Gloomy 对 Windows 内核的分析（介绍）

文章整理 By sunsjw QQ:509207758

INTRO (写给 NT 研究者)
==============================

Мы всего лишь момент во времени
Отблеск в глазах,
Мечты о слепоте,
Образы умирающего рассудка ...
(c) by Anathema

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т .д .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 Poll）。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）。

*FFDFF000-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（00000000-0FFFFFFFF）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    FFDFF000  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 FFDFF000 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 Win3.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:[000000000]
.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:[000000000],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 STATUX_xxxxxx: ExceptionAddres - 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 0xFFDFF000，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].

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net

mailto:gl00my@mail.ru

董岩 译

http://greatdong.blog.edu.cn

Gloomy 对 Windows 内核的分析(研究 CreateProcess)

研究 CreateProcess

=========================

Может быть я всегда знал

Мои хрупкие мечты будут ра з б и т ы ради тебя...

(c) by Anathema

我给出一个反汇编 Win32 API 函数 CreateProcess 的例子，来演示研究子系统的技术，同时演示 Win32 是如何与 Windows NT 的执行系统协同工作的。

从 MSDN 中得到函数原型：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,// pointer to name of executable module
    LPTSTR lpCommandLine,    // pointer to command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes,    // process security attributes
    LPSECURITY_ATTRIBUTES lpThreadAttributes,     // thread security attributes
    BOOL bInheritHandles,    // handle inheritance flag
    DWORD dwCreationFlags, // creation flags
    LPVOID lpEnvironment,    // pointer to new environment block
    LPCTSTR lpCurrentDirectory,      // pointer to current directory name
    LPSTARTUPINFO lpStartupInfo,    // pointer to STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation    // pointer to PROCESS_INFORMATION
  );
```

函数中所有的参数都没有详尽的描述。很快，在开始的几行中，建立了异常处理 __except_handler3（堆栈中的结构体对应于 Visual C 的结构体）。然后根据 dwCreationFlags 进行相应有趣的处理。在任何情况下都会在 dwCreationFlags 里去掉标志 CREATE_NO_WINDOW（对于我来说这是个迷）。之后检查不允许的标志组合 DETACH_PROCESS | CREATE_NEW_CONSOLE。如果这些位被同时设置就会输出错误。从新进程优先级中选择一个优先级（除一个之外，清除所有 dwCreationFlags 中的优先级位）。如果要求是 REAL_TIME 优先级，但不能分到处理器，则设置为 HIGH_PRIORITY。接下来是对参数 lpApplicationName、lpCommandLine、lpEnvironment 的繁琐处理。分析结果标明，函数 CreateProcessW 实际上在文档中已经写明。因此，我们考虑到命令行和应用程序名已不相同。DOS 风格的形式为完整的路径。使用未公开的 ntdll.dll 中的函数：

```
NTSYSAPI
BOOLEAN
NTAPI
RtlDosPathNameToNtPathName_U (char* lpPath,
                              RTL_STRING *NtPath,
                              BOOLEAN AllocFlag,
                              RTL_STRING *Reserved);
```

结果会得到\??\:\样的路径。然后，填充公开了的 OBJECT_ATTRIBUTES 结构体，在 ObjectName 域中放入指向获得的路径的指针并调用未公开的函数：

```
NTSYSAPI
  IOSTATUS
  NTAPI
  NtOpenFile (OUT DWORD* Handle, IN ACCESS_MASK DesiredAccess,
          OBJECT_ATTRIBUTES* ObjAttr, PIO_STATUS_BLOCK IoStatusBlock,
          DWORD ShareAccess, DWORD OpenOptions);
```

访问使用的是 SYNCHRONYZE | FILE_EXECUTE。取得打开文件的句柄用于调用另外一个未公开的函数：

```
NTSYSAPI
  NTSTATUS
  NTAPI
  NtCreateSection(
      OUT PHANDLE SectionHandle,
      IN ACCESS_MASK DesiredAccess,
      IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
      IN PLARGE_INTEGER MaximumSize OPTIONAL,
      IN ULONG Protect,
      IN ULONG Attributes,
      IN HANDLE FileHandle OPTIONAL
      );
```

大多数未公开的系统函数都是由相应的公开的 Win32 API 调用的。API 函数 CreateFileMapping 是对 NtCreateSection 的封装。实际上，即使系统直接调用这些函数，也没人会干扰（而且还节省开销）。有趣的是 NtCreateSection 的一个主要的、由 API 函数生成的参数：

DesiredAccess=(flProtectLow==PAGE_READWRITE)?STANDARD_RIGHTS_REQUIRED|7: STANDART_RIGHTS_REQUIRED | 5;DesiredAccess 只可以取两个值。从 CreateProcessW 中调用的形式如下：

```
NtCreateSection ( &SectionHandle, STANDARD_RIGHTS_REQUIRED| 0x1F,
                    NULL, &qwMaximumSize,
                    PAGE_EXECUTEREAD, SEC_IMAGE, NtFileHandle );
```

这样就得到了映象，并将文件——映象源——关闭。这是用公开的 NtClose 函数进行的。来分析一下 NtCreateSection 返回后的代码。对错误处理这里就不进行讨论了，否则会十分繁琐，要讨论大量的次要的函数。我们来研究没有发生错误而且映象是 PE 映象的情况。调用著名的未公开函数：

```
  NTSYSAPI    NTSTATUS    NTAPI
       NtQuerySection(
            IN HANDLE SectionHandle,
            IN SECTIONINFOCLASS SectionInformationClass,
            OUT PVOID SectionInformation,
            IN ULONG SectionInformationLength,
            OUT PULONG ReturnLength OPTIONAL
            );
```

系统中有一些类似于 NtQueryInformationXxxxx 这样的函数（未公开）。要说是未公开的，在 NTDDK.H 中还是描述了一些函数的原型和调用这些函数用到的结构体信息。Matt Pietrek 在其在 Microsoft Systems 期刊（MSDN 中有）的文章中详细描述了 NTDDK.H 中的 NtQueryInformationProcess 的主要功能。遗憾的是，关于 NtQuerySection 函数的信息是不存在的。所有这样的函数都有实际上相同的原型并处理操作系统中的对象。NtQuerySection 返回两类信息（SectionInformationClass 可以为 0 或 1）。对应于取 0 还是取 1，结构体的大小为 16 或是 58 个字节。

CreateProcessW 调用的 SectionInformation 参数的信息类是 1。

```
Struct SECTION_INFO_CLASS1 {
  DWORD EntryPoint;
```

```
DWORD field_4;
DWORD StackReserved;
DWORD StackCommited;
DWORD SubSystem;
DWORD ImageVersionMinor;
DWORD ImageVersionMajor;
DWORD unknown1;
DWORD Characteristics;
DWORD Machine;
DWORD Unknown[4];
};
```

我们看到，这个信息是从 PE 映象的首部中取得的。在主要的域 Characteristics 中输出的是映象的类型（是否是可执行的）。然后检查机器类型，解析 SubSystem 域，检查映象版本。并最终调用未公开的函数：

```
NtCreateProcess(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN HANDLE ParentProcess, //-1
    IN BOOLEAN InheritHandles,
    IN HANDLE SectionHandle,
    IN HANDLE DebugPort OPTIONAL, // NULL
    IN HANDLE ExceptionPort OPTIONAL //NULL
    );
```

由此建立了 Windows NT 的进程对象。关闭映象，因为已经不再需要了。接着设置对象属性，调用未公开函数 NtSetInformationProcess

```
NTSYSAPI
NTSTATUS
NTAPI
NtSetInformationProcess(
   IN HANDLE ProcessHandle,
   PROCESSINFOCLASS ClassInfo,
   IN PVOID Information,
   IN ULONG Length,
);
```

在 NTDDK.H 中有枚举值_PROCESSINFOCLASS，这个值描述了信息类。调整信息类的值：ProcessDefaultHardErrorMode，ProcessBasePriority。对于这些类，信息结构体本身就是一个 32 位的 DWORD。然后调用未公开的函数，Matt Pietrek 在其文章中介绍过该函数：

```
NTSYSAPI
NTSTATUS
NTAPI
NtQueryInformationProcess(
IN HANDLE ProcessHandle,
IN PROCESSINFOCLASS ProcessInformationClass,
```

```
OUT PVOID ProcessInformation,
IN ULONG ProcessInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

我们取得的信息是 ProcessBasicInfo，NTDDK.H 文件中有对其的描述。

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    KAFFINITY AffinityMask;
    KPRIORITY BasePriority;
    ULONG UniqueProcessId;
    ULONG InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

对于 CreateProcessW 来说，必需的信息是 PEB 的地址。因为在获得这项信息之后很快就调用内部函数_BasePushProcessParameters。从参数判断，其用途是调整仅由此进程产生的地址空间。接下来调用两个内部的复杂函数。先调用_BaseCreateStack。_BaseCreateStack 分配并调整进程堆栈。第一，选出用于保留和提交（reservrd 和 commited）堆栈的值。而且，如果 SizeReserved 和 SizeCommited 为 0，则要从发出 CreateProcess 的进程的 PE 文件的首部中获取这些值。接着对这些值进行修整并在进程产生的地址空间中保留内存，对此用到未公开的函数 NtAllocateVirtualMemory（对应于 Win32 API 函数 VirtualAllocEx，VirtualAllocEx 是对其非常简单的封装，而且这两个函数的参数完全相同）。之后，进行两个调用，用下面的伪码能更简洁的说明：

```
FreeReserved=SizeReserv-SizeCommited;
ReservedAddr+=FreeReserved;
if(SizeReserved<=SizeCommited) fl=0;
else {
     ReservedAddr-=Delta;
     SizeCommited+=Delta;
     fl=1;
      }
NtAllocateVirtualMemory(Han,&ReservedAddr,0,SizeCommited,1000,4);

//[skipped]

NtProtectVirtualMemory
     (ProcHan,&ReservedAddr,Delta,PAGE_READWRITE|PAGE_GUARD,&OldProt);
                              /*  对 VirtualProtectEx 的封装  */
```

可见，这里在保留区域中分配内存（在其末尾）。并且分配的内存大于 Delta。这一部分（大小为 Delta）的属性是 PAGE_GUARD 和 PAGE_READWRITE。最后得到以下结构体：

```
***Stack***
---------------?-ReservedAddr
|                         |
|                         |
|    RESERVED          |<- SizeReserved - (SizeCommited+Delta)
```

```
|                        |
|------------------------|-CommitedAddr
|    GUARD PAGE     |<- Delta
|------------------------|
|    READ_WRITE     |<- SizeCommited
|                        |
L----------------------SS:ESP
```

这样，为堆栈分配了 SizeCommited 字节。保留了 SizeReserved。之后在堆栈之下的保留区分配的内存被转换为 GUARD 页（转换成这种页可以引发异常）。从源代码中可以看到，错误的 Delta 的大小可能会产成悲惨的后果。因为这可是个关键的信息——我们来看从哪里找出 Delta 的值：

```
.text:77F04B99                  mov         eax, large fs:18h
.text:77F04B9F                  mov         ecx, [eax+30h]    ; PEB
.text:77F04BA2                  mov         eax, [ecx+54h]    ; READ ONLY DATA
                                ; ReadOnlyStaticServerData
.text:77F04BA5                  mov         edx, [eax+4]
[skipped]
.text:77F04BB1                  mov         esi, [edx+128h] ; Delta
```

在这一部分里，EAX 寄存器指向用于所有进程的全局区域。这个区域只允许被读取。当然，已给出的关于堆栈的更高层次的信息是众所周知的，而这些信息的真实性在源代码中得到了证实。结果，执行 BaseCreateStack 函数填充 StackInformation 结构体。

```
Typedef struct _StackInformation
{
        DWORD Reserved0;
        DWORD Reserved1;
        DWORD AddressOfTop;
        DWORD CommitAddress;
        DWORD ReservedAddress;
} StackInformation;
```

从这个结构体中得到信息本质上是个参数，用来调用下面这个有趣的函数 BaseInitializeContext：

```
BaseInitializeContext(PCONTEXT Context, // 0x200 bytes
PPEB Peb,
PVOID EntryPoint,
DWORD StackTop,
int Type // union (Process, Thread, Fiber)
);
```

这个函数的几个参数：PEB 的地址，堆栈的入口点和参数定义了要创建的上下文（纤程，进程、线程）。函数填充 CONTEXT 结构体（NTDDK.H 中有）的几个域。其中一个域很有意思，在其中放置了起始点（BaseFiberStart、BaseProcessStartThunk、BaseThreadStartThunk 中的一个）。这个点“分娩”出了线程，产生的线程就在新的上下文中执行。实际上，所有三个偏移处的代码都很简短——就是填充相应的堆栈映象并转到两个函数中的某一个。这两个函数分别是_BaseProcessStart 和_BaseThreadStart。这两个函数很是相象，我们只看_BaseProc

essStart 函数。

这个函数在链表中建立了第一个异常处理（见 TEB）。当对内存进行了错误的访问时，正是这个异常处理调用了那个有 OK 和 CANCEL 的对话框。这个处理程序会结束当前进程。但有时如果异常由错误的服务线程产生，则只结束这个线程。

于是，在 BaseInitializeContext 返回后，就填充相应的结构体。并且这个结构体被用作未公开的 NtCreateThread 函数的参数。NtCreateThread 的原型如下：

```
NTSYSAPI
   NTSTATUS
   NTAPI
   NtCreateThread(
         OUT PHANDLE ThreadHandle,
         IN ACCESS_MASK DesiredAccess,
         IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
         IN HANDLE ProcessHandle,
         OUT PCLIENT_ID ClientID,
         IN PCONTEXT Context, /* see _BaseInitializeContext */
         IN StackInformation* StackInfo, /* see _BaseCreateStack */
         IN BOOLEAN CreateSuspended    /* ==1 */
   );
```

终于，在对 PE 映象的 SubSystem 主要域的数据进行处理之后，通过 LPC 转到 Win32 服务。进程应该只在 Win32 子系统下创建。关于此原因的一些高层次信息可以在 Halen Kaster 的书中读到。

对于 CreateProcess 函数来说，必须完成的任务就是启动线程（当然，如果没有在参数 dwCreationFlags 中设置 CREATE_SUSPEND 标志）。线程的启动进行对 NtResumeThread（对 Win32 的 ResumeThread 的封装）的调用。完成了！现在剩下的还有释放内存和正确的退出。

到此对 Win32 子系统的 CreateProcess 函数的主要分析可以得出结论：子系统通常与 Windows NT 的执行体系统协同工作，子系统大多都使用未公开的函数，子系统通过 LPC 与自己的服务器通讯，许多 Win32 API 函数都是对 Nt 函数的封装。所有这些都是我们熟知的，但我们需要用反汇编来证实。

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net

mailto:gl00my@mail.ru

董岩 译

http://greatdong.blog.edu.cn

Gloomy 对 Windows 内核的分析(内核反汇编技术)

内核反汇编技术

==============================

Windows NT 主要是由 C 写成的，所以总的来说进程本身的反汇编不是很复杂。通常

对局部变量和参数的使用是通过地址和用 EBP 形成的 stack frame 来进行的。例如：

```
PAGE:801932D4    mov     eax, large fs:0
PAGE:801932DA    push    ebp
PAGE:801932DB    mov     ebp, esp ; 堆栈中的 frame
PAGE:801932DD    push    0FFFFFFFFh
PAGE:801932DF    push    offset $T13371
PAGE:801932E4    push    offset __except_handler3
PAGE:801932E9    push    eax
PAGE:801932EA    xor     eax, eax
PAGE:801932EC    mov     large fs:0, esp
PAGE:801932F3    sub     esp, 64h   ; 局部变量在堆栈中的位置
PAGE:801932F6    mov     [ebp+Flag], al ; 使用局部变量
```

有时，编译器会生成更为优化的代码，直接使用 ESP 来引用堆栈。

```
.text:80124320    sub     esp, 8
.text:80124323    test    byte ptr ds:_NtGlobalFlag+2, 8
.text:8012432A    push    ebx
.text:8012432B    push    esi
.text:8012432C    push    edi
.text:8012432D    push    ebp
.text:8012432E    jnz     loc_FFFFF000_80124443
.text:80124334    mov     esi, [esp+18h+arg_0] ; 引用第一个参数
```

IDA PRO 这个特别的反汇编器可以跟踪堆栈的创建，所以建立了下面例子中的相应的常量。在调用函数时，参数以相反的顺序放在堆栈中。调用函数自己要负责清理堆栈（编译器就是这样为 C 函数生成代码的）。

```
PAGE:80193288    push    [ebp+ExceptPort]
PAGE:8019328B    push    [ebp+DebugPort]
PAGE:8019328E    push    [ebp+SectionHandle]
PAGE:80193291    push    [ebp+bInheritHandle]
PAGE:80193294    push    edx                  ; ffffffff
PAGE:80193295    push    [ebp+ObjectAttributes]    ;(参数 3)
PAGE:80193298    push    [ebp+Access]     ;            (参数 2)
PAGE:8019329B    push    ecx                  ; Handle     (参数 1)
PAGE:8019329C    call    _PspCreateProcess@32
; PspCreateProcess (Handle,Access,
; ObjectAttributes,-1,bIbjeritHandle.SectionHandle,DebugPort,ExceptPort);
PAGE:801932A1
PAGE:801932A1 NtCreateProcessExit:           ; CODE XREF: _NtCreateProcess@32+71 j
PAGE:801932A1                                ; _NtCreateProcess@32+80 j
PAGE:801932A1    mov     ecx, [ebp+var_10]
PAGE:801932A4    pop     edi
PAGE:801932A5    mov     large fs:0, ecx
PAGE:801932AC    pop     esi
PAGE:801932AD    pop     ebx
PAGE:801932AE    mov     esp, ebp
```

```
PAGE:801932B0       pop       ebp
PAGE:801932B1       retn      20h ; 清理堆栈(返回后 ESP 加上 0x20)
```

为了提高速度，有时在 OS 内核中会使用 fastcall 的函数，参数通过堆栈传递。例如：

```
PAGE:8018CC9D       mov       ecx, [ebp+pObject] ; 第一个参数(下一个参数在 edx 中)
PAGE:8018CCA0       call      @ObfDereferenceObject@4 ; fastcall 函数
```

在这个例子中使用了内核中内部的非导出函数。在下一个例子中，我们使用 HAL.DLL 中的未公开的导出的 fastcall 函数：

```
.text:801335E2      mov       ecx, eax ; OldIrql
.text:801335E4      call      ds:__imp_@KfLowerIrql@4
```

fastcall 函数在其名字中都有字符‘f’。

Microsoft 提供了符号信息，这些符号信息可以用来调试程序。这些信息可以确定内部函数（非导出的）和全局变量的真实名称。这就简化了辨别函数和变量名称的工作，而且除此之外，通过后缀@N 可以确定函数参数的数量。

调试信息是以 NT 4.0 的.DBG 文件和 Windows 2K 的.PDB 文件的形式提供的。SoftICE 和 IDA 都通晓 PDB 和 DBG 文件（IDA 使用插件来加载 ntoskrnl.pdb）。

内核中的函数、变量和结构体的名称本身都能表达一些信息。前缀通常有两种意思，描述函数的特征或是用于子系统的数据。例如：Mm - 子系统内存，Cc - 缓存，Ob - 对象管理器，Ps - 进程管理，Se - 内存管理器，Ke - 内核其它的结构体，Ex - 执行体系统。如果函数是初始化用的（或其可能会转入此类函数），则在前缀的第一个字母后加一个字符‘i’。例如 Ki、Mi。使用 Fastcall 的函数在前缀后加‘f’。系统调用使用前缀 Nt。这些函数不是内核导出的函数，它们的地址记录在 service table 里。调用服务要通过软中断 0x2e。内核导出了 Zw 函数，这些函数是对中断的封装。

```
.text:8011A49C _ZwCreateFile@44 proc near     ; CODE XREF: _FsRtlpOpenDev@8+4D p
.text:8011A49C arg_0                = byte ptr    4
.text:8011A49C      mov       eax, 17h
.text:8011A4A1      lea       edx, [esp+arg_0]
.text:8011A4A5      int       2Eh                 ; 中断处理程序调用 NtCreateFile
.text:8011A4A7      retn      2Ch
.text:8011A4A7 _ZwCreateFile@44 endp
```

我不知道字符 Zw 是什么意思（好像只有内核的设计者知道）。可能是 Zero Wheel（或是零环）的意思，因为 Zw 函数是从内核模式调用的（在 DDK 中描述了一些）。从用户模式下，系统服务是通过 NTDLL（实现于用户模式）调用的，NTDLL 导出了 NtXXX 的封装函数 ZwXXX。

内核中的名字主要要遵循一种描述规则。当然，名字本身承载着意义。名字的内容通常是行为及其对象，即对对象进行某种行为。如：ObCreateObject。

许多的操作系统函数都仅仅是对内核内部函数的封装。例如，NtCreateSection 调用了 MmCreateSection，用的参数也都相同。现在，如果统计一下的话，许多 Nt 函数的原型都是 Windows NT 内核研究者所熟知的，许多内部函数的原型也就可以不用通过逆向工程而获得。有了 C 语言函数的原型再学习其结构和思想就轻松多了。

理论上讲，取得关于内核的信息的更简单的方法不是反汇编内核的映象，而是其它的代码。例如，使用 WinDbg 的 kernel-mode extensions 的代码。WinDbg 的扩展中有额外的命令，扩展的调试命令集。其中有明显使用内核内部结构的命令，或是能减轻分析内核内部结构工作的命令。例如命令!ca、!tokenfields、!processfields 等等。反汇编 kdextx86.dll 和 kdex2x86.dll 的代码可以得到某些结构的信息。

内核调试扩展是个.DLL。导出扩展的命令所用的名字与在 WinDbg 调试器中遇到的名字是相同的。例如，processfields。扩展的 DLL 导出了函数 WinDbgExtensionDllInit，这个函数是在加载扩展后从 WinDbg 调试器中调用的。函数的原型如下：

```
VOID WinDbgExtensionDllInit(PWINDBG_EXTENSION_APIS lpExtensionApis,
                            USHORT MajorVersion,
                            USHORT MinorVersion)
```

第一个参数是指向在.DLL 中使用的 API 的指针。WINDBG_EXTENSION_APIS 结构体包含以下成员，这些成员定义了访问扩展函数集：

lpOutputRoutine - 在控制台输出字符串
lpGetExpressionRoutine - 计算表达式的值
lpGetSymbolRoutine - 取得符号在内存中的地址
lpDisasmRoutine - 反汇编内存
lpCheckControlCRoutine - 检查是否按下 CTRL-C (未实现)
lpReadProcessMemoryRoutine - 读进程内存，带有对 GPF 的保护。
lpWriteProcessMemoryRoutine - 写内存
lpGetThreadContextRoutine - 取得进程寄存器的值
lpSetThreadContextRoutine - 设置寄存器
lpIoctlRoutine - 未实现
lpStackTraceRoutine - 跟踪堆栈

这样.DLL 导出了对应于扩展命令的函数，并可以用于与调试器的有限而熟悉的函数集的协同工作。进一步给出实现扩展命令的函数的原型。

```
#define DECLARE_API32(s)                              \
        CPPMOD VOID                                   \
        s(                                            \
            HANDLE          hCurrentProcess,          \
            HANDLE          hCurrentThread,           \
            ULONG           dwCurrentPc,              \
            ULONG           dwProcessor,              \
            PCSTR           args                      \
         )
```

有趣的是参数 args，它指向 WinDbg 中命令的字符串。借助反汇编可以取得足够的信息以研究扩展命令的工作逻辑。在首要的研究中可以选出直接操纵内核结构体和能辨别结构体成员的命令。例如，命令!ca 的代码说明了内核结构体 control area 和 segment。这个命令的逻辑并不复杂：辨别命令行，从内核内存中读取所要的结构体，输出域中的内容。

但是，扩展命令经常并不会列出内核结构体的所有内容。并且，从名字中并不总是能明确的推断出域的含义，但是反汇编这条命令可以简化对内核内部函数的分析工作。在任何情况下都会有机会对信息做对比，从各种各样的线索中取得信息。

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net

mailto:gl00my@mail.ru

董岩 译

http://greatdong.blog.edu.cn

Gloomy 对 Windows 内核的分析(系统调用接口)

系统调用接口

===========================

Я смотрел на снег весь день... Падающий...
Всегда вниз. Падающий весь де н ь .
И тогда я закричал "Это жизнь ?"
(c) by My Dying Bride

系统调用是线程由用户模式转向内核模式的接口。自然，如果讲到系统的安全性和可靠性，研究系统调用的实现机制是非常有益的。系统服务实现上的错误就是系统安全上的漏洞，因为任何用户模式下的线程都能利用这个错误来访问内核模式。

