

API函数的调用过程

1. WindowsAPI

- (1) Application Programming Interface,简称API函数
- (2) 主要是存放在C:\WINDOWS\system32下面所有的dll
- (3) 几个重要的DLL
 - ① Kernel32.dll核心功能模块，比如内存管理、进程线程等
 - ② User32.dll 是Windows用户界面相关应用程序接口，如创建窗口，和发送消息等
 - ③ GDI32.dll 图形设备接口
 - ④ Ntdll.dll:大多数API会通过这个DLL进入内核

2.例 ReadProcessMemory，用IDA查看函数三环实现部分

(1)ReadProccessMemory属于Kernel32模块，函数内部调用了其他模块的函数



```
push    [ebp+4hProcess]
call    ds:[ebp+4], NtReadVirtualMemory@20 ; NtRe
mov     ecx, [[ebp+4hNumberOfBytesRead]
test    ecx, ecx
jnz     short loc_7C8021F9
```

(2)导入表追踪该函数，发现ntdll调用的该函数



7C801278	NtRaiseHardError	ntdll
7C80118C	NtReadFile	ntdll
7C801244	NtReadFileScatter	ntdll
7C80140C	NtReadVirtualMemory	ntdll
7C801480	NtReleaseMutant	ntdll
7C8014A4	NtReleaseSemaphore	ntdll

(3)Ntdll追踪NtReadVirtualMemory函数



```
00000000 _NtReadVirtualMemory@20 proc near
; CODE XREF: LdrFindCreate...
; LdrCreateOutOfProcessInag...
; NtReadVirtualMemory
00000000 mov     eax, 0BAh
00000000 mov     edx, 7FFE0300h
00000000 call    dword ptr [edx]
00000000 retu    14h
00000000 _NtReadVirtualMemory@20 endp
```

(4)0BAh编号 对应一个内核编号

7FFE0300 函数地址 我们以什么方式进0环

三环进零环

1. _KUSER_SHARED_DATA

(1) 在User层和Kernel层分别定义了一个_KUSER_SHARED_DATA结构区域

用于零环与三环共享某些数据

(2)他们使用固定的地址值映射，_KUSER_SHARED_DATA结构趋于在User和Kernel层地址分别为：

User层地址为：0x7ffe0000

Kernnel层地址为：0xffdf0000

特别说明：虽然指向的是同一个物理页，但在User层是只读的，在Kernnel层是可写的

```
ffdf0070 00000000 00000000 00000000 00000000
kd> dt _KUSER_SHARED_DATA
ntdll!_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 DiscountCount : Uint4B
+0x2e0 ConPlusPackage : Uint4B
+0x2e4 LastSystemRITEventTickCount : Uint4B
+0x2e8 NumberOfPhysicalPages : Uint4B
+0x2ec SafeBootMode : UChar
+0x2f0 TraceLogging : Uint4B
+0x2f8 TestRetInstruction : Uint8B
+0x300 SystemCall : Uint4B
+0x304 SystemCallReturn : Uint4B
+0x308 SystemCallPad : [3] Uint8B
+0x320 TickCount : _KSYSTEM_TIME
+0x320 TickCountQuad : Uint8B
+0x330 Cookie : Uint4B
```

API进入0环会调用这个位置的函数

这个函数就是 KiFastSystemCall() (系统支持sysenter/sysexit) 或 KiIntSystemCall()

(OD --->cputid指令--->执行--->结果保存在ecx和edx--->edx包含一个SEP(11位)
--->该位为1是支持，为0是不支持)

```
nov    eax, 00Ah ; LdrCreateOutOfProcessImage(x,x,x,x)+7C4p ...
nov    edx, 7FFE0300h ; NtReadVirtualMemory
call   dword ptr [edx]
retn   14h
_NtReadVirtualMemory@20 endp
```

