漫谈兼容内核之二十一：
Windows 进程的用户空间

毛德操

进程的用户空间是应用软件活动的舞台，所以搞清楚 Windows 进程用户空间的格局和轮廓对于深入理解 Windows 进程的运行有着重要的意义。而对于兼容内核的开发者，则这个问题更是非搞清楚不可，因为开发兼容内核的意图就是要让 Windows 应用软件得以在 Linux 系统上运行。其实，在前面的一些漫谈中，随着当时话题的开展也曾讲到过有关这个问题的一些大概，但是一方面只是散见于各处，不够集中；另一方面也不够系统和完整，实际上也因而不够详细和确切；因此有必要专门把它作为一个话题加以比较系统、完整的深入介绍。有道是“耳闻为虚，眼见为实”，对于我们来说，所谓“眼见”就是要见到有关的源代码。当然，我们所能见到的只是 RreactOS 的源代码，好在各种迹象表明 RreactOS 与 Windows 在这方面也是相当贴近和一致的。

我们先从几个宏定义着手：

```
#define MM_HIGHEST_USER_ADDRESS          *MmHighestUserAddress
#define MM_USER_PROBE_ADDRESS            *MmUserProbeAddress
#define MM_LOWEST_USER_ADDRESS           (PVOID)0x10000

#define KI_USER_SHARED_DATA              0xffdf0000
#define SharedUserData                   \
                ((KUSER_SHARED_DATA   *CONST) KI_USER_SHARED_DATA)
```

在 ReactOS 的代码中，每当需要引用用户空间的上限、即最高地址时，一般就把 MM_HIGHEST_USER_ADDRESS 用作常数，但实际上这是个宏操作，是在引用指针 MmHighestUserAddress 的内容。与此相似的还有 MM_USER_PROBE_ADDRESS，实际上是在引用指针 MmUserProbeAddress 的内容。这两个指针的内容是在内核初始化时设置好的：

```
MmInit1(…)
{
    ……
    MmUserProbeAddress = 0x7fff0000;
    MmHighestUserAddress = (PVOID)0x7ffeffff;
    ……
}
```

就是说，应用软件在用户空间可以访问的最高地址(虚拟地址)是 0x7ffeffff，从 0x7fff0000 开始就不能访问了。一般文献中讲 Windows 系统中用户空间与系统空间的分界线是 0x80000000，而这里之所以是 0x7fff0000 而不是 0x80000000，是因为在分界下面留了 64KB 的空间不让访问，以此作为隔离区。

应用软件在用户空间可以访问的最低地址是 MM_LOWEST_USER_ADDRESS，这是个常数，定义为 0x10000，就是第一个 64KB 边界的地方，而从 0 开始的 64KB 也是不让访问

的。

还有个常数 KI_USER_SHARED_DATA 更是值得一说。这个常数定义为 0xffdf0000，是一个区间的起点。这个区间按说是在系统空间，却又划出来让用户空间的程序访问，就好像是用户空间的一小片飞地。这片飞地的目的是用来让用户空间的程序访问内核中的一些数据，好像是在系统空间的地皮上挖了口井。而且，这个区间是由系统空间和所有用户空间共享，也就是为所有进程所共享，所以这个常数名为 KI_USER_SHARED_DATA。那么这个“井”里有些什么数据呢？上面的宏定义为此定义了一个数据结构 KUSER_SHARED_DATA，并定义了一个相应的结构指针 SharedUserData，让它指向这个地址：

```
typedef struct _KUSER_SHARED_DATA {
    ULONG TickCountLowDeprecated;
    ULONG TickCountMultiplier;
    volatile KSYSTEM_TIME InterruptTime;
    volatile KSYSTEM_TIME SystemTime;
    volatile KSYSTEM_TIME TimeZoneBias;
    USHORT ImageNumberLow;
    USHORT ImageNumberHigh;
    WCHAR NtSystemRoot[260];
    ULONG MaxStackTraceDepth;
    ULONG CryptoExponent;
    ULONG TimeZoneId;
    ULONG LargePageMinimum;
    ULONG Reserved2[7];
    NT_PRODUCT_TYPE NtProductType;
    BOOLEAN ProductTypeIsValid;
    ULONG NtMajorVersion;
    ULONG NtMinorVersion;
    BOOLEAN ProcessorFeatures[PROCESSOR_FEATURE_MAX];
    ULONG Reserved1;
    ULONG Reserved3;
    volatile ULONG TimeSlip;
    ALTERNATIVE_ARCHITECTURE_TYPE AlternativeArchitecture;
    LARGE_INTEGER SystemExpirationDate;
    ULONG SuiteMask;
    BOOLEAN KdDebuggerEnabled;
    volatile ULONG ActiveConsoleId;
    volatile ULONG DismountCount;
    ULONG ComPlusPackage;
    ULONG LastSystemRITEventTickCount;
    ULONG NumberOfPhysicalPages;
    BOOLEAN SafeBootMode;
    ULONG TraceLogging;
    ULONGLONG Fill0;
    UCHAR SystemCall[16];
```

```
    union {
        volatile KSYSTEM_TIME TickCount;
        volatile ULONG64 TickCountQuad;
    };
} KUSER_SHARED_DATA, *PKUSER_SHARED_DATA;
```

这样，地址 0xffdf0000 就是 KUSER_SHARED_DATA 数据结构的起点，而 SharedUserData 就指向这个数据结构。而用户空间的程序则可以通过指针 SharedUserData 读取这个结构中各个成分的内容。下面我们看几个通过使用这个指针从内核获取某些信息的例子。第一个例子是 ntdll.dll 中的一个函数 LdrpMapDllImageFile()：

[LdrLoadDll() > LdrpLoadModule() > LdrpMapDllImageFile()]

```
LdrpMapDllImageFile (…)
{
  . . . . . .

  . . . . . .
  if (SearchPath == NULL)
  {
      /* get application running path */
      . . . . . .
      wcscat (SearchPathBuffer, L";");
      wcscat (SearchPathBuffer, SharedUserData->NtSystemRoot);
      wcscat (SearchPathBuffer, L"\\system32;");
      . . . . . .
      SearchPath = SearchPathBuffer;
  }
  . . . . . .
}
```

“宽字符”数组 NtSystemRoot[]是 KUSER_SHARED_DATA 数据结构中的一个成分，其内容是系统根目录的路径名。

再看第二个例子，这是 kernel32.dll 导出的一个函数：

```
DWORD STDCALL GetTickCount(VOID)
{
  return (DWORD)((ULONGLONG)SharedUserData->TickCountLowDeprecated *
                            SharedUserData->TickCountMultiplier / 16777216);
}
```

GetTickCount()是 W32 API 界面上的一个函数，用户空间的程序可以通过这个函数获取内核中的 Tick 计数，类似于 Linux 内核中的 jiffies 计数。

如果应用软件的代码都不直接引用 SharedUserData，而一律都通过 ntdll.dll 或 kernel32.dll

导出的函数间接地访问这个数据结构，那么这两个 DLL 大体上可以把对于这个数据结构的访问嫁接到 Linux 内核上，比方说通过相关的系统调用、或/proc 机制等等，那都还是可行的。例如，Wine 对 GetTickCount()的实现是这样的：

```
DWORD WINAPI GetTickCount(void)
{
    struct timeval t;
    gettimeofday( &t, NULL );
    return ((t.tv_sec * 1000) + (t.tv_usec / 1000)) - server_startticks;
}
```

