CreateProcess的流程

创建一个Windows进程需要执行多个步骤，公社及操作系统的三个部分：Windows客户端库Kernel32.dll(CreateProcessInternalW实际开始执行工作的地方)、Windows执行体，以及Windows子系统进程(Csrss)。由于Windows采用了多环境子系统架构，以及创建执行体进程对象（以便供其他子系统使用）与创建Windows子系统进程这两个工作是分开进行的。

下面的内容概括了Windows的CreateProcess函数创建进程的主要步骤。

1. 验证参数。
   1. 新进程的优先级是通过CreateProcess函数中的CreationFlags参数内的独立位指定的，Windows会选择所设置的最低优先级类，以确定最终要为进程分配哪个优先级。Windows共定义了6个进程优先级类，每个类一个值：
      1. “空闲”或“低”(4)
      2. 低于正常(6)
      3. 正常(8)
      4. 高于正常(10)
      5. 正常(13)
      6. 实时(24)

优先级只会影响进程中的线程，默认“正常”优先级。如果新进程指定了“实时”优先级，且该进程的调用方不具备Increase Scheduling Priority特权（SE\_INC\_BASE\_PRIORITY\_NAME），此时用高优先级代替。

SE\_INC\_BASE\_PRIORITY\_NAME 是 Windows 安全策略中的一个权限，用于提升进程的基本优先级，允许进程在系统资源有限的情况下获得更多的 CPU 时间和内存资源，从而提高进程的执行效率和响应速度。通常情况下，只有操作系统内部的核心进程才会使用该权限，因为这些进程需要更高的执行优先级来保证系统的稳定性和性能。其他普通进程一般不建议使用该权限，因为滥用该权限可能会导致系统资源分配不均衡，影响其他进程的正常运行。

* 1. 如果创建时使用的标志指定了进程需要被调试，Kernel32会调用DbgUiConnectToDbg发起到Ntdll.dll中原生调试代码链接，并从当前线程环境块(TEB)中获取调试对象句柄。

**DbgUiConnectToDbg**是Windows内核调试器中的一个API，用于连接到内核调试器。

在使用该函数时，可以通过指定回调函数，以便在连接到内核调试器时执行一些操作。此外，该函数还接受一个指向通信端口的指针，内核调试器会使用该端口来与进程通信。

* 1. 如果创建时有使用的标志没有指定DEBUG\_ONLY\_THIS\_PROCESS或DEBUG\_PROCESS，则Kernel32.dll会将硬错误模式设置为SEM\_FAILCRITICALERRORS，以确保在出现致命错误时不会弹出消息框，而是直接终止进程。
  2. 与用户有关的属性列表会从Windows子系统格式转换为原生格式，并会添加内部属性
  3. 如果进程时作业对象的一部分，但创建标志请求创建单独的DOS虚拟机(VDM)，则该标志会忽略。
  4. 由CreateProcess函数提供的进程和初始线程安全属性会被转换为相应的内部表现(OBJECT\_ATTRIBUTES结构)
  5. 由CreateProcessInternalW会检查是否要将进程创建为“现代化”形态。

(PROC\_THREAD\_ATTRIBUTE\_PACKAGE\_FULL\_NAME) 创建方本身为现代化形态也会创建出一个现代化形态的进程

Windows 10引入了一种新的应用程序模型，称为“现代化”（Modern Desktop）应用程序模型，它与传统的Win32应用程序有所不同。这种新模型的应用程序运行在沙箱中，与用户桌面分离，受到更多的限制和保护，可以通过Windows Store进行分发和管理。现代化应用程序还有一些额外的功能，比如支持触摸屏、本机通知等。因此，“现代化”形态的进程具有更高的安全性和可靠性，可以提供更好的用户体验。

* 1. 如果进程创建为现代化形态，可以调用BaseCreateLowBox内部函数创建的初始令牌中将包含对应的安全能力。注意，虽然不支持调用CreateProcess直接创建现代化进程（此时应该使用COM接口），MSDN中确实介绍了通过传递该属性创建AppContainer传统桌面应用程序的做法。
  2. 如果创建现代化进程，则随后需要设置一个标志让内核跳过嵌入清单检测操作。现代化进程绝不会具备嵌入清单，因为它们已经有自己的嵌入清单了。

进程的嵌入清单（Manifest）是一种 XML 文件，用于描述应用程序或组件的相关信息，如文件依赖关系、程序集信息、程序版本、管理员权限等。它可以被嵌入到可执行文件、DLL 等二进制文件中，用于告知操作系统和运行时环境如何处理该文件。

嵌入清单可以通过修改资源文件来添加，也可以单独保存在一个 XML 文件中并在编译时进行嵌入。Windows 系统会根据清单文件的内容来决定如何加载、运行该程序，如是否需要以管理员权限运行、是否需要在 DPI 模式下运行等。在 Windows Vista 及其后续版本中，嵌入清单还可以用来声明 UAC 权限请求，提示用户确认是否运行具有管理员权限的程序。

* 1. 如果指定了调试标志(DEBUG\_PROCESS)，那么可执行文件的映像文件执行选项(Image File Execution Options)注册表键下的Debugger值会标记为跳过。否则调试器永远无法创建自己的调试对象(debuggee)进程，因为创建将进入无尽的循环(试图反复不断创建调试器进程)。
  2. 将所有窗口关联给桌面(指对工作空间提供的图像化呈现)。如果STARTUPINFO结构未指定桌面，进程会关联至调用方的当前桌面。

Windows 10虚拟桌面功能在内核对象层面上并未使用多个桌面对象。实际上依然是一个桌面，但窗口可按需显示或隐藏。Sysinternals的desktop.exe采取了截然不同的方法，可以真正创建4个桌面对象。如果试图将窗口从一个桌面移动到另一个桌面，就可以感受到其中的差异。但是Windows的desktops.exe不支持这样的操作。但Windows 10的虚拟桌面虽然支持这样做，但是并没有真正“移动”窗口

