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

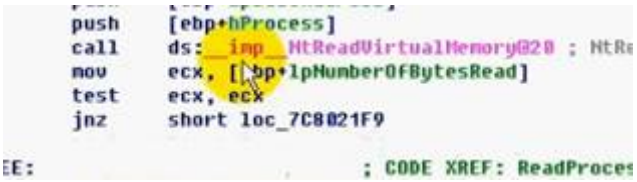
② User32.dll 是Windows用户界面相关应用程序接口，如创建窗口，和发送消息等

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



(2)导入表追踪该函数，发现ntdll调用的该函数

(3)Ntdll追踪NtReadVirtualMemory函数

(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_SAHRED_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 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
   +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 --->cpuid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 ）

(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

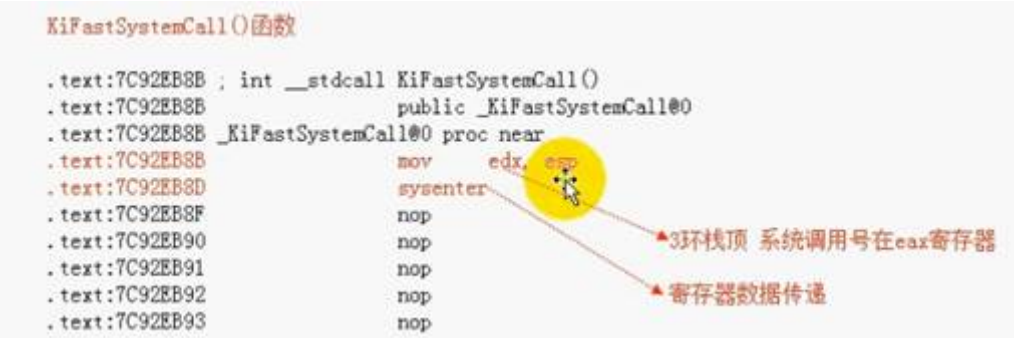
①系统会通过0x2E中断号查找idt表中的中断描述符

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

查看EIP地址反汇编

(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 _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
   +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 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
   +0x06c SegCs            : Uint4B
   +0x070 EFlags           : Uint4B
   +0x074 HardwareEsp      : Uint4B
   +0x078 HardwareSegSs    : Uint4B
   +0x07c V86Es            : Uint4B
   +0x080 V86Ds            : Uint4B
   +0x084 V86Fs            : Uint4B
   +0x088 V86Gs            : Uint4B
```

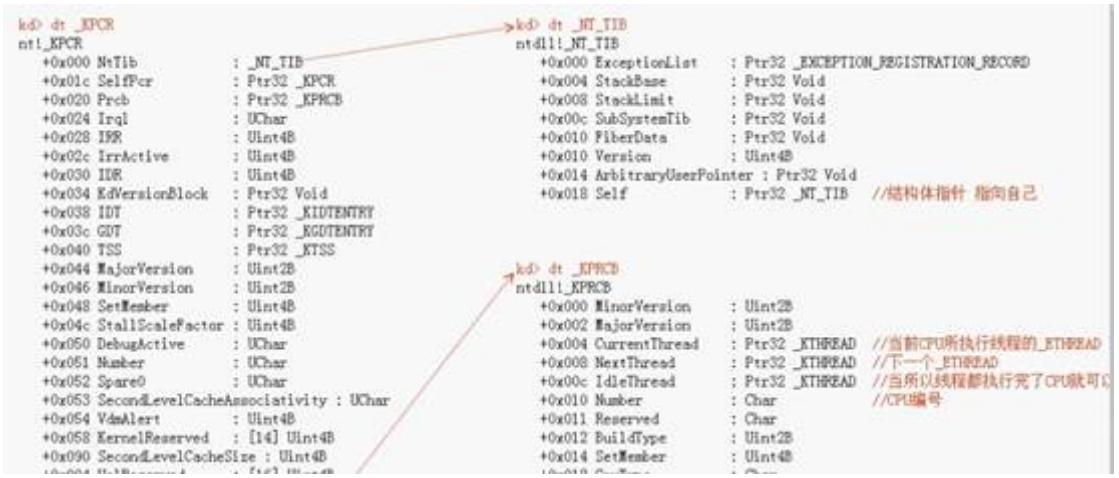
(2) _ETHREAD结构体，3环进0环，存储线程状态

```
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 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 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
   +0x060 SwapListEntry    : _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