显然，这是依靠 Linux 系统调用 gettimeofday()实现了对访问 SharedUserData 中有关数据的模拟。对其它成分的访问也有望通过类似的手法实现。

但是，既然从用户空间可以通过基地址 0xffdf0000 访问这个数据结构，而且已经为人所知，就保不住没有应用软件会不守规矩，放着好好的阳关道不走偏要去走独木桥，直接就通过这个地址来达到目的。这在早期的 Windows 软件中似乎更有可能，因为当初 DOS 软件中就常常会通过一些地址约定来获取某些特定的信息。因此，要达到与 Windows 软件的高度兼容，就得考虑在 Linux 系统空间的相同位置上也挖出这么一口“井”来。

读者在上面看到的还只是对用户空间格局的安排，而并未实际将其实现，还算不上是眼见为实；再说也过于粗线条，只是划定了一个边界而已。下面我们就来看实际的实现和有关的细节。既然用户空间是进程的用户空间，那么用户空间的创建自然就与进程的创建连系在一起。事实上，用户空间的创建及其格局的实现有很多都是在函数 PspCreateProcess()内部实现的。我们以前也看过这个函数的代码，只是当时的侧重面不同，现在我们就把目光集中在与用户空间有关的存储管理上。

[NtCreateProcess() > PspCreateProcess()]

```
PspCreateProcess(OUT PHANDLE ProcessHandle, …)
{
    PEPROCESS Process;.
     . . . . .

    . . . . . .
    MmInitializeAddressSpace(Process, &Process->AddressSpace);
    ObCreateHandleTable(pParentProcess, InheritObjectTable, Process);
    MmCopyMmInfo(pParentProcess ? pParentProcess : PsInitialSystemProcess, Process);
    . . . . . .
    /* Now we have created the process proper */
    MmLockAddressSpace(&Process->AddressSpace);

    /* Protect the highest 64KB of the process address space */
    BaseAddress = (PVOID)MmUserProbeAddress;
    Status = MmCreateMemoryArea(Process, &Process->AddressSpace,
```

```
                    MEMORY_AREA_NO_ACCESS,
                    &BaseAddress, 0x10000, PAGE_NOACCESS,
                    &MemoryArea, FALSE, FALSE, BoundaryAddressMultiple);
    . . . . . .

    /* Protect the lowest 64KB of the process address space */
#if 0
    BaseAddress = (PVOID)0x00000000;
    Status = MmCreateMemoryArea(Process, &Process->AddressSpace,
                    MEMORY_AREA_NO_ACCESS,
                    &BaseAddress, 0x10000, PAGE_NOACCESS,
                    &MemoryArea, FALSE, FALSE, BoundaryAddressMultiple);
    . . . . . .
#endif
```

每个进程都有个用户空间。进程控制块 EPROCESS 数据结构中有个成分 AddressSpace，就是代表着用户空间的 MADDRESS_SPACE 数据结构。所以在创建进程时要通过 MmInitializeAddressSpace()对新建进程的地址空间 AddressSpace 进行初始化，特别是将此数据结构中的成分 LowestAddress 设置成 MM_LOWEST_USER_ADDRESS 即 0x10000。然后通过 MmCopyMmInfo()从创建者进程或系统进程“System”复制其系统空间的页面目录。这里的指针 PsInitialSystemProcess 指向系统进程“System”的 EPROCESS 数据结构。不过，这里复制的只是系统空间的页面目录，而与用户空间无关。

下面就开始从新建的用户空间具体地配置虚拟地址区间了。首先是从 MmUserProbeAddress 所指向的 0x7fff0000 开始往上，长度为 0x10000、即 64KB 的区间，这个区间的性质是禁止访问。这里 MEMORY_AREA_NO_ACCESS 表示区间的性质，将被记入该进程 AddressSpace 中有关的数据结构中，而 PAGE_NOACCESS 则将被记入页面映射目录相应的表项中。本来还把下面从地址 0 开始向上的 64KB 也设置成禁止访问，后来用编译条件“#if 0”将其删去了，原因大概是既然已经把用户空间的地址下限设定为 MM_LOWEST_USER_ADDRESS，因而不会把这下面的区间分配出去，反正没有映射，也就没有必要多此一举了。

这就好像是房地产商“批地”的过程。至此，从 MM_LOWEST_USER_ADDRESS 即 0x10000 开始，到 0x7fff0000 为止的用户空间“地皮”就形成了。不过现在的整块地皮都是空的，只要未经分配和映射，就都不能访问。

读者可能注意到了，上面讲到把用户空间的下限设置成 0x10000，却没有设置上限。这是因为用户空间还要有一块“飞地”在 USER_SHARED_DATA 即 0xffdf0000 的地方，那就差不多已是整个 CPU 寻址范围的尽头了。我们继续往下看：

[NtCreateProcess() > PspCreateProcess()]

```
    /* Protect the 60KB above the shared user page */
    BaseAddress = (char*)USER_SHARED_DATA + PAGE_SIZE;
    Status = MmCreateMemoryArea(Process, &Process->AddressSpace,
                    MEMORY_AREA_NO_ACCESS,
                    &BaseAddress, 0x10000 - PAGE_SIZE, PAGE_NOACCESS,
```

```
                    &MemoryArea, FALSE, FALSE, BoundaryAddressMultiple);
    . . . . . .

    /* Create the shared data page */
    BaseAddress = (PVOID)USER_SHARED_DATA;
    Status = MmCreateMemoryArea(Process, &Process->AddressSpace,
                    MEMORY_AREA_SHARED_DATA,
                    &BaseAddress, PAGE_SIZE, PAGE_READONLY,
                    &MemoryArea, FALSE, FALSE, BoundaryAddressMultiple);
    MmUnlockAddressSpace(&Process->AddressSpace);
    . . . . . .
```

这是在建立前面所讲的用户共享区，就像是在系统空间地皮上的“飞地”。先看后面那个操作，这是把从 USER_SHARED_DATA 即 0xffdf0000 开始的一个页面分配给用户空间，让用户空间的程序可以对此页面作“只读”访问。再看前面那个操作，这是把从 0xffdf0000 开始的 64KB 除已分配的最低那个页面之外全都设置成禁止访问。

至此，用户空间的本土和飞地都已圈好，下面就开始分配和使用其本土的地皮了。

[NtCreateProcess() > PspCreateProcess()]

```
    /* FIXME - Map ntdll */
    Status = LdrpMapSystemDll(hProcess, &LdrStartupAddr);
    . . . . . .
    /* Map the process image */
    if (SectionObject != NULL)
    {
        . . . . . .
        Status = MmMapViewOfSection(SectionObject, Process, (PVOID*)&ImageBase, 0,
                    ViewSize, NULL, &ViewSize, 0, MEM_COMMIT, PAGE_READWRITE);
        ObDereferenceObject(SectionObject);
        . . . . . .
    }
    . . . . . .
    Status = PsCreatePeb(hProcess, Process, ImageBase);
    . . . . . .
    return Status;
}
```