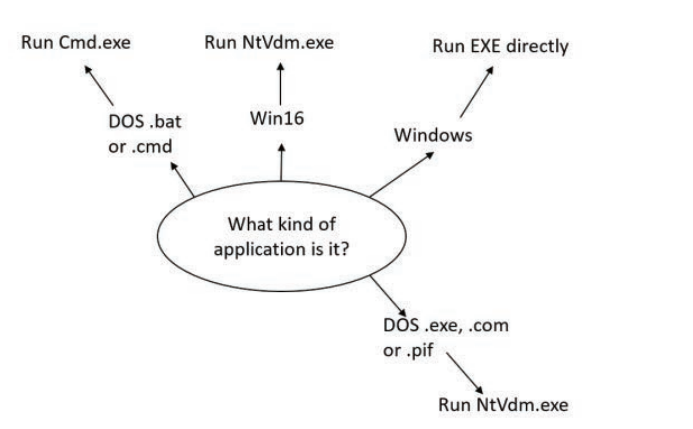
* 1. 可执行文件路径会转换为NT内部名称(eg: c:\temp\a.exe --> \device\harddiskvolume1\temp\a.exe)
  2. 将收集到的大部分信息转换为RTL\_USER\_PROCESS\_PARAMETERS这个结构

上述操作完成后，CreateProcessInternalW会首次调用NtCreateUserProcess，并试图创建进程。由于Kernel32.dll此时还不知道应用程序映像名称是否是真正的Windows应用程序还是批处理文件(.bat or .cmd)、16位或DOS应用程序，此时调用会失败，随后CreateProcessInternalW会查看错误并尝试纠正

1. 打开要在进程执行的映像文件(.exe)

在这以阶段，创建进程的线程会切到内核模式，并在NtCreateUserProcess系统调用实现中继续完成自己的工作。

* 1. NtCreateUserProcess首先会验证参数并构建一个保存所有创建信息的内核结构。再次验证参数的原因在于确保对执行体的调用并非源自某种不当手段(hack)，并恰好通过这种手段使用为造成或恶意的参数模拟了Ntdll.dll转换至内核的方法。
  2. NtCreateUserProcess的下一阶段的工作是找到运行调用方指定的可执行文件对应的Windows映像，并创建区域对象，随后将其映射至新进程的地址空间，如下图，如果调用失败了，会为CreateProcessInternalW返回失败状态，然后CreateProcessInternalW会试图重新执行



* 1. 如果要创建现代化进程，还需要通过许可检查来确保其具备许可并且允许运行。对于内置（Windows自带）应用，无论许可检查的结果如何均可运行；如果允许侧面加载应用（用过“设置”应用配置），那么除了来自应用商店的应用，任何签名的应用均可运行。
  2. 如果要创建Trustlet进程，必须使用一个特殊标志创建区域对象以供安全内核使用。
  3. 如果是EXE文件，NtCreateUserProcess会试图打开该文件并为其创建对象。此时该对象虽已打开但尚未映射至内存。成功创建区域对象并不一定一位置文件是有效地Windows映像，也可能是DLL或POSIX可执行文件。对于POSIX文件，调用会失败，因为POSIX已经不被支持。对于DLL文件，CreateProcessInternalW也会失败
  4. 注意，当NtCreateUserProcess使用进程创建代码找到有效的Windows可执行映像后，还会检查注册表HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options，并在这里查看是否子键包含了可执行映像的文件名和扩展（但不检查目录和路径信息，例如仅仅会查找“Notepad.exe”）。如果存在这样的子键，会通过PspAllocateProcess在该子键下找到Debugger值。如果存在该值，将要运行的映像会成为该值中的字符串，随后CreateProcessInternalW将重新执行第一阶段的操作。

提示： 可以利用创建进程这个行为，在实际启动之前调试Windows服务进程的启动代码，而不需要在启动服务后再连接调试器，这种方式无法调试启动代码

* 1. 此外，如果映像不是EXE文件（例如MS-DOS或Win16应用程序）CreateProcessInternalW会通过一系列步骤找到可用于运行的Windows支持映像。该过程是需要的，因为非Windows应用程序不能直接运行。相反，Windows会使用几个特殊的支持映像来运行它们，并由这些支持映像负责非Windows程序的实际执行工作。例如在Windows 32位系统下运行MS-DOS或Win16可执行文件，实际运行的映像会是Ntvdm.exe。简而言之，无法直接创建非Windows进程类型的进程。如果Windows无法通过某种方法将要激活的映像解析为Windows进程，CreateProcessInterlW将会失败。

|  |  |  |  |
| --- | --- | --- | --- |
| 如果映像…… | 创建状态码 | 将要运行的映像…… | 接下来会发生的事…… |
| 是MS-DOS应用程序，.ex、.com或 .pif | PsCreateFailOnSection Create | Ntvdm.exe | CreateProcessInternalW重新执行第一阶段任务 |
| 是Win16应用程序 | PsCreateFailOnSection Create | Ntvdm.exe | CreateProcessInternalW重新执行第一阶段任务 |
| 是32位系统上运行的Win64应用程序(或是PPC、MIPS或Alpha二进制文件) | PsCreateFailMachine Mismatch | 不可用 | CreateProcessInternalW失败 |
| Debugger值设置为另一个映像文件 | PsCreateFailExeName | Debugger值指定的名称 | CreateProcessInternalW重新执行第一阶段任务 |
| 是无效或损坏的Windows EXE | PsCreateFailExeFormat | 不可用 | CreateProcessInternalW失败 |
| 无法打开 | PsCreateFailOnFileOpen | 不可用 | CreateProcessInternalW失败 |
| 是命令行程序(.bat或.cmd) | PsCreateFailOnSection Create | Cmd.exe | CreateProcessInternalW重新执行第一阶段任务 |

