

- 00.系统组件
- 01.Windows NT操作系统的内存格局
- 02.Windows NT与FLAT模型
- 03.线程信息块 (THREAD INFORMATION BLOCK)
- 04.进程控制域 (PROCESSOR CONTROL REGION)

00.系统组件

=====

实际上,所有的Windows NT组件其本身都是DLL、PE格式的.EXE文件、导入导出函数。Windows NT的主要组件有:

*Ntoskrnl.exe

系统核心。系统执行函数都集中于此,这些函数又会调用其它组件。核心组件有:对象管理器、内存管理器、进线程创建、进线程控制、LPC、安全管理、异常处理、文件系统、输入输出、VDM、和T.D.一般都位于大于80100000h的地址上。

*Hal.dll

硬件抽象层(Hardware Abstraction Layer) — 硬件相关的模块。该隔离层将操作系统中硬件相关的部分隔离出来以增强系统的可移植性。主要的模块实现了非常底层的函数:程序控制中断、硬件输入输出等等。一般位于大于80100000h的地址上。

*Ntdll.dll

实现某些Win32 API函数。提供了核心模式与用户模式之间的接口。位于用户空间。换句话说,系统函数调用主要由这里导出。

*Kernel32.dll

实现了一些函数,类似于Win9x的核心。其中有很多的函数封装在ntdll.dll中。

*Csrss.exe

进程服务器子系统。其是一个单独的进程,因此受到保护以免受其它进程(位于其它进程的地址空间中)影响。对服务的请求要借助于LPC产生。

*Win32k.sys

驱动程序。用以减少调用Csrss服务开销损失。在此程序中实现了GDI和USER函数。不用LPC而用系统调用 — 这很明显提高了Windows NT4.0和2K的图形处理性能。

01.Windows NT Windows NT操作系统的内存格局

=====

在Windows NT中高2G的32位线性地址空间保存了供系统使用的程序。这种格局,地址空间80000000 - ffffffff为系统组件:驱动程序、系统表、系统数据结构等等。系统内存的精确格局是不可能得到的,但是通过其功能用途和大致位置可以区分出各个区域。

*80000000-9FFFFFFF

系统代码。在这里驻留的是Hal和ntoskrnl的代码和数据,还有驱动程序(boot驱动和加载ntosldr的驱动)。GDT、IDT和TSS结构体同样驻留在这些区域中。

*C0000000-C0FFFFFF

系统表的区域。这个线性地址空间区域保存着进程的页表,页目录和其它与进程结构体有关的东西。这个区域不是全局性的,不像其它的系统空间区域,而且对于每一个进程来说会

映射到不同的物理空间, 其保存着当前进程的数据结构。

***E1000000-E57FFFFF**

分页池。这个区域可以换出到磁盘上。操作系统中的大多数对象都在这个区域中产生。实际上一些内存池位于这个区域中。

***FB000000-FFDFEFFF**

不可换出页的区域, 即非分页区(Non Paged Pool)。这个区域中的数据永远不能换出到磁盘上。这个区域中的数据总是系统必需的数据。例如, 这里有进程与线程的信息块(Thread environment block, Process Environment block)。

***FFDF000-FFFFFFFF**

PCR - Processor Control Region (进程控制域) 用于每一个进程。这个区域中保存着PCR结构体。在此结构体中保存着系统状态的信息。例如, 关于IRQL、当前线程、IDT等的信息。

低2G线性地址空间(00000000-0FFFFFFF)为进程用户模式的地址空间(每个进程自己的空间)。Win32地址空间看上去一般是下面这个样子:

***00000000-0000FFFF**

保护区域。访问此区域会引发异常。被用于检测NULL指针。

***00xx0000**

通常, 应用程序加载在这样的地址上。

***70000000-78000000**

Win32子系统的库通常映射到这里。

***7FFB0000-7FFD3FFF**

代码页。

***7FFDE000-7FFDEFFF**

用户模式的Thread Environment Block。

***7FFDF000-7FFDFFFF**

用户模式的Process Environment Block。

***7FFE0000-7FFE0FFF**

共享数据区。

***7FFFF000-7FFFFFFF**

保护区域。

02.Windows NT与FLAT模型

=====

自i286开始, 在Intel的处理器里实现了四级保护机制, 相应的就有四个特权级。代码与数据能够拥有某级别的特权。这样, 应用程序、系统程序、内核等等都运行在自己的特权级上, 而且不能随意访问比自己特权级高的代码和数据。实际上没有一个基于Intel处理器的操作系统能用到所有的四个特权级(不为人知的那些不算在内)。Windows NT操作系统也不例外, 只用到了两个特权级(ring)。0级(最高特权级)下运行内核, 3级(最低特权级)为用户级。Intel处理器提供了强大的内存分段机制, 其与特权级一起实现了直到段级的保护(例如, 程序的每一个逻辑段都可以由一个描述符表来描述)。但是Windows NT实现的是FLAT模型