首先是 LdrpMapSystemDll()，读者想必已经知道，这是把 ntdll.dll 的映像“装入”、即映射到用户空间。其位置和大小取决于映像文件头部的有关信息，事实上 ntdll.dll(版本之一)的装入地址是 0x77f50000，大小为 0x0000A700。

接着是目标 EXE 映像的映射。在调用 NtCreateProcess()之前，调用者已经为目标映像创建了一个 Section、即共享内存区，现在要做的是通过 MmMapViewOfSection()把这个共享区

映射到新建进程的用户空间。映射的具体位置和大小基本上也取决于映像文件头部的有关信息，但可以有一定的弹性，不像 ntdll.dll 和其它几个系统 DLL 那么刚性。以 notepad.exe 为例，其文件头部提供的装入地址为 0x01000000。至于映像大小就更是各不相同了。

注意这里装入的都只是静态的映像。目标映像在实际运行时所消耗的内存可能远不止于此，不过那是以后动态分配内存的事了。

最后是 PsCreatePeb()，显然这是要在用户空间建立“进程环境块”PEB。

[NtCreateProcess() > PspCreateProcess() > PsCreatePeb()]

```
static NTSTATUS
PsCreatePeb(HANDLE ProcessHandle, PEPROCESS Process, PVOID ImageBase)
{
  . . . . . .

  /* Allocate the Process Environment Block (PEB) */
  Process->TebBlock =
     (PVOID) MM_ROUND_DOWN(PEB_BASE, MM_VIRTMEM_GRANULARITY);
  AllocSize = MM_VIRTMEM_GRANULARITY;
  Status = NtAllocateVirtualMemory(ProcessHandle, &Process->TebBlock, 0,
                                   &AllocSize, MEM_RESERVE, PAGE_READWRITE);
  . . . . . .
  Peb = (PPEB)PEB_BASE;
  PebSize = PAGE_SIZE;
  Status = NtAllocateVirtualMemory(ProcessHandle, (PVOID*)&Peb, 0,
                                   &PebSize, MEM_COMMIT, PAGE_READWRITE);
  . . . . . .
  Process->TebLastAllocated = (PVOID) Peb;

  ViewSize = 0;
  SectionOffset.QuadPart = (ULONGLONG)0;
  TableBase = NULL;
  Status = MmMapViewOfSection(NlsSectionObject, Process, &TableBase, 0, 0,
                                      &SectionOffset, &ViewSize,
                                      ViewShare, MEM_TOP_DOWN, PAGE_READONLY);
  . . . . . .
  KeAttachProcess(&Process->Pcb);

  /* Initialize the PEB */
  RtlZeroMemory(Peb, sizeof(PEB));
  Peb->ImageBaseAddress = ImageBase;
  Peb->OSMajorVersion = 4;
  Peb->OSMinorVersion = 0;
  Peb->OSBuildNumber = 1381;
  Peb->OSPlatformId = 2; //VER_PLATFORM_WIN32_NT;
```

```
    Peb->OSCSDVersion = 6 << 8;
    Peb->AnsiCodePageData       = (char*)TableBase + NlsAnsiTableOffset;
    Peb->OemCodePageData        = (char*)TableBase + NlsOemTableOffset;
    Peb->UnicodeCaseTableData = (char*)TableBase + NlsUnicodeTableOffset;
    Process->Peb = Peb;

    KeDetachProcess();
    return(STATUS_SUCCESS);
}
```

这里涉及的两个常数定义为：

```
#define PEB_BASE                                (0x7FFDF000)
#define MM_VIRTMEM_GRANULARITY      (64 * 1024)
```

就是说，PEB 在用户空间的位置是 0x7FFDF000。而内存管理机制在保留和分配用户空间地址区间时的“粒度”是 64KB。就是说，至少是 64KB，并且以 64KB 为一个单元。而 MM_ROUND_DOWN(PEB_BASE, MM_VIRTMEM_GRANULARITY)是计算 PEB 所在的那个 64KB 区间的下部边界：

```
#define MM_ROUND_DOWN(x,s) ((PVOID)(((ULONG_PTR)(x)) & ~((ULONG_PTR)(s)-1)))
```

计算一下就可以知道结果是 0x7FFD0000。注意页面的大小 PAGE_SIZE 是 0x1000 即 4KB，所以 PEB_BASE 与页面边界是对齐的，只是没有与 64KB 边界对齐。

代码中调用了两次 NtAllocateVirtualMemory()，但是注意第一次是要求“保留(MEM_RESERVE)”从 0x7FFD0000 开始的 64KB 区间，而第二次则是要求仅仅“交割(MEM_COMMIT)”PEB 所实际占用的区间。

下面还有一次 MmMapViewOfSection()，这是为了把与 NLS 即“本国语言支持”有关的数据和代码映射到用户空间。像别的可执行映像一样，也要先为其创建一个 Section，然后把这个 Section 映射到需要使用它的用户空间。NLS 的 Section 对象 NlsSectionObject 是在系统初始化的时侯创建的。

注意经过 PsCreatePeb()的处理以后 EPROCESS 结构中的指针 TebBlock 指向 0x7FFD0000，指针 Peb 则指向 PEB 的起点 0x7FFDF000，而 TebLastAllocated 此刻同样指向 PEB 的起点，但是后面我们将看到这个指针是随着 TEB 的分配和建立而变的。事实上，“线程环境块”TEB 都在 PEB 的下方，并且通常不止一个。而上面所保留的 64KB 区间、即 16 个页面，则除顶端的一个页面用于 PEB 外其余 15 个页面都是为 TEB 准备的。

这里没有涉及堆栈，这是因为进程本身并不受调度运行，进程里面的线程才受调度运行，所以堆栈是与线程相连系、而不是与进程相连系的。

在新建进程的用户空间为其建立了 PEB 之后，创建者还要通过 KlInitPeb()对此 PEB 加以进一步的初始化，在此之前创建者已经准备好了一个“进程参数块”Ppb 作为调用参数之一传下来。进程参数块是一个 RTL_USER_PROCESS_PARAMETERS 数据结构，里面带下来的参数中有一个指针 Environment，指向一个“宽字符”串，这就是各环境变量的定义。注意 KlInitPeb()是在创建者进程的用户空间执行的，但是其操作目的和对象则在于新建进程的用户空间。

