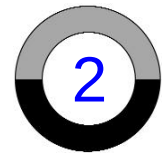


Windows 2000/XP的体系结构

第 2 章

Windows 2000/XP的体系结构

Windows 2000/XP经过十多年的发展已经从最初功能十分有限的32位操作系统演化成了现在的界面友好、管理方便、功能强大的操作系统家族，它之所以有这样大的发展，除了微软公司强大的商业推动力量之外，Windows 2000/XP本身的技术因素也是重要的原因，尤其是它自身的体系结构所具有的可扩充性、可执行性、鲁棒性、兼容性和高效性，正是这些特性让 Windows 2000/XP不断得以进步。

本章将从操作系统设计的角度详细分析一下 Windows 2000/XP所具有的基本体系结构和运行机理。我们将首先解决有关操作系统设计的理论问题，然后再对 Windows 2000/XP所具有的特殊体系结构进行一些必要的介绍和分析。

2.1 操作系统的设计

设计操作系统并不像解一道数学习题那样有着既定的证明思路和规范化的推导过程，这是一项系统工程，有着十分复杂的过程。操作系统作为一个软件系统来说是以庞大而复杂著称的。以IBM公司的OS/360系统为例，它由4000个模块组成，共约100万条指令，花费5000人年，经费达数亿美元。但每个版本都仍然隐藏着无数的错误。其负责人 Brooks在描述OS/360研制过程中的困难和混乱时曾经说过："……巨兽在泥潭中做垂死挣扎，挣扎得越猛，泥浆就沾得越多。最后没有一个野兽能逃脱淹没在泥潭中的命运，……程序设计就像是这样一个泥潭。……一批批程序员在泥潭中挣扎……没有人料到问题会这样棘手"。这就是在20世纪60年代出现的软件危机现象，OS/360研制开发所陷入的困境，推动了软件工程方法与技术的诞生。毫无疑问，今天操作系统的开发应遵循软件工程的原则和方法。

2.1.1 操作系统的设计目标

一个高质量的操作系统应具有可靠性、高效性、易维护性、可移植性、安全性、可适应性和简明性等特征。

1. 可靠性

可靠性包括了正确性和健壮性。

操作系统是计算机系统中最基本、最重要的软件，随着计算机应用范围的日益扩大，对操作系统的可靠性要求也越来越高。无可靠性，将严重影响使用效果。例如，用于导弹控制的操作系

统必须绝对可靠，否则造成的后果可能不堪设想。

影响操作系统正确性的因素有很多，最主要的因素是并发、共享以及随之带来的不确定性。并发使得系统中各条指令流的执行次序可以任意交叉；共享导致对于系统资源的竞争，使不同的指令执行序列之间产生直接和间接的相互制约；以上两点又引起系统的不确定，这种随机性要求系统能动态地应付随时发生的各种内部和外部事件。因此，需要对于操作系统的结构进行研究。一个设计良好的操作系统不仅应当是正确的，而且其正确性应当是可验证的。

可靠性除了正确性这一基本要求外，还应包括能在预期的环境条件下完成所期望的功能的能力以及在发生硬件故障或某种意外的环境下，操作系统仍能做出适当处理，避免造成严重损失的鲁棒性要求。

2. 高效性

对于支持多道程序设计的操作系统来说，其根本目标是提高系统中各种资源的利用率，即提高系统的运行效率。一个计算机系统在其运行过程中或处于目态，或处于管态。处于目态时为用户服务；处于管态时可能为用户服务（如为进程打开文件或完成打印工作），也可能做系统维护工作（如进程切换、调度页面、检测死锁等）。

假设一个计算机系统，在一段时间 T之内，目态下运行程序所用的时间为 T_u，管态下运行程序为用户服务所用的时间为 T_{su}，管态下运行程序做系统管理工作所用的时间为 T_{sm}，则可定义系统运行效率η为：

$$\eta = \frac{T_u + T_{su}}{T_u + T_{su} + T_{sm}} \times 100\%$$

显然，η越大，系统运行效率越高。为了提高系统运行效率，应当尽量减少用于系统管理所需要的时间T_{sm}。我们亦把T_{sm}称为系统开销（时间开销）。

3. 易维护性

易维护性包括易读性、易扩充性、易剪裁性、易修改性等。一个实际的操作系统投入运行后，有时希望增加新的功能，删去不需要的功能，或修改在运行过程中所发现的错误，为了对系统实施增、删、改等维护操作，必须首先了解系统，为此要求操作系统具有良好的可读性。

4. 可移植性

可移植性是指把一个程序从一个计算机系统环境中移到另一个计算机系统环境中并能正常运行的特性。操作系统的开发是一项非常庞大的工程。为了避免重复工作，缩短软件研制周期，现代操作系统设计都将可移植性作为一个重要的目标。而影响可移植性的最大因素就是和机器有关的硬件部分的处理。为了便于将操作系统由一个计算环境迁移到另外一个计算环境中，应当使操作系统程序中与硬件相关的部分相对独立，并且位于操作系统程序的底层，移植时只需修改这一部分。

5. 安全性

操作系统的安全性是计算机软件系统安全性的基础，它为用户数据保护提供了最基本的机制。这一点在网络环境中显得更为重要。

6. 可适应性

可适应性指的是一种特定计算机系统环境中的软件对于另一种计算机系统环境的适应能力。

研制一个大型软件的费用十分昂贵，而使用要求又在不断变化，经常需要对系统做些修改，以适应环境和要求的变化，如果一个系统没有可适应性，它将是一个僵死的系统，是无生命力的。

7. 简明性

无简明性，开发人员就无法了解一个大型程序的设计目的和细节。

具有简明性、可靠性、可适应性的系统称为可维护的系统，我们称之为易管理的。可适应性和可移植性我们合称为灵活性。

2.1.2 操作系统的设计阶段

设计一个操作系统一般可分为三个阶段：功能设计、算法设计和结构设计。

操作系统的三个设计阶段是互相渗透的，因此不能截然分开它们。其总的目的是要求能够设计出一个具有好结构、高功效，又兼备所需功能的系统。

1. 功能设计

功能设计指的是根据系统的设计目标和使用要求，确定所设计的操作系统应具备哪些功能，以及操作系统的类型。

2. 算法设计

算法设计是根据计算机的性能和操作系统的功能，选择和设计满足系统功能的算法和策略，并分析和估算其效能。

3. 结构设计

结构设计则是按照系统的功能和特性要求，选择合适的结构，使用相应方法将系统逐步地分解、抽象和综合，使操作系统结构清晰、简明、可靠、易读、易改，而且使用方便、适应性强。

2.1.3 操作系统的结构问题

1. 程序结构

程序的可靠性和程序结构密切相关。所谓程序结构有两层含义：一是指程序的整体结构，即由程序的成分构造程序的方式，如 PASCAL语言程序是分程序结构， EUCLID语言程序是模块结构等；二是指程序的局部结构，即程序的数据结构和控制结构。

我们的目标是设计一个能正确实现功能要求的程序，也就是说，要设计一个可靠的程序，运行的结果能正确地反映设计要求。为此，我们必须构造出一个结构良好的程序。所谓程序的结构良好，指的是程序的结构清晰、易读、易维护、可移植，当然这样的程序也要易验证、易调试和易修改。

结构化程序设计就是为了使程序有一个合理的结构，以便于保证和验证其正确性而规定的一套如何进行程序设计的准则和方法。按照这样一套准则和方法设计出来的程序是结构化程序。

模块化是一种结构化程序设计方法，它指的是把一个程序按功能分解成若干个彼此具有一定独立性，同时也具有一定联系的组成部分，这些组成部分就称作“ 模块 ”，每个程序由一个或多个模块组成。对大型程序来说，模块化是一种必然趋势。近年来的研究表明，在分析的基础上设计出一组“ 基本模块 ”和一组“ 构成法则 ”，然后由这些基本模块出发，通过有关的组装规则，组装成所需的程序。如果这些模块是正确的，而且这些组装规则也是正确的，则得到的程序也是

易于验证其正确性的。

2. 软件结构

软件结构通常是指大型程序系统的结构，与小规模程序结构具有本质的差别，后者主要研究程序的结构良好性、易读性、易验证性等。大型程序是由小规模程序组成，因此要研究由小规模程序组成大型程序。前者是局部结构，后者的结构只有在能保证小规模程序正确性之后，才能使组成的大型程序的正确性有保证。本书讨论的是如何把小规模程序组成大型程序的问题，这些问题在研究小规模程序的结构设计时是不存在的，至少也是不严重的。

一个大型程序系统总是由一些模块组成，模块之间的接口指的是一个模块中的程序访问另一个模块内的程序或数据的方式，也可以说，接口就是指模块间传递和交换信息的方法。设计大型程序系统时，对于接口必须十分重视。

3. 操作系统体系结构

操作系统是一种大型软件。为了研制操作系统，必须分析它的体系结构。也就是要弄清楚如何把这一大型软件划分成若干较小的模块以及这些模块间有着怎样的接口。在操作系统中，有些模块需要使用另一些模块内的数据，而系统的某些功能又需要若干模块协同工作来实现。

例如，有两个模块，一个是命令接收模块A，另一个是命令处理模块B。当命令接收模块A接收到来自操作员的命令后就要把命令交给命令处理模块B去分析执行，于是，模块A和模块B之间就有接口。在这个例子中，模块A要调用模块B的命令分析程序，并要将接收到的命令传送给模块B的命令分析程序。

操作系统还是一个具有并发特性的大型程序，模块间的接口是相当复杂的，信息交换也是十分频繁的，因而对结构的研究就显得更加重要了。

在操作系统的早期设计中，由于计算机体系结构还比较简单，系统规模也比较小，逻辑关系简单，因此人们关心的是系统的功能和效率。随着计算机结构的复杂化，应用范围的不断扩大，使用要求也不断提高，不仅要求有较强的系统功能，而且要求有较强的可适应性和可靠性。后来，又提出了容错的概念。从而使人们日益认识到：体系结构直接影响到整个系统的性能。因此，近年来人们普遍重视操作系统的体系结构和结构设计方法的研究，它已成为软件工程界的一个重要的研究领域。

2.1.4　操作系统的结构设计

在操作系统的发展过程中，产生了多种多样的体系结构，几乎每一个操作系统在结构上都有自己的特点，但从整体上看，到目前为止，大致可划分为四种类型，并且这种划分又和操作系统的发展阶段相一致。但是，这并不意味着后出现的结构已经取代了早期的结构。目前这四种结构的系统都在实际使用中，各有其适用范围。

要说明的是，对于一个实际的现代操作系统来说，它们的体系结构往往比较难划分到某一个单一的类别之下，设计这些系统的工程师们在考虑实际的性能问题时往往会做出各种各样的权衡，将不同的体系结构整合起来，充分吸取它们的优点。

1. 模块组合结构

操作系统是一个有多种功能的系统程序，可以看成是一个整体模块，也可看成是由若干个模块按一定的结构方式组成的。

在1968年软件工程出现以前的早期操作系统（如IBM的操作系统）以及目前的一些小型操作系统（如DOS操作系统）均属此种类型。系统中的模块不是根据程序和数据本身的特性而是根据它们完成的功能来划分，数据基本上作为全程量使用。在系统内部，不同模块的程序之间可以不加控制地互相调用和转移，信息的传递方式也可以根据需要随意约定，因而造成模块间的循环调用，如图2-1。我们把这种操作系统的结构称之为模块组合结构。它的主要优点是：结构紧密、接口简单直接、系统效率较高。它的缺点有以下三点：

1) 模块间转接随便，各模块互相牵连，独立性差，系统结构不清晰。

2) 数据基本上作为全程量处理，系统内所有模块的任一程序均可对其进行存取和修改，从而造成了各模块间有着很隐蔽的关系。要更换一个模块或修改一个模块都比较困难，因为要弄清各模块间的接口，按当初设计时随意约定的格式来传递信息，这是一件相当复杂的事。

3) 由于模块组合结构常以大型表格为中心，因此为保证数据完整性，往往采用全局关中断办法，从而限制了系统的并发性。系统中实际存在的并发也未能抽象出明确的概念，缺乏规格的描述方法。所以，这种结构的可适应性比较差。它只适用于模块比较小、使用环境比较稳定但要求效率比较高的系统。

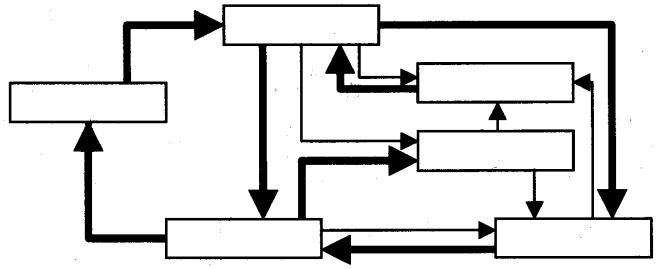


图2-1　模块组合结构

随着系统规模的不断增大，采用这种结构构造的系统的复杂性迅速增长，以致使人们难以驾驭，这就促使人们去寻求新的结构概念和新的结构设计方法。

2. 层次结构

显然，要清除模块接口法的缺点就必须减少各模块之间毫无规则地相互调用、相互依赖的关系，特别是清除循环现象。层次结构设计方法正是从这点出发，它力求使模块间调用的无序性变为有序性。因此所谓层次结构设计方法，就是把操作系统的所有功能模块按功能的调用次序分别排列成若干层，各层之间的模块只能是单向依赖或单向调用（如只允许上层或外层模块调用下层或内层模块）关系。这样，不但操作系统的结构清晰，而且不构成循环，　　THE系统就是E.W.Dijkstra和他的学生按照层次模型在荷兰的Eimdhoven技术学院开发的，如图2-2。

在一个层次结构的操作系统中，如果不仅各层之间是单向调用的，而且每一层中的同层模块之间不存在互相调用的关系，则称这种层次结构关系为全序的层次关系。但是，在实际的大型操

作系统中，要按全序的层次关系来设计几乎是不可能的，往往无法完全避免循环现象，此时我们应使系统中的循环尽量减少。例如，我们可以让各层之间的模块是单向调用的，但允许同层之间的模块互相调用，可以有循环调用现象，这种层次结构关系称为半序的层次结构。

5	操作员
4	用户程序
3	输入/输出管理
2	操作员—进程通信
1	内存和磁盘管理
0	处理器分配和多道程序

图2-2　THE操作系统的层次结构

层次结构的优点是：它既具有模块组合结构的优点——把复杂的整体问题分解成若干个比较简单的相对独立的成分，即把整体问题局部化，使得一个复杂的操作系统分解成许多功能单一的模块；同时它又具有模块组合结构不具有的优点，即各模块之间的组织结构和依赖关系清晰明了。这不但增加了系统的可读性和可适应性，而且还使操作系统的每一步都建立在可靠的基础上。因为层次结构是单向依赖的，所以上一层各模块所提供的功能（以及资源）是建立在下一层的基础上的。或者说上一层功能是下一层的扩充和延续。最内层是硬件基础——裸机，裸机的外层是操作系统的最下面（或内层）的第一层。按照分层虚拟机的观点，每加上一层软件就构成了一个比原来机器功能更强的虚拟机，也就是说进行了一次功能扩充。而操作系统的第一层是在裸机基础上进行的第一次扩充后形成的虚拟机，以后每增加一层软件就是在原机器上的又一次扩充，又成为一个新的虚拟机。因此，只要下层的各模块的设计是正确的，就为上层功能模块的设计提供了可靠基础，从而增加了系统的可靠性。

这种结构的优点还在于增加或替换掉一层可以不影响其他层次，便于修改、扩充。

层次结构的操作系统的各功能模块应放在哪一层，系统一共应有多少层，这是一个很自然会提出的问题。但对这些问题通常并无一成不变的规律可循，必须要依据总体功能设计和结构设计中的功能图和数据流图进行分层，大致的分层原则如下：

1) 为了增加操作系统的可适应性，并且便于将操作系统移植到其他机器上，必须把与机器特点紧密相关的软件（如中断处理、输入输出管理等）放在紧靠硬件的最低层。这样，经过这一层软件扩充后的虚拟机，硬件的特性就被隐藏起来了，方便了操作系统的移植。为了便于修改移植，它把与硬件有关和与硬件无关的模块截然分开，并把与硬件有关的 BIOS（管理输入输出设备）放在最内层。所以当硬件环境改变时只需要修改这一层模块就可以了。

2) 对于一个计算机系统来说，往往具有多种操作方式（例如，既可在前台处理分时作业，又可在后台以批处理方式运行作业，也可进行实时控制）。为了便于操作系统从一种操作方式转变到另一种操作方式，通常把多种操作方式共同使用的基本部分放在内层，而把随着这些操作方式而改变的部分放在外层（例如，批作业调度程序和联机作业调度程序、键盘命令解释程序和作业控制语言解释程序等），这样改变操作方式时仅需改变外层，内层部分保持不变。

3) 当前操作系统的设计都是基于进程的概念，进程是操作系统的基本成分。为了给进程的活动提供必要的环境和条件，必须要有一部分软件——系统调用的各功能，来为进程提供服务，通常这些功能模块（各系统调用功能）构成操作系统内核，放在系统的内层。内层中又分为多个层次，通常将各层均要调用的那些功能放在更内层。

3. 虚拟机结构

这是纯粹以虚拟机的观点构造的操作系统体系结构，典型的代表是用在 IBM大型机上的系列操作系统OS/370/390/400。

OS/360的最早版本是纯粹的批处理系统。然而有许多希望使用分时系统的 360用户，于是IBM公司和另外的一些研究小组决定开发一个分时系统。 IBM公司随后提供了一套分时系统 TSS/360，它非常庞大，运行缓慢，几乎没有什么人用它。结果在花费了约五千万美元的研制费用后，该系统最终被弃之不用。但是麻省剑桥的一个 IBM研究中心开发了另一个完全不同的系统，这个系统被IBM公司最终作为产品，目前仍在 IBM公司的大型主机上广泛使用。这个系统最初被命名为CP／CMS，后来改名为VM／370。它基于如下的设计思想，分时系统应该提供这些功能：① 多道程序；② 一个比裸机有更方便扩展界面的计算机。VM／370的任务是将这二者彻底隔离开来。