因此，用户模式下的线程需要调用系统服务并转入内核模式。系统服务的调用是通过中断 2Eh 进行的。用户模式模块 NTDLL.DLL 将调用转向内核中的许多函数。例如，导出函数 NtQuerySection 的代码形式如下：

```
7F67CDC                     public _NtQuerySection@20
7F67CDC _NtQuerySection@20 proc near
7F67CDC
7F67CDC arg_0               = byte ptr   4
7F67CDC
7F67CDC                     mov      eax, 77h          ; NtQuerySection
7F67CE1                     lea      edx, [esp+arg_0]
7F67CE5                     int      2Eh
7F67CE7                     retn     14h
7F67CE7 _NtQuerySection@20 endp
```

实际上，NTDLL.DLL 中其它所有的对内核服务的调用都是这个样子。从代码中可以看到，调用中断 2Eh 时，EAX 寄存器为服务的功能号，EDX 寄存器为堆栈中参数的地址。现在我们来看 NTOSKRNL.EXE 中_KiSystemService（中断 2Eh 的处理程序）的部分代码。有意思的是下面这一段：

```
[skipped]
8013CB20 _KiEndUnexpectedRange proc near
8013CB20                    cmp      ecx, 10h ; if call to win32k.sys
8013CB23                    jnz      short Kss_LimitError
8013CB25                    push     edx
8013CB26                    push     ebx
8013CB27                    call     _PsConvertToGuiThread@0
8013CB2C                    or       eax, eax
8013CB2E                    pop      eax
8013CB2F                    pop      edx

[skipped]
```

```
8013CBD0 ;                    S u b r o u t i n e
8013CBD0
8013CBD0               public     _KiSystemService
8013CBD0 _KiSystemService proc near                    ; DATA      XREF: INIT:801C7A50 o

[skipped]

8013CBD8               mov    ebx, 30h
8013CBDD               db     66h
8013CBDD               mov    fs, bx                     ; set fs to 30 (processor contol region)
8013CBE0               push          dword ptr ds:0FFDFF000h
8013CBE6               mov    dword ptr ds:0FFDFF000h, 0FFFFFFFFh
8013CBF0               mov    esi, ds:0FFDFF124h      ; Current Kernel Thread Pointer
8013CBF6               push          dword ptr [esi+137h] ; previous mode: Kernel/user

[skipped]

8013CC29 _KiSystemServiceRepeat:
8013CC29               mov    edi, eax    ; function number
8013CC2B               shr    edi, 8
8013CC2E               and    edi, 30h
8013CC31               mov    ecx, edi
8013CC33               add    edi, [esi+0DCh] ; got service tables address
8013CC39               mov    ebx, eax
8013CC3B               and    eax, 0FFFh
8013CC40               cmp    eax, [edi+8]        ; num of services
8013CC43               jnb    _KiEndYnexpectedRange

[skipped]

8013CC6E               mov    esi, edx        ; parameters addres
8013CC70               mov    ebx, [edi+0Ch]      ; table with sizes
8013CC73               xor    ecx, ecx
8013CC75               mov    cl, [eax+ebx]        ; size of parameters
8013CC78               mov    edi, [edi]             ; handler's table
8013CC7A               mov    ebx, [edi+eax*4] ; got function address
8013CC7D               sub    esp, ecx        ; clear stack
8013CC7F               shr    ecx, 2
8013CC82               mov    edi, esp
8013CC84               cmp    esi, ds:_MmUserProbeAddress ; 7fff0000
8013CC8A               jnb    kss80
8013CC90 KiSystemServiceCopyArguments:
8013CC90               repe movsd                  ; copy      to ring0 stack
8013CC92 kssdoit:
```

```
8013CC92            call          ebx
8013CC94 kss60:
8013CC94            mov    esp, ebp
8013CC96 kss70:
8013CC96            mov    ecx, ds:0FFDFF124h
8013CC9C            mov    edx, [ebp+3Ch]
8013CC9F            mov    [ecx+128h], edx
8013CC9F _KiSystemService endp
8013CC9F
8013CCA5 _KiServiceExit     proc near

[skipped]

8013CE34 kss80:
8013CE34            test          byte ptr [ebp+6Ch], 1 ; kernel/user
8013CE38            jz     KiSystemServiceCopyArguments
8013CE3E            mov    eax, 0C0000005h ; STATUS_ACCESS_VIOLATION
8013CE43            jmp    kss60
```

这样，如果调用产生于 ring0，则处理程序检查参数是否位于用户地址区域中（见 8013CC84）。之后，处理程序检查传递给它的参数（向 ring0 堆栈中拷贝参数起始于标号 KiSystemServiceCopyArguments）。如果没有错误，则按照预先从服务地址表中选出的地址进行 CALL EBX。接着，注意到两个有趣的地方。第一个是，所有的核心线程都能够取得服务地址表的地址（参照 8013CBF0 和 8013CC33 处的代码）。第二个有趣的地方是，服务表可以有四个（对于每个线程来说）。标号_KiSystemServiceRepeat 处代码的调用依赖于位 0x3000 的值，从四个描述服务表地址的描述符中选择一个。描述符占据 16 字节并连续排列。这四个描述符总称服务描述符表。对于每一个线程，在内核线程结构体中都有其自己的指向服务描述符表的指针。这个指针可以从线程结构体的 0DCh 偏移处取得（Windows NT 4.0 下）。线程结构体的地址可以在内核模式下从 PCRB 的偏移 124h 处取得。（MOV EAX, FS:[124h]）。每个线程都有自己指向服务描述符表的指针，实际上，所有线程中的指针都指向两个描述符表中的一个。这两个表位于 NTOSKERNEL.EXE，分别叫做 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow。表中的描述符的格式如下：

```
typedef struct _ServiceDescriptor{
      DWORD* ServiceTable; /*  指向服务地址表的指针  */
      DWORD Reserved;          /*  在 checked build 下使用  */
      DWORD ServiceLimit;    /*  表中服务的数目  */
      BYTE* ArgumentTable; /*  指向服务堆栈中参数表大小的指针  */
                               /*  实际上等于 (4*参数数目) */
      }ServiceDescriptor;
```

在系统初始化（KiInitSystem）时，表 KeServiceDescriptorTable 和 KeServiceDescriptorShadow 的描述符 0 被初始化为以下这个样子（伪代码）：

```
KeServiceDescriptorTable [0].ServiceTable = KiServiceTable;
KeServiceDescriptorTable [0].ServiceLimit = KiServiceLimit;
KeServiceDescriptorTable [0].ArgumentTable = KiArgumentTable;
memcpy (&KeServiceDescriptorTableShadow[0], &KeServiceDescriptorTable[0],0x10);
```

其余的描述符都为 0。KiServiceTable 是 NTOSKRNL.EXE 中函数的偏移表。KiArgumentTable 为服务参数数目乘以 4（参数堆栈 frame 的大小）。KiServiceLimit 为 KiServiceTable 表中服务的数目。KeServiceDescriptorTableShadow 表的描述符 0，为创建的描述符的副本。因此，描述符 0 是在内核初始化时填充的，并描述了内核的基本服务。这个描述符对所有线程都是相同的。那剩下的描述符是做什么的？在驱动 WIN32K.SYS 初始化的时候会调用内核函数 KeAddSystemServiceTable。其伪代码如下：

```
BOOL KeAddSystemServiceTable (
            PVOID* ServiceTable,
            ULONG Reserved,
            ULONG Limit,
            BYTE* Arguments,
            ULONG NumOfDesc)
{
      if (NumOfDesc>3) return 0;
      Descriptor= &KeServiceDescriptorTable [NumOfDesc*16];
      if (Descriptor->ServiceTable)return 0;
      ShadowDescriptor= &KeServiceDescriptorTableShadow[NumOfDesc*16];
      if (ShadowDescriptor->ServiceTable) return 0;

      ShadowDescriptor->ServiceTable=ServiceTable;
      ShadowDescriptor->Reserved=Reserved;
      ShadowDescriptor->ServiceLimit=Limit;
      ShadowDescriptor->ArgumentTable=Arguments;

      if (NumOfDesc!=1){
            Descriptor->ServiceTable=ServiceTable;
            Descriptor->Reserved=Reserved;
            Descriptor->ServiceLimit=Limit;
            Descriptor->ArgumentTable=Arguments;
            }
return 1;
}
```

函数很简单，但可从中获取不少信息。这个函数填充四个描述符中的一个，一般来说，可能填充 shadow table，也有可能填充主表（若描述符为 0，则未使用）。但有一个特殊之处很有意思——如果添加描述符 1，则该描述符只会被添加到 shadow table 中。初始化 WIN32K.SYS 时，恰好添加的就是描述符 1。此时，其余的描述符并未使用。我们知道，为了提高效率，在 Windows NT v4.0 下，Win32 子系统的 USER 和 GDI 函数都是在内核中实现的。Win32k 是内核模式驱动程序，它实现了 Win32 函数，描述符 1 描述了这些服务。现在，我们来看一下，这些表都为线程提供了什么。函数 KeInitializeThread 有两行：

```
80119344      mov       esi, [ebp+lpThread]
[skipped]
80119394      mov       dword ptr [esi+0DCh], offset _KeServiceDescriptorTable
```

下面又有两行，但是在 PsConvertToGuiThread 函数中的：

```
80192919        mov        ecx, [ebp+lpServiceDescriptorTable] ; thread struct + 0dch
[skipped]
80192926        mov        dword ptr [ecx], offset _KeServiceDescriptorTableShadow
```

如果调用的是 WIN32K.SYS 的服务，但用于当前线程的描述符 1 并未初始化，则在 2e 中断处理程序中会调用函数 PsConvertToGuiThread。服务描述符表有两个——主表和 shadow 表。在主表中只有一个偏移为 0 的非零描述符，其描述了基本的系统服务。在 shadow 表中除描述符 0 之外，还有 WIN32K.SYS 初始化的描述符 1，其描述了 GDI 和 USER 服务。对于 GUI 线程，在线程结构体的偏移 0DCh 处是 shadow 表的地址，对于其它线程，该处为主表的地址。如果线程请求 WIN32K.SYS 的服务，则它要成为 GUI 线程。在研究了服务表的结构以及描述符 1 的用途之后，可以看到 Win32 子系统与内核非常紧密的结合在一起。描述符 1 的特殊性在于其嵌入到了内核代码中。KeAddSystemServiceTable 函数是未公开的函数，非常简单并可以在添加新服务的驱动程序中静态的调用。我们注意到，IIS 使用了两个描述符。所以最好在第三个描述符中添加自己的服务。

Windows NT 调用的特殊之处在于用户模式下的大量指针。几乎每一个内核函数都是以检查指针区域参数正确性这一繁琐工作开始的。因此所有的用户地址空间都与内核空间重合，并且，在用户模式下工作的同时，内核由于页保护被隔绝开，而在内核模式下，不正确的用户指针可以寻址到内核区域。如果看一下选择子 10 和 23 的界限，则可以看到它们是一样的（0xFFFFFFFF)。选择子 23（用户模式的选择子）的界限应该等于内核空间起始地址减 1（0x7FFFFFFF)。例如，在 LINUX 下就是这样的。如果试图强行在调试器中修改这个界限值，则 Windows NT 会出 BSOD。为什么会这样？答案令人不可思议：在内核中执行线程时竟使用选择子 23。一方面，这样是很方便——驱动使用用户指针就像使用普通的指针。而另一方面，这又会引发潜在的错误。我已经讲过，在 LINUX 下，用户和内核空间并不重合，在内核中使用用户指针时需调用 copy_from_user()之类的函数（对于 i386，这些函数仅仅是一些从不同段中进行拷贝的常规代码)。这种不方便性迫使内核程序控制并最小化了对用户指针的使用。

Windows NT 的内核与用户重叠的空间导致了系统最初版本代码中的许多错误。这些错误都十分隐蔽——要知道 Win32 经常要用到服务，这需要向内核中传递正确的参数。

Windows 2K 内核中系统调用接口的变化

===

这里的主要内容是我看到的一篇文章里的。:(我并不想剽窃别人的著作，但是我实在是忘了这是谁的文章以及是在哪里看见的了。

Windows 2K 的内核除了通过中断 2Eh 的系统调用接口之外，还可以通过 SYSENTER/SYSEXIT 指令转入内核模式。这些指令是 Pentium II+处理器里才有的。SYSENTER 的处理程序位于内核中的 KiSystemService 里并调用 KiFastCallEntry。KiFastCallEntry 开始部分的样子如下：

```
MOV      ESP, SS:[0xFFDFF040]
MOV      ESP, [ESP+4]                ;set ring-0 stack
PUSH     0x23                        ;模拟 ring3 堆栈
PUSH     EDX                         ;指向 ring3 堆栈参数的指针
SUB      DWORD PTR [ESP], 4          ;在 ring3 的堆栈中
PUSHFD
OR       DWORD PTR [ESP], 0x200      ;模拟 ring3 的标志寄存器
```

```
PUSH     0x1B                    ;ring3 的 CS 选择子
PUSH     ECX                     ;ring3 的 EIP
;..fill in KeTrapFrame
;..后面部分同系统调用的处理相同
```

显然，对于与系统调用相同的部分，处理程序完全透明的实现了系统调用——上述的代码模拟了调用中断时的堆栈。除此之外，现在可以使用 Fast System Call 机制来进行系统调用。

```
MOV      EAX, NtCallCode ;  系统调用号
LEA      EDX, [ESP+4]      ;  堆栈中的参数
LEA      ECX, SYSEXIT_POINT      ;  返回点
SYSENTER
SYSEXIT_POINT:
```

所有这些都好象是从 NTDLL.DLL 通过中断 2E 调用的。其它的接口与此类似。系统调用最后部分形式如下：

```
TEST     KeFeatureBits, 0x1000        ;支持  fast system call
JZ       ReturnFromInterrupt          ;非  - iret
TEST     DWORD PTR [ESP+4], 1         ;从 ring3 调用?
JZ       ReturnFromInterrupt          ;  非  - iret
TEST     DWORD PTR [ESP+8], 0x20000 ;从 v86 调用?
JNZ      ReturnFromInterrupt          ;  是  - iret
POP      EDX                          ;返回的 eip
ADD      ESP, 8                       ;回收模拟的中断堆栈
POP      ECX                          ;ring3 的 esp 3
STI
SYSEXIT
ReturnFromInterrupt:
IRETD
```

如此——内核支持两种系统调用接口。但是 NTDLL.DLL 与 Windows NT 4.0 下的相同，包含着对系统调用的封装。这样，Windows NT 就不能使用 fast system call 形式的接口。看来，下一版的 NTDLL.DLL 将包含两种接口。或者对 PII 之前和之后的处理器提供两种不同的 NTDLL.DLL。

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net
mailto:gl00my@mail.ru

董岩 译
http://greatdong.blog.edu.cn

Gloomy 对 Windows 内核的分析(内存与进程管理器)

内存与进程管理器
==========================

But I fear tomorrow I'll be crying,
Yes I fear tomorrow I'll be crying.

King Crimson'69 -Epitaph

关于 Windows NT 内存管理器的高层次信息已经够多的了，所以这里不会再讲什么 FLAT 模型、虚拟内存之类的东西。这里我们只讲具体的底层的东西。我假定大家都了解>i386 的体系结构。

目录
==========

00.内核进程线程结构体
01.页表
02.Hyper Space
03.System PTE'S
04.Frame data base (MmPfnDatabase)
05.Working Set
06.向 pagefile 换页
07.page fault 的处理
08.从内存管理器角度看进程的创建
09.上下文切换
0a.某些未公开的内存管理器函数
0b.结语

附录

0c.某些未公开的系统调用
0d.附注及代码分析草稿

00.内核进程线程结构体
====================================

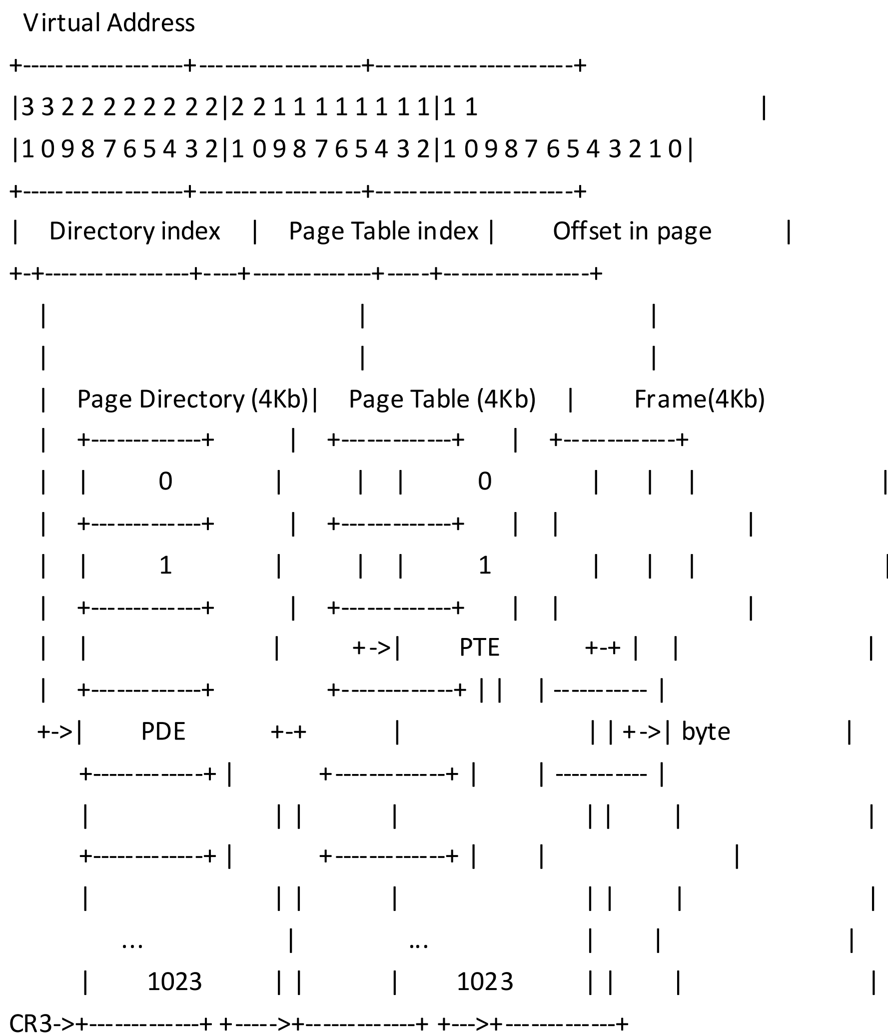
Windows NT 中的每一个进程都是 EPROCESS 结构体。此结构体中除了进程的属性之外还引用了其它一些与实现进程紧密相关的结构体。例如，每个进程都有一个或几个线程，线程在系统中就是 ETHREAD 结构体。我来简要描述一下存在于这个结构体中的主要的信息，这些信息都是由对内核函数的研究而得知的。首先，结构体中有 KPROCESS 结构体，这个结构体中又有指向这些进程的内核线程（KTHREAD）链表的指针（分配地址空间），基优先级，在内核模式或是用户模式执行进程的线程的时间，处理器 affinity（掩码，定义了哪个处理器能执行进程的线程），时间片值。在 ETHREAD 结构体中还存在着这样的信息：进程 ID、父进程 ID、进程映象名、section 指针。quota 定义了所能使用的分页和非分页池的极限值。VAD（virtual address descriptors）树定义了用户地址空间内存区的状况。关于 Working Set 的信息定义了在给定时间内有那些物理页是属于进程的。同时还有 limit 与 statistics。ACCESS TOKEN 描述了当前进程的安全属性。句柄表描述了进程打开的对象的句柄。该表允许不在每一次访问对象时检查访问权限。在 EPROCESS 结构体中还有指向 PEB 的指针。

ETHREAD 结构体还包含有创建时间和退出时间、进程 ID 和指向 EPROCESS 的指针，启动地址，I/O 请求链表和 KTHREAD 结构体。在 KTHREAD 中包含有以下信息：内核模式和用户模式线程的创建时间，指向内核堆栈基址和顶点的指针、指向服务表的指针、基优先级与当前优先级、指向 APC 的指针和指向 TEB 的指针。KTHREAD 中包含有许多其它的数据，通过观察这些数据可以分析出 KTHREAD 的结构。

01.页表

==================

通常操作系统使用页表来进行内存操作。在 Windows NT 中，每一个进程都有自己私有的页表（进程的所有线程共享此页表）。相应的，在进程切换时会发生页表的切换。为了加快对页表的访问，硬件中有一个 translation lookaside buffer（TLB）。在 Windows NT 中实现了两级的转换机制。在 386+处理器上将虚拟地址转换为物理地址过程（不考虑分段）如下：



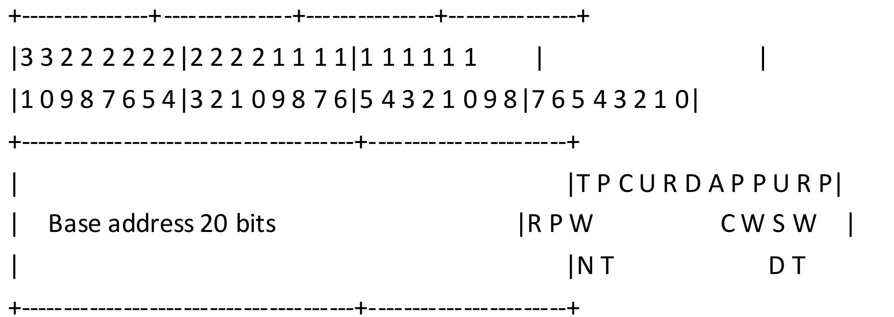
```
 Virtual Address
+-------------------+-------------------+-----------------------+
|3 3 2 2 2 2 2 2 2 2|2 2 1 1 1 1 1 1 1 1|1 1                    |
|1 0 9 8 7 6 5 4 3 2|1 0 9 8 7 6 5 4 3 2|1 0 9 8 7 6 5 4 3 2 1 0|
+-------------------+-------------------+-----------------------+
|   Directory index  |  Page Table index |    Offset in page    |
+-+-----------------+----+--------------+-----+-----------------+
  |                      |                    |
  |                      |                    |
  |  Page Directory (4Kb)|  Page Table (4Kb)  |    Frame(4Kb)
  |  +-------------+     |  +-------------+   |  +-------------+
  |  |      0      |     |  |      0      |   |  |             |
  |  +-------------+     |  +-------------+   |  |             |
  |  |      1      |     |  |      1      |   |  |             |
  |  +-------------+     |  +-------------+   |  |             |
  |  |             |     +->|     PTE     +-+ |  |             |
  |  +-------------+        +-------------+ | |  | ----------- |
  +->|     PDE     +-+      |             | | +->| byte        |
     +-------------+ |      +-------------+ |    | ----------- |
     |             | |      |             | |    |             |
     +-------------+ |      +-------------+ |    |             |
     |             | |      |             | |    |             |
           ...       |          ...         |    |             |
     |    1023     | |      |    1023     | |    |             |
CR3->+-------------+ +----->+-------------+ +--->+-------------+
```

Windows NT 4.0 使用平面寻址。NT 的地址空间为 4G。这 4G 地址空间中，低 2G（地址 0-0x7fffffff）属于当前用户进程，而高 2G（0x80000000-0xffffffff）属于内核。在上下文切换时，要更新 CR3 寄存器的值，结果就更换了用户地址空间，这样就达到了进程间相互隔绝的效果。

注：在 Windows NT 中，从第 4 版起，除 4Kb 的页之外同时还使用了 4Mb 的页（Pentium

及更高）来映射内核代码。但是在 Windows NT 中没有实际对可变长的页提供支持。
PTE 和 PDE 的格式实际上是一样的。

PTE



一些重要的位在 i386+下的定义如下：

P - 存在位。此位如果未设置，则在地址转换时会产生异常。一般说来，在一些情况下 NT 内核会使用未设置此位的 PTE。

例如，如果向 pagefile 换出页，保留这些位可以说明其在页面文件中的位置和 pagefile 号。

U/S - 是否能从 user 模式访问页。正是借助于此位提供了对内核空间的保护（通常为高 2G）。

RW - 是否能写入

NT 使用的为 OS 设计者分配的空闲位

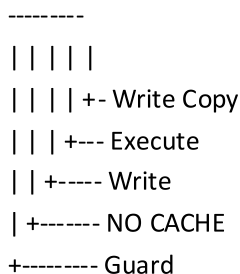
PPT - proto pte

TRN - transition pte

当 P 位未设置时，第 5 到第 9 位即派上用场（用于 page fault 处理）。它们叫做 Protection Mask，样子如下：

----

* MiCreatePagingFileMap



GUARD | NOCACHE 组合就是 NO ACCESS

* MmGetPhysicalAddress

函数很短，但能从中获得很多信息。在虚地址 0xc0000000 - 0xc03fffff 上映射有进程的页表。并且，映射的机制非常精巧。在 Directory Table（以下称 DT）有 1100000000b 个表项（对应于地址 0xc000..-0xc03ff..）指向自己，也就是说对于这些地址 DT 用作了页表（Page Table）！如果我们使用，比如说，地址（为方便起见使用二进制）

```
1100000000.0000000101.0000001001.00b
     ---------- ---------- --------------
0xc0...       页表选择      页表内偏移
页目录
```

