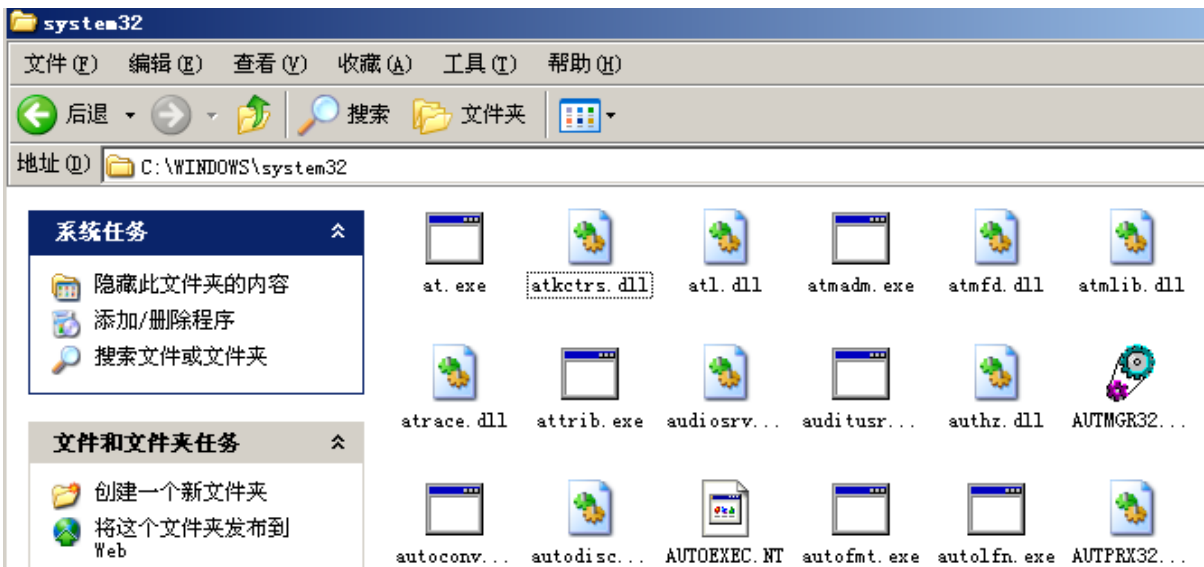


1 API函数的调用过程

1.1 3环部分

API函数即应用程序接口（Application Programming Interface），Windows API主要在C:\Windows\System32\目录下的所有DLL里。

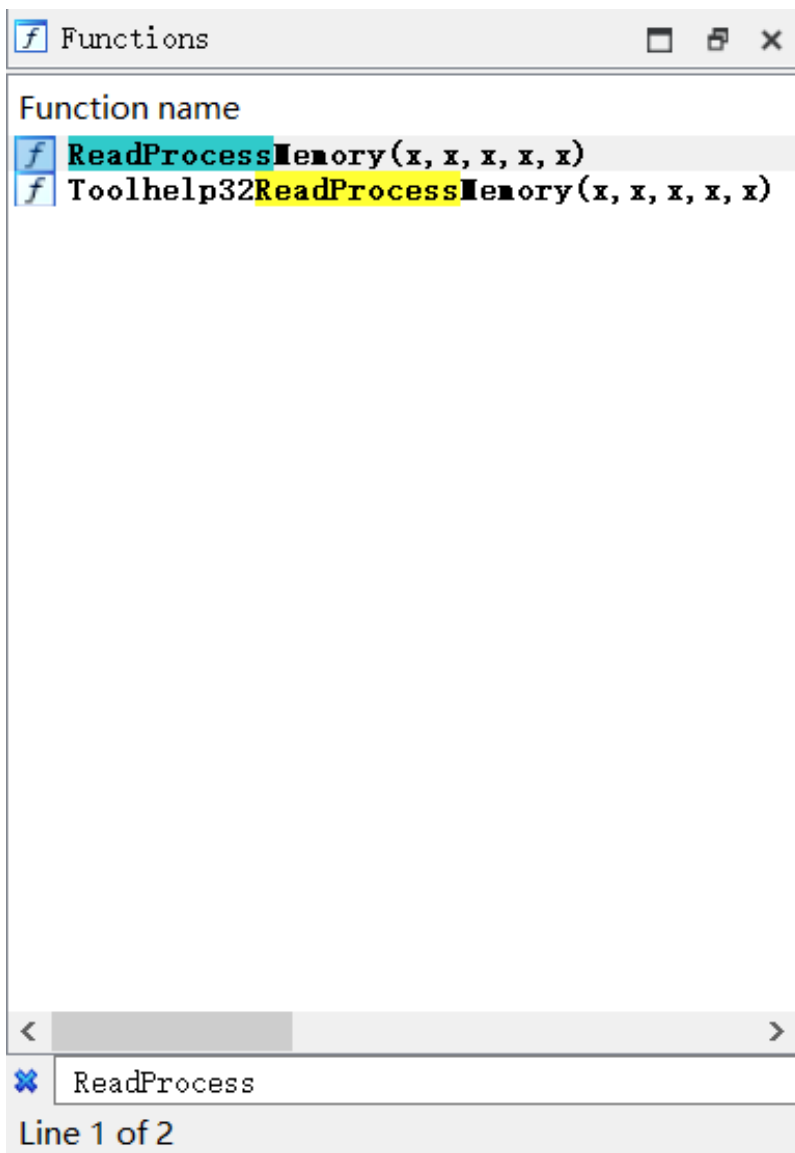


该目录下有几个重要的DLL：

DLL名称	作用
kernel32.dll	最核心的功能模块，如管理内存、进程和线程
user32.dll	Windows用户界面相关API，如创建窗口、发送消息
gdi32.dll	图形设备接口，如画图、显示文本
ntdll.dll	大多数API通过该DLL进入内核（0环）

1.1.1 分析ReadProcessMemory函数

我们可以通过IDA来看一下ReadProcessMemory在DLL文件中的体现。找到kernel32.dll文件，用IDA打开，在Functions窗口使用Ctrl+F快捷键搜索函数：



双击函数名，在IDA的反汇编窗口我们可以看到该函数的反汇编指令：

```

; Exported entry 679. ReadProcessMemory

; Attributes: bp-based frame

; BOOL __stdcall ReadProcessMemory(HANDLE hProcess, LPCVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize, SIZE_T *lpNumberOfBytesRead)
public _ReadProcessMemory@20
_ReadProcessMemory@20 proc near

hProcess= dword ptr 8
lpBaseAddress= dword ptr 0Ch
lpBuffer= dword ptr 10h
nSize= dword ptr 14h
lpNumberOfBytesRead= dword ptr 18h

mov     edi, edi
push    ebp
mov     ebp, esp
lea     eax, [ebp+nSize]
push    eax                ; NumberOfBytesRead
push    [ebp+nSize]        ; NumberOfBytesToRead
push    [ebp+lpBuffer]     ; Buffer
push    [ebp+lpBaseAddress] ; BaseAddress
push    [ebp+hProcess]     ; ProcessHandle
call    ds: _imp_NtReadVirtualMemory@20 ; NtReadVirtualMemory(x,x,x,x,x)
mov     ecx, [ebp+lpNumberOfBytesRead]
test    ecx, ecx
jnz     short loc_7C8021F9

```

①
参数压栈，调用导入表的NtReadVirtualMemory函数

```

loc_7C8021F9:
mov     edx, [ebp+nSize]
mov     [ecx], edx
jmp     short loc_7C8021EE

```

```

loc_7C8021EE:
test    eax, eax
jle     short loc_7C802200

```

②
判断返回值是否小于0，小于则跳转到loc_7C802200
不小于则直接设置返回值

```

xor     eax, eax
inc     eax

```

③
将返回值设为1

```

loc_7C802200:                ; Status
push    eax
call    _BaseSetLastNTError@4 ; BaseSetLastNTError(x)
xor     eax, eax
jmp     short loc_7C8021F5
_ReadProcessMemory@20 endp