是否支持sysenter
支持填入 KiFastSystemCall
不支持 KiIntSystemCall

(3)进入0环需要更改的的寄存器

①CS寄存器

②SS与CS保持一致

③堆栈发生切换ESP

④代码执行位置EIP

(4)中断门进0环

传入了两个参数 **eax** 系统调用号，**edx** 说明了API参数在哪里，中断门进入0环

```
ntdll!KiIntSystemCall()函数
.text:7C92EBA5 ; int __stdcall KiIntSystemCall()
.text:7C92EBA5         public _KiIntSystemCall@0
.text:7C92EBA5 _KiIntSystemCall@0 proc near
.text:7C92EBA5
.text:7C92EBA5 arg_4      byte ptr 8
.text:7C92EBA5
.text:7C92EBA5         lea     edx, [esp+arg_4]
.text:7C92EBA9         int     2Eh
.text:7C92EBA9
.text:7C92EBA9         retn
.text:7C92EBA9 _KiIntSystemCall@0 endp
```

参数指针 系统调用号在eax寄存器

①系统会通过0x2E中断号查找idt表中的中断描述符

```
kd> r idtr
idtr=8003f400
kd> dq 8003f400 140
8003f400 804e8e00'00080360 804e8e00'000804db
8003f410 00008500'0058113e 804dee00'0008857b
8003f420 804eee00'00080a30 804e8e00'00080b91
8003f430 804e8e00'00080d12 804e8e00'0008137a
8003f440 00008500'00501198 804e8e00'0008179f
8003f450 804e8e00'000818bc 804e8e00'000819f9
8003f460 804e8e00'00081c52 804e8e00'00081f48
8003f470 804e8e00'0008267e 804e8e00'000828f3
8003f480 804e8e00'00082a10 804e8e00'00082b46
8003f490 804e8500'00a028f3 804e8e00'00082cac
8003f4a0 804e8e00'000828f3 804e8e00'000828f3
8003f4b0 804e8e00'000828f3 804e8e00'000828f3
8003f4c0 804e8e00'000828f3 804e8e00'000828f3
8003f4d0 804e8e00'000828f3 804e8e00'000828f3
8003f4e0 804e8e00'000828f3 804e8e00'000828f3
8003f4f0 804e8e00'000828f3 806f8e00'00081fd0
8003f500 00000000'00080000 00000000'00080000
8003f510 00000000'00080000 00000000'00080000
8003f520 00000000'00080000 00000000'00080000
8003f530 00000000'00080000 00000000'00080000
8003f540 00000000'00080000 00000000'00080000
8003f550 804dee00'0008fba2 804dee00'0008fca5
8003f560 804dee00'0008fe44 804eee00'0008078c
8003f570 804dee00'0008f631 804e8e00'000828f3
8003f580 804d8e00'0008ecf0 804d8e00'0008ecfa
8003f590 804d8e00'0008ed04 804d8e00'0008ed0e
8003f5a0 804d8e00'0008ed18 804d8e00'0008ed22
8003f5b0 804d8e00'0008ed2c 806f8e00'00081728
8003f5c0 804d8e00'0008ed40 804d8e00'0008ed4a
8003f5d0 804d8e00'0008ed54 804d8e00'0008ed5e
8003f5e0 804d8e00'0008ed68 806f8e00'00082b70
8003f5f0 804d8e00'0008ed7c 804d8e00'0008ed86
```

0环CS == 8，0环EIP==804df631,SS 与ESP由TSS提供

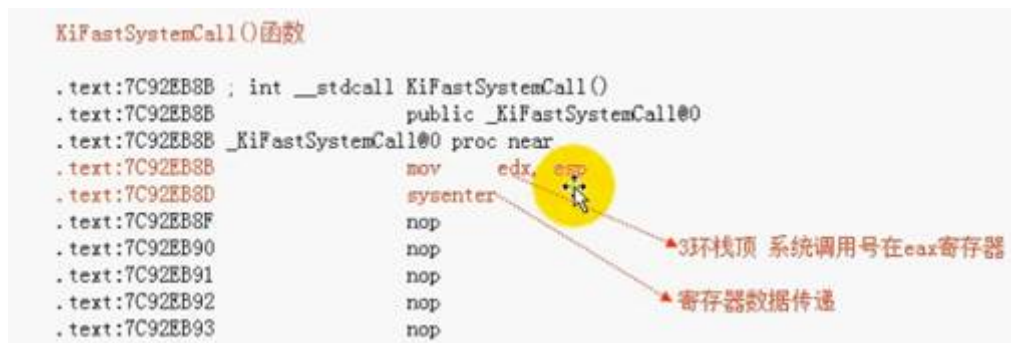
查看EIP地址反汇编

```
kd> u 804df631
nt!KiSystemService:
804df631 6a00      push     0
804df633 55        push     ebp
804df634 53        push     ebx
804df635 56        push     esi
804df636 57        push     edi
804df637 0fa0      push     fs
804df639 bb30000000 mov     ebx,30h
804df63e 8ee3      mov     fs,bx
```