1. 创建Windows执行体进程对象

至此，NtCreateUserProcess已经打开了有效的Windows可执行文件，创建了区域对象，并将其映射到了新进程的地址空间。随后需要调用PspAllocateProcess这个内部系统函数，创建用于运行该映像的Windows执行体进程对象。执行体进程对象的创建（由创建线程来执行）涉及到下列几个子阶段

* 1. 设置EPROCESS
     1. 继承父进程的处理器相关性，除非在进程创建过程中（通过属性列表）明确设置。
     2. 如果属性列表指定了理想NUMA节点，则选择该节点

NUMA（Non-Uniform Memory Access）是一种多处理器计算机体系结构，其中每个处理器核心连接到不同的内存集合，这些内存集合通常称为“节点”。这些节点可能包含一个或多个处理器核心和内存。NUMA体系结构旨在提高系统性能和可伸缩性，因为每个节点都可以与其他节点同时工作而不会互相干扰。

在NUMA体系结构中，处理器核心和内存之间的延迟取决于它们的物理位置，不同节点之间的延迟可能不同。这就是所谓的“非均匀内存访问”，因为不同节点之间的内存访问可能会导致不同的延迟。因此，应用程序需要考虑NUMA架构，以确保最佳性能并避免延迟问题。

* + 1. 继承父进程的I/O页面优先级。如果没有父进程，则使用默认的页面优先级(5)和I/O优先级(正常)。
    2. 将新进程的退出状态设置为STATUS\_PENDING
    3. 选择由属性列表决定的硬错误模式。如果未指定，则继承父进程的处理模式。如果没有父进程，则使用默认处理模式，即显示所有错误。
    4. 将父进程的ID存储到新进程对象的InheritedFromUniqueProcessId字段中。
    5. 除非进程在Wow64下运行，此时无法使用大页面，否则将查询Image File Execution Option(IFEO)键中的UseLargePages值，看是否用大页面映射该进程。此外还会查询该键来确定NTDLL是否被列为需要在该进程中使用大页内存映射的DLL之一。

Wow64进程指的是在64位Windows操作系统下运行的32位应用程序进程。由于64位Windows操作系统与32位Windows操作系统存在着很大的差异，例如指针大小、寄存器数量等等，因此Windows系统采用了Windows on Windows 64（WoW64）技术来支持在64位操作系统下运行32位应用程序。Wow64进程是在WoW64子系统下运行的32位应用程序，通过WoW64子系统将32位应用程序需要的系统服务转换为64位Windows操作系统下的系统服务来完成相应操作。

* + 1. 查询IFEO的性能选项建(PerfOptions，如果存在的话)，其中可能包含任意数量的下列可用值：IoPriority、PagePriority、CpuPriorityClass及WorkingSetLimitInKB。

PerfOptions是Windows操作系统中一个注册表项，用于配置性能选项，以便优化系统的性能。

PerfOptions注册表项的位置在HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters

PerfOptions注册表项通常包含以下子项：

* CpuPriorityClass：指定系统的CPU优先级类别。
* CpuPriorityBoostEnabled：指定系统是否启用了CPU优先级提升。
* CsEnabled：指定系统是否启用了代码完整性检查。
* DisableDynamicTick：指定系统是否启用了动态时钟节拍功能。
* InterruptAffinityPolicy：指定系统在为中断分配CPU时使用的CPU亲和性策略。
* InterruptSteeringEnabled：指定系统是否启用了中断引导功能。
* PowerSettings：包含一组子项，用于指定系统的电源选项。
* ProcessorCapacityPolicy：指定系统处理器的功率管理策略。
* ProcessorIdleStateMax：指定系统处理器的最大空闲状态。
* ProcessorIdleStateMin：指定系统处理器的最小空闲状态。
* ProcessorParked：指定系统是否允许将处理器停靠（减少功耗）。
* ProcessorPerformanceBoostMode：指定系统是否启用了处理器性能提升模式。
* SystemResponsivenessEvents：指定系统响应性事件的配置。

这些选项可以通过更改PerfOptions注册表项中的值来进行配置。

* + 1. 如果进程要在Wow64下运行，则随后会分配Wow64辅助结构(EWOW64PROCESS)，并将其设置到EPROCESS结构的WoW64Process成员中
    2. 如果进程要创建到AppContainer内部（大部分情况下，现代化应用需要这样做），则需要验证令牌是否由LowBox创建
    3. 尝试获取创建该进程所需的全部特权。选择进程优先级类为“实时”，为新进程分配令牌，使用大页面映射进程，在新会话中创建进程，这些操作均需要相应的特权。
    4. 创建进程的主访问令牌（其父进程主令牌的副本）。新进程可以继承父进程的安全配置文件。如果使用CreateProcessAsUser函数为新进程指定不同的访问令牌，则该令牌也会酌情调整。这一改动只在父进程令牌的完整性级别能够支配子进程访问令牌的完整性级别，并且该访问令牌的父令牌真正的子令牌或兄令牌的情况下才会进行。如果父进程具备SeAssignPrimaryToken特权，则会绕过上述检查。
    5. 随后检查新进程令牌的会话ID，以确定是否在进行跨会话创建。如果是，父进程会临时附加到目标会话，以便正确处理配额和地址空间的创建。
    6. 将新进程的配额快设置为其父进程配额块的地址，并增加父进程配额块的引用计数。如果使用CreateProcessAsUser创建新进程，将不执行这一步操作。这种情况将创建默认配额，或与所选用户配置文件相符的配额。
    7. 将进程工作集大小的最小值和最大值分别设置为PspMinimumWorkingSet和PspMaximumWorkingSet的值。如果Image File Execution Options下的PerfOptions键指定了性能选项，这些值可能会被覆盖，此时将从性能选项中获取工作集的最大值。注意，默认的工作集大小是一种软限制，而PerfOptions设置的工作集最大值是一种硬限制。
    8. 初始化进程地址空间。如果涉及不同会话，则从目标会话分离。
    9. 如果未使用继承的组相关性，则随后要为进程设置组相关性。默认的组相关性可以在事先设置过NUMA节点传播（使用拥有NUMA节点的组）的情况下从父进程继承，或按照轮询的方式分配。如果系统运行与强制组感知模式下，并且选择算法选择了组0，则将选择组1来使用(如果存在的话)。