```

③
设置错误代码以供
GetLastError函数使用
并将返回值设为0

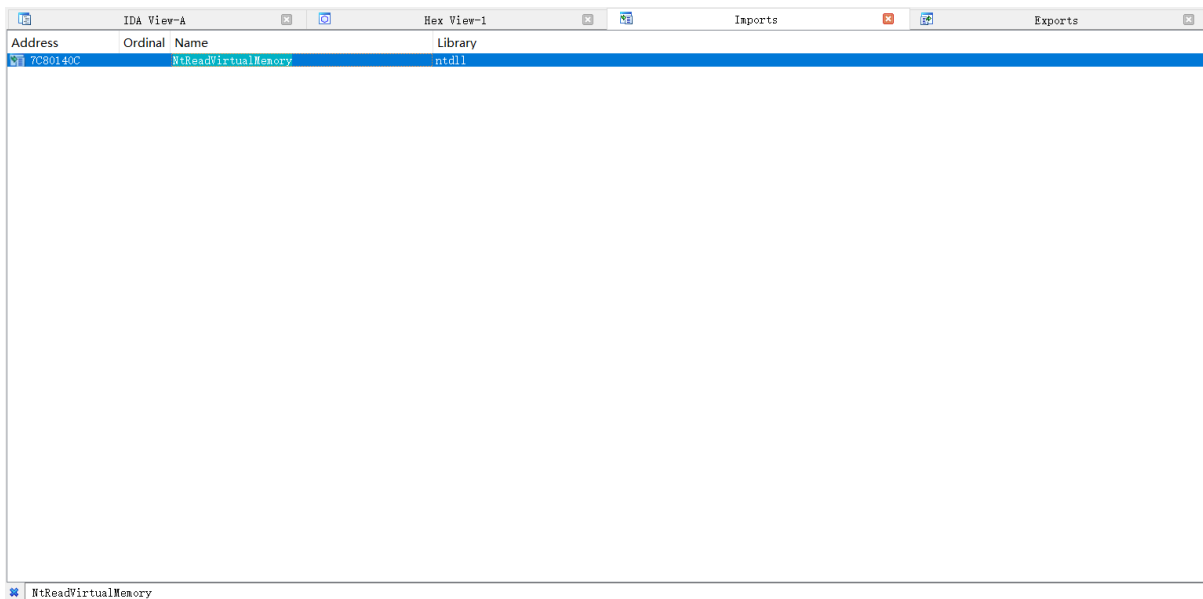
```

loc_7C8021F5:
pop     ebp
ret     14h

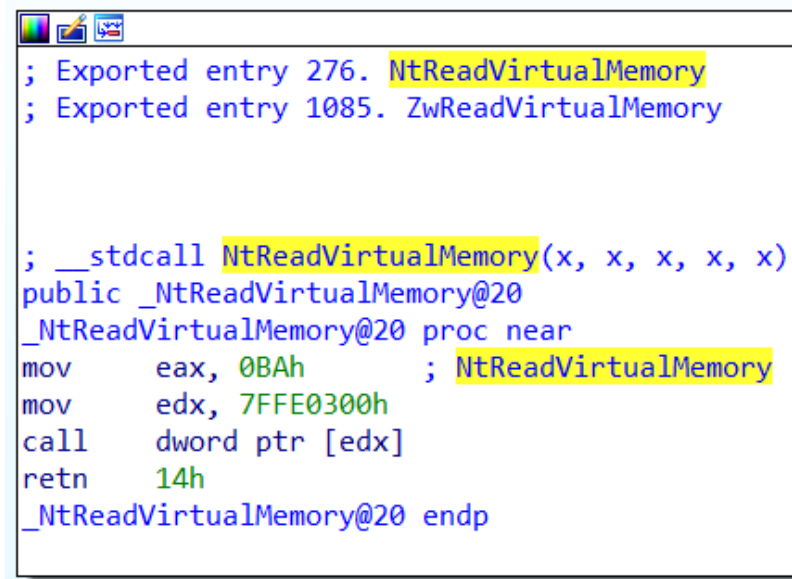
```

④
平栈，返回

我们可以看见它的主要作用就是调用导入表的NtReadVirtualMemory函数，我们可以在IDA的Imports窗口里找到该函数（Ctrl+F搜索），并且能知道它来自ntdll.dll模块：



我们接着使用IDA打开ntdll.dll来看一下该函数，它给了EAX一个值（我们可以理解为是一个编号，也可以称之为系统调用号），并且给了EDX一个内存地址，该内存地址就是一个函数，也就是进入0环的关键：



所以至此我们也可以得出结论，大部分Windows API的真正实现是在0环的，3环只是一个封装调用。

1.1.2 实现ReadProcessMemory函数

我们已经知道Windows API在3环的表现形式，所以我们可以自己不调用DLL的情况下写一个自己的ReadProcessMemory函数。

我们可以直接按照IDA将kernel32和ntdll的反汇编代码整合一下，如下代码，其中提升栈顶的SUB指令是因为在Windows API本身的调用中是有两次CALL指令的，第一次CALL指令会将栈顶提升0x4，并且存入下一行指令地址，所以在最后一次的CALL调用时，我们要保证hProcess在ESP+4的位置，而不是ESP的位置，在最后的汇编指令中我们也要手动恢复到原来的栈顶位置：

```

1  void MyReadProcessMemory(HANDLE hProcess, LPCVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize,
2  SIZE_T *lpNumberOfBytesRead)
3  {
4      _asm
5      {
6          // ReadProcessMemory
7          lea eax, [ebp+0x14]
8          push eax
9          push [ebp+0x14]
10         push [ebp+0x10]
11         push [ebp+0xC]
12         push [ebp+0x8]
13
14         // NtReadVirtualMemory
15         sub esp, 0x4
16         mov eax, 0xBA
17         mov edx, 0x7FFE0300
18         call dword ptr [edx]
19         add esp, 0x18
20     }
}

```

在代码中去调用，成功执行，结果如下：

```

int main(int argc, char* argv[])
{
    int a = 0x123;
    int buffer = 0;
    MyReadProcessMemory(GetCurrentProcess(), &a, &buffer, 4, NULL);
    printf("%x \n", buffer);

    return 0;
}

```



1.2 3环进0环部分

我们已经了解API在3环的体现，并且可以手动的去封装一个在3环的应用API。但我们并不知道3环进0环的具体细节，本章节就来一探究竟。

1.2.1 _KUSER_SHARED_DATA

在应用和内核层分别定义了一个_KUSER_SHARED_DATA结构体，用于应用和内核间的一些数据共享。它们使用的是固定的地址值映射，该结构区域在用户和内核层的地址分别为：0x7FFE0000、0xFFDF0000。从地址来看，**它们不是一个线性地址，但本质上指向的都是同一个物理页**，在用户层对该物理页有只读权限，但在内核层有读、写权限。

在上一章节中我们看到Windows API使用的地址为0x7FFE0300，它就属于_KUSER_SHARED_DATA结构体的一部分，我们可以在Windbg中查看该结构体找到0x300偏移的成员：

```

0: kd> dt _KUSER_SHARED_DATA
nt!_KUSER_SHARED_DATA
+0x000 TickCountLow      : Uint4B
+0x004 TickCountMultiplier : Uint4B
+0x008 InterruptTime     : _KSYSTEM_TIME
+0x014 SystemTime        : _KSYSTEM_TIME
+0x020 TimeZoneBias      : _KSYSTEM_TIME
+0x02c ImageNumberLow     : Uint2B
+0x02e ImageNumberHigh    : Uint2B
+0x030 NtSystemRoot       : [260] Uint2B
+0x238 MaxStackTraceDepth : Uint4B
+0x23c CryptoExponent     : Uint4B
+0x240 TimeZoneId        : Uint4B
+0x244 Reserved2         : [8] Uint4B
+0x264 NtProductType      : _NT_PRODUCT_TYPE
+0x268 ProductTypeIsValid : UChar
+0x26c NtMajorVersion     : Uint4B
+0x270 NtMinorVersion     : Uint4B
+0x274 ProcessorFeatures  : [64] UChar
+0x2b4 Reserved1         : Uint4B
+0x2b8 Reserved3         : Uint4B
+0x2bc TimeSlip          : Uint4B
+0x2c0 AlternativeArchitecture : _ALTERNATIVE_ARCHITECTURE_TYPE
+0x2c8 SystemExpirationDate : _LARGE_INTEGER
+0x2d0 SuiteMask         : Uint4B
+0x2d4 KdDebuggerEnabled  : UChar
+0x2d5 NXSupportPolicy    : UChar
+0x2d8 ActiveConsoleId    : Uint4B
+0x2dc DismountCount      : Uint4B
+0x2e0 ComPlusPackage     : Uint4B
+0x2e4 LastSystemRITEventTickCount : Uint4B
+0x2e8 NumberOfPhysicalPages : Uint4B
+0x2ec SafeBootMode       : UChar
+0x2f0 TraceLogging       : Uint4B
+0x2f8 TestRetInstruction : Uint8B
+0x300 SystemCall         : Uint4B
+0x304 SystemCallReturn   : Uint4B

```

SystemCall

0x7FFE0300就是_KUSER_SHARED_DATA结构体的成员SystemCall，它的作用就是进入0环，而具体进入0环的方式则由CPU决定，当CPU支持SYSENTER、SYSEXIT指令则使用的是ntdll!KiFastSystemCall方式，否则则是ntdll!KiIntSystemCall方式。

如果我们自己想要判断是什么方式，可以选择使用CPUID指令，当EAX=1时指向CPUID指令，处理器的特征信息就会存放在ECX和EDX寄存器中，其中EDX包含了一个SEP位（第11位），该位为1则表示CPU支持SYSENTER、SYSEXIT指令，为0则表示不支持。

我们可以在OD里执行一下CPUID指令：

地址	指令	寄存器 (FPU)
0045403D	\$ 0FA2 cpuid	EAX 00000001
0045403F	- 90 nop	ECX 00000000
00454040	- 90 nop	EDX 00000000
00454041	- 90 nop	EBX 7FFDE000
00454042	^ E9 17FEFFFF jmp Udd_Clea.00454B5E	

执行之后我们可以看下EDX的第11位，它确实为1，也就表示CPU支持SYSENTER、SYSEXIT指令：

地址	指令	寄存器 (FPU)
0045403D	\$ 0FA2 cpuid	EAX 00000001
0045403F	- 90 nop	ECX F7FA3203
00454040	- 90 nop	EDX 00000001
00454041	- 90 nop	EBX 02010800
00454042	^ E9 17FEFFFF jmp Udd_Clea.00454B5E	

所以在当前CPU下，_KUSER_SHARED_DATA结构体的0x300偏移位存储的就是ntdll!KiFastSystemCall函数的地址。

1.2.2 进入0环的方式

在之前各种“门”的学习后，我们知道当进行权限切换时（3环进0环），CS、SS、ESP、EIP寄存器都会发生变化。有了这些前置的知识了解，我们来看一下KiIntSystemCall和KiFastSystemCall这两个进入0环的方式。

KiIntSystemCall

我们可以通过Windbg打开任意一个PE文件，然后找到KiIntSystemCall函数：

```
0:000> u KiIntSystemCall
ntdll!KiIntSystemCall:
770650a0 8d542408      lea     edx,[esp+8]
770650a4 cd2e        int     2Eh
770650a6 c3           ret
770650a7 cc          int     3
770650a8 cc          int     3
770650a9 cc          int     3
770650aa cc          int     3
770650ab cc          int     3
```

