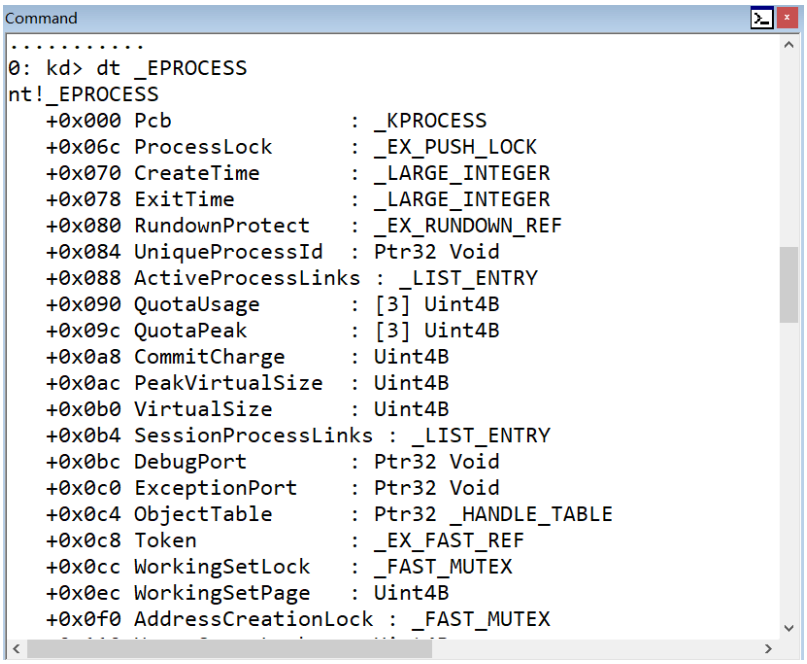
1 进程与线程

在初级班的学习后我们都会有进程、线程的概念，从操作系统的角度去看进程、线程实际上都是结构体，当创建一个进程或线程，本质上就是分配一块内存来填充对应的结构体。因此我们要摸清楚进程、线程的具体细节，就要对它们的结构体足够了解。

1.1 进程结构体

每个Windows进程在0环都有一个对应的结构体：_EPROCESS，该结构体包含了进程所有的重要信息。

我们可以使用Windbg来查看该结构体：



1.1.1 结构体成员

我们通过Windbg可以看到进程结构体_EPROCESS有很多个成员，目前我们只需要了解一些比较重要的成员，其他的可以等用到时再了解。

_KPROCESS

进程结构体_EPROCESS的第一个成员（即0x0偏移位）也是一个结构体_KPROCESS，我们也可以使用Windbg来查看一下：

```
Command
0: kd> dt _KPROCESS
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
```

以下就是一些_KPROCESS的成员，我们可以简单了解一下：

偏移	成员	作用
0x0	Header (_DISPATCHER_HEADER)	可等待对象，可以通过WaitForSingleObject函数来使用可等待对象（例如互斥体、事件）。
0x18	DirectoryTableBase ([2] Uint4B)	页目录表基址，一个进程都有一个页目录表，在该表里记录着线性地址所引用的物理页，所以修改该成员就可以控制整个进程。
0x38 0x3C	KernelTime (Uint4B) UserTime (Uint4B)	统计信息，分别记录了一个进程在0环、3环所花费的时间。

偏移	成员	作用
0x5C	Affinity (Uint4B)	规定了进程里面的所有线程可以在哪个CPU上运行。如果该成员值为1则这个进程的所有线程只能在0号CPU上运行，也就表示我们可以将该成员值转为二进制数值，**第N位为1即表示可以在N号CPU上运行所有线程，举一反三，如果该值为5，则第0、2位为1，也就表示可以在0、2号CPU上运行。** 从我们的实验环境上来看该成员仅有4字节（即32位），所以也就只支持32核CPU。 **我们了解基本逻辑之后就可以明白，假设某台计算机只有1核的CPU，我们将该成员值设为大于1的数值，那么该进程就没法运行了。**
0x62	BasePriority (Char)	基础优先级或最低优先级，规定了该进程中所有线程最基本的优先级。

其他成员

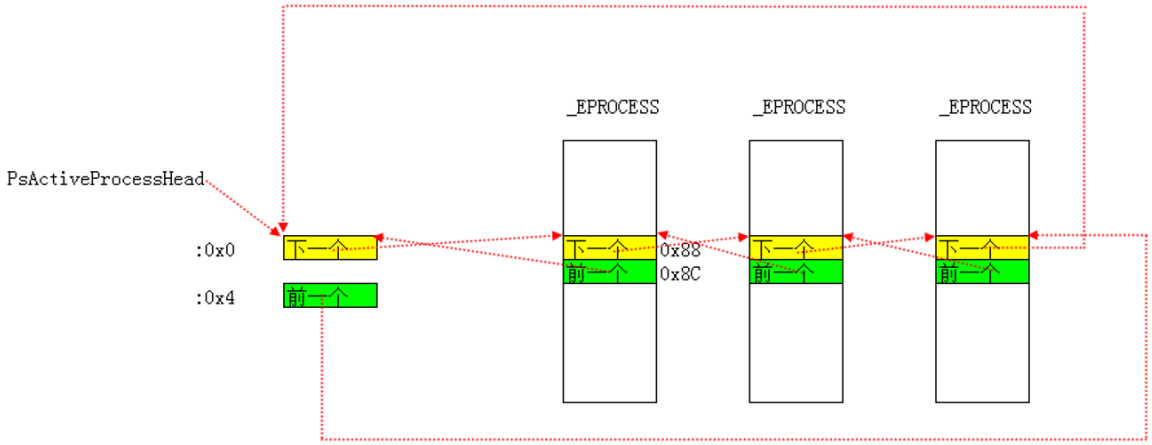
接着我们来了解一下_EPROCESS的其他几个成员：

偏移	成员	作用
0x70 0x78	CreateTime (_LARGE_INTEGER) ExitTime (_LARGE_INTEGER)	记录信息，分别表示进程创建、退出的时间。
0x84	UniqueProcessId (Ptr32 Void)	进程编号，也就是我们通过任务管理器中所看见的PID。
0x88	ActiveProcessLinks (_LIST_ENTRY)	双向链表，将所有活动的进程都链接成一个表，我们可以通过这个成员找到前一个进程和后一个进程的结构体（需要注意这里找到的并不是结构体的起始位置，而是结构体的0x88偏移位）。
0x90 0x9C	QuotaUsage ([3] Uint4B) QuotaPeak ([3] Uint4B)	统计信息，与物理页有关。
0xA8 0xAC 0xB0	CommitCharge (Uint4B) PeakVirtualSize (Uint4B) VirtualSize (Uint4B)	统计信息，与虚拟内存有关。

偏移	成员	作用
0xBC 0xC0	DebugPort (Ptr32 Void) ExceptionPort (Ptr32 Void)	与调试相关的信息。
0xC4	ObjectTable (Ptr32 _HANDLE_TABLE)	句柄表，存储了当前进程使用的其他内核对象的句柄；我们可以通过遍历其他进程的句柄表，如果表中存在当前进程的进程结构体地址，那就说明当前进程被使用，也就是被调试，我们可以此作为一种反调试的手段。
0x11C	VadRoot (Ptr32 Void)	标识了用户空间（低2G）有哪些地址没被占用。
0x174	ImageFileName ([16] UChar)	当前进程的名字。
0x1A0	ActiveThreads (Uint4B)	当前进程的活动线程数量。
0x1B0	Peb (_PEB)	PEB（Process Environment Block，进程环境块）：进程在3环的一个结构体，里面包含了进程的模块列表、是否处于调试状态等信息。

ActiveProcessLinks

如下所示就活动进程双向链表的完整结构，PsActiveProcessHead指向了该链表的表头位置。



我们可以在Windbg中查看表头，并对应找到其指向的进程结构体：

```
0: kd> dd PsActiveProcessHead
805637b8  89a328b8 89674448 00000001 b1b81a64
805637c8  00000000 00040001 00000000 805637d4
805637d8  805637d4 00000000 7c920000 00000000
805637e8  00000000 00000000 00000000 805647a8
805637f8  8052b334 00000000 00000000 00000000
80563808  890aa458 890aa458 00000000 00000000
80563818  00000000 00000000 00000001 bad07d58
80563828  00000000 00040001 00000000 80563834
0: kd> dt _EPROCESS 89a328b8-0x88
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER 0x0
   +0x078 ExitTime         : _LARGE_INTEGER 0x0
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : 0x00000004 Void
```

1.2 线程结构体

每个进程默认都会有一个线程，每启动一个线程，在内存里就会多一个线程结构体，即_ETHREAD。

我们可以使用Windbg来查看该结构体：

Command

```
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
```

1.2.1 结构体成员

我们通过Windbg可以看到进程结构体_ETHREAD有很多个成员，目前我们只需要了解一些比较重要的成员，其他的可以等用到时再了解。

_KTHREAD

与进程结构体一样，线程结构体的第一个成员（即0x0偏移位）也是一个结构体_KTHREAD，我们可以使用Windbg来查看一下：

```
Command
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
```

我们简单了解一下它的几个成员：

偏移	成员	作用
0x0	Header (_DISPATCHER_HEADER)	可等待对象，与进程结构体的_KPROCESS中的Header成员是一样的。
0x18 0x1C 0x28	InitialStack (Ptr32 Void) StackLimit (Ptr32 Void) KernelStack (Ptr32 Void)	与线程切换有关的成员。
0x20	Teb (Ptr32 Void)	TEB（Thread Environment Block，线程环境块），线程在3环的一个结构体，里面包含了线程的相关信息。我们在3环可以通过FS:[0]来找到TEB。