通过页表 101b 的 1001b 号，我们得到了 PTE。但这还没完——DT 本身映射在地址 0xc0300000-0xc0300ffc 上。在 MmSystemPteBase 中有值 0xc0300000。为什么这样——看个例子就知道了：

```
1100000000.1100000000.0000001001.00b
     ---------- ---------- --------------
0xc0...       0xc0...       页目录偏移
页目录          页表一
                页目录
                选择
```

最后，在 c0300c00 包含着用于目录本身的 PDE。这个 PDE 的基地址的值保存在 MmSystemPageDirectory 中。同时系统为映射物理页 MmSystemPageDirectory 保留了一个 PTE，这就是 MmSystemPagePtes。

这样做能简化寻址操作。例如，如果有 PTE 的地址，则 PTE 描述的页的地址就等于 PTE<<10。反过来：PTE=(Addr>>10)+0xc0000000。

除此之外，在内核中存在着全局变量 MmKseg2Frame = 0x20000。该变量指示在从 0x80000000 开始的哪个地址区域直接映射到了物理内存，也就是说，此时虚拟地址 0x80000000 - 0x9fffffff 映射到了物理地址 00000000-1f000000。

还有几个有意思的地方。从 c0000000 开始有个 0x1000*0x200=0x200000=2M 的描述地址的表（0-7ffffff）。描述这些页的 PDE 位于地址 c0300000-0xc03007fc。对于 i486，在地址 c0200000-c027fffc 应该是描述 80000000 到 a0000000 的 512MB 的表，但对于 Pentium 在区域 0xc0300800-0xc03009fc 是 4MB 的 PDE，其描述了从 0 到 1fc00000 的步长为 00400000 的 4M 的物理页，也就是说选择了 4M 的页。对应于这些 PDE 的虚地址为 80000000, 9fffffff。

这样我们就得到了页表的分布：

范围 c0000000 - c01ffffc 用于 00000000-7fffffff 的页表
范围 c0200000 - c027ffff "吃掉" 4M 地址页的地址
范围 c0280000 - c02ffffc 包含用于 a0000000 - bfffffff 的页
范围 c0300000 - c0300ffc PD 本身 (描述范围 c0000000 - c03fffff)
范围 c0301000 - c03013fc c0400000-c04fffff HyperSpace (更准确的说，是 1/4 的 hyper space)
范围 c0301400 - c03fffff 包含用于 c050000 - ffffffff 的页

注：在 0xc0301000-0xc0301ffc 包含有描述 hyper space 的页表。这是内核的地址空间，且对于不同的进程映射的内容是不同的（另一方面，内核空间又总是在每个用户进程的上下文中）。这是进程私有的区域。例如，working set 就位于 hyper space 中。页表的前 256 个 PTE（hyper space 的前 1/4）为内核保留，而且在需要快速向 frame 中映射虚拟地址时使用。

我给出一个向区域 0xc0200000-0xc027f000 中一个地址进行映射的例子。
1100000000.1000000000.000000000000 = 0xc0200000

1) 解析出 PDE #1100000000 (4k 页) 并选出 PageDirectory
2) 在 Directory 中选出 PTE #1000000000 (c0300800)
   这是个 4MB 的 PDE - 但这里忽略位长度，
   因为 PDE 用作了 PTE. 结果 c0200000 - c0200fff 被映射为
   80000000-80000fff

   c0201000 映射到下面的 - 80400000- 80400fff.
   等等直到 c027f000 - 9fc00000

PTE, 位于 c0200000 到 c027fffc - 描述了 80000000 - 9ffffc00 (512m)

02.Hyper Space
==============

HyperSpace 是内核空间中的一块区域 (4mb), 不同的进程映射内容不同。对于转换，4MB 足够放下页表完整的一页。这个表位于地址 0xc0301000 - 0xc0301ffc（PDE 的第 0 个表项位于 0xc0300c04）。在内部，为向 HyperSpace 区域中映射物理页（当需要快速为某个 frame 组织虚拟地址时）要使用函数：

DWORD MiMapPageInHyperSpace(DWORD BaseAddr,OUT PDWORD Irql);

它返回 HyperSpace 中的虚拟地址，这个虚拟地址被映射到所要的物理页上。这个函数是如何工作的，工作的时候用到了什么？

在内核中有这样的变量：

MmFirstReservedMappingPte=0xc0301000
MmLastReservedMappingPte=0xc03013fc

这两个变量描述了 255 个 pte，这些 pte 描述了区域：

0xc0400000-0xc04fffff (1/4 HyperSpace)

在 MmFirstReservedMappingPte 处是一个 pte，其中的基址扮演了计数器的角色（从 0 到 255)（当然，pte 是无效的，p 位无效）。为所需地址添加 pte 时要依赖计数器当前的值……并且计数器使用了下开口堆栈的原理，从 ff 开始。一般来说，页表中的 pte 用作信息上的目

的并不是唯一的情况。

03.System PTE'S
===============

在内核中有一块这样的内存——系统pte。什么是系统pte，以及内核如何使用系统pte？

*见函数 MiReserveSystemPtes(...)

系统为空闲 PTE 维护了某些结构体。首先为了快速满足密集请求（当内核需要 pte 映射某些物理页时）系统中有个 Sytem Ptes Pool。而且 pool 中有 pte blocks（blocks 表示请求是以 block 为单位来满足的，一个 block 中有一些 pte，1、2、4、8 和 16 个 pte）。

系统中有以下这些表：

```
BYTE MmSysPteTables[16]={0,0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4};
DWORD MmSysPteIndex[5]={1,2,4,8,16};
DWORD MmFreeSysPteListBySize[5];
PPTE    MmLastSysPteListBySize[5];
DWORD MmSysPteListBySizeCount[5];
DWORD MmSysPteMinimumFree[5]={100,50,30,20,20}
PVOID MmSystemPteBase;// 0xc0200000
```

在 pool 中的空闲 PTE 被组织成了链表（当然，pte 是位于页表中，也就是说链表结构体位于页表中，这是真的）。链表的元素：

```
typedef struct _FREE_SYSTEM_PTES_BLOCK{
/*pte0*/ SYSPTE_REF NextRef;                    // 指向后面的 block
/*pte1*/ DWORD FlushUnkn;                       // 在 Flush 时使用
/*pte2*/ DWORD ArrayOfNulls[ANY_SIZE_ARRAY]; // 空闲 PTE
    }FREE_SYSTEM_PTES_BLOCK PFREE_SYSTEM_PTES_BLOCK;
```

用作指向后面元素指针的 PTE 的地址可如此获得：VA=(NextRef>>10)+MmSystemPteBase（低 10 位永远为 0，相应的 p 位也为 0）。链表最后一个元素 NextRef 域的值为 0xfffff000 (-1) 。相应的，链表有 5 个（block 大小分别为 1,2,4,8 和 16 个 pte）。

*见函数 MiReserveSystemPtes2(...) / MiInitializeSystemPtes

除 pool 外还有一个 undocumented 的空闲系统 pte 链表。

```
PPTE MmSystemPtesStart[2];
PPTE MmSystemPtesEnd[2];
SYSPTE_REF MmFirstFreeSystemPte[2];
DWORD MmTotalFreeSystemPtes[2];
```

在两个链表中有两个引用。链表的元素：

```
typedef struct _FREE_SYSTEM_PTES{
   SYSPTE_REF Next; // #define ONLY_ONE_PTE_FLAG 2, last = 0xfffff000
   DWORD NumOfFreePtes;
}FREE_SYSTEM_PTES PFREE_SYSTEM_PTES;
```

而且，1 号链表原则上没有组织。0 号链表（MiReleaseSystemPtes）用于释放的 pte。pte 有可能进入 System Ptes Pool。若在请求 MiReserveSystemPtes(...)时 pte 的数目大于 16，则同时 pte 从 0 号链表分配。也就是说，0 号链表与 pool 有关联，而 1 号则没有。

为了使工作的结果不与 TLB 相矛盾，系统要么使用重载 cr3，要么使用命令 invlpg。“高级”函数

```
MiFlushPteList(PTE_LIST* PteList, BOOLEAN bFlushCounter, DWORD PteValue);
```

进行以下工作：

初始化 PTE 并调用 invlpg（汇编指令）。

```
typedef struct PTE_LIST{
        DWORD Counter; // max equ 15
        PVOID PtePointersInTable[15];
        PVOID PteMappingAddresses[15];
        };
```

如果 Counter 大于 15，则调用 KeFlushCurrentTb（只是重载 CR3），并且如果设置了 bFlushCounter，则向 MmFlushCounter 加 0x1000。

04.Page Frame Number Data Base (MmPfnDatabase)
======================================

内核将有关物理页的信息保存在 pfn 数据库中（MmPfnDatabase）。本质上讲，这只是个 0x18 字节长的结构体块。每一个结构体对应一个物理页（顺序排列，所以元素常被称为 Pfn - page frame number）。结构体的数量对应于系统中 4KB 页的数量（或者说是内核可见的页的数量，需要的话可以在 boot.ini 中使用相应的选项来为 NT 内核做出这块“坏”页区）。通常，结构体形式如下：

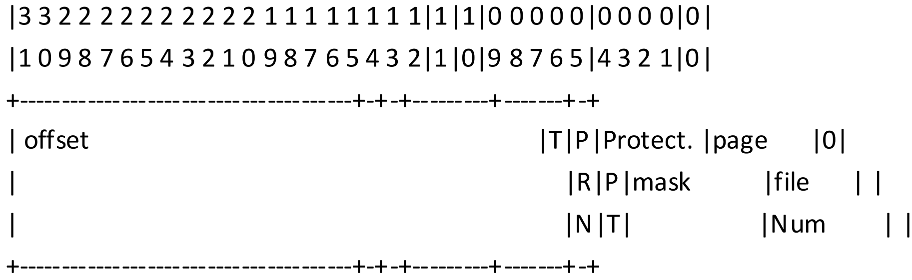
```
typedef struct _PfnDatabaseEntry
      {
      union {
      DWORD NextRef; // 0x0  如果 frame 在链表中，则这个就是 frame 的号
                          //  最后的一个为  -1
```

```
        DWORD Misc;       // 同时另外一项信息，依赖于上下文
                          // 见伪代码 (通常 TmpPfn->0...)
                          // 通常这里有 *KTHREAD, *KPROCESS,
                    // *PAGESUPPORT_BLOCK...
                };
        PPTE PtePpte;   // 0x4 指向 pte 或 ppte
        union {          // 0x8
                DWORD PrevRef;         // 前面的 frame 或 (-1, 第一个)
                DWORD ShareCounter; // Share 计数器
                };
        WORD Flags;          // 0xc 见下面
        WORD RefCounter; // 0xe 引用计数
        DWORD Trans;         // 0x10 ?? 见下面. 用于 pagefile
        DWORD ContFrame;//ContainingFrame; // 14
        }PfnDatabaseEntry;
/*
Flags (名字取自 windbg !pfn 的结果)

掩码    位      名字   值
-----   ----    ---    --------
0001    0       M       Modifyied
0002    1       R       Read In Progress
0004    2       W       WriteInProgress
0008    3       P       Shared
0070    [4:6]   Color Color (In fact Always null for x86)
0080    7       X       Parity Error
0700    [8:10] State 0- Zeroed
                /List 1- Free
                        2- StandBy
                        3- Modified
                        4- ModifiedNoWrite
                        5- BadPage
                        6- Active
                        7- Trans
0800    11      E       InPageError
```

Trans 域的值用在 frame 的内容位于 PageFile 中的时候或是 frame 的内容位于与这个 Page File PTE 对应的其它映象文件中的时候。

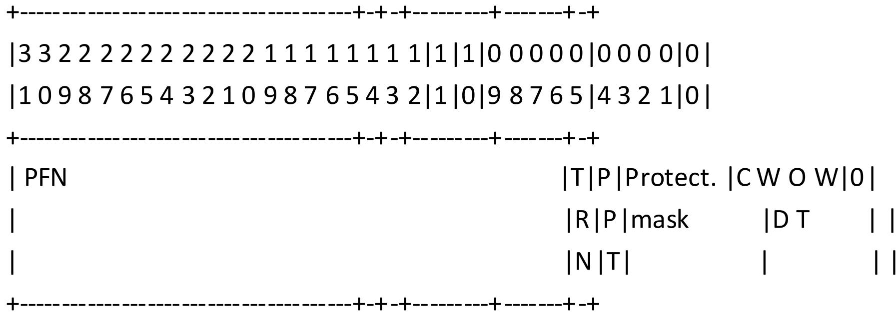
我给出未设置 P 位的 PTE 的例子（这种 PTE 不由平台体系结构确定，而由 OS 确定）。

```
* 取自 @MiReleasePageFileSpace (Trans)

Page File PTE
+--------------------------------------+-+-+---------+-------+-+
```



```
|3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1|1|1|0 0 0 0 0|0 0 0 0|0|
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2|1|0|9 8 7 6 5|4 3 2 1|0|
+--------------------------------------+-+-+---------+-------+-+
| offset                               |T|P|Protect. |page   |0|
|                                      |R|P|mask     |file   | |
|                                      |N|T|         |Num    | |
+--------------------------------------+-+-+---------+-------+-+
```

Transition PTE



```
+--------------------------------------+-+-+---------+-------+-+
|3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1|1|1|0 0 0 0 0|0 0 0 0|0|
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2|1|0|9 8 7 6 5|4 3 2 1|0|
+--------------------------------------+-+-+---------+-------+-+
| PFN                                  |T|P|Protect. |C W O W|0|
|                                      |R|P|mask     |D T    | |
|                                      |N|T|         |       | |
+--------------------------------------+-+-+---------+-------+-+
```

W - write
O - owner
WT - write throuth
CD - cache disable

可能所有这些现在还不很易懂，但是看完下面就能明白了。当然，这个结构体是未公开的。显然，结构体能够组织成链表。frame 由以下结构体支持：

```
struct _MmPageLocationList{

PPfnListHeader ZeroedPageListhead;              //&MmZeroedPageListhead
PPfnListHeader FreePageListHead;                //&MmFreePageListHead
PPfnListHeader StandbyPageListHead;             //&MmStandbyPageListHead
PPfnListHeader ModifiedPageListHead;            //&MmModifiedPageListHead
PPfnListHeader ModifiedNoWritePageListHead;//&MmModifiedNoWritePageListHead
PPfnListHeader BadPageListHead;                 //&MmBadPageListHead
}MmPageLocationList;
```

这其中包含了 6 个链表。各域的名字很好的说明了它们的用处。frame 的状态与这些链表密切关联。下面列举了 frame 的状态：

状态	描述	链表
Zero	清零的可用空闲页	0
Free	可用空闲页	1

```
|Standby        |不可用但可轻易恢复的页                    |  2  |
|Modified       |要换出的 dirty 页                         |  3  |
|ModifiedNoWrite|不换出的 dirty 页                         |  4  |
|Bad            |不可用的页（有错误）                      |  5  |
|Active         |活动页，至少映射一个虚拟地址              |  -  |
+---------------+------------------------------------------------+------+
```

frame 可能处在 6 个链表中的某一个，也可能不在这些链表中（状态为 Active）。如果页属于某个进程，则这个页就被记录在 Working Set 中（见后面）。同时，如果 frame 由内存管理器自己使用，则一般可以不考虑这些 frame 的位置。

每个链表的表头都是下面这个样子：

```
typedef struct _PfnListHeader{
        DWORD Counter; // 链表中 frame 的数目
        DWORD LogNum;    // 链表号.0 - zeroed, 1- Free etc...
        DWORD FirstFn; // MmPfnDatabase 中的第一个 frame 号
        DWORD LastFn;    // --//--- 最后一个.
        }PfnListHeader PPfnListHeader;
```

除此之外，可以用“color”（就是 cache）来寻址空闲 frame（zeroed 或是 free）。如果看一下附录中的伪代码就容易理解了。我给出两个结构体：

```
struct   {
        ColorHashItem* Zeroed; //(-1) нет
        ColorHashItem* Free;
        }MmFreePagesByColor;

typedef struct _ColorHashItem{
                DWORD FrameNum;
                        PfnDatabaseEntry* Pfn;
                        } ColorHashItem;
```

有一套函数使用 color 来处理 frame（处理 cache）。例如，MiRemovePageByColor(FrameNum, Color); 看一下这些函数及其参数返回值的名称和函数的反汇编代码，很容易猜到相应的内容，所以这里就不描述了，在说一句，这些函数都是未导出的。在使用 color 的时候，要考虑 color 掩码，最后选择 color。

Windows NT 符合 C2 安全等级，所以应该在为进程分配页的时候应将页清零。我们来看一下将 frame 清零的系统进程的线程。最后，在 Phase1Initialization()中所作的是调用 MmZeroPageThread。不难猜到——线程将空闲页清零并将其移动到 zeroed 页的链表中。

```
MmZeroPageThread
{
```

```
//
//.... 没意思的东西我们略过 ;)
//
while(1)
 {
 KeWaitForSingleObject(MmZeroingPageEvent,8,0,0,0); // 等待事件
 while(!KeTryToAcquireSpinLock(MmPfnLock,&OldIrql)); // 获取 PfnDatabase
 while(MmFreePageListHead.Count){
            MiRemoveAnyPage(MmFreePageListHead.FirstFn&MmSecondaryColorMask);
               // 从空闲链表中取出页
            Va=MiMapPageToZeroInHyperSpace(MmFreePageListHead.FirstFn);
            KeLowerIrql(OldIrql);

            memset(Va,0,0x1000); // clear page

            while(!KeTryToAcquireSpinLock(MmPfnLock,&OldIrql);
            MiInsertPageInList(&MmZeroedpageListHead,FrameNum);
                        // 将已清零的页插入 Zero 链表
            }
  MmZeroingPageThreadActive=0; // 清标志
  KeLowerIrql(OldIrql);
  }
// 永不退出
}

// 函数只是将 frame 映射到定义的地址上
// 以使其可被清零
DWORD MiMapPageToZeroInHyperSpace(FrameNum)
{
 if(FrameNum<MmKseg2Frame)return ((FrameNum+0x80000)<<12); // 落入直接映射区域

 TmpPte=0xc0301404;
 TmpVa=0xc0501000;
 *TmpPte=0;
 invlpg((void*)TmpVa); // asm instruction in fact
 *TmpPte=FrameNum<<12|ValidPtePte;
 return TmpVa; // always 0xc0501000;
}
```

在何时 MmZeroingPageEvent 被激活？这发生在向空闲页链表中添加 frame 的时候：

```
MiInsertPageInList()
{
.....
```

```
if(MmFreePageListHead.Count>=MmMinimumFreePagesToZero&&
        !MmZeroingPageThreadActive)
    {
     MmZeroingPageThreadActive=1;
     KeSetEvent(&MmZeroingPageEvent,0,0);
    }
….
}
```

注：内核并不总是依赖这个线程，有时会遇到这样的代码，它获取一个空闲页，用过后自己将其清零。

05.Working Set
==============

Working Set——工作集，是属于当前进程的物理页集。内存管理器使用一定的机制跟踪进程的工作集。working set 有两个限额：maximum working set 和 minimum working set。这是工作集的最大值和最小值。内存管理器以这两个值为依据来维护进程的工作集（工作集大小不小于最小值，不大于最大值）。在定义条件的时候，工作集被裁减，这时工作集的 frame 落入空闲链表。内核工作集是结构体的总和。

在进程结构体的偏移 0xc8（NT4.0）有以下结构体。

```
typedef struct _VM{
/* C8*/    LARGE_INTEGER UpdateTime;                //0
/* D0*/    DWORD Pages;                             //8 called so, by S-Ice authors
/* D4*/    DWORD PageFaultCount                     //0c faults;
//                       in fact number of MiLocateAndReserveWsle calls
/* D8*/    DWORD PeakWorkingSetSize;                //10 all
/* DC*/    DWORD WorkingSetSize;                    //14   in
/* E0*/    DWORD   MinimumWorkingSet;               //18   pages, not in
/* E4*/    DWORD   MaximumWorkingSet;               //1c       bytes
/* E8*/    PWS_LIST WorkingSetList;                 //20 data table
/* EC*/    LIST_ENTRY WorkingSetExpansion;          //24 expansion
/* F4*/    BYTE fl0; // Operation???                //2c
     BYTE fl1; // always 2???                   //2d
     BYTE fl2; // reserved??? always 0          //2e
     BYTE fl3; //                               //2f
     }VM *PVM;
```

WinDbg !procfields 的扩展命令用到 VM。这里重要的是，跟踪 page fault 的数量（PageFaultCount），MaximumWorkingSet 和 MinimumWorkingSet，管理器以它们为基础来支持工作集。

注：实际上，PageFaultCount 并非是严格的计数。这个计数在 MiLocateAndReserveWsle

函数中被扩大，因为这个函数不只在 page fault 时被调用，在某些其它情况下也会被调用（真的，很少见）。

下面这个结构体描述了包含工作集页的表。

```
typedef struct _WS_LIST{
DWORD            Quota;                  //0 ??? i'm not shure....
DWORD            FirstFreeWsle;        // 4 start of indexed list of free items
DWORD            FirstDynamic;          // 8 Num of working set wsle entries in the sta
rt
                                         // FirstDynamic
DWORD            LastWsleIndex;         // c above - only empty items
DWORD            NextSlot;               // 10 in fact always == FirstDynamic
                                         // NextSlot
PWSLE          Wsle;                     // 14 pointer to table with Wsle
DWORD            Reserved1               // 18 ???
DWORD            NumOfWsleItems;       // 1c Num of items in Wsle table
                          // (last initialized)
DWORD            NumOfWsleInserted;   // 20 of Wsle items inserted (WsleInsert/
                                         //                                    WsleRemove)
PWSHASH_ITEM HashPtr;                   // 24 pinter to hash, now we can get index of
                          //     Wsle item by address. Present only if
                                         //     NumOfWsleItems>0x180
DWORD            HashSize;               // 28 hash size
DWORD            Reserved2;              // 2c ???
}WS_LIST *PWS_LIST;

typedef struct _WSLE{ // 工作集表的元素
         DWORD PageAddress;
     }WSLE *PWSLE;

// PageAddress 本身是工作集页的虚地址
// 低 12 位用作页属性（虚地址总是 4K 的倍数）

#define WSLE_DONOTPUTINHASH 0x400 // 不放在 cache 中
#define WSLE_PRESENT 0x1 // 非空元素
#define WSLE_INTERNALUSE 0x2 // 被内存管理器使用的 frame

// 未设置 WSLE_PRESENT 的空闲 WSLE 本身是下一个空闲 WSLE 的索引。这样，空闲的 WSLE 就组织
成了链表。最后一个空闲 WSLE 表示为-1。

#define EMPTY_WSLE (next_emty_wsle_index) (next_emty_wsle_index<<4)
#define LAST_EMPTY_WSLE 0xfffffff0
```

```
typedef struct _WSHASH_ITEM{
        DWORD PageAddress; //Value
        DWORD WsleIndex; //index in Wsle table
}WSHASH_ITEM *PWSHASH_ITEM;

//cache 函数很简单。内部函数的伪代码：
//MiLookupWsleHashIndex(Value,WorkingSetList)
//{
//Val=value&0xfffff000;
//TmpPtr=WorkingSetList->HashPtr;
//Mod=(Val>>0xa)%(WorkingSetList->HashSize-1);
//if(*(TmpPtr+Mod*8)==Val)return Mod;
//while(*(TmpPtr+Mod*8)!=Val)){
//      Mod++;
//      if(WorkingSetList->HashSize>Mod)continue;
//      Mod=0;
//      if(fl)KeBugCheckEx(0x1a,0x41884,Val,Value,WorkingSetList);
//      fl=1;
//      }
//return Mod;
//}
```