强制组感知模式（Forced Group Awareness）是一种Windows操作系统中的性能优化技术，用于提高多处理器系统的性能。在多处理器系统中，为了提高并发执行的效率，操作系统会将线程分配到不同的处理器上执行。但是，在线程切换的过程中，需要考虑线程之间的亲和性（Affinity）和群组（Group）关系，以避免不必要的开销。如果线程被分配到一个不同的组，则需要进行群组切换，这会引起一定的性能开销。

强制组感知模式通过改变线程的亲和性，使得相邻的线程都被分配到同一组中，从而减少群组切换的次数，提高系统的性能。

* + 1. 初始化进程对象中的KPROCESS部分
    2. 设置进程的令牌
    3. 将进程优先级设置为“正常”，除非父进程优先级为“低于正常”或“空闲”此时将继承父进程的优先级
    4. 初始化进程句柄表。如果已经设置了父进程的继承句柄标志，那么所有可继承的句柄会从父进程的句柄表复制给新进程。此外可以使用一个进程属性指定句柄子集，如果使用CreateProcessAsUser限制子进程可继承的对象，这将是一个很实用的方法。
    5. 如果通过PerfOptions注册表键指定了性能选项，此时将应用这些选项。PerfOptions键包含了有关进程的工作集限制覆盖值、I/O优先级、页面优先级、CPU优先级类等设置。
    6. 计算并设置最终进程优先级类及其线程的默认量程(quantum)。
    7. 从IFEO注册表项读取并应用各种缓解选项（一个名为Mitigation的64位值）如果进程运行在AppContainer中，则会添加名为TreatAsAppContainer的缓解标志。
    8. 应用其他缓解标志
  1. 创建初始进程地址空间

初始进程地址空间包含下列页

* + 1. 页目录（对于超过两级页表的系统，例如PAE模式的x86系统或x64系统，可能有多个页目录，x64下一级页表一页4KB=4096字节存放页地址，每个页地址指针大小为8字节，能放512个指针，也就是一级页表可以容纳512 \* 4KB=2MB，二级页表使用一个4KB内存页存放一级页表指针，也就是说二级页表可以容纳512 \* 512 \* 4KB = 1GB，以此类推，3级页表就是512GB）
    2. 超空间(hyperspace)页

超空间页（Hyper Page）是Windows操作系统中的一种内存管理技术。它是用于管理高速缓存（如处理器高速缓存、I/O高速缓存等）的内存页面，使得缓存可以映射到用户进程地址空间中。在Windows操作系统中，缓存不仅可以存储数据，还可以存储代码，以此提高系统性能。

超空间页是一种可被多个进程共享的页面，因此它具有很高的重用率和可扩展性。当多个进程需要使用相同的页面时，它们可以共享同一个超空间页，这样就可以节省大量的物理内存。同时，由于超空间页是可重用的，因此可以将其保留在缓存中，以便将来使用。

* + 1. VAD位图页

在Windows内核中，虚拟地址描述符（VAD）是用于跟踪进程虚拟地址空间中的内存段的结构。在VAD中，有一个位图来跟踪该VAD所占用的物理内存页面是否已经被写入了磁盘。这个位图被称为VAD位图页。

VAD位图页是在内核中使用的一种数据结构，用于描述虚拟地址空间中哪些页面已经被写入到磁盘交换文件中。该位图包含多个位数组成，每个位表示一个页面的状态。这些位的值可能为1（表示页面已被写入磁盘）或0（表示页面未被写入磁盘）。VAD位图页的存在可以提高页面回收时的效率，减少系统开销。

* + 1. 工作集列表

要创建这些页，需要执行下列步骤：

* + - 1. 在相应页表中创建页表项，并映射至初始页
      2. 从内核变量MmToalCommittedPages中扣除页的数量，并添加到MmProcessCommit中
      3. 从MmResidentAvailablePages中扣除系统范围内默认的进程最小工作集尺寸(PsMinimumWorkingSet)。
      4. 创建全局系统空间（除前文提到的与进程有关的页，以及与特定会话有关内存之外的其他空间）的页表页
  1. 初始化内核进程结构(KPROCESS)

PspAllocateProcess的下个阶段是初始化KPROCESS结构(EPROCESS的Pcb成员)该工作是由KeInitializeProcess 进行的，将执行下列操作。

1. 对用于连接进程中所有线程(最初为空)的双重链接列表(doubly linked list)进行初始化。
2. 在稍后使用 PspComputeQuantumAndPriority 完成初始化之前，进程默认量程的初始值(或重置值)临时被硬编码为“6”。

注意：Windows 客户端和服务器系统的默认初始量程有所差异。

1. 根据阶段3A的计算结果设置进程的基本优先级
2. 以组相关性的方式为进程中的线程设置默认处理器相关性。组相关性可通过阶段3A 计算得出，或从父进程继承。
3. 将进程交换状态设置为常驻 (resident)。
4. 线程种子基于内核为该进程选择的理想处理器(理想处理器的选择取决于之前创建进程所用的理想处理器，最终效果类似于通过轮询方式进行随机化选择。) 新建进程将更新 KeNodeBlock (初始NUMA 节点块)种子，因此下个进程将获得一个不同的理想处理器种子
5. 如果进程时安全进程，将调用HvlCreateSecureProcess为其创建安全ID。

安全进程（Secure Process）是 Windows 操作系统中的一种特殊进程，它运行在安全且隔离的环境中，与操作系统内核紧密集成，主要负责管理操作系统中的安全相关任务，例如数字版权保护、数字签名、加密等等。安全进程通过运行在安全且隔离的环境中，防止了恶意软件或者攻击者通过攻击安全进程来获取系统权限或者篡改敏感信息的风险。

