**Windows引导过程**

# 硬件自检与BIOS

Windows内核中的各个组件和各种机制在起作用以前，必须首先被初始化。此初始化工作是在系统引导时完成的。当用户打开计算机的电源开关时，计算机便开始运行，但操作系统并不立即获得控制权，而是BIOS代码首先获得控制，它执行必要的硬件检测工作，并允许用户通过一些功能键来配置当前系统中的硬件设置，甚至诊断硬件问题，然后才将控制权交给操作系统。

首先介绍Windows获得控制以及加载内核模块的过程，然后描述Windows内核的初始化，最后介绍Windows的会话创建和用户登录过程。

# MBR

在Intel x86系统上，Windows操作系统获得控制首先从硬盘的主引导记录（MBR，Master Boot Record）开始，Windows Setup程序在安装Windows时填充MBR（其他的磁盘管理程序也可能填充MBR）。MBR包含代码和数据，其代码称为引导代码，在系统引导时首先获得控制；MBR中的数据是一张分区表，指定了每个分区在磁盘上的位置和大小，以及分区的类型。当MBR中的引导代码被执行时，它检查分区表中的每一个分区，若找到一个已被标记为可引导的分区（称为引导分区），则将该分区的第一个扇区（称为引导扇区）读到内存中。由于分区表包含了每一个分区的磁盘位置，所以，引导扇区的位置很容易被确定。然后MBR的代码将控制权交给引导扇区中的代码。

# 引导扇区

Windows Setup程序在确定了要将Windows系统安装到哪个分区中以后，除了可能会写入MBR以外，还会写入该分区的引导扇区。所以，严格意义上讲，Windows操作系统的真正入口点应该是引导扇区中的代码。引导分区必须被格式化成Windows所支持的文件系统，典型的文件系统格式是NTFS和FAT，其中NTFS是Windows NT的原生文件系统，而FAT则是从MS-DOS时代继承和发展过来的。

引导扇区中的代码随硬盘文件系统格式的不同而有所不同，其职责是，给Windows提供有关该硬盘上卷的结构和格式方面的信息，并且从该卷的根目录中读入Windows的加载程序，即ntldr文件；然后将控制权交给ntldr的入口函数。为了能够从根目录中读入加载程序，引导扇区包含了能理解文件系统结构和读取文件的代码，这通常只是文件系统极其简单的一部分功能，而并非完整的实现。尽管引导扇区的职责相对简单，但是单个扇区（512B）的代码和数据往往不足以完成其功能，为此，Windows的做法是，让引导扇区中的代码读入其他扇区的数据，然后跳转到下一个扇区的代码区。这样就可以不受单个引导扇区长度的限制，这种做法相当于将第一个引导扇区当做一个加载器（loader），而真正完成引导扇区功能的扇区随后被加载进来并执行。这一过程对于MBR是透明的，从而保持良好的兼容性。

# NTLDR

Intel x86处理器支持实模式和保护模式，在实模式下，处理器的寄存器都是16位的，而且不支持虚拟地址转译，只能访问物理内存空间中最低的1MB内存。计算机系统的BIOS工作在实模式下，并且，当ntldr获得控制权时，处理器仍然在实模式下运行。Ntldr文件实际上是由两部分组成的：第一部分是实模式代码，即首先获得控制的代码区；第二部分是一个标准的Windows可执行二进制文件（PE文件格式，在ntldr中这部分被称为os loader。

Ntldr的实模式代码首先获得控制，它的任务是，完成需在16位模式下执行的初始化工作，例如清除键盘缓冲区，然后为切换到保护模式做好基本的环境准备，之后将处理器切换到保护模式（32位模式）下，这样它就可以访问整个32位地址空间了。最后它将控制权交给os loader。因此，ntldr中的os loader是Windows真正的32位入口程序。

Os loader刚接获控制时，处理器虽然已经工作在保护模式下，但是它的虚拟地址转译机制尚未开启，所以，处理器仍然直接使用物理地址。Os loader首先做的工作是把物理内存管起来，用一个内存描述符（memory descriptor）数组把每一段内存的大小和用途记录下来，然后构造页目录和页表，使得16MB以下的内存能够通过页面映射（paging）机制进行访问，再设置好页目录寄存器，并打开页面映射机制。之后，os loader继续执行其他的初始化工作，包括I/O设备的初始化等。如果它还需要调用BIOS中的服务（比如中断13h、中断15h等），则必须保护好保护模式下的设置，并暂时切换回到实模式，待服务完成以后再切换到保护模式，并恢复设置。