。这就将选择子的使用降到了最低限度。处理器的全局描述符表GDT(Global Descriptor Table), 由Windows NT操作系统管理, 其包含以下描述符(由SoftIce'a得到):

```
Sel. Type    Base    Limit    DPL Attributes
GDTbase=80036000 Limit=03FF
0008 Code32   00000000 FFFFFFFF 0    P RE
0010 Data32   00000000 FFFFFFFF 0    P RW
001B Code32   00000000 FFFFFFFF 3    P RE
0023 Data32   00000000 FFFFFFFF 3    P RW
0028 TSS32    8024D000 000020AB 0    P B
0030 Data32   FFDF0000 00001FFF 0    P RW
003B Data32   7FFD9000 00000FFF 3    P RW
0043 Data16   00000400 0000FFFF 3    P RW
0048 LDT      E1190000 000001FF 0    P
0050 TSS32    80149F60 00000068 0    P
0058 TSS32    80149FC8 00000068 0    P
0060 Data16   00022940 0000FFFF 0    P RW
0068 Data16   000B8000 00003FFF 0    P RW
0070 Data16   FFFF7000 000003FF 0    P RW
0078 Code16   80400000 0000FFFF 0    P RE
0080 Data16   80400000 0000FFFF 0    P RW
0088 Data16   00000000 00000000 0    P RW
0090 Reserved 00000000 00000000 0    NP
...
00E0 Reserved 00008003 00006100 0    NP
00E8 Data16   00000000 0000FFFF 0    P RW
00F0 Code16   80117DB0 0000028D 0    P EO
00F8 Data16   00000000 0000FFFF 0    P RW
0100 Reserved 00008003 00006108 0    NP
...
03F8 Reserved 00000000 00000000 0    NP
```

前四个选择子全都位于线性地址空间。而且前两个选择子的描述符特权级DPL(Descriptor Privilege Level)等于0, 而后面两个的都是3。选择子8和10由用户应用程序使用。在FLAT模型下, 应用程序本身并不关心段寄存器的内容。在ring3工作时, CS、DS、SS寄存器总是分别为值8、10、10。这样, 系统代码就可以监视段寄存器的值。选择子1b和23用于内核(驱动程序、系统代码)工作时的寻址。选择子30和3b分别指向Kernel Process Region和Thread Information Block。当代码运行在ring0时, FS寄存器的值为30, 如过运行在ring3, 则FS的值为3b。选择子30总是指向基址为FFDF000的描述符。选择子3b指示的基址则依赖于用户线程。选择子48定义了局部描述符表LDT(Local Descriptor Table)。LDT只在Virtual DOS machine(VDM)应用程序下使用。当运行该进程时, 在处理器的LDTR寄存器中加载着相应的指针, 否则, LDTR的值为0。LDT主要用在Windows 3.x应用程序下。Windows 3.x应用程序运行在WOW(Windows On Windows)下, 而WOW则实现在VDM进程里。VDM进程的LDT使用上和Win

3.x里的一样。在GDT表里总会有两个TSS类型的选择子。这是因为运行在Intel处理器上的Windows NT操作系统没有使用基于任务门的任务切换机制。IDT包含以下描述符(由SoftIce'a得到):

```
Int Type    Sel:Offset    Attributes Symbol/Owner
IDTbase=F8500FC8 Limit=07FF
0000 IntG32   0008:8013EC54 DPL=0 P _KiTrap00
...
0007 IntG32   0008:8013F968 DPL=0 P _KiTrap07
0008 TaskG    0050:00001338 DPL=0 P
0009 IntG32   0008:8013FCA8 DPL=0 P _KiTrap09
```

```

...
0012 IntG32 0008:80141148 DPL=0 P _KiTrap0F
...
001F IntG32 0008:80141148 DPL=0 P _KiTrap0F
0020 Reserved 0008:00000000 DPL=0 NP
...
0029 Reserved 0008:00000000 DPL=0 NP
002A IntG32 0008:8013E1A6 DPL=3 P _KiGetTickCount
002B IntG32 0008:8013E290 DPL=3 P _KiCallbackReturn
002C IntG32 0008:8013E3A0 DPL=3 P _KiSetLowWaitHighThread
002D IntG32 0008:8013EF5C DPL=3 P _KiDebugService
002E IntG32 0008:8013DD20 DPL=3 P _KiSystemService
002F IntG32 0008:80141148 DPL=0 P _KiTrap0F
0030 IntG32 0008:80014FFC DPL=0 P _HalpClockInterrupt
0031 IntG32 0008:807E4224 DPL=0 P
0032 IntG32 0008:8013D464 DPL=0 P _KiUnexpectedInterrupt2
0033 IntG32 0008:80708864 DPL=0 P
0034 IntG32 0008:807CEDC4 DPL=0 P
0035 IntG32 0008:807E3464 DPL=0 P
0036 IntG32 0008:8013D48C DPL=0 P _KiUnexpectedInterrupt6
0037 IntG32 0008:8013D496 DPL=0 P _KiUnexpectedInterrupt7
0038 IntG32 0008:80010A58 DPL=0 P _HalpProfileInterrupt
0039 IntG32 0008:8013D4AA DPL=0 P _KiUnexpectedInterrupt9
...
00FF IntG32 0008:8013DC66 DPL=0 P _KiUnexpectedInterrupt207