我们发现它的代码很简单，首先将当前参数的指针存储到EDX寄存器中（系统调用号在EAX寄存器中），然后使用中断门进入0环（所有的API使用中断门进内核时，使用的中断号（IDT表中索引值）都是0x2E）。

调用过程分析

我们来手动分析一下INT 0x2E进0环的过程，首先找到IDT表，然后根据0x2E索引找到对应的门描述符位置：

```

0: kd> r idtr
idtr=8003f400
0: kd> dq 8003f400+(0x2E*0x8)
8003f570 8054ee00`00081441 80548e00`000847d0
8003f580 80548e00`00080b00 80548e00`00080b0a
8003f590 80548e00`00080b14 80548e00`00080b1e
8003f5a0 80548e00`00080b28 80548e00`00080b32
8003f5b0 80548e00`00080b3c 806e8e00`00084864
8003f5c0 80548e00`00080b50 80548e00`00080b5a
8003f5d0 80548e00`00080b64 80548e00`00080b6e
8003f5e0 80548e00`00080b78 806e8e00`00085e2c

```

该门描述符的高32位的第8位为0，也就表示该描述符为中断门描述符。通过中断门描述符我们知道了CS（低32位的第16至31位，也就是段选择子）和EIP寄存器（高32位的第16至31位加上低32位的第0至15位，也就是段内偏移），SS和ESP寄存器也就是基于TSS获取的。

我们跟进EIP，就会发现它是一个内核模块的函数KiSystemService，至此我们就大致了解了中断门进0环的简单流程：

```

0: kd> u 80541441
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

```

KiFastSystemCall

如下是KiFastSystemCall函数，我们可以看到它跟KiIntSystemCall没睡区别，只是它进入0环的方式不是基于中断门，而是使用了SYSENTER指令：

```

0:000> u KiFastSystemCall
ntdll!KiFastSystemCall:
773b5080 8bd4          mov      edx,esp
773b5082 0f34          sysenter
773b5084 8da424000000 lea      esp,[esp]
773b508b eb03          jmp      ntdll!KiFastSystemCallRet (773b5090)
773b508d cc           int      3
773b508e cc           int      3
773b508f cc           int      3
ntdll!KiFastSystemCallRet:
773b5090 c3           ret

```


我们都知道中断门进0环需要提供新的CS、SS、ESP、EIP，CS和EIP在IDT表中，而SS和ESP都在TSS中，这就需要去查内存了，速度相对来说会很慢。所以在这里CPU提供了SYSENTER指令，在执行该指令前操作系统会提前将CS、SS、ESP、EIP的值存储在MSR寄存器中，**该指令执行时CPU会将MSR寄存器中的值直接写入到相关寄存器，就减去了读取内存的过程，提升了调用的速度，因此我们可以称之为快速调用。**

调用过程分析

在MSR寄存器中有三个值即CS段选择子、ESP和EIP寄存器，当我们找到CS选择子时候，将其地址加0x8即可获得SS选择子。当然这些操作都是硬件去完成的，具体的细节可以参考Intel白皮书第二卷中SYSENTER指令的内容。

MSR	地址
IA32_SYSENTER_CS	0x174
IA32_SYSENTER_ESP	0x175
IA32_SYSENTER_EIP	0x176

在操作系统上，我们可以通过RDMSR/WRMST来对MSR寄存器进行读写：

```
0: kd> rdmsr 0x174
msr[174] = 00000000`00000008
0: kd> rdmsr 0x175
msr[175] = 00000000`bacd0000
0: kd> rdmsr 0x176
msr[176] = 00000000`80541510
```

并且我们可以跟进EIP，你就会发现SYSENTER指令进入0环执行的就是内核模块的KiFastCallEntry函数：

```
0: kd> u 80541510
nt!KiFastCallEntry:
80541510 b923000000      mov     ecx,23h
80541515 6a30                push    30h
80541517 0fa1                pop     fs
80541519 8ed9                mov     ds,cx
8054151b 8ec1                mov     es,cx
8054151d 648b0d40000000      mov     ecx,dword ptr fs:[40h]
80541524 8b6104                mov     esp,dword ptr [ecx+4]
80541527 6a23                push    23h
```

总结

API通过中断门进入0环有一个固定的中断号即0x2E，CS、EIP由中断门描述符提供，ESP、SS由TSS提供，进入0环之后执行的内核函数是nt!KiSystemService。

API通过SYSENTER指令进入0环，CS、ESP、EIP由MSR寄存器提供，SS由CS的值加0x8计算得出，进入0环之后执行的内核函数是nt!KiFastCallEntry。

内核函数所在模块为：ntoskrnl.exe（10-10-12分页模式下使用）、ntkrnlpa.exe（2-9-9-12分页模式下使用）。

1.3 保存现场

API进入0环之前是需要将在3环时的寄存器值保存，这样在0环执行切换到新的寄存器执行完成后回到3环时才能恢复原先的寄存器还原现场，那么3环寄存器的值是保存在哪里的，就是本章我们需要学习的保存现场。

1.3.1 结构体

在正式的学习之前我们需要先了解这三个结构体：`_Trap_Frame`，`_ETHREAD`，`_KPCR`。

`_Trap_Frame`

在Windbg中我们可以使用如下命令查看`_Trap_Frame`结构体：

```
1 dt _KTrap_Frame
```

结构体中的成员对应的作用见下图中的文字注释，**该结构体就是保存现场所需要用到的结构体**：

```
0: kd> dt _KTrap_Frame
nt!_KTRAP_FRAME
```

+0x000	DbgEbp	: UInt4B	
+0x004	DbgEip	: UInt4B	
+0x008	DbgArgMark	: UInt4B	
+0x00c	DbgArgPointer	: UInt4B	
+0x010	TempSegCs	: UInt4B	
+0x014	TempEsp	: UInt4B	
+0x018	Dr0	: UInt4B	
+0x01c	Dr1	: UInt4B	
+0x020	Dr2	: UInt4B	① 调试等其他作用
+0x024	Dr3	: UInt4B	
+0x028	Dr6	: UInt4B	
+0x02c	Dr7	: UInt4B	
+0x030	SegGs	: UInt4B	
+0x034	SegEs	: UInt4B	
+0x038	SegDs	: UInt4B	
+0x03c	Edx	: UInt4B	
+0x040	Ecx	: UInt4B	② Windows中非易失性寄存器 需要在中断时保存起来
+0x044	Eax	: UInt4B	
+0x048	PreviousPreviousMode	: UInt4B	
+0x04c	ExceptionList	: Ptr32 _EXCEPTION_REGISTRATION_RECORD	
+0x050	SegFs	: UInt4B	
+0x054	Edi	: UInt4B	
+0x058	Esi	: UInt4B	
+0x05c	Ebx	: UInt4B	
+0x060	Ebp	: UInt4B	
+0x064	ErrCode	: UInt4B	
+0x068	Eip	: UInt4B	③ 中断发生时，保存被中断的代码段和地址及EFlags寄存器。
+0x06c	SegCs	: UInt4B	
+0x070	EFlags	: UInt4B	
+0x074	HardwareEsp	: UInt4B	④ 中断发生时，若权限发生切换 则需要保存旧的堆栈。
+0x078	HardwareSegSs	: UInt4B	
+0x07c	V86Es	: UInt4B	⑤ 保护模式不使用，当处于 8086模式需要存储这几个寄存器的值
+0x080	V86Ds	: UInt4B	
+0x084	V86Fs	: UInt4B	
+0x088	V86Gs	: UInt4B	

_ETHREAD

与线程相关的结构体就是_ETHEREAD，在Windows内核中每一个进程的每一个线程都有着一个_ETHEREAD结构体，它存储着线程相关的信息，我们可以使用如下命令在Windbg中查看该结构体：

```
1 dt _ETHEREAD
```

```

0: kd> dt _ETHREAD
nt!_ETHREAD
+0x000 Tcb : _KTHREAD
+0x1c0 CreateTime : _LARGE_INTEGER
+0x1c0 NestedFaultCount : Pos 0, 2 Bits
+0x1c0 ApcNeeded : Pos 2, 1 Bit
+0x1c8 ExitTime : _LARGE_INTEGER
+0x1c8 LpcReplyChain : _LIST_ENTRY
+0x1c8 KeyedWaitChain : _LIST_ENTRY
+0x1d0 ExitStatus : Int4B
+0x1d0 OfsChain : Ptr32 Void
+0x1d4 PostBlockList : _LIST_ENTRY
+0x1dc TerminationPort : Ptr32 _TERMINATION_PORT
+0x1dc ReaperLink : Ptr32 _ETHREAD
+0x1dc KeyedWaitValue : Ptr32 Void
+0x1e0 ActiveTimerListLock : Uint4B
+0x1e4 ActiveTimerListHead : _LIST_ENTRY
+0x1ec Cid : _CLIENT_ID
+0x1f4 LpcReplySemaphore : _KSEMAPHORE
+0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
+0x208 LpcReplyMessage : Ptr32 Void
+0x208 LpcWaitingOnPort : Ptr32 Void
+0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
+0x210 IrpList : _LIST_ENTRY
+0x218 TopLevelIrp : Uint4B
+0x21c DeviceToVerify : Ptr32 _DEVICE_OBJECT
+0x220 ThreadsProcess : Ptr32 _EPROCESS
+0x224 StartAddress : Ptr32 Void
+0x228 Win32StartAddress : Ptr32 Void
+0x228 LpcReceivedMessageId : Uint4B
+0x22c ThreadListEntry : _LIST_ENTRY
+0x234 RundownProtect : _EX_RUNDOWN_REF
+0x238 ThreadLock : _EX_PUSH_LOCK
+0x23c LpcReplyMessageId : Uint4B
+0x240 ReadClusterSize : Uint4B
+0x244 GrantedAccess : Uint4B
+0x248 CrossThreadFlags : Uint4B
+0x248 Terminated : Pos 0, 1 Bit
+0x248 DeadThread : Pos 1, 1 Bit
+0x248 HideFromDebugger : Pos 2, 1 Bit
+0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
+0x248 SystemThread : Pos 4, 1 Bit
+0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
+0x248 BreakOnTermination : Pos 6, 1 Bit
+0x248 SkipCreationMsg : Pos 7, 1 Bit
+0x248 SkipTerminationMsg : Pos 8, 1 Bit
+0x24c SameThreadPassiveFlags : Uint4B
+0x24c ActiveExWorker : Pos 0, 1 Bit
+0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
+0x24c MemoryMaker : Pos 2, 1 Bit
+0x250 SameThreadApcFlags : Uint4B
+0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
+0x250 LpcExitThreadCalled : Pos 1, 1 Bit
+0x250 AddressSpaceOwner : Pos 2, 1 Bit
+0x254 ForwardClusterOnly : UChar
+0x255 DisablePageFaultClustering : UChar