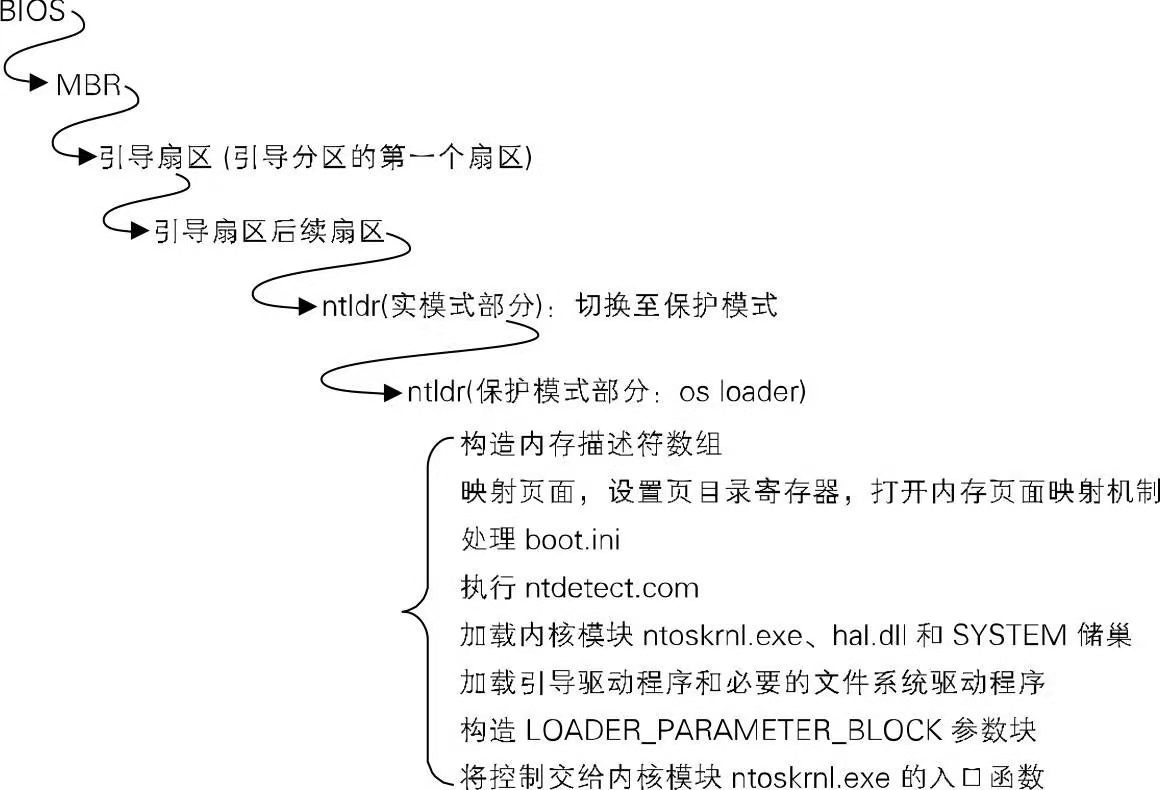
接下来，os loader从系统分区（即引导分区）的根目录下读入boot.ini文件。注意，os loader包含了读取当前文件系统的代码，它能够读取NTFS文件系统的子目录。然后，os loader清除屏幕，并检查在系统分区的根目录下是否有一个有效的hiberfil.sys文件，如果存在的话，这一次引导过程转移到休眠系统的恢复过程。因此，os loader将控制权交给一段能恢复休眠系统的内核代码。

如果当前这次引导不是休眠恢复，那么，os loader解析boot.ini文件，如果该文件中有多个引导选项，则os loader显示一个引导选择菜单；如果boot.ini文件中只包含一个引导选项，那么，此菜单不显示，而是立即应用该引导选项。

Windows的引导选项可以用来指示当前这次引导的各种参数，包括内核模块的文件名称、HAL的文件名称、CPU参数、各种内存参数、调试参数，等等。关于这些引导选项的全面列表和介绍，读者可参考[MSDN-BOOT]。接下来os loader加载并执行NTDETECT.COM程序，这是一个16位实模式程序，它利用系统的BIOS来查询系统的基本设备和配置信息，包括系统的日期和时间、总线的类型、磁盘的信息、输入/输出的接口信息等。这些信息被收集起来，在引导过程的后期被存放到注册表HKLM\HARDWARE\DESCRIPTION键的下面。

然后，os loader加载内核模块映像文件，默认为ntoskrnl.exe，以及HAL映像文件，默认为hal.dll。再加载注册表的SYSTEM储巢，即\WINDOWS\system32\config\system文件。通过检查SYSTEM储巢中的设置信息，它可以知道哪些设备驱动程序必须被加载进来，即被标记为“引导-启动”（SERVICE\_BOOT\_START）的设备驱动程序。然后它加载所有这些被标记为“引导-启动”的设备驱动程序，以及访问系统目录所必需的文件系统驱动程序。注意，在此之前os loader也可以访问系统分区，但并非通过文件系统驱动程序。

至此，引导系统所需的模块（包括内核映像、HAL，以及必要的设备驱动程序）都已经被加载到内存中。而且，在此过程中os loader也已经构造了一个参数块，记录下了这次引导过程中加载器所获得的各种参数信息。参数块的类型为LOADER\_PARAMETER\_BLOCK，Windows内核在初始化过程中将会用到这些参数信息。