[CreateProcessW() > KlInitPeb()]

```
static NTSTATUS KlInitPeb(HANDLE ProcessHandle,
                          PRTL_USER_PROCESS_PARAMETERS Ppb,
                          PVOID *ImageBaseAddress, ULONG ImageSubSystem)
{
    . . . . . .
    PVOID EnvPtr = NULL;
    . . . . . .

    /* create the Environment */
    if (Ppb->Environment != NULL)
    {
        ParentEnv = Ppb->Environment;
        ptr = ParentEnv;
        while (*ptr)
        {
           while(*ptr++);
        }
        ptr++;
        EnvSize = (PVOID)ptr - ParentEnv;
    }
    else if (NtCurrentPeb()->ProcessParameters->Environment != NULL)
    {
        MEMORY_BASIC_INFORMATION MemInfo;
        ParentEnv = NtCurrentPeb()->ProcessParameters->Environment;

        Status = NtQueryVirtualMemory (NtCurrentProcess (), ParentEnv,
                                       MemoryBasicInformation, &MemInfo,
                                       sizeof(MEMORY_BASIC_INFORMATION), NULL);
        EnvSize = MemInfo.RegionSize;
    }
    DPRINT("EnvironmentSize %ld\n", EnvSize);

    /* allocate and initialize new environment block */
    if (EnvSize != 0)
    {
        EnvSize1 = EnvSize;
        Status = NtAllocateVirtualMemory(ProcessHandle, &EnvPtr, 0, &EnvSize1,
                         MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
        . . . . . .
        NtWriteVirtualMemory(ProcessHandle, EnvPtr, ParentEnv, EnvSize, &BytesWritten);
    }
```

```
    /* create the PPB */
    PpbBase = NULL;
    PpbSize = Ppb->AllocationSize;
    Status = NtAllocateVirtualMemory(ProcessHandle, &PpbBase, 0, &PpbSize,
                                     MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    . . . . . .
    //DPRINT("Ppb->MaximumLength %x\n", Ppb->MaximumLength);
    NtWriteVirtualMemory(ProcessHandle, PpbBase, Ppb,
                                                Ppb->AllocationSize, &BytesWritten);

    /* write pointer to environment */
    Offset = FIELD_OFFSET(RTL_USER_PROCESS_PARAMETERS, Environment);
    NtWriteVirtualMemory(ProcessHandle, (PVOID)(PpbBase + Offset),
                                              &EnvPtr, sizeof(EnvPtr), &BytesWritten);
    /* write pointer to process parameter block */
    Offset = FIELD_OFFSET(PEB, ProcessParameters);
    NtWriteVirtualMemory(ProcessHandle, (PVOID)(PEB_BASE + Offset),
                                            &PpbBase, sizeof(PpbBase), &BytesWritten);
    /* Write image subsystem */
    Offset = FIELD_OFFSET(PEB, ImageSubSystem);
    NtWriteVirtualMemory(ProcessHandle, (PVOID)(PEB_BASE + Offset),
                                 &ImageSubSystem, sizeof(ImageSubSystem), &BytesWritten);
    /* Read image base address. */
    Offset = FIELD_OFFSET(PEB, ImageBaseAddress);
    NtReadVirtualMemory(ProcessHandle, (PVOID)(PEB_BASE + Offset),
                                 ImageBaseAddress, sizeof(PVOID), &BytesWritten);
    return(STATUS_SUCCESS);
}
```

参数 Ppb 实际涉及两块数据。一块是进程参数块本身，这是有固定大小的；另一块的内容是一些环境变量字符串，这些字符串游离在进程参数块外面，并且也没有固定的长度。这里的目的就是要把这两块数据都复制到新建进程的用户空间去，并相应地设置好相关数据结构中的指针。为此，程序中首先要确定这些环境变量字符串所占的长度 EnvSize。

知道了这些环境变量字符串所占的长度 EnvSize 以后，如果非 0，就要在新建进程的用户空间分配相应的区间。注意这里在调用 NtAllocateVirtualMemory()时的第 2 个参数、即指针 EnvPtr 的值已预先设置为 NULL，表示对起始地址没有特定的要求；并且第 3 个参数、即要求在所分配起始地址中前导 0 的个数也是 0，因此可以分配在任意的部位(前导 0 的个数实际上大致上给定了一个部位)。对于这样的分配要求，内存管理会在目标空间从低到高扫描，以找到第一个符合大小要求的区间。由于用户空间的起点是 0x10000，这又是第一次要求由内存管理自由分配，所以实际分配的位置一定在 0x10000 处。另一方面，由于区间分配的粒度是 64KB，所以实际分配的一般总是 64KB，因为很难设想 EnvSize 会大于 64KB。实际分配的位置和大小则通过参数 EnvPtr 和 EnvSize1 返回。然后通过 NtWriteVirtualMemory()将来自参数 Ppb、或当前进程的环境变量字符串复制到新建进程用户空间的这个区间中。

接着就是针对进程参数块 PPB 的本身来故伎重演了。注意这里参数 PpbBase 的值也是预先设置成 NULL，并且调用 NtAllocateVirtualMemory()时的第 3 个参数也是 0，所以也是由内存管理自由分配，而实际分配的位置则总是在环境变量块的上方，一般应该是从 0x20000 到 0x2ffff 的 64KB，实际的位置和大小则通过参数 PpbBase 和 PpbSize 返回。然后就是把进程参数块复制过去。

此后还有 3 次对 NtWriteVirtualMemory()的调用，旨在对新建进程用户空间 PPB 和 PEB 中的几个成分作出修正。第一次是修改 PPB 中的指针 Environment，使其指向实际的环境变量字符串(实际上总是在 0x10000 处)。第二次是设置 PEB 中的指针 ProcessParameters，使其指向实际的进程参数块。第 3 次则是把作为参数传下来的 ImageSubSystem 写入新建进程 PEB 中的同名字段，这实际上是个作为编码的 32 位无符号整数，可能的取值有 IMAGE_SUBSYSTEM_WINDOWS_GUI 和 IMAGE_SUBSYSTEM_WINDOWS_CUI 等等，表示新建进程的可执行映像是视窗应用或控制台应用，具体的数值来自目标映像文件的头部。

最后的 NtReadVirtualMemory()从 PEB 读取其 ImageBaseAddress 字段，并通过参数 ImageBaseAddress 返回目标映像装入用户空间后的起始地址。

下面就是创建新进程的第一个线程了，这里涉及的是堆栈和 TEB 的建立。我们先看函数 RtlRosCreateUserThread()的代码，这是由 Win32 API 函数 CreateProcessW()辗转调用下来的，也是由创建者进程在其用户空间执行：

```
[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
   > RtlRosCreateUserThread()]

NTSTATUS STDCALL
RtlRosCreateUserThread(IN HANDLE ProcessHandle,
      IN POBJECT_ATTRIBUTES ObjectAttributes, IN BOOLEAN CreateSuspended,
      IN LONG StackZeroBits, IN OUT PULONG StackReserve OPTIONAL,
      IN OUT PULONG StackCommit OPTIONAL, IN PVOID StartAddress,
      OUT PHANDLE ThreadHandle OPTIONAL, OUT PCLIENT_ID ClientId OPTIONAL,
      IN ULONG ParameterCount, IN ULONG_PTR *Parameters)
{
   . . . . . .

   . . . . . .
   /* allocate the stack for the thread */
   nErrCode = RtlRosCreateStack(ProcessHandle, &usUserInitialTeb,
                                        StackZeroBits, StackReserve, StackCommit);
   . . . . . .
   /* initialize the registers and stack for the thread */
   nErrCode = RtlRosInitializeContext(ProcessHandle, &ctxInitialContext,
                                StartAddress, &usUserInitialTeb, ParameterCount, Parameters
 );
   . . . . . .
   /* create the thread object */
```

```
    nErrCode = NtCreateThread(ThreadHandle, THREAD_ALL_ACCESS, ObjectAttributes,
            ProcessHandle, ClientId, &ctxInitialContext, &usUserInitialTeb, CreateSuspended);
    . . . . . .
    return STATUS_SUCCESS;
}
```

