漫谈兼容内核之十七:
再谈 Windows 的进程创建

毛德操

在漫谈之十中。我根据“Microsoft Windows Internals 4e”一书第六章的叙述介绍了 Windows 的进程创建和映像装入的过程。但是，由于缺乏源代码的支撑，这样的叙述对于只是想对此有个大致了解的读者固然不无帮助，可是对于需要实际从事研发、特别是兼容内核开发的读者就显得过于抽象笼统了。不幸，Windows 内核的代码是不公开的，我们无法通过 Windows 内核的代码来确切地了解和理解它的方方面面。虽说是“科学无禁区”，但是现实往往不那么理想。幸运的是我们有了 ReactOS。当然，ReactOS 不等于 Windows，但是读者将会看到，至少就 Windows 进程的创建而言，它的代码和“Internals”书中的叙述还是相当吻合的。本文引用的代码均取自 ReactOS 的 0.2.6 版，大致上是一年前的版本。

正如“Internals”所述，Windows 进程的创建是个复杂的过程，分成好几个步骤，涉及到好几个系统调用。Win32 API 函数 CreateProcessW()就是这些步骤的整合。这是由动态连接库 kernel32.dll 导出的库函数，其主体就在这个 DLL 中。还有个 CreateProcessA()，是与 CreateProcessW()连在一起的，前者接受 ASCII 码的字符串，而后者要求使用“宽字符”、即 Unicode 的字符串。实际上 CreateProcessA()只是把 ASCII 字符串转换成 Unicode 字符串，然后就调用 CreateProcessW()。

[CreateProcessW()]

```
BOOL STDCALL
CreateProcessW (LPCWSTR lpApplicationName,    LPWSTR lpCommandLine,
                LPSECURITY_ATTRIBUTES lpProcessAttributes,
                LPSECURITY_ATTRIBUTES lpThreadAttributes,
                BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment,
                LPCWSTR lpCurrentDirectory, LPSTARTUPINFOW lpStartupInfo,
                LPPROCESS_INFORMATION lpProcessInformation)
{
    . . . . . .

    TidyCmdLine = GetFileName(lpCurrentDirectory, lpApplicationName, lpCommandLine,
                                        Name, sizeof(Name) / sizeof(WCHAR));
    . . . . . .
    if (lpApplicationName != NULL && lpApplicationName[0] != 0)
      {
        wcscpy (TempApplicationNameW, lpApplicationName);
        i = wcslen(TempApplicationNameW);
        if (TempApplicationNameW[i - 1] == L'.')
        {
            TempApplicationNameW[i - 1] = 0;
```

```
        }
        else
        {
            s = max(wcsrchr(TempApplicationNameW, L'\\'),
                    wcsrchr(TempApplicationNameW, L'/'));
            if (s == NULL)
            {
                s = TempApplicationNameW;
            }
            else
            {
                s++;
            }
            e = wcsrchr(s, L'.');
            if (e == NULL)
            {
                wcscat(s, L".exe");
                e = wcsrchr(s, L'.');
            }
        }
    }
    else if (L'"' == TidyCmdLine[0])
    {
        wcscpy(TempApplicationNameW, TidyCmdLine + 1);
        s = wcschr(TempApplicationNameW, L'"');
        if (NULL == s)
        {
            return FALSE;
        }
        *s = L'\0';
    }
    else
    {
        wcscpy(TempApplicationNameW, TidyCmdLine);
        s = wcschr(TempApplicationNameW, L' ');
        if (NULL != s)
        {
            *s = L'\0';
        }
    }
    s = max(wcsrchr(TempApplicationNameW, L'\\'), wcsrchr(TempApplicationNameW, L'/'));
    if (NULL == s)
    {
        s = TempApplicationNameW;
```

```
}
s = wcsrchr(s, L'.');
if (NULL == s)
{
        wcscat(TempApplicationNameW, L".exe");
}

if (!SearchPathW(NULL, TempApplicationNameW, NULL,
                                sizeof(ImagePathName)/sizeof(WCHAR), ImagePathName, &s))
{
   return FALSE;
}

e = wcsrchr(s, L'.');
if (e != NULL && (!_wcsicmp(e, L".bat") || !_wcsicmp(e, L".cmd")))
{
       // the command is a batch file
       IsBatchFile = TRUE;
       if (lpApplicationName != NULL && lpApplicationName[0])
       {
     // FIXME: use COMSPEC for the command interpreter
     wcscpy(TempCommandLineNameW, L"cmd /c ");
     wcscat(TempCommandLineNameW, lpApplicationName);
     lpCommandLine = TempCommandLineNameW;
     wcscpy(TempApplicationNameW, L"cmd.exe");
     if (!SearchPathW(NULL, TempApplicationNameW, NULL,
                           sizeof(ImagePathName)/sizeof(WCHAR), ImagePathName, &s))
     {
          return FALSE;
     }
  }
  else
  {
          return FALSE;
  }
}

/* Process the application name and command line */
RtlInitUnicodeString(&ImagePathName_U, ImagePathName);
RtlInitUnicodeString(&CommandLine_U, IsBatchFile ? lpCommandLine : TidyCmdLine);

. . . . . .

/* Initialize the current directory string */
```

```
if (lpCurrentDirectory != NULL)
{
    RtlInitUnicodeString(&CurrentDirectory_U, lpCurrentDirectory);
}
else
{
    GetCurrentDirectoryW(256, TempCurrentDirectoryW);
    RtlInitUnicodeString(&CurrentDirectory_U, TempCurrentDirectoryW);
}

/* Create a section for the executable */

hSection = KlMapFile (ImagePathName);
```

因为这是个 W32 API 函数，对于调用参数这里就不作解释了。

代码中首先是对应用名和命令行的处理。这里要考虑许多不同的情况。例如：

- 调用参数 lpApplicationName 或 lpCommandLine 可能是空指针。
- 命令行中的应用名、即可执行文件名可能是加引号的，也可能是不加引号的。
- 应用名可能是个完整的路径，也可能只是不带路径的应用名。
- 应用名可能带扩展名(例如.exe)，也可能不带扩展名。
- 应用名后面可能带一个点，但是不带扩展名字符。
- 如果不带扩展名，那么应用名可能是指一个.exe 文件，也可能是指.com 或.bat 文件。

要是在文件系统的指定目录中发现应用名所代表的是.com 或.bat 文件，那就要把应用名改成 cmd.exe，而把原来的应用名和命令行作为传递给 cmd.exe 的参数。

具体的代码就留给读者自己阅读了。由于这里因篇幅的考虑而作了删节，读者最好还是阅读原始的 ReactOS 代码。代码中 wcscpy()一类的函数相当于 strcpy()一类，只不过所处理的是宽字符而不是普通的 ASCII 字符。

这里的最后一步操作是 KlMapFile()，目的是为目标映像建立一个共享内存区、即 Section 对象。不过，对于 CreateProcessW()而言，这其实还只能说是第一步。

[CreateProcessW() > KlMapFile()]

```
HANDLE KlMapFile(LPCWSTR lpApplicationName)
{
    . . . . . .

    InitializeObjectAttributes(&ObjectAttributes, &ApplicationNameString,
                               OBJ_CASE_INSENSITIVE, NULL, SecurityDescriptor);

    /* Try to open the executable */

    Status = NtOpenFile(&hFile, SYNCHRONIZE|FILE_EXECUTE|FILE_READ_DATA,
                &ObjectAttributes, &IoStatusBlock,
                FILE_SHARE_DELETE|FILE_SHARE_READ,
```

```
                FILE_SYNCHRONOUS_IO_NONALERT|FILE_NON_DIRECTORY_FILE);

    . . . . . .
    Status = NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, NULL,
                                        PAGE_EXECUTE, SEC_IMAGE,
                                        hFile);
    NtClose(hFile);
    . . . . . .
    return(hSection);
}
```