偏移	成员	作用
0x2C	DebugActive (UChar)	如果该值为-1则表示不能使用调试寄存器：Dr0-Dr7，如果当前线程正在被调试则该值就不是-1。
0x34 0xE8 0x138 0x14C	ApcState (_KAPC_STATE) ApcQueueLock (Uint4B) ApcStatePointer ([2] Ptr32 _KAPC_STATE) SavedApcState (_KAPC_STATE)	与APC相关的成员。
0x2D	State (Uchar)	表示线程状态：准备就绪、等待、正在执行。
0x6C	BasePriority (Char)	基础优先级或最低优先级，它的初始值就是所属进程的结构体的BasePriority值，如果你想要修改可以通过KeSetBasePriorityThread函数进行重新设定。
0x70	WaitBlock ([4] _KWAIT_BLOCK)	如果当前线程执行了WaitForSingleObject函数，那么当前线程结构体中_KTHREAD的WaitBlock成员就会记录等待的对象。
0xE0	ServiceTable (Ptr32 Void)	指向系统服务表的基址。
0x134	TrapFrame (Ptr32 _KTRAP_FRAME)	这是一个结构体，用于进0环时保存环境。
0x140	PreviousMode (Char)	先前模式，某些内核函数会判断程序调用它时是0环还是3环，就是通过该成员去判断。
0x1B0	ThreadListEntry (_LIST_ENTRY)	双向链表，一个进程的所有线程都链入这个表中了。

其他成员

除了_KTHREAD以外，我们来看一下其他几个在_ETHREAD结构体中的成员：

偏移	成员	作用
0x1EC	Cid (_CLIENT_ID)	进程有编号，线程也有自己的编号，也就是这里的CID，需要注意的是这里的CID不光是线程ID，也包含了进程ID。即_CLIENT_ID[0]为进程ID，_CLIENT_ID[4]为线程ID。
0x220	ThreadsProcess (Ptr32 _EPROCESS)	指向当前线程所属进程的_EPROCESS结构体地址。
0x22C	ThreadListEntry (_LIST_ENTRY)	双向链表，一个进程的所有线程都链入这个表中了。

1.3 _KPCR

如果要逆向分析操作系统内核代码，需要具备两个前置知识：1.段、页相关代码能够理解；2.至少要知道三个结构体：_EPROCESS、_ETHREAD、_KPCR。因此本章节我们需要了解_KPCR结构体。

1.3.1 介绍

每个进程或线程都一个结构体来描述本身，同样CPU也有这样一个结构体来描述自己，即_KPCR。

当线程进入0环时，FS:[0]指向的就不再是TEB了，而变成了_KPCR；每个CPU（每核）都有一个_KPCR结构体，该结构体中存储了CPU本身需要用到的一些重要数据：如GDT、IDT以及线程相关的一些信息。

我们同样可以在Windbg中查看_KPCR结构体：

Command

```
0: kd> dt _KPCR
nt!_KPCR
   +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
```

1.3.2 成员

_NT_TIB

在_KPCR结构体中的0x0偏移位成员，它是一个结构体_NT_TIB。

```
0: kd> dt _NT_TIB
nt!_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
```

该结构体几个重要的成员如下：

偏移	成员	作用
0x0	ExceptionList (Ptr32 _EXCEPTION_REGISTRATION_RECORD)	表示当前线程内核异常链表(SEH)。
0x4 0x8	StackBase (Ptr32 Void) StackLimit (Ptr32 Void)	表示当前线程内核栈的基址和大小。
0x18	Self (Ptr32 _NT_TIB)	这是一个指针，指向了当前_NT_TIB结构体，也就是_KPCR结构体（便于查找）。

其他

_KPCR结构体其他几个成员的信息如下：

偏移	成员	作用
0x1C	SelfPcr (Ptr32 _KPCR)	这是一个指针，指向了当前_KPCR结构体。
0x20	Prcb (Ptr32 _KPRCB)	这是一个指针，指向了_KPRCP结构体，即_KPCR结构体0x120偏移位的成员。这里也是为了方便找到结构体，防止_KPCR结构体发生了变化，就可以通过该成员找到_KPRCP结构体。
0x38	IDT (Ptr32 _KIDTENTRY)	这是一个指针，指向了中断描述符表IDT。
0x3C	GDT (Ptr32 _KGDTENTRY)	这是一个指针，指向了全局描述符表GDT。
0x40	TSS (Ptr32 _KTSS)	这是一个指针，指向了任务段TSS，每个CPU都有一个TSS。
0x51	Number (UChar)	表示CPU的编号：1、2、3、4...
0x120	PrcbData (_KPRCB)	拓展结构体。

_KPRCB

_KPRCB是_KPCR结构体的拓展结构体，我们先了解几个成员，后续用到了再逐个去了解。

偏移	成员	作用
0x4	CurrentThread (Ptr32 _KTHREAD)	这是一个指针，指向当前线程的结构体。
0x8	NextThread (Ptr32 _KTHREAD)	这是一个指针，指向即将切换的下一个线程的结构体。
0xC	IdleThread (Ptr32 _KTHREAD)	这是一个指针，指向了一个空闲的线程的结构体。

1.4 33个链表

线程有3种状态：就绪、等待、运行。正在运行中的线程存储在_KPCR结构体中，就绪和等待的线程全在另外的33个链表中。这33个链表有1个等待链表核32个就绪链表（我们也称之为调度链表）。

1.4.1 等待链表

所谓等待链表，即当线程调用了Sleep、WaitForSingleObject等函数时，就会被链入该表。

等待链表是一个双向链表，因此我们可以通过KiWaitListHead来遍历这张表，找到所有的等待线程。如下图所示我们可以通过KiWaitListHead这个全局变量，在Windbg中找到等待链表头，它的两个成员分别指向了前、后的等待线程。

```
0: kd> dd KiWaitListHead
8055c488  8991fae8 898b7280 00000011 00000000
8055c498  e57a42bd d6bf94d5 01000013 ffdff980
8055c4a8  ffdff980 805025ae 00000000 0000e2ae
8055c4b8  00000000 00000000 8055c4c0 8055c4c0
8055c4c8  00000000 00000000 8055c4d0 8055c4d0
8055c4d8  00000000 00000000 00000000 89a286d8
8055c4e8  00000000 00000000 00040001 00000000
8055c4f8  89a28748 89a28748 00000000 00000000
```

如果我们想要找到它们对应的线程结构体需要在地址上减去0x60，因为这里记录的地址实际上是_ETHREAD（_KTHREAD）中的0x60偏移位成员WaitListEntry。

```
0: kd> dd KiWaitListHead
8055c488  8991fae8 898b7280 00000011 00000000
8055c498  e57a42bd d6bf94d5 01000013 ffdff980
8055c4a8  ffdff980 805025ae 00000000 0000e2ae
8055c4b8  00000000 00000000 8055c4c0 8055c4c0
8055c4c8  00000000 00000000 8055c4d0 8055c4d0
8055c4d8  00000000 00000000 00000000 89a286d8
8055c4e8  00000000 00000000 00040001 00000000
8055c4f8  89a28748 89a28748 00000000 00000000
0: kd> dt _KTHREAD 8991fae8-0x60
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x8991fa98 -
   +0x018 InitialStack     : 0xb19f2000 Void
   +0x01c StackLimit       : 0xb19ef000 Void
   +0x020 Teb              : 0x7ffdc000 Void
   +0x024 TlsArray         : (null)
   +0x028 KernelStack      : 0xb19f1cbc Void
   +0x02c DebugActive      : 0 ''
```

1.4.2 调度链表

调度链表有32个，也就有32个表头，我们可以通过全局变量KiDispatcherReadyListHead找到。在这32个链表中，每个链表链入的线程的优先级是不一样的。

```
0: kd> dd KiDispatcherReadyListHead L70
8055cf60  8055cf60 8055cf60 8055cf68 8055cf68
8055cf70  8055cf70 8055cf70 8055cf78 8055cf78
8055cf80  8055cf80 8055cf80 8055cf88 8055cf88
8055cf90  8055cf90 8055cf90 8055cf98 8055cf98
8055cfa0  8055cfa0 8055cfa0 8055cfa8 8055cfa8
8055cfb0  8055cfb0 8055cfb0 8055cfb8 8055cfb8
8055cfc0  8055cfc0 8055cfc0 8055cfc8 8055cfc8
8055cfd0  8055cfd0 8055cfd0 8055cfd8 8055cfd8
8055cfe0  8055cfe0 8055cfe0 8055cfe8 8055cfe8
8055cff0  8055cff0 8055cff0 8055cff8 8055cff8
8055d000  8055d000 8055d000 8055d008 8055d008
8055d010  8055d010 8055d010 8055d018 8055d018
8055d020  8055d020 8055d020 8055d028 8055d028
8055d030  8055d030 8055d030 8055d038 8055d038
8055d040  8055d040 8055d040 8055d048 8055d048
8055d050  8055d050 8055d050 8055d058 8055d058
8055d060  00000000 00000000 00000000 00000000
8055d070  00000000 00000000 00000000 00000000
```

同样，在链表中被链入的地址为_ETHREAD（_KTHREAD）中的0x60偏移位成员SwapListEntry。（0x60偏移位有两个含义）

```
0: kd> dd _KTHREAD 8055cf60-0x60
Couldn't resolve error at '_KTHREAD 8055cf60-0x60'
0: kd> dd KiDispatcherReadyListHead
8055cf60  8055cf60 8055cf60 8055cf68 8055cf68
8055cf70  8055cf70 8055cf70 8055cf78 8055cf78
8055cf80  8055cf80 8055cf80 8055cf88 8055cf88
8055cf90  8055cf90 8055cf90 8055cf98 8055cf98
8055cfa0  8055cfa0 8055cfa0 8055cfa8 8055cfa8
8055cfb0  8055cfb0 8055cfb0 8055cfb8 8055cfb8
8055cfc0  8055cfc0 8055cfc0 8055cfc8 8055cfc8
8055cfd0  8055cfd0 8055cfd0 8055cfd8 8055cfd8
0: kd> dt _KTHREAD 8055cf60-0x60
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x8055cf10
   +0x018 InitialStack     : 0xb1a217a8 Void
   +0x01c StackLimit       : 0xb1a217a8 Void
   +0x020 Teb              : 0x8055cf20 Void
   +0x024 TlsArray         : 0x8055cf20 Void
```