这里所调用的三个函数都与用户空间的区间分配有关。先看 RtlRosCreateStack()：

[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
> RtlRosCreateUserThread() > RtlRosCreateStack()]

```
NTSTATUS NTAPI RtlRosCreateStack(IN HANDLE ProcessHandle,
                    OUT PINITIAL_TEB InitialTeb, IN LONG StackZeroBits,
                    IN OUT PULONG StackReserve OPTIONAL,
                    IN OUT PULONG StackCommit OPTIONAL)
{
  /* FIXME: read the defaults from the executable image */
  ULONG_PTR nStackReserve = 0x100000;
  /* FIXME: when we finally have exception handling, make this PAGE_SIZE */
  ULONG_PTR nStackCommit = 0x100000;
  NTSTATUS nErrCode;

  if(StackReserve == NULL)    StackReserve = &nStackReserve;
  else    *StackReserve = ROUNDUP(*StackReserve, PAGE_SIZE);

  if(StackCommit == NULL)    StackCommit = &nStackCommit;
  else    *StackCommit = ROUNDUP(*StackCommit, PAGE_SIZE);

  /* FIXME: no SEH, no guard pages */
  *StackCommit = *StackReserve;

  /* FIXME: this code assumes a stack growing downwards */
  /* fixed stack */
  if(*StackCommit == *StackReserve)
  {
    InitialTeb->StackCommit = NULL;
    InitialTeb->StackCommitMax = NULL;
    InitialTeb->StackReserved = NULL;
    InitialTeb->StackLimit = NULL;
    /* allocate the stack */
    nErrCode = NtAllocateVirtualMemory(ProcessHandle, &(InitialTeb->StackLimit),
                    StackZeroBits, StackReserve, MEM_RESERVE | MEM_COMMIT,
                    PAGE_READWRITE);
```

```
    /* store the highest (first) address of the stack */
    InitialTeb->StackBase = (PUCHAR)(InitialTeb->StackLimit) + *StackReserve;
    *StackCommit = *StackReserve;
  }
  /* expandable stack */
  else
  {
    ULONG_PTR nGuardSize = PAGE_SIZE;
    PVOID pGuardBase;

    InitialTeb->StackBase = NULL;
    InitialTeb->StackLimit =    NULL;
    InitialTeb->StackReserved = NULL;
    /* reserve the stack */
    nErrCode = NtAllocateVirtualMemory(ProcessHandle,
                                       &(InitialTeb->StackReserved), StackZeroBits,
                                       StackReserve, MEM_RESERVE, PAGE_READWRITE);
    /* expandable stack base - the highest address of the stack */
    InitialTeb->StackCommit = (PUCHAR)(InitialTeb->StackReserved) + *StackReserve;
    /* expandable stack limit - the lowest committed address of the stack */
    InitialTeb->StackCommitMax = (PUCHAR)(InitialTeb->StackCommit) - *StackCommit;
    /* commit as much stack as requested */
    nErrCode = NtAllocateVirtualMemory(ProcessHandle, &(InitialTeb->StackCommitMax),
                                       0, StackCommit, MEM_COMMIT, PAGE_READWRITE);
    pGuardBase = (PUCHAR)(InitialTeb->StackCommitMax) - PAGE_SIZE;
    /* set up the guard page */
    nErrCode = NtAllocateVirtualMemory(ProcessHandle, &pGuardBase, 0,
                    &nGuardSize, MEM_COMMIT, PAGE_READWRITE | PAGE_GUARD);
  }
  /* success */
  return STATUS_SUCCESS;
}
```

代码的主体是一个 if 语句，判定的条件是指针 StackCommit 和 StackReserve 所指向的数值是否相等。这两个指针是通过调用参数传下来的，它们所指向的数值决定了堆栈的大小，如果是空指针就采用默认值 0x100000，就是 1MB。如果两个指针所指的值相同就表示要建立大小固定的堆栈；否则就表示要建立可以随时扩充的堆栈，此时需要为限制堆栈的增长设置一个隔离页面。

我们看它 else 部分的代码，因为这相对而言要复杂一点。

先看对 NtAllocateVirtualMemory()的第一次调用，其目的是要保留用于堆栈的区间。注意调用参数 InitialTeb->StackReserved 的值已预先设置为 NULL；StackZeroBits 的值是从上面一路传下来的，实际上也是 0。所以，这个堆栈的位置也是自由分配的。根据前面分配使用用户空间的历史，我们可以得出结论：堆栈的默认位置是 0x30000 至 0x12ffff，共 1MB。实际分配的起始地址和大小则通过 InitialTeb->StackReserved 和参数 StackReserve 返回。

然后，对 NtAllocateVirtualMemory()的第二次调用则要求“交割”目前所要求的大小，即 StackCommit 所指向的数值。注意内存管理分配的空间是由下往上伸展的，而堆栈是由上向下伸展的，因此首先交割的范围是在这区间的上部，所以这里要进行一些计算。注意这里 InitialTeb->StackCommitMax 所指向的倒反而是地址较低的地方。

第三次调用 NtAllocateVirtualMemory()的目的在于在堆栈区间的下部(堆栈的顶部)设置一个保护页面。这里的 nGuardSize 是个局部量，预先设置成 PAGE_SIZE，所以隔离区的大小就是一个页面。当然，这个页面也在前面保留的 1MB 区间之中。

这个过程中所涉及或改变的一些数值都是应该记录在目标线程的 TEB 中的，但是此时目标线程的 TEB 尚未建立，所以上层传下一个指针 InitialTeb，指向一个“初始 TEB”，可以先用起来，以后再把它的内容复制到目标线程的 TEB 中。初始 TEB 的数据结构与实际的 TEB 数据结构略有不同。

回到 RtlRosCreateUserThread()的代码，下一步的操作是 RtlRosInitializeContext()，这是要为新建的线程虚构出一个上下文。

[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
> RtlRosInitializeContext()]

```
NTSTATUS NTAPI
RtlRosInitializeContext(IN HANDLE ProcessHandle, OUT PCONTEXT Context,
                        IN PVOID StartAddress, IN PINITIAL_TEB InitialTeb,
                        IN ULONG ParameterCount, IN ULONG_PTR * Parameters)
{
 static PVOID s_pRetAddr = (PVOID)0xDEADBEEF;
 . . . . . .

 /* Intel x86: linear top-down stack, all parameters passed on the stack */
 /* get the stack base and limit */
 nErrCode = RtlpRosGetStackLimits(InitialTeb, &pStackBase, &pStackLimit);
 /* validate the stack */
 nErrCode = RtlpRosValidateTopDownUserStack(pStackBase, pStackLimit);

 memset(Context, 0, sizeof(CONTEXT));
 /* initialize the context */
 . . . . . .
 Context->Eip = (ULONG_PTR)StartAddress;
 Context->SegGs = USER_DS;
 Context->SegFs = TEB_SELECTOR;
 . . . . . .
 Context->Esp = (ULONG_PTR)pStackBase - (nParamsSize + sizeof(ULONG_PTR));
 Context->EFlags = ((ULONG_PTR)1 << 1) | ((ULONG_PTR)1 << 9);

 /* write the parameters */
 nErrCode = NtWriteVirtualMemory(ProcessHandle, ((PUCHAR)pStackBase) - nParamsSize,
```

```
                                    Parameters, nParamsSize, &nDummy);

 /* write the return address */
 return NtWriteVirtualMemory(ProcessHandle,
                    ((PUCHAR)pStackBase) - (nParamsSize + sizeof(ULONG_PTR)),
                    &s_pRetAddr, sizeof(s_pRetAddr), &nDummy);
}
```