**内核初始化**

ntldr构造了一个类型为LOADER\_PARAMETER\_BLOCK的参数块，把与系统初始化相关的参数信息包装到此结构中，然后将控制权传递给内核

模块ntoskrnl.exe的入口函数。因此，内核的初始化从内核模块的入口函数开始，WRK包含了内核初始化过程的绝大多数代码。此入口函数为KiSystemStartup，它是一个汇编函数，位于base\ntos\ke\i386\newsysbg.asm文件中。

内核的初始化主要是内核各个组件的初始化，但由于这些内核组件之间有紧密的耦合关系，所以它们的初始化并不是简单地顺序执行初始化。为了解决在初始化过程中的相互依赖性问题，内核的初始化分两个阶段进行，称为阶段0和阶段1。大多数内核组件的初始化函数相应地带有一个整数参数，以指明一次调用是阶段0初始化还是阶段1初始化，而有些组件的初始化函数通过检查一个全局变量InitializationPhase的值来判断当前处于哪个阶段。

我们首先来看阶段0初始化。阶段0初始化的目的是，将阶段1初始化所要用到的基本数据结构建立起来。在阶段0初始化过程中，中断被禁止，因此处理器可以顺序地执行自己的初始化逻辑。KiSystemStartup函数首先初始化处理器的状态，包括调整它的IDT，初始化TSS（Task State Segment），以及构造PCR（Processor Control Region）。然后，调用HalInitializeProcessor函数，为当前处理器初始化其HAL中的PCR和处理器间中断向量；接着调用KiInitializeKernel函数，执行内核初始化。最后，当前线程蜕变成一个空闲线程。

因此，KiInitializeKernel函数是实际执行内核初始化的函数，其代码位于base\ntos\ke\i386\kernlini.c文件中。它的职责是：初始化内核数据结构，初始化空闲线程和进程对象，初始化PCR，然后调用执行体初始化函数ExpInitializeExecutive，最后返回。ExpInitializeExecutive函数的代码位于base\ntos\init\initos.c文件中，它调用HalInitSystem以初始化HAL，调用ExInitSystem以初始化执行体组件的各种数据结构，调用MmInitSystem以初始化内存管理器和内存池，调用ObInitSystem以初始化对象管理器，调用SeInitSystem以初始化安全子系统，调用PsInitSystem以初始化进程/线程管理器，调用PpInitSystem以初始化即插即用管理器，调用DbgkInitialize以初始化调试子系统。通常，这些执行体组件的阶段0初始化相对简单，以初始化组件的内部状态为主，因而经过阶段0初始化以后仅可以提供最基本的服务。

阶段0初始化完成以后，系统的线程调度器开始工作。特别值得一提的是，进程管理器在阶段0初始化过程（PspInitPhase0）中，除了初始化其内部的状态变量，它也为初始进程创建一个进程对象，并将其命名为“Idle”。另外，它还创建了“System”进程，以及一个系统线程，此系统线程的开始例程为Phase1Initialization函数。然而，此线程并不立即被执行，因为在阶段0初始化过程中中断是禁止的。

KiInitializeKernel函数返回以后，KiSystemStartup启动中断，将当前的中断请求级别（IRQL，Interrupt Request Level）降低到DISPATCH\_LEVEL，从而允许线程调度器选择新的线程。因此，阶段0初始化完成以后，阶段1初始化例程Phase1Initialization得以运行。注意，如果仔细观察KiSystemStartup汇编函数的代码，可以发现它在跳转到空闲循环KiIdleLoop以前，要经过一个屏障KiBarrierWait。此屏障对于系统的第一个处理器并不起作用，而仅对后续的处理器起作用。后面我们在讨论其他处理器的初始化过程时将会看到此屏障的影响。

现在我们知道，阶段1初始化是在System进程的一个系统线程中运行的。Phase1Initialization函数调用Phase1InitializationDiscard执行阶段1初始化，然后调用MmZeroPageThread函数，从而此线程蜕变成内存管理器的零页面线程（内存管理器中负责在后台将空闲页面清零的辅助线程，参见4.5节关于物理内存管理的描述）。在介绍阶段1初始化的逻辑以前，我们先来看看在多处理器或多核系统上的初始化过程。如图2.16所示，当第一个处理器（也称为引导处理器，或0号处理器，或P0）执行到阶段1初始化时，在Phase1InitializationDiscard的一个特定点上，它调用KeStartAllProcessors函数，以启动其他的处理器。这些处理器从KiSystemStartup函数开始执行。