```

它的第一个成员也是一个结构体_KTHREAD，我们同样可以在Windbg中查看该结构体：

```

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
+0x04c ContextSwitches : Uint4B
+0x050 IdleSwapBlock   : Uchar
+0x051 Spare0         : [3] Uchar
+0x054 WaitStatus      : Int4B
+0x058 WaitIrql       : Uchar
+0x059 WaitMode        : Char
+0x05a WaitNext        : Uchar
+0x05b WaitReason      : Uchar
+0x05c WaitBlockList   : Ptr32 _KWAIT_BLOCK
+0x060 WaitListEntry   : _LIST_ENTRY
+0x060 SwapListEntry   : _SINGLE_LIST_ENTRY
+0x068 WaitTime        : Uint4B
+0x06c BasePriority     : Char
+0x06d DecrementCount  : Uchar
+0x06e PriorityDecrement : Char
+0x06f Quantum        : Char
+0x070 WaitBlock       : [4] _KWAIT_BLOCK
+0x0d0 LegoData        : Ptr32 Void
+0x0d4 KernelApcDisable : Uint4B
+0x0d8 UserAffinity    : Uint4B
+0x0dc SystemAffinityActive : Uchar
+0x0dd PowerState      : Uchar
+0x0de NpxIrql        : Uchar
+0x0df InitialNode     : Uchar
+0x0e0 ServiceTable    : Ptr32 Void
+0x0e4 Queue           : Ptr32 _KQUEUE
+0x0e8 ApcQueueLock    : Uint4B
+0x0f0 Timer           : _KTIMER
+0x118 QueueListEntry  : _LIST_ENTRY
+0x120 SoftAffinity    : Uint4B
+0x124 Affinity        : Uint4B
+0x128 Preempted       : Uchar
+0x129 ProcessReadyQueue : Uchar
+0x12a KernelStackResident : Uchar
+0x12b NextProcessor   : Uchar
+0x12c CallbackStack   : Ptr32 Void
+0x130 Win32Thread     : Ptr32 Void
+0x134 TrapFrame       : Ptr32 _KTRAP_FRAME
+0x138 ApcStatePointer : [2] Ptr32 _KAPC_STATE
+0x140 PreviousMode    : Char
+0x141 EnableStackSwap : Uchar
+0x142 LargeStack      : Uchar
+0x143 ResourceIndex   : Uchar
+0x144 KernelTime      : Uint4B
+0x148 UserTime        : Uint4B
+0x14c SavedApcState   : _KAPC_STATE
+0x164 Alertable       : Uchar
+0x165 ApcStateIndex   : Uchar
+0x166 ApcQueueable    : Uchar
+0x167 AutoAlignment   : Uchar
+0x168 StackBase       : Ptr32 Void
+0x16c SuspendApc      : _KAPC
+0x19c SuspendSemaphore : _KSEMAPHORE
+0x1b0 ThreadListEntry : _LIST_ENTRY
+0x1b8 FreezeCount     : Char
+0x1b9 SuspendCount    : Char
+0x1ba IdealProcessor   : Uchar
+0x1bb DisableBoost    : Uchar

```

_KPCR

每个CPU都有自己的控制区，这块控制区我们称之为KPCR。你的CPU有几核就有几个对应的KPCR。

我们可以通过如下指令在Windbg中查看KPCR相关的信息：

```

1 dt _KPCR // 查看KPCR结构体
2 dd KeNumberProcessors // 查看KPCR数量
3 dd KiProcessorBlock // 查看KPCR位置

```

0: kd> dt _KPCR

nt!_KPCR

结构体

```
+0x000 NtTib          : _NT_TIB
+0x01c SelfPcr        : Ptr32 _KPCR
+0x020 Prcb           : Ptr32 _KPRCB
+0x024 Irql           : UChar
+0x028 IRR            : Uint4B
+0x02c IrrActive       : Uint4B
+0x030 IDR            : Uint4B
+0x034 KdVersionBlock : Ptr32 Void
+0x038 IDT            : Ptr32 _KIDTENTRY
+0x03c GDT            : Ptr32 _KGDTENTRY
+0x040 TSS            : Ptr32 _KTSS
+0x044 MajorVersion   : Uint2B
+0x046 MinorVersion   : Uint2B
+0x048 SetMember      : Uint4B
+0x04c StallScaleFactor : Uint4B
+0x050 DebugActive     : UChar
+0x051 Number         : UChar
+0x052 Spare0         : UChar
+0x053 SecondLevelCacheAssociativity : UChar
+0x054 VdmAlert        : Uint4B
+0x058 KernelReserved : [14] Uint4B
+0x090 SecondLevelCacheSize : Uint4B
+0x094 HalReserved    : [16] Uint4B
+0x0d4 InterruptMode  : Uint4B
+0x0d8 Spare1         : UChar
+0x0dc KernelReserved2 : [17] Uint4B
+0x120 PrcbData       : KPRCB
```

0: kd> dd KeNumberProcessors

```
80555a60 00000002 00000006 00008d01 a0013fff
80555a70 806ccce0 00000000 00000000 0000005d
80555a80 8055f113 00000006 00000000 00000000
80555a90 00000001 00000000 00000001 00000000
80555aa0 00000000 00000000 00000000 00000000
80555ab0 00000000 00000000 00000000 00000000
80555ac0 00000000 00000000 00000000 00000000
80555ad0 00000000 00000000 00000000 00000000
```

0: kd> dd KiProcessorBlock

```
8055c580 ffdff120 bab38120 2个KPCR对应的位置
8055c590 00000000 00000000 00000000 00000000
```

通过KiProcessorBlock我们查看到的KPCR位置实际上是指向KPCR结构体0x120的偏移位，所以如果我们想完整的去查看这个结构体就需要在这个地址上减去0x120：

```

0: kd> dt _KPCR 0xffdff120-0x120
nt!_KPCR
+0x000 NtTib : _NT_TIB
+0x01c SelfPcr : 0xffdff000 _KPCR
+0x020 Prcb : 0xffdff120 _KPRCB
+0x024 Irql : 0 ''
+0x028 IRR : 0
+0x02c IrrActive : 0
+0x030 IDR : 0xffffffff
+0x034 KdVersionBlock : 0x8054d2b8 Void
+0x038 IDT : 0x8003f400 _KIDENTENTRY
+0x03c GDT : 0x8003f000 _KGDTENTRY
+0x040 TSS : 0x80042000 _KTSS
+0x044 MajorVersion : 1
+0x046 MinorVersion : 1
+0x048 SetMember : 1
+0x04c StallScaleFactor : 0x900
+0x050 DebugActive : 0 ''
+0x051 Number : 0 ''
+0x052 Spare0 : 0 ''
+0x053 SecondLevelCacheAssociativity : 0 ''
+0x054 VdmAlert : 0
+0x058 KernelReserved : [14] 0
+0x090 SecondLevelCacheSize : 0
+0x094 HalReserved : [16] 0
+0x0d4 InterruptMode : 0
+0x0d8 Spare1 : 0 ''
+0x0dc KernelReserved2 : [17] 0
+0x120 PrcbData : _KPRCB

```

它的0x120偏移位指向的是另外一个结构体_KPRCB，我们可以理解为它是一个扩展的KPCR。

1.3.2 KiSystemService保存现场

在了解完基本的结构体之后，我们可以手动来分析一下在0环函数KiSystemService，看一下它到底是如何保存现场的。**该函数保存现场的过程实际上就是填充Trap_Frame结构体的过程。**

我们可以通过IDA打开C:/Windows/System32/ntoskrnl.exe文件，在函数列表中找到_KiSystemService函数：


```

_KiSystemService proc near
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
.text:00406D36
    arg_0= dword ptr 4
    push 0
    push ebp
    push ebx
    push esi
    push edi
    push fs
    mov     ebx, 30h ; '0'
    mov     fs, bx
    push    large dword ptr fs:0
    mov     large dword ptr fs:0, 0FFFFFFFFh

```

我们知道在中断门进入0环时会先压入SS、ESP、EFlags、CS、EIP，也就是将这些值填充到**Trap_Frame**结构体，所以在_KiSystemService函数执行第一行指令之前是在**Trap_Frame**结构体的0x68偏移位，但是某些中断情况下如缺页异常就会除了这5个值以外压入第6个值，即**Trap_Frame**结构体中的ErrCode，所以为了使得堆栈平衡，在_KiSystemService函数第一行指令就压入了一个0x0来对齐，这时候我们就处于0x64偏移位，接着就是依次压入EBP、EBX、ESI、EDI、FS。