我们来看一下典型的进程 working set。WorkingSetList 位于地址 MmWorkingSetList (0xc0502000)。这是 hyper space 的区域，所以在进程切换时，要更新这些虚地址，这样，每个进程都有自己的工作集结构体。在地址 MmWsle (0xc0502690)上是 Wsle 动态表的起始地址。表的结尾的地址总是 0x1000 的倍数，也就是说表可以结束在地址 0xc0503000、0xc0504000 等等上（这是为了简化对 Wsle 表大小的操作）。Cache（如果有）位于一个偏移上，Wsle 不会向这个偏移增长。我们来详细看一下这个表：

```
// WsList-0xc0502000---
// ....
// -------0xc0502030----
// pde 00 fault counter
// pde 01 fault counter
// pde 02 fault counter
//
// +-Wsle==0xc0502690---                 +--Pde/pte        +-----Pfn[0]------
// |0 c0300000|403 Page Directory        |c0300c00 pde     |pProcess
// |4 c0301000|403 Hyper Space           |c0300c04 pte     |1
// |8 MmWorkingSetList(c0502000)|403     |c0301408 pte     |2
// |c MmWorkingSetList+0x1000 | 403      |.                |3
// |10 MmWorkingSetList+0x2000 | 403     |.                .
// |              ....
```

```
// |FirstDynamic*4 FrameN
// |....                                        |.                        .
//                                                                             .
// |LastWsleIndex*4 FrameM
// +--------                                   +------             +-------
// | free items
// ....
// | 0xfffffff0
// +------------------


// Cache
// ....
```

这里有个有意思的地方，在表的起始部分有 FirstDynamic 的页，用于建立 Wsle，WorkingSetList 和 cache。同时这里还有页目录 frame，HyperSpace 和某些其它的页，这些页是内存管理器所需要的，不能从工作集中移出（标志 WSLE_INTERNALUSE）。之后，我们还能看到两种对 Pfn frame 域偏移 0 使用的变体。对于页目录 frame，这是指向进程的指针，对于通常的属于工作集的页，这是在表内的索引。

在 WorkingSetList 和 Wsle 表的起始地址之间还有不大的 0x660 字节的空闲空间。关于如何分配这些空间的信息是没有的，但是很快在 WorkingSetList 开始有用于用户空间（通常为低 2GB）的 page fault counter，也就是说如果，譬如说，索引 0x100 的元素有值 3，则表示从 3 开始（如果不考虑可能的溢出）page fault 用于范围[0x40000000-0x403fffff]的页。

工作集的限额在内核模式下可以通过导出的未公开函数来修改：

```
NTOSKRNL MmAdjustWorkingSetSize(
            DWORD MinimumWorkingSet OPTIONAL, // if both == -1
            DWORD MaximumWorkingSet OPTIONAL, // empty working set
            PVM Vm OPTIONAL);
```

为处理 WorkingSet，管理器使用了许多内部函数，了解了这些函数就能明白其工作的原理。

06.向 pagefile 换页
==

frame 可以是空闲的——当 RefCounter 等于 0 且位于一个链表中时。frame 可以属于工作集。在缺少空闲 frame 时或是在达到 treshhold 时，就会发生 frame 的换出。这方面的高层次函数是有的。这里的任务是用伪代码来证实。

在 NT 中有最多 16 个 pagefile。pagefile 的创建发生于模块 SMSS.EXE。这时打开文件及其句柄向 PsInitialSystemProcess 进程的句柄表拷贝。我给出创建 pagefile 的未公开系统函数

的原型（如果不从核心调用的话就必须有创建这种文件的权限）。

```
NTSTATUS NTAPI NtCreatePagingFile(
        PUNICODE_STRING FileName,
        PLARGE_INTEGER MinLen, //  高位双字应为 0
        PLARGE_INTEGER MaxLen, // minlen 应大于 1M
        DWORD Reserved //  忽略
          );
```

每个 pagefile 都有一个 PAGING_FILE 结构体。

```
typedef struct _PAGING_FILE{
            DWORD MinPagesNumber;            //0
            DWORD MaxPagesNumber;            //4
            DWORD MaxPagesForFlushing; //8 (换出页的最大值)
            DWORD FreePages;                   //c(Free pages in PageFile)
            DWORD UsedPages;                   //10  忙着的页
            DWORD MaxUsedPages;                //14
            DWORD CurFlushingPosition; //18 -???
            DWORD Reserved1;                   //1c
            PPAGEFILE_MDL Mdl1; //             20 0x61 - empty ???
            PPAGEFILE_MDL Mdl2; //             24 0x61 - empty ???
            PRTL_BITMAP PagefileMap; //    28 0 -  空闲, 1 -  包含换出页
            PFILE_OBJECT FileObject;      //2c
            DWORD NumberOfPageFile;       //30
            UNICODE_STRING FileName;      //34
            DWORD Lock;                        //3d
        }PAGING_FILE *PPAGING_FILE;

DWORD MmNumberOfActiveMdlEntries;
DWORD MmNumberOfPagingFiles;

#define MAX_NUM_OF_PAGE_FILES 16
PPAGING_FILE MmPagingFile[MAX_NUM_OF_PAGE_FILES];
```

在内存子系统启动时（MmInitSystem(...)）会启动线程 MiModifiedPageWriter，该线程进行以下工作：初始化 MiPaging 和 MiMappedFileHeader，在非换出域中创建并初始化 MmMappedFileMdl，建立优先级 LOW_REALTIME_PRIORITY+1，等待 KEVENT，初始化 MmMappedPageWriterEvent 和 MmMappedPageWriterList 链表，启动 MiMappedPageWriter 线程，启动函数 MiModifiedPageWriterWorker。

在任务 MiModifiedPageWriterWorker 中会等待事件 MmModifiedPageWriterEvent，处理链表 MmModifiedNoWritePageList 和 MmModifiedPageList 并准备实现向映象文件或 pagefile 的页换出（调用 MiGatherMappedPages 或是 MiGatherPagefilePages）。

在 MiGatherPagefilePages 中使用 IoAsynchronousPageWrite( )函数进行 frame 的换出。而且不是一个 frame，而是一簇（页数目总和为 MmModifiedWriteClasterSize）。向 pagefile 换出页是由 PAGING_FILE 结构体中的 PagefileMap 来跟踪的。

研究函数的伪代码在 appendix.txt 中。这里描述伪代码没有什么意义——都很简单。

07.page fault 的处理
==============================

对于转向对 pagefault 的研究，我们现在有了所有必须的信息了。转换线性地址时，当线性地址（分页机制打开）的所用的 PDE/PTE 的 P（present）位无效或是违反了保护规则，在+i386 处理器里会产生异常 14。这时，在堆栈中有错误代号，包含有以下信息：用户/内核错误位（异常发生在 ring3 还是 ring0？），读写错误位（试图读还是写？），页存在位。除此之外，在 CR2 寄存器中存有产生异常的 32 位线性地址。内核中处理 14 号中断的是 _KiTrap0E。当要转换的页没有相应的物理页时，内存管理器执行确定好的工作来“修正”。这些是由异常处理函数调用高层函数 MmAccessFault (Wr,Addr,P);来完成的。在对伪代码的进行分析之前，想一下在什么样的情况下会发生 page fault 是很有用的。

最显然的就是访问错误，这时 ring3 的代码试图写入 PTE/PDE 中未设置 U 位的页或是写入了只读的页（PTE/PDE 中未设置 W 位）。再有，页可以被换出到页面文件中，对应于这些页的 PTE 中未设置 P 位，但有信息指示在哪个页面文件中寻找 frame，以及 frame 的偏移。还有一个类似的情况——frame 属于映象文件。除此之外，所转换的页可能只属于已分配的内存区（使用 NtAllocateMemory），也可能转换的是原先没转换过的页，这中情况下，VMM 分配清零过的 frame（这是 C2 的要求）。最后，异常还可能是由写 copy on write 页和转换共享内存引发。以上只列出了主要的情况。

处理的结果通常是向当前进程的 Working Set 中添加相应的 frame。

异常的每一种情况都相应有一个内部的结构体与之相关联，VMM 就处理这些结构体。这些结构体十分复杂，要对它们进行完整的描述的话，需要反汇编大量的函数。目前还没有大部分结构体的完整信息，但对于理解异常处理程序来说并不要求知道这些。我来大致描述一下 VAD 和 PPTE 的概念，研究异常处理程序的伪代码要用到。

VAD

操作虚拟地址需要用到 VAD (Virtual Address Descriptor)。我们熟知的（有一个几乎与之同名的 Win32 函数调用这个函数）未公开函数 NtAllocateVirtualMemory（ring0 下是 ZwAllocateVirtualMemory）操作这些结构体。

每一个 VAD 都描述了虚地址空间中的区域，实际上，除了区域的起止地址外还有保护信息（见 ZwAllocateViralMemory 函数的参数）。而同时还有其它一些特殊的信息（目前除了首部之外还没有 VAD 的完整信息）。VAD 结构体只对用户地址（低 2GB）有意义，使用这些结构体 VMM 可以捕获到发生异常的区域。VAD 的结构是一个平衡二叉树（有内部函数负

责修整此树)，这是为查找而进行的优化。在 VAD 中有两个指向后面元素——左右子树——的指针。树的根位于 EPROCESS 结构体的 VadRoot 域（NT 4.0 下是偏移 0x170)。当然，每一个进程都有自己的 VAD 树。VAD 的首部形式如下：

```
typedef struct vad_header {
        void *StartingAddress;
        void *EndingAddress;
        struct vad *ParentLink;
        struct vad *LeftLink;
        struct vad *RightLink;
        ULONG Flags;
}VAD_HEADER, *PVAD;
```

PPTE

Prototype Pte 是又一级的线性地址转换并用于共享内存。假设有个文件映射到了几个（3 个）进程的地址空间。PPTE 表包含有 PPTE，这些 PPTE 描述了加载到内存的文件的物理页。某些 PPTE 可以有 P 位（其位置与含义与 PTE/PDE 的相同)，而某些则没有，没有 P 位的有信息用来决定是从页来加载 frame 还是从映象文件来加载文件。所有三个进程的文件都映射在不同的地址上，对应于这些页的 PTE 的 P 位未设置，并且包含有文件页的 PPTE 的引用。这样，在转换映射到文件的线性地址的时候，在一号进程中发生异常 14，VMM 找到 PTE，得到对 PPTE 的引用，现在可以直接“修正”相应的 PTE，以使其指向属于文件的 frame，这时必需从文件中加载 frame。我给出未设置 P 位 PTE 的格式，在页表中其指向原型 PTE。

PTE points to PPTE

```
+---------------------------------------+-+---+------------+-+
|3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1|1|0 0|0 0 0 0 0 0 0|0|
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1|0|9 8|7 6 5 4 3 2 1|0|
+---------------------------------------+-+---+------------+-+
| Address [7:27]                        |1|Un | Address    |0|
|                                       | |use|   [0:6]    | |
|                                       | |d  |            | |
+---------------------------------------+-+---+------------+-+
```

*MmAccessFault

我们开始来研究一下 MmAccessFault 的伪代码。其原型：

NTSTATUS MmAccessFault (BOOL Wr,DWORD Addr, BOOL P)

参数的意义很明显：写入标志，发生异常的地址和页存在位。对于确定异常的原因，这些信息就足够了。根据 Addr 是属于内核地址空间还是用户地址空间，处理程序从两个执行

分支中选择一个。第一种情况下的处理程序较为简单，跟踪 ACCESS VIOLATION 或是收回在 Working Set 中的页（MiDispatchFault）。若是用户空间的地址情况就就更为复杂一些。首先，如果 PDE 不在内存中则执行用于 PDE 的异常处理程序。然后，出现了一个分支。第一个分支——页存在。这表示要么是 ACCESS VIOLATION，要么就是对 copy on write 的处理。第二个分支——处理清零页请求、ACCESS VIOLATION、页边界（GUARD）（堆栈增长）以及必须的对 working set 中页的回收。有趣的是，在大量发生 page fault 的时候，系统会增大 working set 的大小。在零 PTE 的情况下，为确定状况，处理程序不得不使用 VAD 树来确定试图访问区域的属性。这些都是 MiAccessCheck 的工作，这个函数返回访问的状态。

一般情况下，异常处理程序的主要奠基工作是由 MiDispatchFault 函数执行的。它能更精确的确定状况并决定下一步的工作。

轮到 MiDispatchFault 了，它主要是基于一些更低级的函数：MiResolveTransitionFault、MiResolveDemandZeroFault 、 MiResolveDemandZeroFault 、 MiResolveProtoPteFault 和 MiResolvePageFileFault。从这些函数的名字可以明显看出，这个函数用于确定更为具体的情况：状态为'transition'（可能会很快回收入 Working Set）的页应该是空白的 frame，PTE 指向 PPTE 并且 frame 换出到相应的页面文件中。在与页面文件有关的和某些与 PPTE 有关的情况下，接着可能需要从文件中读取 frame，此时函数返回值为 0xc0033333，表示必须从文件中读取页。这在 MiDispatchFault 中是靠 IoPageRead 进行的。我们来更仔细的研究一下所提到的函数。我们从 MiResolveDemandZeroFault 开始。

如果看一下这个函数的伪代码，则可以轻易的明白它的工作逻辑。请求 zero frame 并且进程得到这个 frame。这时执行函数 MiRemoveZeroPage 或是 MiRemoveAnyPage。第一个函数从 zero 页的链表中取一页。如果未能成功，则通过第二个函数选择任何一页。这样的话，该页就由 MiZeroPhysicalPage 来清零。最终，在 MiAddValidPageToWorkingSet 中，该清零的页被添加到工作集中（恰好，这个事实证明在分配内存时进程不能取得对未处理页的访问）。现在我们来研究一下更为复杂的情况——页位于页面文件中。

前面的伪代码需要一个结构体。在准备从文件中读取页的时候，会填充 PAGE_SUPPORT_BLOCK 结构体。之后，对所有即将参与到操作中来的 PFN 进行以下操作：设置 read in progress 标志并在 Misc 域中写入 PAGE_SUPPORT_BLOCK 的地址（函数 MiInitializeReadInProgressPfn）。最后，函数返回 magic number 0xc0033333，表示随后要在 IoPageRead 调用中使用此结构体（恰巧，IoPageRead 被导出了，但是未公开的。从其伪码中可以很容易地得到其原型）。

```
typedef struct _PAGE_SUPPORT_BLOCK{          //    size: 0x98
        DISPATCHER_HEADER DispHeader;    // 0 FastMutex
        IO_STATUS_BLOCK IoStatusBlock; // 0x10
        LARGE_INTEGER AddrInPageFile;    // 0x18 (file offset)
        DWORD RefCounter;                // 0x20 (0|1) ???
        KTHREAD Thread;                  // 0x24
        PFILE_OBJECT FileObject;         // 0x28
        DWORD AddrPte;                   // 0x2c
        PPFN pPfn;                       // 0x30
```

```
        MDL Mdl;                           // 0x34
        DWORD MdlFrameBuffer[0x10];        // 0x50
        LIST_ENTRY PageSupportList;        // 0x90  与 MmInPageSupportList 有关的链表
}PAGE_SUPPORT_BLOCK *PAGE_SUPPORT_BLOCK;

struct _MmInPageSupportList{
     LIST_ENTRY PageSupportList;
     DWORD Count;
     }MmInPageSupportList;
```

函数 MiResolvePageFileFault 本身非常简单，除了填充相应的结构体并返回 0xc0033333 之外什么也不干。剩下的就是执行 MiDispatchFault。这很合乎情理，如果还记得复用代码的原则的话。

还有一个不太复杂的函数 MiResolveTransitionFault。对于状态为 transition 的 frame 还需要再多说几句。从这个状态中 frame 可以很快地返回到进程的 Working Set 中。

于是，剩下了最后一种情况——PROTO PTE。这种情况的处理函数也不太复杂，而且支撑其的基础我们已经讲过了。实际上还有一个函数与这种情况有关，这就是 MiCompleteProtoPteFault，从 MiDispatchFault 中调用。要想理解这些函数的工作就去看一下伪代码。

07. section 对象
================

NT 中的 section 对象就是一块内存，这块内存由一个进程独有或几个进程共享。在 Win32 子系统中 section 就是文件映射（file mapping object）。我们来看一下 section 对象到底是什么。

section 是 NT 下非常常用的对象，执行系统使用 section 来将可执行映象加载到内存中并用其来管理 cache。section 同时也用在向进程地址空间中映射文件。这时访问文件就像访问内存。section 对象，就像其它的对象一样，是由对象管理器创建的。高层次的信息告诉我们，对象的 body 中包含着以下类型的信息：section 的最大值，保护属性，其它属性。什么是 section 的最大可访问值，这不说也知道。保护属性是用于 section 页的属性。其它 section 属性有表示是文件 section 还是为空值（映射入页面文件）的标志，以及 section 是否是 base 的。base 的 section 以相同的虚拟地址映射入所有进程的地址空间。

为了得到此对象结构的真实信息，我反汇编了一些用于 section 的内存管理器函数。下面的信息可是在别的地方见不到的。我们先来看结构体。

系统中的每一个文件都是对象（NTDDK.H 中有描述）FILE_OBJECT。在这个结构体中有 SectionObjectPointer。NTDDK.H 中同样有它的结构。

```
//
```

```
:
PSECTION_OBJECT_POINTERS SectionObjectPointer;
:
//

typedef struct _SECTION_OBJECT_POINTERS {
    PVOID DataSectionObject;
    PVOID SharedCacheMap;
    PVOID ImageSectionObject;
} SECTION_OBJECT_POINTERS;
```

在结构体中有两个指针——DataSectionObject 和 ImageSectionObject。NTDDK.H 把它们写成了 PVOID，因为它们引用的是未公开的结构体。DataSectionObject 用在将文件作为数据打开的时候。ImageSectionObject——此时当作映象。这些指针的类型全都一样，且可以称之为 PCONTROL_AREA。所有下面这些结构体都是 Windows 2K 的，较之 NT 4.0 的有些变化。

```
typedef struct _CONTROL_AREA { // for NT 5.0, size = 0x38
        PSEGMENT    pSegment;           //00
        PCONTROL_AREA    Flink;         //04
        PCONTROL_AREA    Blink;         //08
        DWORD            SectionRef;    //0c
        DWORD            PfnRef;        //10
        DWORD            MappedViews;   //14
        WORD             Subsections;   //18
        WORD             FlushCount;    //1a
        DWORD            UserRef;       //1c
        DWORD            Flags;         //20
        PFILE_OBJECT     FileObject;    //24
        DWORD            Unknown;       //28
        WORD             ModWriteCount; //2c
        WORD             SystemViews;   //2e
        DWORD PagedPoolUsage;           //30
        DWORD NonPagedPoolUsage; //34
        } CONTROL_AREA, *PCONTROL_AREA;
```

我们可以看到，CONTROL_AREA 形成了一个链表，结构体中包含着统计值和标志。为了理解标志所代表的信息，我给出它们的值（用于 NT5.0）

```
/******************* nt5.0 *****************/
#define BeingDeleted            0x1
#define BeingCreated            0x2
#define BeingPurged             0x4
#define NoModifiedWriting       0x8
#define FailAllIo               0x10
```

```
#define Image                    0x20
#define Based                    0x40
#define File                     0x80
#define Networked                0x100
#define NoCache                  0x200
#define PhysicalMemory           0x400
#define CopyOnWrite              0x800
#define Reserve                  0x1000
#define Commit                   0x2000
#define FloppyMedia              0x4000
#define WasPurged                0x8000
#define UserReference            0x10000
#define GlobalMemory             0x20000
#define DeleteOnClose            0x40000
#define FilePointerNull          0x80000
#define DebugSymbolsLoaded       0x100000
#define SetMappedFileIoComplete  0x200000
#define CollidedFlush            0x400000
#define NoChange                 0x800000
#define HadUserReference         0x1000000
#define ImageMappedInSystemSpace 0x2000000
```

紧随 CONTROL_AREA 之后的是 Subsection 的数目 Subsections。每一个 Subsection 都描述了关于具体的文件映射 section 的信息。例如，read-only, read-write, copy-on-write 等等的 section。NT5.0 的 SUBSECTION 结构体：

```
typedef struct _SUBSECTION { // size=0x20 nt5.0
          // +0x10 if GlobalOnlyPerSession
          PCONTROL_AREA   ControlArea; //38, 00
          DWORD           Flags;       //3c, 04
          DWORD           StartingSector;//40, 08
          DWORD           NumberOfSectors; //44, 0c
          PVOID           BasePte;     //48, 10 pointer to start pte
          DWORD           UnusedPtes;  //4c, 14
          DWORD           PtesInSubsect; //50, 18
          PSUBSECTION     pNext;       //54, 1c
          }SUBSECTION, *PSUBSECTION;
```

在 subsection 中有指向 CONTROL_AREA 的指针，标志，指向 base Proto PTE 的指针，Proto PTE 的数目。StartingSector 是 4K block 的编号，文件中的 section 起始于此。在标志中还有额外的信息：

```
#define SS_PROTECTION_MASK        0x1f0
#define SS_SECTOR_OFFSET_MASK     0xfff00000 // (low 12 bits)
```

```
#define SS_STARTING_SECTOR_HIGH_MASK        0x000ffc00    //    (nt5 only) (in pages)

//other 5 bit(s)

#define ReadOnly            1
#define ReadWrite           2
#define CopyOnWrite         4
#define GlobalMemory        8
#define LargePages 0x200
```

我们来看剩下的最后一个结构体 SEGMENT，它描述了所有的映射和用于映射 section 的 Proto PTE。SEGMENT 的内存是从 paged pool 中分配的。我给出 SEGMENT 结构体（NT 5.0）

```
typedef struct    _SEGMENT {
            PCONTROL_AREA ControlArea;        //00
            DWORD                BaseAddr;              //04
            DWORD                TotalPtes;           //08
            DWORD                NonExtendedPtes;//0c
            LARGE_INTEGER SizeOfsegemnt;    //10
            DWORD               ImageCommit;          //18
            DWORD               ImageInfo;              //1c
            DWORD               ImageBase;              //20
            DWORD               Commited;                //24
            PTE                PteTemplate;         //28 or 64 bits if pae enabled
            DWORD BasedAddr;                 //2c
            DWORD BaseAddrPae;              //30 if PAE enabled
            DWORD ProtoPtes;                //34
            DWORD ProtoPtesPae;                    //38 if PAE enabled
            }SEGMENT,*PSEGMENT;
```

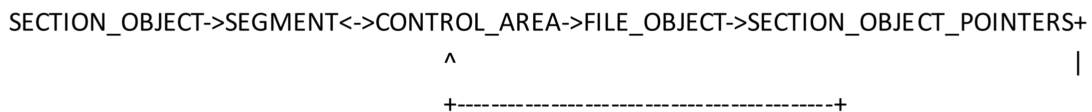
正如我所料，结构体包含着对 CONTROL_AREA 的引用，指向 Proto PTE 的 pool 的指针和所有 section 的信息。有个东西需要解释一下。结构体的样子依赖于是否支持 PAE。PAE 就是 Physical Address Extenion。从第 5 版开始，Windows NT 包含了支持 PAE 的内核 Ntkrnlpa.exe。总的来讲，支持 PAE 就意味着在 NT 里可以使用的虚拟地址不是 4GB 而是 64GB。在使用 PAE 时的地址转换又多了一级——所有的虚地址空间被分为 4 部分。在打开 PAE 时 PTE 和 PDE 的大小不是 4B 而是 8B，这我们可以从 SEGMENT 结构体中看出。现在还不需要进一步详细的讲 PAE，毕竟很少用到，所以我们就此打住。

描述 section 的所有结构体都介绍过了，而 section 对象结构体本身还没有提到。从直观上可以想到，它应该会引用到 SEGMENT 或是 CONTROL_AREA，因为有了这两个结构体后就可以得到保存的所有信息。通过反汇编得到的 section 对象的 body 为以下形式：

```
typedef struct _SECTION_OBJECT { // size 0x28
                VAD_HEADER VadHeader; // 0
```

```
PSEGMENT pSegment;                  //0x14 Segment
LARGE_INTEGER SectionSize; //0x18
DWORD ControlFlags;               //0x20
DWORD PgProtection;               //0x24
} SECTION_OBJECT, *SECTION_OBJECT;

#define PageFile          0x10000
#define MappingFile       0x8000000
#define Based             0x40
#define Unknown           0x800000 // not sure, in fact it's AllocAttrib&0x400000
```

我们看到，所得的结构体完全符合现有的高层信息的描述。唯一可能有疑问的就是 VAD_HEADER。它描述了 base section 在地址空间中的位置。VAD_HEADER 位于顶点为 _MmSectionBasedRoot 的 VAD 树中。我们再次体会到，要理解操作系统的工作原理，就要理解其内部的结构。为了有一个总体上的把握，下面给出了描述 section 的结构体间互相联系的一个图。


```
SECTION_OBJECT->SEGMENT<->CONTROL_AREA->FILE_OBJECT->SECTION_OBJECT_POINTERS+
                                  ^                                         |
                                  +-----------------------------------------+
```

08.从内存管理器角度看进程的创建
==

前面我们从 Win32 角度介绍过进程的创建，也讲过内存管理器和对象管理器的工作原理，以及 section 对象结构体。现在最有意思的当然就是在进程创建中将内存管理器也考虑进来。