这里涉及对新建进程用户空间的两次写操作。一次是把上面传下来的指针数组 Parameters 复制到用户空间的堆栈上，另一次是把 s_pRetAddr 的值 0xDEADBEEF 也复制到堆栈上，这样就在用户空间的堆栈上虚构出一个函数调用框架。这个框架的返回地址是 0xDEADBEEF，但是实际上不会被用到，只是摆个样子、充个数。而调用参数则是由上层传下来的，下面读者就会看到一共有两个参数，其一是函数指针，其二则是 PEB 所在的地址。

这里指针 pStackBase 的值来自前面的 InitialTeb->StackCommit，注意这个地址比 InitialTeb->StackCommitMax 更高，因为堆栈是由上向下伸展的。

注意这里所说的参数是为新建线程在用户空间的程序入口提供的，不要把它们跟前面所说的进程参数块相混淆。实际上，进程的第一个线程正式进入用户空间以后首先执行的是 RtlBaseProcessStart()，这里所说的参数就是对于这个函数的调用参数。为了搞清这两个参数到底是什么，我们往上回溯到 KlCreateFirstThread()对 RtlRosCreateUserThreadVa()的调用：

```
HANDLE STDCALL KlCreateFirstThread(…)
{
  ……
  nErrCode = RtlRosCreateUserThreadVa(ProcessHandle, &oaThreadAttribs,
             dwCreationFlags & CREATE_SUSPENDED, 0,
             &(Sii->StackReserve), &(Sii->StackCommit),
             pTrueStartAddress, &hThread, &cidClientId,
             2, (ULONG_PTR)lpStartAddress, (ULONG_PTR)PEB_BASE);
  ……
}
```

显然，这里往下传的是 2 个参数。一个是 lpStartAddress，这是目标线程的入口地址；另一个是 PEB_BASE，就是进程环境块所在的地址。注意这里的 lpStartAddress 和 pTrueStartAddress 是不同的两个地址。读者在以前的漫谈中看到过，pTrueStartAddress 的值来自 RtlBaseProcessStartRoutine，所指向的是 RtlBaseProcessStart()。而 lpStartAddress 所指向的才是来自目标映像的程序入口。

又回到 RtlRosCreateUserThread()，下一步是系统调用 NtCreateThread()。这当然又是个比较大的操作，但是涉及用户空间格局的只是 PsCreateTeb()、即 TEB 的建立(此外还涉及内核空间堆栈的建立，但那与用户空间无关)。我们分段阅读 PsCreateTeb()的代码：

```
[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
> NtCreateThread() > PsCreateTeb()]
```

```
static NTSTATUS PsCreateTeb(HANDLE ProcessHandle, PTEB *TebPtr,
                            PETHREAD Thread, PINITIAL_TEB InitialTeb)
{
    . . . . . .

    TebSize = PAGE_SIZE;
    if (NULL == Thread->ThreadsProcess)
    {
        /* We'll be allocating a 64k block here and only use 4k of it, but this
           path should almost never be taken. Actually, I never saw it was taken,
           so maybe we should just ASSERT(NULL != Thread->ThreadsProcess) and
           move on */
        TebBase = NULL;
        Status = ZwAllocateVirtualMemory(ProcessHandle, &TebBase, 0, &TebSize,
                                         MEM_RESERVE | MEM_COMMIT | MEM_TOP_DOWN,
                                         PAGE_READWRITE);
        . . . . . .
    }
    else
    {
        Process = Thread->ThreadsProcess;
        ExAcquireFastMutex(&Process->TebLock);
        if (NULL == Process->TebBlock || Process->TebBlock == Process->TebLastAllocated)
        {
           Process->TebBlock = NULL;
           RegionSize = MM_VIRTMEM_GRANULARITY;
           Status = ZwAllocateVirtualMemory(ProcessHandle,
                                            &Process->TebBlock, 0, &RegionSize,
                                            MEM_RESERVE | MEM_TOP_DOWN,
                                            PAGE_READWRITE);
           . . . . . .
           Process->TebLastAllocated = (PVOID) ((char *) Process->TebBlock + RegionSize);
        }
        TebBase = (PVOID) ((char *) Process->TebLastAllocated - PAGE_SIZE);
        Status = ZwAllocateVirtualMemory(ProcessHandle, &TebBase, 0, &TebSize,
                                         MEM_COMMIT, PAGE_READWRITE);
        . . . . . .
        Process->TebLastAllocated = TebBase;
        ExReleaseFastMutex(&Process->TebLock);
    }
    . . . . . .
```

这部分代码基本上就是一个 if 语句。判定的条件是 Thread->ThreadsProcess 为 0，表示

目标线程并不属于任何一个进程，但是这在正常的情况下实际上是不该发生的，所以代码的作者在注释中也说“也许我们只要放上 ASSERT(NULL != Thread->ThreadsProcess)”就行。在正常的情况下，Thread->ThreadsProcess 指向所属进程的 EPROCESS 数据结构，此时又分两种情况。一种是 Process->TebBlock 为 0 或者与 Process->TebLastAllocated 相同，前者说明尚未为 TEB 保留空间，后者说明为 TEB 保留的空间已经用完。实际上，Process->TebBlock 是在 PsCreatePeb()中设置的(见前面的代码)，应该指向 PEB 所在 64KB 区间的下部边界，所以不应该是 0，而 Process->TebLastAllocated 则随着 TEB 的创建而逐次下移，每次下移一个页面，当它指向这个区间的下部边界时，就说明这个区间已经耗尽了。这时候就在原有区间的下面再分配一个区间(但是这段代码好像有点问题)。

在绝大多数的情况下，TEB 区间已经保留，其顶部的一个页面用于 PEB，并且 Process->TebLastAllocated 也已作了相应调整，现在则在它下方再获取一个页面用作目标线程的 TEB，并对 Process->TebLastAllocated 作相应的调整。

为目标线程的 TEB 分配了一个页面以后，下面就要把有关的数据写入 TEB 了。具体的数据主要来自前面的“初始 TEB”，参数 InitialTeb 就是指向初始 TEB 结构的指针。我们继续往下看：