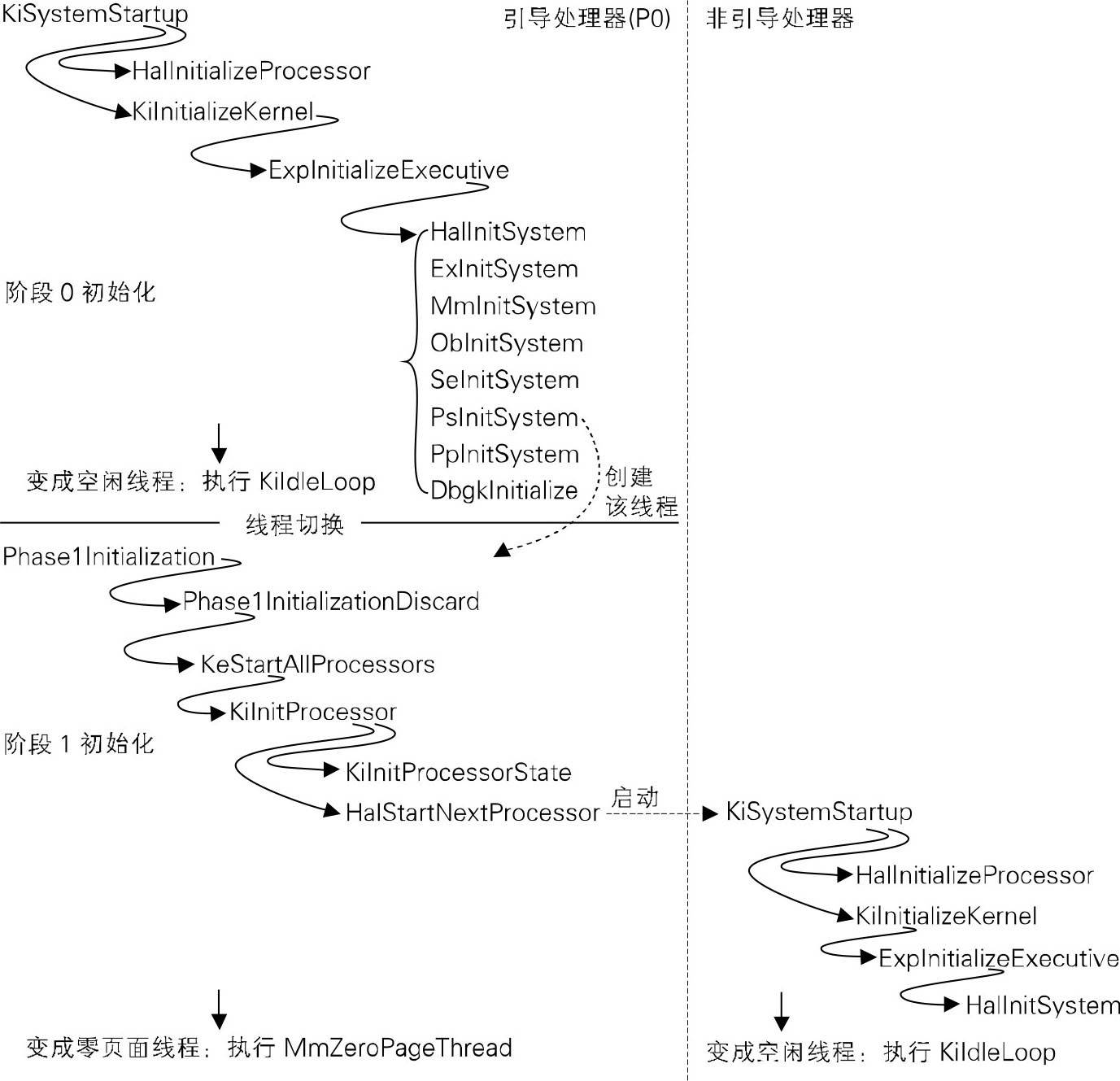


图2.16 多处理器系统的内核初始化过程

KeStartAllProcessors函数的代码位于base\ntos\ke\i386\allproc.c文件中，它设置好前文提到的位于KiSystemStartup函数结束处的屏障KiBarrierWait，然后依次调用KiInitProcessor函数来启动每个处理器。KiInitProcessor函数为每个处理器构造一份状态信息（KPROCESSOR\_STATE结构），然后调用HalStartNextProcessor函数启动该处理器。处理器状态信息是通过调用KiInitProcessorState函数来构造的，在该函数的代码中，我们可以看到，新处理器的起始指令地址（即KPROCESSOR\_STATE结构的ContextFrame.Eip成员）为KiSystemStartup例程。KeStartAllProcessors函数在启动了其他所有处理器以后，设置好每个处理器的控制块（PRCB）中的相关信息，并同步所有处理器的性能计数器，最后打开屏障KiBarrierWait，允许其他处理器进入空闲线程循环。这意味着，此后其他的处理器可以按照线程调度器的选择来运行比空闲线程优先级更高的线程了。所以，屏障KiBarrierWait可以看成是引导处理器对于非引导处理器的一个约束，在放任它们参与线程调度以前执行必要的初始处理。