```

表中主要的门类型为中断门。中断任务只是用在关键的时刻，保证特殊情况下能正确完成应急处理，比如说，双重异常（8号中断）。中断2e是系统调用。中断30到3f对应于irq0 - irq15。于是，总是有两个TSS选择子。其中一个（50）用于双重异常。TSS中除了其自己必需部分所占空间外，还在104个字节中保存了一个输入输出位图。在执行Win32应用程序的时候，TSS中的指针保存着不正确的值，这样任何对端口的操作都会引发异常。在VDM工作的时候，位图用来选择是否禁止对端口的访问。我们来看一下在FLAT模型下是如何进行保护的。要研究这个问题需要将注意力转向下面这个条件——保护的原则应该是：保护内核不受用户进程的干扰、保护一个进程不受另一个进程的干扰、保护子系统的代码和数据不受用户进程的干扰。Windows NT的线性地址空间可以分成用户空间（通常为0-7fffffff）和系统与内核空间（通常为80000000-ffffffff）。切换上下文时，内核空间对于所有进程几乎都是一样的。在Windows NT中使用了分页保护的内存。这样，实现内核同用户进程隔离方法就是将核心空间地址页的页表项指针的U/S位置零。这样就不能在ring3下随意访问ring0下的代码和数据了。同样页的寻址也使得进程彼此间的地址空间得到隔离。Windows NT保留了4KB的页作为区域c0300000中的页目录，该页目录映射了所有的4GB物理地址空间。在页目录中有1024个页目录项。每个页目录项都应是4B大小（4KB/1024）。1024个页目录项指向1024个页表，每个页表指向一个4KB的页。页目录的基地址在上下文切换时会相应发生变化并指向将被使用的页目录，当执行新线程时这个页目录被用于线性地址向物理地址的转换（在Intel处理器中这个物理基地址位于CR3寄存器中）。

结果，不同的上下文中里的地址空间（00000000-7fffffff）中一样的线性地址能够映射到不同物理地址上，实现了进程地址空间的彼此隔离。内核空间实际上对所有的上下文都是一样的，其被保护起来不能由用户代码访问，内核代码的页表项的U/S位为零。下面我们将不讨论内核模式而是将注意力转到用户模式上来。的确，ring0和ring3寻址使用的描述符有相同的基址和界限，这可以作为用户在内核或内核驱动中出错的理由。发生错误时可能会破坏当前进程地址空间中的代码和数据。例如，用户进程指针校验错误时调用内核服务可能破坏内核的代码和数据。