这个系统的核心被称为虚拟机监控程序，它在裸机上运行并且具备了多道程序功能。该系统向上层提供了若干台虚拟机，如图 2-3所示。它不同于其他操作系统的是：这些虚拟机不是那种具有文件等优良特征的扩展计算机。与之相反，它们仅仅是精确复制的裸机硬件。它包含：核心态/用户态；I/O功能；中断；以及其他真实硬件所具有的全部内容。

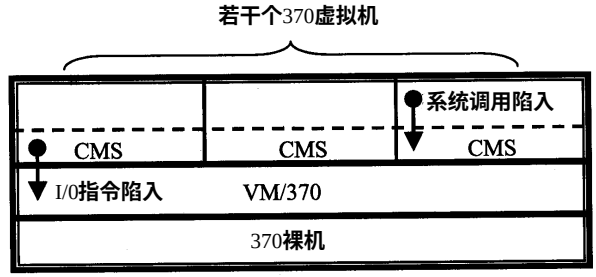


图2-3 带CMS的VM/370体系结构

因为每台虚拟机都与裸机相同，所以每台虚拟机可以运行一台裸机所能够运行的任何类型的操作系统。不同的虚拟机可以运行不同的操作系统，而且实际上往往如此。有一些虚拟机运行OS／360的后续版本，从事着批处理或事务处理；而另一些虚拟机运行单用户、交互式系统，供分时用户们使用，这个系统称作会话监控系统（ CMS)。

CMS的程序在执行系统调用时，它的系统调用陷入其虚拟机中的操作系统，而不是调用VM／370，不是在虚拟机上，就像在真正的计算机上一样。然后 CMS发出硬件I／O指令，在虚拟磁盘上读或者执行为该系统调用所需的其他操作。这些 I／O指令被VM／370捕获，作为对真实硬件模拟的一部分， VM／370随后就执行这些指令。这样，将多道程序的功能和提供扩展机器的功能完全分开后，它们各自都更简单、更灵活和更易于维护。

4. 客户/服务器体系结构

操作系统结构技术的发展是与整个计算机技术的发展相联系的。当前计算机技术发展的突出特点是要求广泛的信息和资源的共享。这一要求促使网络技术的普遍应用和发展。因为网络技术逐渐成熟并实用化，再加上数据库联网已是计算机应用的新趋势，所以为用户提供一个符合企业

信息处理应用要求的分布式的系统环境是十分必要的。事实上，在一个企业或部门中，数据一般总是在它产生的各个现场上就近被存储、管理、加工、组织和使用，只有少量的数据或加工后的信息才是供全局共享或为局部所使用的。所以，分布式处理才是真正合乎客观实际和新的应用需要的潮流。如果操作系统是采用客户 /服务器结构，它将非常适于应用在网络环境下，应用于分布式处理的计算环境中。这种体系结构又被称为微内核的操作系统体系结构，它所具有的一些特征将在下面讨论。

典型的采用客户/服务器结构模式的操作系统有卡内基·梅隆大学研制的 Mach和Windows NT 的早期版本。它们的共同特点是操作系统由下面两大部分组成：

1) 运行在核心态的内核。它提供所有操作系统基本都具有的那些操作，如线程调度、虚拟存储、消息传递、设备驱动以及内核的原语操作集和中断处理等。这些部分通常采用层次结构并构成了基本操作系统。因为这时的内核只提供一个很小的功能集合，所以通常又称为微内核。

2) 运行在用户态并以客户 /服务器方式运行的进程层。这意味着除内核部分外，操作系统所有的其他部分都被分成若干个相对独立的进程，每一个进程实现一组服务，称为服务进程（用户应用程序对应的进程，虽然也以客户 /服务器方式活动于该层，但不将其看成操作系统的功能构成成分）。这些服务进程可以提供各种系统功能、文件系统服务以及网络服务等。服务进程的任务是检查是否有客户提出要求服务的请求，并在满足客户进程的请求后将结果返回。而客户可以是一个应用程序，也可以是另一个服务进程。客户进程与服务器进程之间的通信是采用发送消息进行的，这是因为每个进程属于不同的虚拟地址空间，它们之间不能直接通信，必须通过内核进行，而内核则是被映射到每个进程的虚拟地址空间内的，它可以操纵所有进程。客户进程发出消息，内核将消息传给服务进程。服务进程执行相应的操作，其结果又通过内核用发消息方式返回给客户进程，这就是客户/服务器的运行模式。

这种模式的优点在于，它将操作系统分成若干个小的、自包含的分支（服务进程），每个分支运行在独立的用户进程中，相互之间通过规范一致的方式接收发送消息而联系起来。操作系统在内核中建立起最小的机制，而把策略留给在用户空间中的服务进程，这带来了很大的灵活性，直接的好处是：

- 可靠。因为每个分支是独立的和自包含的（分支之间耦合最为松散），所以即使某个服务器失败或产生问题，也不会引起系统其他服务器和系统其他组成部分的损坏或崩溃。
- 灵活。便于操作系统增加新的服务功能，这是因为它们是自包含的，且接口规范。同时修改一个服务器的代码不会影响系统其他部分，可维护性好。
- 适宜于分布式处理的计算环境。由于不同的服务可以运行在不同的处理器或计算机上，从而使操作系统自然地具有分布式处理的能力。

当然这种体系结构也有它的缺陷，主要是对于效率的考虑。因为所有的用户进程只能通过微内核相互通信，所以微内核本身就成为系统的瓶颈，在一个通信很频繁的系统中，微内核往往不能提供很好的效率。例如，高性能的图形用户界面系统中经常有大量的数据在不同的进程中来回拷贝，那么把图形引擎作为一个运行在用户态的服务进程对一个有着高性能图形需求的系统来说将是不明智的选择。

2.2 Windows 2000/XP的操作系统模型

作为一个实际应用中的操作系统，Windows 2000/XP没有单纯地使用某一种体系结构，它的设计融合了分层操作系统和客户/服务器（微内核）操作系统的特点。

Windows 2000/XP像其他许多操作系统一样通过硬件机制实现了核心态（管态，kernel mode）以及用户态（目态，user mode）两个特权级别。当操作系统状态为前者时，CPU处于特权模式，可以执行任何指令，并且可以改变状态。而在后面一个状态下，CPU处于非特权（较低特权级）模式，只能执行非特权指令。一般来说，操作系统中那些至关紧要的代码都运行在核心态，而用户程序一般都运行在用户态。当用户程序使用了特权指令，操作系统就能借助于硬件提供的保护机制剥夺用户程序的控制权并做出相应处理。

在Windows 2000/XP中，只有那些对性能影响很大的操作系统组件才在核心态下运行。在核心态下，组件可以和硬件交互，组件之间也可以交互，并且不会引起描述表切换和模式转变。例如，内存管理器、高速缓存管理器、对象及安全管理器、网络协议、文件系统（包括网络服务器和重定向程序）和所有线程和进程管理，都运行在核心态。因为核心态和用户态的区分，所以应用程序不能直接访问操作系统特权代码和数据，所有操作系统组件都受到了保护，以免被错误的应用程序侵扰。这种保护使得Windows 2000/XP可能成为坚固稳定的应用程序服务器，并且从操作系统服务的角度，如虚拟内存管理、文件I/O、网络和文件及打印共享来看，Windows 2000/XP作为工作平台仍是稳固的。

Windows 2000/XP的核心态组件使用了面向对象设计原则，例如，它们不能直接访问某个数据结构中由单独组件维护的消息，这些组件只能使用外部的接口传送参数并访问或修改这些数据。但是Windows 2000/XP并不是一个严格的面向对象系统，出于可移植性以及效率因素的考虑，Windows 2000的大部分代码不是用某种面向对象语言写成，它使用了C语言并采用了基于C语言的对象实现。

Windows 2000/XP的最初设计是相当微内核化的，随着不断的改型以及对性能的优化，目前的Windows 2000/XP已经不是经典定义中的微内核系统。出于对效率的考虑，经典的微内核系统在商业上并不具有实践价值，因为它们太低效了。Windows 2000/XP将很多系统服务的代码放在了核心态，包括像文件服务、图形引擎这样的功能组件。应用的事实证明这种权衡使得Windows 2000/XP更加高效而且并不比一个经典的微内核系统更容易崩溃。

Windows 2000/XP的体系结构的框架如图2-4所示，接下来的几小节将对这个图做出详细的说明。

2.2.1 Windows 2000/XP的构成

图2-4中的粗线将Windows 2000/XP分为用户态和核心态两部分。粗线上部的方框代表了用户进程，它们运行在私有地址空间中。用户进程有四种基本类型：① 系统支持进程（system support process），例如登录进程WINLOGON和会话管理器SMSS，它们不是Windows 2000/XP的服务，不由服务控制器启动；② 服务进程（service process），它们是Windows 2000/XP的服务，例如事件日

志服务；③ 环境子系统（enviroment subsystem），它们向应用程序提供运行环境（操作系统功能调用接口），Windows 2000/XP有三个环境子系统：Win32、POSIX和OS/2 1.2；④ 应用程序（user application），它们是Win32、Windows 3.1、MS-DOS、POSIX 或OS/2 1.2这五种类型之一。

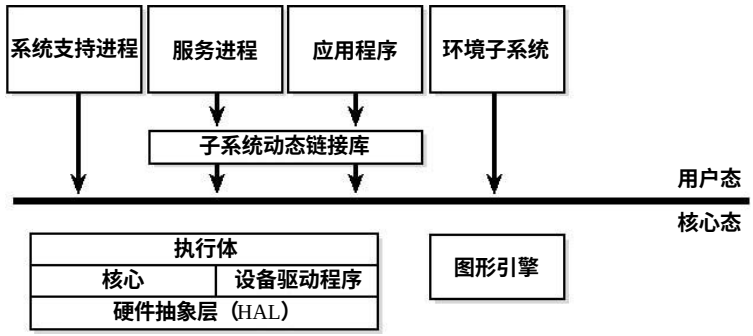


图2-4　Windows 2000/XP体系结构框图

从图2-4中可以看到，服务进程和应用程序是不能直接调用操作系统服务的，它们必须通过子系统动态链接库（subsystem DLLs）和系统交互。子系统动态链接库的作用就是将文档化函数（公开的调用接口）转换为适当的 Windows 2000/XP内部系统调用。这种转换可能会向正在为用户程序提供服务的环境子系统发送请求，也可能不会。

粗线以下是Windows 2000/XP的核心态组件，它们都运行在统一的核心地址空间中。核心类组件包括以下内容：① 核心（kernel）包含了最低级的操作系统功能，例如线程调度、中断和异常调度、多处理器同步等，同时它也提供了执行体（ Executive）来实现高级结构的一组例程和基本对象；② 执行体包含了基本的操作系统服务，例如内存管理器、进程和线程管理、安全控制、I/O以及进程间的通信；③ 硬件抽象层（Hardware Abstraction Layer, HAL）将内核、设备驱动程序以及执行体同硬件分隔开来，使它们可以适应多种平台；④ 设备驱动程序（Device Drivers）包括文件系统和硬件设备驱动程序等，其中硬件设备驱动程序将用户的 I/O函数调用转换为对特定硬件设备的I/O请求；⑤ 图形引擎包含了实现图形用户界面（ Graphical User Interface，GUI）的基本函数。

从基本的构成看， Windows 2000/XP和大多数的UNIX系统很相似，它也是一个集成操作系统 —— 它的重要组件和设备驱动程序共享内核受保护的地址空间，任何操作系统组件和设备驱动程序可以很容易地破坏其他组件和驱动程序使用的数据，不过实际中这种事情很少发生。这些重要的系统成分都和应用程序隔离，这种保护使得 Windows 2000/XP保持了高效和健壮。

2.2.2　Windows 2000/XP的可移植性

Windows 2000/XP的设计目标之一就是能够在各种硬件体系结构上运行，它用两种方法实现了对硬件结构和平台的可移植性。首先是一个分层的设计，依赖于处理器体系结构或平台的系统底层部分被隔离在单独的模块之中，系统的高层可以被屏蔽在千差万别的硬件平台之外。提供操作系统可移植性的两个关键组件是HAL和内核。依赖于体系结构的功能（如线程描述表切换）在

内核中实现，在相同体系结构中，因计算机而异的功能在 HAL中实现。第二个方法是 Windows 2000/XP几乎全部使用高级语言写成——执行体、实用程序和设备驱动程序都是用 C语言编写的，图形子系统部分和用户界面是用 C++编写的。只有那些必须和系统硬件直接通信的操作系统部分（如中断陷阱处理程序），或性能极度敏感（如描述表切换）的部分是用汇编语言编写的。汇编语言代码主要分布在内核及HAL中，极少量分布于执行体的少数区域（例如实现互锁指令的执行体例程）、Win32子系统的核心部分和少数用户态库中，例如在 NTDLL.DLL中的进程启动代码。

2.2.3 Windows 2000/XP的对称多处理的支持

Windows 2000/XP支持“对称多处理”（Symmetric MultiProcessing, SMP）。在SMP中不存在主处理器——操作系统和用户线程能被安排在任一处理器上运行；所有的处理器共享一个内存空间。这种模型与“非对称多处理”（ASymmetric MultiProcessing，ASMP）形成对比，后者只能在某个特定处理器上执行操作系统代码，而其他处理器只能运行用户代码。

多处理器系统的一个关键问题是可伸缩性。为了保证系统能在 SMP系统上正确运行，操作系统代码必须严格遵守某些规则以确保操作正确。在多处理器系统中，资源竞争及其他性能问题比在单处理器系统中更加复杂，Windows 2000/XP集成了许多关键特性，使之成为一个成功的多处理器操作系统。

Windows 2000/XP能在任何可用的处理器上运行，并且它的完全可重入的代码可以同时在多个处理器上运行。当一个较高优先权的线程需要获得处理器时间时，利用系统陷阱调度（ trap dispatching）机制，所有操作系统代码都可以被抢先（强制释放一个处理器）。在不同的处理器中，每一个线程基本上都可以同时执行。核心以及设备驱动程序和服务进程内部的精确同步允许更多的组件在多处理器上同时运行，在进程间共享对象的机制及灵活的进程间的通信能力，包括共享内存和优化的消息传送工具。

除了HAL对于单处理器系统和多处理器系统在本质上有所不同外， Windows 2000/XP只包含了执行体和内核的核心操作系统映像。 NTOSKRNL.EXE这一个文件在单处理器和多处理器版本中是不同的。其他二进制文件则在单处理器和多处理器系统上都能正确运行。

2.3 Windows 2000/XP的体系结构

从本节开始我们将把图2-4的细节逐渐展开（见图2-5），逐一介绍构成Windows 2000/XP的各个组成部分体系结构的细节。

2.3.1 内核

内核执行Windows 2000/XP中最基本的操作，主要提供下列功能：① 线程安排和调度；② 陷阱处理和异常调度；③ 中断处理和调度；④ 多处理器同步；⑤ 供执行体使用的基本内核对象（在某些情况下可以导出到用户态）。

Windows 2000/XP的内核始终运行在核心态，代码短小紧凑，可移植性也很好。一般来说，除了中断服务例程（Interrupt Service Routine，ISR），正在运行的线程是不能抢先内核的。

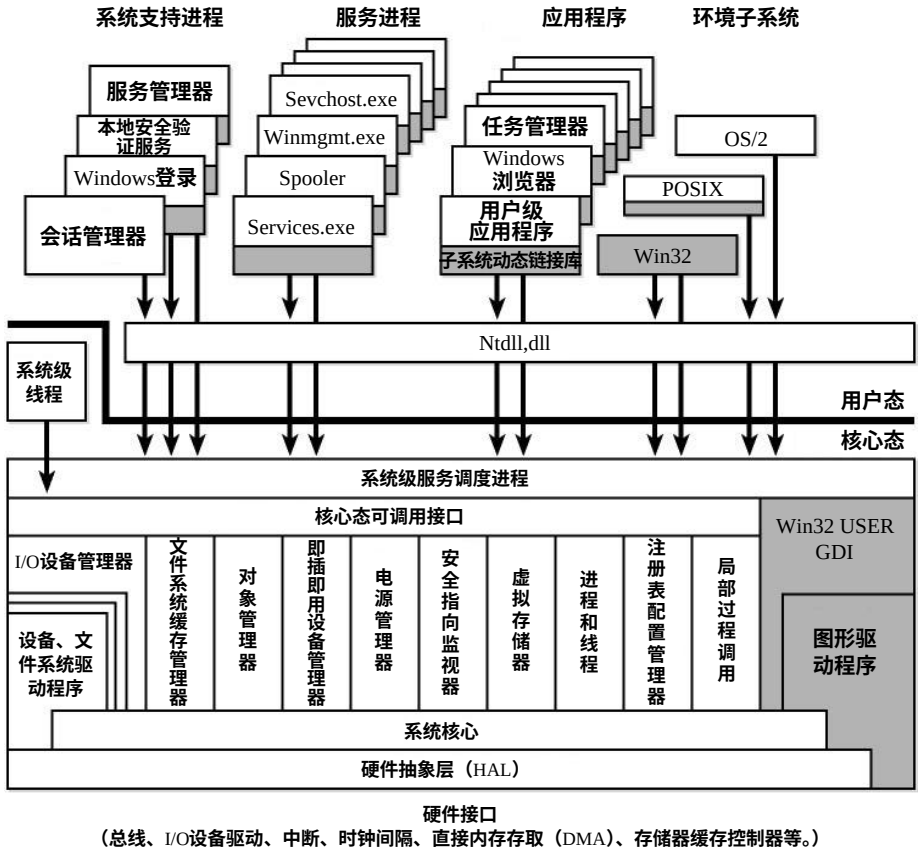


图2-5　Windows 2000/XP的体系结构详图

1. 内核对象

内核提供了一组严格定义的、可预测的、使得操作系统得以工作的基础设施，这为执行体的高级组件提供了必须的低级功能接口。内核除了执行线程调度外，几乎将所有的策略制定留给了执行体。这一点充分体现了Windows 2000/XP将策略与机制分离的设计思想。