进程是用未公开的系统调用 NtCreateProcess()创建的。下面给出其伪代码：

```
/******************************************************************/
/* -- Here it is, just wrapper -- */
NtCreateProcess(
        OUT Handle,
        IN ACCESS_MASK Access,
        IN POBJECT_ATTRIBUTES ObjectAttrib,
        IN HANDLE Parent,
        IN BOOLEAN InheritHandles,
        IN HANDLE SectionHandle,
        IN HANDLE DebugPort,
        IN HANDLE ExceptionPort
        )
{
```

```
if(Parent)
        {
    ret=PspCreateProcess(Handle,
                    Access,
                    ObjectAttrib,
                    Parent,
                    InheritHandles,
                    SectionHandle,
                    DebugPort,
                    ExceptionPort);
        }
else ret=STATUS_INVALID_PARAMETER;
return ret;
}
```

我们看到，NtCreateProcess 是对另一个内部函数 PspCreateProcess 的封装。NtCreateProcess 进行的唯一工作就是检查 Parent（父进程句柄）。但是接下来我们看到，对于 NT 来说这并没有什么意义，因为总的来说，进程的继承性本身没有特别的意义。现在我们来看 PspCreateProcess()。

```
PspCreateProcess(
      OUT PHANDLE Handle,
      IN ACCESS_MASK Access,
      IN POBJECT_ATTRIBUTES ObjectAttrib,
      IN HANDLE Parent,
      IN BOOLEAN InheritHandles,
      IN HANDLE SectionHandle,
      IN HANDLE DebugPort,
      IN HANDLE ExceptionPort
      );
```

我很快注意到，函数中的 Parent 参数可以接受值 0，这就表明在 NtCreateProcess 中检验此参数是为了限制用户模式。函数的参数中有对 section、debug port 和 exception port、父进程的引用。通过调用 ObReferenceObjectByHandle，可以得到指向这些对象的指针。实际上父进程句柄通常传递的是-1，这表示是当前进程。如果 Parent 等于 0，则进程的 affinity 就不从父进程处取得，而是从系统变量中取得。

```
if(Parent)
      { //Get pointer to father's body
        ObReferenceObjectByHandle(Parent,0x80,PsProcessType,PrevMode,&pFather,0)
;
        AffinityMask=pFather->Affinity; // on witch processors will be executed
        Prior=8;
      }
```

```
else {
          pFather=0;
          AffinityMask=KeActiveProcessors;
          Prior=8;
        }
```

优先级总是为 8。随后，创建进程对象。NT4.0 下其大小为 504 字节。

```
// size of process body - 504 bytes
// creating process object... (type object PsProcessType)
ObCreateObject(PrevMode,PsProcessType,ObjectAttrib,PrevMode,0,504,&pProcess);
// clear body
memset(pProcess,0,504);
```

初始化某些域和 Quota Block（见对象管理器的相关介绍）。

```
pProcess->CreateProcessReported=0;
pProcess->DebugPort=pDebugPort;
pProcess->ExceptPort=pExceptPort;

// Inherit Quota Block, if pFather==NULL, PspDefaultQuotaBlock
PspInheritQuota(pProcess,pFather);

if(pFather){
                pProcess->DefaultHardErrorMode=pFather->DefaultHardErrorMode;
                pProcess->InheritedFromUniqueProcessId=pFather->UniqueProcessId;
                }
else {
          pProcess->InheritedFromUniqueProcessId=0;
          pProcess->DefaultHardErrorMode=1;
        }
```

之后，调用 MmCreateProcessAddressSpace，创建地址上下文。参数是函数得到的指向进程的指针、工作集的大小和指向结果结构体的指针。这个结构体形式如下：

```
struct PROCESS_ADDRESS_SPACE_RESULT{
  dword Dt; // dict. table phys. addr.
  dword HypSpace; // hyp space page phys. addr.
  dword WorkingSet; // working set page phys. addr.
}CASResult;

MmCreateProcessAddressSpace(PsMinimumWorkingSet,pProcess,&CASResult);
```

我们看到，函数向我们返回的是页表的物理地址描述符（用于新地址空间的 CR3 的内

容），Hyper Space 的页地址和工作集的页地址。在此之后是初始化进程对象的某些域：

```
  pProcess->MinimumWorkingSet=MinWorkingSet;
  pProcess->MaximumWorkingSet=MaximumWorkingSet;

KeInitializeProcess(pProcess,Prior,AffinityMask,&CASResult,pProcess->
                                                        DefaultHardErrorProcessing&0x4);

pProcess->ForegroundQuantum=PspForegroundQuantum;
```

如果有父进程且设置了标志参数，则会继承父进程的句柄表：

```
  if(pFather) // if there is father and inherithandle, so, inherit handle db
        {
          pFather2=0;
          if(bInheritHandle)pFather2=pFather;
          ObInitProcess(pFather2,pProcess); // see info about ObjectManager
        }
```

下面的东西比较有意思，证明了 NT 执行系统的灵活性，从表面上是看不出来的。如果在参数中有指定的 section，则使用这个 section 来初始化进程的地址空间，否则其工作就会像*UNIX 中的 fork()。

```
  if(pSection)
      {
          MmInitializeProcessAddressSpace(pProcess,0,pSection);
          ObDereferenceObject(pSection);
          res=ObInitProcess2(pProcess); //work with unknown byte +0x22 in process
          if(res>=0)PspMapSystemDll(pProcess,0);
          Flag=1; //Created addr space
      }
else { // if there is futher, but no section, so, do operation like fork()
        if(pFatherProcess){
                        if(PsInitialSystemProcess==pFather){
                                MmRes=MmInitializeProcessAddressSpace(pProcess,0,0)
;
                                                                }
                        else {
                             pProcess->SectionBaseAddress=pFather->SectionBaseAddress;
                             MmRes=MmInitializeProcessAddressSpace(pProcess,pFather,0);
                             Flag=1; //created addr space
                                }
                       }
      }
```

接下来是使用 PsActiveProcessHead 将进程插入 Active Process 链表，创建 Peb 和做其它辅助性的工作。我们不再赘述。最后，当所有的工作都做完后，进行安全子系统方面的工作。我们过去曾研究过安全子系统（见对象管理器部分），所以这里只简单的给出其伪代码。只是我注意到，如果父进程是 system（句柄值等于 PspInitialSystemProcessHandle），则不对其安全性进行检验。

```
// finally, security operations
if(pFather&&PspInitialSystemProcessHandle!=Father)
              {
                ObGetObjectSecurity(pProcess,&SecurityDescriptor,&MemoryAllocated);
                pToken=PsReferencePrimaryToken(pProcess);
                AccessRes=SeAccessCheck(SecurityDescriptor,&SecurityContext,
                                    0,0x2000000,
                                    0,0,&PsProcessToken->GenericMapping,
                                    PrevMode,pProcess->GrantedAccess,
                                    &AccessStatus);
                ObDereferenceObject(pToken);
                ObReleaseObjectSecuryty(SecurityDescriptor,MemoryAllocated);
                if(!AccessRes)pProcess->GrantedAccess=0;
                pProcess->GrantedAccess|=0x6fb;
                }
    else{
          pProcess->GrantedAccess=0x1f0fff;
         }

    if(SeDetailedAuditing)SeAuditProcessCreation(pProcess,pFather);
```

最有意思的是函数 KeInitializeProcess 和 MmCreateProcessAddressSpace。前一个函数除了初始化进程对象的其它成员之外，还要初始化 TSS 中的 IO 位图的偏移。

```
pProcess->IopmOffset=0x20ad; // IOMAP BASE!!!
                                    // You can patch kernel here and
                                    // got i/o port control ;)
```

偏移的选取是这样的，它指向 I/O 位图，这样就能阻止进程直接使用 I/O 端口。

在函数 MmCreateProcessAddressSpace 中进行的是进程地址空间的创建。我就不给出所有的伪代码了，只简要的写写主要的操作。它为 Hyper Space, Working Set 和 Page Directory 选择页。反汇编后的代码证实了，它们是从 zero frame 链表中选出或是由 MiZeroPhysicalPage 函数来清零的。之后初始化新创建的 Page Directory。

```
pProcess->WorkingSetPage=Frame3; // WorkingSetPage
(MmPfnDatabase+0x18*Frame)->Pte=0xc0300000;
```

```
ValidPde_U=ValidPdePde&0xeff^Frame2; // HyperSpace

/**************IMPORTANT!!!!!!!!!!!!!!************************/
/*  重要！ 这里初始化 PD                                       */
/*************************************************************/

Va=MiMapPageInHyperSpace(Frame,&LastIrql);
// no we got Va of our new Page Directory
// Fill some fields
*(Va+0xc04)=ValidPde_U; // HyperSpace
ValidPde_U=ValidPde_U&0xfff^PhysAddr; // DT
*(Va+0xc00)=ValidPde_U; // self-pde

// copy from current process, kernel address mapping
memcpy(
        (MmVirtualBias+0x80000000)>>0x14+Va, // it's like that we found,
                                              // what MmVirtualBias is it ;)
        (MmVirtualBias+0x80000000)>>0x14+0xc0300000,
        0x80 // 32 pdes -> 4Mb*32=128Mb
        );

memcpy( // copy pdes, corresponding to NonPagedArea
        MmNonPagedSystemStart>>0x14+Va,
        MmNonPagedSystemStart>>0x14+0xc0300000,
        (0xc0300ffc-MmNonPagedSystemStart>>0x14+0xc0300000)&0xfffffffc+4);

memcpy(Va+0xc0c, // cache, forgot about it now, it's another story ;)
          0xc0300c0c,
          (MmSystemCacheEnd>>0x14)-0xc0c+4
          );
```

也就是将 PDE 拷贝到内核地址空间中去（其对所有的进程不变，Hyper Space 除外），而且是拷贝到不可换出的区域。同时这个空间是属于系统 cache 的。

09.上下文切换

==========================

知道了 ETHREAD、EPROCESS 结构体和内存管理器的工作原理，就不难猜到上下文切换时会发生什么。Windows NT 的设计者使用线程，不关心共享的是谁的地址空间，也就是说有两种可能：线程属于当前进程——必需要切换到另一个线程（更新堆栈并更换 GDT 描述符），而线程属于另一个进程，必需切换到那个进程（重新加载 CR3）。对此，为了证实我的推测，我反汇编了 KeAttachProcess 函数。这个函数是未公开的，但所有已知的函数都用其来切换到另一进程的地址空间。通过 KeDetachProcess 可以返回到当前进程。KeAttachProcess

使用下述内部函数：

KiAttachProcess - KeAttachProcess 仅仅是对这个函数的封装
KiSwapProcess - 更换地址空间。（本质上就是重新加载 CR3）
SwapContext - 更换上下文。一般不管地址空间的切换，只调整线程上下文。
KiSwapThred - 切换到链表中的下一个线程（SwapContext）调用

下面给出这些内部函数的伪代码。

```
---------------------------------------------------------------------------
/************************ KeAttachProcess **************************/
// just wrapper
//
KeAttachProcess(EPROCESS *Process)
{
KiAttachProcess(Process,KeRaiseIrqlToSynchLevel);
}
/************************ KiAttachProcess **************************/

KiAttachProcess(EPROCESS *Process,Irql){

//CurThread=fs:124h
//CurProcess=CurThread->ApcState.Process;

if(CurProcess!=Process){
        if(CurProcess->ApcStateIndex || KPCR->DpcRoutineActive)KeBugCheckEx...
        }

//if we already in process's context
if(CurProcess==Process){KiUnlockDispatcherDatabase(Irql);return;}

Process->StackCount++;
KiMoveApcState(&CurThread->ApcState,&CurThread->SavedApcState);

// init lists
CurThread->ApcState.ApcListHead[0].Blink=&CurThread->ApcState.ApcListHead[0];
CurThread->ApcState.ApcListHead[0].Flink=&CurThread->ApcState.ApcListHead[0];
CurThread->ApcState.ApcListHead[1].Blink=&CurThread->ApcState.ApcListHead[1];
CurThread->ApcState.ApcListHead[1].Flink=&CurThread->ApcState.ApcListHead[1];;

//fill curtheads's fields
CurThread->ApcState.Process=Process;

CurThread->ApcState.KernelApcInProgress=0;
CurThread->ApcState.KernelApcPending=0;
```

```
CurThread->ApcState.UserApcPending=0;

CurThread->ApcState.ApcStatePointer.SavedApcState=&CurThread->SavedApcState;
CurThread->ApcState.ApcStatePointer.ApcState=&CurThread->ApcState;

CurThread->ApcStateIndex=1;

//if process ready, just swap it...
if(!Process->State)//state==0, ready
	{
	KiSwapProcess(Process,CurThread->SavedApcState.Process);
	KiUnlockDispatcherDatabase(Irql);
	return;
	}

CurThread->State=1; //ready?
CurThread->ProcessReadyQueue=1;

//put Process in Thread's waitlist
CurThread->WaitListEntry.Flink=&Process->ReadyListHead.Flink;
CurThread->WaitListEntry.Blink=Process->ReadyListHead.Blink;

Process->ReadyListHead.Flink->Flink=&CurThread->WaitListEntry.Flink;
Process->ReadyListHead.Blink=&CurThread->WaitListEntry.Flink;

// else, move process to swap list and wait
if(Process->State==1){//idle?
	Process->State=2; //trans
	Process->SwapListEntry.Flink=&KiProcessInSwapListHead.Flink;
	Process->SwapListEntry.Blink=KiProcessInSwapListHead.Blink;
	    KiProcessInSwapListHead.Blink=&Process->SwapListEntry.Flink;
	KiSwapEvent.Header.SignalState=1;
	if(KiSwapEvent.Header.WaitListHead.Flink!=&KiSwapEvent.Header.WaitListHead.
Flink)
				KiWaitTest(&KiSwapEvent,0xa); //fastcall
	}

CurThread->WaitIrql=Irql;
KiSwapThread();
return;
}
```

从这个函数可以得到以下结论。进程可以处于以下状态——0（准备），1（Idle），2（Trans——切换）。这证实了高层次的信息。KiAttachProcess 使用了另外两个函数 KiSwapProce

ss 和 KiSwapThread。

```
/************************* KiSwapProcess ****************************/

KiSwapProcess(EPROCESS* NewProcess, EPROCESS* OldProcess)
{
// just reload cr3 and small work with TSS

        // TSS=KPCR->TSS;
        // xor eax,eax
        // mov gs,ax
TSS->CR3=NewProcess->DirectoryTableBase;//0x1c
        // mov cr3,NewProcess->DirectoryTableBase
TSS->IopmOffset=NewProcess->IopmOffset;//0x66
if(WORD(NewProcess->LdtDescriptor)==0){lldt 0x00; return;//}
        //GDT=KPCR->GDT;
(QWORD)GDT->0x48=(QWORD)NewProcess->LdtDescriptor;
(QWORD)GDT->0x108=(QWORD)NewProcess->Int21Descriptor;
lldt 0x48;
return;
}
```

切换进程上下文。正如我所料，这个函数只是重新加载 CR3 寄存器，再加上一点相关的操作。例如，用 IopmOffset 域的值建立 TSS 中的 I/O 位图的偏移。还必需将选择子的值加载到 ldt（只用于 VDM session）。

```
/************************* SwapContext ******************************/

SwapContext(NextThread,CurThread,WaitIrql)
{

NextThread.State=ThreadStateRunning; //2
KPCR.DebugActive=NextThread.DebugActive;

cli();

//Save Stack
CurThread.KernelStack=esp;

//Set stack
KPCR.StackLimit=NextThread.StackLimit;
KPCR.StackBase=NextThread.InitialStack;

tmp=NextThread.InitialStack-0x70;
```

```
newcr0=cr0&0xfffffff1|NextThread.NpxState|*(tmp+0x6c);
if(newcr0!=cr0)reloadcr0();
if(!*(tmp-0x1c)&0x20000)tmp-=0x10;
TSS=KPCB.TSS;
TSS->ESP0=tmp;

//set pTeb
KPCB.Self=NextThread.pTeb;
esp=NextThread.KernelStack;
sti();

//correct GDT
GDT=KPCB.GDT;
WORD(GDT->0x3a)=NextThread.pTeb;
BYTE(GDT->0x3c)=NextThread.pTeb>>16;
BYTE(GDT->0x3f)=NextThread.pTeb>>24;

//if we must swap processes, do it (like KiSwapProcess)

if(CurThread.ApcState.Process!=NextThread.ApcState.Process)
        {
        //******** like KiSwapProcess
        }

NextThread->ContextSwitches++;

KPCB->KeContextSwitches++;

if(!NextThread->ApcState.KernelApcPending)return 0;

//popf;
// jnz HalRequestSoftwareInterrupt// return 0

return 1;
}
```

切换堆栈，修正 GDT，以使 FS 寄存器指向 TEB。如果线程属于当前进程，则不进行上下文切换。否则，进行的操作和 KiSwapProcess 中的大致差不多。

为了一致，我给出 KeDetachProcess 的原型。

```
KeDetachProcess(EPROCESS *Process,Irql);
```

我们看到——这些函数的伪码实际上完全描述出了操作系统的上下文切换。总的说来，代码分析表明，理解 OS 的主要途径就是要知道它的内部结构。

0a.某些未公开的内存管理器函数
==
SP3 的 ntoskrnl.exe 的内存管理器导出了以下符号：

467 1D0 00051080 MmAdjustWorkingSetSize
468 1D1 0001EDFA+MmAllocateContiguousMemory
469 1D2 00051A14+MmAllocateNonCachedMemory
470 1D3 0001EAE8+MmBuildMdlForNonPagedPool
471 1D4 000206BC MmCanFileBeTruncated
472 1D5 0001EF5A+MmCreateMdl
473 1D6 0002095C MmCreateSection
474 1D7 00021224 MmDbgTranslatePhysicalAddress
475 1D8 000224AC MmDisableModifiedWriteOfSection
476 1D9 000230C8 MmFlushImageSection
477 1DA 0001FA9C MmForceSectionClosed
478 1DB 0001EEA0+MmFreeContiguousMemory
479 1DC 00051AFE+MmFreeNonCachedMemory
480 1DD 0001EEAC+MmGetPhysicalAddress
481 1DE 00024028 MmGrowKernelStack
482 1DF 0004E144 MmHighestUserAddress
483 1E0 0002645A+MmIsAddressValid
484 1E1 00026CD8+MmIsNonPagedSystemAddressValid
485 1E2 0001F5D8 MmIsRecursiveIoFault
486 1E3 00026D56+MmIsThisAnNtAsSystem
487 1E4 000766C8+MmLockPagableDataSection
488 1E5 000766C8 MmLockPagableImageSection
489 1E6 0001F160+MmLockPagableSectionByHandle
490 1E7 0001ED18+MmMapIoSpace
491 1E8 0001EB74+MmMapLockedPages
492 1E9 0001F5F6 MmMapMemoryDumpMdl
493 1EA 00076A14 MmMapVideoDisplay
494 1EB 0005206C MmMapViewInSystemSpace
495 1EC 00079B0E MmMapViewOfSection
496 1ED 0007A7EE+MmPageEntireDriver
497 1EE 0001E758+MmProbeAndLockPages
498 1EF 00026D50+MmQuerySystemSize
499 1F0 00052A8A+MmResetDriverPaging
500 1F1 0004E0A4 MmSectionObjectType
501 1F2 00079D28 MmSecureVirtualMemory
502 1F3 0001EFCE MmSetAddressRangeModified

503 1F4 0007684E MmSetBankedSection
504 1F5 0001EF2C+MmSizeOfMdl
505 1F6 0004E0A0 MmSystemRangeStart
506 1F7 0001F516+MmUnlockPagableImageSection
507 1F8 0001EA16+MmUnlockPages
508 1F9 0007669A+MmUnmapIoSpace
509 1FA 0001ECA8+MmUnmapLockedPages
510 1FB 00076A2E MmUnmapVideoDisplay
511 1FC 00052284 MmUnmapViewInSystemSpace
512 1FD 0007AFE4 MmUnmapViewOfSection
513 1FE 0007A00A MmUnsecureVirtualMemory
514 1FF 0004DDCC MmUserProbeAddress

这里的符号'+'表示函数在 DDK 中有记载。我这里给出某些未公开函数的原型。

```
// 调整 working set 的大小.

NTOSKRNL NTSTATUS MmAdjustWorkingSetSize(
            DWORD MinimumWorkingSet OPTIONAL, // if both == -1
            DWORD MaximumWorkingSet OPTIONAL, // empty working set
            PVM Vm OPTIONAL);

//can file be truncated???
NTOSKRNL BOOLEAN MmCanFileBeTruncated(
            PSECTION_OBJECT_POINTERS SectionPointer, // see FILE_OBJECT
                PLARGE_INTEGER NewFileSize
            );

// create section. NtCreateSection call this function...

NTOSKRNL NTSTATUS MmCreateSection (
    OUT PVOID                   *SectionObject,
    IN ACCESS_MASK              DesiredAccess,
    IN POBJECT_ATTRIBUTES       ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER           MaximumSize,
    IN ULONG                    SectionPageProtection,//PAGE_XXXX
    IN ULONG                    AllocationAttributes,//SEC_XXX
    IN HANDLE                   FileHandle OPTIONAL,
    IN PFILE_OBJECT             File OPTIONAL
);

typedef enum _MMFLUSH_TYPE {
    MmFlushForDelete,
    MmFlushForWrite
```

```
} MMFLUSH_TYPE;

NTOSKRNL BOOLEAN MmFlushImageSection (
	IN PSECTION_OBJECT_POINTERS		SectionObjectPointer,
	IN MMFLUSH_TYPE				FlushType
);

NTOSKRNL DWORD MmHighestUserAddress; // 一般为 0x7ffeffff


NTOSKRNL BOOLEAN MmIsRecursiveIoFault();
//其代码
#define _MmIsRecursiveIoFault() ( \
	(PsGetCurrentThread()->DisablePageFaultClustering) | \
	(PsGetCurrentThread()->ForwardClusterOnly) \
)

NTOSKRNL POBJECT_TYPE MmSectionObjectType; //标准的 Section 对象

NTOSKRNL DWORD MmSystemRangeStart; //一般为 0x80000000
NTOSKRNL DWORD MmUserProbeAddress; //一般为 0x7fff0000

NTOSKRNL PVOID MmMapVideoDisplay( // для i386 враппер в MmMapIoSpace
		IN PHYSICAL_ADDRESS PhysicalAddress,
		IN ULONG NumberOfBytes,
		IN BOOLEAN CacheEnable
		);

NTOSKRNL VOID MmUnmapVideoDisplay ( // для i386 враппер в MmUnmapIoSp
ace
		IN PVOID BaseAddress,
		IN ULONG NumberOfBytes
		);

// 将 frame 的范围标记为更改并进行相应的操作
NTOSKRNL VOID MmSetAddressRangeModified(
	PVOID StartAddress,
	DWORD Length
	);

// 在 NtMapViewOfSection 中调用
typedef enum _SECTION_INHERIT {
  ViewShare=1;
  ViewUnmap=2;
```

```
 }SECTION_INHERIT;

NTOSKRNL NTSTATUS MmMapViewOfSection(
        PVOID pSection,
        PEPROCESS pProcess,
        OUT PVOID *BaseAddress,
        DWORD ZeroBits,
        DWORD CommitSize,
        OUT PLARGE_INTEGER SectionOffset OPTIONAL,
        OUT PDWORD ViewSize,
        SECTION_INHERIT InheritDisposition,
        DWORD AllocationType,
        DWORD ProtectionType
        );

NTOSKRNL NTSTATUS MmUnmapViewOfSection(
        PEPROCESS Process,
        PVOID Address
        );

PVOID MmLockPagableImageSection(
        PVOID AddressWithinImageSection // same entry as MmLockPagableDataSection
        );

// 减少 StackLimit（堆栈增长）
NTSTATUS MmGrowKernelStack(
        PVOID CurESP //栈顶的地址
        );
```

I talk to the wind
My words are all carried away
I talk to the wind
The wind does not hear
The wind cannot hear.