内核模块

(5)快速调用进0环

也是两个参数，**eax**系统调用号，**edx**参数的位置



为什么叫快速调用

中断门进0环，需要的CS、EIP在IDT表中，需要查内存（SS与ESP由TSS提供）

而CPU如果支持**sysenter**指令时，操作系统会提前将CS/SS/ESP/EIP的值存储在MSR寄存器中，**sysenter**指令执行时，CPU会将**MSR寄存器**中的值直接写入相关寄存器，没有读内存的过程，所以叫快速调用，本质是一样的

sysenter进0环

在执行**sysenter**指令之前，操作系统必须指定0环的CS段、SS段、EIP以及ESP

MSR	地址
IA32_SYSENTER_CS	174H
IA32_SYSENTER_ESP	175H
IA32_SYSENTER_EIP	176H

SS段选择子的值由IA_SYSENTER_CS寄存器的地址+8取出

可以通过RDMSR/WRMST来进行读写（操作系统使用WRMST写该寄存器）：

```
kd> rdmsr 174 //查看CS
kd> rdmsr 175 //查看ESP
kd> rdmsr 176 //查看EIP
```

(6)总结

①API通过中断门进0环

1)固定中断号为0x2E

2)CS/EIP由门描述符提供 ESP/SS由TSS提供

3)进入0环后执行的内核函数：NT!KiSystemService

②API通过**sysenter**指令进入0环

1)CS/ESP/EIP是由MSR寄存器提供（SS是算出来的）

2)进入0环后执行的内核函数：NT!KiFastCallEntry

2. 保存环境（进入0环后，原来的寄存器存在哪里了）

(1) Trap_Frame结构体（无论那种调用方式进入0环，都会将3环寄存器环境保存进此结构体）（0环结构体，由操作系统进行维护）

```
kd> dt _Trap_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
+0x024 Dr3             : Uint4B
+0x028 Dr6             : Uint4B
+0x02c Dr7             : Uint4B
+0x030 SegGs           : Uint4B
+0x034 SegEs           : Uint4B
+0x038 SegDs           : Uint4B
+0x03c Edx             : Uint4B
+0x040 Ecx             : Uint4B
+0x044 Eax             : Uint4B
+0x048 PreviousMode    : 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
+0x06c SegCs           : Uint4B
+0x070 EFlags          : Uint4B
+0x074 HardwareEsp     : Uint4B
+0x078 HardwareSegSs   : Uint4B
+0x07c V86Es           : Uint4B
+0x080 V86Ds           : Uint4B
+0x084 V86Fs           : Uint4B
+0x088 V86Gs           : Uint4B
```

(2) _ETHREAD结构体，3环进0环，存储线程状态

```
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

kd> dt _KTHREAD
nt!_KTHREAD
+0x000 Header           : _DISPATCHER_HEADER
+0x010 MutantListHead   : _LIST_ENTRY
+0x018 InitialStack     : Ptr32 Void
+0x01c StackLimit       : Ptr32 Void
+0x020 Tcb              : 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 VdmSafe          : UChar
+0x052 Spare0           : {2} 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
+0x06f0 SpecializedEntry : SINGLE_LIST_ENTRY
```