* 1. 结束进程地址空间的设置

为新进程创建地址空间的过程有些复杂，因此我们将逐个介绍整个过程涉及的不同步骤。若要充分理解这部分内容，必须对 Windows 内存管理器的原理有所了解。在创建地址空间的过程中，大部分工作都是由MmInitializeProcessAddressSpace例程负责处理的。该例程还可以从其他进程克隆地址空间。在实现POSIX的Fork系统调用时这种能力非常实用。未来还可能会通过这种能力支持其他UNIX 风格的 Fork(这也是目前处于里程碑1阶段的、适用于Linux的Windows 子系统中实现Fork 的方式)。下列步骤并未涉及地址空间的克隆，只专注于常规的进程地址空间初始化过程。

1. 虚拟内存管理器将进程的最后裁剪(trim)时间值设置为当前时间。工作集管理器(运行于平衡集管理器系统线程的上下文中)使用该值确定何时进行工作集裁剪。

在 Windows 操作系统中，平衡集管理器是一种用于调整系统资源分配以实现负载均衡和资源利用率最大化的机制。它可以根据系统的负载情况自动调整 CPU、内存、网络和磁盘等资源的分配，以确保所有任务都得到合理的分配和处理，从而提高系统的响应速度和性能。

1. 内存管理器初始化进程的工作集列表，至此已经可以处理页面错误了。
2. （在打开映像文件时创建的）区域已经可以映射至新进程的地址空间，进程区域的基址将设置为映像的基址。
3. 创建并初始化进程环境块(PEB)
4. Ntdll.dll被映射至进程。对于Wow64进程，还将映射32位的Ntdll.dIl
5. 如果请求，还将为该进程创建新会话。这一特殊步骤主要是为了方便会话管理器(Smss)初始化新会话。
6. 为标准句柄创建副本，并将新值写入进程的参数结构中。
7. 处理属性列表中列出的所有内存保留。此外可以使用两个标志对地址空间头部的1MB或16MB内存进行大块保留。这些标志是内部使用的，例如用于映射实模式下的向量和ROM代码(它们必须位于虚拟地址空间的低范围，通常用于放置堆或其他进程结构的地方)。
8. 将用户进程参数写入进程，复制并固定(也就是说会从绝对形式转换为相对形式这样只需要一个内存块。
9. 将相关性信息写入PEB。
10. 将MinWin API重定向集映射至进程，并将其指针存储至PEB。
11. 确定并存储进程唯一ID。内核并不能区分进程ID、线程ID和柄之间的唯一性进程ID和线程ID(句柄)会存储在一个与任何进程均无关的全局柄表(PspCidTable)中
12. 对于安全进程(在IUM下运行)，将初始化安全进程并关联给内核进程对象。
    1. 设置PEB

NtCreateUserProcess 调用MmCreatePeb，首先将系统范围内的国家语言支持 (NationalLanguage Support，NLS)表映射至进程的地址空间，随后调用 MiCreatePebOrTeb为PEB分配一个页，并初始化一系列字段，其中大部分字段基于通过注册表配置的内部变量，如MmHeap值、MmCriticalSectionTimeout 和MmMinimumStackCommitInBytes; 部分字段可使用链接的可执行映像中的设置覆盖，例如 PE头中的 Windows版本或PE头加载配置目录中的相关性掩码。

如果像头特征中的IMAGE\_FILE\_UP\_SYSTEM\_ONLY 标志已设(意味着改映像只能在单处理器系统中运行)，则会为这个新进程所运行的全部线程选择同一个CPU（MmRotaingUniprocessorNumber）。选择过程很简单，会直接循环使用所有可用处理器。每次此类映像运行时，都将使用下一颗处理器，借此，这类映像就可以均匀地分配到所有处理器上。

* 1. 完成执行体进程对象的设置工作。

在返回新进程的句柄前，还需要完成一些最终的设置工作，这些工作是由PspInsertProcess及其辅助函数完成的。

1. 如果对进程启用了系统范围内的审核(也许来自本地策略设置或域控制器提供的组策略设置)，进程的创建会写入安全事件日志。
2. 如果父进程包含在作业中，则会从父进程作业级别的集合中恢复该作业，随后将其绑定至新建进程的会话中。最终，新进程也会被加入该作业。
3. 新进程对象插入 Windows活动进程列表的末尾(PsActiveProcessHead)，随后即可通过诸如EnumProcesses和OpenProcess 等函数访问该进程。
4. 除非设置了NoDebugInherit 标志(可在建进程时请求该标志)，否则父进程的进程调试端口将复制给新建的子进程。如果指定了调试端口，则该端口会附加到新进程上
5. 作业对象可以做出限制，指定作业内的进程中的线程可以在哪个或哪些组中运行。因此PspInsertProcess必须保证进程的组相关性不会违反作业的组相关性。另外还有个值得注意的问题:作业的访问权限是否可以修改进程的相关性权限，因为低特权作业对象可能会干扰到高特权进程的相关性要求。
6. PspInsertProcess调用ObOpenObjectByPointer为新进程创建句柄，随后将该句柄返回给调用方。注意，在进程中创建出第一个线程之前不会发出进程创建回调函数，并且代码始终会在发送基于对象管理的回调之前，先发送基于进程的回调。

注意：只有一种情况下不存在父进程：系统初始化（即创建System进程的）过程中，在这之后，始终需要由父进程为新进程提供安全上下文(一般桌面打开的EXE件的进程的父进程是explorer.exe，也就是资源管理器)