King Crimson'69 -I Talk to the Wind

0b.结语
=============

就到这里吧。如果综合的来看所有这些描述，对内存管理器多少可以得到一些概念。遗憾的是，这些东西还远不能称之为完整。内存管理器，大概是最复杂和最重要的内核组件，

对其要进行完整的描述，我还得深挖不止十个八个的函数。但是主要的基本的东西我这里都写到了。对于进一步反汇编内核来说，这些应该是很有帮助的吧，谁知道呢... ;)

Best Regards, Peter Kosy aka Gloomy.
Melancholy Coding '2001.

mailto:gl00my@mail.ru

P.S. 我知道我的“大作”不可避免的会有错误。我将非常高兴的听取批评和建议。

附录

0c.某些未公开的系统调用
===

这里我描述了一些有用的 Zw/Nt 函数，这些函数可以在 USER 模式下或是驱动程序中调用（Zw 类的）。几乎所有这些函数都来自于Коберниченко的“Недокументированные возможности Windows NT”一书。再加上 Working Set 结构体的值，就可以描述用于 NtQueryVirtualMemory 的 MEMORY_WORKING_SET_INFORMATION。

```
NTSYSAPI NTSTATUS NTAPI NtAllocateVirtualMemory(
        HANDLE Process,
                OUT PVOID *BaseAddr,
                DWORD ZeroBits,
                OUT PDWORD RegionSize,
                DWORD                                   AllocationType,//
MEM_RESERVE|MEM_COMMIT|MEM_TOP_D
OWN
                DWORD Protect); // PAGE_XXXX...

NTSYSAPI NTSTATUS NTAPI NtFreeVirtualMemory(
        HANDLE Process,
                OUT PVOID* BaseAddr,
                OUT PULONG RegionSize,
                DWORD FreeType //MEM_DECOMMIT|MEM_RELEASE
        );

NTSYSAPI NTSTATUS NTAPI NtCreateSection(
        OUT PHANDLE Section,
        ACCESS_MASK DesirdAccess, //SECTION_MAP_XXX...
        POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
```

```
                    PLARGE_IBTEGER MaximumSize OPTIONAL,
                    DWORD SectionPageProtection, //PAGE_...
                    DWORD AllocationAttributes, //SEC_XXX
                    HANDLE FileHandle OPTIONAL // NULL - pagefile
                    );

typedef enum _SECTION_INHERIT {
  ViewShare=1;
  ViewUnmap=2;
  }SECTION_INHERIT;

NTSYSAPI NTSTATUS NTAPI NtMapViewOfSection(
        HANDLE Section,
        HANDLE Process,
        OUT PVOID *BaseAddress,
        DWORD ZeroBits,
        DWORD CommitSize,
        OUT PLARGE_INTEGER SectionOffset OPTIONAL,
        OUT PDWORD ViewSize,
        SECTION_INHERIT InheritDisposition,
        DWORD                                                   AllocationType,
//MEM_TOP_DOWN,MEM_LARGE_BAGE,MEM_AUTO_ALIGN=0x400000
00
        DWORD ProtectionType // PAGE_...
        );

#define UNLOCK_TYPE_NON_PRIVILEGED 0x00000001L
#define UNLOCK_TYPE_PRIVILEGED              0x00000002L

NTSYSAPI NTSTATUS NTAPI NtLockVirtualMemory(
        IN HANDLE ProcessHandle,
        IN OUT PVOID *RegionAddress,
        IN OUT PULONG RegionSize,
        IN ULONG UnlockTypeRequired
        );

NTSYSAPI NTSTATUS NTAPI NtUnlockVirtualMemory(
        IN HANDLE ProcessHandle,
        IN OUT PVOID *RegionAddress,
        IN OUT PULONG RegionSize,
        IN ULONG UnlockTypeRequiested
        );

NTSYSAPI NTSTATUS NTAPI NtReadVirtualMemory(
```

```
        IN HANDLE ProcessHandle,
        IN PVOID StartAddress,
        OUT PVOID Buffer,
        IN ULONG BytesToRead,
        OUT PULONG BytesReaded OPTIONAL
        );

NTSYSAPI NTSTATUS NTAPI NtWriteVirtualMemory(
        IN HANDLE ProcessHandle,
        IN PVOID StartAddress,
        IN PVOID Buffer,
        IN ULONG BytesToWrite,
        OUT PULONG BytesWritten OPTIONAL
        );

NTSYSAPI NTSTATUS NTAPI NtProtectVirtualMemory(
        IN HANDLE ProcessHandle,
        IN OUT PVOID *RegionAddress,
        IN OUT PULONG RegionSize,
        IN ULONG DesiredProtection,
        OUT PULONG OldProtection
        );

NTSYSAPI NTSTATUS NTAPI NtFlushVirtualMemory(
        IN HANDLE ProcessHandle,
        IN PVOID* StartAddress,
        IN PULONG BytesToFlush,
        OUT PIO_STATUS_BLOCK StatusBlock
        );

typedef enum _MEMORYINFOCLASS {
        MemoryBasicInformation,
        MemoryWorkingSetInformation,

        // 还有 class 2 - 这是 VAD 中的信息，我目前还不完全了解 VAD 结构体，也就不能写出
相应的 INFO 结构。

} MEMORYINFOCLASS;

typedef struct _MEMORY_BASIC_INFORMATION {
      PVOID BaseAddress;
      PVOID AllocationBase;
      ULONG AllocationProtect;
      ULONG RegionSize;
```

```
    ULONG State;
    ULONG Protect;
    ULONG Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;

#define WSFRAMEINFO_SHARED_FRAME 0x100
#define WSFRAMEINFO_INTERNAL_USE 0x4
#define WSFRAMEINFO_UNKNOWN 0x3

typedef struct _MEMORY_WORKING_SET_INFORMATION {
     ULONG SizeOfWorkingSet;
     DWORD WsEntries[ANYSIZE_ARRAY]; // is Page VA | WSFRAMEINFO...


} MEMORY_ENTRY_INFORMATION, *PMEMORY_ENTRY_INFORMATION;

NTSYSAPI NTSTATUS NTAPI NtQueryVirtualMemory(
     IN HANDLE ProcessHandle,
     IN PVOID RegionAddress,
     IN MEMORYINFOCLASS MemoryInformationClass,
     IN PVOID VirtualMemoryInfo,
     IN ULONG Length,
     OUT PULONG ActualLength OPTIONAL
     );
```

0d.附注及代码分析草稿
==

**** К MmCreateProcessAddressSpace ... ****
===

```
__fastcall MiTotalCommitLimit(PVOID pProcess, DWORD NumOfPages); // edx:ecx 有 statistic

dd MmTotalCommitLimit
dd MmTotalCommitedPages
```

如果 NumOfPages+MmTotalCommitedPages 不超过 Limit - 一切 OK，并只是简单的修正 statistic.

否则开始线程间的协作。

选择 time out 值（如果请求>=10 页，则为 20 秒），否则为-1 秒。接着填充某个结构体，大概是这个样子：

```
typedef struct _REQUEST_FOR_COMMITED_MEMORY{
        LIST_ENTRY ListEntry;
        DWORD PagesToCommit;
        DWORD Result;
        KSEMAPHORE Semaphore;
                }_REQUEST_FOR_COMMITED_MEMORY;
```

这个结构体（或链表的元素）被插入到全局结构体中的全局链表 ListOfRequest：

```
[Pre List Item]<->[Our List Item]<->[ListOfRequest]
```

```
typedef struct _COMMIT_MEMORY_REQUEST_LIST{
        KSEMAPHORE CommitMemorySemaphore;
        LIST_ENTRY ListOfRequest;
              }COMMIT_MEMORY_REQUEST_LIST;
```

之后对 CommitMemorySemaphore 使用 KeReleaseSemaphore 并等待 REQUEST_FOR_COMMITED_MEMORY 中带有 time out 的信号量。

如果未超出 time out 并因此 Result 不为 0，则再校验一次 Limit 并输出 OK（如果 limit 有问题——则所有都重新开始）。如果结果为 0，MiCouseOverCommitPopup。如果发生了 time out，分析如下：

如果 ListOfReques.Flink==&ListOfReques.Flink，也就是说所有的请求都在队列的尾部，则再一次等待信号量——并且已经没有 time out 了，因为不是我们的问题;)

如果 ListOfReques.Flink==&RequestForCommitedMemory.ListEntry，就是说队列中的下一个是我们的请求(???)。则从队列中收回请求，因为是从我们这里来的。

现在来看我们想看的几个页。如果>=10 则 MiCouseOverCommitPopup，否则 MiChargeCommitmentCantExpand，之后输出。

所有的操作都需要 cli sti，同时使用 FastMutex（进程的 10ch 偏移），在进程创建时调用这个函数不会进行此操作。

现在，MiCouseOverCommitPopup(PagesNum,CommitTotalLimitDelta);又做些什么呢——如果我们想要页数大于 128——则 ExRaiseStatus(STATUS_COMMITMENT_LIMIT); 如果小于则 IoRaiseInformationalHardError(STATUS_COMMITMENT_LIMIT,0,0);（这些函数都是公开的）。如果成功调用最后一个函数——则累加 statistic：

```
MiOverCommitCallCount++;
MmTotalCommitLimit+=CommitTotalLimitDelta;
MmExtendedCommit+=CommitTotalLimitDelta;
MmTotalCommittedPages+=PagesNum;
```

且不修正 MmPeakCommintment;

如果不成功但 MiOverCommitCallCount==0，所有都等于 statistic，否则 ExRaiseStatus(STATUS_COMMITMENT_LIMIT);

辅助函数：

DWORD NTOSKRNL RtlRandom(PDWORD Seed);

不奇怪，这个函数没有公开。该函数使用一个 128 个 DWORD 的表。在操作之后被此表和 Seed 被修正。可以看到，这给出了最大周期。

如果有两个 event
MmAvailablePagesEventHigh 和
MmAvailablePagesEventHigh.

MiSectionInitialization:
MmDereferenceSegmentHeader: это структура описанная выша
с добавленным
spinlock сверху.
创建线程 MiDereferenceSegmentThread

PsChargePoolQuota(PVOID Process,DWORD Type(NP/P),DWORD Charge);

[TO DO] -->> MmInfoCounters!!!! 使用相应的 NtQueryInfo...可以获得非常多有用的信息，П О С М О Т Р Е Т Ь !!!

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net
mailto:gl00my@mail.ru

董岩 译
http://greatdong.blog.edu.cn

Gloomy 对 Windows 内核的分析(对象管理器)

Inside WINDOWS NT Object Manager
======================================

(c) by Anathema

目录
==========

Inside WINDOWS NT Object Manager

00."对象化的" Windows NT
01.Windows NT 对象管理器
02.对象类型
03.Object Directory 与 SymbolicLink
04.对象的创建与对象的结构
05.Object Type 结构体
06.Дальнейшая жизнь объекта - простейший с
л у ч а й
07.句柄数据库
08.对象的安全性
09.命名对象
0a.对象的消亡
0b.其它管理器函数
0c.结语

附录

13.从用户模式下获取对象信息
14.某些与对象管理相关的系统服务

00."对象化的" Windows NT
========================

果您熟悉 Windows NT 体系结构，当然会知道在这个系统中资源都是以对象的形式存在的。这样集中了对资源的管理。实际上，所有的 Windows NT 子系统无论如何都要使用资源，也就是说都要经过对象管理器。对象管理器是 Windows NT 的子系统，它对对象提供支持。对对象的使用贯穿了整个操作系统，真正的“对象化”的 NT 对应用程序隐藏了 Win32 子系统，甚至于部分的隐藏了系统调用接口。

常，当提到 NT 的体系结构时，都会说到各种管理器（对象管理器、内存管理器等等）。所有这些管理器都是 Windows NT 内核（ntoskrnl.exe）的一部分。NT 的内核主要使用 C 语言编写的，每一个管理器本身都是一组函数，这些函数都被封装在相应的模块中。并不是所有的都定义了接口。管理器之间借助于函数调用相互协作（在内核内部）。一部分函数从内核中导出的并公开了使用方法以在内核模式下使用。

象管理器是一组形如 ObXXXX 的函数。实际上并不是所有的函数都是从内核中导出的。一般说来，对象管理器主要在内核中实现。例如，许多系统调用（NtCreateProcess、NtCreateEvent...）的处理都与这样或是那样的对象相联系。但是服务隐藏了对象管理器，不让我们的程序知道。甚至于驱动程序的设计人员未必需要大量直接的与对象管理器打交道。尽管如此，了解内核是如何实际工作的无疑非常重要，更为重要的是对象管理器还与安全管

理器密切相关。理解了对象管理器就揭开了 NT 中许多起初并不明显，但潜在具有的可能性……

01.Windows NT 对象管理器
================================

象管理器是 NT 内核中许多形如 ObXXX 的函数。某些函数并未导出并只在内核内部由子系统使用。对于在内核之外的使用，主要是通过未公开的函数。

ObAssignSecurity, ObCheckCreateObjectAccess , ObCheckObjectAccess,
ObCreateObject, ObDereferenceObject, ObFindHandleForObject,
ObGetObjectPointerCount, ObGetObjectSecurity, ObInsertObject,
ObMakeTemporaryObject, ObOpenObjectByName, ObOpenObjectByPointer,
ObQueryNameString, ObQueryObjectAuditingByHandle,
ObReferenceObjectByHandle, ObReferenceObjectByName,
ObReferenceObjectByPointer, ObReleaseObjectSecurity,
ObSetSecurityDescriptorInfo, ObfDereferenceObject, ObfReferenceObject

为了描述对象管理器，我们先来看一下什么是对象。

02.对象类型
================

象管理器本身就工作在某些对象之上。其中之一就是 object type。object type 用于描述所有对象的共同属性。object type 由函数 ObCreateObjectType(...)在内核初始化子系统的时候创建（也就是说是在系统初始化的时候）。我们所感兴趣的并不是 ObCreateObjectType(...)函数，因为其并未由内核导出。但是，在 object type 中的信息是非常重要的。关于这些我们少后再说。

概的讲，任何对象都可以分为两个部分：一部分包含管理上必需的信息（我们称之为首部），另一部分填充并确定创建该对象的子系统的必需信息（这是对象的 body）。对象管理器对首部进行操作，而实际上并不对对象的内容感兴趣，还有一些对象是对象管理器本身使用的。object type 正如其名字所示的那样，定义了对象的类型。每一个对象都有对其 object type 的引用，对象管理器也会非常频繁的使用对象的 body。object type 结构体的一个域是类型的名字。在 NT 下有以下各类型：

Type, Directory, SymbolicLink, Event, EventPair, Mutant, Semaphore,
Windows, Desktop, Timer, File, IoCompletion, Adapter, Controller, Device,
Driver, Key, Port, Section, Process, Thread, Token, Profile

Type 类型的 object type 不足一惧。要知道扼要的讲 object type 是用于管理器的对象，其和所有其它的对象是一样的，也有自己的 Type 类型（它自己的 Type 类型对象）。如果好好的想一下的话，这样做似乎也并不奇怪。下面，在 Type 这个话题下，主要来讲几个与 object type 相关的名词。

03.Object Directory 与 SymbolicLink
===================================

所有列举出的类型都是由不同的子系统创建的。我们现在讨论 Directory 和 SymbolicLink 类型，因为在对象管理器本身的工作中要用到这些类型的对象（这些类型是其在其自己初始化时创建的）。

NT 的对象可以有名字，还可以组织成树型的结构。Directory 对象正是用来组织这种结构的。它的用途非常简单——保存对其它对象的引用，例如，另外一个 Directory 对象。Directory 的 body 的结构非常简单:

```
typedef _DIR_ITEM{
        PDIR_ITEM Next;
        PVOID Object;
        }DIR_ITEM,*PDIR_ITEM;

typedef struct _DIRECTORY{
        PDIR_ITEM HashEntries[37];
        PDIR_ITEM LastHashAccess; //94h
        DWORD LastHashResult;      //98h
        }DIRECTORY,*PDIRECTORY;
```

路径用于保存命名对象。路径本身是 37 个 entry 的哈希表。表中的元素除保存着指向后面元素的指针之外，还保存着指向属于该路径对象的 body 的指针。哈希函数形式如下：

```
DWORD Hash(const char* str);
{
char *ptr=str;
int str_len=strlen(str);
char Sym;
int hash=0;
        while(str_len--)
                {
                Sym=*ptr++;
                ToUpperCase(&Sym);
                Sym-=' ';
                hash=hash*3+(hash>>1)+Sym;// умножение знаковое
                }
return hash%37;
}
```

当然，这里的代码只是给出程序的逻辑，并不是真正的实现。这个例子主要是基于对 ObpLookupDirectoryEntry(...)函数的分析写出的。最后的两个域反映出对哈希表的最后访问。

LastHashAccess 是指向最后访问的 entry 的指针。LastHashResult 包含成功查找到的单元。NT 想优化对路径的查找，而且如果在查找的结果中找到的entries中的一个entry不在表的开头，则这个表的元素就要被移到表的开头（认为后面对该对象的访问的几率最大）。

Directory 是对象，和其它的对象一样，它也可以有名字。如果再考虑到对象还有对自己所在目录的引用，则对可以画出树型的层级结构这个问题就能够理解了。

在 Windows NT 中包含着根目录 ObpRootDirectoryObject，Windows NT 的命名对象树就从这里开始。同时，还有 ObpTypeDirectoryObject 目录，它包含所有的 object type（使用这个树是为了防止 object type 重名）。内核表 ObpObjectTypes 中保存着指向 object type 的指针。再有，对于每一个类型的创建，子系统都保存着单独的指针。

SymbolicLink 这个 object type 用于保存字符串。在树中进行查找时它们被用作引用。例如，对象\??\C:（这表示，在 ObpRootDirectoryObject 中有对路径“??”的引用，在这个路径下包含着对象“C:”）就是 SymbolicLink 对象并包含字符串“\Device\Harddisk0\Partition1”。该对象格式如下：

```
typedef struct _SYMBOLIC_LINK{
    LARGE_INTEGER CreationTime; // время создания от 1601 года
        UNICODE_STRING Link;
    }SYMBOLIC_LINK,*PSYMBOLIC_LINK;
```

现在有了对管理器用到的类型和对象的一般概念，我们可以来研究更为具体的信息了。

04.对象的创建与对象的结构
================================

对象由 ObCreateObject 函数创建。这个函数由 ntoskrnl.exe 导出，下面给出其原型：

```
NTSTATUS NTOSKRNL
    ObCreateObject
     (
     KPROCESSOR_MODE bMode,              // kernel / user
     POBJECT_TYPE Type,                  // 对象类型
     POBJECT_ATTRIBUTES Attributes,      // 属性
     BOOLEAN bObjectMode,                // kernel/user 对象类型
     DWORD Reserved,                     // 函数未使用
     DWORD BodySize,                     // 对象 body 的大小
     DWORD PagedPoolQuota OPTIONAL,      // 如果为 0
     DWORD NonPagedPoolQuota OPTIONAL,// 则回收
     PVOID* pObjectBody                  // 指向 body 的指针
     );
```

参数 Reserved 可以简单的忽略。bObjectMode 定义了是否要从用户模式访问。PagedPollQuota 和 NonPagedPollQuota 是与这些对象相关联的分页池和非分页池的限额。限额记录了进程每一次其打开句柄的情况。进程有一个限额的阈值，其不能超越。POBJECT_ATTRIBUTES 结构体是公开的（ntdef.h 中有），但我这里还要将其给出，因为后面会

经常引用其中的域。

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
typedef OBJECT_ATTRIBUTES *POBJECT_ATTRIBUTES;
```

现在来集中讨论重要的域 RootDirectory 和 ObjectName。RootDirectory 是什么，现在我想已经能猜到了——将要包含对象（如果需要）的目录。ObjectName——见名知意。顺便说一下，可以不指明 RootDirectory，但在 ObjectName 中要指定对象的完整路径，这时对象将在起始于 ObpRootDirectoryObject 的树中创建。

关于安全方面的话我们单独讲，所以 Security 域嘛……暂时留下不提。剩下 Attributes 域。ntdef.h 中描述了各位的意义。下面给出这些属性。

```
#define OBJ_INHERIT                0x00000002L
#define OBJ_PERMANENT              0x00000010L
#define OBJ_EXCLUSIVE              0x00000020L
#define OBJ_CASE_INSENSITIVE       0x00000040L
#define OBJ_OPENIF                 0x00000080L
#define OBJ_OPENLINK               0x00000100L
```

结果，函数返回指向对象 body 的指针。

到了描述对象结构的时候啦。

```
+---------------+<-----------------------------> -??h
|               |
|    ........   |  --长度可变的结构体 (到 30h)
|               |
+---------------+<--对象首部--------------> 00h
|               |
|    Header     |  --标准的首部 18h 字节
|               |
+---------------+<--对象 body----------------> 18h
|               |
|               |
|    Body       | - тело объекта желаемого размера
|               |
+---------------+<------------------------------> 18h+BodySize
```

管理器经常要操作指向对象首部的指针。它返回给我们的是指向 body 的指针。首部之上还有一块区域，这块区域长度可变并依赖于所建对象的参数和类型。这个“帽子”可以保存下面这个结构体：

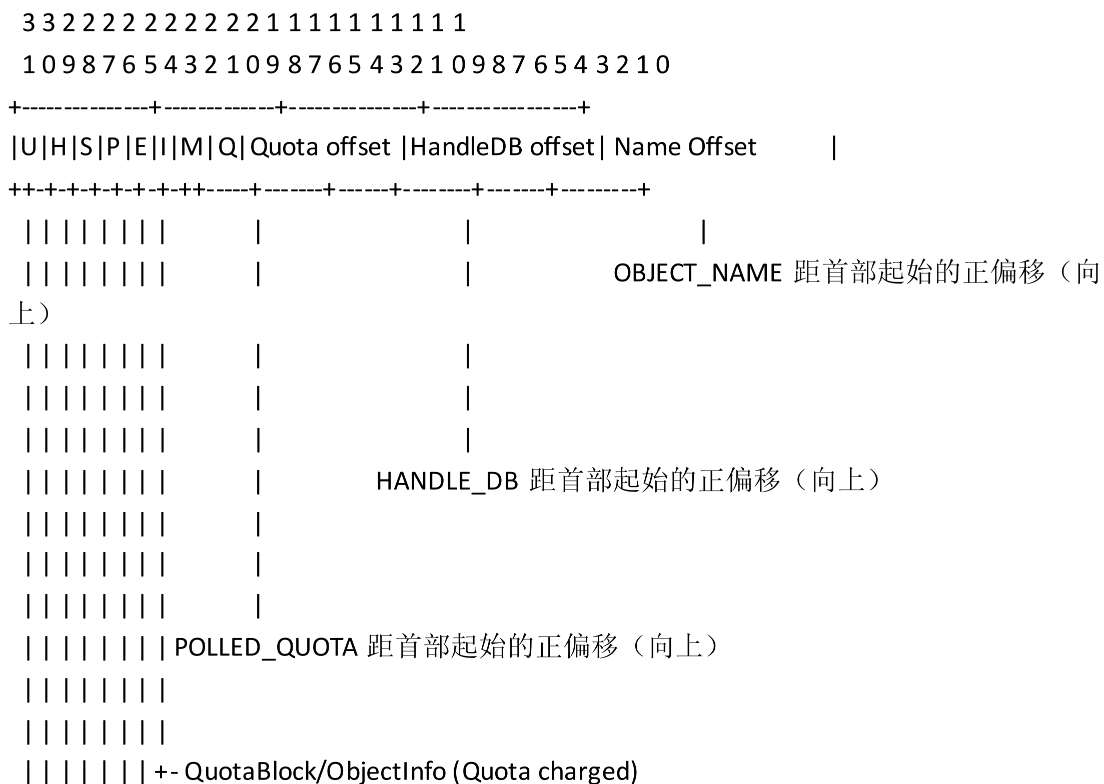
```
typedef struct _POLLED_QUOTA // 如果限额不等于 default 则增加
	{                               // 保存限额的消耗量
	DWORD PagedPoolQuota;
	DWORD NonPagedPoolQuota;
	DWORD QuotaInformationSize;    // PagedPollQuota 的总和
	PPROCESS_OBJECT pProcess;    // 拥有此对象的进程
	}POLLED_QUOTA;

typedef struct _HANDLE_DB      // 关于打开句柄的信息
	{
	union {
	PPROCESS_OBJECT pProcess;// 如果只有一个进程打开了
	                                        // 指针
	PVOID HandlesDBInfo;         //------------+--------
	}HanInfo;                          //                      |dd Num;2
	                                 //                     +-----------
	                                     //                     |dd pProcess1
	                                 //                     |dd Counter2
	                                 //                     +----------
	                                         //                     |dd pProcess2
	                                 //                     |dd Counter2
	                                     //                     +-----------
	                                         //                     |....
	DWORD Counter;                  // 句柄总数.
	}HANDLE_DB;

typedef struct _OBJECT_NAME
	{
	PDIRECTORY Directory; // 对象所属的路径
	UNICODE_STRING ObjectName; //对象名
	DWORD Reserved;            //对齐
	}OBJECT_NAME;

typedef struct _CREATOR_INFO
	{
	    LIST_ENTRY Creator;      // 闭合的链表 (包含着类型)
	DWORD UniqueProcessId;// 对象父进程的 ID
	DWORD Reserved;            // 对齐
	}CREATOR_INFO;
```

这个或是其它的结构体可以存在，也可以不存在，但顺序不能变。在注释中写明了这些结构体都是作什么用的了。它们中最后一个（哪怕只有一个）的后面就是标准的对象首部了。