(3) KPCR 叫CPU控制区（Process Control Region），3环进入0环存储CPU的状态

CPU也有自己的控制块，每一个CPU有一个，叫KPCR

查看CPU数量

```
kd> dd KeNumberProcessors
```

查看KPCR

```
kd> dd KiProcessorBlock L2  
80554040 ffdfff120 00000000
```

//如果有2个核，那么就会出现
//2个地址

```
kd> dt _KPCR
nt!_KPCR
+0x000 NtTib      : _NT_TIB
+0x01c SelfPcr    : Ptr32 _KPCR
+0x020 Preb       : 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 KPRCB        : Ptr32 _KPRCB

kd> dt _NT_TIB
ntdll!_NT_TIB
+0x000 ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase      : Ptr32 Void
+0x008 StackLimit     : Ptr32 Void
+0x00c SubSystemTib   : Ptr32 Void
+0x010 FiberData      : Ptr32 Void
+0x010 Version        : UInt4B
+0x014 ArbitraryUserPointer : Ptr32 Void
+0x018 Self           : Ptr32 _NT_TIB //结构体指针 指向自己

kd> dt _KPRCB
ntdll!_KPRCB
+0x000 MinorVersion   : UInt2B
+0x002 MajorVersion   : UInt2B
+0x004 CurrentThread  : Ptr32 _KTHREAD //当前CPU所执行线程的_KTHREAD
+0x008 NextThread     : Ptr32 _KTHREAD //下一个_KTHREAD
+0x00c IdleThread     : Ptr32 _KTHREAD //当所以线程都执行完了CPU就可C
+0x010 Number         : Char //CPU编号
+0x011 Reserved       : Char
+0x012 BuildType      : UInt2B
+0x014 SetMember      : UInt4B
```

(4) _KiSystemService

```
_KiSystemService proc near
; CODE XREF: ZwAcceptConnectPort(x,x,x,x,x,x,x,x)+C1p
; ZwAccessCheck(x,x,x,x,x,x,x,x)+C1p ...

arg_0 = dword ptr 4

push    [ ] ; _KTRAP_FRAME + 0x064 ErrCode
push    ebp ; +0x060 Ebp
push    ebx ; +0x05c Ebx
push    esi ; +0x058 Esi
push    edi ; +0x054 Edi
push    fs  ; +0x050 SegFs
mov     ebx, 30h ; 为FS寄存器赋值 指向KPCR结构体
db      66h
mov     fs, bx

; Windows内核有个特殊的基本要求，就是只要CPU在内核中运行，就得
; 称为KPCR的数据结构(在Windows中性地址为0xFFDFF000)，怎么使FS
; 位保护模式中，段寄存器的内容并非一个段的基地址，而是一个16位
; 某个表项：
;
; nov ebx,30 //0011 0000
; nov fs,bx
```

下图为TRAP_FRAME结构体保存三环环境

3 环原本的寄存器环境

+0x044 Eax		
+0x048 PreviousPreviousMode		
+0x04c ExceptionList		
+0x050 SegFs		
+0x054 Edi		
+0x058 Esi		
+0x05c Ebx		
+0x060 Ebp		
+0x064 ErrCode		
+0x068 Eip		
+0x06c SegCs		
+0x070 EFlags		
+0x074 HardwareEsp		
+0x078 HardwareSegSs		
+0x07c V86Es		
+0x080 V86Ds		
+0x084 V86Fs		
+0x088 V86Gs		