[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
> NtCreateThread() > PsCreateTeb()]

```
    RtlZeroMemory(&Teb, sizeof(TEB));
    /* set all pointers to and from the TEB */
    Teb.Tib.Self = TebBase;
    if (Thread->ThreadsProcess)
    {
        Teb.Peb = Thread->ThreadsProcess->Peb; /* No PEB yet!! */
    }

    /* store stack information from InitialTeb */
    if(InitialTeb != NULL)
    {
      if(InitialTeb->StackBase && InitialTeb->StackLimit)    /* fixed-size stack */
      {
        Teb.Tib.StackBase = InitialTeb->StackBase;
        Teb.Tib.StackLimit = InitialTeb->StackLimit;
        Teb.DeallocationStack = InitialTeb->StackLimit;
      }
      else    /* expandable stack */
      {
       Teb.Tib.StackBase = InitialTeb->StackCommit;
       Teb.Tib.StackLimit = InitialTeb->StackCommitMax;
       Teb.DeallocationStack = InitialTeb->StackReserved;
      }
    }
```

```
    /* more initialization */
    Teb.Cid.UniqueThread = Thread->Cid.UniqueThread;
    Teb.Cid.UniqueProcess = Thread->Cid.UniqueProcess;
    Teb.CurrentLocale = PsDefaultThreadLocaleId;
    /* Terminate the exception handler list */
    Teb.Tib.ExceptionList = (PVOID)-1;
    . . . . . .
    /* write TEB data into teb page */
    Status = NtWriteVirtualMemory(ProcessHandle, TebBase,
                                        &Teb, sizeof(TEB), &ByteCount);
    . . . . . .
    if (TebPtr != NULL)
    {
        *TebPtr = (PTEB)TebBase;
    }
    return Status;
}
```

先根据初始 TEB 以及目标线程 ETHREAD 结构中的一些信息准备好一个作为草稿的 TEB 数据结构，然后通过 NtWriteVirtualMemory()将其复制到目标线程的 TEB。从代码中可见线程的 TEB 中记录着有关其堆栈的信息。TEB 数据结构的第一个成分是个 NT_TIB 数据结构，即“线程信息块”Tib。Tib 中的 StackBase 指向本线程堆栈的原点、即地址最高处，而 StackLimit 则指向堆栈所在区间的下部边界、即地址最低处。

最后，如果参数 TebPtr 非 0，还要通过这个指针返回目标线程所在的地址 TebBase。

读者很自然会有个问题：同一个进程中以后还会创建新的线程，它们的堆栈和 TEB 在那里呢？Win32 API 为随后的线程创建提供了一个库函数 CreateThread()，这个库函数则调用 CreateRemoteThread()。可别被“Remote”给搞糊涂了，这只是说也可以在别的进程中创建线程，而 CreateThread()则总是在调用者本身所在进程的内部创建线程。

```
HANDLE STDCALL
CreateRemoteThread(HANDLE hProcess, LPSECURITY_ATTRIBUTES lpThreadAttributes,
                   DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
                   LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId)
{
 . . . . . .
 PIMAGE_NT_HEADERS pinhHeader =
                              RtlImageNtHeader(NtCurrentPeb()->ImageBaseAddress);

 . . . . . .
 /* FIXME: do more checks - e.g. the image may not have an optional header */
 if(pinhHeader == NULL)
 {
    nStackReserve = 0x100000;
```

```
    nStackCommit = PAGE_SIZE;
  }
  else
  {
    nStackReserve = pinhHeader->OptionalHeader.SizeOfStackReserve;
    nStackCommit = pinhHeader->OptionalHeader.SizeOfStackCommit;
  }
. . . . . .
  /* create the thread */
  nErrCode = RtlRosCreateUserThreadVa(hProcess,
                &oaThreadAttribs, dwCreationFlags & CREATE_SUSPENDED,
                0, &nStackReserve, &nStackCommit,
                (PTHREAD_START_ROUTINE)ThreadStartup, &hThread, &cidClientId,
                2, lpStartAddress, lpParameter);
  /* success */
  if(lpThreadId) *lpThreadId = (DWORD)cidClientId.UniqueThread;
  return hThread;
}
```

从 RtlRosCreateUserThreadVa()开始往下的操作就跟以前所见到的一样了。只是所分配的堆栈区间未必就紧挨着前一个线程的堆栈，因为在此之前可能有已经在运行的线程通过 NtAllocateVirtualMemory()分配了虚存区间。所以，除第一个线程的堆栈固定在 0x20000 处以外，别的就没有固定的位置了。至于 TEB，则前面已经看到，是随着指针 Process->TebLastAllocated 的值逐次下移，每次一个页面。这样，从第一个线程的 TEB 开始，依次为 0x7FFDE000，0x7FFDD000，0x7FFDC000，0x7FFDB000 等等。

如前所述，进程环境块 PEB 的起点是 0x7FFDF000，大小为 0x1000；而从 0x7fff0000 开始的 64KB 是隔离区。那么从 0x7ffe0000 开始的 64KB 呢？“Undocumented Windows 2000 Secrets”书中说这里用作 KUSER_SHARED_DATA、即 0xffdf0000 处共享数据区的镜像，就是说从 0x7ffe0000 处可以读到本来应该在 0xffdf0000 处的数据。不过在 ReactOS 的代码中我们没有看到这样的实现。当然，要实现也很容易。

最后还要谈一下段寄存器 FS 在用户空间的作用。从前面 RtlRosInitializeContext()的代码中可以看到，在为新建线程虚构的上下文中把段寄存器 FS 的映像 Context->SegFs 设置成了 TEB_SELECTOR。这样，当新建线程进入用户空间运行时，段寄存器 FS 的内容就会“恢复”成 TEB_SELECTOR。

而当这个线程因为系统调用、中断、异常等原因而进入内核时，则 FS 的内容又被替换成 PCR_SELECTOR，这一点可以从例如_KiSystemService 的代码中看出：

```
_KiSystemService:
    /* Construct a trap frame on the stack. The following are already on the stack. */
    // SS                                                    + 0x0
    // ESP                                                   + 0x4
    // EFLAGS                                                + 0x8
```

```
// CS                                                    + 0xC
// EIP                                                    + 0x10
pushl $0                                                 // + 0x14
pushl %ebp                                               // + 0x18
pushl %ebx                                               // + 0x1C
pushl %esi                                               // + 0x20
pushl %edi                                               // + 0x24
pushl %fs                                                // + 0x28
/* Load PCR Selector into fs */
movw $PCR_SELECTOR, %bx
movw %bx, %fs
. . . . . .
```

所以，当 CPU 运行于用户空间时，段寄存器 FS 的内容是 TEB_SELECTOR；而当 CPU 运行于系统空间时则是 PCR_SELECTOR。这两个常数有什么不同呢？且看它们的定义：

```
#define NULL_SELECTOR           (0x0)
#define KERNEL_CS               (0x8)
#define KERNEL_DS               (0x10)
#define USER_CS                 (0x18 + 0x3)
#define USER_DS                 (0x20 + 0x3)
/* Task State Segment */
#define TSS_SELECTOR            (0x28)
/* Processor Control Region */
#define PCR_SELECTOR            (0x30)
/* Thread Environment Block */
#define TEB_SELECTOR            (0x38 + 0x3)
#define RESERVED_SELECTOR       (0x40)
/* Local Descriptor Table */
#define LDT_SELECTOR            (0x48)
#define TRAP_TSS_SELECTOR       (0x50)
```

就是说 PCR_SELECTOR 是 0x30，而 TEB_SELECTOR 是(0x38 + 0x3)、即 0x3b。

段寄存器的可见部分是 16 位的，其最低两位为 RPL、即运行级别；bit2 是“段描述表”选择位，为 0 时选择“全局段描述表”GDT，为 1 时选择“局部段描述表”LDT；从 bit3 到 bit15 为下标，表示从 GDT 或 LDT 中选用哪一个表项。所以：