1. 创建初始线程（栈、上下文、Windows执行体线程对象）

接下来需要创建所需线程。一般来说，可以通过 PspCreateThread 例程处理与线程创建有关的工作，例如在创建新线程时调用NtCreateThread。然而，由于初始线程是内核在无须用户模式输入的情况下在内部创建的，此时 PspCreateThread 需要依靠两个辅助例程:PspAllocateThread和PspInsertThread。PspAllocateThread 承担了执行体线程对象本身的创建和初始化工作，PspInsertThread承担了线程句柄和安全属性的创建工作，并负责调用KeStartThread 将执行体对象转换为系统中可调度的线程。然而目前线程还法开始工作，因为在创建时就处于挂起状态，只有在进程全面初始化之后才会恢复。

注意 线程参数(无法通过CreateProcess 指定，但可通过 CreateThread 指定)是PEB的地址。该参数可被新线程上下文内运行的初始化代码使用。

PspAllocateThread 将执行下列操作

* + 1. 可避免在 Wow64进程中创建出用户模式调度(UMS)线程，并避免用户模式的调用放在系统进程中创建线程。
    2. 创建并初始化一个执行体线程对象。
    3. 如果系统启用了能耗估算 (energy estimation，Xbox 会始终禁用)，则会分配并初始化一个指向 ETHREAD对象的THREAD\_ENERGY\_VALUES 结构
    4. 初始化被 LPC、IO 管理以及执行体使用的各种列表。

LPC是指本地过程调用（Local Procedure Call），是Windows系统内核中进程间通信（IPC）机制之一。LPC是一种高效、可靠、安全的进程间通信方式，用于进程之间传递消息、请求服务或共享资源。LPC是由Windows NT系统引入的一种内核通信机制，可以在内核空间和用户空间之间传递消息和数据，不同于其他的IPC机制，如管道、信号量等，LPC是一种零拷贝的机制，能够提高进程间通信的效率。在Windows系统中，进程可以使用LPC来向其他进程或操作系统内核发送消息，并可以等待其他进程或操作系统内核的响应消息。

* + 1. 设置线程的创建时间并创建线程ID(TID)。
    2. 线程在执行之前需要栈和上下文方能运行，此时将设置这些东西。初始线程的栈大小来自映像文件，无法修改为其他大小。对于 Wow64 进程，还会初始化 Wow64 线程上下文。
    3. 用户模式线程的起始地址将保存在ETHREAD(的StartAddress 字段)中。这是系统提供的线程启动函数，位于Ntdll.dll 内 (RtlUserThreadStart)。用户指定的 Windows启动地址将保存在 ETHREAD 中一个不同的位置(Win32StartAddress 字段)，因此诸如Process Explorer 等调试工具才可以显示此类信息。
    4. 为新线程分配线程环境块 (TEB)。
    5. 用KeInitThread 设置 KTHREAD结构。线程的初始和当前基准优先级将设置为进程的基准优先级，相关性和量程也会设置为与进程相符。随后 KelnitThread 会为线程分配一个内核栈，并为线程初始化与计算机相关的硬件上下文，包括上下文、陷阱和异常帧。设置了上下文的线程即可开始在内核模式的 KiThreadStartup 中启动。最终KeInitThread会将线程的状态设置为“已初始化”并返回给 PspAllocateThread。
    6. 对于UMS 线程，将调用 PspUmsInitThread 来初始化UMS 状态。

上述工作完成后，NtCreateUserProcess将调用PspInsertThread 来执行下列步骤

* + 1. 如果通过属性指定，将对线程的理想处理器进行初始化。
    2. 如果通过属性指定，将对线程的组相关性进行初始化。
    3. 如果进程是某个作业的一部分，将通过检查确保线程的组相关性不会违背作业本身的限制。
    4. 检查进程是否已终止、线程是否已终止，以及线程是否根本没能开始运行。如果遇到上述任何一种情况，线程的创建将失败。
    5. 调用KeStartThread 初始化线程对象中的KTHREAD部分。这一过程需要从拥有者进程继承调度器设置，设置理想节点和处理器，更新组相关性，设置基本和动态优先级(从进程中复制)，设置线程量程，将线程插入由 KPROCESS 维护的进程列表(与EPROCESS不同的另一个列表)中。
    6. 如果线程是安全进程(IUM)的一部分，则会创建并初始化安全线程对象。
    7. 如果进程处于深度冻结状态(此时任何线程都无法运行，包括新建的线程)，则该线程也将被冻结。
    8. 在非x86系统中，如果该线程是进程中的第一个线程(且该进程不是Idle 进程),随后该进程会被插入由全局变量 KiProcessListHead 维护的、在另一个系统范围内的进程列表中。
    9. 递增进程对象中的线程计数，并继承拥有者进程的I/O 优先级和页面优先级。如果该线程是进程中序号最高的线程，线程计数高水印(high watermark) 也会同步更新。如果是进程中的第二个线程，则主令牌将被冻结
    10. 线程将被插入进程的线程列表，如果创建方进程请求，该线程会被挂起
    11. 线程对象会被插入进程句柄表。
    12. 如果是进程中创建的第一个线程(即线程创建操作是在 CreateProcess函数调过程中进行的)，还将调用所有已注册的进程创建回调函数，随后还会调用所有已注册“线程回调函数。如果有任何回调函数拒绝创建操作，线程创建将失败，并会向调用方返回相应的状态信息。
    13. 如果(使用属性) 提供作业列表，且这是进程中的第一个线程，那么该进程会被分配给作业列表中的所有作业。
    14. 该线程已经可以调用 KeReadyThread 来执行了。它将进入延迟就绪状态。

1. 执行创建之后需要的、与Windows子系统有关的进程初始化操作(设置新的进程和线程)。

当NtCreateUserProcess 返回了成功代码后，必要的执行体进程和线程对象就已创建完成。随后将由CreateProcessInternalW 执行多种与Windows 子系统有关的操作，借此完成进程的初始化。

