI+0x4a(KTHREAD.ApcState.UserApcPending)是否为0
jz      loc_4063BE                          ; 是0就进行跳转，不是0就向下继续走

mov     byte ptr [esi+4Ah], 0               ; 将ESI+0x4a(KTHREAD.ApcState.UserApcPending)设为0
lea     edi, [eax-0Ch]                     ; 跟内核APC里的操作一样，取KAPC首地址
mov     ecx, [edi+1Ch]                     ; 接下来就是将KAPC里的几个成员分别压入栈内
mov     ebx, [edi+14h]
mov     [ebp+var_4], ecx
mov     ecx, [edi+20h]
mov     [ebp+var_10], ecx
mov     ecx, [edi+24h]
mov     [ebp+var_C], ecx
mov     ecx, [edi+28h]
mov     [ebp+var_8], ecx
mov     ecx, [eax]                          ; 链表操作，将当前APC从队列中摘除
mov     eax, [eax+4]
mov     [ecx], ecx
mov     [ecx+4], eax
lea     ecx, [ebp+LockHandle]               ; LockHandle
mov     byte ptr [edi+2Eh], 0
call    ds: __imp_@KeReleaseInStackQueuedSpinLock@4 ; KeReleaseInStackQueuedSpinLock(x)

lea     eax, [ebp+var_8]
push    eax
lea     eax, [ebp+var_C]
push    eax
lea     eax, [ebp+var_10]
push    eax
lea     eax, [ebp+var_4]
push    eax
push    edi
call    ebx                                ; 调用KAPC.KernelRoutine对应的函数，释放当前APC

cmp     [ebp+var_4], 0                     ; 判断NormalRoutine是否为0
jz      loc_44B9FA                          ; 为0则跳转，当前是用户APC，则NormalRoutine为用户APC的总入口

push    [ebp+var_8]                        ; 将SystemArgument2、SystemArgument1、NormalContext、NormalRoutine分别压入栈内，用于函数传参
push    [ebp+var_C]
push    [ebp+var_10]
push    [ebp+var_4]
push    [ebp+arg_8]
push    [ebp+arg_4]
call    _KiInitializeUserApc@24             ; KiInitializeUserApc(x,x,x,x,x,x)

```

因此在这里NormalRoutine并不是用户APC的执行函数，也没法直接去执行，因为当前环境在内核空间，如果直接执行了用户空间的APC函数，就可能存在安全问题，因此需要通过KiInitializeUserApc函数来返回用户空间执行，我们可以来分析一下这个函数。

## 5.2 KiInitializeUserApc函数

线程进0环时，原来的运行环境（寄存器栈顶等）保存到Trap\_Frame结构体中，如果要提前返回3环去处理用户APC，就必须修改Trap\_Frame结构体。比如进0环时的位置存储在EIP中，现在要提前返回，而且返回的并不是原来的位置，那就意味着必须修改EIP为新的返回位置，还需要修改为处理APC时需要的栈（ESP）。

那么处理完APC后，原来的值该怎么办呢？实际上在KiInitializeUserApc函数中，要做的第一件事情就是将原来Trap\_Frame的值备份到一个新的结构体中（\_Context），新的结构体与Trap\_Frame结构体大致一样，我们可以来看一下。

```

kd> dt _CONTEXT
nt!_CONTEXT
    +0x000 ContextFlags      : Uint4B
    +0x004 Dr0               : Uint4B
    +0x008 Dr1               : Uint4B
    +0x00c Dr2               : Uint4B
    +0x010 Dr3               : Uint4B
    +0x014 Dr6               : Uint4B
    +0x018 Dr7               : Uint4B
    +0x01c FloatSave        : _FLOATING_SAVE_AREA
    +0x08c SegGs             : Uint4B
    +0x090 SegFs             : Uint4B
    +0x094 SegEs             : Uint4B
    +0x098 SegDs             : Uint4B
    +0x09c Edi               : Uint4B
    +0x0a0 Esi               : Uint4B
    +0x0a4 Ebx               : Uint4B
    +0x0a8 Edx               : Uint4B
    +0x0ac Ecx               : Uint4B
    +0x0b0 Eax               : Uint4B
    +0x0b4 Ebp               : Uint4B
    +0x0b8 Eip               : Uint4B
    +0x0bc SegCs             : Uint4B
    +0x0c0 EFlags            : Uint4B
    +0x0c4 Esp               : Uint4B
    +0x0c8 SegSs             : Uint4B
    +0x0cc ExtendedRegisters : [512] UChar

```

我们来分析KiInitializeUserApc函数，就会发现备份功能是由其子函数KeContextFromKframes来完成的。