1.5 线程切换

1.5.1 模拟线程切换

Windows下的线程切换是比较复杂的，为了更好的学习我们需要读一份代码，这份代码是用来进行模拟Windows线程切换的。

模拟线程结构体

模拟代码的第一部分就是要模拟线程结构体，我们保留最重要的成员进行模拟：

```
1  //线程信息的结构
2  typedef struct
3  {
4      const char* name;                    // 线程名
5      int Flags;                           // 线程状态
6      int SleepMillsecondDot;              // 休眠时间
7
8      void* initialStack;                  // 线程堆栈起始位置
```

```
    void* StackLimit;                   // 线程堆栈界限
    void* KernelStack;                  // 线程堆栈当前位置，也就是ESP

    void* lpParameter;                  // 线程函数的参数
    void(*func)(void* lpParameter);     // 线程函数
}GMThread_t;
```

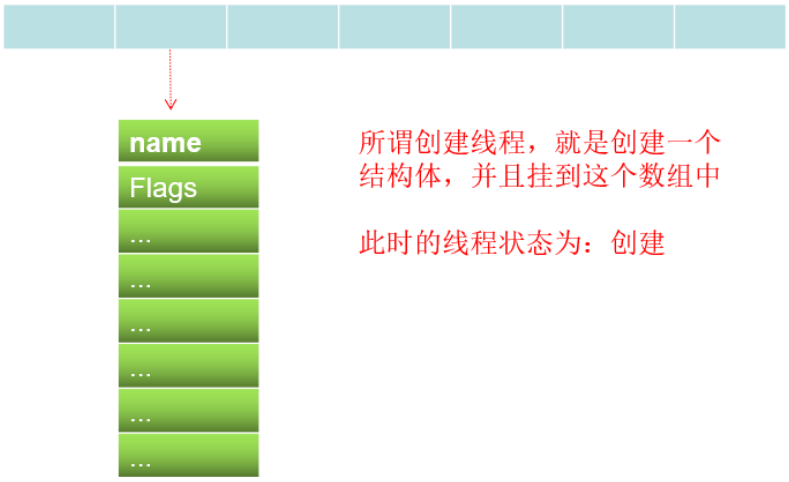
调度链表

在代码中有一个结构体数组，即线程结构体数组：

```
extern GMThread_t GMThreadList[MAXGMTHREAD];
```

我们知道线程有三种不同的状态，这些不同状态的线程根据优先级存储在不同的链表里，在这里我们模拟的没有那么复杂，只使用一个链表来存储所有状态的线程。

所谓创建线程，本质就是创建一个结构体，然后将结构体存到这个结构体数组中，我们可以根据线程结构体的Flags成员来判断线程的状态。



结构体存储的时候是从下标为1的索引中开始的，下标为0的索引中存储的是当前函数的线程信息。

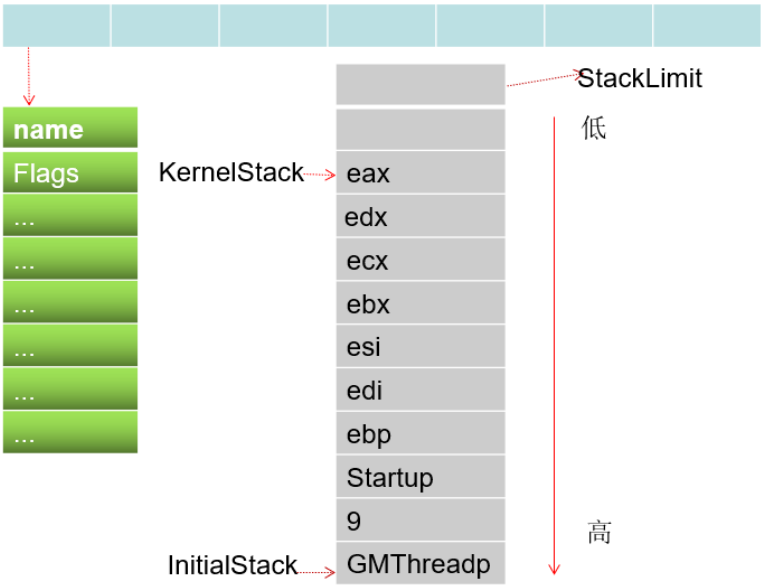
初始化线程

我们创建一个线程，就是创建结构体，接着我们初始化线程，就是要准备内存空间，并将结构体成员的值填充好，如下代码就完成了这些操作：

```
// 初始化线程的信息
void initGMThread(GMThread_t* GMThreadp, const char* name, void(*func)(void* lpParameter), void* lpParameter)
{
    unsigned char* StackPages;
    unsigned int* StackDWordParam;
    // 结构体初始化赋值：状态、名称、函数地址、函数参数
    GMThreadp->Flags = GMTHREAD_CREATE;
```

```
8     GMThreadp->name = name;
9     GMThreadp->func = func;
10    GMThreadp->lpParameter = lpParameter;
11    // 申请内存空间，大小为GMTHREADSTACKSIZE = 0x8000
12    StackPages = (unsigned char*)VirtualAlloc(NULL, GMTHREADSTACKSIZE,
      MEM_COMMIT, PAGE_READWRITE);
13    // 空间清零
14    ZeroMemory(StackPages, GMTHREADSTACKSIZE);
15    // 由于堆栈的操作是从高地址往低地址的，所以为了模拟这一效果，我们将将申请的内存空间
      地址加上初始化的大小就得到了线程的内存空间初始化地址
16    GMThreadp->initialStack = StackPages + GMTHREADSTACKSIZE;
17    StackDWordParam = (unsigned int*)GMThreadp->initialStack;
18    // 入栈
19    PushStack(&StackDWordParam, (unsigned int)GMThreadp); // 线程结构体，用
      于线程函数地址、函数参数地址的寻找
20    PushStack(&StackDWordParam, (unsigned int)9); // 平衡堆栈
21    PushStack(&StackDWordParam, (unsigned int)GMThreadStartup); // 启动线程
      的函数，用于调用线程函数
22    // 压入的对应寄存器初始值
23    PushStack(&StackDWordParam, (unsigned int)5); // ebp
24    PushStack(&StackDWordParam, (unsigned int)7); // edi
25    PushStack(&StackDWordParam, (unsigned int)6); // esi
26    PushStack(&StackDWordParam, (unsigned int)3); // ebx
27    PushStack(&StackDWordParam, (unsigned int)2); // ecx
28    PushStack(&StackDWordParam, (unsigned int)1); // edx
29    PushStack(&StackDWordParam, (unsigned int)0); // eax
30    // 设置当前线程的栈顶
31    GMThreadp->KernelStack = StackDWordParam;
32    // 设置线程状态为准备
33    GMThreadp->Flags = GMTHREAD_READY;
34    return;
35 }
```

这段代码执行完成之后的内存空间分布就如下：



线程切换

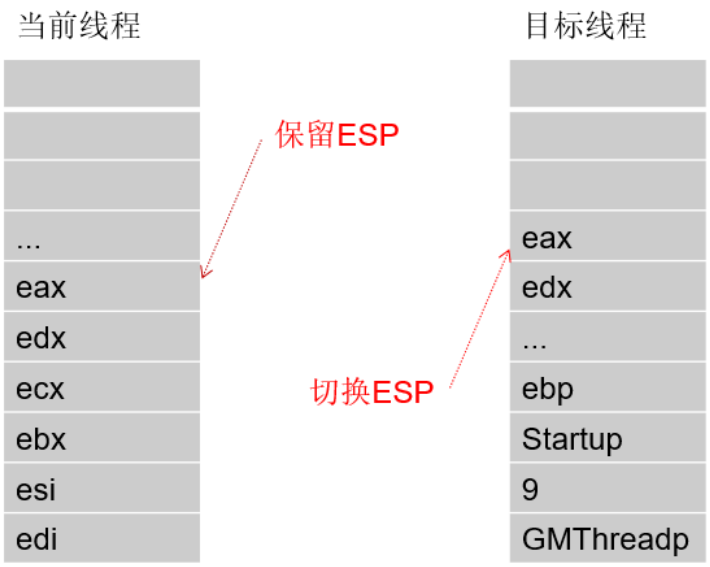
初始化线程之后，也就是执行完RegisterGMThread函数，就进入了线程切换运行，即Scheduling函数。该函数遍历调度链表，找到其中的线程结构体，并根据Flags成员判断线程是否为准备状态，获取对应的线程结构体通过SwitchContext函数进行线程切换。

```
1   void Scheduling(void)
2   {
3       int i;
4       int TickCount;
5       GMThread_t* SrcGMThreadp;
6       GMThread_t* DstGMThreadp;
7       TickCount = GetTickCount();
8       SrcGMThreadp = &GMThreadList[CurrentThreadIndex];
9       DstGMThreadp = &GMThreadList[0];
10
11      for (i = 1; GMThreadList[i].name; i++) {
12          // 判断休眠状态的线程休眠时间是否小于程序启动至今的时间，如果是则将线程状态调整为等待
13          if (GMThreadList[i].Flags & GMTHREAD_SLEEP) {
14              if (TickCount > GMThreadList[i].SleepMillsecondDot) {
15                  GMThreadList[i].Flags = GMTHREAD_READY;
16              }
17          }
18          if (GMThreadList[i].Flags & GMTHREAD_READY) {
19              DstGMThreadp = &GMThreadList[i]; // 获取准备状态的线程结构体
20              break;
21          }
22      }
23
24      CurrentThreadIndex = DstGMThreadp - GMThreadList;
```

```
    SwitchContext(SrcGMThreadp, DstGMThreadp);
    return;
}
```

接着我们来看线程切换的函数SwitchContext，一开始进行堆栈的提升然后将当前线程用到的寄存器入栈，接着分别给ESI、EDI存入当前和要切换的线程结构体地址，并将当前的栈顶即ESP寄存器给到当前线程结构体的KernelStack成员，然后就是经典的线程切换操作，将当前的栈顶指向要切换的线程结构体成员KernelStack，再将其对应存入的结构体进行弹出，**最后RET指令最为精巧，将启动线程的函数地址给到EIP，这样下一次要执行的函数就是启动线程的函数**：

