漫谈兼容内核之十三：
关于“进程挂靠”

毛德操

上一篇漫谈在介绍 APC 机制时提到：线程在 Windows 内核中运行时有时候需要暂时“挂靠(Attach)”到别的进程的用户空间，即暂时切换到另一个进程的用户空间。这称为“进程挂靠”，因为用户空间是一个进程最主要的特征。

显然，要是当前线程的操作与用户空间无关、不需要访问用户空间，那么当时的用户空间到底是谁的用户空间根本就无关紧要，所以这必定发生在与用户空间有关的操作中。

一般而言，如果线程 T 属于进程 P，那么当这个线程在内核中运行时的用户空间应该就是进程 P 的用户空间，它也没有必要访问到别的进程的用户空间去。可是，Windows 内核允许一些跨进程的操作，特别是跨用户空间的操作，所以有时候就需要把当时的用户空间切换到别的进程的用户空间，或者说挂靠到别的进程。在 Windows 中，一个进程实际上只是意味着一个用户(地址)空间，说一个线程属于某个进程的意思是它使用的是某个特定的用户空间，系统空间则是由所有线程共用的。那么“某个特定的用户空间”是什么意思呢？实质上就是一个具体的页面映射方案，或者一套具体的映射目录和页面表，以及相关的其它数据结构。而所谓“切换到某个进程的用户空间”，就是把这套具体的映射目录和页面表装入 CPU 中的页面映射机构，使其真正发生作用。当然，在完成了有关的操作以后还要回到原来的用户空间，否则就无法从内核“返回”自己的用户空间了。

然而究竟什么时候需要用到进程挂靠呢？最好还是通过一个实例来加以说明。

前几篇漫谈中说到，在启动一个 PE 格式的 EXE 映像运行时先要创建一个进程，然后把目标 EXE 映像和 ntdll.dll 的映像映射到新建进程的用户空间，并且在映射后的 ntdll.dll 映像中找到 LdrInitializeThunk()等函数的入口。在这个过程中，当前线程属于作为创建者的那个进程，或“父进程”，而其部分操作的对象则在新建进程、即“子进程”的用户空间。所以此时就用到了进程挂靠，使当前线程挂靠到新建进程的用户空间。下面我们通过 LdrpMapSystemDll()的代码来说明为什么有进程挂靠、以及怎样实现进程挂靠。