在内核以外有很多的系统组件，处理它们的资源分配、安全认证等都要执行体付出不可忽略的策略开销。内核通过一组称作“内核对象”的简单对象帮助控制、处理并支持执行体对象的创建，以降低这种开销。大多数执行体级别的对象都封装了一个或多个内核对象。

一个称作“控制对象”的内核对象集合为控制各种操作系统功能建立了语义。这个对象集合包括内核进程对象、异步过程调用（Asynchronous Procedure Call，APC）对象、延迟过程调用（Deferred Procedure Call，DPC）对象和几个由I/O系统使用的对象（例如中断对象）。

另一个称作“调度程序对象”的内核对象集合负责同步操作并影响线程调度。调度程序对象包括内核线程、互斥体（Mutex）、事件（Event）、内核事件对、信号量（Semaphore）、定时器和可等待定时器。执行体使用内核函数创建内核对象的实例，使用它们来构造更复杂的对象提供

给用户态。

2. 硬件支持

内核的另外一个重要功能就是把执行体和设备驱动程序同硬件体系结构的差异隔离开，包括处理功能之间的差异，例如中断处理、异常情况调度和多处理器同步。对于与硬件有关的函数，内核的设计也是尽可能使公用代码的数量达到最大。内核支持一组在整个体系结构上可移植、语义完全相同的接口，大多数这种接口的实现在整个体系结构上是完全相同的。当然也有一些接口的实现因体系结构而异。Windows 2000/XP可以在任何机器上调用那些独立于体系结构的接口，不管代码是否随体系结构而异，这些接口的语义总是保持不变。一些内核接口实际上是在 HAL中实现的，因为同一体系结构内接口的实现可能也因平台系统而异。

内核包含少量支持老版本 MS-DOS程序所必需的x86专用代码，这些接口是不可移植的。另一个内核中的体系结构专用代码的例子是提供缓冲区和 CPU高速缓存转化支持的接口。因高速缓存执行方式的不同，对于不同的体系结构，这一支持需要的代码也不同。还有就是描述表切换，虽然在更高层次上来看，线程选择和描述表切换使用的是同一种算法，但它们在不同处理器中执行时还是存在结构上的差异。由于描述表是用处理器状态来描述的，因此保存与加载什么取决于体系结构。

2.3.2 硬件抽象层

Windows 2000/XP设计的一个至关重要的方面就是在多种硬件平台上的可移植性， HAL就是使这种可移植性成为可能的关键部分。 HAL是一个可加载的核心态模块 HAL.dll，它为运行在Windows 2000/XP上的硬件平台提供低级接口。 HAL隐藏各种与硬件有关的细节，例如 I/O接口、中断控制器以及多处理器通信机制等任何体系结构专用的和依赖于计算机平台的函数。

2.3.3 执行体

Windows 2000/XP的执行体是NTOSKRNL.EXE的上层（内核是其下层）。执行体包括五种类型的函数：

1) 从用户态导出并且可以调用的函数。这些函数的接口在 NTDLL.DLL中。通过Win32API或一些其他的环境子系统可以对它们进行访问。

2) 从用户态导出并且可以调用的函数，但当前通过任何文档化的子系统函数都不能使用。

3) 在Windows 2000 DDK中已经导出并且文档化的核心态调用的函数。

4) 在核心态组件中调用但没有文档化的函数。例如在执行体内部使用的内部支持例程。

5) 组件内部的函数。

执行体包含下列重要的组件，这些组件将在后续的小节中陆续加以介绍：

1) 进程和线程管理器创建及中止进程和线程。对进程和线程的基本支持在 Windows 2000内核中实现，而执行体给这些低级对象添加附加语义和功能。

2) 虚拟内存管理器实现“虚拟内存”。内存管理器也为高速缓存管理器提供基本的支持。

3) 安全引用监视器在本地计算机上执行安全策略。它保护了操作系统资源，执行运行时对象

的保护和监视。

4) I/O系统执行独立于设备的输入/输出，并为进一步处理调用适当的设备驱动程序。

5) 高速缓存管理器通过将最近引用的磁盘数据驻留在主内存中来提高文件 I/O的性能，并且通过在把更新数据发送到磁盘之前将它们在内存中保持一个短的时间来延缓磁盘的写操作，这样就可以实现快速访问。

另外，执行体还包括四组主要的支持函数，它们由上面列出的执行体组件使用。其中大约有三分之一的支持函数在DDK中已经文档化。这四类支持函数提供下面的功能：

1) 对象管理，创建、管理以及删除Windows 2000/XP的执行体对象和用于代表操作系统资源的抽象数据类型，例如进程、线程和各种同步对象。

2) 本地过程调用（Local Procedure Call，LPC）机制，在同一台计算机上的客户进程和服务进程之间传递信息。LPC是一个灵活的、经过优化的“远程过程调用”（Remote Procedure Call，RPC）版本。

3) 一组广泛的公用运行时函数，例如字符串处理、算术运算、数据类型转换和完全结构处理。

4) 执行体支持例程，例如系统内存分配（页交换区和非页交换区）、互锁内存访问和两种特殊类型的同步对象（资源和快速互斥体）。

2.3.4 设备驱动程序

设备驱动程序是可加载的核心态模块（通常以.SYS为扩展名），它们是I/O系统和相关硬件之间的接口。Windows 2000/XP上的设备驱动程序不直接操作硬件，而是调用HAL功能作为与硬件的接口。

Windows 2000/XP中有如下几种类型的设备驱动程序：① 硬件设备驱动程序操作硬件，它将输出写入物理设备或网络，并从物理设备或网络获得输入；② 文件系统驱动程序接受面向文件的I/O请求，并把它们转化为对特殊设备的I/O请求；③ 过滤器驱动程序截取I/O并在传递I/O到下一层之前执行某些特定处理。

因为安装设备驱动程序是把用户编写的核心态代码添加到系统的唯一方法，所以某些程序通过简单地编写设备驱动程序的方法来访问操作系统内部函数或数据结构，但它们不能从用户态访问。

Windows2000/XP增加了对即插即用和高级电源选项的支持，它使用 Windows驱动程序模型（Windows Driver Model，WDM）作为标准驱动程序模型，同时它也支持Windows NT的驱动程序，不过因为这些驱动不支持即插即用和电源选项，所以使用这些驱动的系统的实际能力将会降低。

从WDM的角度看，有三种驱动程序：

1) 总线驱动程序用于各种总线控制器、适配器、桥或者可以连接子设备的设备，这是必须的驱动程序。

2) 功能驱动程序用于驱动那些主要的设备，提供设备的操作接口。一般来说，这也是必须的，除非采用一种原始的方法来使用这个设备（功能都被总线驱动和总线过滤器实现了，例如 SCSI PassThru）。

3) 过滤器驱动程序用于为一个设备或者一个已经存在的驱动程序增加功能，或者改变来自其他驱动程序的I/O请求和响应行为。过滤器驱动程序是可选的，并且可以有任意的数目，它存在于功能驱动程序的上层或者下层、总线驱动程序的上层。

在WDM的驱动程序环境中，没有一个单独的设备驱动控制着某个设备。总线设备驱动程序负责向即插即用管理器报告它上面有的设备，而功能驱动程序则负责操纵这些设备。

2.3.5 环境子系统和子系统动态链接库

Windows 2000/XP有三种环境子系统：POSIX、OS/2和Win32（OS/2 只能用于x86系统）。在这三个子系统中，Win32子系统比较特殊，如果没有它，Windows 2000/XP就不能运行。而其他两个子系统只是在需要时才被启动，而Win32子系统必须始终处于运行状态。

环境子系统的作用是将基本的执行体系统服务的某些子集提供给应用程序。每个子集都可以提供访问Windows 2000/XP中本地服务的不同子集，函数调用不能在子系统之间混用。用户应用程序不能直接调用Windows 2000/XP系统服务，这种调用必须通过一个或多个子系统动态链接库作为中介才可以完成。例如，Win32子系统动态链接库（如KERNEL32.DLL、USER32.DLL和GD132.DLL）实现Win32API函数，POSIX子系统动态链接库则实现POSIX 1003.1API。

每一个可执行的映像（.EXE）都受限于唯一的子系统，进程创建时，程序映像头中的子系统类型代码会告诉Windows新进程所属的子系统。类型代码可以使用 Windows 2000资源管理器中内置的快速查看器、Link/DUMP命令或者在Windows 2000资源工具包中的Exetype工具来查看。

当一个应用程序调用子系统动态链接库中的函数时，会出现下面三种情况之一：

1) 函数完全在子系统动态链接库的用户态部分中实现，这时并没有消息发送到环境子系统进程，也没有调用执行体服务。函数在用户态中执行，结果返回到调用者。

2) 函数需要一个或多个对执行体的调用。

3) 函数要求某些工作在环境子系统进程中进行。在这种情况下，将产生一个客户 /服务器请求到环境子系统，其中的一个消息将被发送到子系统去执行某些操作，这可能会使用执行体的“本地过程调用”（LPC）机制。然后，子系统动态链接库在消息返回给调用者之前会一直等待应答。

此外，某些函数可能是上述第二与第三项的结合，如 Win32 Create Process和Create Thread函数。

Windows 2000/XP可以支持多重独立环境子系统，但从实用角度来看，每个子系统执行所有的代码并处理窗口和显示I/O将有大量系统函数的重复，这很可能对系统大小和性能产生负面影响。因而，Windows 2000/XP中Win32是主子系统，基本函数都放在该子系统中，并且让其他子系统调用Win32子系统来执行显示I/O。这样，POSIX与OS/2将调用Win32子系统中的服务来执行显示I/O。目前的POSIX子系统只能执行一个非常有限的函数集（仅POSIX 1003.1），这对于移植UNIX应用程序来说并不是一种有用的环境。在Windows XP中，这两个部分实际上已经被去除掉了，其中POSIX子系统将在改型优化以后出现在微软公司更加新的系统中。

1. Win32子系统

Win32子系统由下列重要组件构成：

- Win32环境子系统进程CSRSS，包括对下列功能的支持：控制台（文本）窗口、创建及删除进程与线程、支持16位DOS虚拟机（VDM）进程的部分。
- 其他混杂的函数，如GetTempFile、DefineDosDevice、ExitWindowsFx和几种自然语言支持函数。
- 核心态设备驱动程序（WIN32K.SYS），包括下列功能：窗口管理器控制窗口显示；管理屏幕输出；收集来自键盘、鼠标和其他设备的输入信息；以及将用户信息传送给应用程序。
- 图形设备接口（Graphics Device Interface，GDI）是一个用于图形输出设备的函数库，它包括线条、文本、绘图和图形操作函数。
- 子系统动态链接库（例如 USER32.DLL、ADVAPI32.DLL、GDI32.DLL和KERNEL32.DLL），它调用NTOSKRNL.EXE和WIN32.SYS将文档化的Win32 API函数转化为适当的非文档化的核心系统服务。
- 图形设备驱动程序，它包括依赖于硬件的图形显示驱动程序、打印机驱动程序和视频小端口驱动程序。

应用程序调用标准的USER函数在显示器上创建窗口和按钮。窗口管理器传递这些请求到GDI，GDI再将这些请求传送给图形设备驱动程序，在这里将按照显示设备的要求将其规格化。显示驱动程序与视频小型端口驱动程序相配合来完成对视频显示的支持。每个视频小端口驱动程序都对与之相关的显示驱动程序提供硬件级支持。

GDI提供了一组标准的函数，它使得应用程序可以同图形设备（包括显示器和打印机）通信而不必知道关于这些设备的任何事情。GDI的各种函数在应用程序与图形设备（例如显示驱动程序及打印机驱动程序）之间起协调作用。GDI解释应用程序对图形输出的要求，并把它们发送到图形显示驱动程序。GDI也能够为应用程序提供使用不同图形输出设备的标准接口。这个接口可以让应用程序代码独立于硬件设备和硬件设备驱动程序。GDI为使其信息适合设备的功能，常常把要求划分为易于处理的各个部分。

对于在客户端的每一个线程，这里都有专用的成对的服务器线程在Win32子系统进程中等待客户进程的请求。一个称作“快速LPC”的在进程间通信的特殊机制在这些线程间发送信息。同正常的线程描述表切换不同，通过快速LPC在一对线程之间进行的转换不会在内核中产生再调度事件，这样客户线程在内核抢先线程调度程序中获得它的时间片之前，允许服务线程在客户线程的剩余时间片中运行。此外，共享内存缓冲器被用作允许快速传递一些大的数据结构（例如位图），同时为了最小化在客户和Win32服务之间线程/进程转换的要求，客户可以直接（但是以只读方式）访问关键的服务器数据结构。GDI操作也是批处理化的（现在仍然是）。“批处理”意味着一系列由Win32应用程序调用的图形不会被“推”到服务器上，也不会被画到输出设备上，直到一个GDI批处理队列被装满。您可以使用Win32 GdiSetBatchLimit函数设置队列的大小，也可以在任何时候使用GdiFlush刷新队列。相反，对于GDI的只读属性和数据结构，一旦从Win32子系统进程得到它们，它们就会在客户端被高速缓存以用于快速后继访问。

尽管采用了这些优化，整个系统性能仍然不能满足图形密集型应用程序的要求。最明显的解决方案就是通过将窗口和图形系统移入到核心态来消除对附加线程和因此带来的对描述表切换的要求。同样，一旦应用程序调用进入到窗口管理器和 GDI中，这些子系统就可以直接访问其他执行体组件，而不会有用户态或核心态转换的费用。在通过 GDI调用视频驱动程序（一个进程，它涉及与视频硬件在高频、高带宽下的交互作用）的情况下，这种直接访问尤其重要。

Win32子系统的用户态进程部分还包括所有对控制台或文本窗口的描绘和更新。但是除了控制台窗口支持以外，只有少数 Win32函数会给 Win32子系统进程发送消息：进程和线程的创建和中止、映射网络驱动器以及创建临时文件。一般来说，正在运行的 Win32应用程序不会引起太多到Win32子系统进程的描述表切换。

2. POSIX子系统

POSIX代表了UNIX类型的操作系统接口的国际标准集，它鼓励制造商实现兼容的 UNIX风格接口，以使编程者能够很容易地将他们的应用程序从一个系统移到另一个系统。 Windows 2000/XP只实现了POSIX.1标准（ISO/IEC 9945-1 1990或IEEE POSIX 1003.1-1990）。所需的POSIX一致性文档位于Platform SDK中的\HELP目录中。

Windows 2000/XP被设计成确保提出所需的基本操作系统支持以便考虑 POSIX.1子系统的实现，然而，因为POSIX.1只定义了一组有限的服务，所以单独的 POSIX子系统并不是一个完整的编程环境。并且因为在 Windows 2000/XP上应用程序不能在子系统之间混合调用，所以 POSIX应用程序被严格限制在 POSIX.1中定义的一组服务。这种限制意味着在 Windows 2000/XP上可执行的POSIX不能创建线程或窗口，也不能使用远程过程调用（ RPC）或套接字，然而您可以在Win32应用程序中做所有这些事情。

不过，这种情况在下一代的 Windows（Interix）操作系统中将有所改变。我们可以看一下Interix的体系结构图（图2-6、图2-7）。

从图2-6可以知道，在 Interix的计划中， POSIX/ UNIX子系统得到了相当大的改善，支持了主流的UNIX标准和UNIX应用（例如X以及标准UNIX Shell）。它几乎成为和 Win32子系统平行的部分。

3. OS/2子系统

OS/2子系统在实用性方面受到很大的限制，它仅支持 X86系统以及基于16位字符的OS/2 1.2或视频I/O应用程序。

4. NTDLL.DLL

NTDLL.DLL是一个特殊的系统支持库，主要用于子系统动态链接库。 NTDLL.DLL包含两种类型的函数：执行体提供的系统服务调度占位程序；子系统、子系统动态链接库以及其他本机映像使用的内部支持函数。其中第一组函数提供了可以从用户态调用的作为 Windows 2000/XP执行体系统服务的接口。这些函数的大部分功能都可以通过 Win32 API访问。对于这些函数中的每个函数， NTDLL都包含一个有相同名称的入口点。在函数内的代码含有体系结构专用的指令，它能够产生一个进入核心态的转换以调用系统服务调度程序。在进行一些验证后，系统服务调度程序将调用包含在NTOSKRNL.EXE内的实代码的实际的核心态系统服务。 NTDLL也包含许多支持