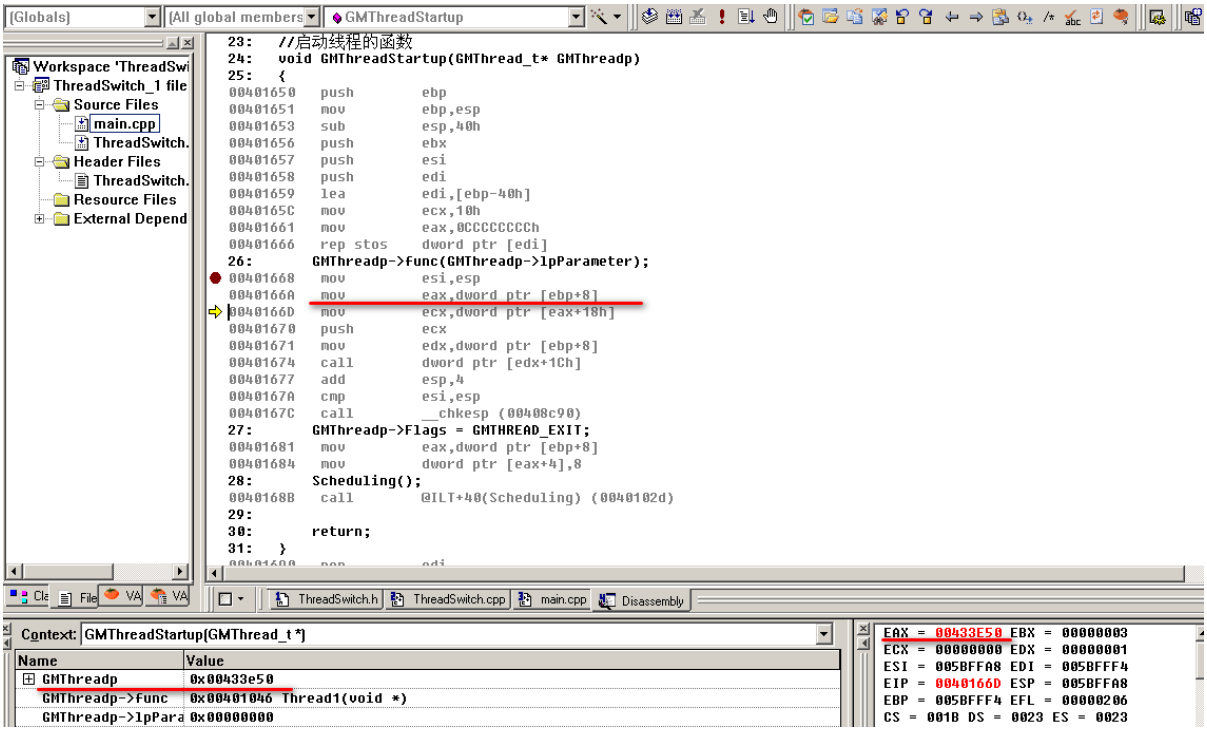
```
// 切换线程
__declspec(naked) void SwitchContext(GMThread_t* SrcGMThreadp, GMThread_t*
DstGMThreadp)
{
    __asm {
        // 提升堆栈
        push ebp
        mov ebp, esp
        // 当前线程用到的寄存器入栈
        push edi
        push esi
        push ebx
        push ecx
        push edx
        push eax
        // 将当前线程结构体和要切换的线程结构体分别存入ESI和EDI寄存器
        mov esi, SrcGMThreadp
        mov edi, DstGMThreadp
        // 把当前线程的栈顶存入到结构体的KernelStack成员中
        mov [esi + GMThread_t.KernelStack], esp
        // 经典线程切换，另外一个线程复活
        // 将要切换的线程结构体成员KernelStack给到ESP寄存器（栈顶），也就是栈的切换
        mov esp, [edi + GMThread_t.KernelStack]
        // 分别弹出要切换的线程结构体初始化的寄存器值
        pop eax
        pop edx
        pop ecx
        pop ebx
        pop esi
        pop edi
        pop ebp
        ret // 将启动线程的函数地址给到EIP
    }
}
```



启动线程的函数如下，我们可以看到它使用线程结构体的func成员函数，传递参数为lpParameter成员，这样我们就可以顺利的去执行对应线程的函数，最后将线程的状态设为退出状态，再进入Scheduling函数进行线程切换：

```
1  void GMThreadStartup(GMThread_t* GMThreadp)
2  {
3      GMThreadp->func(GMThreadp->lpParameter);
4      GMThreadp->Flags = GMTHREAD_EXIT;
5      Scheduling();
6
7      return;
8  }
```

这里实际上有一个细节，**虽然当RET指令将启动线程的函数地址弹给了EIP，但是启动线程函数是有参数的，这个参数就是一个线程结构体，而这里我们并没有进行参数的传递，那么它是如何找到参数的呢？**实际上我们可以通过反汇编来看一下，通过反汇编我们可以看到即使你没有传递参数，但是在反汇编的指令层它会通过[EBP+8]的方式去取参数。

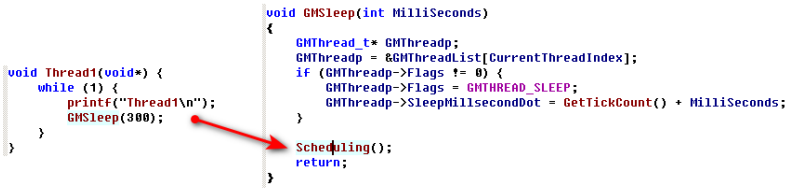


所以我们在初始化线程的代码中，向堆栈中随便压入了一个值，用于平衡堆栈，这样就可以确保当执行该函数时能顺利的通过[EBP+8]的方式来取到线程结构体，然后再找到对应的成员函数、函数参数进行调用。

```
1   PushStack(&StackDWordParam, (unsigned int)9); // 平衡堆栈
```

总结

1. 线程不是被动切换的，而是主动的，这点我们在线程函数的实现代码上有体现，即每个线程函数都调用了线程切换的函数



2. 线程切换并没有通过TSS来保护寄存器，而是通过堆栈的方式；
3. 线程的切换过程本质上就是堆栈的切换过程。

1.5.2 主动切换

在线程模拟的代码中，有一个重要的函数SwitchContext，它是用于线程切换的，而对应在Windows上有着类似功能的函数就是KiSwapContext。

我们可以通过IDA打开Ntoskrnl.exe内核模块找到该函数，来看一下它的作用，根据反汇编我们可以看见，该函数一开始就是保存当前线程所使用的寄存器，然后根据_KPCR取出_KTHREAD结构体，并且从父函数传递的参

数ECX中拿到要切换的线程结构体地址，**将该地址替换至_KPCR结构体中用于当前线程的_KTHREAD结构体成员的位置**。

```
.text:00405D48                          ; __fastcall KiSwapContext(x)
.text:00405D48                          @KiSwapContext@4 proc near              ; CODE XREF: KiSwapThread()+33↓p
.text:00405D48
.text:00405D48                          var_10= dword ptr -10h
.text:00405D48                          var_C= dword ptr -0Ch
.text:00405D48                          var_8= dword ptr -8
.text:00405D48                          var_4= dword ptr -4
.text:00405D48
.text:00405D48 83 EC 10                 sub     esp, 10h
.text:00405D4B 89 5C 24 0C              mov     [esp+10h+var_4], ebx            ; 保存当前线程寄存器现场
.text:00405D4F 89 74 24 08              mov     [esp+10h+var_8], esi
.text:00405D53 89 7C 24 04              mov     [esp+10h+var_C], edi
.text:00405D57 89 2C 24                 mov     [esp+10h+var_10], ebp
.text:00405D5A 64 8B 1D 1C 00 00 00     mov     ebx, large fs:1Ch               ; 取_KPCR结构体地址
.text:00405D61 8B F1                    mov     esi, ecx                        ; ECX给到ESI，即要切换线程的_KTHREAD结构体
.text:00405D63 8B BB 24 01 00 00        mov     edi, [ebx+124h]                 ; 从KPCR中取出当前线程的_KTHREAD结构体（即_KPRCB的0x4偏移位成员）
.text:00405D69 89 B3 24 01 00 00        mov     [ebx+124h], esi                 ; 修改_KPCR结构体中当前线程的_KTHREAD结构体为要切换的线程的结构体
.text:00405D6F 8A 4F 58                 mov     cl, [edi+58h]
.text:00405D72 E8 00 01 00 00           call    SwapContext                     ; 进入线程切换
```

从父函数传递的参数ECX，我们可以跟进看一下KiSwapThread函数，如下图所示ECX是通过EAX得来的，而EAX是通过KiFindReadyThread函数返回的，该函数从字面意思就是寻找准备就绪的线程，返回的自然就是一个线程结构体了：

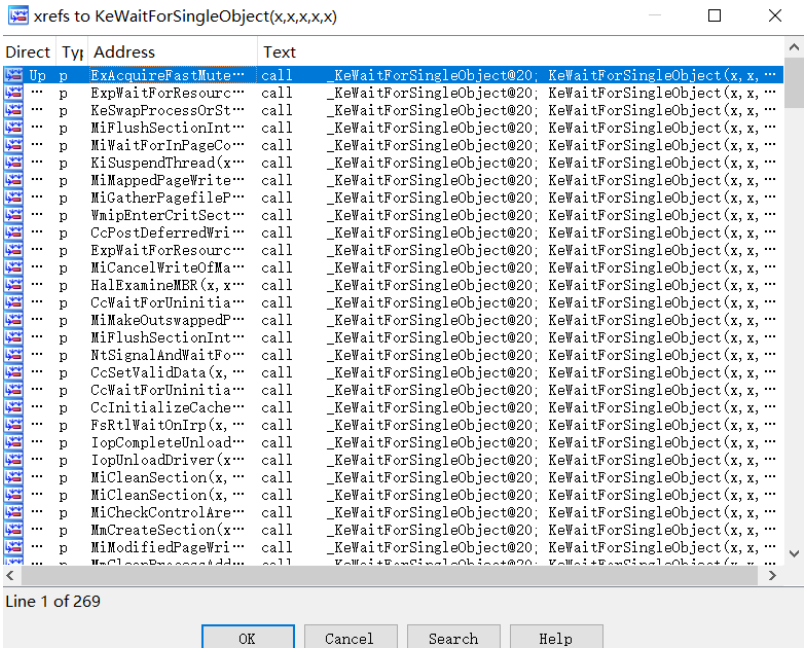
```
0040B1DE E8 9B A5 FF FF         call    @KiFindReadyThread@8            ; KiFindReadyThread(x,x)
0040B1DE
0040B1E3 85 C0                  test    eax, eax
0040B1E5 0F 84 39 5D 00 00      jz      loc_410F24
0040B1E5
0040B1EB
0040B1EB                        loc_40B1EB:                             ; CODE XREF: KiSwapThread()+5D89↓j
0040B1EB                                                                ; KiSwapThread()+5DA2↓j
0040B1EB 5B                     pop     ebx
0040B1EB
0040B1EC
0040B1EC                        loc_40B1EC:                             ; CODE XREF: KiSwapThread()+5F1F↓j
0040B1EC 8B C8                  mov     ecx, eax
0040B1EE E8 55 AB FF FF         call    @KiSwapContext@4                ; KiSwapContext(x)
```