非引导处理器的初始化过程虽然也执行KiSystemStartup函数，但其执行逻辑相对要简单很多。同样地，KiSystemStartup也调用HalInitializeProcessor和KiInitializeKernel函数来执行HAL和内核部分的初始化，并且，KiInitializeKernel函数也调用ExpInitializeExecutive，但是，ExpInitializeExecutive函数仅仅简单地调用HalInitSystem而已。在这些函数中，Number参数代表了当前处理器的序号，如果为0，则表明这是引导处理器，否则为非引导处理器。这样的逻辑不难理解：有些初始化过程，特别是内核执行体组件（比如内存管理器和对象管理器）的初始化例程，它们的处理逻辑针对整个系统，而并非某个处理器，所以只需在引导处理器上运行一次即可；而另外有些初始化过程，则是针对单个处理器的初始化（包括设置处理器的中断描述符表、PRCB的状态等），因而需要针对每个处理器都执行一次。在阅读这些函数的代码时，我们可以清楚地看到这一基本差别。

前面我们提到了阶段1初始化是在System进程的一个系统线程中进行的，它在一个恰当的点上调用KeStartAllProcessors函数以启动其他的处理器。阶段1初始化是内核的完全初始化，它占据了系统初始化过程中相当一部分时间。在Windows 2000系统的引导过程中，可以在屏幕上看到一个进度条，它指示了阶段1初始化的进度；而在Windows XP/Server 2003及以后的Windows系统中，则是一个闪烁的Windows标志图案出现在屏幕上，而并非进度条。如前所述，阶段1初始化的主函数为Phase1InitializationDiscard，其代码位于base\ntos\init\initos.c文件的933～1865行。从该函数的代码可以看出，阶段1的初始化进度仍然有一个百分比估计。该函数按以下步骤完成其主要功能：

(1) 设置全局变量InitializationPhase为1，标志着当前这次系统引导过程进入内核的阶段1初始化。

(2) 调用HalInitSystem函数，执行HAL的阶段1初始化。

(3) 初始化图形引导驱动程序，显示Windows启动屏幕，设置引导进度条范围(0,100)。虽然在Windows Server 2003中已不再显示进度条，但进度指示逻辑仍然保留。

(4) 调用PoInitSystem，完成电源管理器的阶段0初始化。

(5) 调用HalQueryRealTimeClock，初始化系统时间，这一步必须在HAL的阶段1初始化以后才能进行。再调用KeSetSystemTime以设置内核的系统时间。

(6) 调用KeStartAllProcessors，启动并初始化其他的处理器。启动了这些辅助处理器以后，重新调用HalQueryRealTimeClock以初始化系统时间，并调用KeSetSystemTime设置内核的系统时间。

(7) 接下来，调用ObInitSystem，完成对象管理器的阶段1初始化；调用ExInitSystem，完成执行体组件的阶段1初始化；调用KeInitSystem，完成微内核部分的初始化；调用KdInitSystem，完成内核调试器的阶段1初始化；调用SeInitSystem，完成安全子系统的阶段1初始化。

(8) 调用InbvUpdateProgressBar函数，更新进度条至10%。

(9) 创建符号链接“\SystemRoot”。

(10) 调用MmInitSystem，完成内存管理器的阶段1初始化。

(11) 将国家语言支持（NLS）表映射到系统空间中，然后重置翻译表。

(12) 调用CcInitializeCacheManager，初始化缓存管理器。

(13) 调用CmInitSystem1，初始化配置管理器，建立起基本的注册表对象操作，把引导时获得的数据转变成注册表格式。现在，HKLM\SYSTEM和HKLM\HARDWARE已可以使用，这一步必须在I/O子系统初始化之前完成。

(14) 调用CcPfInitializePrefetcher，初始化内核中的预取器（prefetcher）。

(15) 进度条前进到15%。

(16) 调用FsRtlInitSystem，初始化文件系统支持库（FsRtl）。

(17) 调用KdDebuggerInitialize1，即kdcom.dll中的调试器初始化。

(18) 调用PpInitSystem，初始化即插即用管理器。

(19) 进度条前进到20%。

(20) 调用LpcInitSystem，初始化LPC子系统。

(21) 现在系统时间已经正常运行，调用ExInitSystemPhase2，再次初始化执行体组件。

(22) 进度条更新范围设置为25～75%。

(23) 调用IoInitSystem，初始化I/O系统。这是系统引导过程中较为复杂的一部分，将占据进度条50%的范围。IoInitSystem函数所做的工作包括：I/O子系统中的状态变量初始化、驱动程序对象类型和设备对象类型的创建、加载“引导-启动”类型的驱动程序、加载“系统-启动”类型的驱动程序，以及WMI初始化等。详细的执行过程，请参考IoInitSystem函数的代码，位于base\ntos\io\iomgr\ioinit.c文件中。