函数，例如映像加载程序、堆管理器和Win32子系统进程通信函数以及通用运行时库例程。此外，它还包含用户态异步过程调用（APC）调度器和异常调度器。

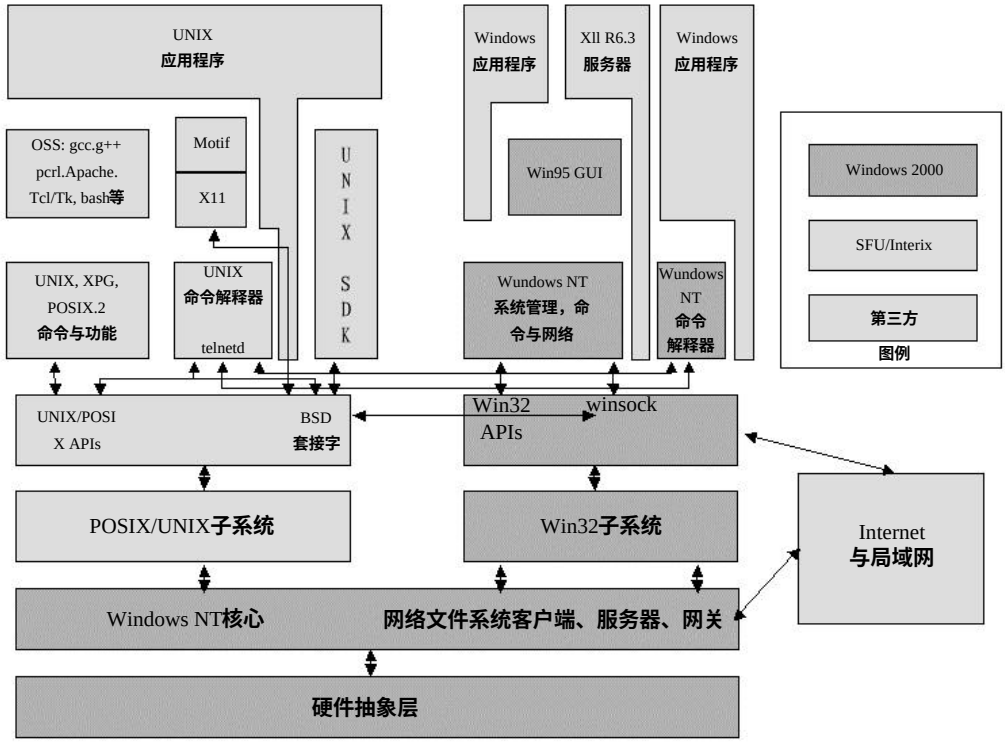


图2-6　Interix的体系结构

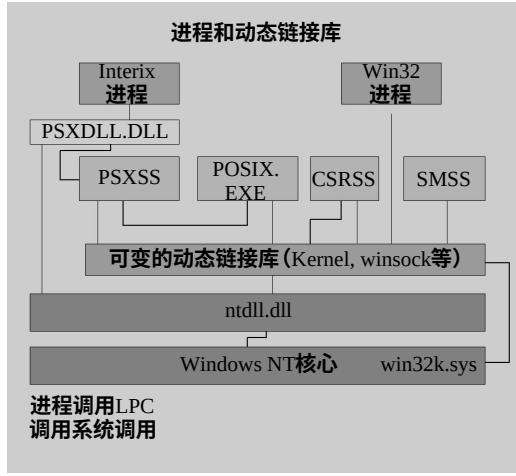


图2-7　Interix的调用机制

2.3.6 系统支持进程

下面的系统支持进程在所有的 Windows 2000/XP系统上都有。

- Idle进程（对于每个CPU，Idle进程都包含一个相应的线程，用来统计空闲 CPU时间）。
- 系统进程（包含核心态系统线程）。
- 会话管理器（SMSS）。
- Win32子系统（CSRSS）。
- 登录进程（WINLOGIN）。
- 本地安全身份验证服务器（LSASS）。
- 服务控制器（SERVICES）及其相关的服务进程。

1. Idle进程

在Windows 2000/XP中Idle进程的ID总是0，而不管进程的名称是什么。一般的进程都有它们的映像名标识， Idle（以及进程ID2，名称是System）不是运行在真正的用户态，因此由不同的进程观察程序显示的名称是随该程序的不同而不同的值。

2. 系统进程和系统线程

系统进程的ID总是 2，它是一种特殊类型的、只运行在核心态的“ 系统线程 ”的宿主。系统线程具有一般用户态线程的所有属性和描述表，不同点在于它们仅运行在核心态，执行加载于系统空间中的代码，而不管它们是在 NTOSKRNL.EXE中还是在任何其他已经加载的设备驱动程序中。另外，系统线程没有用户进程地址空间，因此必须从系统内存堆中分配动态存储区。

系统线程只能从核心态调用。Windows 2000/XP以及不同的设备驱动程序在系统初始化时创建系统线程以执行那些需要线程描述表的操作，例如，发布和等待I/O或其他对象，轮询一个设备等。

3. 会话管理器

会话管理器是第一个在系统中创建的用户进程。由核心系统线程运行例程 ExInitializeSystem创建。除了执行一些关键的系统初始化步骤以外，会话管理器还作为应用程序和调试器之间的开头和监视器。

下面列出了由SMSS的主线程执行的初始化步骤：

1) 创建LPC端口对象（\SmApiPort）和两个线程，等待客户的请求。
2) 创建系统环境变量。
3) 定义用于MS-DOS设备名称的符号链接（例如COMI或LPT1）。
4) 创建附加的页面调度文件。
5) 打开已知的动态链接库。
6) 加载Win32子系统的核心态部分（WIN32K.EXE）。
7) 启动子系统进程。
8) 启动登录进程。
9) 创建用于调试事件消息的LPC端口并创建一些线程来监视这些端口。

在执行这些初始化步骤之后， SMSS中的主线程将永远等待 CSRSS和WINLOGON的进程句

柄，如果这些进程意外终止，SMSS将使系统崩溃。

当然，SMSS中的其他线程会负责把消息发送给上面提到的 LPC端口，例如请求加载子系统、启动新的子系统和调试事件。

4. 登录进程

Windows 2000的登录进程WINLOGON处理用户登录和注销的内部活动。当输入“安全注意序列”（SAS，Secuity Attention Sequence，常为Ctrl＋Alt＋Del）的组合键时，用户登录请求就通知WINLOGON，使用SAS的原因是保护用户不受那些能模拟登录进程的密码捕获程序的干扰。一旦用户名和密码被捕获，它们将被发送到本地安全身份验证服务器进程以确认其合法性。如果确认相符，将创建一个叫做 USERINIT.EXE的进程，这个线程在注册表中查找并创建系统定义的Shell，随后USERINIT就退出了。

登录进程的标识和身份验证是在名为 GINA的可替换动态链接库中实现的。作为标准 Windows 2000/XP 的GINA DLL, MSGINA.DLL来实现其他的标识和身份验证机制以代替标准的Windows 2000/XP用户名/密码方法。另外，WINLOGON可以加载附加的需要执行二级身份验证的网络提供者动态链接库。这种能力允许多个网络提供者收集所有的在一次正常登录时的标识和身份验证信息。

WINLOGON不仅在用户登录和注销时是活动的，无论何时从键盘截取 SAS，它也是活动的。

5. 本地安全身份验证服务器

本地安全身份验证服务器进程接收来自 WINLOGON的身份验证请求，并调用适当的身份验证包来执行实际的验证，例如检查一个密码是否与存储在 SAM文件中的密码匹配。

在身份验证成功时， LSASS将生成一个包含用户安全配置文件的访问令牌对象。WINLOGON随后使用这个访问令牌去创建初始外壳进程。这些进程将从外壳启动，然后默认地继承这个访问令牌。

6. 服务控制器

在Windows 2000中，“服务”既可以指服务进程也可以指设备驱动程序，这里特指用户态进程服务。服务就像UNIX的“守护进程”或VMS的“派遣进程”一样，可以配置成在系统引导时自动启动而不需要交互式登录，当然，服务也可以手工启动。

服务程序是真正合法的 Win32映像，这些映像调用特殊的 Win32函数以与服务控制器相互使用，例如注册、启动、响应状态请求、暂停或关闭服务。一些 Windows 2000组件是作为服务来实现的，例如假脱机、事件日志、用于 RPC的支持和其他各种各样的网络组件。

服务由服务控制器启动和停止。服务控制器是一个运行映像为 SERVICES.EXE的特殊系统进程，它负责启动、停止和与服务控制器交互。

2.4　Windows 2000/XP的系统机制

在上一节中，我们已经了解了 Windows 2000/XP基本的组成，从这一节开始我们将讨论有关这个体系结构如何运作的问题。 Windows 2000/XP提供了核心态组件工作的基本机制：

- 陷阱调度（Trap Dispatching），包括中断调度（Interruption Dispatching）、延迟过程调用

（Deferred Procedure Call，DPC）、异步过程调用（Asychronous Procedure Call，APC）、异常调度（Exception Dispatching）和系统服务调度（System Service Dispatching）。

- 执行体对象管理器（Executive Object Manager）。
- 同步（Synchronization），包括自旋锁（Spinlock）、内核调度程序对象（Kernel Dispatcher Objects）。
- 其他方面的机制，如Window NT 全局标志。
- 本地过程调用（LPC，Local Procedure Call）。

2.4.1 陷阱调度

陷阱处理器提供给操作系统用来处理意外事件的硬件机制。当异常或中断发生时，硬件或软件可以检测到它们，并捕获正在执行的线程，处理器会从用户态切换到核心态，并将控制转交给操作系统中相对固定的地址。在Windows 2000/XP中，处理器将控制转交给内核的陷阱处理程序，该模块检测异常和中断的类型，并将控制交给处理相应情况的代码。图 2-8给出了Windows 2000/XP陷阱调度的框架。

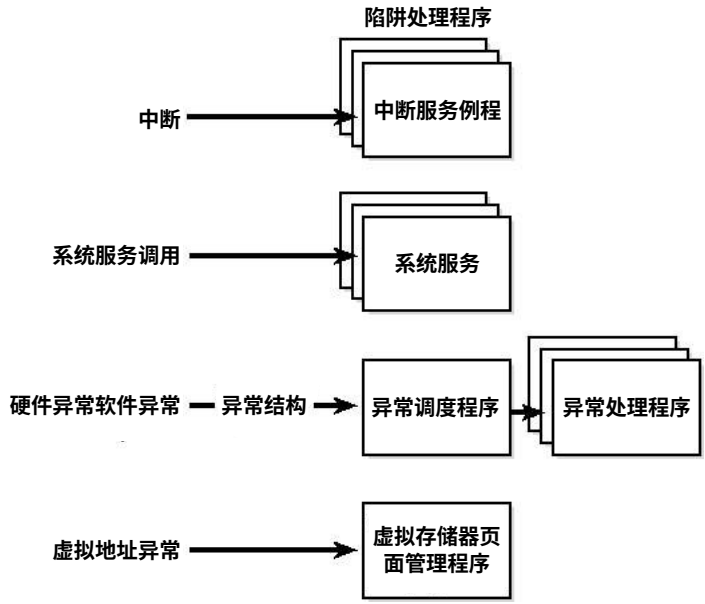


图2-8 Windows 2000/XP陷阱调度框架

一般来说，中断是异步事件，可能随时发生，与处理器正在执行的内容无关。中断主要由I/O设备、处理器时钟或定时器产生，可以被启用或禁用；异常是同步事件，它是某一特定指令执行的结果。在相同条件下，异常可以重现。例如，内存访问错误、调试指令以及被零除。系统服务调用也视作异常。软件和硬件都可以产生异常和中断。例如，总线错误异常是由硬件问题造成的，而被零除异常则是由软件错误引起的。同样，I/O设备可以产生中断，内核自身也可以发出软件中断。

当陷阱处理程序被调用时，将在记录机器状态时暂时禁用中断。它会创建一个陷阱帧（Trap Frame）来保存被中断线程运行现场，这用来在合适的时候恢复线程的执行。陷阱帧通常是完整的线程描述表的子集。陷阱处理程序本身可以处理一些事件，但大多数情况下，陷阱处理程序判定发生的情况，并将控制转交给其他的内核或执行体模块。例如，如果情况是设备中断产生的，内核把控制转交给设备驱动程序提供给该中断设备的中断服务例程（ISR）。如果情况是由调用系统服务产生的，陷阱处理程序会将控制转交给执行体中的系统服务代码。

1. 中断调度

最典型的硬件中断是由I/O设备产生的，当这些设备需要服务时，必须通知处理器。中断驱动的设备允许操作系统通过将指令执行与I/O操作重叠进行来获得处理器的最大利用率。处理器启动发往设备的I/O传送或来自设备的I/O传送，然后在设备完成传送时执行其他线程。当设备执行完后，它中断处理器以获得服务。定点设备、打印机、键盘、磁盘驱动器以及网卡通常都是中断驱动的。

软件也可以产生中断，例如，内核可以发布启动线程调度的软件中断。内核也可以禁用中断以使处理器不被中断，但这种情况很少出现，只在处理中断或调度异常的关键时刻才这样做。

中断由中断调度程序的子模块响应。它确定中断源并将控制转交给处理中断的外部例程（ISR），或转交给响应中断的内核例程。设备驱动程序给服务设备中断提供ISR，内核则提供其他类型中断的中断处理例程。

(1) 中断类型和优先级

不同的处理器中断机制也不一样，Windows 2000/XP的中断调度程序将硬件中断级映射到由操作系统识别的中断请求级别（Interrupt ReQuest Level，IRQL）的标准集上（如图2-9）。

IRQL与线程的调度优先级具有完全不同的含义。调度优先级是线程的属性，而IRQL是中断源的属性，每个处理器都具有一个IRQL设置，其值随着操作系统代码的执行而改变。

内核定义了一组可移植的IRQL，如果处理器具有与中断相关的特性，则可以增加IRQL。IRQL按优先级排列中断，并进行中断服务，较高优先级的中断服务可以抢占较低优先级的中断服务。

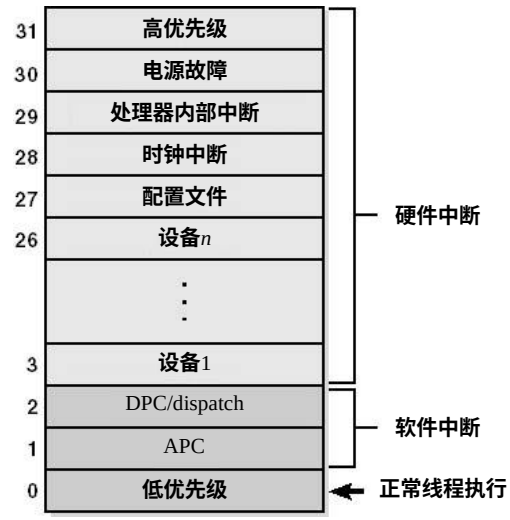


图2-9　中断请求级别

IRQL从高往低到设备都是为硬件中断保留的。Dispatch/DPC和APC级中断是内核和设备驱动器产生的软件中断（DPC和APC在本章稍后将详细介绍）。低优先级（也称作被动级）实际上并不是真正的中断级，在该级上执行普通的线程，并允许发生所有的中断。

IRQL设置决定了每个处理器可以接收的中断。当核心态线程运行时，它可以提高或降低处理器的IRQL来屏蔽一些事件。如果中断源的IRQL高于当前中断设置，则它的中断可以中断该处

理器；如果中断源的IRQL等于或低于当前中断设置，则它的中断将被封锁或“屏蔽”，直到一个正在执行的线程降低了IRQL。当产生中断时，陷阱处理程序提高处理器的IRQL直到与中断源所指定的IRQL相同，这可以保证服务于该中断的处理器不会被同级或较低级的中断抢先。被屏蔽的中断将被另一个处理器处理或阻挡，直到IRQL降低。因为改变处理器的IRQL对操作系统具有如此重要的影响，所以它只能在核心态下改变。

每个中断级都有特定的用途。例如，内核为了请求另一个处理器执行某个操作而发出处理器间中断；系统时钟以一定的间隔产生中断，内核通过更新时钟和测量线程的执行时间做出响应；HAL提供了许多中断驱动设备使用的中断级，确切数目依据处理器与系统配置而定；内核使用软件中断可启动线程调度并异步中断线程的执行。

(2) 硬件中断处理

当中断产生时，陷阱处理程序将保存计算机的状态，然后禁用中断并调用中断调度程序。中断调度程序立刻提高处理器的IRQL到中断源的级别，以便在中断服务进行时屏蔽等于或低于中断源级别的中断。然后，重新启用中断，以使高优先级的中断仍然能够得到服务。

Windows 2000使用中断分配表（Interrupt Dispatch Table，IDT）来查找处理特定中断的例程。中断源的IRQL作为表的索引，表的入口指向中断处理例程。在x86系统中，外部中断源触发中断控制器去中断处理器，一旦处理器被中断，它将向中断控制器询问中断向量。处理器利用此向量索引查询硬件IDT并将控制转交给适当的中断调度例程。

在服务例程执行之后，中断调度程序将降低处理器的IRQL到该中断发生前的级别，然后加载保存的机器状态，中断线程将从它停止的地方继续执行。在内核降低了IRQL后，被封锁的低优先级中断就可能出现。在这种情况下，内核将重复以上过程处理新的中断。

每个处理器都有独立的IDT，不同的处理器可以运行不同的ISR。例如，在多处理器系统中，每个处理器都收到时钟中断，但只有一个处理器在响应该中断时更新系统时钟。然而，所有的处理器都使用该中断来测量线程的时间片并在线程时间片结束后做一次调度。同样地，某些系统配置可能要求特殊的处理器处理某一设备中断。

大多数处理中断的例程都在内核中，为了管理它们，内核提供了称为中断对象的内核控制对象。中断对象是可移植的，并且允许设备驱动程序注册其设备的ISR。中断对象包含内核所需的将设备ISR与中断的特定级相联系的所有信息，包括ISR的地址、设备中断的IRQL以及与ISR相联系的内核中的入口。当中断对象被初始化后，称为调度代码的一些汇编语言代码指令就会被存储在对象中。当中断发生时，这些代码会调用真正的中断调度程序，并传递一个指向中断对象的指针。中断对象包含了第二个调度程序例程所需要的信息，以便定位和正确地调用设备驱动程序提供的ISR。需要两步过程的原因是自硬件完成初始调度后，没有一种方法可以在初始调度上传递一个指向中断对象的指针。

把ISR与某个中断级相关联叫做“连接一个中断对象”，而从IDT入口分离ISR叫做“断开一个中断对象”。这些操作允许设备驱动程序在被加载到系统时“打开”ISR，在卸载驱动程序时“关闭”ISR，它们可以通过调用内核函数来完成。

使用中断对象来注册ISR，可防止设备驱动程序直接随意中断硬件，并使设备驱动程序无需

了解IDT的任何细节。内核的这个特性有助于创建可移植的设备驱动程序，这是因为它消除了在设备驱动程序中体现处理器差异的需要。

中断对象使内核更容易调用多个任意中断级的 ISR。如果多个设备驱动程序创建多个中断对象并将它们连结到同一个 IDT入口，那么当中断在指定的中断级上发生时，中断调度程序会调用每一个例程。这样就使得内核很容易地支持“菊花链”配置，在这种构造中几个设备在相同的中断行上中断。