```
typedef struct _OBJECT_HEADER
    {
    DWORD RefCounter;    //  对象的引用计数                    00
    DWORD HandleCounter;    //  句柄数目                       04
    POBJECT_TYPE ObjectType; //  对象类型                      08
    DWORD SubHeaderInfo; //  后面会讲                          0c
    UNION                                                      //    10
         {
            POBJECT_INFO ObjectInfo;
            PQUOTA_BLOCK pQuotaBlock;
       } a;
      PSECURITY_DESCRIPTOR    SecurityDescriptor;              //     14 Optional
    } OBJECT_HEADER;
```

先不讲这个联合体，以后再说。其它域的作用都很显然。对象管理器将统计打开的对象句柄以及对对象的引用，以此来知道何时删除对象的名字和 body（如果对象不是 permanent 的，也就是说没有设置 OBJ_PERMANENT 位）。还有对对象类型的引用和有关安全的信息。首部前面的结构体中的域在 SubHeaderInfo 域中也有体现。

SubHeaderInfo 中各个位的含义如下：



```
| | | | | | +-- 对象的 User/Kernel 模式
| | | | | +---- 是否有 CREATOR_INFO
| | | | +------ 对应于位 OBJ_EXCLUSIVE
| | | +-------- 对应于位 OBJ_PERMANENT.
| | |              用于该对象的创建，必须对应相应的权限
| | |
| | +---------- 存在 SecurityDescriptor
| +------------ HandleInfo 未初始化
+-------------- Unused/Reserved
```

*在 SubHeaderInfo 没有 CREATOR_INFO 的字节偏移，但因为这个结构体（如果有）总是最后一个，故指向它的指针可以获得，只需从首部指针中减去 sizeof(CREATOR_INFO)就可以了。

当我第一次开始反汇编对象管理器函数并看到 ObCreateObject(...)的时候，我断定这个函数做了使对象开始其"生活"的所有必需的工作。但是一个普遍的原则——直到最后再下定论（很好的原则），在这里也成立。实际上，对于前面描述的各域，这个函数只填充了结构体的最低限度的域。倒不如说这个函数在内存中把对象折腾了一番，捎带脚儿地填充了几个域。

再详细些。SubHeaderInfo 中的位：Q=1 总是，S=0 总是，H=1 如果存在 HANDLE_DB。OBJECT_NAME 中 Directory = NULL。CREATOR_INFO 中的 LIST_ENTRY 指向自己。POLLED_QUOTA 中 pProcess = NULL。HANDLE_DB 清零。HandleCounter = 0; RefCounter = 1; 其余的域以相应的 information entries 和对象类型中的信息来填充。除此之外，分配并填充 OBJECT_INFO 结构体，首部的 UNION {..}a 中的相应指针（达，当然所有的内存都是从 NonPaged pool 里分配的）。

```
typedef struct _OBJECT_INFO{
    DWORD Attributes;                  //00h OBJECT_ATTRIBUTES.Attributes
    HANDLE RootDirectory;              //04h
    DWORD Reserved;                    //08h - Unknown or Res.
    KPROCESSOR_MODE bMode;             //0ch
    BYTE    Reserved[3];               //0dh - Alignment
    DWORD    PagedPoolQuota;           //10h
    DWORD    NonPagedPoolQuota;        //14h
        DWORD    QotaInformationSize;//18h - 组 SID 的大小
                            //+ DACL 的大小（圆整后）
    PSECURITY_DESCRIPTOR SelfRelSecDescriptor;
                            //1ch - 指向 Self Relativ 的指针.
                            //Non Paed Pool 里的安全描述符
    PSECURITY_QUALITY_OF_SERVICE pSecQual;       //20h
    SECURITY_QUALITY_OF_SERVICE SecQuality;      //24h
                                                 //30h
    } OBJECT_INFO,*POBJECT_INFO;
```

顺便说一句，系统为 OBJECT_INFO 结构体维护了一个叫 ObpCreateInfoLookasideList 的

Lookaside list（见 DDK）。实际上 Reserved 域有时也会用到（在对象方法中作参数），但我尚未碰到过这个域的值不为 0 的情况。QuotaInformationSize 用在从 Paged 和 NonPaged pool 中移除限额用的。

05.Object Type 结构体
========================

这里我只描述一下 object type 的 body 的结构。

```
typedef struct _OBJECT_TYPE
{
  ERESOURCE TypeAsResource; //0x0 可用作资源
                            //34h
  PLIST_ENTRY FirstCreatorInfo;//38h 我注意到了这个结构体
  PLIST_ENTRY LastCreatorInfo; //3ch 只是用于 object type
  UNICODE_STRING TypeName;    //40h 类型名
  DWORD Unknown2[2];          //48h
  DWORD RefCount;             //50h 该类型对象的计数
  DWORD HanCount;             //54h 该类型句柄的计数
  DWORD PeakRef;              //58h 对象的峰值
  DWORD PeakHandles;          //5ch 句柄的峰值
  DWORD Unknown3;             //60h
  DWORD AllowedAttributeMask;//64h 可能的属性 0 - 允许所有的
  GENERIC_MAPPING GenericMapping;//68 отображение родовых пра
в    на специальные
  DWORD AllowedAccessMask;    //78h (ACCESS_SYSTEM_SECURITY 总是设置的)
  BOOLEAN bInNameSpace;       //7ch 这个类型的对象在对象路径中
                              //     可能我会弄错，但也类似.
  BOOLEAN bHandleDB;          //7dh 是否包含对象句柄的信息(HANDLE_DB)
  BOOLEAN bCreatorInfo;       //7eh ----//---- CreatorInfo + 38h 处的链表
  BOOLEAN Unknown5;           //7fh
  DWORD Unknown6;             //80h 如果 !=0 则在 NpSuccess 里创建
  DWORD PagedPoollQuota;      //84h default
  DWORD NonPagedPollQuota;    //88h         限额
  PVOID DumpProcedure;        //8ch 原型未知 (?)
  PVOID OpenProcedure;        //90h 原型已知
  PVOID CloseProcedure;       //94h 原型已知
  PVOID DeleteProcedure;      //98h 原型已知
  PVOID ParseProcedure;       //9ch 原型已知
  PVOID SecurityProcedure;    //a0h 原型已知
            // 可以有 4 种调用情况:
                  //0-set sec_info, 1-query descriptor, 2-delete, 3-assign

  PVOID QueryNameProcedure;   //a4h 原型已知
```

```
    PVOID Tag;                          //a8h 通过高层次信息判断
          // 这应该是方法 OkayToCloseProcedure;
               // 实际上, 对于所有的对象我都发现在这个地址上有四字符的类型 Tag,例如 Dire
(Directory)
} OBJECT_TYPE,*POBJECT_TYPE;
```

我就不给出已知方法的原型了，因为这里不大用得到。于是，类型种包含了类型名，操作对象的函数，还有属于该类型的用于对象的一般信息。结构体的域都已有注释，不再细说。

我们将视线转向偏移 0 处的 ERESOURCE 结构体（可以在 NTDDK.H 中找到）。换句话说，这个对象可以用作资源。对象管理器使用 ExAcquireResourceExclusiveLite 函数（见 DDK）从 object type 中取得资源。PagedPollQuota 和 NonPagedPollQuota 将限额指定为默认值。如果我们使用限额值为零来调用 ObCreateObject 函数（或与默认的限额值相同）而且如果没有设置 OBJ_EXCLUSIVE 且如果 QuotaInformationSize 小于 0x800，则在创建对象时将不会创建 POLLED_QUOTA 结构体。AllowedAttributeMask 和 AllowedAccessMask 域指定了属性的掩码和允许的访问。最后有指向方法的指针，这些方法在定义的时候调用。例如，在增加句柄的时候，在这个进程句柄的基址里调用 OpenProcedure 函数。这并不是说这些域就是我们想象的那样（以及我从Х е л е н　К а с е р（译注：我想是 Helen Custer）的关于 WIndows NT 的书中读到的那样）是必需的。但是它们能够扩展对象的功能。下面我来讲调用某些方法的条件。

06.Дальнейшая жизнь объекта - простейший случ
а й
===

ObCreateObject 创建对象，但没有什么该对象本质上的东西。没有它的句柄，不能用名字打开它（ObCreateObject 函数并不向对象树中添加对象）。我们回来看一下对象管理器导出的函数。下面的三个函数很令人好奇：ObOpenObjectByName，ObOpenObjectByPointer，ObInsertObject。第一个函数先不说。那是说第二个呢还是第三个？ObOpenObjectByPointer 自然应该讲，可是 ObInsertObject 平时更少被提到，所以我们从它开始。

```
NTSTATUS NTOSKRNL
        ObInsertObject(
                PVOID pObject,                          //body
                PACCESS_STATE pAccessState OPTIONAL,
                ACCESS_MASK Access,
                DWORD RefCounterDelta OPTIONAL,      //0- default (т.е. 1)
                PVOID     OUT *ObjectExist OPTIONAL, //如果已经存在
                PHANDLE OUT Handle                      //句柄
                        );
```

函数的逻辑十分明显。它从填充好的先前的 OBJECT_INFO 结构体体得到主要的信息是其即将用到的从对象首部中获得的一个指针（指向对象首部的指针可以简单的用 body 指针减去 18h）。函数的工作方式有两种：一，对象没有名字且没有在类型中设置 bInNameSpace 标志；二，设置了该标志或是有名字。我们来看这两种情况。

如果对象没有名字，则调用 ObpCreateUnnamedHandle(..)。后面，我将引用一个内部函数而不给出其原型，其原型是已知的。这个函数本身是以许多内部函数为基础的，这些内部函数共同完成了该函数的工作。

当前进程记录了限额。在这种情况下，填充 a.pQuotaBlock 域并清 SubHeaderInfo 中的 Q 位。现在对象“填充好了”限额。a.pQuotaBlock 指向与每一个进程相联系的 QUOTA_BLOCK 结构体。我们现在先不研究对象－进程的结构，我想下面这个结构体值得一提。

```
typedef _QUOTA_BLOCK{
                    DWORD KSPIN_LOCK QuotaLock;
                            DWORD RefCounter; // 用于计数该 block 的进程
                            DWORD PeakNonPagedPoolUsage;
                    DWORD PeakPagedPoolUsage;
                            DWORD NonPagedpoolUsage;
                            DWORD PagedPoolUsage;
                        DWORD NonPagedPoolLimit;
                    DWORD PagedPoolLimit;
                        DWORD PeakPagefileUsage;
                        DWORD PagefileUsage;
                            DWORD PageFileLimit;
                         }QUOTA_BLOCK,*PQUOTA_BLOCK;
```

我想各域的名字已经很清楚的说明了其用途。行，我们接着往下进行。增加句柄的数目并恢复 HANDLE_DB 中的信息，如果有的话。HANDLE_DB 本身是个结构体，保存了对打开对象的各进程的引用。它为每一个进程都维护了一个计数器。在第一遍中将 HanInfo 与此结构体相联结显示了对“懒惰”原则的遵守。如果对象只是从一个进程中打开，则联结本身就是指向这个进程的指针。不可命名的对象可以是 exclusive 的（属性中的 OBJ_EXCLUSIVE 位）。这样控制了只能从一个与 POLLED_QUOTA 中的域对应的进程中打开对象。我们记得，这个结构体总是存在于 exclusive 的对象中。这样，进程就独自占据了对象——如此就没有了控制访问的问题。合情合理的，对象独占的句柄能够继承（OBJ_INHERIT），并被管理器检查。如果 CREATOR_INFO 是必需的信息，则要有 CREATOR_INFO 链表，object type 有相应的域指向它。object type 本身也在这个链中。实际上，我发现 bCreatorInfo 位只在类型对象“Type”中设置。这表明链表只用于对象类型。对于类型已定的对象，则不含此链表。好了，只剩下描述 ObpIncrementUnnamedHandleCount(..)函数的主要内容的逻辑了。在该函数执行时还会调用 OpenProcedure（如果有）。

当上述操作完成时，会使首部中的计数器 RefCounter 增加 RefCounterDelta+1。最后，使用 ExCreateHandle 创建句柄。在成功创建句柄后，就离开相应的 OBJECT_INFO 结构体，这个结构体役期已满，不再需要……

07.句柄数据库
======================

进程（也是个对象，这个以后再说）有个与对象相联系的句柄数据库。知道了对象的句柄就可以很容易的对对象进行访问，而同时也出现了继承和复制的问题。我们来详细研究一

下句柄数据库。

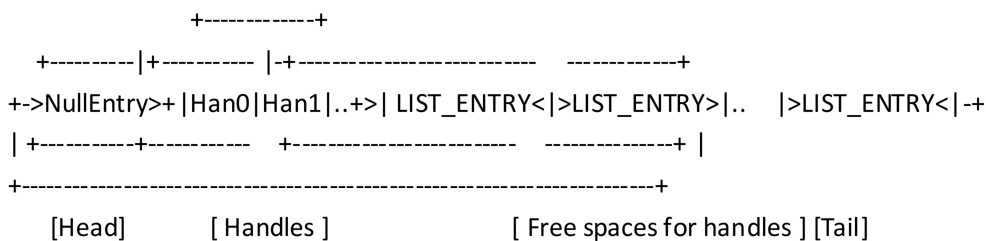
我们不在'thread'和'process'对象的具体格式上做停顿，演示如何得到指向句柄数据库的指针（自然，要在内核模式下）。

```
mov eax,large fs:124h ; Got current thread body pointer
mov eax,[eax+44h]          ; Got current process body pointer
mov ecx,[eax+104h]        ; Got handle_db header pointer !
```

数据库的结构可以分成两部分——header 和 body（实际上就是数据库本身）。header 的形式如下：

```
typedef struct _HANDLE_DB_HEADER{
        WORD    Unknown0[2];       //0x0 - эти поля всегда были 0
            DWORD Unknown1;              //0x4 - -----..-----
            SPIN_LOCK SpinLock;      //0x8
        PVOID Head;                   //0xc
            PVOID Tail;                    //0x10
            DWORD HandleCount;        //0x14
            PPROCESS pProcess;        //0x18
            DWORD ProcessUniqueId;//0x1c
            WORD GrowDelta;              //0x20
            LIST_ENTRY HandleTableList; //0x24  系统有个句柄数据库链表. 链表的
entry - HandleTableListHead
            KEVENT    EventObj;            //0x2c
            KSEMAPHORE SemaphoreObj; //0x3c
           }HANDLE_DB_HEADER,*PHANDLE_DB_HEADER;
```

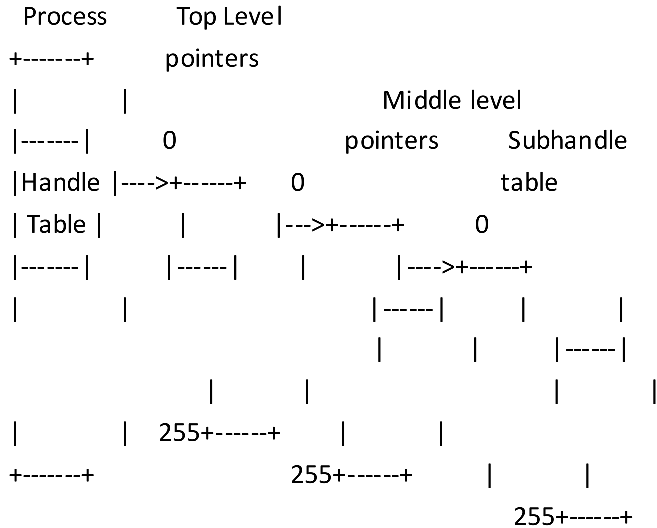
数据库本身形式如下：



DB 首部中的 Head 指向数据库的起始位置，Tail 指向末尾。数据库初始化时，数据库的全部内存都填充为循环链表（除链表外没别的东西，也就是说每一个元素都是 LIST_ENTRY）。并且链表是从左到右依次建立的。当向数据库中添加句柄时，句柄被加在 NullEntry 之后。head 有指向链表中空闲位置的指针，句柄就在此处添加。当没有空闲位置时（链表的 head 指向了自己），就将数据库扩展 GrowDelta 个位置。当然，在句柄数据库工作时会产生相应的限额的“装填”。所有这些都很简单……

****

注意！所有上述的信息主要是针对 Windows NT 4.0 的。在 Windows NT 2K 下句柄数据库的结构变了样子。现在这个表是三级的。



现在没有必需改造的句柄表了。不用再创建一个指针块并添加子句柄（subhandle）表了。

这样，OBJECT_HANDLE 结构体（见下面）没有改变。只是指针中的第 31 位用作了繁忙或加锁的指示器，通常情况下可抛开（因为所有的指针都属于内核空间，第 31 位不会改变指针指向——所有的地址都属于高于 0x80000000 的地址空间）。我给出某些内部函数的伪码，来说明 NT5.0 下句柄数据库的运作。

```
ExMapHandleToPointer(Table,Handle)
{
        TableEntry=ExpLookupHandleTableEntry(Table,Handle);
        if(!TableEntry)return 0;
        return ExLockHandleTableEntry(Table,TableEntry);
}

ExpLookupHandleTableEntry(Table,Handle)
{
level1=(Handle>>18) & 0xff;
level2=(Handle>>10) & 0xff;
level3=(Handle>>2)   & 0xff;

if(Handle&0xfc000000)return 0;

pPtr=Table->0x8;

pPtr=pPtr+level1*4;
```

```
if(pPtr)
        {
        pPtr=pPtr+level2*4;
        if(pPtr)
                {
                return pPtr+level3*8;
                }
        }
return 0;
}

ExLockHandleTableEntry(Table,TableEntry)
// Table not used

{

ObjectPointer = TableEntry->0x00;
if (!ObjectPointer) return 0;

if (ObjectPointer > 0)
        {
        NewObjectPointer = ObjectPointer | 0x80000000;
        //xmpxchg         NewObjectPointer,ObjectPointer in ObjectPointer
        }
else {
        // wait logic
        }
return 1;
}
```

****

为了完整，我给出用于句柄数据库的主要内部函数的原型。

用于创建数据库的基本函数：

```
NTSTATUS ExCreateHandleTable(
          PPROCESS Process,
          WORD Size,          //Optional
          WORD GrownDelta //Optional
          );
```

这个函数用于数据库的扩展：

```
NTSTATUS ExpAllocateHandleTableEntries(
        PHANDLE_DB_HEADER HandleDB,
        PLIST_ENTRY Head,
        DWORD OldSize,    // 在元素中
        DWORD NewSize
        );
```

更高层的函数：

```
NTSTATUS ObInitProcess(
        PPROCESS pFatherProcess, //如果!=0 - 数据库是复制的.
                                 //否则 - 是创建的
                              //包含 audit
        PPROCESS pProcess
        )
```

对象句柄本身是怎样的结构呢？看这里：

```
typedef struct _OBJECT_HANDLE{
        PVOID ObjectHeaderPointer,
        ACCESS_MASK GrantedAccess
        }OBJECT_HANDLE,*POBJECT_HANDLE;
```

正如我们看到的——句柄只是简单的一个指向对象首部的指针，再加上一个访问掩码。是的，ObjectHeaderPointer 还保存了某些信息。问题在于所有对象的首部的地址都是 8 字节对齐的。这样，对于对对象的访问只需使用 ObjectHeaderPointer 的位[31:3]。低三位是 OBJECT_HANDLE_INFORMATION 结构体中的 HandleAttributes（见 DDK 中的 ObReferenceObjectByHandle 函数）。各个位的值：0x01 - protect from close，0x02 - inherit，0x04 - generate on close。因此，使用句柄能进行有效的寻址。返回给用户的句柄值，左移两位后等于 OBJECT_HANDLE 结构体在数据库中的序号。在 ntdef.h 中，OBJ_HANDLE_TAGBITS 定义的附近可以看到以下几行：

```
//
// Low order two bits of a handle are ignored by the system and available
// for use by application code as tag bits.   The remaining bits are opaque
// and used to store a serial number and table index.
//
```

怎么样，全明白了吧。

如果您还没忘的话，我们只是分析过 ObInsertObject 的最简单的情况，即非命名对象的情况。如果一定要将对象要放入命名树中会怎样呢？简单讲就是对命名树进行操作，最终将对象放入目录中，这样就可以用名字打开对象了。如果这个目录是系统树的根目录，则这个

对象就对所有进程可见。如果这个目录是某个进程的——则只对该进程可见（通常所有这样的对象都在系统树中创建，因为大多数系统服务都使用系统树）。对此，为了展示进程的完整格局，我们先来分析清楚某些问题……

08.对象的安全性
========================

之前，我一直都在回避这个问题，因为在前面不涉及这方面的东西是可以的。但是，要向下继续的话，就要知道一些了。幸运的是，这里要讲的大多数结构体都在这个或那个头文件中有描述，是公开的。但是，我觉得将这些必需的信息结合在一起是合乎情理的，因为下面我将经常引用这些结构体。有时，我会讲到一些未公开的东西。在创建时，帐户记录生成一个随机的 SID，这样就保证了 SID 的随机性。

为了识别认证用户和组，安全系统使用了 SID（Security IDentifier）。

```
typedef struct _SID_IDENTIFIER_AUTHORITY {
    BYTE   Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;

typedef struct _SID {
        BYTE   Revision;
        BYTE   SubAuthorityCount;
        SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
        DWORD SubAuthority[ANYSIZE_ARRAY];
   } SID, *PISID;
```

当前 Revision 的值为 1。SubAuthorityCount 的最大值为 15。SubAuthority 域最后一个值为 RID（Relativ. Id.）。从文本的角度看，NT 的 SID 的形式为 S-1-A-SA1-SA2-...-RID。其中 S-1 表示 SID rev.1，A - IdentifierAuthority，SA? - SubAuthority，RID — SubAuthority 的最后一项。在 NT 中有标识的预定义，同时也有 RID 值的预定义。例如，RID 等于 500 表示帐户 administrator。详细信息可参考 WINNT.H。

用户可被授予某种特权级 SeXXXPrivilege，每一个特权级在系统中都是 8 位数字的 LUID（Local Unique Id.）。起初，每一个特权级都是一个文本字符串（特权级的文本表示可以参见 WINNT.H）。

这个信息不依赖于环境。在用户进入系统时，LSA（Local Security Administrator）创建访问令牌（TOKEN），访问令牌包含着用于安全检查的重要信息。该结构体形式如下：

```
typedef struct _ACCESS_TOKEN{
     char SourceName[8];       //0 Source of token'a
     LUID SourceIdentifier; //8
        LUID TokenId;              //10
        LUID   AuthenticationId;//18
     LARGE_INTEGER ExpirationTime; //20
```

```
LUID    ModifiedId;                    //28  修改 token'a 时改变
DWORD NumberOfUserAndGroupSids;//30
DWORD NumberOfPrivileges            //34
DWORD Unknown;                         //38
DWORD DynamicCharged;                //3c
DWORD DynamicAvailable;             //40
DWORD NumberOfOwnerSid               //44
PVOID SidDB;                          //48
PSID    PrimaryGroup;                  //4c
    PLUID_AND_ATTRIBUTES TokenPrivileges; //50
    DWORD Unknown1;                          //54
PACL DefaultDacl;                              //58
TOKEN_TYPE TokenType;                            //5c  原始的还是
                                                        // IMPERSONATION
SECURITY_IMPERSONATION_LEVEL ImpLevel; //60 IMPERSONATION
    DWORD UnknownFlag;                                 //64
DWORD Tail[ANYSIZE_ARRAY];                        //????
}ACCESS_TOKEN,*PACCESS_TOKEN;
```

数据库 SidDB 大约是下面这个样子：

```
                    +-TokenUser----+
                       |00 PSID            |
                 |04 Attributes |
                       +-TokenGroups--+
                       |....                 |
                       |                     |
                       |                     |
                       +--------------+
                          <--NumberOfUserAndGroupSids

                       +-OwnerSid-----+
                       |                     |<--NumberOfOwnerSid
                       +--------------+
                       |....
```

我再给出与之相应的未公开的函数，其用于取得'Token'对象。