操作系统的子系统通过导出调用的DLL模块来提供自己的服务。在用户进程向子系统转换时，

比如调用Win32 API函数, 开始总是运行DLL模块的代码。随后, DLL模块的代码可能通过LPC进行系统调用。所以, DLL模块被映射到用户空间中, 共享进程上下文使用库函数必须保护DLL模块中的代码和数据不会受到可能发生的直接访问。事实上, 这种并不能保护数据。所有的保护都是通过页表项的R/W和U/S位来实现的。DLL模块的代码和数据通常只允许被读取。用户进程能够不经允许而向数据中写入, 当然, 如果告知它们大致的线性地址也可以允许写入。但是, 如果需要, 每一个进程能产生自己的数据副本(copy-on-write机制)。例如, 会有以下的情况出现: 分析PE文件kernel32.dll的文件头可以确定data section的虚拟地址, 随后写程序向这个区域写入错误数据, 之后校对脏数据。在新进程启动时, 数据将是“恢复了的”。因此, 如果有进程和子系统破坏系统的正常运行, 则其只会在自己的上下文中发挥作用。除此之外, 所有关键的数据(直接访问可能会损害子系统的整体性的数据或是重要的数据)都位于单独的进程地址空间中——子系统服务器。使用这些数据的操作都是通过从子系统DLL模块转向服务来进行的(通过LPC)。我们注意到, 这个操作开销较大, 所以在Windows NT 4.0中开始进行一系列的尝试来使之减少。特别重要的是, 现在GDI和USER函数都实现在内核里(更准确的说是在驱动程序Win32k.sys里)。事实上, 产生了一个结论, 在安全性上子系统大量依赖于对其的周密考虑。页保护机制本身提供的保护是不够的。

03.线程信息块(THREAD INFORMATION BLOCK)

=====

在线程执行的时候, 在用户模式下, FS寄存器的值为3b。这个选择子用于寻址一个结构体, 这个结构体位于NTDDK.H文件中, 叫做THREAD ENVIRONMENT BLOCK。每一个线程都有其自己的结构体, 并在上下文切换时, 选择子中的基址会改变, 以指向当前线程的这个结构体。在NTDDK.H中对这个结构体只描述了很少的一部分, 如_NT_TIB(Thread Information Block)。其实_NT_TIB是结构体NT_TEB和NT_TIB的结合。NT_TEB是用户模式线程访问的结构体。NT_TIB通过选择子30从内核模式下访问。TEB结构体的几个域在NTDDK.H(Windows NT 4.0)中没有描述:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
    struct _EXCEPTION_REGISTRATION_RECORD * pNext;
    FARPROC                                pfnHandler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;

typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // 00h Head of exception
    // record list

    PVOID StackBase; // 04h
    PVOID StackLimit; // 08h

    PVOID SubSystemTib; // 0Ch
    union { // 10h
        PVOID FiberData; // for TIB
        ULONG Version; // for TEB
    };
    PVOID ArbitraryUserPointer; // 14h Available
    // for application use
    struct _NT_TIB *Self; // 18h Linear address
    // of TEB structure
} NT_TIB;
typedef NT_TIB *PNT_TIB;
```

```

typedef struct _TEB {           // Size: 0xF88
/*000*/ NT_TIB NtTib;
/*01C*/ VOID *EnvironmentPointer;
/*020*/ CLIENT_ID ClientId;    // PROCESS id, THREAD id
/*028*/ HANDLE ActiveRpcHandle;
/*02C*/ VOID *ThreadLocalStoragePointer;
/*030*/ PEB *ProcessEnvironmentBlock; // PEB
/*034*/ ULONG LastErrorValue;
/*038*/ ULONG CountOfOwnedCriticalSections;
/*03C*/ ULONG CsrClientThread;
/*040*/ ULONG Win32ThreadInfo;
/*044*/ UCHAR Win32ClientInfo[0x7C];
/*0C0*/ ULONG WOW32Reserved;
/*0C4*/ ULONG CurrentLocale;
/*0C8*/ ULONG FpSoftwareStatusRegister;
/*0CC*/ UCHAR SystemReserved1[0xD8]; // ExitStack ???
/*1A4*/ ULONG Spare1;
/*1A8*/ ULONG ExceptionCode;
/*1AC*/ UCHAR SpareBytes1[0x28];
/*1D4*/ UCHAR SystemReserved2[0x28];
/*1FC*/ UCHAR GdiTebBatch[0x4E0];
/*6DC*/ ULONG gdiRgn;
/*6E0*/ ULONG gdiPen;
/*6E4*/ ULONG gdiBrush;
/*6E8*/ CLIENT_ID RealClientId;
/*6F0*/ ULONG GdiCachedProcessHandle;
/*6F4*/ ULONG GdiClientPID;
/*6F8*/ ULONG GdiClientTID;
/*6FC*/ ULONG GdiThreadLocalInfo;
/*700*/ UCHAR UserReserved[0x14];
/*714*/ UCHAR glDispatchTable[0x460];
/*B74*/ UCHAR glReserved1[0x68];
/*BDC*/ ULONG glReserved2;
/*BE0*/ ULONG glSectionInfo;
/*BE4*/ ULONG glSection;
/*BE8*/ ULONG glTable;
/*BEC*/ ULONG glCurrentRC;
/*BF0*/ ULONG glContext;
/*BF4*/ ULONG LastStatusValue;
/*BF8*/ LARGE_INTEGER StaticUnicodeString;
/*C00*/ UCHAR StaticUnicodeBuffer[0x20C];
/*E0C*/ ULONG DeallocationStack;
/*E10*/ UCHAR TlsSlots[0x100];
/*F10*/ LARGE_INTEGER TlsLinks;
/*F18*/ ULONG Vdm;
/*F1C*/ ULONG ReservedForNtRpc;
/*F20*/ LARGE_INTEGER DbgSsReserved;
/*F28*/ ULONG HardErrorsAreDisabled;
/*F2C*/ UCHAR Instrumentation[0x40];
/*F6C*/ ULONG WinSockData;
/*F70*/ ULONG GdiBatchCount;
/*F74*/ ULONG Spare2;
/*F78*/ ULONG Spare3;

```