中断处理的另一个考虑是有关实时性的。时限要求是所有实时环境的共同特征，核反应堆控制系统等硬实时系统必须满足所设定的时限要求，以避免设备或生命损失等巨大的灾难性后果。轿车燃料控制系统等软实时系统的时限要求是可以错过的，但实时性仍然是系统追求的重要特征。实时系统中的计算机一般都有传感输入设备和控制输出设备。实时计算机系统的设计者必须知道，从输入设备产生中断请求到设备驱动程序控制输出设备作出响应之间的最长延时。这种延时分析必须考虑到操作系统、应用程序和驱动程序所占用的时间。

由于Windows 2000/XP并不以任何可控制的方式区分设备中断请求的优先顺序，用户级应用程序只能在处理器的被动中断优先级执行，因此它并不总是一个合适的实时操作系统。这不是由操作系统决定，而是由系统中的设备和设备驱动程序决定最坏情况下的延迟时间。当设计者希望使用平台无关的硬件时，这就成了一个很大的问题，因为无法决定每一个硬件的 ISR或者DPC需要多长时间。即使经过测试，设计者也不能保证在实际系统中不会由于某种特殊情况而超过一个重要的时间限制。所有系统设备的 DPC和ISR延时总和也大大超过了时间敏感系统的时限要求。

尽管打印机、车载计算机等许多嵌入式操作系统都有实时的要求，但 Windows NT的嵌入式版本并不具有实时特性。它只是一个简化的 Windows NT版本，该版本是微软公司利用VenturCom的技术设计完成的，适合于运行在资源有限的设备上。例如，在一个没有网络通信能力的设备上，嵌入式Windows NT可省掉通用Windows NT中的网络管理工具、网卡和协议堆驱动程序等所有与网络相关的部分。

尽管如此，一些第三方厂商为 Windows NT 4和Windows 2000/XP提供实时内核。它们将实时内核嵌在一个自定义的HAL中，把Windows 2000/XP或Windows NT 4当做实时系统中的一个任务来运行。这个Windows 2000和Windows NT任务可作为系统的用户接口，具有比设备管理任务低的优先级。

(3) 软件中断

虽然硬件产生了大多数的中断，但是 Windows 2000/XP内核也为多种任务产生软件中断，它们包括：启动线程调度、处理定时器到时、在特定线程的描述表中异步执行一个过程以及支持异步I/O操作等。

2. 调度或延迟过程调用

当一个线程不能继续执行时，可能是由于它已经结束或者它进入了等待状态，内核直接调用调度程序将立即实现描述表切换。然而，有时内核在深入多层代码内时检测到应该进行重调度，在这种情况下，理想的解决方法是请求调度，延迟它的产生直到内核完成当前的活动为止。使用DPC软件中断是实现这种延迟的简便方法。

当需要同步访问共享的内核结构时，内核总是将处理器的 IRQL提高到Dispatch/DPC级或高于Dispatch/DPC级，这样就禁用了其他的软件中断和线程调度。当内核检测到调度应该发生时，它将请求一个Dispatch/DPC级的中断；但由于IRQL等于或高于Dispatch/DPC级，处理器将在检查期间保存该中断。当内核完成当前活动后，它将 IRQL降至低于Dispatch/DPC级，于是调度中断便可出现。通过使用软件中断来激活线程调度程序是延迟调度直到条件合适为止的一种方法。而Windows 2000/XP也使用软件中断来延迟其他类型的处理。

除了线程调度以外，内核在其他 IRQL上也处理延迟过程调用。有一种 DPC是执行系统任务的函数，该任务比当前任务次要。这些函数叫做“ 延迟函数 ”，因为它们可能不立即执行。 DPC为操作系统提供了在内核态下产生中断并执行系统函数的能力。内核使用 DPC处理定时器到时（并释放在定时器上等待的线程）和在线程时间片结束后重调度处理器。设备驱动程序使用 DPC完成I/O请求。

DPC由DPC对象表示，它是一个内核控制对象。内核控制对象对于用户态的程序是不可见的，但对于设备驱动程序和其他系统代码是可见的。 DPC对象包含的最重要的信息是当内核处理 DPC中断时将调用的系统函数的地址。等待执行的 DPC例程被保存在叫做“ DPC队列 ”的内核管理队列中。为了请求一个 DPC，系统代码将调用内核来初始化 DPC对象，然后将它放入 DPC队列中，如图2-10所示。

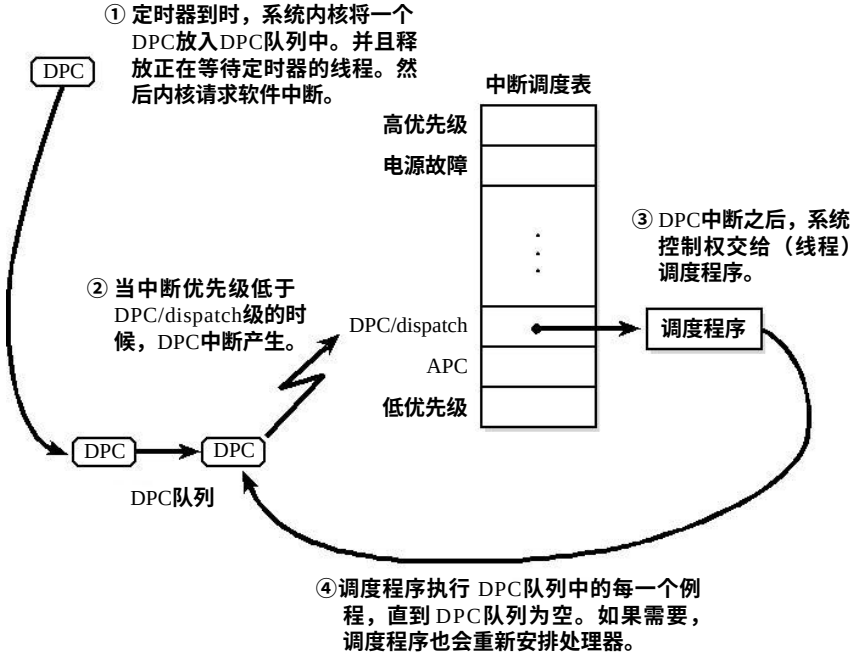


图2-10　延迟过程调用的提交

将一个DPC放入DPC队列会促使内核请求一个在 Dispatch/DPC级的软件中断。因为通常 DPC是由运行在较高IRQL级的软件对它进行排队的，所以被请求中断直到内核降低 IRQL到APC级或

低于APC级时才出现。

用户态线程是以低IRQL执行的，这是DPC中断普通用户线程执行的良好时机。DPC例程执行不考虑什么线程正在运行，因而它不能假定当前映射的进程地址空间是什么。DPC例程可以调用内核函数，但不能调用系统服务、产生页面故障以及创建或等待对象。不过，它们可以访问非页面系统内存地址，因为不管当前是什么进程，系统地址空间总是可以被映射的。

DPC主要是为设备驱动程序提供的，但内核也使用它们，最经常的应用就是处理时间片到时。系统时钟的每个跳动在时钟IRQL都产生一个中断，时钟中断处理程序（运行在时钟IRQL）更新系统时间，并减小用来记录当前线程运行时间的计数器值。当计数器值到达零时，线程的时间片就已经到时，内核就可能需要重调度处理器，这是一个应该在Dispatch/DPC IRQL完成的低优先级的任务。时钟中断处理程序对DPC排队以启动线程调度，然后完成它的工作并降低处理器的IRQL。因为DPC中断的优先级低于设备中断的优先级，所以任何挂起的的设备中断将在DPC中断产生之前得到处理。

3. 异步过程调用

异步过程调用（APC）为用户程序和系统代码提供了一种在特殊用户线程的描述表（一个特殊的进程地址空间）中执行代码的方法。APC在特殊线程描述表中执行，使用专用队列，它们以低于2的IRQL运行，所受的限制和DPC有很大的不同。APC例程可以获得资源（对象）、等待对象句柄、导致页错误以及调用系统服务。

APC也是由内核控制对象描述，称为APC对象。等待执行的APC在由内核管理的APC队列中。APC队列与DPC队列的不同在于：DPC队列是系统范围的；而APC队列是特定于线程的——每个线程都有自己的APC队列。当内核被要求对APC排队时，内核将APC插入到将要执行APC例程的线程的APC队列中。内核依次请求APC级的软件中断，并当线程最终开始运行时执行APC。

有两种APC，用户态APC和核心态APC。核心态APC在线程描述表中运行并不需要得到目标线程的“允许”，而用户态APC则需要得到目标线程的“允许”。核心态APC可以中断线程及执行过程，而不需要线程的干预或同意。

执行体使用核心态APC来执行必须在特定线程的地址空间（在描述表中）中完成的操作系统工作。例如，可以使用核心态APC命令一个线程停止执行可中断的系统服务，或记录在线程地址空间中的异步I/O操作的结果。环境子系统使用核心态的APC将线程挂起或终止自身的运行，或者得到或设置它的用户态执行描述表。POSIX子系统使用核心态APC来模仿POSIX信号到POSIX进程的发送。

设备驱动程序也使用核心态APC。例如，如果启动了一个I/O操作并且线程进入等待状态，则另一个进程中的另一个线程就可以被调度而去运行。当设备完成传输数据时，I/O系统必须以某种方式重新进入到启动I/O系统线程的描述表中，以便它能够来执行这个动作。

几个Win32 API，例如ReadiEx、WriteFileEx和QueueUserAPC，使用用户态APC。例如，ReadFileEx和WriteFileEx函数允许调用者指定I/O操作完成时将被调用的完成例程。该完成例程是通过把APC排队到发出I/O操作的线程来实现的。然而，在对APC排队时，对完成例程的回调是没有必要的，因为仅当线程在“可报警等待状态”（alterable wait state）时，用户态APC才被

传送给线程。线程可以通过等待对象句柄并且指定它的等待是可报警的（使用 Win32 WaitForMultipleObjectsE函数）进入等待状态，也可以通过直接测试它是否有一个挂起的 APC（使用SleepEx）进入等待状态。在两种情况下，如果用户态 APC是挂起的，内核将中断（报警）线程，将控制转交给APC例程，并在APC例程执行完成后，继续线程执行；如果发送 APC的线程处于等待状态，在APC例程执行完成后，等待被重新发出或重新执行。如果等待仍没有解决，线程将返回到等待状态，但此时它将位于它正在等待的对象列表的末尾。

4. 异常调度

异常直接由运行程序的执行所产生，除了那些简单的可由陷阱处理程序解决的异常之外，所有异常都是由称作“异常调度程序”的内核模块提供服务。异常调度程序的工作是找到可以“处理”该异常的异常处理程序。由内核定义的与体系结构无关的异常例子包括内存访问越界、被零除、整数溢出、浮点异常和调试程序断点。 Win32引入了称为“结构化异常处理”的工具，它允许应用程序在异常发生时可以得到控制。应用程序可以固定这个状态并返回到异常发生的地方展开堆栈，也可以向系统声明不能识别异常，并继续搜寻能处理异常的处理程序。

内核捕获和处理某些对用户程序透明的异常。内核通过给调用者返回不成功的状态代码来处理某些其他的异常。

少数异常可以被允许原封不动地过滤回用户态。环境子系统能够建立专用的异常处理程序来处理这些异常，它们与特殊过程的激活相关，称为基于框架的异常处理程序。当过程被调用时，表示该过程激活的堆栈框架就会被推入堆栈。堆栈框架可以有一个或多个与它相关的异常处理程序，每个处理程序都保护在源程序的一个特定代码块中。当异常发生时，内核将查找与当前堆栈框架相关的异常处理程序。如果没有，内核继续查找与前一个堆栈框架相关的异常处理程序，如此下去，直到找到一个基于框架的异常处理程序。如果还没有找到，内核将调用自己默认的异常处理程序。

异常调度的过程可见如图2-11所示的示意图。

异常会内核中会产生一个事件链，这与异常的来源无关。控制将转移到内核陷阱处理程序，陷阱处理程序将创建一个陷阱帧。如果完成了异常处理，陷阱帧将允许系统从中断处继续运行。陷阱处理程序同样将创建一个包含异常原因和其他有关信息的异常记录。

如果异常产生于核心态，异常调度程序将简单地调用一个例程来定位处理该异常的基于框架的异常处理程序。由于没有被处理的核心态异常将被视为致命的操作系统错误，因此可以假定调度程序总是能够找到异常处理程序。

程序调试断点也是异常的通常来源。异常调度程序的第一个动作就是查看引发异常的进程是否有相关的调试程序进程。如果有，它就给与引发异常的进程相关的调试程序端口发送第一个调试消息（通过本地过程调用 LPC端口）。如果进程没有挂接调试程序进程，或者调试程序不处理异常，那么异常调度程序将切换到用户态，并调用例程来找到某个基于框架的异常处理程序。如果没找到任何基于框架的异常处理程序，或没有处理异常，异常调度程序就将切换回核心态，并再一次调用调试程序，以重新允许用户进行调试。

所有Win32线程都由处于堆栈顶的异常处理程序来处理未处理的异常。如果线程有一个没有

处理的异常，它就会调用Win32未处理异常筛选程序。如果调试程序未运行，并且没有找到基于框架的处理程序，内核就会给与线程的进程相关的异常端口发送一条消息。现存的异常端口是由控制这个线程的环境子系统登记的。异常端口给了假定正在该端口监听的环境子系统一个将异常转换为环境指定的信号或异常的机会。

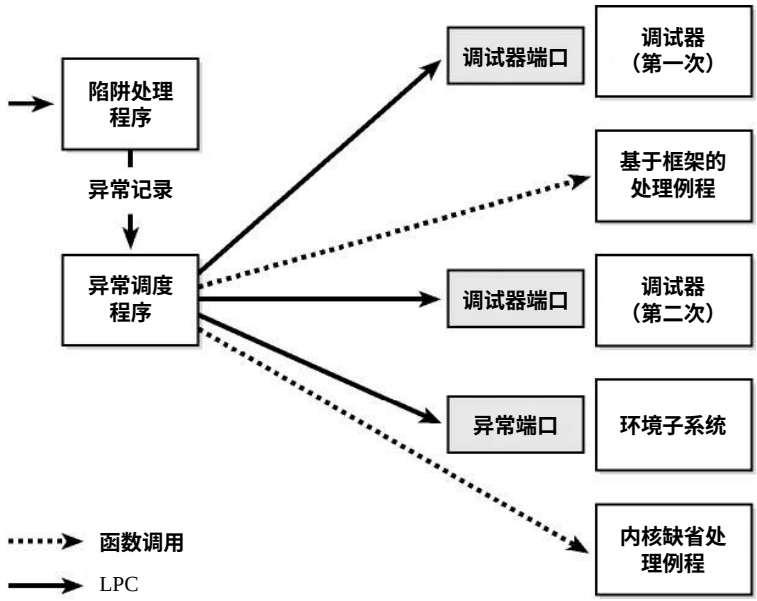


图2-11　Windows 2000/XP的异常调度

5. 系统服务调度

内核陷阱处理程序可以调度中断、异常以及系统服务调用。在 Alpha处理器上执行syscall指令或在Intel x86处理器上执行int2E指令都会引起系统服务调度，这两个指令都可以使系统陷入系统服务调度程序。被传递的数值参数指明了被请求的系统服务号，内核使用这个参数查找位于“系统服务调度表”(system service dispatch table）中的系统服务信息。这个表和中断调度表相似，只是每个入口包含了一个指向系统服务的指针，而不是一个指向中断处理例程的指针。

系统服务调度程序将校验参数，并将调用者的参数从线程的用户堆栈复制到它的核心堆栈中，然后再执行系统服务。如果传递给系统服务的参数指向用户空间的缓冲区，则核心态代码在访问缓冲区之前，会查明这些缓冲区的可访问性。

每个线程都有一个指向系统服务表的指针。 Windows 2000/XP有两个内置的系统服务表：第一个默认表定义了在 NTOSKRNL.EXE中实现的核心执行体系统服务；另一个表是在 Win32子系统Win32K.SYS的核心态部分中实现的 Win32 USER及GDI。当Win32 线程第一次调用 Win32 USER或GDI时，线程系统服务表的地址将指向包含 Win32 USER和GDI的服务表。

用于执行体服务的系统服务调度指令存于系统库 NTDLL.DLL中。子系统动态链接库通过调用NTDLL中的函数来实现它们的文档化函数。这里有一个例外是 Win32 USER及GDI，出于效率

的考虑，其中的系统服务调度指令是在 USER32.DLL和GDI32.DLL中直接实现的 —— 中间没有NTDLL.DLL。如图2-12所示。

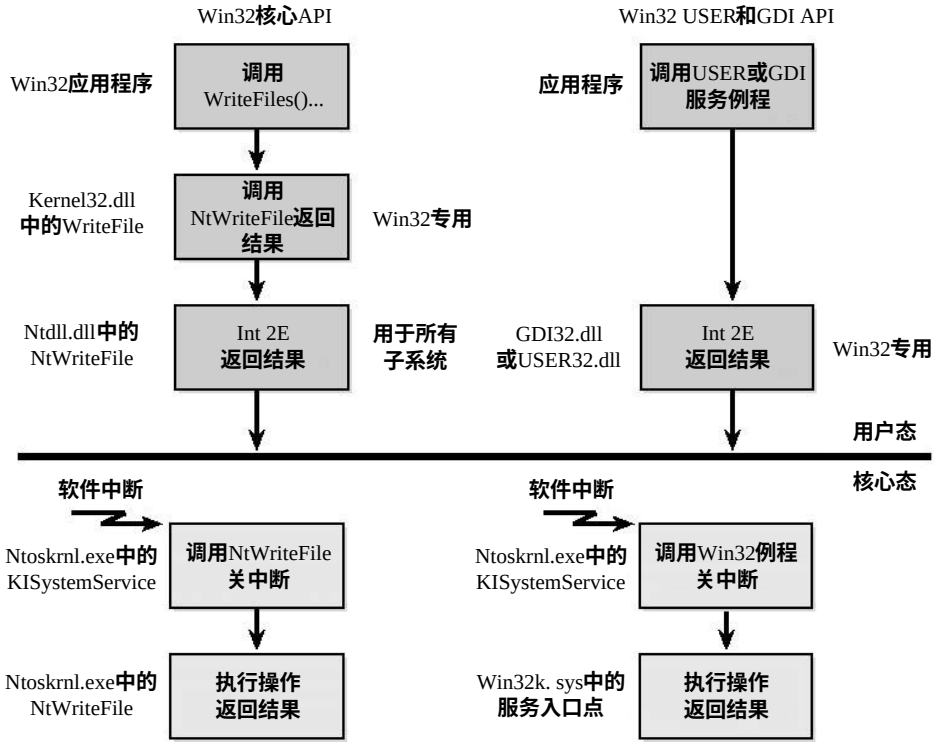


图2-12　系统服务调度

2.4.2　对象管理器

Windows 2000/XP通过对象管理机制为执行体中的各种内部服务提供一致的和安全的访问手段。对象管理器是一个用于创建、删除、保护和跟踪对象的执行体组件。对象管理器提供使用系统资源的公共、一致的机制；把对象保护孤立到操作系统的一个位置，实现安全保护。它提供了一种机制来控制进程使用对象，实现了资源的访问控制。对象管理器有一套对象命名方案和统一的保留规则，能够容易地操纵现有对象。同时，对象管理器能支持各种操作系统环境的需要。