在创建进程的过程中要调用到一个函数 LdrpMapSystemDll()，其作用是把“系统 DLL”、即 ntdll.dll 映射到新建进程的用户空间，并从中获取几个重要函数的入口。当然，这是个内核函数，是在系统空间运行的。

```
NTSTATUS LdrpMapSystemDll(HANDLE ProcessHandle, PVOID* LdrStartupAddr)
{
    CHAR    BlockBuffer [1024];
    . . . . . .
    UNICODE_STRING DllPathname =
                    ROS_STRING_INITIALIZER(L"\\SystemRoot\\system32\\ntdll.dll");
    . . . . . .

    /*
     * Locate and open NTDLL to determine ImageBase
     * and LdrStartup
     */
```

```
InitializeObjectAttributes(&FileObjectAttributes, &DllPathname, 0, NULL, NULL);
DPRINT("Opening NTDLL\n");
Status = ZwOpenFile(&FileHandle, FILE_READ_ACCESS, &FileObjectAttributes,
                    &Iosb, FILE_SHARE_READ, FILE_SYNCHRONOUS_IO_NONALERT);
. . . . . .
Status = ZwReadFile(FileHandle, 0, 0, 0, &Iosb, BlockBuffer, sizeof(BlockBuffer), 0, 0);
. . . . . .
. . . . . .
DosHeader = (PIMAGE_DOS_HEADER) BlockBuffer;
NTHeaders = (PIMAGE_NT_HEADERS) (BlockBuffer + DosHeader->e_lfanew);
. . . . . .
ImageBase = NTHeaders->OptionalHeader.ImageBase;
ImageSize = NTHeaders->OptionalHeader.SizeOfImage;

/*
 * Create a section for NTDLL
 */
DPRINT("Creating section\n");
Status = ZwCreateSection(&NTDllSectionHandle, SECTION_ALL_ACCESS, NULL,
                    NULL, PAGE_READWRITE, SEC_IMAGE | SEC_COMMIT, FileHandle);
. . . . . .
ZwClose(FileHandle);

/*
 * Map the NTDLL into the process
 */
ViewSize = 0;
ImageBase = 0;
Status = ZwMapViewOfSection(NTDllSectionHandle, ProcessHandle,
                                    (PVOID*)&ImageBase, 0, ViewSize, NULL,
                                    &ViewSize, 0, MEM_COMMIT, PAGE_READWRITE);
. . . . . .
. . . . . .

CurrentProcess = PsGetCurrentProcess();
if (Process != CurrentProcess)
   {
      DPRINT("Attaching to Process\n");
      KeAttachProcess(&Process->Pcb);
   }

/*
 * retrieve ntdll's startup address
 */
```

```
    if (SystemDllEntryPoint == NULL)
      {
        RtlInitAnsiString (&ProcedureName,
                                        "LdrInitializeThunk");
        Status = LdrGetProcedureAddress ((PVOID)ImageBase,
                                                &ProcedureName,
                                                0,
                                                &SystemDllEntryPoint);
        . . . . . .
        *LdrStartupAddr = SystemDllEntryPoint;
      }
    . . . . . .
    . . . . . .
    if (Process != CurrentProcess)
      {
        KeDetachProcess();
      }
    ObDereferenceObject(Process);
    ZwClose(NTDllSectionHandle);
    return(STATUS_SUCCESS);
}
```

先看一下大致的流程：

- 通过 InitializeObjectAttributes()设置好一个 OBJECT_ATTRIBUTES 数据结构 FileObjectAttributes；然后用这个数据结构作为参数之一，通过系统调用 ZwOpenFile()打开目标文件 ntdll.dll。之所以如此，是因为 ZwOpenFile()并不接受文件名作为参数，而必须把文件名放在 OBJECT_ATTRIBUTES 数据结构中。当然，这个数据结构中还有别的信息。
- 通过 ZwReadFile()读入目标文件的开头 1K 字节，目的在于获取其 DosHeader 和 NTHeaders，进而获取其 NTHeaders->OptionalHeader 中的 ImageBase 和 SizeOfImage 两项信息，前者是映像在文件中的起点，后者是映像的大小。
- 通过 ZwCreateSection()为目标文件建立(并打开)一个 Section 对象。从逻辑的意义上，这个 Section 对象就与目标文件的内容划上了等号。
- 至此，目标文件已经可以关闭，因为不再需要通过文件读写等常规的文件操作访问这个文件了。
- 通过 ZwMapViewOfSection()将已建立的 Section、即目标文件的内容映射到目标进程的用户空间。
- 通过 KeAttachProcess()将当前线程挂靠到目标进程。
- 通过 LdrGetProcedureAddress()从已经映射到目标进程用户空间的映像中获取函数 LdrInitializeThunk()的入口地址。
- 再通过 LdrGetProcedureAddress()获取若干其它函数的入口地址。
- 通过 KeDetachProcess()撤销挂靠，回到当前线程所属的进程。
- 关闭所创建的 Section 对象。

首先要说明，函数名以 Zw 开头的函数实际上就是以 Nt 开头的对应系统调用。以打开

文件为例，在用户空间调用时要用 NtOpenFile()，在内核中调用则用 ZwOpenFile()。

显然，这个流程中的进程挂靠、即 KeAttachProcess()和 KeDetachProcess()、是因为要执行 LdrGetProcedureAddress()而产生的需求。对此我们很自然地就会有两个问题：首先，为什么 LdrGetProcedureAddress()需要进程挂靠？其次，既然 LdrGetProcedureAddress()需要，那为什么 ZwMapViewOfSection()倒又不需要？二者不是都涉及目标进程的用户空间吗?