```

/*F7C*/ ULONG Spare4;
/*F80*/ ULONG ReservedForOle;
/*F84*/ ULONG WaitingOnLoaderLock;
} TEB, *PTEB;

```

在Windows 95下, 位于TIB中的偏移0x30的是指向拥有该线程的进程的基址数据指针。在Windows NT 4.0中, 这个偏移保存的是指向结构体的指针, 该结构体实现于kernel32.dll。遗憾的是, 到现在为止, 除了几个域之外我还不清楚这个结构体的格式。除此之外, 类似的, 在Win 2K中PEB结构体也发生了变化。

```

typedef struct _PROCESS_PARAMETERS {
/*000*/ ULONG AllocationSize;
/*004*/ ULONG ActualSize;
/*008*/ ULONG Flags;//PPFLAG_xxx
/*00c*/ ULONG Unknown1;
/*010*/ ULONG Unknown2;
/*014*/ ULONG Unknown3;
/*018*/ HANDLE InputHandle;
/*01c*/ HANDLE OutputHandle;
/*020*/ HANDLE ErrorHandle;
/*024*/ UNICODE_STRING CurrentDirectory;
/*028*/ HANDLE CurrentDir;
/*02c*/ UNICODE_STRING SearchPaths;
/*030*/ UNICODE_STRING ApplicationName;
/*034*/ UNICODE_STRING CommandLine;
/*038*/ PVOID EnvironmentBlock;
/*03c*/ ULONG Unknown[9];
        UNICODE_STRING Unknown4;
        UNICODE_STRING Unknown5;
        UNICODE_STRING Unknown6;
        UNICODE_STRING Unknown7;
} PROCESS_PARAMETERS, *PPROCESS_PARAMETERS;

```

```

typedef struct _PEB {
// Size: 0x1D8
/*000*/ UCHAR InheritedAddressSpace;
/*001*/ UCHAR ReadImageFileExecOptions;
/*002*/ UCHAR BeingDebugged;
/*003*/ UCHAR SpareBool; // Allocation size
/*004*/ HANDLE Mutant;
/*008*/ HINSTANCE ImageBaseAddress; // Instance
/*00c*/ VOID *DllList;
/*010*/ PPROCESS_PARAMETERS *ProcessParameters;
/*014*/ ULONG SubSystemData;
/*018*/ HANDLE DefaultHeap;
/*01c*/ KSPIN_LOCK FastPebLock;
/*020*/ ULONG FastPebLockRoutine;
/*024*/ ULONG FastPebUnlockRoutine;
/*028*/ ULONG EnvironmentUpdateCount;
/*02c*/ ULONG KernelCallbackTable;
/*030*/ LARGE_INTEGER SystemReserved;
/*038*/ ULONG FreeList;
/*03c*/ ULONG TlsExpansionCounter;
/*040*/ ULONG TlsBitmap;
/*044*/ LARGE_INTEGER TlsBitmapBits;

```