```
PACCESS_TOKEN NTOSKRNL PsReferenceImpersonationToken(
          KTHREAD * Thread,              //IN
          PBOOLEAN* CopyOnOpen,        //OUT
          PBOOLEAN* EffectiveOnly, //OUT
          SECURITY_IMPERSONATION_LEVEL* ImpersonationLevel //OUT
```

);

PACCESS_TOKEN NTOSKRNL PsReferencePrimaryToken(KPROCESS* Process);

我就不再讲访问令牌结构体都保存了那些信息了（现在还没到讲安全描述符的时候）。同时我也不会解释 Impersonation 的机制。指向 TOKEN 的指针放在'Process'对象 body 的偏移 108h 处。

当使用 ObCreateObject(..)创建对象时，所穿参数实际上是指向 OBJECT_ATTRIBUTES 的指针，在其中有 SecurityDescriptor 和 SecurityQualityOfService 域——指向公开的结构体的指针。

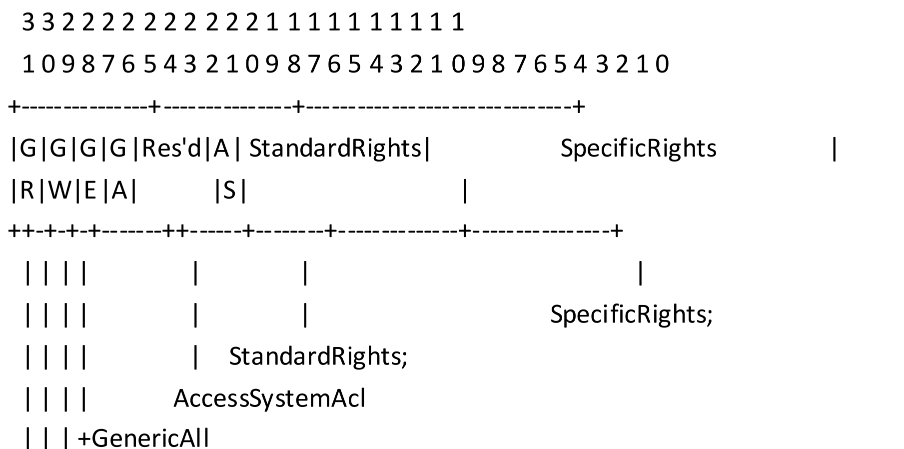
SecurityQualityOfService 包含关于服务客户的 Impersonation Level 的信息，我们就不多说了。

SecurityDescriptor 描述了对象的安全性——这是需要说的。

```
typedef struct _SECURITY_DESCRIPTOR {
        BYTE    Revision;
        BYTE    Sbz1;
        SECURITY_DESCRIPTOR_CONTROL Control;
        PSID Owner;
        PSID Group;
        PACL Sacl;
        PACL Dacl;
    } SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

Control 指定了不同的标志，其中一个指示描述符 Self Relative（就是说其中所有的指针都是相对于结构体首部的）。详细描述见 WINNT.H。域 Owner 和 Group 的值的意义是显然的。Dacl——枚举用户和组的链表，通过其来允许或是禁止对对象的访问。Sacl 与 audit 密切相关。与 ACL 有关的结构体在 WINNT.H 中都有很好的描述。所以，描述用于各种用户的对象访问权限的信息与对象密切相联（在对象首部中有指向安全描述符的指针）。

在取得对对象的访问权时（如 ObInsertObject）会被赋予 ACCESS_MASK。通常，这时要使用相应的类型定义 STANDARD_RIGHTS_READ 等等。现在，重要的是来看一下这个信息，该信息的位结构体是下面这个样子：



```
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---------------+---------------+-------------------------------+
|G|G|G|G|Res'd|A| StandardRights|          SpecificRights          |
|R|W|E|A|       |S|                |                                  |
++-+-+-+-------++------+--------+-------------+----------------+
 | | | |         |         |                          |
 | | | |         |         |              SpecificRights;
 | | | |         |   StandardRights;
 | | | |       AccessSystemAcl
 | | | +GenericAll
```

```
| | +--GenericExecute
| +----GenericWrite
+------GenericRead
```

Generic 类的位是为了方便程序员（都是标准的而且很简单）。实际上，在对象类型结构体中有 GenericMapping 域，该域包含着关于在 SpecificRights（特殊权限——此位的含义依赖于对象类型）中转换 Generic 位的信息。StandardRights，从名字就可看出，它对于所有的对象有相同的含义。对象句柄有个 GrantedAccess 域，这样，在使用句柄时不必总是解析安全描述符，这里的信息就足够了。除此之外，对象类型还包含着 AllowedAccessMask，这样就可以控制对于对象类型的上下文所进行的可能的访问。

在内部，NT 使用了下面这样一些结构体，如 ACCESS_STATE。

```
typedef struct _ACCESS_STATE {
    LUID OperationID;
    BOOLEAN SecurityEvaluated;
    BOOLEAN GenerateAudit;
    BOOLEAN GenerateOnClose;
    BOOLEAN PrivilegesAllocated;
    ULONG Flags;
    ACCESS_MASK RemainingDesiredAccess;
    ACCESS_MASK PreviouslyGrantedAccess;
    ACCESS_MASK OriginalDesiredAccess;
    SECURITY_SUBJECT_CONTEXT SubjectSecurityContext;
    PSECURITY_DESCRIPTOR SecurityDescriptor;
    PVOID AuxData;
    union {
        INITIAL_PRIVILEGE_SET InitialPrivilegeSet;
        PRIVILEGE_SET PrivilegeSet;
        } Privileges;

    BOOLEAN AuditPrivileges;
    UNICODE_STRING ObjectName;
    UNICODE_STRING ObjectTypeName;
    } ACCESS_STATE, *PACCESS_STATE;

typedef struct _SECURITY_SUBJECT_CONTEXT {
     PACCESS_TOKEN ClientToken;
     SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
     PACCESS_TOKEN PrimaryToken;
     PVOID ProcessAuditId;
     } SECURITY_SUBJECT_CONTEXT, *PSECURITY_SUBJECT_CONTEXT;
```

正像我们看到的，这个结构体在其中结合了在访问对象时所有用于控制对象访问的必需

的信息。我给出几个与之相关的未公开函数。

```
// 填充 ACCESS_STATE 结构体
// 留下了一个关于优先级和对象的信息未填充

NTSTATUS NTOSKRNL SeCreateAccessState(
                    PACCESS_STATE AccessState,
                    PVOID AuxData,
                    ACCESS_MASK Access,
                    PGENERIC_MAPPING GenericMapping
                    );

NTSTATUS NTOSKRNL SeDeleteAccessState(PACCESS_STATE AccessState);

// 填充 SUBJECT_CONTEXT,  用在 SeCreateAccessState

void NTOSKRNL SeCaptureSubjectContext(
                    PSUBJECT_CONTEXT SubjectContext
                    );
```

在 DDK 中同时描述了 ObGetObjectSecurity 和 ObReleaseObjectSecurity 函数。其中第一个函数使用代号 1 调用了方法 SecurityProcedure（参见对象类型结构体的格式）。如果有安全描述符，则这个方法负责收回。否则就通过对象首部来取得描述符。

09.命名对象

==========================

我们来研究 ObInsertObject 的第二个分支，该分支用于处理命名对象。如果对象是未命名的或是 exclusive 的，则其不需要安全描述符。否则可以通过名字访问对象。

在开始部分，函数创建 ACCESS_STATE 结构体并在以 OBJECT_INFO.RootDirectory（这个结构体当时是存在的）为根在树中查找这个名字。查找是通过 ObpLookupObjectName 函数进行的。其工作逻辑十分显然——解析相应对象的 Directory 和 SymbolicLink。一般 ObpLookupObjectName 不只用于查找，还用于在树中创建对象。在选择路径时同时要用安全检查点（ObpCheckTraverseAccess 函数）在树中验证对象的访问权。到目前，暂时还没有结束对路径的解析或是在路径中没有遇到对象管理器不知道的对象（这叫做另一类的名字空间）。这时调用方法 ParseProcedure（如果有的话），其中的一个参数是指向未被选中的路径的指针。下面的分析要用到这个方法。使用 Parse 方法可以查找整个名字空间，但对象可能只在其“亲本”的树中。ParseProcedure 方法只用于查找并在在树中创建对象时被忽略（如果它没有指向默认的函数 ObpParseSymbolicLink)。但符号链接对于对象描述符来说是自有的，而且总是可以被处理的)。如果发生了名字冲突（对象已存在)，则 ObInsertObject 的进一步的动作就依赖于 OBJ_OPENIF 标志。这个标志可被译为 open if exist。其它的词将打开现有的对象。一般情况下正常的控制流在对象创建的最后会调用未公开函数 ObAssignSecurity（其“逆”函数 ObGetObjectSecurity 是公开的）。这个函数被封装在了公开函数 SeAssignSecurity(...)中。

```
BOOLEAN NTOSKRNL ObAssignSecurity{
        PACCESS_STATE AccessState,
        PSECURITY_DESCRIPTOR OPTIONAL OldSecurityDescriptor,
        PVOID Object,
        POBJECT_TYPE Type);
```

参数 SecurityDecriptor 只用作传递给 SecurityProcedure 方法的参数，函数用代号 3 来调用此方法。所有的必需的安全管理信息都在 ACCESS_STATE 结构体中。在该函数完成后，对象首部中的 SecurityDescriptor 域就被填充好（对于非命名对象则无此项工作）。然后调用内部函数 ObpCreateHandle。尽管该函数未被导出，但是为了比较函数 ObpCreateHandle 和 ObpCreateUnnamedHandle，我在这里给出其原型。

```
NTSTATUS ObpCreateHandle(
    DWORD SecurityMode,    //  通常==1，访问在这里被检查
      PVOID Object,
      POBJECT_TYPE Type,
      PACCESS_STATE AccessState,
      DWORD RefCounterDelta,
      DWORD Attributes,
      DWORD OPTIONAL UnknErrMode,
      KPROCESSOR_MODE bMode,
      PVOID Object,
      PHANDLE Handle
      );

NTSTATUS ObpCreateUnnamedHandle(
    PVOID Object,
      ACCESS_MASK Access,
    DWORD RefCounterDelta,
    DWORD Attributes,
    KPROCESSOR_MODE bMode,
    PVOID Object,
    PHANDLE Handle
            );
```

两个函数做的工作实际上是相同的，最终都要调用 ExCreateHandle(..)。但是，从原型中可以直观的看出，第一个函数处理的是安全描述符，创建句柄的过程我已经描述过，我们都已经知道了。最后，OBJECT_INFO 结构体被释放。我已经说过这些结构体都是分配在 LookasideList 中的，现在明白是为什么了吧。在系统中经常会对象被创建和删除，对 OBJECT_INFO 的创建工作会大量进行（这些结构体从对象被创建起就存在于每一个对象中）。LookasideList 专门用于优化这类经常用到的内存分配/回收操作（见 DDK）。

0a.对象的消亡

=================

如果对象管理器不再需要某对象，或对象不是 permanent（OBJ_PERMANENT 位）的，则管理器就将对象删除。这里对象的消亡分为两步。在首部中有两个域，RefCounter 和 HandleCounter。在句柄关闭时（ZwClose）会依次调用 ExDeleteHandle 和 ObpDecrementHandleCount。第一个函数只是从句柄数据库中删除句柄（释放的空间被返回给链表的空闲位置）。ObpDecrementHandleCount 包含着关于句柄打开数量的信息，同时保存着句柄对象数据库的信息（HANDLE_DB）。除此之外，在这些函数中还要调用方法 CloseProcedure。如果关闭的是最后一个句柄且对象不是 permanent 的，则调用 ObpDeleteNameCheck 函数，该函数从对象名树中将有关该对象的信息删除。这时，以代号 delete 调用方法 SecurityProcedure。但是对象继续存在——它只是不再可见。

除此之外，在首部中还有对对象的引用计数域，这个域通过 ObDereferenceObject 函数来递减。(在递增/递减句柄的引用计数时该域也同时递增/递减，但可以修改对象的引用计数而不改变句柄的数量)。如果引用计数值达到了零，则一般会在删除队列中添加这个对象（这要依情况而定，对象可以被立即删除）。这时，RefCount 和 HandleCount 就扮演了 LIST_ENTRY 的角色。对于这些域，我没有在对象首部的格式中将它们结合起来，以不使内容过快的复杂化。在直接删除对象后就要用代号 2 调用方法 SecurityProcedure 和 DeleteProcedure。现在所提到的所有的内核函数，除了 ObDereferenceObject 都是内部函数，它们的原型我就不给出了，都是已知的。

0b.其它管理器函数
============================

实际上，对 ObInsertObject 函数的研究打开了对象管理器逻辑的主要部分。我们来简要的研究一下剩下的管理器导出函数。

DDK 文档中有以下函数：

ObReferenceObject, ObDereferenceObject, ObGetObjectSecurity, ObReleaseObjectSecurity, ObReferenceObjectByHandle, ObReferenceObjectByPointer.

从已知的信息来看，在这些函数里的管理器的工作逻辑都是明显的。其工作就是处理对象首部、句柄数据库和同步内部对象。剩下的一些函数我们不再讨论：

ObCheckCreateObjectAccess , ObCheckObjectAccess,
ObFindHandleForObject, ObGetObjectPointerCount,
ObMakeTemporaryObject, ObOpenObjectByName, ObOpenObjectByPointer,
ObQueryNameString, ObQueryObjectAuditingByHandle,
ObReferenceObjectByName, ObSetSecurityDescriptorInfo

关于所有这些函数的信息都是已知的。

// 将完整的对象名返回到参数中

```
// 该参数就是用于保存 UNICODE_STRING 和名字的缓冲区
// 通常开始先不用参数 Result 调用，以获得
// 缓冲区的长度。
// 如果定义了方法 QueryName - 则调用它

NTSTATUS NTOSKRNL ObQueryNameString(
        PVOID Object,
        PUNICODE_STRING Result OPTIONAL,
        DWORD Len,
        PDWORD RetLen OPTIONAL
        );

// 调用 ObpLookupObjectName

NTSTATUS NTOSKRNL ObReferenceObjectByName(
        PUNICODE_NAME Name,
        DWORD Attributes, // 经常使用 OBJ_CASE...
            PACCESS_STATE AccessState OPTIONAL,
        ACCESS_MASK Access,
        POBJECT_TYPE Type,
        KPROCESSOR_MODE bMode,
        DWORD Unknown OPTIONAL, // 与其说是保留,
            // 不如说是只用作 SecurityProcedure 的参数
            // 总是为 0
        PVOID BodyPointer
        );

// ObpLookupObjectName/ ObpCreateHandle 的主要逻辑

NTSTATUS ObOpenObjectByName(
        POBJECT_ATTRIBUTES Attributes,
        POBJECT_TYPE Type,
        KPROCESSOR_MODE bMode,
        PACCESS_STATE AccessState OPTIONAL,
        ACCESS_MASK Access,
        DWORD Unknown OPTIONAL, // 见 ObReferenceObjectByName 中的 unknown
        PHANDLE Handle OUT
        );

// ObReferenceObjectByPointer / ObpCreateHandle 的主要逻辑

NTSTATUS NTOSKRNL ObOpenObjectByPointer(
        PVOID Object,
        DWORD Attributes,
```

```
                PACCESS_STATE AccessState OPTIONAL,
                ACCESS_MASK Access,
                POBJECT_TYPE Type,
                KPROCESSOR_MODE bMode,
                PHANDLE Handle OUT
                );

// 函数代码 :
// mov eax,[esp+pObject]
// mov eax,[eax-18h]
// retn 4
// 完了!

DWORD NTOSKRNL ObGetObjectPointerCount(PVOID Object);

// 有趣的函数. 一般不用在内核中同时也未公开.
// 对于域 Object, Type 和 HandleAttributes 其值可以为 0,
// 此时将查找任意符合的值
// 此函数可以查找关于句柄的信息

BOOLEAN NTOSKRNL ObFindHandleForObject(
                PPROCESS Process,
                PVOID Object,
                POBJECT_TYPE Type,
                DWORD HandleAttributes, //低 3 位. 见 '句柄表'                PHANDLE Handle O
UT          // 返回句柄
                );


// 此函数清除对象首部中的相应标志
// 如果不再引用该对象，则允许将其删除，并调用内部函数 ObpDeleteNameCheck

NTSTATUS NTOSKRNL ObMakeTemporaryObject(
                PVOID Object
                );

// ObGetObjectSecurity/SeAccessCheck 的主要逻辑

BOOLEAN NTOSKRNL ObCheckObjectAccess(
                PVOID Object,
                PACCESS_STATE AccessState,
                BOOLEAN bTypeLocked,
                KPROCESSOR_MODE Mode,
                OUT PNTSTATUS AccessStatus
```

```
);

// 当在树中创建对象时，在 ObpLookupObjectName 中被调用

BOOLEAN NTOSKRNL ObCheckCreateObjectAccess(
        PDIRECTORY Directory,
        ACCESS_MASK Access,
        PACCESS_STATE AccessState,
        PUNICODE NameFromDir,
        DWORD UnknLockedFlags,
        KPROCESSOR_MODE Mode,
        OUT PNTSTATUS AccessStatus
        );

// п о с л е д о в а т е л ь н о с т ь   ExMapHandleToPointer/取出位 0x4 (见'句柄数据库')

BOOLEAN NTOSKRNL ObQueryObjectAuditingByHandle(
        HANDLE Handle,
        PDWORD ReturnValue);

// 此函数在方法 SpDefaultObjectMetod 中调用，用于代号 0 - set security descriptor.
// 该函数的工作是基于 SeSetSecurityDesriptorInfo 函数的。现在我们不详细研究安全描述符，除此之外，某些参数还不明了，
// 故此函数的原型我就不给出了
// ObSetSecurityDescriptorInfo(...);
```

Не останавливаться на бу
д у щ е м -
Я понял, что его не бу
д е т
И ты знаешь, когда я уйду
-
Ты услышишь мой плач в ветре...
(c) by Anathema

0c.结语
==============

实际上，对象管理器的大多数导出函数都不是完全 functional 的，这点现在是完全显然的了。一般来说，如果提出关于可行性的问题，可以表达这样的见解。Kernel 驱动并不使用

句柄。一般说来，在普通情况下并不知道位于驱动代码在哪个进程的上下文中执行。实际使用的函数允许用名字或指针来引用对象。

对于剩下的函数，如果我们没写过系统服务的话，是很难想到其真实的用法的（但是完全可能的）。对于实现 Windows NT 的潜力，管理器运作的知识总是很有益的。我们怎样确信处理对象的系统是十分灵活的——要知道 NT 的执行系统要使用它来模拟不同的操作系统。当然，本文可能还有不准确的地方。任何评论意见请发至 peter_k@vivos.ru (Peter Kosyh)。

Best regards, Gloomy

附录

13.从用户模式下获取对象信息
==

在 Windows NT 中有许多系统调用以 Query 或 QueryInformation 命名。这些函数都能获得有趣的信息，而且大多是未公开的。下面给出 NtQueryObject 函数的原型，使用该函数能轻松获取关于对象格式的信息。

```
typedef enum _OBJECTINFOCLASS {
    BaseObjectInfo,
    NameObjectInfo,
    TypeObjectInfo,
    AllTypesObjectInfo,
    HandleObjectInfo
  } OBJECTINFOCLASS;

NTSYSAPI NTSTATUS NTAPI NtQueryObject(
        HANDLE ObjHandle,
        OBJECTINFOCLASS ObjectInfoClass,
        OUT PVOID ObjectInfo, // 信息缓冲区
        DWORD ObjectInfoLen,    // 缓冲区长度
        OUT PDWORD RetInfoLen // 返回信息的长度
        );

typedef struct _BASE_OBJECT_INFO{
    DWORD HandleAttributes,
    ACCESS_MASK GrantedAccess,
    DWORD HandleCount,
    DWORD RefCount,
    DWORD PagedPollQuota,
    DWORD NonPagedPollQuota,
    DWORD ReservedAndAlwaysNull[3],
    DWORD NameObjectInfoLength,
```

```
    DWORD TypeObjectInfoLength,
    DWORD SecurityDescriptorLengh,
    LARGE_INTEGER SymLinkCreationTime //自 1601 年起的时间
    };

typedef struct _NAME_OBJECT_INFO{
    UNICODE_STRING Name;
    // 这里应当是名字的位置. 参数 ObjectInfo 可能为 0
      // 以获取缓冲区的大小
    }NAME_OBJECT_INFO;

typedef struct _TYPE_OBJECT_INFO{
    UNICODE_STRING Name;
    DWORD InstanceCount;
    DWORD HandleCount;
    DWORD PeakObjects;
    DWORD PeakHandles;
    DWORD AllowedAttributesMask;
    GENERIC_MAPPING GenericMapping;
    DWORD AllowedAccessMask;
    BOOLEAN bInNameSpace;
    BOOLEAN bHandleDBInfo;
    BOOLEAN Align[2];
    DWORD Unknown6;              // 见对象类型的 unknown6 域
    DWORD DefaultPagedPollQuota;
    DWORD DefaultNonPagedPollQuota;
    }TYPE_OBJECT_INFO;

typedef struct _ALL_TYPES_OBJECT_INFO{
    DWORD NumOfTypes;
    TYPE_OBJECT_INFO [ANY_SIZE_ARRAY];
    }ALL_TYPES_OBJECT_INFO;

typedef struct _HANDLE_OBJECT_INFO{
    BOOLEAN Inherit;
    BOOLEAN ProtectFromClose;
    }HANDLE_OBJECT_INFO;
```

14.某些与对象管理相关的系统服务
==

在Н е д о к у м е н т и р о в а н н ы е в о з м о ж н о с т и Windows NT

一书中А .В . К о б е р н и ч е н к о 给出了某些系统服务接口的原型。作者的目的是描述未公开系统调用的原型，于是从 kernel32.dll 中使用的系统调用出发。大概对于这个目的，这是个捷径（恰巧，这本书也试图描述 NtQueryObject 函数，但是其描述在这里是不适用的，而且书中的函数的描述很不完整）。对于某些函数的反汇编，我检查了它的输出并确信定义的正确性。下面我给出某些调用，因为作者对它们进行了描述。

```
//用于任何对象的函数

NTSYSAPI NTSTATUS NTAPI
    NtClose(IN HANDLE Handle);

NTSYSAPI NTSTATUS NTAPI NtMakeTemporaryObject(
        IN HANDLE Handle
        );

#define DUPLICATE_CLOSE_SOURCE       0x00000001
#define DUPLICATE_SAME_ACCESS        0x00000002

NTSYSAPI NTSTATUS NTAPI
NtDuplicateObject(
    IN HANDLE SourceProcessHandle,
    IN HANDLE SourceHandle,
    IN HANDLE TargetProcessHandle,
    OUT PHANDLE TargetHandle OPTIONAL,
    IN ACCESS_MASK DesiredAccess,
    IN ULONG Attributes,//OBJ_xxx
    IN ULONG Options
    );

//对象目录

#define DIRECTORY_QUERY                 (0x0001)
#define DIRECTORY_TRAVERSE              (0x0002)
#define DIRECTORY_CREATE_OBJECT         (0x0004)
#define DIRECTORY_CREATE_SUBDIRECTORY   (0x0008)

#define DIRECTORY_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | 0xF)

NTSYSAPI NTSTATUS NTAPI
NtCreateDirectoryObject(
    OUT PHANDLE DirectoryHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
    );
```

```
NTSYSAPI NTSTATUS NTAPI
NtOpenDirectoryObject(
        OUT PHANDLE DirectoryHandle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes
        );

typedef struct _OBJECT_NAMETYPE_INFO {
          UNICODE_STRING ObjectName;
          UNICODE_STRING ObjectType;
} OBJECT_NAMETYPE_INFO, *POBJECT_NAMETYPE_INFO;

typedef enum _DIRECTORYINFOCLASS {
          ObjectArray,
          ObjectByOne
} DIRECTORYINFOCLASS, *PDIRECTORYINFOCLASS;


NTSYSAPI NTSTATUS NTAPI
NtQueryDirectoryObject(
          IN HANDLE DirectoryObjectHandle,
          OUT PVOID ObjectInfoBuffer,
          IN ULONG ObjectInfoBufferLength,
          IN DIRECTORYINFOCLASS DirectoryInformationClass,
          IN BOOLEAN First,
          IN OUT PULONG ObjectIndex,
          OUT PULONG LengthReturned
          );

//对象符号链接

#define SYMBOLIC_LINK_QUERY (0x0001)

#define SYMBOLIC_LINK_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | 0x1)

NTSYSAPI NTSTATUS NTAPI
NtCreateSymbolicLinkObject(
        OUT PHANDLE ObjectHandle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes,
          IN PUNICODE_STRING SubstituteString
        );
```

```
NTSYSAPI NTSTATUS NTAPI
NtOpenSymbolicLinkObject(
        OUT PHANDLE ObjectHandle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes
        );

typedef struct _OBJECT_NAME_INFORMATION {
        UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;

NTSYSAPI NTSTATUS NTAPI
NtQuerySymbolicLinkObject(
          IN HANDLE ObjectHandle,
          OUT POBJECT_NAME_INFORMATION SubstituteString,
          OUT PULONG SubstituteStringLength //字节
          );
```

我想，现在领悟描述过的大部分管理器系统调用行为应该不难了……

在网上可以找到 "Недокументированные возможномти Windows NT"一书的例子——作者做了一件大好事。在源代码中描述了大部分未公开的系统调用以及给出的例子的用法。

(c)Gloomy aka Peter Kosyh, Melancholy Coding'2001

http://gloomy.cjb.net
mailto:gl00my@mail.ru

董岩 译
http://greatdong.blog.edu.cn