Windows 2000/XP中有两种类型的对象：执行体对象和内核对象。执行体对象就是由执行体的各种组件实现的对象（例如进程管理器、内存管理器、 I/O子系统等）。内核对象是由内核实现的一个更原始的对象集合，这些对象对用户态代码是不可见的，它们仅在执行体内创建和使用。内核对象提供了一些基本性能，许多执行体对象内包含着一个或多个内核对象。

执行体对象和对象服务都是基本设施，环境子系统用它们来构造自己版本的对象和资源。执行体对象典型地由代表某个用户应用程序的环境子系统创建，或者由操作系统的不同组件作为它

们正常操作的一部分来创建。环境子系统为其应用程序提供的对象集一般与执行体所提供的对象集有些差异。Win32子系统使用执行体对象导出它自己的对象集，其中的大部分对象直接符合执行体对象。

每一个对象都有一个对象头和一个对象体。对象管理器控制对象头，各执行体组件控制它们自己创建的对象类型的对象体。另外，每一个对象头都指向打开该对象的进程的列表，同时还有一个叫做类型对象的特殊对象，它包含的信息对每一个对象的实例是公用的。

1. 对象头

对象管理器利用存储在对象头中的数据来管理对象，对象体的格式和内容对于它的对象类型来说则是唯一的。通过创建类型对象并为它提供服务，执行体组件可以控制这个类型的所有对象体的数据处理。对象管理器提供一个小的通用服务集用于操作对象头中存储的属性，不过基于效率和实现复杂度的考虑，每个对象都单独实现自己的创建、打开和查询服务。

2. 类型对象

对象头包含的数据对所有的对象是公用的，但是对于对象的每个实例，其数据可以取不同的值。例如，每个对象都有唯一的名称，并且可以有唯一的安全描述体。对象同样包含对特定类型的所有对象都是常数的一些数据。为了节省内存，对象管理器在创建一个新的对象类型时，就存储这些静态的指定对象类型的属性。它使用自己的叫做类型对象的对象来记录这些数据。因为对象管理器没有为它们提供服务，所以类型对象是不能从用户态直接进行处理的，不过，它们定义的一些属性通过某些本机服务和Win32 API程序则是可见的。

同步是指一个线程通过等待一个对象从一种状态转变为另一种状态来同步执行的能力。线程可以和执行体进程、线程、文件、事件、信号量、互斥体以及定时器对象同步。区域、端口、访问令牌、对象目录、符号链接、配置文件和键对象不支持同步。

3. 对象方法

方法包含与C++构造函数和析构函数类似的一组内部例程。对象管理器通过在另外一些情况下调用对象方法延伸了这个思想，有些对象类型指定方法，有些则不指定，这依赖于对象类型是如何被使用的。

当一个执行体组件创建一个新的对象类型时，它可以向对象管理器注册一个或多个方法。此后，对象管理器在那个类型对象的生命周期中预先定义好的点上可以调用这些方法，通常是在对象以某种方式被创建、删除或做某些修改时调用它们。

一个关闭方法的使用例子发生在I/O系统中。I/O管理器为文件对象类型注册了一个关闭方法，这样对象管理器在每次关闭文件对象句柄时都将调用此关闭方法。这个关闭方法检查正在关闭文件句柄的进程中是否有任何没有打开的文件锁，如果有，就把它移走。检查文件锁并不是对象管理器自己能够或应该做的。如果一个删除方法已经被注册，对象管理器在从内存中删除一个临时对象前会调用该删除方法。如果二级对象管理器找到一个存在于对象管理器名字空间之外的对象时，分析方法允许对象管理器把寻找对象的控制下放给一个二级对象管理器。当对象管理器寻找一个对象名时，一旦它遇到了有相应分析方法的对象，它就挂起搜索。对象管理器调用此分析方法，把正在搜索对象名的其余部分传递给它。

I/O系统使用的安全方法与分析方法相类似。无论何时线程要改变保护文件的安全信息时，就调用安全方法。文件的安全信息与其他对象的安全信息不同，因为安全信息被存储在文件自身而不是存储在内存中。因此，必须调用I/O系统来发现安全信息并改变它。

4. 对象句柄和进程句柄表

当进程通过名称来创建或打开一个对象时，它会收到一个代表进程访问对象的句柄。通过句柄指向对象比使用名称要快，因为对象管理器可以跳过名称搜索而直接找到对象。进程也可以通过在进程创建时继承句柄或从其他的进程接收复制句柄来为对象获得句柄。

所有用户态进程只有获得了对象句柄之后才可以使用这个对象。句柄作为系统资源的间接指针来使用，这种不直接的方式阻止了应用程序对系统数据结构直接地随便操作。

对象句柄提供另外的一些好处。首先，除了它们引用了什么以外，在文件句柄、事件句柄和进程句柄之间并没有不同。这种相似性为引用对象提供了统一的界面而忽略它们的类型。其次，对象管理器有创建句柄和定位句柄引用对象的专用权限。这就意味着对象管理器能够细察影响对象的每个用户态的操作，以检查调用者的安全配置文件是否允许在该对象上执行所请求的操作。

对象句柄是一个由执行体进程EPROCESS所指向的进入进程句柄表的索引。进程句柄表包含进程已为其打开句柄的所有对象的指针。它由一个固定的表头和一个大小可变的部分组成。大小可变的部分是一个句柄表入口数组，每一项描述了一个打开的句柄。如果一个进程打开了许多句柄，而且比可变部分能容纳的句柄多，则系统就将重新分配一个新的、更大的数组，并把旧数组复制到新的数组中。

每个句柄入口由一个包含两个32位成员的结构组成。第一个32位成员包含一个指向对象头的指针和三个标志。第一个标志是继承标志——即由这个进程创建的一些进程是否将在它们的句柄表中得到句柄的副本。第二个标志指出是否允许调用者关闭这个句柄。第三个标志指出在关闭对象时是否将生成一个审计消息。第二个32位成员是用于那个对象的已授权的访问掩码。

5. 对象安全

当进程创建对象或打开一个现存对象的句柄时必须指定一组期望的访问权限，此时对象管理器将调用安全引用监视程序（安全系统的核心态部分），并把进程的一组期望的访问权限集发送给它。安全引用监视程序将检查对象的安全描述体是否允许进程正在请求的访问类型。如果允许，安全引用监视程序将返回允许进程访问的一组授予访问权限，对象管理器将把它们存储在它创建的对象句柄中。

此后，无论何时当进程的线程使用句柄时，对象管理器都可以快速地检查在句柄中存储的已授权访问权限集是否符合由线程调用的对象服务隐含的用法。例如，如果调用者请求读取访问某个区域对象，但是在这之后他调用了一个对区域对象的写入服务，那么写入就会失败。

6. 对象保留

因为所有访问对象的用户态进程必须首先为它打开一个句柄，所以对象管理器可以很容易地跟踪有多少进程，甚至是哪些进程正在使用该对象。跟踪这些句柄代表了实现对象保留的第一步——即只在使用对象时临时保留对象，然后删除它们。

对象管理器分两个阶段实现对象保留。第一个阶段叫做“名称保留”，它是由对象的打开句

柄数目来控制的。每次进程为对象打开句柄时，对象管理器就增加对象头中的打开句柄计数器值。当进程使用完对象并且关闭它的句柄时，对象管理器就减少打开句柄计数器值。当计数器减少到零时，对象管理器就从全局名字空间中删除对象名。这个删除防止了新进程为这个对象打开句柄。对象保留的第二阶段是当对象不再被使用时，停止保留它们。因为操作系统代码经常使用指针代替句柄访问对象，所以对象管理器必须同样记录有多少对象指针已经分配给操作系统进程。每次对象管理器给对象分配一个指针，它就为此增加一个引用计数；当核心态组件结束使用该指针时，调用对象管理器减少对象的引用计数。所以，即使在一个对象的打开句柄计数器值为零以后，该对象的引用计数仍然可能是正的，这就表明操作系统仍然在使用该对象。最终，引用计数也会降为0。当这种情况发生时，对象管理就从内存中删除该对象。

根据对象保留的工作方法，只要简单地为对象保持一个打开它的句柄，应用程序就能确保对象和它的句柄仍然保留在内存之中。

7. 资源记账

资源记账像对象保留一样，与使用对象句柄密切相关。资源对象打开的句柄计数为正，表明某个进程正在使用其资源，它同样指明该进程由于对象占用内存而正被记入账内。

许多操作系统使用一个配额系统来限制进程对系统资源的访问。然而，为进程分配的配额类型有时是多样的、复杂的，并且跟踪配额的代码分布在操作系统内各处。 Windows 2000/XP的对象管理器为资源记账提供中心服务。每一个对象头包含一个称为配额账的属性来记录当进程中的一个线程为对象打开句柄时对象管理器从分配给进程的页交换区和 /或非页交换区配额中减去了多少。

在Windows 2000/XP上的每个进程都指向一个配额结构 ,它记录了用于非页交换区 /页交换区和页面文件用法的限制和当前值。然而，在您的交互式会话中，所有进程共享同一个配额块，系统进程没有配额限制。

页交换区的配额大小从 521KB开始，非页交换区的配额大小从 64KB开始。这种限制是“ 柔性的 ”，当进程超出配额限制时，系统会试图自动增加进程配额。如果打开一个对象将超出页交换区或非页交换区配额，系统就将调用内存管理器看看能否增加配额，如果不能，那么打开对象的请求就会失败，并将给出“ 超出配额 ”的错误。但是在大多数系统中，配额会继续根据需要而增加。

8. 对象名

创建大批对象的一个重要考虑是如何跟踪它们。对象管理器提供了把某个对象与其他对象区别的方法以及查找并检索一个特定对象的方法。

这两点通过给对象命名实现，执行体允许任何由对象所代表的资源有一个名称。如果对象管理器用名称存储对象，就能通过寻找名称来查找对象。对象名还使得对象可以在不同进程间共享。执行体对象名字空间是全局性的，对系统中所有过程可见。一个进程可以创建一个对象并将它的名称放入全局名字空间，而且第二个进程可以通过指定对象名称来为对象打开句柄。如果对象不想以这种方式共享，它的创建者不需要给它命名。

为了提高效率，对象管理器并非在每次有人使用对象时就检索对象名，而只在两种情况下寻

找名称，第一种情况是当进程创建一个名称的对象时，对象管理器要在名字空间中寻找这个名称以证实它还没有存在；第二种情况是当进程为一个命名的对象打开句柄时，对象管理器寻找名称并找到对象，然后给调用者返回一个对象句柄。

基本的核心对象（如互斥体、事件、信号量、可等待定时器和区域）将各自的名称存储在同一个对象目录中，它们是不能重名的。对象名对单个计算机来说是全局性的，但它们在网络上是不可见的。

对象目录对象是对象管理器支持这种层次命名结构的方法。这个对象与文件系统目录类似，并且包含其他对象，甚至可能是对象目录的名称。对象目录对象保持了足够的信息将这些对象名称转化成指向对象自身的指针。

在某些文件系统中，符号链接使用户能创建一个文件名或目录名，而当使用时，它被操作系统转化成不同的文件或目录名。使用符号链接是一个简单的方法，它允许用户间接地共享一个文件或目录的内容，来用的是在通常层次目录结构中不同目录之间创建交叉链接的方法。对象管理器实现一个叫做符号链接对象的对象，符号链接对象为在它的对象名空间内的对象名执行一个相似的功能。符号链接可以在对象名字符串中的任何地方发生。执行体使用符号链接对象的一个场合是把MS-DOS中的设备名转换成Windows 2000/XP的内部设备名。此外，用户可以使用 subst命令添加伪驱动器名或映射一个驱动器名到网络共享。一旦它们被创建，对于在系统内的所有进程，它们都必须是可见的。

2.4.3 同步

互斥指的是保证有一个、并且每次只有一个线程可以访问一个特殊的资源。当资源不支持它自身被共享访问或当共享时会导致意外的输出时，互斥是必要的。互斥对于紧耦合的支持对称多处理的操作系统来说尤其重要，在这些操作系统中，相同的系统代码能同时在多个处理器上运行，共享存储在共用内存中的某些数据。在 Windows 2000/XP中，内核的工作是提供一种机制，它可以使用系统代码来防止两个线程同时修改同一个结构。内核提供简单的互斥，它和执行体的其余部分用于同步对共用数据结构的访问。

1. 内核同步

在内核执行的不同阶段，它必须保证有一个、并且每次只有一个处理器在临界区执行。内核的临界区是修改共用数据结构的代码段，例如内核的调度程序数据库或它的 DPC队列。只有在内核能保证线程以互斥方式访问这些数据结构时，操作系统才能正常工作。

所涉及的最大问题是中断。当中断产生时，它的中断处理例程会修改公共数据结构，而此时内核也许正在更新这个共用数据结构。 Windows 2000/XP内核在使用共用资源以前，将暂时屏蔽那些其中断处理程序也要使用该资源的中断。它通过把处理器的“ 中断请求级 ”（IRQL）提高到由任意潜在的访问全局数据的中断资源使用的最高级来达到这个目的。

这种策略对于单处理器系统是很好的，但不适用于多处理器结构。内核用自旋锁实现了多处理器互斥机制。自旋锁是一个与共用数据结构有关的锁定机制，如图 2-13。

在进入图2-13中所示的两个临界区中的一个之前，内核必须获得与所保护的 DPC队列有关的

自旋锁。如果自旋锁非空，内核将一直尝试得到锁直到成功。因为内核被保持在过渡状态“旋转”，直到它获得锁，所以自旋锁由此而得名。

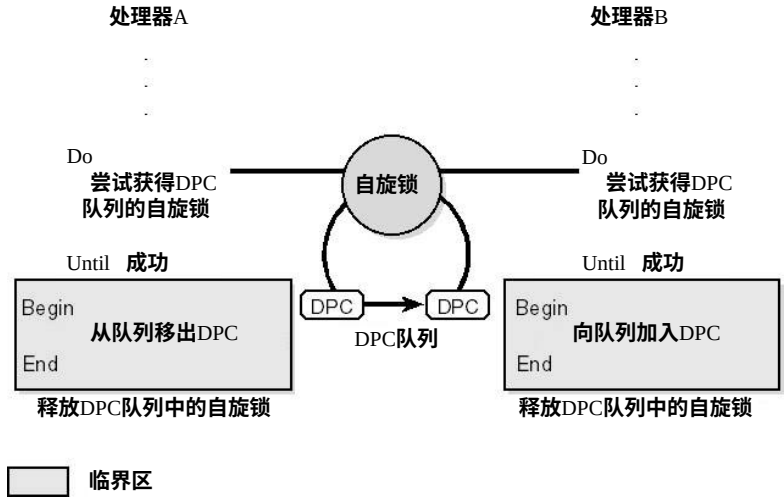


图2-13　自旋锁的使用

自旋锁像它们所保护的数据结构一样，存储在共用内存中。为了提高速度和使用任何在处理器体系下提供的锁定机构，获取和释放自旋锁的代码是用汇编语言写成的。在很多体系结构中，自旋锁是使用一个硬件支持的测试并设定单一指令来实现的，它测试锁变量值并在一条基本指令中获得锁。在一条指令中测试和获得锁，防止了当第一个线程在测试变量到获得锁的这一时间段内第二个线程夺取锁。

当线程试图获得自旋锁时，在处理器上的所有其他工作将终止。因此，拥有自旋锁的线程永远不会被抢先，但允许它继续执行以使它尽快把锁释放。内核对于使用自旋锁十分小心，当它拥有自旋锁时，它执行的指令数将减到最少。

通过一组内核函数，内核使执行体的其他部分也可以使用自旋锁。

2. 执行体同步

在多处理器环境下，在内核之外的执行体软件同样需要同步访问共用数据结构。自旋锁只是部分满足了执行体对同步机制的需要，这是因为在自旋锁上等待，实际上是使处理器暂停，自旋锁只可以在下列严格受限的环境使用：

- 被保护的资源必须被快速访问，并且没有与其他代码的复杂的交互作用。
- 临界区代码不能换出内存，不能引用可分页数据，不能调用外部程序（包括系统服务），不能生成中断或异常情况。

内核以内核对象的形式给执行体提供额外的同步机构，统称为调度程序对象。 Win32应用程序中的一个线程可以与一个Win32的进程、线程、事件、信号量、互斥体、可等待定时器、 I/O完成接口或文件对象同步。执行体同步对象的类型叫做执行体资源，它实现了独占访问（如互斥体）以及共享只读访问。但是，它们仅对核心态代码是可以使用的，而不能从 Win32 API访问。资源