```

/*04C*/ ULONG ReadOnlySharedMemoryBase;
/*050*/ ULONG ReadOnlySharedMemoryHeap;
/*054*/ ULONG ReadOnlyStaticServerData;
/*058*/ ULONG AnsiCodePageData;
/*05C*/ ULONG OemCodePageData;
/*060*/ ULONG UnicodeCaseTableData;
/*064*/ ULONG NumberOfProcessors;
/*068*/ LARGE_INTEGER NtGlobalFlag;          // Address of a local copy
/*070*/ LARGE_INTEGER CriticalSectionTimeout;
/*078*/ ULONG HeapSegmentReserve;
/*07C*/ ULONG HeapSegmentCommit;
/*080*/ ULONG HeapDeCommitTotalFreeThreshold;
/*084*/ ULONG HeapDeCommitFreeBlockThreshold;
/*088*/ ULONG NumberOfHeaps;
/*08C*/ ULONG MaximumNumberOfHeaps;
/*090*/ ULONG ProcessHeaps;
/*094*/ ULONG GdiSharedHandleTable;
/*098*/ ULONG ProcessStarterHelper;
/*09C*/ ULONG GdiDCAttributeList;
/*0A0*/ KSPIN_LOCK LoaderLock;
/*0A4*/ ULONG OSMajorVersion;
/*0A8*/ ULONG OSMinorVersion;
/*0AC*/ USHORT OSBuildNumber;
/*0AE*/ USHORT OSCSDVersion;
/*0B0*/ ULONG OSPlatformId;
/*0B4*/ ULONG ImageSubsystem;
/*0B8*/ ULONG ImageSubsystemMajorVersion;
/*0BC*/ ULONG ImageSubsystemMinorVersion;
/*0C0*/ ULONG ImageProcessAffinityMask;
/*0C4*/ ULONG GdiHandleBuffer[0x22];
/*14C*/ ULONG PostProcessInitRoutine;
/*150*/ ULONG TlsExpansionBitmap;
/*154*/ UCHAR TlsExpansionBitmapBits[0x80];
/*1D4*/ ULONG SessionId;
} PEB, *PPEB;

```

在TEB的开头是NT_TIB结构体(TEB和TIB的结合)。这个结构体中的大部分名字都很易懂, 最有意思的是指向异常处理链表的指针peExcept(Fs:[0])。这个域经常被引用。如果在随便某个Win32应用程序下看一下实际的情况, 可以看到类似下面这样的代码:

```

.01B45480: 64A100000000      mov     eax,fs:[00000000]
.01B45486: 55                push    ebp
.01B45487: 8BEC             mov     ebp,esp
.01B45489: 6AFF             push    0FF
.01B4548B: 68F868B401       push    001B468F8
.01B45490: 687256B401       push    001B45672
.01B45495: 50                push    eax
.01B45496: 64892500000000   mov     fs:[00000000],esp
.01B4549D: 83EC78           sub     esp,078

```

这段有代表性的代码是由编译器生成的, 用于在堆栈中生成_EXCEPTION_REGISTRATION_RECORD。这个堆栈中的结构体用于实现称作“structured exception handling”的机制, 这就是结构化异常处理。接着, 我们来看Windows NT下的结构化异常处理。这个机制可真是十分著名, 而且实现在编译器的细节之中。在MSDN中可以找到Matt Petriek写得非常详细的文章, 题为“A Crash Course on the Depths of Win32 Structured Exception Handling”,

此文介绍的就是这项机制。

FS:[0]中的指针是指向_EXCEPTION_REGISTRATION_RECORD首部的指针。对应地, 每个结构体在pNext域中包含着指向下一个结构体的指针和指向回调函数pfnHandler的指针。不难猜到, 这就是异常处理的处理程序。函数的原型如下:

```
EXCEPTION_DISPOSITION __cdecl _except_handler(  
    struct _EXCEPTION_RECORD *ExceptionRecord,  
    void * EstablisherFrame,  
    struct _CONTEXT *ContextRecord,  
    void * DispatcherContext  
);
```

我们来分析函数的参数。第一个参数是指向下面结构体的指针。

```
typedef struct _EXCEPTION_RECORD {  
    DWORD ExceptionCode;  
    DWORD ExceptionFlags;  
    struct _EXCEPTION_RECORD *ExceptionRecord;  
    PVOID ExceptionAddress;  
    DWORD NumberParameters;  
    DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];  
} EXCEPTION_RECORD;
```

ExceptionCode是Windows NT的异常代号。异常在NTSTATUS.H文件中被描述为STATUS_xxxxxxx: ExceptionAddress - 发生异常的地址。

第三个参数是指向CONTEXT结构体的指针。

```
typedef struct _CONTEXT  
{  
    DWORD ContextFlags;  
    DWORD Dr0;  
    DWORD Dr1;  
    DWORD Dr2;  
    :.  
    DWORD Esp;  
    DWORD SegSs;  
} CONTEXT;
```

这个结构体定义于WINNT.H文件。其意义是不言而喻的, 这里就不全写了。函数返回下面枚举类型值中的一个:

```
typedef enum _EXCEPTION_DISPOSITION {  
    ExceptionContinueExecution,  
    ExceptionContinueSearch,  
    ExceptionNestedException,  
    ExceptionCollidedUnwind  
} EXCEPTION_DISPOSITION;
```

ExceptionFlags定义了下面的位标志:

```
#define EH_NONCONTINUABLE 1  
#define EH_UNWINDING      2
```

```
#define EH_EXIT_UNWIND    4
#define EH_STACK_INVALID  8
#define EH_NESTED_CALL 0x10
```

在发生异常时, 控制传递到`ntoskrnl.exe`中相应的处理程序。例如, 如果试图下面这段代码:

```
mov eax,80100000h
mov dword ptr [eax],0
```

这会引发异常`0e`, 控制传递向处理程序`KiTrap0E`, 堆栈中错误号为`07` (试图在用户模式下向内核写入。该页位于内存中, 因为线性地址`80100000h`是内核加载的起始地址)。之后, 控制传递到`ntdll.dll`, 在这里解析线程的TEB并顺次执行链表中所有的处理程序, 直到某个函数的返回代号不是`ExceptionContinueSearch`。在此之后, 再次调用链表中的所有处理函数, 直到找到某个函数, 但这次用的是另外一个代号`ExceptionCode (STATUS_UNWIND)`并在`ExceptionFlags`设置位`EH_UNWINDINGS`。这个标志用于异常处理, 进行堆栈的清除和其它必要的工作。如果没有一个处理程序能处理, 则就来到链表中最后一个处理程序, 由Win32子系统建立的处理程序。这个处理程序是如何被建立的以及建立在哪里在以后研究`CreateProcessW`函数时会讲到。关于异常处理的完整介绍可以从MSDN获得, 因为在反汇编源代码中所得到的的是扩大化了的异常处理机制(更高层次的机制)。

04. 进程控制域(PROCESSOR CONTROL REGION)

=====

当线程在内核模式下执行时, 在FS寄存器中加载的是选择子`30`, 用于寻址PCR结构体(基址`0xFFDF000`, 界限`0x00001FFF`)。在`NTDDK.H`中远没有描述结构体所有的成员, 只是其中不多的部分。为了说明PCR中的成分信息, 这里列出用于i386的结构体(Windows NT 4.0)

```
typedef struct _KPCR {           // Size: 0xB10
/*000*/ NT_TIB NtTib;
/*01C*/ struct _KPCR* SelfPcr;
/*020*/ struct _KPRCB* Prcb;    // Current PCB
/*024*/ KIRQL Irql;           // Current IRQL
/*028*/ ULONG Irr;
/*02C*/ ULONG IrrActive;
/*030*/ ULONG IDR;
/*034*/ ULONG Reserved2;
/*038*/ struct _KIDTENTRY* ULONG IDT;
/*03C*/ struct _KGDTENTRY* GDT;
/*040*/ struct _KTSS* TSS;
/*044*/ USHORT MajorVersion;    // 1
/*046*/ USHORT MinorVersion;    // 1
/*048*/ KAFFINITY SetMember;
/*04C*/ ULONG StallScaleFactor;
/*050*/ UCHAR DebugActive;
/*051*/ UCHAR Number;
```

```
// End of official portion of KPCR
```

```
/*052*/ BOOLEAN VdmAlert;
/*053*/ UCHAR Reserved;
/*054*/ UCHAR KernelReserved[0x90-0x54];
/*090*/ ULONG SecondLevelCacheSize;
/*094*/ UCHAR HalReserved[0xD4-0x94];
```

```

/*0D4*/ ULONG InterruptMode;
/*0D8*/ ULONG Spare1;
/*0DC*/ UCHAR KernelReserved2[0x120-0xDC];

// Note that current thread is at offset 0x124 (pointer to KTHREAD)
// This essentially means that the 1st PRCB must match the active CPU

/*120*/ UCHAR PrcbData[0xB10-0x120]; // PCBs for all CPUs supported
} KPCR, *PKPCR;

```

PCR包含着指向PCRB(Processor Control Region)的指针,但实际上,PCRB位于PCR的偏移0x120处,并且所有的向PCRB域的转换都是相对于PCR的起点的。在WINNT.H中描述了PCRB结构体。这里给出整个结构体(Windows NT 4.0):