先打开目标映像文件，再通过系统调用 NtCreateSection()为已经打开的映像文件创建一个共享内存区对象。注意 NtCreateSection()只是创建了一个共享内存区对象，并将它与一个已打开文件挂上钩，而并未将其映射到任何进程的用户空间，所以函数名 KlMapFile()不免误导。由于调用时使用了参数 SEC_IMAGE，表明目标文件是个映像文件，NtCreateSection()会对目标文件的头部进行检验，以确认其为 PE 格式的可执行映像。如果发现并非 PE 格式映像，就会通过 hSection 返回 0。

回到 CreateProcessW()的代码中，如果 KlMapFile()的返回值非 0 就说明为目标映像文件创建的共享内存区对象已经成功，下面就可以用这个已打开对象(hSection 为其 Handle)去创建进程了。可是，如果返回值是 0，那就说明目标映像文件并不是一个 PE 格式的文件。既然扩充名是.exe，却又不是 PE 格式的文件，那是怎么回事呢？原来，DOS 格式的可执行文件也用.exe 作为扩充名，这种文件的头部并非 PE 格式，但是也有 DOS 格式的“签名”IMAGE_DOS_SIGNATURE 可供验证。

[CreateProcessW()]

```
    if (hSection == NULL)
    {
        . . . . . .
        DPRINT("Inspecting Image Header for image type id\n");
        . . . . . .
        InitializeObjectAttributes(&ObjectAttributes, &ApplicationNameString,
                        OBJ_CASE_INSENSITIVE, NULL, SecurityDescriptor);

        // Try to open the executable
        Status = NtOpenFile(&hFile, SYNCHRONIZE|FILE_EXECUTE|FILE_READ_DATA,
            &ObjectAttributes, &IoStatusBlock,
            FILE_SHARE_DELETE|FILE_SHARE_READ,
            FILE_SYNCHRONOUS_IO_NONALERT|FILE_NON_DIRECTORY_FILE);
        . . . . . .

        // Read the dos header
        Offset.QuadPart = 0;
        Status = ZwReadFile(hFile, NULL, NULL, NULL, &Iosb,
```

```
                                &DosHeader, sizeof(DosHeader), &Offset, 0);
        . . . . . .
        if (Iosb.Information != sizeof(DosHeader)) {
            DPRINT("Failed to read dos header from file\n");
            SetLastErrorByStatus(STATUS_INVALID_IMAGE_FORMAT);
            return FALSE;
        }

        // Check the DOS signature
        if (DosHeader.e_magic != IMAGE_DOS_SIGNATURE) {
            DPRINT("Failed dos magic check\n");
            SetLastErrorByStatus(STATUS_INVALID_IMAGE_FORMAT);
            return FALSE;
        }
        NtClose(hFile);

        DPRINT("Launching VDM...\n");
        return CreateProcessW(L"ntvdm.exe", (LPWSTR)lpApplicationName,
                lpProcessAttributes, lpThreadAttributes, bInheritHandles, dwCreationFlags,
                lpEnvironment, lpCurrentDirectory, lpStartupInfo, lpProcessInformation);
    }
```

DOS 格式的可执行映像都是 16 位的，不能直接在 32 位的 WinNT 内核上运行，而得要借助系统工具软件 vdm.exe 的支持才能运行。为了在 32 位 x86 结构的 CPU 芯片上兼容为 16 位芯片开发的软件，Intel 在 x86 芯片上提供了一种“虚拟机模式”。而 VDM.、即“虚拟 DOS 机”，则是微软开发的一种系统软件，利用 x86 芯片的虚拟机模式为 16 位的 DOS 应用软件供一个虚拟的 DOS 环境，使 DOS 应用软件感觉到就好像是在 DOS 操作系统上运行一样。可想而知，ntvdm.exe 就是实现于 WinNT 内核上的 VDM。注意 ntvdm.exe 本身是 32 位软件，它所支持的目标软件才是 16 位的，ntvdm.exe 连同其所支持的目标软件一起作为一个进程在 WinNT 内核上运行。所以，对于 16 位软件这里递归地调用 CreateProcessW()，而以 ntvdm.exe 作为新的应用名，但是命令行则不变。

当然，16 位软件只是少数，WinNT 上运行的绝大多数软件都是 32 位 PE 格式的，所以 NtCreateSection()一般都会返回一个非 0 的 Handle，从而跳过上面这段代码。我们继续往下看：

[CreateProcessW()]

```
    /* Get some information about the executable */
    Status = ZwQuerySection(hSection, SectionImageInformation, &Sii, sizeof(Sii), &i);
    . . . . . .
    if (0 != (Sii.Characteristics & IMAGE_FILE_DLL))
    {
       NtClose(hSection);
       DPRINT("Can't execute a DLL\n");
```

```
   SetLastError(ERROR_BAD_EXE_FORMAT);
   return FALSE;
}

if (IMAGE_SUBSYSTEM_WINDOWS_GUI != Sii.Subsystem
      &&   IMAGE_SUBSYSTEM_WINDOWS_CUI != Sii.Subsystem)
{
   NtClose(hSection);
   DPRINT("Invalid subsystem %d\n", Sii.Subsystem);
   SetLastError(ERROR_CHILD_NOT_COMPLETE);
   return FALSE;
}

/* Initialize the process object attributes */
if(lpProcessAttributes != NULL)
{
   if(lpProcessAttributes->bInheritHandle)
   {
      ProcAttributes |= OBJ_INHERIT;
   }
   ProcSecurity = lpProcessAttributes->lpSecurityDescriptor;
}

InitializeObjectAttributes(&ProcObjectAttributes, NULL,
                                    ProcAttributes, NULL, ProcSecurity);
/* initialize the process priority class structure */
PriorityClass.Foreground = FALSE;

if(dwCreationFlags & IDLE_PRIORITY_CLASS)
{
   PriorityClass.PriorityClass = PROCESS_PRIORITY_CLASS_IDLE;
}
else if(dwCreationFlags & BELOW_NORMAL_PRIORITY_CLASS)
{
   PriorityClass.PriorityClass = PROCESS_PRIORITY_CLASS_BELOW_NORMAL;
}
else if . . . . . .
. . . . . .

/* Create a new process */
Status = NtCreateProcess(&hProcess, PROCESS_ALL_ACCESS,
                              &ProcObjectAttributes, NtCurrentProcess(),
                              bInheritHandles, hSection, NULL, NULL);
. . . . . .
```

为目标映像创建的 Section 对象中含有许多来自目标映像 PE 头部的信息，可以通过 ZwQuerySection()、即 NtQuerySection()询问、获取这些信息。所获取的信息在数据结构 Sii 中，这是个 SECTION_IMAGE_INFORMATION 数据结构，其 Subsystem 字段表明了映像的模式。一个 PE 映像可以是 GUI 模式的、面向“视窗”和图像的应用，也可以是“控制台”、即 CUI 模式的面向命令行和字符的应用。但是二者必居其一，否则就错了。

然后，这里对一个局部量的数据结构 PriorityClass 进行了一些设置，设置的依据来自调用参数 dwCreationFlags 中的一些标志位

接着就是对 NtCreateProcess()的调用了。不过刚才的 PriorityClass 并不用于进程的创建，而是用于进程创建之后，这里只是先作好准备。

[CreateProcessW() > NtCreateProcess()]

```
NTSTATUS STDCALL
NtCreateProcess(OUT PHANDLE ProcessHandle,    IN ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
            IN HANDLE ParentProcess,    IN BOOLEAN InheritObjectTable,
            IN HANDLE SectionHandle    OPTIONAL,
            IN HANDLE DebugPort    OPTIONAL,    IN HANDLE ExceptionPort    OPTIONAL)
{
    KPROCESSOR_MODE PreviousMode;
    NTSTATUS Status = STATUS_SUCCESS;

    PAGED_CODE();
    PreviousMode = ExGetPreviousMode();
    if(PreviousMode != KernelMode)
    {
       _SEH_TRY     _SEH_END;
    }

    if(ParentProcess == NULL)
    {
       Status = STATUS_INVALID_PARAMETER;
    }
    else
    {
       Status = PspCreateProcess(ProcessHandle, DesiredAccess, ObjectAttributes,
                                 ParentProcess, InheritObjectTable, SectionHandle,
                                 DebugPort, ExceptionPort);
    }
    return Status;
}
```