接着我们回到KiSwapContext，向下走就会发现真正的线程切换是在SwapContext函数内，跟进该函数我们可以先忽略其他的一些细节，关注最重要的堆栈切换那一部分，我们可以清晰的看见这里的操作与之前的模拟线程切换一样，将当前的ESP存入当前线程的结构体中，并且将要切换线程的KernelStatck成员给到当前ESP。

```
.text:00405EC7 FA                       cli
.text:00405EC8 89 67 28                 mov     [edi+28h], esp                  ; 将当前的栈顶给到当前线程的_KTHREAD结构体中的KernelStack成员
.text:00405ECB 8B 46 18                 mov     eax, [esi+18h]
.text:00405ECE 8B 4E 1C                 mov     ecx, [esi+1Ch]
.text:00405ED1 2D 10 02 00 00           sub     eax, 210h
.text:00405ED6 89 4B 08                 mov     [ebx+8], ecx
.text:00405ED9 89 43 04                 mov     [ebx+4], eax
.text:00405EDC 33 C9                    xor     ecx, ecx
.text:00405EDE 8A 4E 31                 mov     cl, [esi+31h]
.text:00405EE1 83 E2 F1                 and     edx, 0FFFFFFF1h
.text:00405EE4 0B CA                    or      ecx, edx
.text:00405EE6 0B 88 0C 02 00 00        or      ecx, [eax+20Ch]
.text:00405EEC 3B E9                    cmp     ebp, ecx
.text:00405EEE 0F 85 DD 00 00 00        jnz     loc_405FD1
.text:00405EEE
.text:00405EF4 8D 09                    lea     ecx, [ecx]
.text:00405EF4
.text:00405EF6
.text:00405EF6                          loc_405EF6:                             ; CODE XREF: SwapContext+15D↓j
.text:00405EF6 F7 40 E4 00 00 02 00     test    dword ptr [eax-1Ch], 20000h
.text:00405EFD 75 03                    jnz     short loc_405F02
.text:00405EFD
.text:00405EFF 83 E8 10                 sub     eax, 10h
.text:00405EFF
.text:00405F02
.text:00405F02                          loc_405F02:                             ; CODE XREF: SwapContext+86↑j
.text:00405F02 8B 4B 40                 mov     ecx, [ebx+40h]
.text:00405F05 89 41 04                 mov     [ecx+4], eax
.text:00405F08 8B 66 28                 mov     esp, [esi+28h]                  ; 将要切换线程的_KTHREAD结构体的KernelStack成员给到ESP寄存器，即堆栈切换
.text:00405F0B 8B 46 20                 mov     eax, [esi+20h]
```

总结

其实在内核函数中大多都用到了线程切换，我们可以根据IDA的XREF来看调用链，最终看见有很多个内核函数调用了线程切换：



在线程切换时会比较是否属于同一个进程，如果不是同一个进程，就会切换Cr3，这样对应进程也就切换了，**因此进程的切换实际上也是线程的切换**。

1.5.3 时钟中断切换

之前我们了解到大多数内核函数都最终会调用到SwapContext来进行线程切换，但如果有线程没有去调用系统API是否就不会进行线程切换呢？其实不然，时钟中断也会导致线程切换。

如下就是系统时钟中断对应的信息，在Windows操作系统中，每10~20毫秒便会触发一次时钟中断，要想获取当前版本Windows时钟间隔值，可使用GetSystemTimeAdjustment函数：

（IDT表）中断号	IRQ号	说明
0x30	IRQ0	时钟中断

执行流程

我们可以通过IDA打开Ntoskrnl.exe内核模块，找到IDT表中0x30中断号对应的处理函数，也就是_KiStartUnexpectedRange函数：

```
INIT:005EFEC0 00 EE 08 00                dd 8EE00h
INIT:005EFEC4 36 6D 40 00                dd offset _KiSystemService
INIT:005EFEC8 00 EE 08 00                dd 8EE00h
INIT:005EFECC 75 A1 40 00                dd offset _KiTrap0F
INIT:005EFED0 00 8E 08 00                dd 88E00h
INIT:005EFED4 F0 63 40 00                dd offset _KiStartUnexpectedRange@0    ; KiStartUnexpectedRange()
INIT:005EFED8 00 8E 08 00                dd 88E00h
INIT:005EFEDC FA 63 40 00                dd offset _KiUnexpectedInterrupt1
```

我们跟进这个函数发现有两个跳转，先是走到_KiEndUnexpectedRange函数，最终是走到了_KiUnexpectedInterruptTail函数：

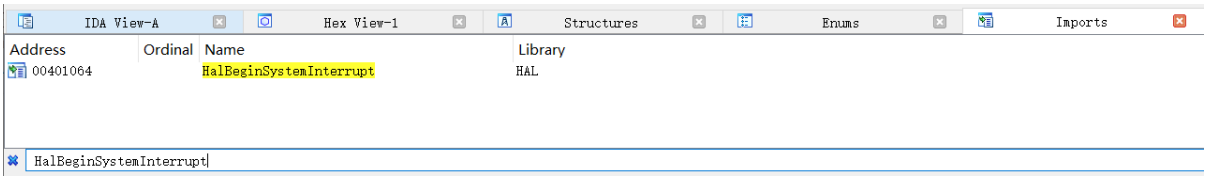
```
.text:004063F0 68 30 00 00 00              push    30h ; '0'
.text:004063F5 E9 DD 07 00 00              jmp     _KiEndUnexpectedRange@0         ; KiEndUnexpectedRange()

.text:00406BD7 2E FF 25 DE 6B 40 00        jmp     cs:off_406BDE
.text:00406BD7
.text:00406BD7                             _KiEndUnexpectedRange@0 endp
.text:00406BD7
.text:00406BD7                             ; ---------------------------------------------------------
.text:00406BDE E6 76 40 00                 off_406BDE dd offset _KiUnexpectedInterruptTail
```

在这个函数里，我们可以看见它在函数内又调用了一个导入表的函数HalBeginSystemInterrupt和HalEndSystemInterrupt：

```
64 FF 05 C4 05 00 00          inc     large dword ptr fs:5C4h
8B 1C 24                      mov     ebx, [esp+68h+var_68]
83 EC 04                      sub     esp, 4
54                            push    esp
53                            push    ebx
6A 1F                         push    1Fh
FF 15 64 10 40 00             call    ds:__imp__HalBeginSystemInterrupt@12 ; HalBeginSystemInterrupt(x,x,x)
```

```
.text:004077AE
.text:004077AE                             loc_4077AE:                             ; CODE XREF: _KiUnexpectedInterruptTail+C1↑j
.text:004077AE FA                          cli
.text:004077AF FF 15 68 10 40 00           call    ds:__imp__HalEndSystemInterrupt@8 ; HalEndSystemInterrupt(x,x)
```

我们可以在IDA的Imports导入表窗口中找到该导入函数对应的模块，也就是如下图所示的HAL模块：




```
.text:80012F48 C7 05 80 00 FE FF 41 00 00 00 mov     dword ptr ds:0FFFE0080h, 41h ; 'A'
.text:80012F52 53                          push    ebx
.text:80012F53 51                          push    ecx
.text:80012F54 FB                          sti
.text:80012F55 64 C6 05 96 00 00 00 00     mov     large byte ptr fs:96h, 0
.text:80012F5D FF 15 30 04 01 80           call    ds:__imp__KiDispatchInterrupt@0 ; KiDispatchInterrupt()
.text:80012F5D
.text:80012F63 FA                          cli
```

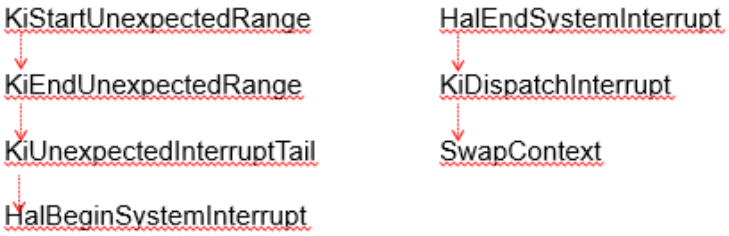
跟进HAL模块（HAL.DLL文件在C:/Windows/System32目录下）在HalEndSystemInterrupt函数内，我们可以看到它又调用了KiDispatchInterrupt函数：

```
mov     large byte ptr fs:96h, 0
call    ds:__imp__KiDispatchInterrupt@0 ; KiDispatchInterrupt()
```

再回到Ntoskrnl.exe内核模块，跟进KiDispatchInterrupt函数你会发现它最终又调用了SwapContext函数，因此也就证明了时钟中断也会导致线程切换：

```
.text:00405E2C E8 56 FD FF FF              call    @KiReadyThread@4                ; KiReadyThread(x)
.text:00405E2C
.text:00405E31 B1 01                       mov     cl, 1
.text:00405E33 E8 3F 00 00 00              call    SwapContext
```

我们简单梳理一下，系统时钟中断的执行流程图如下：



总结

线程切换的几种情况：

1. 主动调用API函数（SwapContext）
2. 时钟中断
3. 异常处理（缺页、INT N指令）

如果一个线程不调用API，在代码中屏蔽中断（CLI指令），并且不会出现异常，那么当前线程将永久占有CPU，单核占有率100%，2核就是50%。

1.5.4 时间片管理与备用线程

时钟中断导致线程进行切换也是有前置条件的，**第一是当前的线程CPU时间片到期，第二是有备用的线程（即_KPCR.PrcbData.NextThread存储了另外的线程地址）**，以上两种情况时候发生的时间中断才会进行线程切换。