(4) _KiSystemService

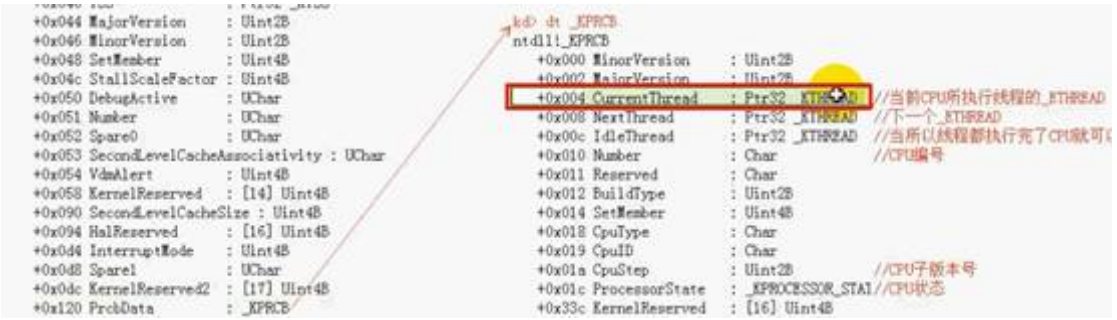
下图为TRAP_FRAME结构体保存三环环境

3 环原本的寄存器环境

函数会继续执行到如下图所示

30 ---> 0011 0000 ---> ffc093df f0000001 ---> Base:ffdff000指向当前CPU的——KPCR结构

查询KPCR FFDFF000处 0x124的偏移

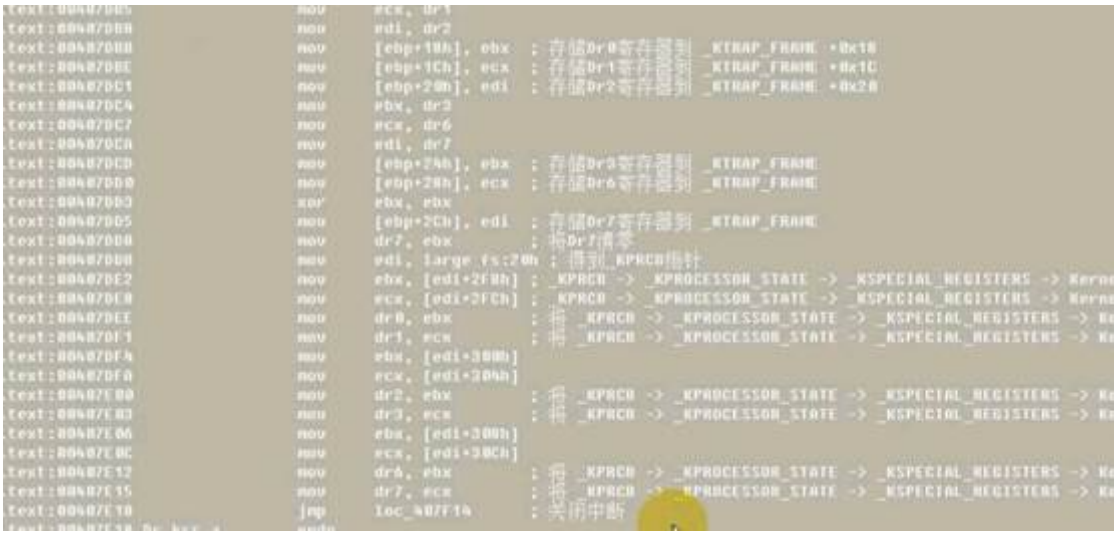


```
mov     esi, ds:0FFDFF124h ; 得到当前正在执行的线程信息，
                           ; kpcr偏移
                           ;     +0x120 + 0x4  CurrentThread    : Ptr32 _KTHREAD
push    dword ptr [esi+140h] ; 保存老的“先前模式”到堆栈
                           ; KTHREAD
                           ;     +0x140 PreviousMode
sub     esp, 4Bh           ; 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:00408073                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                                        ;     mov edx,esp
.text:00407EFA                                        ;     sysenter
.text:00407EFD                mov     dword ptr [ebp+8], 0BADB0D00h
.text:00407F04                mov     [ebp+0], 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
.text:00407F14 loc_407F14:                            ; CODE XREF: Dr_kss_a+10Tj
.text:00407F14                                        ; Dr_kss_a+7CTj
.text:00407F14                sti                     ; 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



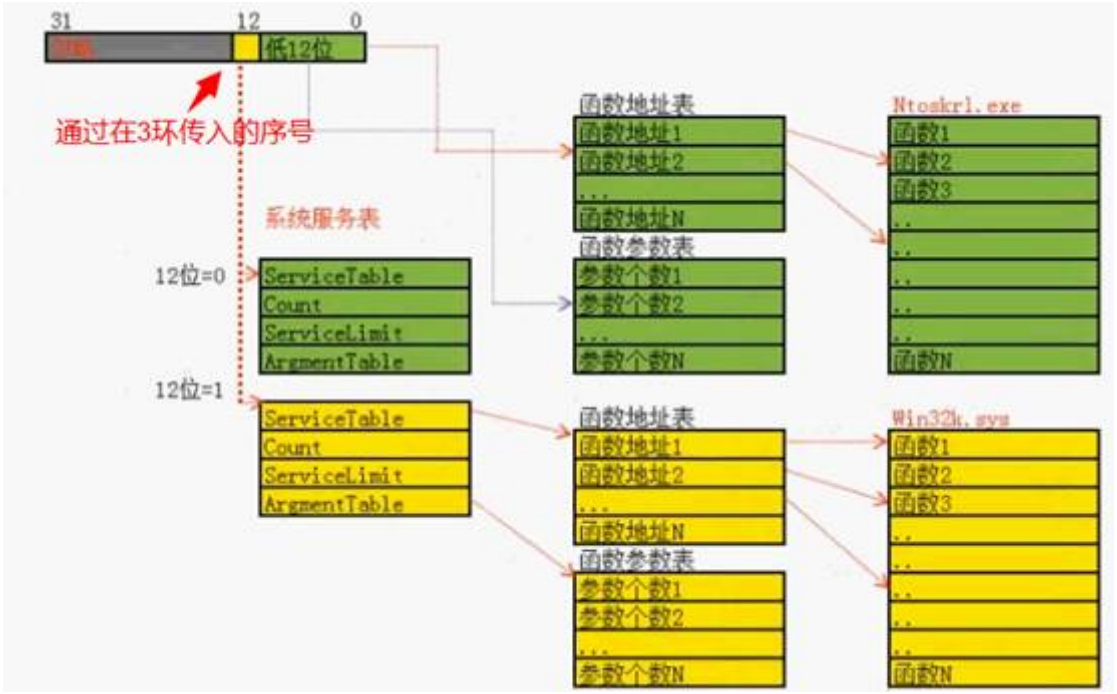
_Trap_Frame结构

+0x000	DbgEbp	
+0x004	DbgEip	
+0x008	DbgArgMark	
+0x00c	DbgArgPointer	
+0x010	TempSegCs	
+0x014	TempEsp	
+0x018	Dr0	
+0x01c	Dr1	
+0x020	Dr2	调试等其他作用
+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: _KiDBTUnexpectedRange+181j
                        ; _KiSystemService+6F↑j
mov     edi, eax        ; 取出系统调用号，3环传进来的
shr     edi, 8          ; 系统调用号右移8位
and     edi, 10h        ; 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）系统调用有两百多个，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
mov     ecx, edi        ; ECX中存储的值为0x00或者0x10(调用号大于0x1000的话)
add     edi, [esi+0E0h] ; _KTHREAD -> ServiceTable(+0xE0) 就是SSDT表
mov     ebx, eax
and     eax, 0FFFh      ; 系统调用号  只要后12位
cmp     eax, [edi+8]    ; 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_TAB
                        ;
                        ; +8 是服务函数的个数
jnb     _KiDBTUnexpectedRange
cmp     ecx, 10h        ; 与0x10比较
jnz     short loc_40803F ; _KPRCB -> +0x518 KeSystemCalls 增加1
mov     ecx, ds:0FFDFF018h ; 如果调用号大于0x1000
xor     ebx, ebx

                        ; DATA XREF: _KiTrap0E+113↓o
or      ebx, [ecx+0F70h]
jz      short loc_40803F ; _KPRCB -> +0x518 KeSystemCalls 增加1
push    edx
push    eax
call    ds:_KeGdiFlushUserBatch
pop     eax
pop     edx
_KeGdiFlushUserBatch dd 0               ; DATA
```

```
                        ; _KiFastCallEntry+C4↑j
inc     dword ptr ds:0FFDFF638h ; _KPRCB -> +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
cmp     esi, ds:_MmUserProbeAddress
jnb     loc_408210
```