显然，NtCreateProcess()只是 PspCreateProcess()的包装，其作用主要是把有关的参数从

用户空间拷贝到内核中，并进行一些参数合理性的检查。这里的参数 DesiredAccess、ObjectAttributes、和 InheritObjectTable 与一般创建对象时所用的大致相同。参数 ParentProcess 则是父进程的 Handle，一般这就是当前进程，但也可以不是，也就是说当前进程可以为别的进程创建子进程。另一个参数 SectionHandle 是一个共享内存区的 Handle，这个共享内存区代表着目标映像文件。最后是两个进程间通信端口 DebugPort 和 ExceptionPort 的 Handle。顾名思义，这两个端口是供新建进程发送调试信息和异常处理信息的端口，是可以为所有进程共用的系统资源。

下面我们看 PspCreateProcess()的代码：

[CreateProcessW() > NtCreateProcess() > PspCreateProcess()]

```
NTSTATUS
PspCreateProcess(OUT PHANDLE ProcessHandle,  IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes  OPTIONAL,
        IN HANDLE ParentProcess  OPTIONAL,  IN BOOLEAN InheritObjectTable,
        IN HANDLE SectionHandle  OPTIONAL,
        IN HANDLE DebugPort  OPTIONAL,  IN HANDLE ExceptionPort  OPTIONAL)
{
  . . . . . .

  PreviousMode = ExGetPreviousMode();
  . . . . . .
  if(ParentProcess != NULL)
  {
    Status = ObReferenceObjectByHandle(ParentProcess,
                PROCESS_CREATE_PROCESS, PsProcessType,
                PreviousMode, (PVOID*)&pParentProcess, NULL);
    . . . . . .
  }
  else
  {
    pParentProcess = NULL;
  }

  . . . . . .

  if (SectionHandle != NULL)
  {
      Status = ObReferenceObjectByHandle(SectionHandle,
                                0, MmSectionObjectType, PreviousMode,
                                (PVOID*)&SectionObject, NULL);
      . . . . . .
  }
```

```
Status = ObCreateObject(PreviousMode, PsProcessType, ObjectAttributes,
                PreviousMode, NULL, sizeof(EPROCESS), 0, 0, (PVOID*)&Process);
. . . . . .

KProcess = &Process->Pcb;
RtlZeroMemory(Process, sizeof(EPROCESS));
Status = PsCreateCidHandle(Process, PsProcessType, &Process->UniqueProcessId);
. . . . . .
Process->DebugPort = pDebugPort;
Process->ExceptionPort = pExceptionPort;
. . . . . .

KeInitializeDispatcherHeader(&KProcess->DispatcherHeader,
                                ProcessObject, sizeof(EPROCESS), FALSE);

/* Inherit parent process's affinity. */
if(pParentProcess != NULL)
{
   KProcess->Affinity = pParentProcess->Pcb.Affinity;
   Process->InheritedFromUniqueProcessId = pParentProcess->UniqueProcessId;
   Process->SessionId = pParentProcess->SessionId;
}
else
{
   KProcess->Affinity = KeActiveProcessors;
}

KProcess->BasePriority = PROCESS_PRIO_NORMAL;
KProcess->IopmOffset = 0xffff;
KProcess->LdtDescriptor[0] = 0;
KProcess->LdtDescriptor[1] = 0;
InitializeListHead(&KProcess->ThreadListHead);
KProcess->ThreadQuantum = 6;
KProcess->AutoAlignment = 0;
MmInitializeAddressSpace(Process, &Process->AddressSpace);

ObCreateHandleTable(pParentProcess, InheritObjectTable, Process);
MmCopyMmInfo(pParentProcess ? pParentProcess : PsInitialSystemProcess, Process);

KeInitializeEvent(&Process->LockEvent, SynchronizationEvent, FALSE);
Process->LockCount = 0;
Process->LockOwner = NULL;

Process->Win32WindowStation = (HANDLE)0;
```

```
    ExAcquireFastMutex(&PspActiveProcessMutex);
    InsertTailList(&PsActiveProcessHead, &Process->ProcessListEntry);
    InitializeListHead(&Process->ThreadListHead);
    ExReleaseFastMutex(&PspActiveProcessMutex);

    ExInitializeFastMutex(&Process->TebLock);
    Process->Pcb.State = PROCESS_STATE_ACTIVE;

    /* 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

    /* 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);
    ......

#if 1
    /* FIXME - the handle should be created after all things are initialized, NOT HERE! */
    Status = ObInsertObject ((PVOID)Process, NULL, DesiredAccess, 0, NULL, &hProcess);
    ......
#endif
```

对于新建的进程而言，其父进程是至关重要的，因为它可能需要从父进程继承许多资源和性质、例如已经打开的对象等等。所以，这里先根据父进程的 Handle 获取指向其 EPROCESS 数据结构的指针。当然，已经创建并打开的 Section 对象也是至关重要，因为它代表着目标映像，所以也要根据其 Handle 获取指向其 SECTION_OBJECT 数据结构的指针。

接着就通过 ObCreateObject()创建新进程的进程对象。当然，对象的类型是 PsProcessType。我们知道，进程对象的数据结构是 EPROCESS，其头部是 KPROCESS，而 KPROCESS 的头部则是 DISPATCHER_HEADER。代码中通过 KeInitializeDispatcherHeader()对此进行了初始化。进程对象也是“可等待”对象，可以作为等待的目标。

读者要注意区分“对象”和“已打开对象”。“对象”是实体，是归系统所有的；而“已打开对象”则只是一个进程对某个对象的访问权和上下文。就像因为对于文件对象的管理而需要一个有组织的“文件系统”一样，系统对于进程和线程也需要有组织的管理。为此，系统为每个进程和线程都创建一个 CID 对象，加以统一管理。CID 是“客户身份号(Client ID)”的缩写，一般由进程号和线程号两部分构成。系统有个 CID 表 PspCidTable，类似于每个进程的已打开对象表。

有了进程对象以后，下一步就是通过 MmInitializeAddressSpace()为它创建一个用户空间。所谓一个地址空间，实际上就是一套数据结构、就是一个“账本”。我们在这里就不深入到具体的代码中去了，熟悉 Linux 存储管理的读者应该不难理解此中的机理。

接着的ObCreateHandleTable()为新建进程创建其打开对象表，这很简单。而InsertTailList()则把新建进程的数据结构挂入系统的进程队列。注意这并不意味着这个进程从此就有可能被调度运行，因为在 Windows 中受调度运行的是线程而不是进程。

现在我们已经有了新建进程的进程对象和一个空白的用户空间，下面就要对其用户空间进行一些初步的“城市规划”了，这就是下面的一连串 MmCreateMemoryArea()。这些调用的形式都是一样，只是参数不同，特别是变量 BaseAddress 的值各不相同。以代码中的第一次调用为例，此时的 BaseAddress 设置成另一个变量 MmUserProbeAddress 的值，据查这个变量的值为 0x7fff0000。我们知道，Windows 把用户空间与系统空间之间的边界划在 0x80000000。所以，0x7fff0000 是在边界下方 64KB 的地方。而另一个参数表示所需的区间大小为 0x10000，那就是 64KB。至于区间的类型是 MEMORY_AREA_NO_ACCESS，访问模式为 PAGE_NOACCESS。显然，这是要建立一个“无人区”。还有几个就留给读者自己看了，这些还只是属于“城市规划”，下面才是“基本建设”。