```

mov     eax, ds:___security_cookie
mov     [ebp+var_1C], eax
mov     eax, [ebp+arg_0]
mov     [ebp+var_2F4], eax
mov     ebx, [ebp+arg_4]           ; Trap_Frame, 源自传参, 该传参又是KiDeliverApc的传参
mov     [ebp+BugCheckParameter3], ebx
test    byte ptr [ebx+72h], 2
jnz     loc_410DE4

mov     [ebp+var_2E8], 10017h
lea     ecx, [ebp+var_2E8]         ; 获取CONTEXT结构体首地址
push    ecx                       ; 压入CONTEXT结构体首地址
push    eax
push    ebx                       ; 压入_Trap_Frame首地址
call    _KeContextFromKframes@12  ; KeContextFromKframes(x,x,x)

```

备份完成之后, 我们不可能一直让\_Context结构体存于当前的内核空间中, 如下图代码所示, 会将原3环的ESP按4字节对齐, 然后提升栈顶, **提升的高度正好是\_Context结构体加4个APC执行所需参数的大小**, 接着将提升后的ESP暂存至内核空间栈中, 最后**通过循环指令将\_Context结构体成员逐个复制到3环的栈中**。处理好\_Context结构体之后, 就可以对Trap\_Frame的值进行修改了。

```

and     dword ptr [ebp-4], 0
mov     eax, 2DCCh
mov     [ebp+var_2F8], eax
mov     esi, [ebp+var_224]         ; 已知EBP+0x2E8为CONTEXT结构体首地址, 2E8-224 = C4
                                         ; 这个位置就是结构体中ESP成员的位置, 即3环原来的栈顶
and     esi, 0FFFFFFCh           ; 将原来的栈顶地址, 最后2位按与运算, C->1100
                                         ; 因此不管结果如何都是0, 这是为了保证按4字节方式对齐
sub     esi, eax                 ; 直接需改3环的栈顶, 减去0x2DC, 提升栈顶
                                         ; 也就是CONTEXT结构体的大小(0x2CC)+运行APC必须用到的4个参数(0x10)
mov     [ebp+var_2EC], esi        ; 在0环栈中暂存修改后的3环的栈顶
push    4                        ; Alignment
push    eax                      ; Length
push    esi                      ; Address
call    _ProbeForWrite@12         ; ProbeForWrite(x,x,x)

lea     edi, [esi+10h]
mov     ecx, 0B3h                ; 下面3行, 将CONTEXT结构体复制到3环的栈中
lea     esi, [ebp+var_2E8]
rep movsd
mov     dword ptr [ebx+6Ch], 1Bh   ; 开始修改Trap_Frame的值, 将SegCs修改为0x1B
push    23h ; '#'
pop     eax
mov     [ebx+78h], eax            ; HardwareSegSs修改为0x23
mov     [ebx+38h], eax           ; SegDs修改为0x23
mov     [ebx+34h], eax           ; SegEs修改为0x23
mov     dword ptr [ebx+50h], 3Bh ; ';' ; SegFs修改为0x3B
and     dword ptr [ebx+30h], 0    ; SegGs修改为0
mov     ecx, [ebp+var_228]
test    ecx, 20000h
jnz     loc_44BF3C

```

我们接着向下看代码, 就会发现它会将Trap\_Frame.Eip修改为KiUserApcDispatcher函数地址, 也就是修改了返回3环时的地址, 然后将执行APC所需要的4个参数依次填充到3环的栈内, 这里的步骤我们就可以理解为下移栈顶然后填入内容, 由于当前是在0环空间因此只可以使用MOV+ADD指令来下移栈顶, 依次复制到栈中。