(24) 进度条更新范围恢复到0～100%。

(25) 再次调用MmInitSystem，将当前已加载内核模块中的PAGE段标记为“可换页”。

(26) 进度条前进到80%。

(27) 调用PoInitSystem，完成电源管理器的阶段1初始化。

(28) 调用PsInitSystem，完成进程管理器的阶段1初始化。

(29) 进度条前进到85%。

(30) 调用SeRmInitPhase1，执行安全引用监视器（SRM）的阶段1初始化，包括创建安全引用监视器的命令服务线程。该线程创建一个名为“引用监视器命令端口”的LPC端口，以接收来自lsass.exe进程的命令。参见SeRmInitPhase1函数的代码。

(31) 进度条前进到90%。

(32) 创建会话管理器子系统进程（smss.exe）。首先准备各种参数信息（RTL\_USER\_PROCESS\_PARAMETERS结构），包括环境参数字符串，然后调用RtlCreateUserProcess函数以创建smss进程。

(33) 进度条前进到100%。

(34) 最后，调用ZwWaitForSingleObject函数，在smss进程的句柄上等待，超时值设置为5s。如果等待成功，则意味着在5s内smss进程退出了，于是调用“KeBugCheck(SESSION5\_INITIALIZATION\_FAILED)”，系统崩溃；若等待超时，则认为会话管理器已经正常运行，于是阶段1初始化完成，当前线程蜕变成零页面线程。

阶段1初始化完成以后，内核已经完全初始化，执行体的各个组件进入正常运行状态。但作为一个操作系统，Windows的引导过程尚未完成，仅仅内核正常工作还不够，系统必须让应用程序也能够运行起来。接下来的引导工作由刚刚启动起来的smss进程继续进行。

# 建立用户登录会话

Windows内核在阶段1初始化的最后，启动了一个用户模式进程——会话管理器子系统（smss）。Smss进程是Windows操作系统的关键组成部分，它尽管是一个用户模式进程，但有其特殊性：首先，它是可信的，这意味着它可以做一些其他用户进程无法做的事情，比如创建安全令牌；其次，它直接建立在Windows内核的基础上，只使用Windows内核提供的系统服务，而不依赖于任何一个环境子系统。这不难理解，因为smss启动时，Windows子系统尚未启动，而且，启动Windows子系统本身正是smss的任务之一。

Smss进程启动以后，继续完成系统引导过程。Smss做的工作有相当一部分依赖于注册表中的设置。在内核的阶段1初始化过程中，配置管理器已经初始化，但是直到smss进程启动起来，只有SYSTEM储巢已被加载到内存中。加载其他的储巢也是smss的职责，它通过调用系统服务NtInitializeRegistry来初始化注册表。在NtInitializeRegistry系统服务中，除了用户轮廓储巢以外的所有其他储巢，均会被加载到系统中，并且注册表树也将建立起来。实际上，加载这些储巢的工作由CmpInitializeHiveList函数来完成，它仅仅加载6个储巢（包括SYSTEM储巢），正好是表2.6中的前6个储巢。

Smss在注册表中的主键是HKLM\SYSTEM\CurrentControlSet\Control\Session Manager，用于指示smss在系统初始化阶段做一些必要的工作。图2.17显示了在Windows Server 2003 SP1系统中该键的一个典型设置。