要回答这两个问题，就得近一步深入到这两个函数的代码中。

如前所述，系统调用 NtCreateSection()在内核中创建一个 Section 对象，并使这个对象与一个(已经打开的)目标文件挂上勾，此后就可以通过另一个系统调用 NtMapViewOfSection()将目标文件的部分或全部内容映射到某个用户空间(Section 可以为多个进程共享，分别映射到不同空间的相同或不同地址上)。

下面先看 NtMapViewOfSection()。

[LdrpMapSystemDll() > NtMapViewOfSection()]

```
NTSTATUS STDCALL
NtMapViewOfSection(IN HANDLE SectionHandle,
                   IN HANDLE ProcessHandle,
                   IN OUT PVOID* BaseAddress    OPTIONAL,
                   IN ULONG ZeroBits    OPTIONAL,
                   IN ULONG CommitSize,
                   IN OUT PLARGE_INTEGER SectionOffset    OPTIONAL,
                   IN OUT PULONG ViewSize,
                   IN SECTION_INHERIT InheritDisposition,
                   IN ULONG AllocationType    OPTIONAL,
                   IN ULONG Protect)
{
    PVOID SafeBaseAddress;
    LARGE_INTEGER SafeSectionOffset;
    ULONG SafeViewSize;
    PSECTION_OBJECT Section;
    PEPROCESS Process;
    KPROCESSOR_MODE PreviousMode;
    PMADDRESS_SPACE AddressSpace;
    NTSTATUS Status = STATUS_SUCCESS;

    PreviousMode = ExGetPreviousMode();

    if(PreviousMode != KernelMode)
    {
      ......
    }
    else
    {
      SafeBaseAddress = (BaseAddress != NULL ? *BaseAddress : NULL);
```

```
        SafeSectionOffset.QuadPart = (SectionOffset != NULL ? SectionOffset->QuadPart : 0);
        SafeViewSize = (ViewSize != NULL ? *ViewSize : 0);
    }

    . . . . . .
    AddressSpace = &Process->AddressSpace;
    . . . . . .

    Status = MmMapViewOfSection(Section,
                                Process,
                                (BaseAddress != NULL ? &SafeBaseAddress : NULL),
                                ZeroBits,
                                CommitSize,
                                (SectionOffset != NULL ? &SafeSectionOffset : NULL),
                                (ViewSize != NULL ? &SafeViewSize : NULL),
                                InheritDisposition,
                                AllocationType,
                                Protect);

    . . . . . .

    return(Status);
}
```

参数 SectionHandle 代表着一个 Section 对象，ProcessHandle 则代表着一个用户空间，BaseAddress 是要求装入的地址，而 SectionOffset 是目标文件中的起点。还有个参数 Protect 是对映射后的内存区间(而不是目标文件)的访问保护，在这里是 PAGE_READWRITE。