CPU时间片到期

时间片就是CPU分配给各个程序的时间，每个进程被分配一个时间段，称作它的时间片，表示该进程允许运行的时间，使各个程序从表面上看是同时进行的。 如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换，这样就不会造成CPU资源浪费。

当一个新的线程开始执行，初始化程序会给_KTHREAD结构体的Quantum成员赋予初始值，该值的大小由_KPROCESS结构体的ThreadQuantum成员决定。如下图所示我们找一个进程来看下结构体的ThreadQuantum成员，它的值为6，也就表示当该进程的线程开始执行，初始化程序会将该值赋予_KTHREAD结构体的Quantum成员，该值就表示当前线程时间片的大小。

```
0: kd> dt _KPROCESS 890d8020
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY [ 0x890d8030 - 0x890d8030 ]
   +0x018 DirectoryTableBase : [2] 0xa3c0240
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : 0x20ac
   +0x032 Iopl             : 0 ''
   +0x033 Unused           : 0 ''
   +0x034 ActiveProcessors : 0
   +0x038 KernelTime       : 1
   +0x03c UserTime         : 1
   +0x040 ReadyListHead    : _LIST_ENTRY [ 0x890d8060 - 0x890d8060 ]
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : (null)
   +0x050 ThreadListHead   : _LIST_ENTRY [ 0x897721d0 - 0x897721d0 ]
   +0x058 ProcessLock      : 0
   +0x05c Affinity         : 3
   +0x060 StackCount       : 1
   +0x062 BasePriority     : 4 ''
   +0x063 ThreadQuantum    : 6 ''
   +0x064 AutoAlignment    : 0 ''
```

每次时钟中断会调用KeUpdateRunTime函数，该函数每次执行就会将当前线程结构体成员Quantum的值减少3个单位，如果减到0则将_KPCR.PrcbData.QuantumEnd的值设置为非0来表示当前时间片已经到期了，我们可以通过IDA来看到这一过程。

```
64 A1 1C 00 00 00           mov     eax, large fs:1Ch           ; 获取_KPCR
53                          push    ebx
FF 80 C4 05 00 00           inc     dword ptr [eax+5C4h]
8B 98 24 01 00 00           mov     ebx, [eax+124h]             ; 获取当前线程的_KTHREAD结构体

80 6B 6F 03                 sub     byte ptr [ebx+6Fh], 3       ; 将_KTHREAD的Quantum的值减3
7F 19                       jg      short loc_40BF40            ; 如果_KTHREAD的Quantum的值不为0则跳转

3B 98 2C 01 00 00           cmp     ebx, [eax+12Ch]
74 11                       jz      short loc_40BF40

89 A0 AC 09 00 00           mov     [eax+9ACh], esp             ; 将_KPCR.PrcbData.QuantumEnd的值设为一个不为0的值
```

系统时钟执行完毕之后都会去调用KiDispatchInterrupt函数，这个函数用于判断当前线程的时间片是否到期，我们可以来看一下该函数的执行流程，先判断_KPCR.PrcbData.QuantumEnd的值，当值为非0时进行跳转，接着修改该值为0继而调用_KiQuantumEnd函数，在这个函数里就是重新设置_KTHREAD结构体的Quantum成员为6，然后进入到KiFindReadThread函数寻找下一个就绪状态的线程。

```
mov     ebx, large fs:1Ch               ; 获取_KPCR结构体

cmp     dword ptr [ebx+9ACh], 0         ; 判断_KPCR.PrcbData.QuantumEnd是否为0，即判断时间片是否到期
jnz     loc_405E4F                      ; 时间片到期进行跳转

mov     dword ptr [ebx+9ACh], 0         ; 修改_KPCR.PrcbData.QuantumEnd的值为0
call    _KiQuantumEnd@0                 ; KiQuantumEnd()

mov     eax, large fs:124h              ; 获取当前线程的_KTHREAD结构体
xor     ecx, ecx
mov     esi, eax

mov     al, [eax+63h]                   ; 重新设置时间片
mov     [esi+6Fh], al

call    @KiFindReadyThread@8            ; KiFindReadyThread(x,x)
```

以上流程结束之后就返回到KiDispatchInterrupt函数了，接着跟进发现将_KPCR.PrcbData.CurrentThread设为下一个就绪状态的线程，并且把当前线程挂到调度链表中，最终进行堆栈的切换（线程切换）

```
                                        or      eax, eax
                                        jnz     short loc_405E00
                                                  ↓
mov     esi, eax                        ; 将找到的下一个就绪状态的线程存入ESI
mov     edi, [ebx+124h]                 ; 将当前线程_KPCR.PrcbData.CurrentThread存入到EDI
mov     dword ptr [ebx+128h], 0         ; 将_KPCR.PrcbData.NextThread设为0
mov     [ebx+124h], esi                 ; 将当前线程_KPCR.PrcbData.CurrentThread设为下一个就绪状态的线程
mov     ecx, edi
mov     byte ptr [edi+50h], 1
call    @KiReadyThread@4                ; KiReadyThread(x)
                                                  ↓
        add     esi, 60h ; '`'                  ; 将ESI指向_ETHREAD[0x60]，即SwapListEntry调度链表
        cmp     [ebp+var_1], 0
        lea     ecx, _KiDispatcherReadyListHead[eax*8] ; 挂到调度链表中
                                                  ↓
                                        mov     cl, 1
                                        call    SwapContext
```

以上就是大致的CPU时间片到期的执行流程。

备用线程

除了CPU时间片到期的情况，还有存在备用线程的情况也会进行线程切换，我们可以继续分析KiDispatchInterrupt这个函数，回到最开始，我们可以看到即使你的CPU时间片没有过期，但如果_KPCR.PrcbData.NextThread不为0，也就是存在备用线程，也会进行行线程切换。

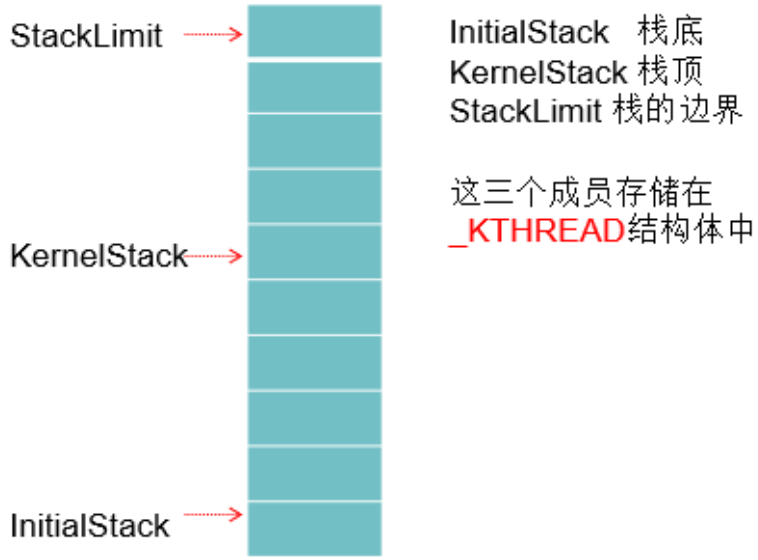
```
  cmp     dword ptr [ebx+128h], 0         ; 判断_KPCR.PrcbData.NextThread是否为0，即判断是否有备用线程
  jz      short locret_405E46
                                                  ↓
loc_405E00:                               ; CODE XREF: KiDispatchInterrupt()+CF↓j
sub     esp, 0Ch
mov     [esp+0Ch+var_4], esi
mov     [esp+0Ch+var_8], edi
mov     [esp+0Ch+var_C], ebp
mov     esi, eax                          ; 将找到的下一个就绪状态的线程存入ESI
mov     edi, [ebx+124h]                   ; 将当前线程_KPCR.PrcbData.CurrentThread存入到EDI
mov     dword ptr [ebx+128h], 0           ; 将_KPCR.PrcbData.NextThread设为0
mov     [ebx+124h], esi                   ; 将当前线程_KPCR.PrcbData.CurrentThread设为下一个就绪状态的线程
mov     ecx, edi
mov     byte ptr [edi+50h], 1
call    @KiReadyThread@4                  ; KiReadyThread(x)

mov     cl, 1
call    SwapContext
```

1.5.5 线程切换与TSS的关系

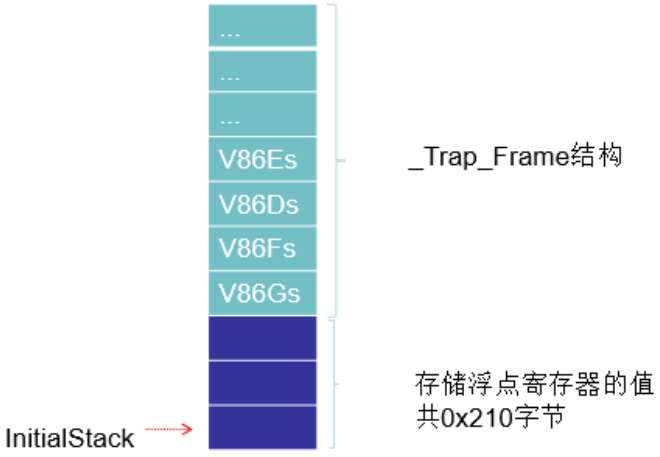
SwapContext这个函数是Windows线程切换的核心，无论是主动切换还是系统时钟导致的线程切换，最终都会调用这个函数。在这个函数中除了切换堆栈以外，还做了很多其他的事情，了解这些细节对我们学习操作系统至关重要。这节课我们讲一下线程切换与TSS的关系。

内核堆栈

每一个线程都有一个内核堆栈，在系统调用章节时我们了解到使用API从3环进0环都要进行一个堆栈的切换，进0环所切换的堆栈就是内核堆栈。在线程结构体_KTHREAD中有关于内核堆栈的相关信息（栈顶、栈底、栈边界）。