1. 通过多种检查确定 Windows 是否允许该可执行文件。这些检查包括验证文件头中的映像版本，以及检查 Windows 应用程序证书是否(通过组策略)禁止了该进程。在特殊版本的Windows Server 2012 R2(例如Windows Storage Server 2012 R2)中，还将通过额外的检查确定应用程序是否导入了不允许的API。
2. 如果软件限制策略有指示，则将为新进程创建一个受限令牌。随后会查询应用程序兼容性数据库，以确认注册表或系统应用程序数据库中是否有关于该进程的任何记录此时还不会应用兼容性填充码(shim)，相关信息会保存在PEB中并在初始线程开始执行时(第6阶段)应用。
3. CreateProcessIntermalw 调用一些内部函数(针对非受保护进程)以获取 SxS信息，例如清单文件和DLL重定向路径，以及诸如EXE文件所在介质是否为可移动存储介质、安装程序检测标志等其他信息。对于沉浸式进程(UWP应用的“现代化”进程)，还将返回通过包清单获取的版本和目标平台信息。

SxS（Side-by-Side）是Windows操作系统的一项技术，旨在允许多个版本的同一应用程序在同一计算机上同时存在和运行，而不会相互干扰。在Windows中，SxS信息主要用于支持应用程序和系统组件的版本控制，允许多个版本的同一组件在同一计算机上同时存在并被不同的应用程序调用，从而避免了应用程序之间的冲突和依赖问题。

在Windows中，SxS信息通常存储在WinSxS目录中，该目录包含了各种组件文件和元数据文件。每个组件都有一个唯一的标识符（GUID），该标识符用于区分不同版本的组件。当应用程序需要调用一个组件时，它会在WinSxS目录中查找该组件，并使用组件的标识符确定要使用的版本。由于每个组件都是在独立的文件夹中保存的，因此不同版本的同一组件可以同时存在，而不会相互干扰。

SxS信息也可以用于管理应用程序的依赖关系。在安装应用程序时，安装程序会检查所需的组件是否已经安装在系统中，如果没有，它会从安装程序的资源中提取所需的组件，并将它们放入WinSxS目录中。这样，应用程序可以确保它所依赖的所有组件都存在，并且是正确的版本。

1. 根据收集到的上述信息创建一条要发送给 Windows 子系统(Csrss)的消息，该消息包含下列信息:

* 路径名称和SxS路径名称;
* 进程和线程句柄;
* 区域句柄;
* 访问令牌句柄;
* 媒体信息;
* AppCompat和填充码数据;
* 沉浸式进程信息;
* PEB地址
* 各种标志，如是否为受保护进程、是否需要在提权后运行等;
* 代表该进程是否属于某个Windows应用程序（Csrss借此决定是否显示启动光标）的标志
* UI语言信息
* DLL重定向和.local标志
* 清单文件信息

收到该消息后，Windows 子系统将执行下列步骤。

1. CsrCreateProcess 复制进程和线程的句柄。在这一步中，进程和线程的引用计数会从(创建时设置的)1 递增到2
2. 分配 Csrss 进程结构(CSR PROCESS)。
3. 新进程的异常端口将设置为Windows子系统的通用功能端口，借此Windows子系统能在进程中发生第二次异常(second-chance cxceplion)时收到消息。
4. 如果要新建进程组，且该新进程会成为进程组的根(在 CreateProcess 中使用了CREATE\_NEW\_PROCESS\_GROUP 标志)，则会在 CSR\_PROCESS 中设置。进程组可用于将控制事件发送给一组共享了同一个控制台的进程。
5. 分配并初始化Csrss 线程结构(CSR\_THREAD)
6. CsrCreateThread将该线程插入进程的线程列表
7. 递增会话中的进程计数。
8. 进程的关机级别(shutdown level)设置为0x280（0b001010000000），这是默认的进程关机级别。
9. 新建的Csrss进程结构被插入 Windows 子系统范围内的进程列表Csrss 执行完这些步骤后，CreateProcessInternalW 将检查进程是否需要先提权再运行(这意味着需要通过 ShellExecute 行，并当用户在 UAC 对话框中选择“同意”后通过AppInfo 服务进行提权)。这一过程中需要检查该进程是否为安装程序。如果是，则会打开进程的令牌并开启虚拟化标志，借此实现该应用程序的虚拟化。如果该应用程序包含需要提权的填充码或在清单中请求了提升级别此时将销毁进程并将提权请求发送给 AppInfo 服务。

注意，受保护进程无须进行大部分此类检查，因为受保护进程必须是专为 Windows Vista 及后续版本操作系统设计的，没理由需要对其进行提权、虚拟化或应用程序兼容性检查和处理。此外，如果允许使用诸如填充码引擎等机制对受保护进程使用常规的挂钩和内存补丁技术，当有人发现如何插入任意填充码进而修改受保护进程时，这将会产生安全漏洞。最后，因为填充码引擎是由父进程安装的，所以可能无法访问受保护的子进程，甚至合法的填充码也可能无法生效。

1. 开始执行初始进程(除非指定CREATE\_SUSPENDED)
2. 在新进程和线程的上下文中完成地址空间的初始化操作（如加载所需的DLL），并还是执行程序入口点

新线程的生命始于内核模式线程启动例程 KiStartUserThread。KiStartUserThread会将线程的IRQL 级别从延迟过程调用(DPC)级别降低至APC 级别，随后调用系统初始线程例程 PspUserThreadStartup。与用户相关的线程启动地址将作为参数传递给该例程PspUserThreadStartup 将执行下列操作。

IRQL 是 Interrupt Request Level（中断请求级别）的缩写，是 Windows 系统中用于控制硬件中断处理程序优先级的机制。在 Windows 中，IRQ（Interrupt Request）是用于指示硬件事件已经发生并请求 CPU 中断处理程序进行响应的机制。为了保证系统的可靠性和稳定性，操作系统必须管理 IRQ，以避免多个硬件设备之间的冲突和竞争，以及确保高优先级的中断能够及时得到处理。