Windows 中非易失性寄存器需要在中断例程中先保存

中断发生时，保存被中断的代码段和地址，iret 返回到此地址

中断发生时，若发生权限变换，则要保存旧堆栈

虚拟 8086 方式下，变换需要保存段寄存器

6.13 Error Code

6.14 Exception and Mode

三环环境下，通过中断门进入0环，入栈的参数，SS,ESP,EFLAG,C S,EIP

函数会继续执行到如下图所示

```

mov     ebx, 30h           ; 为FS寄存器赋值 指向KPCR结构体
db      66h
mov     fs, bx

; Windows内核有个特殊的基本要求，就是只要CPU在内核中运行，就得
; 称为KPCR的数据结构(在Windows中线性地址为0xFFDF000)，怎么使F
; 位保护模式中，段寄存器的内容并非一个段的基地址，而是一个16位
; 某个表项：
;
; mov ebx,30 //0011 0000
; mov fs,bx
;
; 0环的FS.Base指向CPU自己的KPCR，不是指向当前线程
; 选择码：0x30的结构分析如下：
; (1) Bit0~1:RPL(Requested Privilege Level),要求运行级别，0~3
; (2) Bit2:TI(Table Indicator),0表示采用GDT,1表示采用LDT
; (3) Bit3~15:Index,用做GDT或者LDT的下表。
; 所以，0x30表示运行级别为0环，采用GDT，下标为6。
; windbg查看段描述符：ffc093df f0000001
; Base:FFDF000 指向当前CPU的 KPCR结构
; 偏移+0x00 -> _NT_TIB -> ExceptionList
; KPCR偏移+0x00 -> _NT_TIB -> ExceptionList
; 新的ExceptionList为空白
; 得到当前正在执行的线程信息：
; kPCR偏移

```

30 ---> 0011 0000 ---> ffc093df f0000001 ---> Base:ffdf000指向当前CPU的——KPCR结构

查询KPCR FFDF000处 0x124的偏移

+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 ProcbData : _KPCR		

kd> dt _KPCR

ntdll!_KPCR

+0x000 MinorVersion : UInt2B		
+0x002 MajorVersion : UInt2B		
+0x004 CurrentThread : Ptr32 _KTHREAD		
+0x008 NextThread : Ptr32 _KTHREAD		
+0x00c IdleThread : Ptr32 _KTHREAD		
+0x010 Number : Char		
+0x011 Reserved : Char		
+0x012 BuildType : UInt2B		
+0x014 SetMember : UInt4B		
+0x018 CpuType : Char		
+0x019 CpuID : Char		
+0x01a CpuStep : UInt2B		
+0x01c ProcessorState : _KPROCESSOR_STATE //CPU状态		
+0x03c KernelReserved : [16] UInt4B		

当前CPU所执行线程的_KTHREAD

下一个_KTHREAD

当所以线程都执行完了CPU就可

CPU编号

CPU子版本号


```

mov     esi, ds:0FFDFF124h ; 得到当前正在执行的线程信息,
                                ; kpcr偏移
                                ; +0x120 + 0x4 CurrentThread : Ptr32 _KTHREAD
push    dword ptr [esi+140h] ; 保存老的“先前模式”到堆栈
                                ; KTHREAD
                                ; +0x140 PreviousMode
                                ; ESP_KTRAP_FRAME 结构指针
mov     ebx, [esp+68h+arg_0] ; 取出3环压入的参数CS_KTRAP_FRAME + 0x6C
and     ebx, 1                ; 0环最低位为0 3环最低位为1
mov     [esi+140h], bl        ; 新的“先前模式”
mov     ebp, esp              ; ESP=EBP_KTRAP_FRAME 结构指针
mov     ebx, [esi+134h]       ; KTHREAD中的TrapFrame
mov     [ebp+3Ch], ebx        ; 将_KTHREAD中的Trap_Frame暂时存在这个位置,后面
                                ; .text:00400073          mov     edx, [ebp+3Ch]
                                ; 处会将这个值取出来 重新恢复给_KTHREAD的Trap_Frame
mov     [esi+134h], ebp       ; 将堆栈中形成的_KTRAP_FRAME结构指针赋值给_KTHREAD中的TrapFrame
cld
mov     ebx, [ebp+60h]        ; 3环的EBP
mov     edi, [ebp+68h]        ; 3环的EIP
mov     [ebp+0Ch], edx        ; edx存储的是3环参数的指针:

```

总的来说就是填充TrapFrame结构体，然后把TrapFrame结构体地址填充到KThread 134h偏移处

```

.text:00407EF4          mov     ebx, [ebp+60h]        ; 3环的EBP
.text:00407EF7          mov     edi, [ebp+68h]        ; 3环的EIP
.text:00407EFA          mov     [ebp+0Ch], edx       ; edx存储的是3环参数的指针:
.text:00407EFA          ;
.text:00407EFA          ; _KiFastSystemCall函数,
.text:00407EFA          ;
.text:00407EFA          ;     nov edx, esp
.text:00407EFA          ;     sysenter
.text:00407EFD          mov     dword ptr [ebp+8], 00AD0000h
.text:00407F04          mov     [ebp+8], ebx         ; 3环的ebp存储到_KTRAP_FRAME
.text:00407F04          ; +0x000 DbgEbp 的位置
.text:00407F07          mov     [ebp+4], edi         ; 3环的eip存储到_KTRAP_FRAME
.text:00407F07          ; +0x004 DbgEip 的位置
.text:00407F0A          test    byte ptr [esi+2Ch], 0FFh ; 判断_KTHREAD的 +0x02c DebugActive 是否为-1
.text:00407F0E          jnz     Dr_kss_a             ; 如果处于调试状态, 跳转
.text:00407F14          loc_407F14:                 ; CODE XREF: Dr_kss_a+101j
.text:00407F14          sti                                     ; Dr_kss_a+7C1j
.text:00407F14          ; 关闭中断

```

如果处于调试状态DebugActive == -1，会将调试寄存器也保存到TrapFrame结构体

```

.text:00407D05          mov     ecx, dr1
.text:00407D08          mov     edi, dr2
.text:00407D0B          mov     [ebp+10h], ebx       ; 存储Dr0寄存器到_KTRAP_FRAME + 0x10
.text:00407D0E          mov     [ebp+1Ch], ecx       ; 存储Dr1寄存器到_KTRAP_FRAME + 0x1C
.text:00407D11          mov     [ebp+20h], edi       ; 存储Dr2寄存器到_KTRAP_FRAME + 0x20
.text:00407D14          mov     ebx, dr3
.text:00407D17          mov     ecx, dr6
.text:00407D1A          mov     edi, dr7
.text:00407D1D          mov     [ebp+24h], ebx       ; 存储Dr3寄存器到_KTRAP_FRAME
.text:00407D20          mov     [ebp+28h], ecx       ; 存储Dr6寄存器到_KTRAP_FRAME
.text:00407D23          xor     ebx, ebx
.text:00407D26          mov     [ebp+2Ch], edi       ; 存储Dr7寄存器到_KTRAP_FRAME
.text:00407D29          mov     dr7, ebx            ; 将Dr7清零
.text:00407D2C          mov     edi, large fs:20h    ; 得到_KPCR指针
.text:00407D2F          mov     ecx, [edi+2F0h]      ; _KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Kern
.text:00407D32          mov     dr0, ebx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D35          mov     dr3, ecx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D38          mov     ecx, [edi+300h]      ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D3B          mov     dr2, ebx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D3E          mov     dr3, ecx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D41          mov     ebx, [edi+300h]      ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D44          mov     ecx, [edi+30Ch]      ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D47          mov     dr6, ebx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D4A          mov     dr7, ecx            ; 将_KPCR -> _KPROCESSOR_STATE -> _KSPECIAL_REGISTERS -> Ke
.text:00407D4D          jmp     loc_407F14           ; 关闭中断