6A 00	push	0	;	_KTrap_Frame	+0x064	ErrCode	保持对齐
55	push	ebp	;	_KTrap_Frame	+0x060	EBP	
53	push	ebx	;	_KTrap_Frame	+0x05C	EBX	
56	push	esi	;	_KTrap_Frame	+0x058	ESI	
57	push	edi	;	_KTrap_Frame	+0x054	EDI	
0F A0	push	fs	;	_KTrap_Frame	+0x050	SegFs	

继续读代码，先是向EBX存入一个0x30，接着以0x30为段选择子在GDT表中找到段描述符加载到FS段寄存器中。

BB 30 00 00 00	mov	ebx, 30h ; '0'	; 将0x30赋值给EBX
66 8E E3	mov	fs, bx	; 加载段描述符至FS段寄存器

我们可以手动来看一下加载的段描述符到底是谁，首先拆分段选择子取第3至第15位作为索引，接着按索引在GDT表中找到段描述符。

```
1 // 0x30 拆分
2 0000 0000 0011 0000
3 索引: 0110 -> 0x6
4 段描述符: ffc093df`f0000001
```



```

0: kd> r gdt
gdt=8003f000
0: kd> dq 0x8003f000+(0x6*0x8)
8003f030 ffc093df`f0000001 0040f300`00000fff
8003f040 0000f200`0400ffff 00000000`00000000
8003f050 80008955`17000068 80008955`17680068
8003f060 00009302`2f30ffff 0000920b`80003fff
8003f070 ff0092ff`700003ff 80009a40`0000ffff
8003f080 80009240`0000ffff 00009200`00000000
8003f090 00000000`00000000 00000000`00000000
8003f0a0 890089e2`80f00068 00000000`00000000

```

接着我们根据段描述符的Base，即0xffdf000，它指向的就是当前CPU的KPCR结构：

```

0: kd> dt _KPCR 0xffdf000
nt!_KPCR
+0x000 NtTib           : _NT_TIB
+0x01c SelfPcr         : 0xffdf000 _KPCR
+0x020 Prcb           : 0xffdf120 _KPRCB
+0x024 Irql           : 0 ''
+0x028 IRR            : 0
+0x02c IrrActive       : 0
+0x030 IDR            : 0xffffffff
+0x034 KdVersionBlock : 0x8054d2b8 Void
+0x038 IDT             : 0x8003f400 _KIDENTENTRY
+0x03c GDT             : 0x8003f000 _KGDTENTRY
+0x040 TSS             : 0x80042000 _KTSS

```

然后就是压入FS段的0x0偏移位的内容，也就是指向的KPCR结构0x0偏移位成员_NT_TIB->ExceptionList，接着再将当前的ExceptionList设为-1，即清空。

```

.text:00406D46 64 FF 35 00 00 00 00      push     large dword ptr fs:0      ; 将旧的ExceptionList保存
.text:00406D4D 64 C7 05 00 00 00 00 FF FF FF+mov     large dword ptr fs:0, 0FFFFFFFh ; 将当前的ExceptionList设为-1，即清空

```

```

0: kd> dt _KPCR
nt!_KPCR
+0x000 NtTib           : _NT_TIB

```

```

0: kd> dt _NT_TIB
nt!_NT_TIB
+0x000 ExceptionList   : Ptr32 _EXCEPTION_REGISTRATION_RECORD

```

再继续向下看，将当前CPU执行的线程信息给到了ESI寄存器，即FS段的0x124偏移位的内容，也就是_KPRCB结构体的0x4偏移位成员CurrentThread；它是一个结构体_KTHREAD，接着压入该结构体的0x140偏移位成员，也就是先前模式PreviousMode：

```

.text:00406D58 64 8B 35 24 01 00 00      mov     esi, large fs:124h        ; 将当前CPU执行线程信息给到ESI
.text:00406D5F FF B6 40 01 00 00      push   dword ptr [esi+140h]      ; 压入PreviousMode

```

```

+0x120 PrcbData      : _KPRCB
0: kd> dt _KPRCB
nt!_KPRCB
+0x000 MinorVersion  : Uint2B
+0x002 MajorVersion  : Uint2B
+0x004 CurrentThread : Ptr32 _KTHREAD
+0x008 NextThread    : Ptr32 _KTHREAD
+0x00c IdleThread     : Ptr32 _KTHREAD

+0x138 ApcStatePointer : [2] Ptr32 _KAPC_STATE
+0x140 PreviousMode    : Char
+0x141 EnableStackSwap : UChar
+0x142 LargeStack      : UChar
+0x143 ResourceIndex   : UChar

```

接着看下面的代码，首先提升堆栈至Trap_Frame结构体顶部，也就是指向了结构体的第一个成员，然后将3环原来CS的值即Trap_Frame+0x6C+arg_0(0x4)给到EBX，在将CS和1进行与运算判断调用0环函数的来源权限，也就是CPL，然后将与运算的结果给到结构体_KTHREAD的先前模式PreviousMode，这样做的意义是因为有些内核函数可以被3环、0环调用，但由于权限的不同所以执行的内容也不一样，因此需要通过先前模式来决定执行的内容。再接下来就是将栈底指向栈顶的位置，也就是都指向Trap_Frame结构体的第一个成员位置，再将_KTHREAD中的Trap_Frame结构体地址取出，临时存放至当前Trap_Frame结构体的0x3C偏移位，由于Trap_Frame的内容发生了变化，最后再把当前最新的Trap_Frame结构体地址存入_KTHREAD中的Trap_Frame结构体位置（即0x134偏移位）。

```

0406D65 83 EC 48          sub     esp, 48h                ; 提升堆栈至Trap_Frame顶部，即第一个成员的位置
0406D68 8B 5C 24 6C         mov     ebx, [esp+68h+arg_0]    ; 将3环原来的CS的值（Trap_Frame+0x6C）给到EBX
0406D6C 83 E3 01            and     ebx, 1                 ; 将CS和1进行与运算
0406D6F 8B 9E 40 01 00 00    mov     [esi+140h], bl         ; 赋值新PreviousMode
0406D75 8B EC              mov     ebp, esp              ; 将栈顶、栈底都指向Trap_Frame的第一个成员位置
0406D77 8B 9E 34 01 00 00    mov     ebx, [esi+134h]        ; 将_KTHREAD中Trap_Frame结构体的地址取出放入EBX
0406D7D 89 5D 3C            mov     [ebp+3Ch], ebx         ; 临时存放至Trap_Frame的0x3C偏移位
0406D80 89 AE 34 01 00 00    mov     [esi+134h], ebp        ; 将最新的Trap_Frame放入_KTHREAD中的Trap_Frame位置

```

如下代码就比较简单了，将3环的EBP、EIP、3环传参指针存入Trap_Frame结构体调试相关的成员当中，判断当前线程是否开启了调试状态，如果开启则进行跳转，跳转过去之后实际上也是填充Trap_Frame结构体中的调试相关的成员。

```

0406D87 8B 5D 60            mov     ebx, [ebp+60h]          ; 将3环的EBP放入EBX
0406D8A 8B 7D 68            mov     edi, [ebp+68h]          ; 将3环的EIP放入EDI
0406D8D 89 55 0C            mov     [ebp+0Ch], edx          ; EDX即3环传参的指针，存入Trap_Frame+0xC -> DbpArgPointer
0406D90 C7 45 08 00 00 DB BA mov     dword ptr [ebp+8], 0BADB0D00h ; 操作系统用到的一个标志存入Trap_Frame+0x8 -> DbpArgMark
0406D97 89 5D 00            mov     [ebp+0], ebx            ; 3环的EBP存入Trap_Frame+0x0 -> DbpEbp
0406D9A 89 7D 04            mov     [ebp+4], edi            ; 3环的EIP存入Trap_Frame+0x4 -> DbpEip
0406D9D F6 46 2C FF         test    byte ptr [esi+2Ch], 0FFh ; 判断_KTHREAD+0x2C（即DebugActive）是否为-1
0406DA1 0F 85 85 FE FF      jnz     Dr_kss_a                ; 如果DebugActive不是-1表示当前为调试状态，则需要跳转

```

```

text:00406C42                                     loc_406C42:                                     ; CODE XREF: Dr_kss_a+7↑j
text:00406C42 0F 21 C3                                     mov     ebx, dr0
text:00406C45 0F 21 C9                                     mov     ecx, dr1
text:00406C48 0F 21 D7                                     mov     edi, dr2
text:00406C4B 89 5D 18                                     mov     [ebp+18h], ebx
text:00406C4E 89 4D 1C                                     mov     [ebp+1Ch], ecx
text:00406C51 89 7D 20                                     mov     [ebp+20h], edi
text:00406C54 0F 21 DB                                     mov     ebx, dr3
text:00406C57 0F 21 F1                                     mov     ecx, dr6
text:00406C5A 0F 21 FF                                     mov     edi, dr7
text:00406C5D 89 5D 24                                     mov     [ebp+24h], ebx
text:00406C60 89 4D 28                                     mov     [ebp+28h], ecx
text:00406C63 33 DB                                     xor     ebx, ebx
text:00406C65 89 7D 2C                                     mov     [ebp+2Ch], edi
text:00406C68 0F 23 FB                                     mov     dr7, ebx
text:00406C6B 64 8B 3D 20 00 00 00                     mov     edi, large fs:20h
text:00406C72 8B 9F F8 02 00 00                     mov     ebx, [edi+2F8h]
text:00406C78 8B 8F FC 02 00 00                     mov     ecx, [edi+2FCh]
text:00406C7E 0F 23 C3                                     mov     dr0, ebx
text:00406C81 0F 23 C9                                     mov     dr1, ecx
text:00406C84 8B 9F 00 03 00 00                     mov     ebx, [edi+300h]
text:00406C8A 8B 8F 04 03 00 00                     mov     ecx, [edi+304h]
text:00406C90 0F 23 D3                                     mov     dr2, ebx