- PCR_SELECTOR 为 0x30，表示下标为 6，选择 GDT，RPL 为 0。
- TEB_SELECTOR 为 0x3b，表示下标为 7，选择 GDT，RPL 为 3。

可见，段寄存器 FS 指向何处并不仅仅取决于它本身，也取决于 GDT 中的表项。ReactOS 代码中的数组 KiBootGdt[]提供了一个原始的 GDT 映像：

```
USHORT KiBootGdt[11 * 4] =
{
  0x0, 0x0, 0x0, 0x0,                    /* Null */
```

```
  0xffff, 0x0000, 0x9a00, 0x00cf,          /* Kernel CS */
  0xffff, 0x0000, 0x9200, 0x00cf,          /* Kernel DS */
  0xffff, 0x0000, 0xfa00, 0x00cf,          /* User CS */
  0xffff, 0x0000, 0xf200, 0x00cf,          /* User DS */
  0x0, 0x0, 0x0, 0x0,                      /* TSS */
  0x0fff, 0x0000, 0x9200, 0xff00,          /* PCR */
  0x0fff, 0x0000, 0xf200, 0x0000,          /* TEB */
  0x0, 0x0, 0x0, 0x0,                      /* Reserved */
  0x0, 0x0, 0x0, 0x0,                      /* LDT */
  0x0, 0x0, 0x0, 0x0                       /* Trap TSS */
};
```

表中的每一行代表 GDT 的一个表项，每个表项的大小是 4 个 16 位短字，即 64 位，按从低位到高位的次序排列。“Intel 系统结构软件开发手册”第 3 卷中有对 GDT 表项格式的说明，此处不拟详述，只是简要地作一些说明：

- 下标为 6 的表项是 PCR，数据 0x0fff 和 0x0000 表示段的长度为 4KB，数据 0x9200 表示其优先级 DPL 为最高的 0 级，为数据描述项、类型为 2、即可读可写。其余数据表明基地址为 0xff000000。
- 下标为 7 的表项是 TEB，数据 0x0fff 和 0xff00 表示段的长度为 4KB，数据 0xf200 表示其优先级 DPL 为最低的 3 级，为数据描述项、类型为 2、即可读可写。其余数据表明基地址为 0。

这个数据结构所提供的只是系统刚引导后的原始 PCR 表项和 TEB 表项，它们与基地址有关的位段随着系统的运行还要改变。

首先是下标为 6 的 PCR 表项。在单 CPU 的系统中这个段的基地址是 0xff000000，以后读者将会看到，这实际上是个指针，指向内核中代表着 CPU 的“处理器控制区”，即 KPCR 数据结构。但是，在多 CPU 的系统中则每个 CPU 都有这么一个数据结构，因而它们的基地址要互相岔开。所以系统在初始化时要根据具体情况改变各 CPU 的 GDT 中的这个表项，这是在函数 KiInitializeGdt()中完成的：

［KiSystemStartup() > KeApplicationProcessorInit() > KiInitializeGdt()］

```
KiInitializeGdt(PKPCR Pcr)
{
   ......
   /*
    * Set the base address of the PCR
    */
   Base = (ULONG)Pcr;
   Entry = PCR_SELECTOR / 2;
   Gdt[Entry + 1] = (USHORT)(((ULONG)Base) & 0xffff);

   Gdt[Entry + 2] = Gdt[Entry + 2] & ~(0xff);
   Gdt[Entry + 2] = (USHORT)(Gdt[Entry + 2] | ((((ULONG)Base) & 0xff0000) >> 16));
```

```
    Gdt[Entry + 3] = Gdt[Entry + 3] & ~(0xff00);
    Gdt[Entry + 3] = (USHORT)(Gdt[Entry + 3] | ((((ULONG)Base) & 0xff000000) >> 16));
    ......
}
```

参数 Pcr 是个指针，具体的值取决于这是系统中第几个 CPU 的 GDT，但总是在 0xff000000 以上不远的地方，因为每个 CPU 只占一个页面。这里的代码，如果读者有兴趣阅读的话，需要结合 Intel 手册中 GDT 表项的格式说明才能明白，这里就不多花时间了。

当然，这里修改的是数据结构的内容，只是 GDT 的映像，修改完了以后还要通过执行 lgdt 指令把这映像装入 CPU。这样，当 CPU 运行于系统空间时，%%fs:0 就总是指向 0xff000000 以上不远的某个地方，这就是所在 CPU 的 KPCR 数据结构。

这里还要说明一下，“Undocumented Windows 2000 Secrets”书中说当 CPU 运行于系统空间时%%fs:0 指向 0xffdff000，而不是我们从 ReactOS 代码中所见的 0xff000000。不过有一点是共同的，那就是都说指向 KPCR 数据结构。其实地址的绝对值在这里并不那么重要，重要的是指向 KPCR，找到 KPCR 数据结构才是关键。另一方面，之所以要为获取 KPCR 结构的地址而专门使用一个段寄存器，正是因为这个地址可能并不固定，否则就不合理了。

下标为 7 的 TEB 表项则又不同。每当内核在调度一个线程运行、并且要切换到这个目标线程时，就要根据这是在哪个 CPU 上运行而修改相应 GDT 中的这个表项，使其指向目标线程的 TEB。下面是函数 Ki386ContextSwitch()中的片断：

```
_Ki386ContextSwitch
        ......
        /*
        * Get the pointer to the new thread.
        */
        movl    8(%ebp), %ebx

        /* Set the base of the TEB selector to the base of the TEB for this thread. */
        pushl   %ebx
        pushl   KTHREAD_TEB(%ebx)
        pushl   $TEB_SELECTOR
        call    _KeSetBaseGdtSelector
        addl    $8, %esp
        popl    %ebx
        ......
```

调用函数 KeSetBaseGdtSelector()是为了设置 GDT 表项中的基地址部分，这个函数有两个参数。第一个是目标表项的“选择码”，表示选择哪一个表项，这里压入堆栈的是 TEB_SELECTOR，这就是当线程进入用户空间时寄存器 FS 应有的数值。第二个参数是新的基地址，这里来自 KTHREAD_TEB(%ebx)。寄存器 Ebx 的值来自 KeSetBaseGdtSelector()的第一个调用参数，这是个指向目标线程 KTHREAD 数据结构的指针。另一方面，常数 KTHREAD_TEB 定义为 0x20，而 KTHREAD 数据结构中位移为 0x20 的字段就指向该线程的 TEB。所以，KTHREAD_TEB(%ebx)就是目标线程的 TEB 起始地址。当然，修改后的映像也要通过 lgdt 指令装入 CPU。

这样，在刚完成线程切换的时侯，GDT 中下标为 7 的表项已经指向新线程的 TEB，但是此时寄存器 FS 的内容还是 PCR_SELECTOR、即下标为 6。而当目标线程进入用户空间时，FS 的内容就变成了 TEB_SELECTOR，下标变成了 7。于是，在用户空间，%%fs:0 就总是指向当前线程的 TEB 了。