上面最后的 ObInsertObject()把新建的进程对象插入当前进程的打开对象表。对于当前进程，新建的进程对象当然是已经打开的对象。

我们继续往下看“基本建设”：

[CreateProcessW() > NtCreateProcess() > PspCreateProcess()]

```
    /* FIXME - Map ntdll */
    Status = LdrpMapSystemDll(hProcess, &LdrStartupAddr);
                                /* FIXME - hProcess shouldn't be available at this point! */
    . . . . . .
    /* Map the process image */
    if (SectionObject != NULL)
    {
       ULONG ViewSize = 0;
       Status = MmMapViewOfSection(SectionObject, Process, (PVOID*)&ImageBase,
                                                  0, ViewSize, NULL, &ViewSize, 0,
                                                  MEM_COMMIT, PAGE_READWRITE);
       ObDereferenceObject(SectionObject);
       . . . . . .
    }

  if(pParentProcess != NULL)
  {
     /*
      * Duplicate the token
      */
     Status = SepInitializeNewProcess(Process, pParentProcess);
     . . . . . .
  }
  else
  {
     /* FIXME */
  }

    . . . . . .
    Status = PsCreatePeb(hProcess, Process, ImageBase);
                                /* FIXME - hProcess shouldn't be available at this point! */
    . . . . . .

    /*
     * Maybe send a message to the creator process's debugger
     */

    PspRunCreateProcessNotifyRoutines(Process, TRUE);

    . . . . . .

    return Status;
}
```

读者想必知道，LdrpMapSystemDll()的作用就是把 ntdll.dll 的映像映射到目标进程的用户空间，还包括获取其 LdrInitializeThunk()、KiUserApcDispatcher()等函数的入口地址，具体的代码这里就不看了。接着的 MmMapViewOfSection()则把目标 EXE 映像也映射到目标进程的用户空间。至于 PsCreatePeb()，当然是创建 PEB。PEB 在用户空间的位置是 0x7FFDF000。

最后的 PspRunCreateProcessNotifyRoutines()依次调用预先设置在一个函数指针数组 PiProcessNotifyRoutine[]中的函数，向有关方面发出已经创建了一个进程的通知。不过在 ReactOS 的 0.2.6 版中似乎还没有谁来设置这些函数，所以还只是空操作。

倒是 PsCreatePeb()还需要补充几句，因为从代码中看它还搞了点“副业”，就是把一个用于 NLS 的 Section 对象 NlsSectionObject 所代表的映像也映射到了目标进程的用户空间。这是因为 PEB 中的一些字段本来就与所用的语言文字有关。NLS 是“National Language Support”即“本国语言支持”的缩写，主要就是不同的文字输入法。不过对此恐怕得要另作研究、另行撰文介绍，否则这儿就离题太远了。

至此，作为一个对象的新建进程已经创建，并且已经挂入进程队列。不过，由 NtCreateProcess()创建的新建进程在某些方面还是空白，有些手续也尚未完成，还是一个半成品，需要由创建者进一步将其变为成品并完成有关的手续，主要包括：

- 设置新建进程的调度优先级。
- 建立新建进程的“进程参数块”PPB。
- 让新建进程从父进程继承允许遗传的已打开对象的 Handle，把这些 Handle 复制给新建进程。
- 向 csrss 发出已经创建了一个进程的通知。
- 把 PPB 中的数据写入新建进程的 PEB。

下面我们回到 CreateProcessW()的代码。

[CreateProcessW()]

```
    Status = NtSetInformationProcess(hProcess, ProcessPriorityClass,
                                     &PriorityClass, sizeof(PROCESS_PRIORITY_CLASS));
    . . . . . .

    /* Create the PPB */
    RtlCreateProcessParameters(&Ppb, &ImagePathName_U, NULL,
                               lpCurrentDirectory ? &CurrentDirectory_U : NULL,
                               &CommandLine_U, lpEnvironment, NULL, NULL, NULL,
                               lpStartupInfo && lpStartupInfo->lpReserved2 ?
                                                          &RuntimeInfo_U : NULL);
    . . . . . .
    /*
     * Translate some handles for the new process
     */
    if (Ppb->CurrentDirectoryHandle)
    {
        Status = NtDuplicateObject (NtCurrentProcess(),Ppb->CurrentDirectoryHandle,
```

```
                                hProcess, &Ppb->CurrentDirectoryHandle, 0, TRUE,
                                DUPLICATE_SAME_ACCESS);
}

/* Close the section */
NtClose(hSection);

/* Get some information about the process */
NtQueryInformationProcess(hProcess,
                          ProcessBasicInformation,
                          &ProcessBasicInfo,
                          sizeof(ProcessBasicInfo),
                          &retlen);
DPRINT("ProcessBasicInfo.UniqueProcessId 0x%x\n",
    ProcessBasicInfo.UniqueProcessId);
lpProcessInformation->dwProcessId = (DWORD)ProcessBasicInfo.UniqueProcessId;

/* Tell the csrss server we are creating a new process */
CsrRequest.Type = CSRSS_CREATE_PROCESS;
CsrRequest.Data.CreateProcessRequest.NewProcessId =
   ProcessBasicInfo.UniqueProcessId;
if (Sii.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_GUI)
   {
      /* Do not create a console for GUI applications */
      dwCreationFlags &= ~CREATE_NEW_CONSOLE;
      dwCreationFlags |= DETACHED_PROCESS;
   }
else if (Sii.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_CUI)
   {
      if (NULL == Ppb->hConsole)
         {
            dwCreationFlags |= CREATE_NEW_CONSOLE;
         }
   }
CsrRequest.Data.CreateProcessRequest.Flags = dwCreationFlags;
CsrRequest.Data.CreateProcessRequest.CtrlDispatcher = ConsoleControlDispatcher;
Status = CsrClientCallServer(&CsrRequest, &CsrReply,
                         sizeof(CSRSS_API_REQUEST), sizeof(CSRSS_API_REPLY));
. . . . . .

Ppb->hConsole = CsrReply.Data.CreateProcessReply.Console;

InputSet = FALSE;
OutputSet = FALSE;
```

```
ErrorSet = FALSE;

/* Set the child console handles */

/* First check if handles were passed in startup info */
if (lpStartupInfo && (lpStartupInfo->dwFlags & STARTF_USESTDHANDLES))
{
    if (lpStartupInfo->hStdInput)
    {
        Ppb->hStdInput = lpStartupInfo->hStdInput;
        InputSet = TRUE;
        InputDup = TRUE;
    }
    if (lpStartupInfo->hStdOutput)
    {
  Ppb->hStdOutput = lpStartupInfo->hStdOutput;
        OutputSet = TRUE;
        OutputDup = TRUE;
    }
    if (lpStartupInfo->hStdError)
    {
  Ppb->hStdError = lpStartupInfo->hStdError;
        ErrorSet = TRUE;
        ErrorDup = TRUE;
    }
}

/* Check if new console was created, use it for input and output if not overridden */
if (0 != (dwCreationFlags & CREATE_NEW_CONSOLE)
      && NT_SUCCESS(Status) && NT_SUCCESS(CsrReply.Status))
{
    if (! InputSet)
    {
        Ppb->hStdInput = CsrReply.Data.CreateProcessReply.InputHandle;
        InputSet = TRUE;
        InputDup = FALSE;
    }
    if (! OutputSet)
    {
        Ppb->hStdOutput = CsrReply.Data.CreateProcessReply.OutputHandle;
        OutputSet = TRUE;
        OutputDup = FALSE;
    }
    if (! ErrorSet)
```

```
    {
        Ppb->hStdError = CsrReply.Data.CreateProcessReply.OutputHandle;
        ErrorSet = TRUE;
        ErrorDup = FALSE;
    }
}

/* Use existing handles otherwise */
if (! InputSet)
{
    Ppb->hStdInput = NtCurrentPeb()->ProcessParameters->hStdInput;
    InputDup = TRUE;
}
if (! OutputSet)
{
    Ppb->hStdOutput = NtCurrentPeb()->ProcessParameters->hStdOutput;
    OutputDup = TRUE;
}
if (! ErrorSet)
{
    Ppb->hStdError = NtCurrentPeb()->ProcessParameters->hStdError;
    ErrorDup = TRUE;
}

/* Now duplicate handles if required */
if (InputDup)
{
    if (IsConsoleHandle(Ppb->hStdInput))
    {
        Ppb->hStdInput = CsrReply.Data.CreateProcessReply.InputHandle;
    }
    else
    {
        DPRINT("Duplicate input handle\n");
        Status = NtDuplicateObject (NtCurrentProcess(), Ppb->hStdInput,
                                    hProcess, &Ppb->hStdInput,
                                    0, TRUE, DUPLICATE_SAME_ACCESS);
        . . . . . .
    }
}

if (OutputDup)
{
    if (IsConsoleHandle(Ppb->hStdOutput))
```

```
    {
        Ppb->hStdOutput = CsrReply.Data.CreateProcessReply.OutputHandle;
    }
    else
    {
        DPRINT("Duplicate output handle\n");
        Status = NtDuplicateObject (NtCurrentProcess(), Ppb->hStdOutput,
                                    hProcess, &Ppb->hStdOutput,
                                    0, TRUE, DUPLICATE_SAME_ACCESS);
        . . . . . .
    }
}

. . . . . .

/* Initialize some other fields in the PPB */
if (lpStartupInfo)
   {
      Ppb->dwFlags = lpStartupInfo->dwFlags;
      if (Ppb->dwFlags & STARTF_USESHOWWINDOW)
   {
      Ppb->wShowWindow = lpStartupInfo->wShowWindow;
   }
   else
   {
      Ppb->wShowWindow = SW_SHOWDEFAULT;
   }
   Ppb->dwX = lpStartupInfo->dwX;
   Ppb->dwY = lpStartupInfo->dwY;
   Ppb->dwXSize = lpStartupInfo->dwXSize;
   Ppb->dwYSize = lpStartupInfo->dwYSize;
   Ppb->dwFillAttribute = lpStartupInfo->dwFillAttribute;
}
else
{
      Ppb->Flags = 0;
}

/* Create Process Environment Block */
DPRINT("Creating peb\n");

KlInitPeb(hProcess, Ppb, &ImageBaseAddress, Sii.Subsystem);

RtlDestroyProcessParameters (Ppb);
```