```

// 0x120 from KPCR
typedef struct _KPRCB {
/*000*/ USHORT MinorVersion;
/*002*/ USHORT MajorVersion;
/*004*/ struct _KTHREAD *CurrentThread;
/*008*/ struct _KTHREAD *NextThread;
/*00c*/ struct _KTHREAD *IdleThread;
/*010*/ CCHAR Number;
/*011*/ CCHAR Reserved;
/*012*/ USHORT BuildType;
/*014*/ KAFFINITY SetMember;
/*015*/ struct _RESTART_BLOCK *RestartBlock;
/*018*/ CCHAR CpuType;
/*019*/ CCHAR CpuID;
/*01A*/ CCHAR CpuStep;
/*01C*/ KPROCESSOR_STATE ProcessorState;
/*13C*/ CCHAR KernelReserved[0x40];
/*17C*/ CCHAR HalReserved[0x40];
/*1BC*/ ULONG NpxThread;
/*1C0*/ ULONG InterruptCount;
/*1C4*/ ULONG KernelTime;
/*1C8*/ ULONG UserTime;
/*1CC*/ ULONG DpcTime;
/*1D0*/ ULONG InterruptTime;
/*1D4*/ ULONG ApcBypassCount;
/*1D8*/ ULONG DpcBypassCount;
/*1DC*/ ULONG AdjustDpcThreshold;
/*1E0*/ UCHAR Spare2[0x14];
/*1F4*/ ULONG64 ThreadStartCount;
/*1FC*/ SINGLE_LIST_ENTRY FsRtlFreeSharedLockList;
/*200*/ SINGLE_LIST_ENTRY FsRtlFreeExclusiveLockList;
/*204*/ ULONG CcFastReadNoWait;
/*208*/ ULONG CcFastReadWait;
/*20C*/ ULONG CcFastReadNotPossible;
/*210*/ ULONG CcCopyReadNoWait;
/*214*/ ULONG CcCopyReadWait;
/*218*/ ULONG CcCopyReadNoWaitMiss;
/*21C*/ ULONG KeAlignmentFixupCount;
/*220*/ ULONG KeContextSwitches;
/*224*/ ULONG KeDcacheFlushCount;
/*228*/ ULONG KeExceptionDispatchCount;

```

```

/*22C*/  ULONG KeFirstLevelTbFills;
/*230*/  ULONG KeFloatingEmulationCount;
/*234*/  ULONG KeIcacheFlushCount;
/*238*/  ULONG KeSecondLevelTbFills;
/*23C*/  ULONG KeSystemCalls;
/*240*/  SINGLE_LIST_ENTRY FsRtlFreeWaitingLockList;
/*244*/  SINGLE_LIST_ENTRY FsRtlFreeLockTreeNodeList
/*248*/  CCHAR ReservedCounter[0x18];
/*260*/  PVOID SmallIrpFreeEntry;
/*264*/  PVOID LargeIrpFreeEntry;
/*268*/  PVOID MdlFreeEntry
/*26C*/  PVOID CreateInfoFreeEntry;
/*270*/  PVOID NameBufferFreeEntry
/*274*/  PVOID SharedCacheMapEntry
/*278*/  CCHAR CachePad0[8];
/*280*/  CCHAR ReservedPad[0x200];
/*480*/  CCHAR CurrentPacket[0xc];
/*48C*/  ULONG TargetSet;
/*490*/  PVOID WorkerRoutine;
/*494*/  ULONG IpiFrozen;
/*498*/  CCHAR CachePad1[0x8];
/*4A0*/  ULONG RequestSummary;
/*4A4*/  ULONG SignalDone;
/*4A8*/  ULONG ReverseStall;
/*4AC*/  ULONG IpiFrame;
/*4B0*/  CCHAR CachePad2[0x10];
/*4C0*/  ULONG DpcInterruptRequested;
/*4C4*/  CCHAR CachePad3 [0xc];
/*4D0*/  ULONG MaximumDpcQueueDepth;
/*4D4*/  ULONG MinimumDpcRate;
/*4D8*/  CCHAR CachePad4[0x8];
/*4E0*/  LIST_ENTRY DpcListHead;
/*4E8*/  ULONG DpcQueueDepth;
/*4EC*/  ULONG DpcRoutineActive;
/*4F0*/  ULONG DpcCount;
/*4F4*/  ULONG DpcLastCount;
/*4F8*/  ULONG DpcRequestRate;
/*4FC*/  CCHAR KernelReserved2[0x2c];
/*528*/  ULONG DpcLock;
/*52C*/  CCHAR SkipTick;
/*52D*/  CCHAR VendorString[0xf];
/*53C*/  ULONG MHz;
/*540*/  ULONG64 FeatureBits;
/*548*/  ULONG64 UpdateSignature;
/*550*/  ULONG QuantumEnd;

```

```

} KPRCB, *PKPRCB, *RESTRICTED_POINTER PRKPRCB;

```

PCRB中最有用的就是指向当前线程的指针(KTHREAD结构体)。通常内核代码以以下代码取得此指针：

```

MOV REG, FS:[124h].

```