```


_Trap_Frame结构

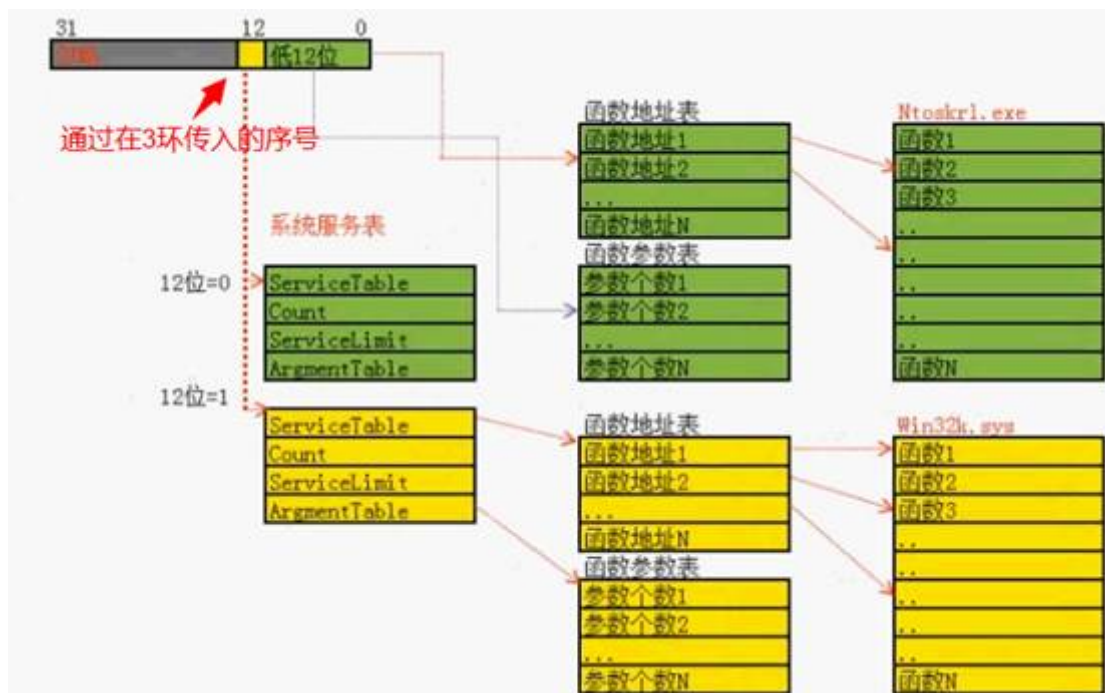
+0x000	DbgEbp	调试等其他作用
+0x004	DbgEip	
+0x008	DbgArgMark	
+0x00c	DbgArgPointer	
+0x010	TempSegCs	
+0x014	TempEsp	
+0x018	Dr0	
+0x01c	Dr1	
+0x020	Dr2	
+0x024	Dr3	
+0x028	Dr6	
+0x02c	Dr7	
+0x030	SegGs	
+0x034	SegEs	
+0x038	SegDs	
+0x03c	Edx	
+0x040	Ecx	
+0x044	Eax	
+0x048	PreviousPreviousMode	
+0x04c	ExceptionList	

3.SysTemServiceTable系统服务表

(1)寻找方式



(2)找到要执行的函数与参数个数



_KiSystemService分析

```

; CODE XREF: _KiBtUnexpectedRange+181j
; KiSystemService+6F1j
mov     edi, eax
shr     edi, 8
and     edi, 0FFFh
; 取出系统调用号, 3环传进来的
; 系统调用号右移8位
; EDI的值右移8位, 然后检测12位是否为1, 也就是检测系统
; WindowsNT基本的 (Native) 系统调用有二百多个, 这些编
; 开始微软把本来由系统进程实现的视窗操作, 也就是图形界
; 速度。这些新增的系统调用号都是大于0x1000的。这些扩充
; 动态安装的模块中, 就是win32k.sys
; ECX中存储的值为0x00或者0x10(调用号大于0x1000的话)
; _KTHREAD -> ServiceTable(+0xE0) 就是SSDT表
mov     ecx, edi
add     edi, [esi+0E0h]
mov     ebx, eax
and     eax, 0FFFh
cmp     eax, [edi+8]
; 系统调用号 只要后12位
; typedef struct _SYSTEM_SERVICE_TABLE
; {
;     PVOID ServiceTableBase; //这个指向系统服务函数地址表
;     PULONG ServiceCounterTableBase;
;     ULONG NumberOfService; //服务函数的个数
;     ULONG ParamTableBase; //参数表基址
; } SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE;

```

```

; } SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE
; +8 是服务函数的个数
jnb     _KiBtUnexpectedRange
cmp     ecx, 10h
jnz     short loc_40803F
mov     ecx, ds:0FFDFF018h
xor     ebx, ebx
; DATA XREF: _KiTrap0E+113j
or      ebx, [ecx+0F70h]
jz      short loc_40803F
push    edx
push    eax
call    ds:_KeGdiFlushUserBatch
pop     eax
pop     edx
; KeGdiFlushUserBatch dd 0
; DATA