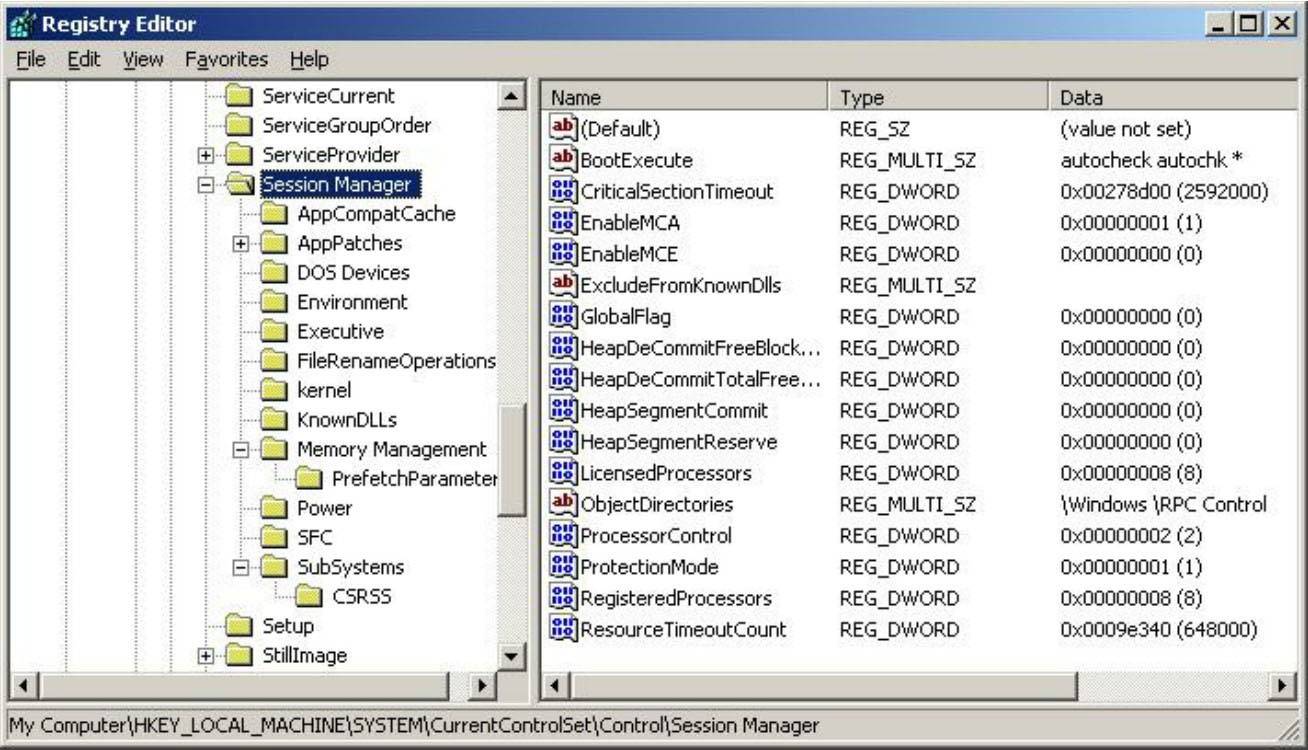


图2.17 注册表Session Manager键的内容

按照Session Manager键中的指示，smss主线程完成以下事项：

· 运行在启动时执行的程序，这些程序由BootExecute值指定。

· 执行启动时的文件删除或重命名任务，这由FileRenameOperations子键来指定。

· 打开已知的DLL，并且在对象管理器名字空间的\KnownDlls目录下创建相应的内存区对象，这些已知DLL的列表位于KnownDLLs子键中。已知DLL是系统全局共享DLL，包括子系统DLL等，各种应用程序通常都会用到这些DLL。

· 创建页面文件，页面文件列表由Memory Management子键中的PagingFiles值指定。

· 建立系统的全局环境变量，这些环境变量由Environment键下的值指定。

· 加载Windows子系统的内核模式模块（win32k.sys），这是通过系统服务NtSetSystemInformation来完成的，该函数也会调用win32k.sys中的初始化例程（即入口函数）。子系统内核模块的文件路径由SubSytstems子键的Kmode值指定。关于Windows子系统的初始化过程，请参考9.2节。

· 启动Windows子系统进程（csrss.exe）。子系统进程的命令行字符串由SubSystems子键的Windows值指定。

如果利用Microsoft提供的符号信息来调试WRK系统，则可以观察到以上这些行为均在smss主模块（smss.exe）的SmpLoadDataFromRegistry函数中完成。由此也可以看出，Windows子系统作为会话（session）的一部分，它的实例由smss来启动。Smss除了依据注册表中的设置来完成必要的系统引导工作以外，还执行以下的步骤，以进一步提供多会话和本地登录服务：

· 创建LPC端口对象（\SmApiPort），以接收“加载子系统”或“创建会话”的请求。

· 启动登录进程（winlogon.exe），登录进程将会承担起与用户登录有关的事项。

Smss的主线程在完成了以上描述的初始化工作以后，将在csrss进程和winlogon进程的句柄上等待。一旦等待成功，则意味着这两个进程中至少有一个退出了，于是系统崩溃。Windows操作系统依赖于这两个进程，所以，它们也是保持Windows操作系统正常运行不可缺少的组成部分。

接下来引导过程转到了winlogon进程。它的职责包括：

· 创建初始的窗口站（WinSta0），并且为该窗口站创建一个桌面线程和RIT（Raw Input Thread）以便接收标准输入。

· 创建登录桌面和默认桌面。登录桌面只有winlogon进程才可以访问，因而也称为winlogon桌面；而默认桌面允许其他进程访问。因此，当登录桌面活动时，其他进程无法访问与该桌面关联的代码或数据。Windows用这种方式来保护与口令相关的操作，以及锁定桌面或解除桌面锁定这样的安全操作。

· 启动服务控制管理器（SCM，Service Control Manager）进程（services.exe）。在启动Windows服务的过程中，会有更多的窗口站被创建。SCM进程加载所有“自动-启动”类型的服务和设备驱动程序。

· 启动本地安全权威子系统（lsass）进程。然后与它建立一个LPC连接（LsaAuthenticationPort端口），以便在登录、注销和口令操作过程中交换信息。