```

再接着就是跳到某个代码段，但是这个代码段是属于_KiFastCallEntry函数的，所以从本质上来说两个函数最终都是走向同一个地方，无非就是中断门调用会传入5个值需要保存而快速调用不会传入这5个值：

```

text:00406DA8 E9 F5 00 00 00                     jmp     loc_406EA2
text:00406EA2                                     loc_406EA2:                                     ; CODE XREF: _KiBBTUnexpectedRange+18↑j
text:00406EA2                                     ; _KiSystemService+72↑j
text:00406EA2 8B F8                                     mov     edi, eax
text:00406EA4 C1 EF 08                                     shr     edi, 8
text:00406EA7 83 E7 30                                     and     edi, 30h
text:00406EAA 8B CF                                     mov     ecx, edi
text:00406EAC 03 BE E0 00 00 00                     add     edi, [esi+0E0h]
text:00406EB2 8B D8                                     mov     ebx, eax
text:00406EB4 25 FF 0F 00 00                     and     eax, 0FFFh
text:00406EB9 3B 47 08                                     cmp     eax, [edi+8]
text:00406EBC 0F 83 20 FD FF FF                     jnb     _KiBBTUnexpectedRange
text:00406EBC                                     ;
text:00406EC2 83 F9 10                                     cmp     ecx, 10h
text:00406EC5 75 1B                                     jnz     short loc_406EE2
text:00406E0F                                     _KiFastCallEntry proc near                     ; DATA XREF: _KiIrap01+74↑o
text:00406E0F                                     ; KiLoadFastSyscallMachineSpecificRegisters(x)+244↑o
text:00406E0F

```

至此，我们就了解了保存现场的过程。

1.4 系统服务表

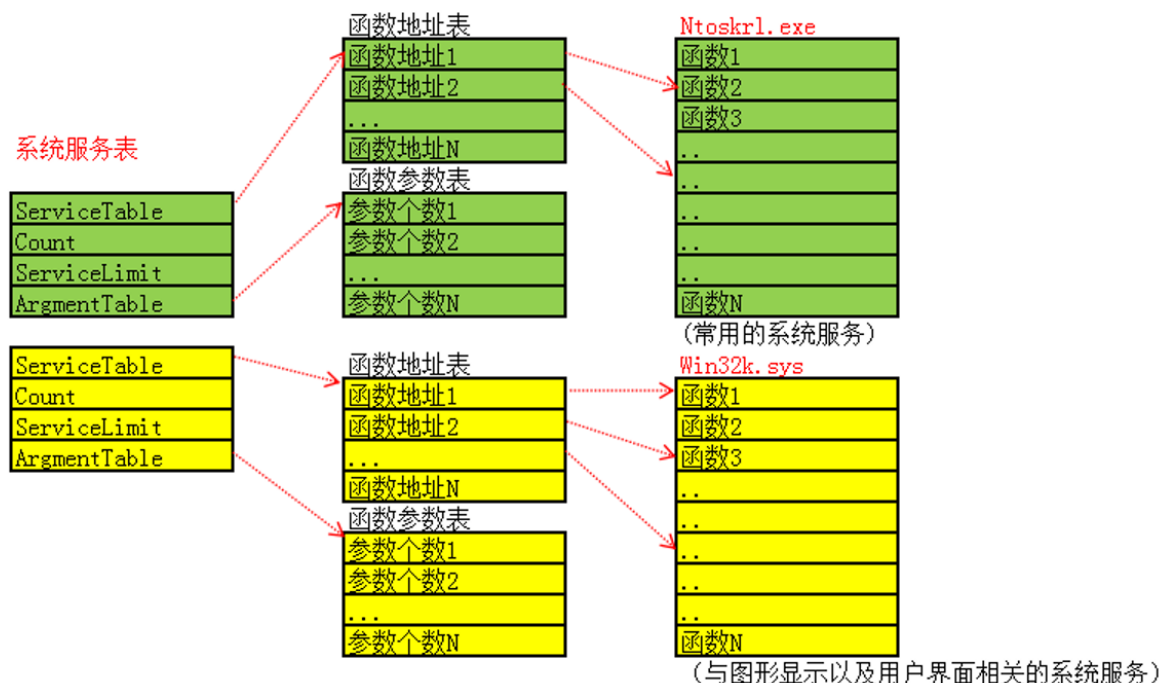
我们知道3环API的本质就是调用0环的函数，那么操作系统是如何根据系统调用号找到要执行的内核函数的，这就需要我们了解一下系统服务表。

1.4.1 SystemServiceTable

如下图是系统服务表的结构，它有四个成员分别是ServiceTable、Count、ServiceLimit、ArgumentTable。ServiceTable里存储的是一个指针，指向了一张函数地址表，表内存储的是函数地址，表内的每个成员大小为4字节；Count里存储的是当前系统服务表被调用的次数；ServiceLimit里存储这当前系统服务表内存储函数的个数；ArgumentTable里存储的是一个指针，指向了函数参数表，表内存储的是函数参数的个数，它的单位是字节，表内的每个成员大小为1字节。

在函数地址表内存储的并不是所有的内核函数，而是在3环经常使用的内核函数，函数地址表和函数参数表是对应关系，即函数地址表成员1的参数个数就是函数参数表成员1。

在Windows XP系统中，系统服务表有两张，下图中绿色、黄色表即体现。它们的结构都是一样的，只是存储的函数、参数不一样，前者存储的是常用的内核函数（Ntoskrl.exe），后者存储的是与图像显示用户界面相关的函数（Win32k.sys）。