```

loc_410D93:                                ; CODE XREF: KiInitializeUserApc(x,x,x,x,x,x)+34B36↓j
mov     eax, [ebp+var_2EC]                  ; 将暂存的3环的栈顶取出
mov     [ebx+74h], eax                      ; 给到Trap_Frame.HardwareEsp, 也就是新的栈顶
mov     ecx, ds:_KeUserApcDispatcher        ; 取出ntdll._KeUserApcDispatcher的地址
mov     [ebx+68h], ecx                     ; 给到Trap_Frame.Eip, 也就是修改回3环时的地址
and     dword ptr [ebx+64h], 0
mov     ecx, [ebp+arg_8]                    ; 将NormalRoutine给到ECX
mov     [eax], ecx                          ; 再通过ECX给到3环栈顶位置处
push    4
pop     ecx
add     eax, ecx                            ; +4, 栈顶下移
mov     [ebp+var_2EC], eax                  ; 将下移后的3环栈顶继续保存在0环的栈中
mov     edx, [ebp+arg_C]                    ; 下面的操作就是将NormalContext、SystemArgument1、SystemArgument2
                                           ; 依次压入3环的栈中, 并依次下移栈顶

mov     [eax], edx
add     eax, ecx
mov     [ebp+var_2EC], eax
mov     edx, [ebp+arg_10]
mov     [eax], edx
add     eax, ecx
mov     [ebp+var_2EC], eax
mov     edx, [ebp+arg_14]
mov     [eax], edx
add     eax, ecx
mov     [ebp+var_2EC], eax                  ; 继续保存下移后的3环栈顶至0环的栈中

```

上述操作执行完成之后, 3环的栈分布大致如下:



准备会用户层需要的环境:

1、段寄存器: SS DS FS GS

2、修改EFLAGS寄存器

3、修改ESP

4、修改EIP

ntdll.KiUserApcDispatcher

然后当返回3环时, 就会跳转到KiUserApcDispatcher函数开始执行, 我们的传递的4个参数也是给它用的。

### 5.3 KiUserApcDispatcher函数

KiUserApcDispatcher函数是在ntdll.dll模块中，我们可以通过IDA来分析一下它是干啥用的。这个函数很简单，首先就是取NormalRoutine，然后调用对应的函数，接着再取\_Context结构体，给到ZwContinue用于顺利返回0环。

```

; __stdcall KiUserApcDispatcher(x, x, x, x, x)
public _KiUserApcDispatcher@20
_KiUserApcDispatcher@20 proc near

arg_C= byte ptr 10h

lea     edi, [esp+arg_C] ; 取CONTEXT结构体地址
pop     eax              ; 当前栈顶指向的就是NormalRoutine，将该地址弹给EAX
call    eax              ; 调用NormalRoutine
push    1
push    edi              ; 压入CONTEXT用于传参
call    _ZwContinue@8    ; 这是一个返回0环的步骤，来用CONTEXT顺利返回0环

```

我们知道用户APC时NormalRoutine表示的是用户APC函数的总入口，那么为什么直接调用就可以了呢，这是因为当用户在3环调用QueueUserApc函数来插入APC时，不需要提供NormalRoutine，这个参数是在QueueUserApc内部指定的BaseDispatchApc，通过这个入口，内部会调用真正的用户APC函数执行，**用户调用时只需要提供函数和参数即可。**

ZwContinue函数的意义就是返回内核，如果还有用户APC，重复上面的执行过程；如果没有需要执行的用户APC，会将\_Context结构体赋值给Trap\_Frame结构体，这样就像从来没有修改过一样，ZwContinue后面的代码不会执行，线程从哪里进0环仍然会从哪里回去。