内核堆栈的结构大致分为两个部分。第一部分就是从InitialStack开始一共0x210个字节，存储的就是浮点寄存器的值；剩下的第二部分就是我们之前了解过的_Trap_Frame结构体。



_Trap_Frame结构体在系统调用章节已经学过了，它的结构成员如下：

+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	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	
+0x068 Eip	中断发生时，保存被中断的代码段和地址，iret 返回到此地址
+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	

调用API进0环

API进0环有两种方式，分别为普通调用和快速调用。普通调用通过API进0环后会从TSS的ESP0得到0环的堆栈；快速调用则是先从MSR得到一个临时的0环栈来提供代码执行的环境，但是这段代码的执行实际上就是从TSS的ESP0得到0环的堆栈。

我们可以通过IDA来看一下_KiFastCallEntry函数，它先通过_KPCR找到TSS，而后通过TSS的ESP0得到0环的堆栈：

```
mov     ecx, 23h ; '#'
push    30h ; '0'
pop     fs
mov     ds, ecx
mov     es, ecx
mov     ecx, large fs:40h               ; _KPCR 偏移 0x40 -> TSS
mov     esp, [ecx+4]                    ; TSS 偏移 0x4 -> ESP0
push    23h ; '#'
push    edx
pushf
```

所以我们需要明白一个事情，无论是什么调用方式进入0环都是要通过TSS来切换堆栈的。

TSS

Intel设计TSS的目的是为了任务切换（线程切换），但是Windows和Linux都没有使用，而是采用堆栈来保护线程的各种寄存器。

一个CPU只有一个TSS，但是线程是有很多个的，每个线程在0环对应的堆栈是不一样的，那么到底是什么样的方式能让一个TSS来保存所有线程的ESP0呢？这里的细节在线程切换中，因此我们需要分析SwapContext函数。

SwapContext分析

我们可以通过IDA来分析一下SwapContext函数，重点关注TSS部分，首先是取出TSS然后将EAX给到TSS的ESP0。

```
loc_405F02:                             ; CODE XREF: SwapContext+86↑j
mov     ecx, [_KPCR+40h]                ; 取出TSS
mov     [ecx+4], eax                    ; 将TSS中ESP0修改为修正后的堆栈
mov     esp, [_ETHREAD+28h]             ; 将目标线程的ESP存储到当前ESP中
```

那么这个EAX又是从何而来呢，我们可以向上找一下，EAX的值是目标线程的栈底，并且将堆栈中存储的浮点寄存器和_Trap_Frame结构体中用于虚拟8086模式下的成员去除，也就是进行偏移位的修正。

```
mov     [edi+28h], esp
mov     eax, [_ETHREAD+18h]             ; 将目标线程的栈底（InitialStack）给到EAX
mov     ecx, [_ETHREAD+1Ch]
sub     eax, 210h                       ; EAX减去0x210，是因为从栈底开始的0x210个字节存储的是浮点寄存器，此时目标线程的栈底就指向了_Trap_Frame
mov     [_KPCR+8], ecx
mov     [_KPCR+4], eax
xor     ecx, ecx
mov     cl, [_ETHREAD+31h]
and     edx, 0FFFFFFF1h
or      ecx, edx
or      ecx, [eax+20Ch]
cmp     ebp, ecx
jnz     loc_405FD1

lea     ecx, [ecx]


loc_405EF6:                             ; CODE XREF: SwapContext+15D↓j
test    dword ptr [eax-1Ch], 20000h
jnz     short loc_405F02

sub     eax, 10h                        ; 再减去0x4，也就是减去_Trap_Frame结构体成员，这些都是给到虚拟8086模式下的使用的成员
```

从_Trap_Frame结构来看，目标线程的栈底（EAX）就指向了HardwareSegSs成员的位置，当然这是快速调用时所指向的位置，如果是普通调用（通过中断门的方式）由于CPU会帮我们压入5个寄存器值，所以栈底是指向ErrCode成员的。

除了ESP0之外，还有两个TSS的成员会被使用到。一个是TSS的Cr3会被修改为目标进程的Cr3；另外一个是之前学习中没有提到的IO权限位图，是将当前线程的IO权限位图存到TSS的0x66偏移位成员，但是这个在Windows2000以后已经不再使用了。（**自行了解IO权限位图**）

```
mov     ebp, [_KPCR+40h]                ; 取出TSS
mov     ecx, [edi+30h]                  ; 将TSS的Cr3修改为目标进程的Cr3
mov     [ebp+1Ch], eax
mov     cr3, eax
mov     [ebp+66h], cx                   ; 存储当前线程的IO权限位图到TSS（Windows2000以后不用了）
```

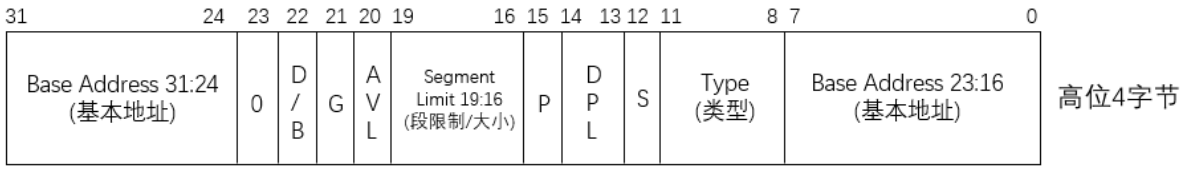
那么通过以上的分析，我们发现虽然Intel的初衷是希望操作系统用TSS去存储更多内容，但是在Windows的实际实现中只用到了TSS的ESP0、Cr3、IO权限位图这3个成员（实际对于当前的Windows系统来说只有2个）来进行线程切换。

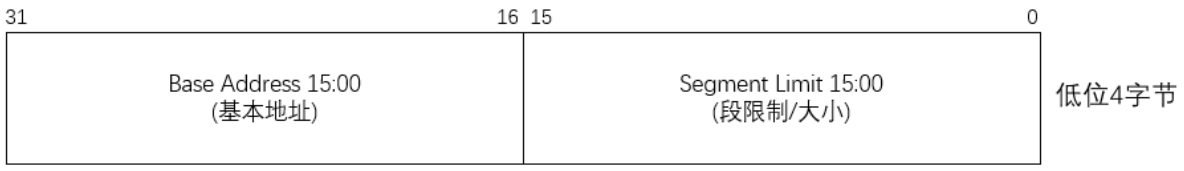
1.5.6 线程切换与FS寄存器的关系

我们知道在3环下FS:[0]指向的是TEB，而在0环下FS:[0]指向的是_KPCR。**在系统中同时会有很多个线程运行，这就意味着FS:[0]需要存储的不同线程的相关信息，但是在实际的使用中发现3环下不同线程的FS寄存器（段选择子）都是一样的值，那么到底是什么样的一个实现能让同一个寄存器指向多个不同的TEB呢**，要了解其中的细节我们还是需要来分析SwapContext这个函数的代码。

如下我们找到与FS寄存器有关系的代码段，可以看出它显示取出目标线程的TEB地址，接着取出GDT表，并且向表中的成员写入值。我们知道FS在Windows XP下的段选择子是0x3B，也就表示它的段描述符是在GDT表的第7个，即0x38偏移位，再结合如下的代码，我们可以根据段选择子的结构体，**发现它实际上就是将目标线程的TEB地址写入到FS段描述符的Base Address成员位中**。

```
loc_405F69:                             ; CODE XREF: SwapContext+E2↑j
mov     eax, [_KPCR+18h]                ; 取出目标线程TEB（_KPCR.NtTib.Self）
mov     ecx, [_KPCR+3Ch]                ; 取出GDT表（_KPCR.GDT）
mov     [ecx+3Ah], ax
shr     eax, 10h
mov     [ecx+3Ch], al
mov     [ecx+3Fh], ah
```





综上所示，也就解答了我们的困惑，在多个线程中，FS选择子不变，依旧可以获取对应线程的信息。

1.5.7 线程优先级

我们发现在主动调用、时间片到期的情况下导致的线程切换，它们所用到的API都会通过KiFindReadyThread函数来找到下一个要切换的线程，那么这时候就抛出了一个问题：该函数是根据什么样的条件取寻找下一个切换的线程呢？

KiFindReadyThread就是根据调度链表去查询的线程，**它的查询优先级就是根据调度链表的级别（32个调度链表），按线程级别大小从高到低查找：31、30、29、28...**，如果在线程级别31的链表中找到了线程，那么查找就结束了。

但是如果每次都是按照这个从高到低的优先级寻找，从效率和性能来看是很差的，所以Windows就通过一个DWORD类型的变量（_KiReadySummary）来记录，当向调度链表中挂入或取出某个线程时，会先判断当前级别的链表是否为空，为空则在DWORD变量的对应偏移位的值设为0，反之不为空就设为1。

如下图所示，就意味着线程级别为30、29的链表中是有线程的，其他链表中没有线程。

如下图：

判断链表是否为空可以通过KiDispatcherReadyListHead全局变量找到调度链表，然后根据它的链表成员判断是否前后指向的地址都是同一个，且该地址与链表的成员地址一致，如果都是一致的则表示该链表为空。

```
0: kd> dd KiDispatcherReadyListHead
8055cf60  8055cf60 8055cf60 8055cf68 8055cf68
8055cf70  8055cf70 8055cf70 8055cf78 8055cf78
8055cf80  8055cf80 8055cf80 8055cf88 8055cf88
8055cf90  8055cf90 8055cf90 8055cf98 8055cf98
8055cfa0  8055cfa0 8055cfa0 8055cfa8 8055cfa8
8055cfb0  8055cfb0 8055cfb0 8055cfb8 8055cfb8
8055cfc0  8055cfc0 8055cfc0 8055cfc8 8055cfc8
8055cfd0  8055cfd0 8055cfd0 8055cfd8 8055cfd8
```

假设调度链表中都没有就绪状态的线程了，CPU就会取_KPCR.PrcbData.IdleThread即空闲线程去运行，这一点我们可以通过IDA来看一下KSwapThread调用KiFindReadyThread的前后代码来论证。