前面已经准备下了一个数据结构 PriorityClass，这里的 NtSetInformationProcess()就把与调度优先级有关的信息设置到目标进程对象中去。

此外，前面已经创建了 PEB，并对其进行了最基本的设置，但是 PEB 是个不小的数据结构。特别地，PEB 中还有个指针 ProcessParameters，指向一个“进程参数块”PPB、即 RTL_USER_PROCESS_PARAMETERS 数据结构。这 PPB 也是在用户空间的，虽然是个独立存在的数据结构，逻辑上却可以看作是 PEB 的一部分。这里先通过 RtlCreateProcessParameters()根据有关的信息在(创建者的用户空间)为新建进程创建起一个 PPB 的副本，经修改补充以后再通过 KlInitPeb()写入新建进程的用户空间、并与其 PEB 建立起连接。

下面陆续有一些对 NtDuplicateObject()的调用。这就是用于跨进程复制 Handle 的系统调用，目的在于让新建进程继承其父进程的一些已打开的对象。不过这里只是复制了当前目录、标准输入、标准输出这三个对象，加上对标准出错信息通道的有条件复制(这里的代码中已略去)，并且是从当前进程、而不是从父进程复制的，这样处理之正确与否待考。

还有件事，就是向 csrss 报告新进程的创建，相当于“报户口”吧，这是由 CsrClientCallServer()完成的。这些操作的结果有些也要记录在 PPB 中，所以对 KlInitPeb()的调用是在完成了所有这些操作之后。

至此，新进程的创建已经完成了。

在 Windows 中，进程并非一个实际受调度运行的实体，也没有执行的上下文，而只是一个让线程赖以存身的框架。所以光创建进程而不创建其第一个线程、即其主线程，是没有意义的。

所以第三阶段就是创建其初始线程。

[CreateProcessW()]

```
    /*
     * Create the thread for the kernel
     */
    . . . . . .
    hThread =   KlCreateFirstThread(hProcess, lpThreadAttributes, &Sii,
                                    (PVOID)((ULONG_PTR)ImageBaseAddress + Sii.EntryPoint),
                                    dwCreationFlags, &lpProcessInformation->dwThreadId);
    . . . . . .

    lpProcessInformation->hProcess = hProcess;
    lpProcessInformation->hThread = hThread;

    return TRUE;
}
```

KlCreateFirstThread()是 kernel32.dll 中的一个内部(未导出)函数，这实际上是个不小的操

作，而且“技术含量”比较高，从某种意义上说这才是 CreateProcessW()的核心。这里传下去的参数中有两个是特别值得一提的。一个是 SECTION_IMAGE_INFORMATION 数据结构 Sii，这个数据结构中的信息来自代表着目标映像文件的共享内存区对象，是前面通过 ZwQuerySection()获取的，里面有些信息实际上来自目标映像文件。例如，Sii.EntryPoint 就是程序入口在映像中的位移；而 Sii.Subsystem 说明了目标映像属于哪一个子系统，是 GUI 还是 CUI。读者如果用微软的 Visual Studio 开发过软件，就一定知道在创建一个项目(Project)时首先就要确定目标软件是面向图形界面的还是面向控制台的。另一个参数(ImageBaseAddress + Sii.EntryPoint)则是目标映像映射到用户空间以后的程序入口地址。

[CreateProcessW() > KlCreateFirstThread()]

```
HANDLE STDCALL
KlCreateFirstThread(HANDLE ProcessHandle,
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    PSECTION_IMAGE_INFORMATION Sii,
                    LPTHREAD_START_ROUTINE lpStartAddress,
                    DWORD dwCreationFlags,
                    LPDWORD lpThreadId)
{
  OBJECT_ATTRIBUTES oaThreadAttribs;
  CLIENT_ID cidClientId;
  PVOID pTrueStartAddress;
  NTSTATUS nErrCode;
  HANDLE hThread;

  /* convert the thread attributes */
  RtlRosR32AttribsToNativeAttribs(&oaThreadAttribs, lpThreadAttributes);

  /* native image */
  if(Sii->Subsystem != IMAGE_SUBSYSTEM_NATIVE)
    pTrueStartAddress = (PVOID)BaseProcessStart;       /* Win32 image */
  else
    pTrueStartAddress = (PVOID)RtlBaseProcessStartRoutine;

  . . . . . .

  /* create the first thread */
  nErrCode = RtlRosCreateUserThreadVa(ProcessHandle, &oaThreadAttribs,
                    dwCreationFlags & CREATE_SUSPENDED, 0,
                    &(Sii->StackReserve), &(Sii->StackCommit),
                    pTrueStartAddress, &hThread, &cidClientId,
                    2, (ULONG_PTR)lpStartAddress, (ULONG_PTR)PEB_BASE);
 . . . . . .
  if(lpThreadId) *lpThreadId = (DWORD)cidClientId.UniqueThread;
```

```
  return hThread;
}
```