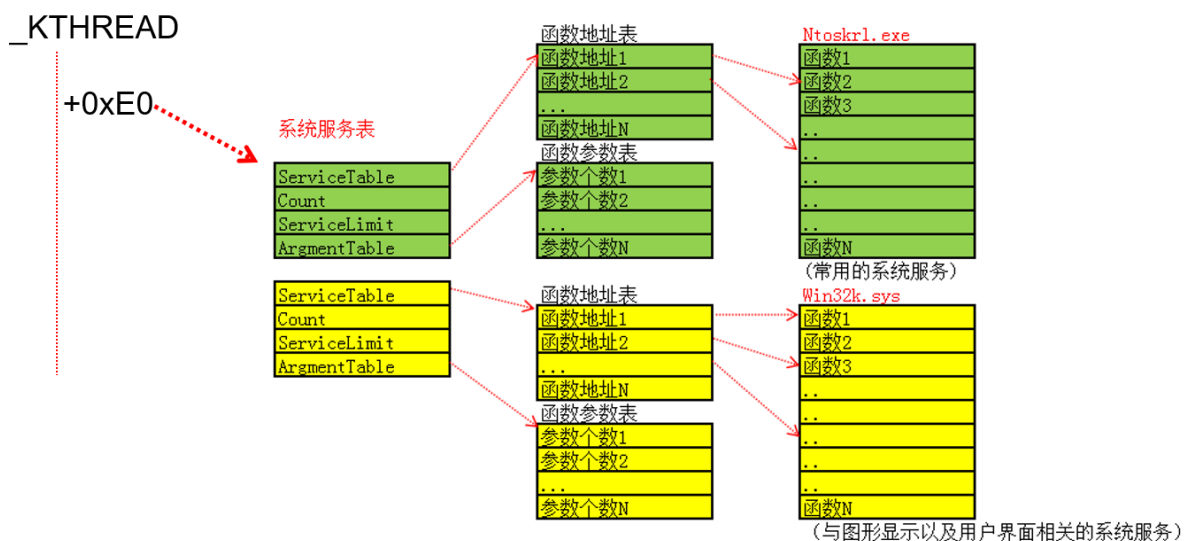
表位置

在了解保存现场内容时，我们知道在Windows内核中每一个进程的每一个线程都有着一个_KTHREAD结构体，它存储着线程相关的信息，在这个结构体里有另外一个结构体_KTHREAD，该结构体的0xE0偏移位成员就存储着系统服务表的地址。

```

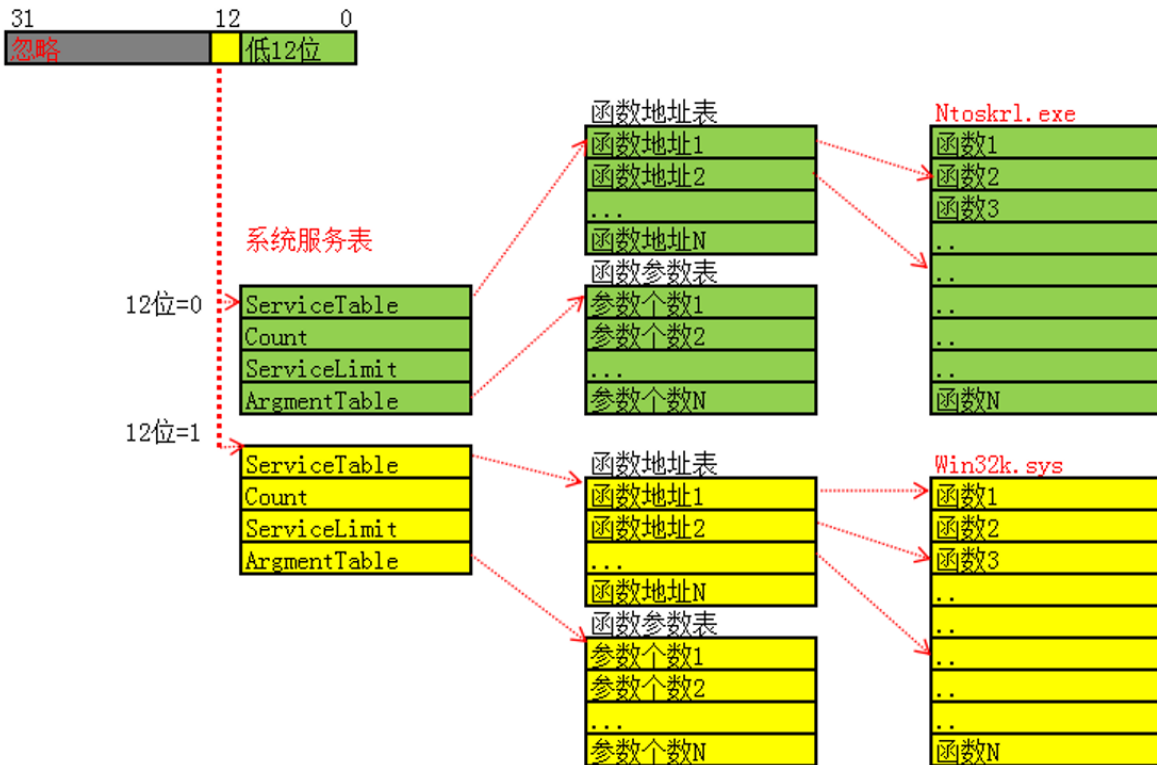
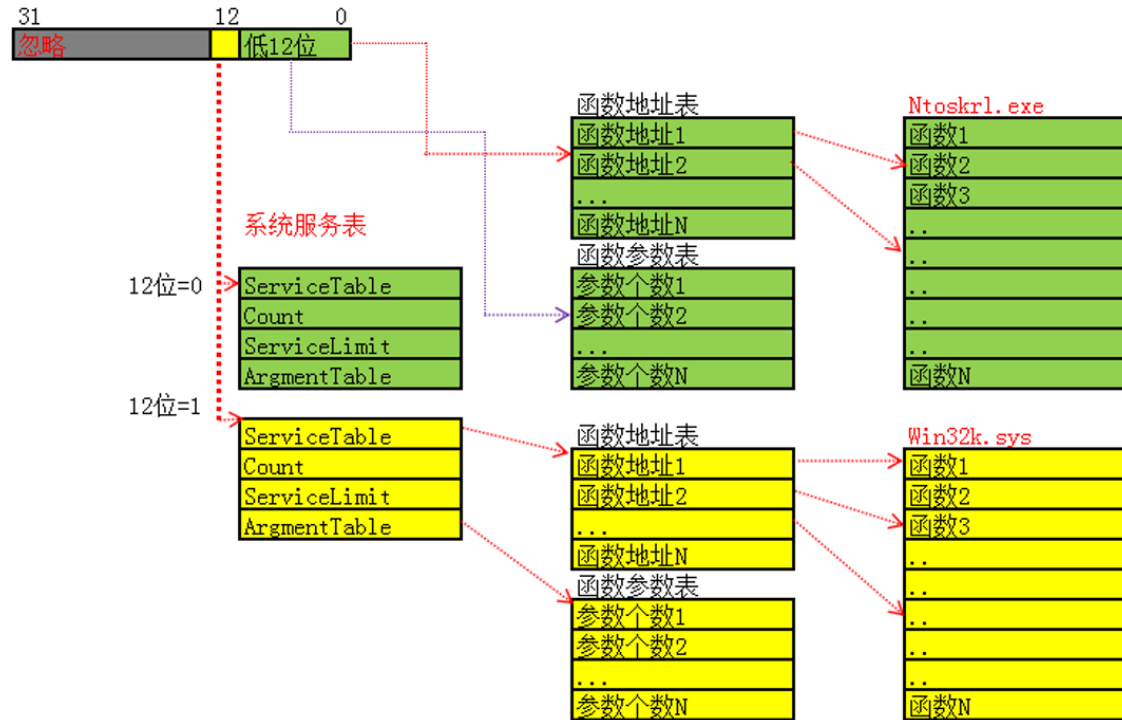
+0x06f Quantum           : Char
+0x070 WaitBlock         : [4] _KWAIT_BLOCK
+0x0d0 LegoData          : Ptr32 Void
+0x0d4 KernelApcDisable  : Uint4B
+0x0d8 UserAffinity      : Uint4B
+0x0dc SystemAffinityActive : UChar
+0x0dd PowerState        : UChar
+0x0de NpxIrql           : UChar
+0x0df InitialNode       : UChar
+0x0e0 ServiceTable      : Ptr32 Void
+0x0e4 Queue             : Ptr32 _KQUEUE
+0x0e8 ApcQueueLock      : Uint4B

```



1.4.2 系统调用号查找函数

3环的API调用就是根据系统调用号找到函数再去使用，虽然系统调用号有32位（4字节）但是真正使用的只有低13位，这13位也分成两部分：第12位值为0则为第一张表（`Ntoskr1.exe`），值为1则为第二张表（`Win32k.sys`）；低12位（第0至第11位）就是对应的函数地址、参数表索引。



代码分析

在保存现场的学习时我们知道无论是KiSystemService还是KiFastCallEntry方法最终都会走到KiFastCallEntry函数内的某块。这一块代码就是系统调用号查找函数的过程，我们来分析一下。

首先是取3环传入的系统调用号给到EDI，然后因为系统调用号我们实际使用的只有13位，其他位都是0，所以将EDI右移8位之后就还剩下5位，接着这5位又和0x30进行与运算，也就表示运算结果只有两种，即0x10或0x0。继续将运算后的结果给到ECX，因为在之前的KiSystemService函数中ESI指向的是_KTHREAD结构体，所以ESI+0xE0指向的就是系统服务表地址，又将与运算结果和系统服务表地址相加，即表示取的系统服务表是第一张还是第二张（系统服务表的宽度就是16字节，即0x10），因此我们可以知道这里的计算方式是很精巧的。

```
.text:00406EA2 8B F8          mov     edi, eax
.text:00406EA4 C1 EF 08       shr     edi, 8
.text:00406EA7 83 E7 30       and     edi, 30h
.text:00406EAA 8B CF         mov     ecx, edi
.text:00406EAC 03 BE E0 00 00 00 add     edi, [esi+0E0h]
```

其次再取系统调用号给到EBX，然后由于之前已经通过下标为12的位找到了对应的表，所以将系统调用号和0xFFFF进行与运算，也就是把下标为12的位设为0，接着EDI+8与EAX进行比较，即系统服务表的ServiceLimit成员与系统调用号，以此来判断传递的函数索引是否越界，如果越界则跳转异常处理。

```
.text:00406EB2 8B D8          mov     ebx, eax
.text:00406EB4 25 FF 0F 00 00 and     eax, 0FFFh
.text:00406EB9 3B 47 08       cmp     eax, [edi+8]
.text:00406EBC 0F 83 20 FD FF FF jnb     _KiBBTUnexpectedRange
```

再接着就是将之前的与运算结果和0x10进行比较，也就是判断系统调用号对应的表，如果与运算结果是0x10则向下会调用到_KeGdiFlushUserBatch函数（[自行查阅手册](#)），如果与运算的结果是0x0则跳转。

```
.text:00406EC2 83 F9 10       cmp     ecx, 10h
.text:00406EC5 75 1B          jnz     short loc_406EE2
.text:00406EC5
.text:00406EC7 64 8B 0D 18 00 00 00 mov     ecx, large fs:18h
.text:00406ECE 33 DB         xor     ebx, ebx
.text:00406ECE
.text:00406ED0
.text:00406ED0
loc_406ED0:
.text:00406ED0 0B 99 70 0F 00 00 or      ebx, [ecx+0F70h]
.text:00406ED6 74 0A         jz      short loc_406EE2
.text:00406ED6
.text:00406ED8 52           push    edx
.text:00406ED9 50           push    eax
.text:00406EDA FF 15 48 C5 48 00 call    ds:_KeGdiFlushUserBatch
.text:00406EDA
.text:00406EE0 58           pop     eax
.text:00406EE1 5A           pop     edx
```

继续向下看，将_KPCR的0x638偏移位的成员值加1，也就是_KPCRB的0x518偏移的成员KeSystemCalls，然后将EDX（即3环参数的指针）给到ESI，接着将系统服务表指向的函数参数表ArgumentTable的地址给到EBX，然后XOR清空ECX。紧接着就是根据系统调用号的索引获取想要调用的函数参数的字节数给到CL，再将系统服务表指向的函数地址表ServiceTable的地址给到EDI，然后再根据系统调用号的索引获取需要调用的函数（这里乘以4是因为存储的函数地址宽度为4字节）给到EBX。

```

.text:00406EE2 64 FF 05 38 06 00 00      inc     large dword ptr fs:638h
.text:00406EE9 8B F2                    mov     esi, edx
.text:00406EEB 8B 5F 0C                mov     ebx, [edi+0Ch]
.text:00406EEE 33 C9                    xor     ecx, ecx
.text:00406EF0 8A 0C 18                mov     cl, [eax+ebx]
.text:00406EF3 8B 3F                    mov     edi, [edi]
.text:00406EF5 8B 1C 87                mov     ebx, [edi+eax*4]