```

```

; KiFastCallEntry+C47j
inc     dword ptr ds:0FFDF638h ; KPCR -> +0x518 KeSystemCalls 增加1
mov     esi, edx                ; edx存储着三环传入函数的指针
mov     ebx, [edi+0Ch]          ; _SYSTEM_SERVICE_TABLE -> ParamTableBase SSDT参数表的地址
xor     ecx, ecx
mov     cl, [eax+ebx]           ; eax系统调用号 参数表+调用号得到参数的个数
mov     edi, [edi]              ; _SYSTEM_SERVICE_TABLE -> ServiceTableBase
mov     ebx, [edi+eax*4]        ; 0环函数的地址
sub     esp, ecx                ; 提升堆栈 提升高度为CL
shr     ecx, 2                  ; 参数总长度/4 == 参数的个数
mov     edi, esp                ; 设置要COPY的目的地址
cmp     esi, ds:_HmUserProbeAddress
jnb     loc_400210

```

4.SSDT系统服务描述符表

(1)SSDT 的全称是 System Services Descriptor Table ， 系统服务描述符表

dd KeServiceDescriptorTable(SSDT)

导出的 声明一下就可以使用了

dd KeServiceDescriptorTableShadow(SSDT Shadow)

未导出 需要用其他的方式来查找

(2)SSDT中每一个成员都是一个系统描述符

5.总结

(1)系统调用API两种方式 KiSystemService KiFastEntryCall

①KiSystemService 使用中断门int 2E 进入0环，CS与EIP由中断门描述符提供，SS与ESP由TSS任务段提供（从内存中取）

②KiFastEntryCall 使用MSR寄存器提供0环的寄存器环境，

MSR	地址
IA32_SYSENTER_CS	174H
IA32_SYSENTER_ESP	175H
IA32_SYSENTER_EIP	176H

SS段选择子的值由IA_SYSENTER_CS寄存器的地址+8取出

(2) 无论哪种方式，保存3环寄存器环境使用TrapFrame结构体（注：进入0环时，FS地址保存KPCR,通过KPCR寻找ETHREAD结构体，填充TrapFrame）

+0x048	PreviousPreviousMode	windows 中非易失性寄存器需要在中断例程中先保存
+0x04c	ExceptionList	
+0x050	SegFs	
+0x054	Edi	
+0x058	Esi	
+0x05c	Ebx	
+0x060	Ebp	
+0x064	ErrCode	中断发生时, 保存被中断的代码段和地址, iret 返回到此地址
+0x068	Eip	
+0x06c	SegCs	
+0x070	EFlags	
+0x074	HardwareEsp	中断发生时, 若发生权限变换, 则要保存旧堆栈
+0x078	HardwareSegSs	
+0x07c	V86Es	虚拟 8086 方式下, 变换需要保存段寄存器
+0x080	V86Ds	
+0x084	V86Fs	
+0x088	V86Gs	

6.13 Error Code

6.14 Exception and Mode

三环环境下, 通过中断门进入0环, 入栈的参数, SS,ESP,EFLAG,C S,EIP

6. 临界区

并发是指多个线程在同时执行:

单核 (是分时执行, 不是真正的同时)

多核 (在某一个时刻, 会同时有多个线程再执行)

同步则是保证在并发执行的环境中各个线程可以有序的执行

(1)

(2) 防止线程切换时, 多个线程访问同一片内核空间, 造成数据的冗乱 (线程的切换)



7. 自旋锁

(1) 防止多核系统同时访问同一片内存空间, 造成数据冗乱

2、Windows自旋锁

参考: KeAcquireSpinLockAtDpcLevel

关键代码:

```
lock bts dword ptr [ecx], 0
```

LOCK是锁前缀，保证这条指令在同一时刻只能有一个CPU访问

BTS指令：设置并检测 将ECX指向数据的第0位置1

如果[ECX]原来的值==0 那么CF=1 否则CF=0

自旋锁只对多核有意义

自旋锁与临界区、事件、互斥体一样，都是一种同步机制，都可以让当前线程处于等待状态，区别在于自旋锁不用切换线程