显然，实际的线程创建是由 RtlRosCreateUserThreadVa()完成的。这里特别要注意的是指针 pTrueStartAddress 的设置。对于 GUI 或 CUI 的 Windows 应用，这个指针设置成指向 BaseProcessStart() 。这是 Kernel32.dll 中的一个内部函数。再注意调用 RtlRosCreateUserThreadVa()时的参数中有两个指针：一个是 pTrueStartAddress；另一个是 lpStartAddress，那就是作为参数传下来的目标映像程序入口。后面读者就会看到，当新建的线程受调度运行而“返回”到用户空间时，首先进入的是 pTrueStartAddress 所指的 BaseProcessStart()，然后再由 BaseProcessStart()把 lpStartAddress 作为函数指针加以调用。之所以如此，是为了把目标映像的执行置于“结构化出错保护”即 SHE 的保护之下。按理说，目标映像的程序入口才是“真正”的起始地址；但是说 pTrueStartAddress 是“真正”的起始地址也没有错，因为这确实是新建线程进入用户空间时的起始地址。或许应该说，前者是逻辑意义上的起始地址，而后者则是物理意义上的起始地址。

看了漫谈十中关于 BaseProcessStart()的叙述(来自“Internals”一书)、以及与此有关的代码后，我们兼容内核研发团队的胡晨展先生提出了质疑：把 pTrueStartAddress 设置成指向 BaseProcessStart()的是当前线程，所以是当前进程用户空间的 BaseProcessStart()，在当前进程用户空间的 kernel32.dll 映像中。可是，这里实际需要的却是新建线程所在用户空间中的 BaseProcessStart()，这应该在新建线程所在用户空间的 kernel32.dll 映像中，然而此刻新建线程的 kernel32.dll 映像尚未映射。所以，这样的安排只有在 kernel32.dll 的装入地址不变、即不管在哪一个进程中都是装入到相同的地址的条件下才能成立。为此，他又在 Windows 上做了实验，故意把 kernel32.dll 期望装入的地址先占了，结果是 Windows 显示出错信息说不能装入 kernel32.dll、所以不能运行目标映像。他的实验结果说明这样的安排在实际效果上还是可行的。但是，如果 Windows 的代码中也是这样处理的话(按“Internals”书中所说确实就是这样)，这说明了 Windows 的程序设计至少在这个问题上存在着逻辑混乱。

言归正传，我们往下看 RtlRosCreateUserThreadVa()的代码。

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

```
NTSTATUS CDECL
RtlRosCreateUserThreadVa(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,
                         ...)
{
  va_list vaArgs;
  NTSTATUS nErrCode;
```

```
  va_start(vaArgs, ParameterCount);

  /*  FIXME: this code makes several non-portable assumptions:
  all parameters are passed on the stack, the stack is a contiguous array of cells as
  large as an ULONG_PTR, the stack grows downwards. This happens to work on
  the Intel x86, but is likely to bomb horribly on most other platforms  */

  nErrCode = RtlRosCreateUserThread(ProcessHandle, ObjectAttributes, CreateSuspended,
                                    StackZeroBits, StackReserve, StackCommit,
                                    StartAddress, ThreadHandle, ClientId,
                                    ParameterCount, (ULONG_PTR *)vaArgs);
  va_end(vaArgs);
  return nErrCode;
}
```

这个函数的调用参数表是可变长度的，参数 ParameterCount 表明后面还有几个参数。前后比对一下，就可以知道后面还用两个参数，一个是 lpStartAddress，另一个是常数 PEB_BASE，即新建进程的 PEB 指针。

注意这里的参数 StartAddress 其实是前面代码中的 pTrueStartAddress，指向 BaseProcessStart()。而前面的 lpStartAddress、即目标映像的程序入口，则在 ParameterCount 后面，在代码中不能直接看到了。

所以，这个函数的作用只是把可变长度的参数表转换成固定长度的参数表。不过 RtlRosCreateUserThreadVa()的参数表长度虽然是固定的，但是最后的指针数组 vaArgs 却是大小可变的，所以实质上仍是一样。我们继续往下看：

```
[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;
}
```

这里要做的是三件大事，前两件都是为最后对 NtCreateThread()的调用作准备。

首先通过 RtlRosCreateStack()建立新线程的用户空间堆栈。在所用到的几个参数中，StackZeroBits 表示堆栈起始地址中前导 0 的位数，所以这是选择堆栈位置的一个准则。另一个参数 StackReserve 表示堆栈大小的上限，而 StackCommit 则是此刻需要加以映射的区间大小。二者相等就是固定大小的堆栈，否则就是可变大小的堆栈。

下面的 RtlRosInitializeContext()可以说是许多奥妙所在，正是这个函数伪造了新线程的“上下文(Context)”，从而决定了当新建线程被调度运行并进入其用户空间运行时所走的路线。可以理解，一个逻辑意义上的、完整的“上下文”、应该由两部分构成，一部分在新建线程的系统空间堆栈上，另一部分在它的用户空间堆栈上。前者先建立在这里作为缓冲区使用的数据结构 ctxInitialContext 上，然后将其作为参数传递给 NtCreateThread()，使其被复制到新建线程的系统空间堆栈上。至于后者，则取决于刚才建立的用户空间堆栈在什么地方。上下文的这两个部分都是由 RtlRosInitializeContext()创建的。

```
[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
   > RtlRosCreateUserThread() > 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);
  . . . . . .

  /* initialize the context */
  Context->ContextFlags = CONTEXT_FULL;
  Context->FloatSave.ControlWord = FLOAT_SAVE_CONTROL;
  Context->FloatSave.StatusWord = FLOAT_SAVE_STATUS;
  Context->FloatSave.TagWord = FLOAT_SAVE_TAG;
  Context->FloatSave.DataSelector = FLOAT_SAVE_DATA;
  Context->Eip = (ULONG_PTR)StartAddress;
  Context->SegGs = USER_DS;
  Context->SegFs = TEB_SELECTOR;
  Context->SegEs = USER_DS;
  Context->SegDs = USER_DS;
  Context->SegCs = USER_CS;
  Context->SegSs = USER_DS;
  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);
}
```

在 CONTEXT 数据结构中准备的是系统空间的那一部分上下文。注意这里的参数 StartAddress 就是前面代码中的 pTrueStartAddress，即指向用户空间的 BaseProcessStart()。而上下文中的 Esp 则指向用户空间堆栈的地址顶点往下减去参数数组的大小(这里是 2，实际是 8 个字节)外加一个指针的地方。目标映像的实际入口地址 lpStartAddress 就在这些参数中，而且是其中地址较低的那一个。这个参数数组被写入用户空间堆栈(地址区间)的顶部，在这些参数下面则是一个伪造的“返回地址”s_pRetAddr。

这样，当目标线程从系统空间“返回”用户空间时，寄存器 Eip 将指向 BaseProcessStart()的起点，而 Esp 将指向其用户空间堆栈上那个伪造的“返回地址”s_pRetAddr，在“返回地址”上方则是参数中的 lpStartAddress。如此的安排制造出一个假象，似乎是：

- 用户空间的某个函数调用了 BaseProcessStart()，调用点的返回地址为 s_pRetAddr。
- 但是在执行 BaseProcessStart()的第一条指令时就发生了异常，从而进入了内核。

- 经过异常处理以后又回来了，回来就要重新执行 BaseProcessStart()的第一条指令。
- 堆栈上的参数就是调用 BaseProcessStart()时的参数，其中参数 1 就是 lpStartAddress。

后面读者就会看到，BaseProcessStart()其实是不返回的，所以返回地址 s_pRetAddr 是虚设的，只是占个位置而已。从上面的代码中可以看到，这个地址设置为 0xDEADBEEF。显然，这个地址属于系统空间(大于 0x80000000)，从用户空间访问这个地址会引起异常，所以万一真的从 BaseProcessStart()返回也因此而受到保护。