```

然后将堆栈按**ECX即函数参数字节数**进行提升，这样是为了在0环将3环的参数复制进来，接着将**ECX即函数参数字节数**右移2位也就是除以4获得参数的个数，因为在下面的REP指令中每次复制是4字节的，并且该指令的循环次数是ECX的值，所以我们要除以4。接着将ESP给EDI是为设置要循环复制的目的地址，然后判断3环参数地址范围有没有越界，也就是有没有大于_MmUserProbeAddress，如果越界了则进行跳转。最后，执行完REP指令就调用EBX也就是内核函数。

```

.text:00406EF8 2B E1                    sub     esp, ecx
.text:00406EFA C1 E9 02                shr     ecx, 2
.text:00406EFD 8B FC                    mov     edi, esp
.text:00406EFF 3B 35 B4 1E 49 00        cmp     esi, ds:_MmUserProbeAddress
.text:00406F05 0F 83 A8 01 00 00        jnb     loc_4070B3
.text:00406F05
.text:00406F0B
.text:00406F0B                    loc_406F0B:
.text:00406F0B
.text:00406F0B F3 A5                    rep movsd
.text:00406F0D FF D3                    call    ebx

```

1.4.3 SSDT

在上面的学习中我们知道可以通过_KTHREAD结构体的0xE0偏移来会找到系统服务表，除了这个方式以外我们还可以通过另外一种方式来访问，那就是SSDT（System Service Descriptor Table，系统服务描述符表）。

SSDT也有两张表：KeServiceDescriptorTable、KeServiceDescriptorTableShadow。KeServiceDescriptorTable是内核文件导出的，我们可以在内核文件Ntoskrnl.exe的导出表中找到。

CFF Explorer VIII - [ntoskrnl - 副本.exe]

File Settings ?

ntoskrnl - 副本.exe

File: ntoskrnl - 副本.exe

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
 - Section Headers [x]
- Export Directory
- Import Directory
- Resource Directory
- Relocation Directory
- Debug Directory
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor
- UPX Utility

Member	Offset	Size	Value
Characteristics	001A0E00	Dword	00000000
TimeDateStamp	001A0E04	Dword	41107FAA
MajorVersion	001A0E08	Word	0000
MinorVersion	001A0E0A	Word	0000
Name	001A0E0C	Dword	001BCA16
Base	001A0E10	Dword	00000001
NumberOfFunctions	001A0E14	Dword	000005CB
NumberOfNames	001A0E18	Dword	000005CB

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	001A179C	001A413A	001A2E90	001A7C6B
(nFunctions)	Dword	Word	Dword	szAnsi
0000025B	0002B15F	025A	001BFE2B	KeRundownQueue
0000025C	00026C9D	025B	001BFE3A	KeSaveFloatingPointState
0000025D	00061627	025C	001BFE53	KeSaveStateForHibernate
0000025E	0008C500	025D	001BFE6B	KeServiceDescriptorTable
0000025F	000442C9	025E	001BFE84	KeSetAffinityThread
00000260	000267F0	025F	001BFE98	KeSetBasePriorityThread

我们可以在Windbg下直接查看这张表，这张表里一共有4个成员，每个成员都是一个系统服务表，所以SSDT的数据宽度为64字节，但是我们知道Windows XP系统使用了两张系统服务表，目前只能看到一张系统服务表，第二张表却无法看见：

```
0: kd> dd KeServiceDescriptorTable
```

```
8055c6e0 80504734 00000000 0000011c 80504ba8
8055c6f0 00000000 00000000 00000000 00000000
8055c700 00000000 00000000 00000000 00000000
8055c710 00000000 00000000 00000000 00000000
8055c720 00000002 00002710 bf80da45 00000000
8055c730 bada1a80 ba2f39e0 898c0a90 806f5040
8055c740 00000000 00000000 509f92fb 00000001
8055c750 904c533b 01d8dc76 00000000 00000000
```

这是因为第二张系统服务表没有放在SSDT中，而是放在了SSDT Shadow（即KeServiceDescriptorTableShadow）里，我们可以通过Windbg查看这张表并且能看到里面有2张系统服务表，但是这张表是未导出的我们想要使用的话，通常采用内存搜索的方式来找到。

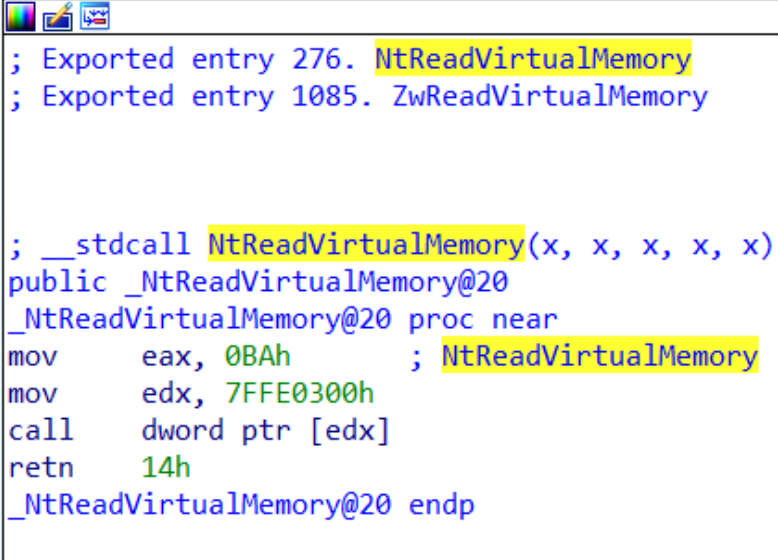
```

0: kd> dd KeServiceDescriptorTableShadow
8055c6a0 80504734 00000000 0000011c 80504ba8
8055c6b0 bf997600 00000000 0000029b bf998310
8055c6c0 00000000 00000000 00000000 00000000
8055c6d0 00000000 00000000 00000000 00000000
8055c6e0 80504734 00000000 0000011c 80504ba8
8055c6f0 00000000 00000000 00000000 00000000
8055c700 00000000 00000000 00000000 00000000
8055c710 00000000 00000000 00000000 00000000

```

实战分析

我们了解了SSTD表后，也就知道了系统服务表的具体成员值，可以根据系统调用号自己来找一下内核函数。如我们之前分析的ReadProcessMemory函数，它的系统调用号就是0xBA。



```

; Exported entry 276. NtReadVirtualMemory
; Exported entry 1085. ZwReadVirtualMemory

; __stdcall NtReadVirtualMemory(x, x, x, x, x)
public _NtReadVirtualMemory@20
_NtReadVirtualMemory@20 proc near
mov     eax, 0BAh          ; NtReadVirtualMemory
mov     edx, 7FFE0300h
call    dword ptr [edx]
retn    14h
_NtReadVirtualMemory@20 endp

```

我们来手动分析一下，先将第一张系统服务表按成员结构进行拆分：

- | | |
|---|---|
| 1 | ServiceTable (函数地址表地址) : 80504734 |
| 2 | Count (系统服务表调用次数) : 00000000 |
| 3 | ServiceLimit (系统服务表内存储函数的个数) : 0000011c |
| 4 | ArgumentTable (函数参数表地址) : 80504ba8 |

接着根据函数地址表地址带入系统调用号找到内核函数的地址，如下图所示我们就找到了ReadProcessMemory最终调用的内核函数NtReadVirtualMemory，它的汇编代码我们也一清二楚：

```

0: kd> dd 80504734+(0xBA*0x4)
80504a1c 805b3c52 805d1f26 80616744 8061411e
80504a2c 80577b68 80642006 80622a6e 80624c28
80504a3c 805a4f10 805a5ed8 805a58e0 805a51fa
80504a4c 805c7e7e 805a246e 805a279a 805c7c8c
80504a5c 8060e108 80521776 80621450 805d4266
80504a6c 805d4148 806214f2 80621582 8062164e
80504a7c 805a3788 8061581e 8061581e 805d0f26
80504a8c 80644cbe 80613b0a 8060fc50 806104c2
0: kd> u 805b3c52
nt!NtReadVirtualMemory:
805b3c52 6a1c      push    1Ch
805b3c54 68c8ae4d80    push    offset nt!MmClaimParameterAdjustDownTime+0x90 (804daec8)
805b3c59 e8127ff8ff     call    nt!_SEH_prolog (8053bb70)
805b3c5e 64a124010000   mov     eax,dword ptr fs:[00000124h]
805b3c64 8bf8          mov     edi,eax
805b3c66 8a8740010000   mov     al,byte ptr [edi+140h]
805b3c6c 8845e0         mov     byte ptr [ebp-20h],al
805b3c6f 8b7514         mov     esi,dword ptr [ebp+14h]

```

同样我们也可以查看该函数的参数总字节数，也就是0x14（20）个字节，按每个参数4字节算，即有5个参数：

```

0: kd> db 80504ba8+0xBA
80504c62 14 04 08 0c 14 08 08 0c-08 10 14 08 04 08 0c 04 .....
80504c72 08 0c 0c 04 08 08 0c 0c-24 08 08 08 0c 04 08 04 .....$.
80504c82 10 08 04 04 04 14 14 10-10 10 10 10 08 14 18 .....
80504c92 04 04 10 0c 08 14 0c 0c-08 08 1c 0c 04 18 14 04 .....
80504ca2 10 04 04 04 08 18 08 08-08 00 10 10 04 04 08 14 .....
80504cb2 10 08 08 10 14 0c 04 04-24 24 18 14 00 10 0c 10 .....$.
80504cc2 10 00 00 00 00 00 00 cc cc-cc cc cc cc cc 0f 57 .....W
80504cd2 c0 b8 40 00 00 00 0f 2b-41 00 0f 2b 41 10 0f 2b ..@....+A..+A..+

```