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，

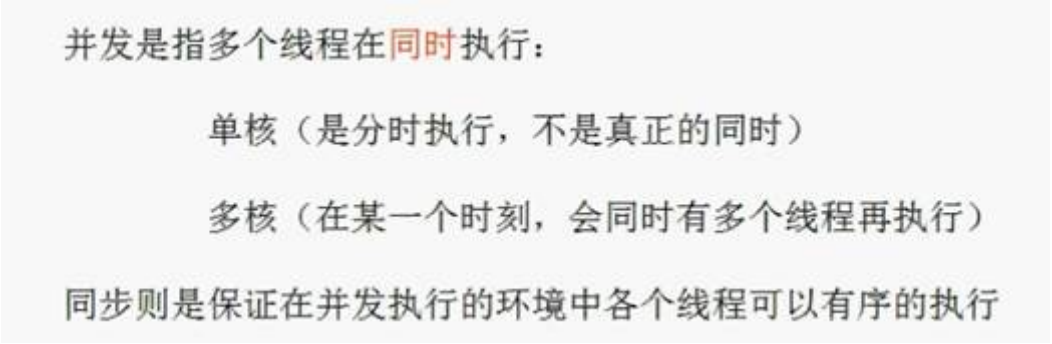
MSR	地址
IA32_SYSENTER_CS	174H
IA32_SYSENTER_ESP	175H
IA32_SYSENTER_EIP	176H

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）

6. 临界区

(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

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