所以，新建的主线程在其投入运行的过程中将要涉及四个起始地址：

1. 当新建线程受调度运行时，一开始是在内核中运行，所以有个内核中的起始地址，这个地址我们尚未见到。
2. 新建线程需要返回到用户空间运行，但是用户空间的 DLL 动态连接尚未完成，所以先要在用户空间执行一个 APC 函数 LdrInitializeThunk()。
3. 完成了 DLL 的动态连接以后，新建线程的执行又回到内核中，继续其奔向用户空间目标映像的征程。当新建线程通过 reti 指令回到用户空间的时候，当然要有个开始执行的起始地址，这就是它的上下文中 Eip 所指的地址，实际上是 BaseProcessStart()。这个地址作为“断点”安插在新建线程的系统空间堆栈上。
4. 之所以需要 BaseProcessStart()，是因为要把整个目标映像的执行置于“结构化出错处理”即 SEH 的保护之下，而真正意义上的起始地址在目标映像中的位移是由映像头部的 EntryPoint 字段提供的，映射到用户空间后的地址为(ImageBaseAddress + Sii.EntryPoint)。这个地址已经被安排在新建线程的用户空间堆栈上，使其正好成为 BaseProcessStart()的调用参数。

万事俱备，只欠东风。现在只剩下主线程的创建了。

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

NTSTATUS STDCALL
NtCreateThread(OUT PHANDLE ThreadHandle,
               IN ACCESS_MASK DesiredAccess,
               IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
               IN HANDLE ProcessHandle,
               OUT PCLIENT_ID ClientId,
               IN PCONTEXT ThreadContext,
               IN PINITIAL_TEB InitialTeb,
               IN BOOLEAN CreateSuspended)
{
  . . . . . .

  if(PreviousMode != KernelMode)
  {
    _SEH_TRY    _SEH_END;
    . . . . . .
  }
```

```
Status = ObReferenceObjectByHandle(ProcessHandle, PROCESS_CREATE_THREAD,
                                   PsProcessType, PreviousMode, (PVOID*)&Process, NULL);
. . . . . .
Status = PsInitializeThread(Process, &Thread, ObjectAttributes, PreviousMode, FALSE);
. . . . . .

/* create a client id handle */
Status = PsCreateCidHandle(Thread, PsThreadType, &Thread->Cid.UniqueThread);
. . . . . .
Status = KiArchInitThreadWithContext(&Thread->Tcb, ThreadContext);
. . . . . .
Status = PsCreateTeb(ProcessHandle, &TebBase, Thread, InitialTeb);
. . . . . .
Thread->Tcb.Teb = TebBase;
Thread->StartAddress = NULL;

/* Maybe send a message to the process's debugger */
DbgkCreateThread((PVOID)ThreadContext->Eip);

/* First, force the thread to be non-alertable for user-mode alerts. */
Thread->Tcb.Alertable = FALSE;

/*
 * If the thread is to be created suspended then queue an APC to
 * do the suspend before we run any userspace code.
 */
if (CreateSuspended)
{
    PsSuspendThread(Thread, NULL);
}

/* Queue an APC to the thread that will execute the ntdll startup routine. */
LdrInitApc = ExAllocatePool(NonPagedPool, sizeof(KAPC));
KeInitializeApc(LdrInitApc, &Thread->Tcb, OriginalApcEnvironment,
                LdrInitApcKernelRoutine, LdrInitApcRundownRoutine,
                LdrpGetSystemDllEntryPoint(),UserMode, NULL);
KeInsertQueueApc(LdrInitApc, NULL, NULL, IO_NO_INCREMENT);

/*
 * The thread is non-alertable, so the APC we added did not set UserApcPending to TRUE.
 * We must do this manually. Do NOT attempt to set the Thread to Alertable before the call,
 * doing so is a blatant and erronous hack.
 */
Thread->Tcb.ApcState.UserApcPending = TRUE;
```

```
    Thread->Tcb.Alerted[KernelMode] = TRUE;

    oldIrql = KeAcquireDispatcherDatabaseLock ();
    PsUnblockThread(Thread, NULL, 0);
    KeReleaseDispatcherDatabaseLock(oldIrql);

    Status = ObInsertObject((PVOID)Thread, NULL, DesiredAccess, 0, NULL, &hThread);
    . . . . . .
    return Status;
}
```

这里我们只看其中的 KiArchInitThreadWithContext()，其余的都留给读者了。里面除 DbgkCreateThread()是向内核 Debug 端口发送调试信息外，别的函数读者都已经耳熟能详了。不过有个事情倒是要提一下，线程的系统空间堆栈是在由 PsInitializeThread()调用的 KeInitializeThread()内部为其分配存储空间的。分配了空间以后，Thread->KernelStack 指向其系统空间堆栈。

KiArchInitThreadWithContext()其实是个宏定义，因具体 CPU 类型的不同而定义成不同的函数。对于 386 结构的 CPU 定义成 Ke386InitThreadWithContext()：

```
#define KiArchInitThreadWithContext    Ke386InitThreadWithContext

[CreateProcessW() > KlCreateFirstThread() > RtlRosCreateUserThreadVa()
    > RtlRosCreateUserThread() > NtCreateThread() > Ke386InitThreadWithContext()]

NTSTATUS
Ke386InitThreadWithContext(PKTHREAD Thread, PCONTEXT Context)
{
    PULONG KernelStack;
    ULONG InitSize;
    PKTRAP_FRAME TrapFrame;
    PFX_SAVE_AREA FxSaveArea;

    /* Setup a stack frame for exit from the task switching routine */
    InitSize = 6 * sizeof(DWORD) + sizeof(DWORD) + 6 * sizeof(DWORD) +
                + sizeof(KTRAP_FRAME) + sizeof (FX_SAVE_AREA);
    KernelStack = (PULONG)((char*)Thread->KernelStack - InitSize);

    /* Set up the initial frame for the return from the dispatcher. */
    KernelStack[0] = (ULONG)Thread->InitialStack - sizeof(FX_SAVE_AREA); /* TSS->Esp0 */
    KernelStack[1] = 0;          /* EDI */
    KernelStack[2] = 0;          /* ESI */
    KernelStack[3] = 0;          /* EBX */
    KernelStack[4] = 0;          /* EBP */
    KernelStack[5] = (ULONG)&PsBeginThreadWithContextInternal;     /* EIP */
```

```
/* Save the context flags. */
KernelStack[6] = Context->ContextFlags;

/* Set up the initial values of the debugging registers. */
KernelStack[7] = Context->Dr0;
KernelStack[8] = Context->Dr1;
KernelStack[9] = Context->Dr2;
KernelStack[10] = Context->Dr3;
KernelStack[11] = Context->Dr6;
KernelStack[12] = Context->Dr7;

/* Set up a trap frame from the context. */
TrapFrame = (PKTRAP_FRAME)(&KernelStack[13]);
TrapFrame->DebugEbp = (PVOID)Context->Ebp;
TrapFrame->DebugEip = (PVOID)Context->Eip;
TrapFrame->DebugArgMark = 0;
TrapFrame->DebugPointer = 0;
TrapFrame->TempCs = 0;
TrapFrame->TempEip = 0;
TrapFrame->Gs = (USHORT)Context->SegGs;
TrapFrame->Es = (USHORT)Context->SegEs;
TrapFrame->Ds = (USHORT)Context->SegDs;
TrapFrame->Edx = Context->Edx;
TrapFrame->Ecx = Context->Ecx;
TrapFrame->Eax = Context->Eax;
TrapFrame->PreviousMode = UserMode;
TrapFrame->ExceptionList = (PVOID)0xFFFFFFFF;
TrapFrame->Fs = TEB_SELECTOR;
TrapFrame->Edi = Context->Edi;
TrapFrame->Esi = Context->Esi;
TrapFrame->Ebx = Context->Ebx;
TrapFrame->Ebp = Context->Ebp;
TrapFrame->ErrorCode = 0;
TrapFrame->Cs = Context->SegCs;
TrapFrame->Eip = Context->Eip;
TrapFrame->Eflags = Context->EFlags | X86_EFLAGS_IF;
TrapFrame->Eflags &= ~(X86_EFLAGS_VM | X86_EFLAGS_NT | X86_EFLAGS_IOPL);
TrapFrame->Esp = Context->Esp;
TrapFrame->Ss = (USHORT)Context->SegSs;
/* FIXME: Should check for a v86 mode context here. */