不是调度程序对象，而是从非页交换区直接分配的数据结构，它们有自己的专门服务来初始化、锁定、释放、查询和等待。

(1) 等待调度程序对象

一个线程通过等待对象的句柄可以与调度程序对象同步，这样做会使内核将线程挂起并把它的调度状态从运行改为等待。在任何给定时刻，同步对象会处于“有信号状态”或“无信号状态”。一个线程在内核将其调度状态由等待状态改为就绪状态时才能恢复它的执行。当线程正在等待其句柄的调度程序对象也在经历一次状态改变时，这种改变就会发生。线程调用由对象管理器提供的等待系统服务程序来与对象同步，给希望同步的对象传递句柄。线程可以等待一个或几个对象，如果等待在一定时间内没有结束，线程也可以指定取消它的等待。无论什么时候当内核将一个对象设置为有信号状态时，它将检查是否有线程在等待这个对象。如果有，内核会把一个或几个线程从它们的等待状态释放，这样它们就能继续执行。

下面设置事件的例子说明了同步是如何与线程调度互相作用的（如图 2-14）。

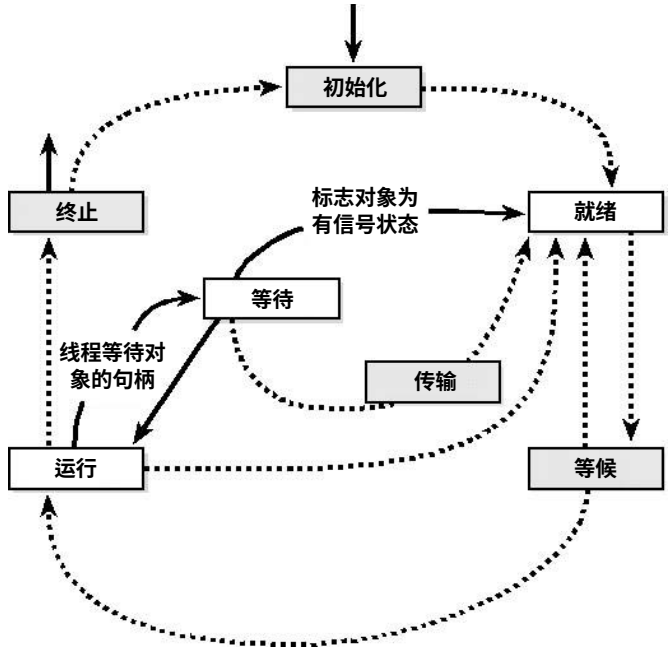


图2-14　等待一个调度程序对象

1) 用户态线程等待事件对象句柄。

2) 内核把线程的调度状态由就绪改为等待，然后将线程添加到等待事件的线程列表中。

3) 另一个线程设置事件。

4) 内核从等待事件的线程列表向下匹配。如果满足了线程的等待条件，内核就把线程的状态由等待改为就绪。如果是可变优先级的线程，内核也可能提高它的执行优先级。

5) 因为新线程已经成为就绪状态而去执行，所以调度程序将重新安排线程。如果它发现正在

运行线程的优先级比新的就绪线程的优先级低时，它就会抢先低优先级线程，并发布一个软件中断为高优先级的线程初始化描述表切换。

6) 如果没有处理器可以被抢先，调度程序将把就绪线程放入调度程序就绪队列中，以备稍后调度。

(2) 何种情况把对象置为有信号状态

对于不同对象，有信号状态的定义不同。在线程对象的生存期内，它处于无信号状态；当线程终止时，内核把它设置为有信号状态。类似地，当进程的最后一个线程终止时，内核把进程对象设置为有信号状态。与之相反，定时器对象（像闹钟一样）在一个确定的时间被设置为“发声”，当它的时间期满以后，内核设置定时器对象为有信号状态。

在选择同步机制时，程序必须考虑管理不同的同步对象行为的规则。当对象被设置为有信号状态时，线程的等待是否结束取决于线程正在等待的对象类型。当一个对象被设置为有信号状态时，处于等待状态的线程通常被立刻释放。

通知事件对象被用来宣布某些事件的发生。当事件对象被设置为有信号状态时，将释放所有正等待该事件的线程；但当一个线程每次等待的事件多于一件时除外。在另外的对象达到有信号状态以前，这样的线程可能将继续等待。

与事件对象相反，互斥体有与它自身相关的所有权。它被用来获得对某个资源的互斥访问，并且每次只有一个线程能拥有互斥体。当互斥体对象空闲，内核就将它设置为有信号状态并选择一个正在等待的线程去执行，被内核先中的线程将获得互斥体对象，而所有其他的线程都将继续等待。

(3) 数据结构

对于跟踪谁在等待什么，有两个数据结构是很关键的：调度程序头和等待块。在 DDK的包含文件ntddk.h中的这两种结构是公开的。

```
typedef struct_DISPATCHER_HEADE
{
    UCHAR Type;
    UCHAR Absolute;
    UCHAR Size;
    UCHAR Inserted;
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISRPATCHER_HEADER;
typedef struct _KWAIT_BLOCK
{
    LIST _ENTRY WaitListEntry
    struct _KTHREAD *RESTRICTED_POINTER Thread;
    PVOID object;
    struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
    USHORT waitKey;
    USHORT waitType;
}WAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PRKWAIT_BLOCK;
```

调度程序头包含了对类型、有信号状态和正在等待对象线程的列表。等待块代表了正在等待对象的线程。每个处于等待状态的线程都有等待块列表，它代表了线程正在等待的对象。每个调度程序对象都有一个等待块列表，代表正在等待该对象的是哪些线程。这个列表将被一直保存，这样当调度程序对象处于有信号状态时，内核就能迅速判断谁正在等待这个对象。等待块有一个指向被等待对象的指针、一个指向等待该对象的线程指针和一个指向下一个等待块的指针。指针也记录等待类型以及在句柄数组中入口的位置，该位置是由线程在调用 WaitForMultipleObjects 函数时传送的（若线程只等待一个对象时为 0）。

2.4.4 本地过程调用

本地过程调用（LPC）是一个用于高速信息传输的进程间通信机构，在 Win32 API下它是不可用的；它是一个对Windows 2000/XP操作系统组件有效的内部机制。下面是一些使用LPC的例子：

- 远程过程调用使用LPC在同一个系统中的进程之间通信。
- 少数Win32APL导致给Win32子系统进程发送信息。
- WinLogon使用LPC与LSASS通信。
- 安全引用监视器，使用LPC与LSASS进程通信。

典型地，在一个服务器进程与该服务器的一个或多个客户进程之间的两个用户态进程之间或一个核心态组件和一个用户态进程之间可以建立 LPC连接。

LPC被设计成允许三种交换信息的方法：

1) 使用包含信息的缓冲区调用 LPC可以发送少于256字节的信息。然后，这个信息又从发送进程的地址空间复制到系统地址空间，再从那里拷贝到接收进程的地址空间。

2) 如果用户和服务器想交换大于 256字节的数据，那么他们可以选择使用双方都映射了的共享区。发送方将信息数据放到共享区，然后向接收方发送一小段信息表明在共享区的什么地方可以找到数据。

3) 当服务器想读或写大量数据，而共享区又太小时，数据可以直接从客户地址空间读出或向客户地址空间写入。LPC组件提供了两个函数，服务器可以用它们来完成这些操作。以第一种方式发送的信息被用于同步正在传送的信息。

LPC导出一个称为端口对象的单个执行体对象，用它来保持通信所需要的状态。尽管 LPC使用单个对象类型，但是它有几个端口：① 服务器连接端口是一个已命名的服务器连接请求指向端口，客户可通过与这个端口连接从而连接到服务器上；② 服务器通信端口是一个未命名的用于特殊通信的端口，服务器与每一个活动客户都有一个这样的端口；③ 客户通信端口是特殊客户线程用来与特殊服务器通信的未命名的端口；④ 未命名通信端口是为用于同一进程中的两个线程而创建的未命名的端口。

LPC典型地应用于以下情况：服务器创建已命名的服务器连接端口对象。客户提出与这个端口连接的请求。如果同意该请求，客户通信端口和服务器通信端口就会被创建。客户得到了客户通信端口的句柄，服务器得到服务器通信端口的句柄。然后客户和服务器将为了它们之间的通信而使用一些新的端口。

2.4.5 系统工作线程

在系统初始化的过程中，Windows 2000/XP 在系统进程中产生了很多线程，成为系统工作线程，它们独立的存在，代表其他线程履行职责。很多时候，线程运行在延迟过程调用或者调度级别时需要调用在较低中断请求级别才能执行的功能，而当它们不能降低中断请求级别的时候，它们就要将处理权交给一个运行在更低中断请求级别的工作线程。

有些设备驱动程序和执行体组件产生它们自己的线程用于在被动级别处理工作；不过，大多数设备驱动程序和执行体组件使用系统工作线程，因为它们避免了由于增加额外系统线程而造成的不必要的调度和内存开销。设备驱动程序和执行体组件通过调用 ExQueueWorkItem和IoQueueWorkItem请求一个系统工作线程。这些功能放置一个工作项在工作线程寻找工作的队列调度器对象上。工作项包括一个指向例程的指针和工作线程处理这些工作项时需要的参数。这些例程由相应的请求被动级别执行的设备驱动程序和执行体组件实现。

有三种系统工作线程：① 延迟工作线程运行在优先级 12，处理非时间关键的工作项，它们的堆栈在等待工作项时可以被换出到页交换文件；② 关键工作线程运行在优先级 13，处理时间关键的工作项，在Windows2000 server中它们的堆栈始终在物理内存中；③ 一个单独的高度关键的工作线程运行在优先级15，堆栈也始终在物理内存中，处理管理器使用这种工作线程的“收割机”功能释放终止的线程。

2.5 Windows 2000/XP的注册表

注册表在配置和管理Windows 2000系统的机制中扮演着关键的角色。它存储着所有关于系统和每个用户的设置信息。

2.5.1 注册表的数据类型

注册表是一个数据库，它的结构与磁盘的逻辑结构相似。注册表包括主键和键值，它们之间的关系就像磁盘上的目录和文件一样。一个主键可以包含若干个主键（或称为这个主键的子键）和键值，而键值则存储数据，顶级主键称为根键。

主键和键值借用了文件系统的命名习惯。每个键值都有一个名称，我们可以用这个名称唯一确定一个键值。在这个命名机制中，唯一的例外是未命名的主键，我们可以从两个不同的注册表编辑器中看出其中区别，regedit用<default>来代表未命名的键值，而 regedt32则用<no name>来表示未命名的键值。

键值可以存储不同类型的数据。注册表中大多数的键值是 REG_DWORD、 REG_BINARY或REG_SZ类型的。REG_DWORD存储数值或布尔类型的数据；REG_BINARY可以存储任意长度的二进制数以及一些原始数据，如加密的口令；REG_BINARY存储Unicode的字符串，比如姓名、文件名、路径和类型。

REG_LINK类型比较特别，它可以让一个主键显式地指向另一个主键。当访问这种类型的主键时，系统会沿着链接直到目的地。 Windows 2000/XP在很多情况下都用到了指针类型的键值，

在6个根主键中有3个根主键是指向另3个非指针类型根主键的子键。指针类型的主键不会被存储，它们必须在每次系统启动时被动态地创建。

2.5.2 注册表的逻辑结构

注册表有6个根主键，这些根主键不可以被删除，也不能添加新的根主键，它们存储的信息如下：

- HKEY_CURRENT_USER 存储与当前登录用户有关的信息。
- HKEY_USER 存储了所有用户的信息。
- HKEY_CLASSES_ROOT 存储与文件类型和COM对象相关的信息。
- HKEY_LOCAL_MACHINE 存储与系统设置相关的信息。
- HKEY_PERFORMANCE_DATA 存储与系统性能相关的信息。
- HKEY_CURRENT_CONFIG 存储了当前硬件配置文件的信息。

1. HKEY_CURRENT_USER

HKCU 包含了与当前登录用户相关的软件配置和参数。它对应当前用户的用户配置文件，这个文件在磁盘上的位置是：“ \Documents and Settings\<username>\ntuser.dat ”。当系统载入用户配置文件（比如登录或这个用户调用一个服务进程时），系统将创建一个指向HKEY_USERS下代表该用户的子键的HKCU类型的链接。

2. HKEY_USERS

它包含了为每一个用户配置文件所创建的子键以及用户在系统数据库中注册的类信息。它还包括一个叫做“ HKU\.DEFAULT ”的子键，这个子键指向缺省的工作站配置文件。

3. HKEY_CLASSES_ROOT

这个主键包括文件扩展名和COM组件类的注册信息。大多数的主键都包含一个 REG_SZ类型的键值，它指向HKCR中与存储这种文件扩展名信息相关的另一个主键。比如，“ HKCR\.xls ”指向与Microsoft Excel文件信息相关的主键（如“ HKCU\Excel.Sheet.8 ”）。其他主键包含了那些已在系统里注册过的COM组件的配置信息。

HKEY_CLASSES_ROOT下的数据来自两个方面：

- HKCU\SOFTWARE\Classes包括了用户的注册类（对应的磁盘文件映射是“ \Documents and Settings\<username> \Local Settings\Application Data\Microsoft\Windows\Usrclass.dat ”）。
- HKLM\SOFTWARE\Classes包括了系统的注册类。

4. HKEY_LOCAL_MACHINE

HKLM根主键包含了所有有关整个系统的配置信息的子键： HARDWARE、SAM、SECURITY、SOFTWARE和SYSTEM.

HKLM\HARDWARE子键包含系统硬件配置脚本和所有设备驱动程序的映像。设备管理工具所提供的系统的硬件配置信息就是直接从 HARDWARE这个主键中读出的。

HKLM\SAM子键保存本地的用户和组信息，如用户密码、组和域的定义。和域控制器相似，Windows 2000/XP Server将域和组的用户信息存储在活动目录里，这个活动目录实际是一个存储

域级设置和信息的数据库。在缺省状态下， SAM主键的安全级别会设为对任何用户（包括系统管理员）都不可访问。如果你想要了解这个主键的内容，你可以将这个主键的安全级别修改为对管理员可读，但你可能得不到什么有用的信息。

HKLM\SECURITY子键存储了系统的安全策略以及用户的权限设置。上面所提到的HKLM\SAM事实上是链接到HKLM\SECURITY\SAM下的SECURITY子键的。在缺省情况下，你是不可能看到HKLM\SECURITY或 HKLM\SECURITY\SAM的内容的，因为它们的安全级别被设为系统级用户才可以访问。

HKLM\SOFTWARE子键存储了一些Windows 2000系统级软件配置信息，但通常要使对这些信息的更改生效是不用重新启动系统的。这个子键还存储了第三方应用程序的系统级设置，如应用程序中文件和目录的路径，以及有关使用许可证和期限的信息。

HKLM\SYSTEM 子键同样存储了一些系统级的配置信息，但我们必须重新启动系统才可以使对这些信息的更改生效，比如加载哪些设备驱动程序或是启动哪些服务。因为这些信息对于系统启动是至关重要的，所以Windows 2000/XP对这部分信息加以备份。当由于改变的配置而使得系统不能启动时，管理员可以选择原先的备份来重新启动系统。

5. HKEY_PERFORMANCE_DATA

注册表提供HKEY_PERFORMANCE_DATA根主键使得用户可以访问Windows 2000的系统性能评估数据，其中包括操作系统组件和服务器应用程序的性能数据。这个主键在注册表编辑器里是不可见的，只能通过Win32API的注册表函数来操纵它。

6. HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG主键只是一个指向当前硬件配置文件的链接，这个文件存储在“HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current”下。硬件配置文件使管理员能为系统的基本驱动程序设置参数。尽管配置文件可能在每次启动时有所改变，但应用程序总能通过这个主键的键值取得当前活动的配置文件。

2.6　Windows 2000/XP服务

几乎每一个操作系统都有一个在系统启动时自动运行服务进程的机制。在 Windows 2000/XP里，这些进程被称之为服务（service）。这些服务进程类似于UNIX的守护进程，在客户/服务器应用程序中扮演服务器角色。一个Win32 服务的例子就是Web 服务器，由于它不论任何用户登录计算机都必须运行，而且在系统启动的时候就会开始运行，因此管理员不必记得甚至不用经过提醒就可以启动它。

Win32 服务由三部组成：一个服务应用程序；一个服务控制程序（ SCP）；服务控制管理器（SCM）。

2.6.1　服务应用程序

类似Web服务这样的服务应用程序，至少由一种Win32 可执行程序组成。用户用SCP来启动、停止或者配置服务。虽然Windows 2000/XP 内置的SCP提供集成好的功能来实现对一般的启动、

停止、暂停和继续的控制，但是一些服务应用还是包括了它们自带的 SCP以便允许管理员来指定他们针对特殊服务的配置。

简单的服务应用程序类似于 Win32 可执行程序，其中有附加的代码接收来自 SCM的命令，并且将服务应用程序的状态反馈给 SCM。由于大多数的服务应用没有用户界面，因此它们被编译成控制台程序。

当安装了包含服务的应用程序后，安装程序必须向系统注册这个服务。安装程序调用 Win32的CreateService 函数实现注册。Asvapi32（高级应用程序接口动态链接库，Advanced API DLL）包括了所有的客户端SCM应用程序接口。

注册了一个服务之后，一条消息将会被发送到 SCM中服务的所在地，然后 SCM将为服务在“HKLM\SYSTEM\Currentcontrolset\Services”新建一个注册主键。每一个服务的独立的注册主键定义了可执行程序的包括服务、参数、配置内容的镜像的路径。

新建服务后，可以通过 StartService函数启动一个安装或管理应用程序。因为一些基于服务的应用程序在启动时必须初始化，所以下述情况并不少见：安装程序将服务注册成为一个自动启动的服务，要求用户重启完成安装，由 SCM在系统启动时启动服务。

当一个程序调用CreateService函数的时候，它必须指定描述服务特征的一系列参数。服务的特征包括：服务类型；服务可执行的程序镜像文件的地址；一个可选择显示名；一个可选择的帐号名和密码用来启动在一个特殊的帐号安全关联的服务；启动类型指明在系统启动时还是在 SCP的指示下自动启动；错误处理代码则指明在服务启动时监测到有错误的情况下系统的对策，并且包括了假如服务为自启时当此服务与其他服务有关联启动的时候指定的可选信息。