显然，实际的操作是由 MmMapViewOfSection()完成的，函数名中的前缀 Mm 表示这个函数属于内存管理。

[LdrpMapSystemDll() > NtMapViewOfSection() > MmMapViewOfSection()]

```
NTSTATUS STDCALL
MmMapViewOfSection(IN PVOID SectionObject, ……)
{
    . . . . . .
    PMADDRESS_SPACE AddressSpace;
    . . . . . .
    Section = (PSECTION_OBJECT)SectionObject;
    AddressSpace = &Process->AddressSpace;

    MmLockAddressSpace(AddressSpace);

    if (Section->AllocationAttributes & SEC_IMAGE)
```

```
{
    ULONG i;
    ULONG NrSegments;
    ULONG_PTR ImageBase;
    ULONG ImageSize;
    PMM_IMAGE_SECTION_OBJECT ImageSectionObject;
    PMM_SECTION_SEGMENT SectionSegments;

    ImageSectionObject = Section->ImageSection;
    SectionSegments = ImageSectionObject->Segments;
    NrSegments = ImageSectionObject->NrSegments;
    ImageBase = (ULONG_PTR)*BaseAddress;
    if (ImageBase == 0)
    {
        ImageBase = ImageSectionObject->ImageBase;
    }

    ImageSize = 0;
    for (i = 0; i < NrSegments; i++)
    {
        if (!(SectionSegments[i].Characteristics & IMAGE_SCN_TYPE_NOLOAD))
        {
            ULONG_PTR MaxExtent;
            MaxExtent = (ULONG_PTR)SectionSegments[i].VirtualAddress +
                            SectionSegments[i].Length;
            ImageSize = max(ImageSize, MaxExtent);
        }
    }

    /* Check there is enough space to map the section at that point. */
    if (MmLocateMemoryAreaByRegion(AddressSpace, (PVOID)ImageBase,
                                            PAGE_ROUND_UP(ImageSize)) != NULL)
    {
        . . . . . .
        /* Otherwise find a gap to map the image. */
        ImageBase = (ULONG_PTR)MmFindGap(AddressSpace,
                                PAGE_ROUND_UP(ImageSize), PAGE_SIZE, FALSE);
        . . . . . .
    }

    for (i = 0; i < NrSegments; i++)
    {
        if (!(SectionSegments[i].Characteristics & IMAGE_SCN_TYPE_NOLOAD))
        {
```

```
            PVOID SBaseAddress = (PVOID)
                    ((char*)ImageBase + (ULONG_PTR)SectionSegments[i].VirtualAddress);
            MmLockSectionSegment(&SectionSegments[i]);
            Status = MmMapViewOfSegment(Process,
                                        AddressSpace,
                                        Section,
                                        &SectionSegments[i],
                                        &SBaseAddress,
                                        SectionSegments[i].Length,
                                        SectionSegments[i].Protection,
                                        0,
                                        FALSE);
            MmUnlockSectionSegment(&SectionSegments[i]);
            . . . . . .
        }
    }
    *BaseAddress = (PVOID)ImageBase;
}
else
{
    . . . . . .
    if (SectionOffset == NULL)
    {
        ViewOffset = 0;
    }
    else
    {
        ViewOffset = SectionOffset->u.LowPart;
    }
    . . . . . .
    if ((*ViewSize) == 0)
    {
        (*ViewSize) = Section->MaximumSize.u.LowPart - ViewOffset;
    }
    else if (((*ViewSize)+ViewOffset) > Section->MaximumSize.u.LowPart)
    {
        (*ViewSize) = Section->MaximumSize.u.LowPart - ViewOffset;
    }

    MmLockSectionSegment(Section->Segment);
    Status = MmMapViewOfSegment(Process,
                                AddressSpace,
                                Section,
                                Section->Segment,
```

```
                                        BaseAddress,
                                        *ViewSize,
                                        Protect,
                                        ViewOffset,
                                        (AllocationType & MEM_TOP_DOWN));
        MmUnlockSectionSegment(Section->Segment);
        . . . . . .
    }

    MmUnlockAddressSpace(AddressSpace);

    return(STATUS_SUCCESS);
}
```