/* Set up the initial floating point state. */
/* FIXME: Do we have to zero the FxSaveArea or is it already? */
```

```
  FxSaveArea = (PFX_SAVE_AREA)
                        ( (ULONG_PTR)KernelStack + InitSize - sizeof(FX_SAVE_AREA));
  if (KiContextToFxSaveArea(FxSaveArea, Context))
    {
      Thread->NpxState = NPX_STATE_VALID;
    }
  else
    {
      Thread->NpxState = NPX_STATE_INVALID;
    }

  /* Save back the new value of the kernel stack. */
  Thread->KernelStack = (PVOID)KernelStack;

  return(STATUS_SUCCESS);
}
```

这里，FX_SAVE_AREA 数据结构用于与浮点处理器有关的信息。KTRAP_FRAME 数据结构就是在发生异常、中断时所保留的“现场”，这个现场当然是在线程的系统空间堆栈上。而前面所准备的 CONTEXT 数据结构，则现在就要将其内容转移到 KTRAP_FRAME 数据结构中。此外，还要安排好新建线程在系统空间的那部分上下文，这也在系统堆栈上。

前面曾提到，Thread->KernelStack 指向线程的系统空间堆栈。由于此刻的系统空间堆栈是空的，所以 Thread->KernelStack 指向其区间的顶点(地址最高)，逻辑上则是堆栈的底部。需要伪造的原始堆栈(内容)的大小为 InitSize，里面包括一个 FX_SAVE_AREA 数据结构、一个 KTRAP_FRAME 数据结构，以及一个数组 KernelStack[]。如果从堆栈的地址顶点往下数，则首先是 FX_SAVE_AREA 数据结构，然后是 KTRAP_FRAME 数据结构，地址最低的是 KernelStack[]。注意堆栈是由上向下伸展的，所以最先恢复/使用的是 KernelStack[]。

所以，KernelStack[]中是系统空间的那部分上下文，这是线程在受调度运行时最先恢用到的。其中的 Eip 指向内核函数 PsBeginThreadWithContextInternal()，所以这是新建线程最早的入口。从代码中可以看出，这部分上下文的大小是从 KernelStack[0]到 KernelStack[12]共 13 个 32 位整数。

在这上面(按地址高低)是 KTRAP_FRAME 数据结构 TrapFrame。虽然也是在系统空间堆栈上，这实际上却是用户空间上下文的一部分，这是在新建线程“返回”用户空间时才予以恢复的，其大部分内容已经在前面的 CONTEXT 数据结构中准备好，有些则是固定的。

再往上的 FX_SAVE_AREA 数据结构，我们就不关心了。

所以，前面所说尚未见到的第一个起始地址就是 PsBeginThreadWithContextInternal()。

[注：不同版本的 ReactOS 在这一部分代码上有明显不同，例如 0.2.9 版中的第一个起始地址是 KiThreadStartup()，而且 Ke386InitThreadWithContext()的代码也有较大不同，但是基本的原理和过程还是一样的]。

当前线程在 NtCreateThread()中执行完 PsUnblockThread()，把新建线程的数据结构插入调度队列之后，新建线程就可以受调度运行了。当新建线程受调度运行时，首先进入的就是 PsBeginThreadWithContextInternal()。

```
__declspec(naked)
```

```
VOID PsBeginThreadWithContextInternal(VOID)
{
   /* This isn't really a function, we are called as the return address of a context switch */
   /* Do the necessary prolog before the context switch */
   __asm
   {
      call    PiBeforeBeginThread

       /* Load the context flags. */
      pop    ebx

      /* Load the debugging registers */
      test    ebx, (CONTEXT_DEBUG_REGISTERS & ~CONTEXT_i386)
      jz       L1
      pop       eax       __asm       mov    dr0, eax
      pop       eax       _asm       mov    dr1, eax
      pop       eax       __asm       mov    dr2, eax
      pop       eax       __asm       mov    dr3, eax
      pop       eax       __asm       mov    dr6, eax
      pop       eax       __asm       mov    dr7, eax
      jmp       L3
L1:
      add       esp, 24
L3:
      /* Load the floating point registers */
      mov        eax, HardwareMathSupport
      test         eax,eax
      jz       L2
      test       ebx, (CONTEXT_FLOATING_POINT & ~CONTEXT_i386)
      jz       L2
      frstor       [esp]
L2:
      add       esp, 112

      /* Load the rest of the thread's user mode context. */
      mov       eax, 0
      jmp       KeReturnFromSystemCallWithHook
   }
}
```

这里的 PiBeforeBeginThread()把 CPU 的运行级别降低到 PASSIVE_LEVEL。我们在这里所关心的是最后的 jmp 指令，这是跳转到了 KeReturnFromSystemCallWithHook()：

```
__declspec(naked)
```

```
void KeReturnFromSystemCallWithHook()
{
  __asm
  {
    /* Call the post system call hook and deliver any pending APCs */
    push   esp
    push   eax
    call   KiAfterSystemCallHook
    add    esp, 8

    // TMN: Added, to be able to separate this into different functions
    jmp    KeReturnFromSystemCall
  }
}
```

我们在这里关心的是最后向 KeReturnFromSystemCall 的跳转，那已经是在从中断/异常/系统调用返回的那段汇编代码中了。正是在那里，将会检查是否有 APC 函数需要执行，并且在“正常”返回用户空间之前偏离航向，先进入用户空间执行 APC 函数 LdrInitializeThunk()的。执行完 LdrInitializeThunk()以后，CPU 又回到内核，并继续其“正常”返回用户空间的航程。

如前所述，当新建线程受调度运行而“返回”到用户空间时，首先进入的是 BaseProcessStart()，而且用户空间堆栈的安排使得逻辑上的起始地址、即目标映像的程序入口地址成为 BaseProcessStart()的第一个参数，这就是下面的参数 lpStartAddress：

```
VOID STDCALL
BaseProcessStart(LPTHREAD_START_ROUTINE lpStartAddress, DWORD lpParameter)
{
    UINT uExitCode = 0;
    DPRINT("BaseProcessStart(..) - setting up exception frame.\n");
    __try1(_except_handler)
    {
        uExitCode = (lpStartAddress)((PVOID)lpParameter);
    } __except1

    ExitProcess(uExitCode);
}
```

可见，之所以需要 BaseProcessStart()，是要把整个目标映像的执行置于 Windows 的“结构化异常处理”即 SHE 的控制之下，不过那已经不在本文的范围之内。

最后，注意这里的 ExitProcess(uExitCode)，这意味着 BaseProcessStart()是不返回的。下面是 ExitProcess 的代码，留给读者自己阅读：

[BaseProcessStart() > ExitProcess()]

```
VOID STDCALL    ExitProcess(UINT uExitCode)
{
   . . . . . .

   /* unload all dll's */
   LdrShutdownProcess ();
   /* notify csrss of process termination */
   CsrRequest.Type = CSRSS_TERMINATE_PROCESS;
   Status = CsrClientCallServer(&CsrRequest, &CsrReply, sizeof(CSRSS_API_REQUEST),
                                                        sizeof(CSRSS_API_REPLY));
   . . . . . .
   NtTerminateProcess (NtCurrentProcess (),uExitCode);

   /* should never get here */
   ASSERT(0);
   while(1);
}
```