IRQL 是 Windows 系统中对中断处理程序优先级的划分，共分为 33 个级别，从 0 到 31，加上一个特殊级别 DISPATCH\_LEVEL（2），用于表示所有可能的高优先级中断。在任何给定时间，系统都只会处理当前 IRQL 或更高优先级的中断。较低优先级的中断将被阻止，直到当前正在处理的中断完成并返回到较低优先级时才能被处理。这样可以确保高优先级中断的及时响应，提高系统的稳定性和可靠性。

Windows中的IRQL共有33个级别，分别为：

* 0 - Passive Level：被动级别
* 1 - APC Level：异步过程调用级别
* 2 - Dispatch Level：调度级别
* 3-13 - Reserved：保留级别
* 14 - Clock Interrupt Level：时钟中断级别
* 15 - IPI (Interprocessor Interrupt) Level：多处理器中断级别
* 16 - High Level：高级别
* 17-31 - Reserved：保留级别
* 32 - Processor Base Interrupt Level (x64 only)：处理器基本中断级别

其中，级别0到级别14用于单处理器系统，级别15到级别31用于多处理器系统，级别32用于x64架构。级别越低，表示优先级越高，需要更快速地处理中断请求。通常情况下，驱动程序会在IRQL DISPATCH\_LEVEL级别及以上的中断下禁用操作系统的时间分片机制，以保证中断的实时性。可以使用KeRaiseIrql来提升IRQL级别，执行完需要的操作之后再用KeLowerIrql降低IRQL级别，不能长时间处于0以上的IRQL级别，容易导致系统BSOD(蓝屏)或者其他问题。

1. 在x86 系统中，将安装一个异常链。
2. 将IRQL 降低至PASSIVE\_LEVEL(具体来说是降低至“0”这是唯一可运行IRQL用户代码的级别)。
3. 禁止在运行过程中交换主进程令牌。
4. 如果线程在启动过程中(因为任何情况)被关闭，则线程会被终止且不采取任何后续操作。
5. 根据内核模式数据结构中提供的信息，在TEB 中设置区域ID和理想处理器，并检查线程的实际创建是否失败。
6. 调用DbgkCreateThread，由它来检查是否为新进程发送了映像通知。如果未发送且启用了通知，将首次为该进程发送映像通知，并为Ntdll.dll 的映像加载发送通知。

注意 上述操作在这一阶段内进行，而非在首次映射映像时进行的原因在于(内核外调所需的)进程ID在那时还不可用。

1. 上述检查完成后，还需要执行另一个检查来确定该进程是否为被调试者(debuggee)。如果是且尚未发送调试器通知，随后将通过调试对象(debug object，如果存在的话)发送进程创建消息，以便将进程启动调试事件(CREATE\_PROCESS\_DEBUGINFO)发送给相应的调试器进程。随后还将发送一个类似的线程启动调试事件和一个Ntdll.dll 映像加载调试事件。接着 DbgkCreateThread会等待来自调试器(通过ContinueDebugEvent函数给出)的回应。
2. 检查系统是否启用了应用程序预取 (prefetching)功能，如果启用了，则会调用预取器(以及 Superfetch)处理预取指令文件(如果有的话)，并将该进程上次运行前 10s内引用的内存页预取到内存中。

应用程序预取功能（Application Pre-fetching）是Windows操作系统中的一项性能优化功能，旨在通过预先加载和缓存应用程序相关的数据和代码，加速应用程序的启动和运行。

应用程序预取功能分为两个方面：预取和超级预取。预取是指操作系统在用户启动应用程序时，自动预取应用程序相关的数据和代码到内存中。超级预取则更进一步，不仅预取应用程序相关的数据和代码，还预取应用程序可能使用的其他相关数据和代码。

通过应用程序预取功能，Windows可以提高应用程序的启动速度，提高系统的响应速度，减少应用程序闪退的概率，提升用户体验。

1. 检查是否已经在SharedUserDatastructure 中设置了系统范围的Cookie。如果尚未设置，则会根据某些系统信息(例如已处理的中断数、DPC 递交数、页错误、中断时间以及一个随机数)的散列生成这些数据。这种系统范围的 Cookie 可用于指针的内部编码和解码，例如堆管理器用它防范某些类型的非法利用。
2. 对于安全进程(IUM进程)，随后会调用HvlStartSecureThread 将控制转移给安全内核，进而开始执行线程。该函数只会在线程已存在时反馈。
3. 设置初始形式转换(thunk)上下文，以便运行映像加载程序初始化例程(Ntdl.dl中的 LdrInitializeThunk)，以及系统范围内的线程启动存根(stub，Ntdlldll 中RtlUserThreadStart)。这些步骤是通过在原编辑进程的上下文并从系统服务的操作中退出这种方式完成的，借此加载特别修改过的用户上下文。随后 LdrInitializeThunk 例程会初始化加载程序、堆管理器、NLS 表、线程本地存(TLS)和本地存 (FLS)数组以及关键的区域结构。接着它会加载任何必要的DLL 并使用 DLL PROCESS ATTACH函数代码调用 DLL入口点。

该函数返回后，NtContinue会还原新的用户上下文并返回到用户模式。线程可以开始真正执行了。

RtlUserThreadStart会使用实际映像入口点的地址和启动参数调用应用程序的入口点这两个参数应该已经由内核推送至栈中。进行这一系列复杂操作的目的主要有如下两个

1. 可以让Ntdll.dll中的映像加载程序在内部、幕后设置进程，进而顺利运行其他用户模式代码。(否则将不具备堆、线程本地存储之类的东西
2. 让所有线程从一个通用例程开始，即可将其包装到异常处理机制中。这样一旦崩溃，Ntdll.dll 将获知这一情况并可调用 Kernel32.dll 中的未处理异常筛选器。此外也可以在线程从启动例程返回后协调线程的退出操作，并执行各种清理工作。应用程序开发者也可以调用SetUnhandledExceptionFilter 添加自己的未处理异常处理代码。