我把这段程序留给读者自己阅读，只是略加提示：Section 对象所代表的目标文件分为两大类，一类是可执行映像文件，一类是不同文件。可执行映像文件的映射比普通文件要复杂一些，因为映像文件中一般有好多不同的段，需要映射到不同的地址上去，这就是代码中有两个 for 循环的原因。每个段的映射则都是由 MmMapViewOfSegment()完成的。

```
[LdrpMapSystemDll() > NtMapViewOfSection() >
  MmMapViewOfSection() > MmMapViewOfSegment()]

NTSTATUS STATIC
MmMapViewOfSegment(PEPROCESS Process,
                   PMADDRESS_SPACE AddressSpace,
                   PSECTION_OBJECT Section,
                   PMM_SECTION_SEGMENT Segment,
                   PVOID* BaseAddress,
                   ULONG ViewSize,
                   ULONG Protect,
                   ULONG ViewOffset,
                   BOOL TopDown)
{
    PMEMORY_AREA MArea;
    NTSTATUS Status;
    KIRQL oldIrql;
    PHYSICAL_ADDRESS BoundaryAddressMultiple;

    BoundaryAddressMultiple.QuadPart = 0;

    Status = MmCreateMemoryArea(Process,
                                AddressSpace,
                                MEMORY_AREA_SECTION_VIEW,
                                BaseAddress,
```

```
                                        ViewSize,
                                        Protect,
                                        &MArea,
                                        FALSE,
                                        TopDown,
                                        BoundaryAddressMultiple);
    . . . . . .

    KeAcquireSpinLock(&Section->ViewListLock, &oldIrql);
    InsertTailList(&Section->ViewListHead,
                        &MArea->Data.SectionData.ViewListEntry);
    KeReleaseSpinLock(&Section->ViewListLock, oldIrql);

    ObReferenceObjectByPointer((PVOID)Section,
                                        SECTION_MAP_READ,
                                        NULL,
                                        ExGetPreviousMode());
    MArea->Data.SectionData.Segment = Segment;
    MArea->Data.SectionData.Section = Section;
    MArea->Data.SectionData.ViewOffset = ViewOffset;
    MArea->Data.SectionData.WriteCopyView = FALSE;
    MmInitialiseRegion(&MArea->Data.SectionData.RegionListHead,
                            ViewSize, 0, Protect);

    return(STATUS_SUCCESS);
}
```

这里 MmCreateMemoryArea()的作用是为一个段的影射分配虚存区间：

- 按给定的地址要求在目标进程的用户空间找到足够大的“空隙”
- 如果并非必须映射在给定的地址，就找一个足够大的空隙，
- 从这个空隙中划出一块给定大小的区间
- 分配/创建一个 MEMORY_AREA 数据结构，并将其挂入相应的 AddressSpace 队列。
- MEMORY_AREA 数据结构除可挂入 AddressSpace 队列外还可挂入 Section 对象中的队列，这样就把内存区间、Section 对象、以及目标文件结合了起来。

对于了解 Linux 内核中存储管理和共享内存区映射的读者，这些操作和过程应该是容易理解的。但是我在这里要说的重点却并不在于这个过程本身，而在于这个过程中并无进程挂靠。

读者或许已经注意到，上面在以 NtMapViewOfSection()为入口的整个流程中，我们并没有看到对于 KeAttachProcess()的调用、即并没有进行进程挂靠。虽然这是在父进程的上下文中把一个 Section、即“区间”、影射到子进程的用户空间，但是却并不需要挂靠到子进程，这是为什么呢？要回答这个问题，我们先要搞清：所谓一个进程的用户空间是怎么体现的。简而言之，这主要体现为“一本账、一个表”。

首先，一个“用户空间”是一大片虚拟地址空间，在 Linux 中是 3GB、在 Windows 中是 2GB 的地址空间。但是这么大一片虚拟地址空间并不是都已分配使用，都已经映射到了

物理页面、或是某个映射文件或盘区。所以就需要有个账本，记下哪一些虚拟地址区间已经分配使用了，这就是“一本账”。在 Linux 内核中，这个账本就是以 mm_struct (在上面的代码中是 MADDRESS_SPACE)为根的一整套数据结构，在“进程控制块”task_struct 中有个指针指向本进程的 mm_struct 数据结构(在上面的代码中是&Process->AddressSpace)。由于已分配使用(而尚未释放)的虚拟地址区间一般都是不连续的，例如用于堆栈的区间和可执行代码的区间就不会连续，所以从数据结构的角度看这“账本”的具体内容总是一个链表，链表中的每一个结点都代表着一个已分配使用的地址区间，在 Linux 内核中这就是 vm_area_struct 数据结构(在上面的代码中是 MEMORY_AREA 数据结构)。在这一方面，不同操作系统的内核在具体的数据结构和程序实现上可以有所不同，但是大体上都是一样的，变不出太多的花样。所以，要把一个 Section 映射到一个进程的用户空间，首先是对这“账本”的操作。