它们是Windows窗口管理中的重要组成部分，由Windows子系统内核模块win32k.sys实现。Winlogon的登录是通过一种称为GINA（图形化识别和认证，Graphical Identification and Authentication）的可扩展机制来完成的，例如，默认的GINA为msgina，可显示标准的Windows登录对话框，指示用户输入用户名和口令。Winlogon对于Ctrl+Alt+Del按键序列做了特殊的保护，以防止其他程序截取此按键序列。第三方厂商可以利用像指纹或虹膜这样的生物特征来识别用户，然后从一个加密的数据库中提取出他们的口令，这样就不需要用户再输入口令了。

当winlogon通过GINA获得了用户名和口令以后，它首先调用lsass函数LsaLookupAuthenticationPackage以获得一个认证包的句柄，然后调用LsaLogonUser将登录信息传递给认证包。一旦认证包认证了当前用户，则winlogon继续该用户的登录过程；否则认证失败。因此，认证过程是由lsass来完成的。

在登录过程的最后，winlogon检查注册表HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit的值，并创建一个进程来运行该值字符串。该值串的默认值为userinit.exe程序的路径。Userinit进程加载当前登录用户的轮廓，然后检查HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell的值，并创建一个进程来运行该值字符串；如果该值不存在，则运行HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell的值，其默认值为explorer.exe。然后，userinit进程退出。由于当前登录会话的Shell程序（explorer.exe）已经启动，因此用户可以在桌面上操作了。

至此，引导过程结束，用户登录到系统中，并可通过explorer.exe程序的用户界面操作系统中的资源，例如文件系统中的目录和文件；也可以启动各种应用程序。图2.18显示了从会话管理器启动一直到Shell程序启动之间发生的事项。

在系统引导过程中，有多个进程被创建，包括smss、csrss、winlogon、SCM（services.exe）、lsass、userinit（登录完成后自动退出）、explorer等。这些进程都是操作系统的一部分，而且大多数还是可信的。实际上，在系统启动以后，当用户开始在Shell程序中操作时，他们通常可以看到更多的进程，这其中有些进程是由SCM启动并初始化的。在Windows中，SCM提供了在系统启动时启动进程的机制，这些进程被称为服务（service）进程，它们类似于UNIX中的守护进程。例如，Web服务器或者数据库服务器显然应该以这样的方式来启动，因为无论是否有人登录到系统中，它们都必须被启动。在每个服务进程中，它可以宿纳一个或多个Windows服务（Windows service）。

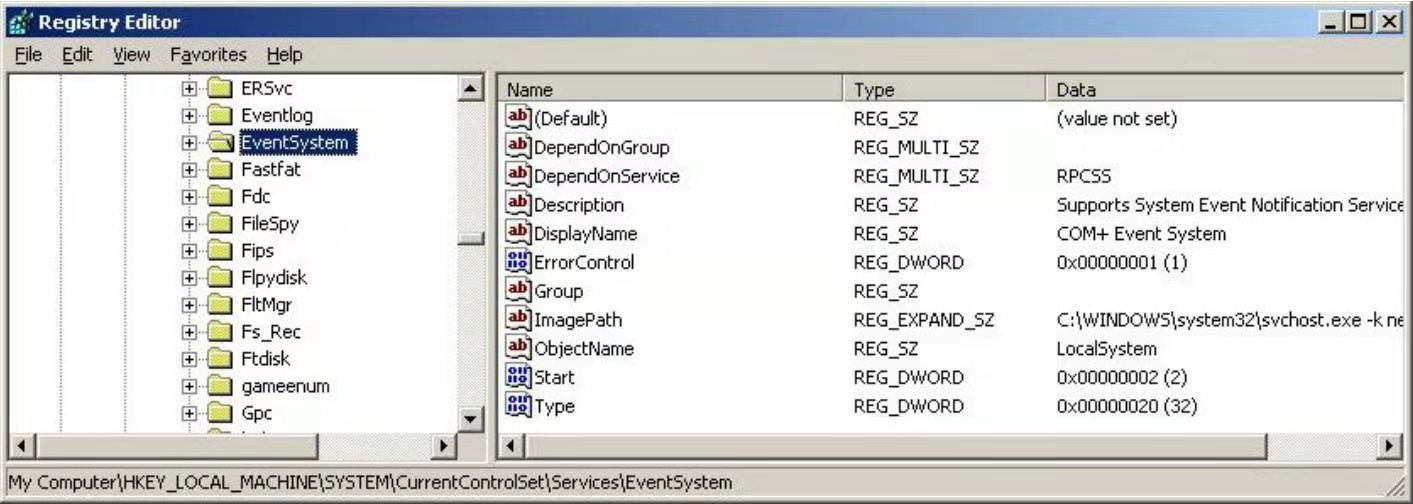


图2.18 Windows会话创建和用户登录过程

Windows服务既可以是用户模式应用程序，也可以是内核模式驱动程序。但只要它们被标记了“自动-启动”（SERVICE\_AUTO\_START），在引导过程的后期阶段就会被启动或加载到系统中。实际上，Windows的许多重要组件，包括网络服务、存储管理服务等，都是以Windows服务的形式来配置和启动的。