先是给了ESI一个_KPRCB的地址，接着调用KiFindReadyThread，比较返回结果是否为空，如果为空则进行跳转。

```
mov     edi, edi
push    esi
push    edi
mov     eax, large fs:20h
mov     esi, eax                        ; _KPRCB
mov     eax, [esi+8]
test    eax, eax
mov     edi, [esi+4]
jnz     loc_4110D6

push    ebx
movsx   ebx, byte ptr [esi+10h]
xor     edx, edx
mov     ecx, ebx
call    @KiFindReadyThread@8            ; KiFindReadyThread(x,x)

test    eax, eax
jz      loc_410F24
```

跟进跳转我们就能看见将空闲线程的值就赋值给了EAX，然后跳回之前的位置：

```
loc_410F24:                             ; CODE XREF: KiSwapThread()+2A↑j
mov     eax, [esi+0Ch]                  ; IdleThread
xor     edx, edx
inc     edx
mov     ecx, ebx
shl     edx, cl
or      ds:_KiIdleSummary, edx
mov     ecx, [esi+4D0h]
mov     edx, ecx
and     edx, ds:_KiIdleSummary
cmp     edx, ecx
jnz     loc_40B1EB
```

1.6 进程挂靠

进程为线程提供资源，也就是提供Cr3的值，Cr3中存储的就是页目录表的基址，我们有了Cr3也就表示知道了线程能访问的内存。

1.6.1 进程与线程的关系

在Win32的课程学习中，我们了解了进程的创建，知道真正运行代码的实际上是线程而不是进程。如下这个汇编指令就是线程运行的，当运行这段代码时CPU是如何解析0x12345678这个地址的呢？

```
1  MOV EAX, DWORD PTR DS:[0x12345678]
```

0x12345678是一个线性地址，CPU要解析这个线性地址就需要通过页目录表找到对应的物理页。**而要找到页目录表就需要通页目录表基址，也就表示我们需要Cr3寄存器**，Cr3寄存器值来源于当前的进程结构体（_KPROCESS.DirectoryTableBase）。**因此我们可以理解为进程提供给线程运行的空间环境（界定了哪些内存可以访问）。**

既然进程给线程提供了环境信息，线程也要能找到进程才能知道这些信息，因此在线程结构体中实际上也有成员是指向了进程结构体的。

在_ETHREAD结构体中有两处位置都指向了进程结构体，分别是_ETHREAD.ThreadsProcess（0x220偏移位）和_ETHREAD.Tcb.ApcState.Process（0x44偏移位）。

```
0: kd> dt _ETHREAD
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD

0: kd> dt _KTHREAD
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   ...
   +0x034 ApcState         : _KAPC_STATE
```

```
+0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
+0x220 ThreadsProcess   : Ptr32 _EPROCESS
+0x224 StartAddress     : Ptr32 Void

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
```

那么在实际的线程切换当中用到的到底是0x220偏移位成员还是0x44偏移位成员呢，我们可以来看一下SwapContext函数的实现。

从实现来看SwapContext使用的是0x44偏移位成员，即_ETHREAD.Tcb.ApcState.Process来进行线程间的所属进程比对。

```
mov     esi, eax                        ; 将找到的下一个就绪状态的线程存入ESI
mov     edi, [ebx+124h]                 ; 将当前线程_KPCR.PrcbData.CurrentThread存入到EDI

  mov     eax, [edi+44h]                ; 取出当前进程（_ETHREAD.Tcb.ApcState.Process）
  cmp     eax, [esi+44h]                ; 比较当前线程与目标线程是否属于同一个线程
  mov     byte ptr [edi+50h], 0
  jz      short loc_405F5E
```

如果所属进程不是同一个，则会从进程结构体中找到Cr3取出，最后进行Cr3切换。

```
mov     edi, [esi+44h]                  ; 将目标线程所属进程的结构体给到EDI

  mov     eax, [edi+18h]                ; 取出DirectoryTableBase，即Cr3
  mov     ebp, [_KPCR+40h]              ; 取出TSS
  mov     ecx, [edi+30h]                ; 将TSS的Cr3修改为目标进程的Cr3
  mov     [ebp+1Ch], eax
  mov     cr3, eax                      ; 切换Cr3
```

综上所述，我们就知道线程所需要的Cr3寄存器值是来源于_ETHREAD线程结构体0x44偏移位的成员。那么是否就表示0x220是多余的呢，其实不然，**0x220偏移位成员用于表示的是当前线程是哪个进程所创建，而0x44偏移位成员用于表示当前线程的资源（Cr3）是哪个进程所提供**，在一般情况下这两个偏移位成员都指向同一个进程。

1.6.2 Cr3的修改

正常情况下，Cr3的值是由_ETHREAD.Tcb.ApcState.Process提供，**但Cr3的值也可以改成和当前线程毫不相干的其他进程的DirectoryTableBase，这样的手法我们称之为进程挂靠**。

我们可以通过如下汇编指令来修改Cr3寄存器的值：

```
mov cr3, A.DirectoryTableBase
mov eax, dword ptr ds:[0x12345678] // A进程的0x12345678内存
mov cr3, B.DirectoryTableBase
mov eax, dword ptr ds:[0x12345678] // B进程的0x12345678内存
mov cr3, C.DirectoryTableBase
mov eax, dword ptr ds:[0x12345678] // C进程的0x12345678内存
```

进程挂靠的目的就是让当前线程可以访问其他进程的内存空间。

1.6.3 NtReadVirtualMemory函数分析

直接修改Cr3的方法如果遇到了线程切换就会变回去，我们可以来看一下Windows下的函数NtReadVirtualMemory，该函数的作用就是读取其他进程的内存，我们看一下它是如何实现的。

这个函数很复杂，我们只需要找关键的部分，也就是切换Cr3的那一步，追踪它的调用关系，最终在_KiSwapProcess函数找到了关键部分，调用链如下：

```
_MmCopyVirtualMemory -> _MiDoPoolCopy -> _KeStackAttachProcess ->
_KiAttachProcess -> _KiSwapProcess
```

在_KiSwapProcess函数中，我们发现它就是将Cr3修改为目标进程的DirectoryTableBase：

```
mov     ecx, large fs:40h
mov     edx, [esp+arg_0]                ; 取出_KPROCESS
xor     eax, eax
mov     gs, ax
mov     eax, [edx+18h]                  ; 取出DirectoryTableBase
mov     [ecx+1Ch], eax
mov     cr3, eax                        ; 修改Cr3
mov     ax, [edx+30h]
mov     [ecx+66h], ax
```

但是在这个函数调用之前，又将_ETHREAD.Tcb.ApcState.Process的值修改为要读取的进程结构体_KPROCESS地址，这样就避免了线程切换时导致的Cr3值还原回去：

```
cmp     [ebp+arg_C], eax
mov     [esi+44h], edi                  ; 将_ETHREAD.Tcb.ApcState.Process的值修改为要读取的进程结构体_KPROCESS地址
mov     byte ptr [esi+48h], 0
```

如果你不想要使用这个函数来跨进程读取内存，自己实现的话就需要切换Cr3后来关闭中断，并且在程序中不再使用导致线程切换的Windows API。

1.6.4 总结

正常情况下，当前线程使用的Cr3是由其所属进程提供的（_ETHREAD，0x44偏移位指定的_EPROCESS），正是因为如此，A进程中的线程只能访问A的内存。如果要让A进程中的线程能够访问B进程的内存，就必须要修改Cr3的值为B进程的页目录表基址（B.DirectoryTableBase），这就是所谓的进程挂靠。

1.7 跨进程读写

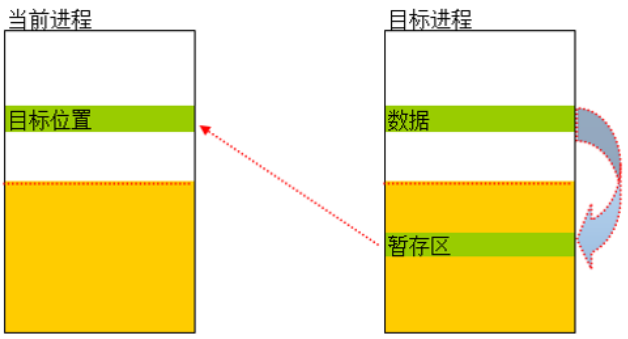
如下这段汇编指令是向通过A进程的线程执行，获取B进程的0x12345678的值再存入到0x00401234，这样操作很明显是无意义的，因为你操作了半天都只是将数据存入到了B进程的0x00401234，而A进程的0x00401234对应值并没有数据。

```
mov cr3, B.DirectoryTableBase // 切换Cr3的值为B进程
mov eax, dword ptr ds:[0x12345678] // 将B进程0x12345678的值存的eax中
mov dword ptr ds:[0x00401234], eax // 将数据存储到0x00401234中
mov cr3, A.DirectoryTableBase // 切换回Cr3的值
```

如果你还清晰的记得保护模式相关的内容就会想到一个进程有4GB内存空间，低2G是自己的，高2G是共享的，就可以利用高2G的内存空间来存储数据。

在Windows下的NtReadVirtualMemory、NtWriteVirtualMemory函数来跨进程读写就是利用的高2G内存空间来进行读写，具体流程如下：

NtReadVirtualMemory流程解析：



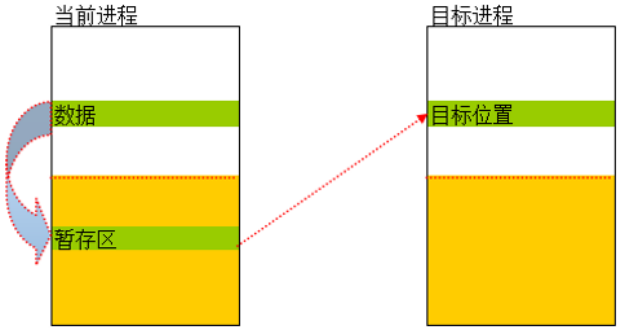
1、切换Cr3

2、将数据读复制到高2G

3、切换Cr3

4、从高2G复制到目标位置

NtWriteVirtualMemory流程解析：



1、将数据从目标位置复制到高2G地址

2、切换Cr3

3、从高2G复制到目标位置

4、切换Cr3