但是，光有这账本还不够，因为这账本并不直接对 CPU 中的页面映射部件 MMU 起作用，所以还需要有一个用于 MMU 的页面映射表，这就是“一个表”。所谓挂靠到某个进程，就是把这个进程的页面映射表装入 MMU，使得访问用户空间的某个地址时使用的是目标进程的页面映射表。当然，在任何特定的时刻，MMU 中只能有一个页面映射表，既然装入了目标进程的页面映射表，就离开了原来进程的页面映射表。但是，不管是什么进程的页面映射表，他们的系统空间部分、即内核部分、则都是共同的。由此可见，“进程挂靠”(和恢复)只能在内核中进行，而不能在用户空间进行。

这里还要注意，对于页面映射表的“准备”和“使用”是两码事，建立映射时所涉及的是准备，而把准备好了的页面映射表装入 MMU 才开始了它的使用。

所以，ZwMapViewOfSection()之所以不需要挂靠到目标进程，是因为建立映射的过程只是账面的操作，而并不真的要去访问(目标进程)用户空间的某个地址。

按理说，既然是把一个 Section 映射到目标进程的用户空间，就应该同时完成对账本和映射表的操作。但是 ReactOS 的代码把这两种操作分离了开来，在 NtMapViewOfSection()中只是对账本的操作，而把对映射表的操作推迟了(下面就会看到)，那当然也是可以的。

至此，ntdll.dll 的映射已经完成，回到 LdrpMapSystemDll()的代码中，下一步是要从这映像中获取 LdrInitializeThunk()等函数的入口地址，这时候就需要实施进程挂靠了。

[LdrpMapSystemDll() > KeAttachProcess()]

```
VOID    STDCALL
KeAttachProcess(PKPROCESS Process)
{
    KIRQL OldIrql;
    PKTHREAD Thread = KeGetCurrentThread();

    DPRINT("KeAttachProcess: %x\n", Process);

    /* Make sure that we are in the right page directory */
    UpdatePageDirs(Thread, Process);

    /* Lock Dispatcher */
    OldIrql = KeAcquireDispatcherDatabaseLock();

    . . . . . .
```

```
    /* Check if the Target Process is already attached */
    if (Thread->ApcState.Process == Process ||
                            Thread->ApcStateIndex != OriginalApcEnvironment) {

        DPRINT("Process already Attached. Exitting\n");
        KeReleaseDispatcherDatabaseLock(OldIrql);
    } else {

        KiAttachProcess(Thread, Process, OldIrql, &Thread->SavedApcState);
    }
}
```

前面的映射只是记在了新建进程的账本上，却没有改变它的页面映射表，这里的UpdatePageDirs()就来处理这页面映射表了。

这里 KeAcquireDispatcherDatabaseLock()的作用是通过提高中断优先级达到禁止线程调度的目的。因为下面的 KiAttachProcess()即将实现用户空间的切换，在这个当口上是不能允许线程调度的。

下面就是“挂靠”的实施了。

[KeAttachProcess() > KiAttachProcess()]

```
VOID STDCALL
KiAttachProcess(PKTHREAD Thread, PKPROCESS Process,
                                        KIRQL ApcLock, PRKAPC_STATE SavedApcState)
{
    . . . . . .

    /* Increase Stack Count */
    Process->StackCount++;

    /* Swap the APC Environment */
    KiMoveApcState(&Thread->ApcState, SavedApcState);

    /* Reinitialize Apc State */
    InitializeListHead(&Thread->ApcState.ApcListHead[KernelMode]);
    InitializeListHead(&Thread->ApcState.ApcListHead[UserMode]);
    Thread->ApcState.Process = Process;
    Thread->ApcState.KernelApcInProgress = FALSE;
    Thread->ApcState.KernelApcPending = FALSE;
    Thread->ApcState.UserApcPending = FALSE;

    /* Update Environment Pointers if needed*/
    if (SavedApcState == &Thread->SavedApcState) {
```

```
        Thread->ApcStatePointer[OriginalApcEnvironment] = &Thread->SavedApcState;
        Thread->ApcStatePointer[AttachedApcEnvironment] = &Thread->ApcState;
        Thread->ApcStateIndex = AttachedApcEnvironment;
    }

    /* Swap the Processes */
    KiSwapProcess(Process, SavedApcState->Process);

    /* Return to old IRQL*/
    KeReleaseDispatcherDatabaseLock(ApcLock);

    DPRINT("KiAttachProcess Completed Sucesfully\n");
}
```

注意代码中的 Process->StackCount 与进程的“堆栈”并无关系，而是指进程挂靠的嵌套深度。

前面讲过，所谓挂靠到某个进程，就是切换到那个进程的用户空间，就是把那个进程的页面映射表装入 MMU，这里调用 KiSwapProcess()的原因就在于此。不过在此之前还需要把当前进程的 APC 队列从 ApcState 转移到 SavedApcState 去，所以还调用了 KiMoveApcState()，读者可以结合前一篇漫谈把这里的程序读懂。此外，这里 KeReleaseDispatcherDatabaseLock()一方面是解除对线程调度的禁令，一方面是回到原来的中断优先级。与之配对的是前面 KeAttachProcess()中的 KeAcquireDispatcherDatabaseLock()。

我们接着看 KiSwapProcess()的代码。

[KeAttachProcess() > KiAttachProcess() > KiSwapProcess()]

```
VOID
STDCALL
KiSwapProcess(PKPROCESS NewProcess, PKPROCESS OldProcess)
{
    //PKPCR Pcr = KeGetCurrentKpcr();

    /* Do they have an LDT? */
    if ((NewProcess->LdtDescriptor) || (OldProcess->LdtDescriptor)) {
        /* FIXME : SWitch GDT/IDT */
    }
    DPRINT("Switching CR3 to: %x\n", NewProcess->DirectoryTableBase.u.LowPart);
    Ke386SetPageTableDirectory(NewProcess->DirectoryTableBase.u.LowPart);

    /* FIXME: Set IopmOffset in TSS */
}
```

这里 Ke386SetPageTableDirectory()的作用就是切换用户空间，即装入目标进程的页面映

射表，这主要是对寄存器 CR3 的操作。

读懂了 KeAttachProcess()，自然也就懂得了 KeDetachProcess()。

回到前面 LdrpMapSystemDll()代码中，可以看到夹在 KeAttachProcess()和 KeDetachProcess()之间的操作主要是 LdrGetProcedureAddress()，就可以明白这是为什么了。因为 LdrGetProcedureAddress()是根据一个函数名从给定的映像中找到该函数的程序入口(当然，这必须是由目标映像导出的函数，否则也找不到)。这里要找的就是 LdrInitializeThunk()以及其它几个函数的入口。要在目标映像中寻找函数入口，当然就得访问这个映像、即访问这个映像在用户空间的所在地址区间，这就涉及页面映射表的使用(而不是准备)了。于是，就需要暂时切换到目标进程的用户空间，也就是“挂靠”到目标进程。当然，完成了操作之后还得切换回来，那就是 KeDetachProcess()的事了。

这里还要说一下，从程序的角度看，KeAttachProcess()以后就可以根据目标映像在用户空间的起始地址访问这个映像了，似乎很简单。但是实际的过程却并不那么简单。这个映像虽然已经在用户空间有了映射，也就是在页面映射表中有了相应的表项，但是此刻可能(应该说多半)还没有相应的物理页面，所以在第一次访问这个映像时就会发生缺页异常。然后，在内核对缺页异常的处理中，将会发现所映射的是一个磁盘文件、即映像文件中的一个逻辑页面，就为其分配一个物理页面并从磁盘文件读入这逻辑页面。从缺页异常返回以后，CPU 重新执行访问用户空间的那条指令，才能获得成功。就这样，访问到哪，就缺页到哪、读入到哪，慢慢地就星罗棋布、把许多页面从磁盘读了进来。然而，也许到目标映像结束运行时还有许多页面是从未读入内存的。所谓“工作集”的概念就是这样来的，但是那已经不在本文的话题之内了。