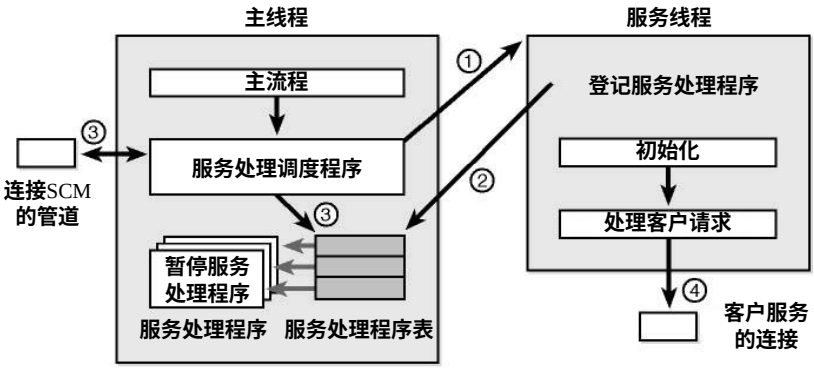
SCM在服务注册表主键中以键值的形式保存了每种服务特征。如果服务需要保存服务的私有配置信息，通常会建立一个被称为参量的子键，在此子键中以键值的形式保存配置信息。此项服务就可以通过标准的的注册表函数找到这些配置信息。

当SCM启动一个服务进程的时候，这个进程立刻调用 StartServiceCtrlDispatcher函数，此函数接收一个服务入口列表或单个服务进程的单个入口。每个入口点通过与入口通信的服务名来鉴别。在建立了一个命名管道同SCM通信之后，此函数陷入循环等待来自管道的 SCM的命令。SCM在每一次启动进程所属的服务时发送一个服务启动命令。而 StartServiceCtrlDispatcher函数每收到一次服务启动命令就创建一个线程（叫做服务线程）来调用启动服务的入口和执行服务的循环命令。

StartServiceCtrlDispatcher不确定的等待来自 SCM的命令，在所有进程的服务线程都停止并允许进程在其离开时清除资源后，才将控制权交还进程的主函数。

服务的入口点的第一个动作是调用 RegisterServiceCtrlHandler函数。这个函数接收和保存了一张这个服务用以处理来自 SCM的各种命令的函数列表。 RegisterServiceCtrlHandler并不同SCM通信，但是它为StartServiceCtrlDispatcher函数在本地进程内存中保存了这张表。然后，服务入口继续初始化服务，包括分配内存、创建通信终端、从注册表中读取私有配置信息。在初始化服务的同时，入口点可能会阶段性地发送状态信息给 SCM表明服务的启动的执行情况。在初始化结束后，服务线程通常陷入循环等待来自客户端的请求。以 Web服务为例，它会初始化一个 TCP监听socket，然后等待正确的HTTP连接请求。

服务进程的主线程由StartServiceCtrlDispatcher函数执行，它接收在服务进程中指示的SCM命令并且使用处理函数的列表定位和调用可信的服务函数来响应命令。SCM命令包括停止、暂停、恢复、查询和关闭等，当然也包括应用程序定义的命令。图2-16给出了一个服务进程的内在组成，描述了一个双线程进程组成的一个服务：主线程和服务线程。



①服务处理调度程序启动服务线程。
②服务线程登记服务处理程序。
③服务处理调度程序调用服务处理程序响应SCM命令。
④服务线程处理用户请求。

图2-15　服务的组成

2.6.2　服务帐号

对于服务开发者和系统管理员来说，服务的安全环境是一个非常重要的考虑因素。一般来说，除非服务的安装程序或管理员指定，服务运行于本地系统帐号的安全环境下。

1. 本地系统帐号

本地系统帐号与所有Windows 2000的用户模式操作系统组件运行的帐号一样，包括会话管理器、Win32子系统进程和Winlogon进程。从安全的远景来看，本地系统帐号非常强大——在本地系统上它比任何本地或域的帐号在本地安全权利都要大。这种帐号是本地管理员组的一个成员，具备授权所有实质性特权的权利。许多文件和注册表主键赋予本地系统帐号完全的存取访问权限。在本地系统帐号下运行的进程运行于缺省用户的状态（HKU\DEFAULT），因而不能访问保存在其他用户帐号里的配置信息。

当系统是Windows 2000/XP的一个域成员时，本地系统帐号就包含了服务进程运行于计算机的安全标志（Secuity IDentifier，SID）。因此，通过计算机域帐号，运行于本地系统帐号的服务将和其他域树中的计算机中的服务一起被自动鉴别出来。除非机器帐号被明确地赋予了对资源的访问存取权限，否则进程可以访问允许无会话的网络资源。

2. 备用帐号运行服务

一些服务由于我们所描述的限制，必须有用户的帐号的安全证书才可以运行。可以配置一个服务运行于备用帐号，假定此服务由一个帐号和密码所创建或指定。服务必须在Windows 2000

服务MMC snap-in下运行。

2.6.3 交互式服务

另一个在本地系统帐号下运行服务的限制是它们不能在交互式的用户桌面上显示对话框或者窗口。这个限制并不是在本地系统帐号下运行的直接结果，而是服务控制器指派服务进程给Windows工作站（Windows station）的方法导致的结果。

Win32子系统用Windows工作站把每一个Win32进程联系起来，它包含了桌面，而桌面包含了视窗。在一个控制台上只有一个Windows工作站可见并且接收用户鼠标、键盘的输入信息。在服务环境终端，每一个Windows工作站的会话都是可见的，但是服务都是作为控制台会话的一部分运行。Win32把这种可见的Windows工作站称之为WinSta0，并且所有交互式的进程都对其进行访问。

除非有其他的命令，否则SCM把服务用一个不可见的名为Service-0x0-3e7$的Windows工作站联系起来，这个Windows工作站是所有非交互式的服务所共享的。

这个Windows工作站名字中的数字——3e7——代表了LSASS指派给登录会话的会话登录标志，登录会话被SCM用来在本地系统帐号下运行非交互式的服务。

被配置成在用户帐号状态下运行的服务运行于有区别的不可见 Windows工作站状态，并以Lsass指派给登录会话的标志命名。

无论是运行于用户帐号还是本地系统帐号下，没有运行于可见的 Windows工作站下的服务都不能在控制台上接收输入或者显示窗口。事实上，如果一个服务在此 Windows工作站下弹出了一个普通对话框，那么此服务将会表现出挂起状态，因为没有用户能见到这个对话框，所以当然阻止了用户用鼠标或键盘关闭它而使此服务继续执行下去。

虽然不常见，一些服务还是需要通过对话框或窗体同用户进行交互。一个含有这种需要的Windows 2000/XP内置的服务例子就是Windows 安装程序。为了配置一个拥有同用户交互权利的服务，在服务注册的主键类型参数中必须含有SERVICE_INTERATIVE_PROCESS modifier。

当SCM启动一项标记为交互式的服务时，它在本地系统帐号安全环境下启动服务进程，并且把服务连接到Winsta0而不是非交互式的Windows工作站。这项连接允许服务在控制台上显示对话框和窗体的对用户的输入反馈。

2.6.4 服务控制器

SCM 的可执行文件是“ \Winnt\System32\Services.exe ”，和许多服务进程一样，它以Win32 控制台程序模式运行，Winlogon进程在系统启动早期启动SCM。SvcCtrlMain在紧接着屏幕变为空白桌面的时刻运行，通常在Winlogon装载图形化身份鉴定并给出登录界面（GINA）之前运行。

SvcCtrlMain首先创建一个nonsignaled初始化的名为SvcCtrlEvent_A3752DX的同步事件，只有在完成准备接收来自 SCP的命令的必要步骤后， SCM才设定此事件为signaled状态。SCP通过一个对话来确认SCM的函数是OpenSCManager。这个函数通过等待SvcCtrlEvent_A3752DX变成signaled来阻止SCP试图在SCM初始化完成之前接触SCM 。

然后SvcCtrlMain开始工作并调用ScCreateServiceDB，这个函数建立了SCM的内部服务数据库。ScCreateServiceDB存取“HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List”中的内容，这是一个列出了定义好的服务组名称和命令的REG_MULTI_SZ值。服务的注册表主键值包括了可选的组键值，表示服务或者设备驱动是否需要从其他的服务组角度考虑而控制它的启动顺序。举例来说，Windows 2000的网络堆栈是从下向上建立的，所以，在它们的启动顺序中，网络服务必须指定组键值来把它放到网络设备的后面。 SCM内建了一个组列表保存了从注册表键值读出的组顺序。组包括NDIS、TDI、Primary Disk、Keyboard Port和Keyboard Class。附加的和第三方的应用软件可以定义自己的组并把它们加入列表。微软公司的事件服务器就加了一个名为MS Transactions的组。

ScCreateServiceDB随后搜索“HKLM\SYSTEM\CurrentControlSet \Services”里的内容，在服务的数据库里为每一个遇到的主键创建一个条目。一个数据库的条目包括所有的为一个服务定义好的服务关联参数和跟踪服务状态的域。由于SCM启动标记为自启的服务和设备驱动并且监测标记为引导启动和系统启动的驱动器启动错误，因此SCM为服务和设备驱动加入了这些条目。在所有用户模式进程执行前， I/O管理器装载标记为引导启动和系统启动的驱动器，因此任何有这些启动标记的驱动器将在SCM启动前被装载。

ScCreateServiceDB读取服务的组键值来确定此服务在组中的成员资格，并且把这个键值同早先被建的组列表联系起来。这个函数还通过查询服务的 DependOnGroup和DependOnService 注册表键值在数据库中读取记录服务的组和从属信息。

在服务启动的时候，SCM有可能需要调用Lsass，所以SCM等待Lsass在其初始化结束时通知LSA_RPC_SERVER_ACTIVE同步事件，Winlogon也启动Lsass进程，所以SCM同Lsass的初始化是同步的，而且它们两个的初始化结束顺序是不确定的。 SvcCtrlMain调用ScGetBootAndSystemDriverState遍历服务数据库来查询引导启动和系统启动的设备驱动器条目。 ScGetBootAndSystemDriverState通过查询驱动器在对象管理器中的名字域目录 \Driver中的名字来确定驱动器是否成功的启动。当一个设备驱动器被成功装载， I/O管理器在它的目录下把驱动器对象插入到名字域，所以如果它的名字没有给出，那么它就不能被装载。如果一个驱动器没有被装载，SCM在PnP_DeviceList函数返回的驱动器列表中查询它的名字。 PnP_DeviceList支持被包括在当前系统通用硬件配置中的驱动器。 SvcCtrlMain记录了没有启动的驱动器的名字，并作为ScFailedDrivers列表中当前配置文件的一部分。

在启动自启服务之前， SCM做了一些额外的事。它创建了叫做管道的远程过程调用Pipe\Ntsvcs，并且开始一个线程来监听来自 SCP的消息，然后通知它的初始化结束事件SvcCtrlEvent_A3752DX。SCM通过由RegisterServiceProcess注册一个控制台应用程序关闭事件处理并向Win32子系统进程注册来为系统关闭做准备。

1. 服务启动

SvcCtrlMain 调用SCM的ScAutoStartServices来启动所有有自启动初始值的服务，也装入自启动的设备驱动程序。它按正确的顺序启动服务进程，算法是阶段性执行的，每一个阶段由此符合每一组，并且由组次序定义好的顺序执行，它保存在“ HKLM\SYSTEM\CurrentControlSet\

Control\ServiceGroupOrder\List”注册表主键值中。因此，除考虑对属于其他组的服务有妨碍的情况下对服务启动的调整之外，把一个服务添加到一组是没有什么影响的。

当一个阶段开始时，ScAutoStartServices为了启动而把所有属于此阶段的组的服务条目做标记。然后它循环所有标记过的服务，检测是否每一个服务都能启动。检测内容包括确定此服务的启动是否依赖于其他组的服务，这由服务的注册主键值中的 DependOnGroup是否存在决定。如果依赖关系存在，服务所依赖的组必须已经完成初始化，并且那组中至少有一个服务已经成功地启动了。如果在组启动顺序中一个服务所依赖的组比服务所在的组靠后， SCM标记一个“ circular dependency ”错误给服务。如果 ScAutoStartService考虑一个Win32服务而不是设备驱动器，它下一步会检测此服务依赖于一项还是多项服务，这些服务是否都已经启动。服务依赖关系在服务的注册主键值的 DependOnService键值中被指定。如果一个服务所依赖的其他服务属于在其后启动的组，SCM同样产生一个“ circular dependency ”错误并且不启动这个服务。如果服务依赖于本组的任何未启动服务，则此服务的执行将会被跳过。

当服务的依赖关系通过检测后，在启动服务之前，ScAutoStartServices为此服务是否为当前通用引导配置的一部分做最后的检测。若系统以安全模式启动， SCM确保服务在正确的安全引导注册主键值中通过名字或组被鉴别出来。在“ HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot ”中，有两种安全引导的键值：最小配置和网络支持配置，启用哪种配置则由用户所选的安全模式类别来决定。如果用户在专用启动菜单中选取普通安全模式或命令行安全模式， SCM将启用最小配置；如果用户选取网络安全模式， SCM将启用网络支持配置。在安全启动的键值中 Option的值不仅指明系统以安全模式启动，还指明用户选择的安全类别。

一旦SCM决定启动一项服务，它调用 ScStartService，ScStartService对服务和设备驱动器的启动采取不同的步骤。当 ScStartService启动一个Win32服务时，它首先读取服务的注册主键值中的ImagePath来确定运行服务进程的文件名。然后检查服务类型值，其值是否为 SERVICE_WIN32_SHARE_PROCESS（0x20），SCM确保插入的服务进程与已启动的服务用相同的帐号登录。服务的注册键值中的对象名保存了服务运行的帐号信息。没有对象名或对象名为本地系统的服务运行于本地系统帐号环境下。

在SCM内部的image database数据库中，SCM通过检查服务的ImagePath是否有一个条目来核实服务进程是否在另外的帐号下运行。如果 image database不存在ImagePath值的条目，SCM将会创建一个，并且保存服务所用的登录帐号和从服务的 ImagePath 得来的数据。SCM 要求服务含有一个ImagePath值，如果没有，SCM报告一个它不能找到服务路径和不能启动服务的错误。如果SCM定位了一个存在的image database条目和匹配的ImagePath数据，SCM确保服务启动的用户帐号信息同保存在数据库里的信息一致 —— 一个进程仅能以一个帐号登录，所以在同一个进程中，一个服务指定的帐号同其他已启动的服务在此进程中指定的帐号不同时， SCM将会报告错误。

如果服务指定了配置， SCM调用ScLogonAndStartImage登录服务并启动服务进程。 SCM调用Lsass函数来登录不在系统帐号下运行的服务。当 SCM调用LsaLogonUser，它指派服务的登录类型，Lsass在注册表子键 Secrets形如_SC_<service name>的条目下寻找密码。当 SCP 配置服务的登录信息时，SCM指导Lsass隐式地保存登录密码。当成功登录时， LsaLogonUser返回一个调

用者访问的句柄。Windows 2000使用访问标记来代表用户的安全权限，SCM稍后把实现服务的进程同访问标记联系起来。

成功登录后，如果没有装载帐号信息，则SCM通过调用UserEnv DLL的LoadUserProfile 函数来装载帐号的信息。“HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<user profile key>\ProfileImagePath”值包括了LoadUserProfile函数装载到注册表的registry hive磁盘地址，为在hive中的HKEY_CURRENT_USER创建信息。

交互式的服务必须打开WinSta0，但是在ScLogonAndStartImage允许交互式服务访问WinSta0之前，它检查是否已设定“HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices”值。管理员设定这个值来阻止交互式服务在控制台上显示窗口。这个选项满足了无人服务器环境的需要，此环境下交互式服务中并没有用户可以反馈。

下一步，如果服务进程没有被启动，ScLogonAndStartImage会继续启动服务进程。SCM用CreateProcessAsUser Win32函数以挂起状态启动进程，然后创建一个命名管道来同服务进程通信，并把管道命名为“\Pipe\Net\NetControlPipeX”，每一次SCM创建一个新管道都对X加一。SCM通过ResumeThread函数来恢复服务进程和等待服务连接到 SCM管道。如果“HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout”键值存在，它决定了 SCM等待服务调用 StartServiceCtrlDispatcher的超时时间。如果ServicesPipeTimeout键值不存在，SCM使用30秒作为缺省时间，SCM在所有的服务通信中使用相同的超时时限。

当服务通过管道同SCM连接上时，SCM发送一个启动命令给服务。在超时时限内，如果服务没有对启动命令做出明确响应，SCM将放弃并开始启动下一个服务。在超时时限内，服务对启动命令没有响应时，SCM并不像服务没有调用StartServiceCtrlDispatcher那样终止进程，而是在系统事件日志中记录一个错误，说明此服务没有在时限内成功启动。

如果SCM 调用ScStartService启动的服务的注册表主键值类型为 SERVICE_KERNEL_DRIVER或SERVICE_FILE_SYSTEM_DRIVER，此服务为设备驱动器，所以ScStartService调用ScLoadDeviceDriver来装载驱动器。ScLoadDeviceDriver为SCM进程启用安全特权，然后调用内核服务NtLoadDriver，在ImagePath中传递驱动器注册主键表值的数据。与服务不同，驱动器不需要指定一个ImagePath值，如果此值不存在，SCM通过添加驱动器名到“\Winnt\System32\Drivers\”字符串来建立一个image path。

ScAutoStartServices持续循环组内的服务直到所有的服务不是启动就是发生了依赖关系错误为止。这种循环是SCM依照服务的DependOnService依赖关系自动排列服务的次序的方法。SCM将会先循环其他服务所依赖的服务，跳过依赖于其他服务的服务直到子序列开始循环。注意 SCM忽略Win32服务的标志符，在“HKLM\SYSTEM\CurrentControlSet\Services”键值里可以看到这个标志符；而I/O管理器则以此标识符来排列有关引导和系统启动的设备驱动器的启动。

一旦SCM结束了在ServiceGroupOrder\List中所有列出的服务组的所有阶段，接着它执行属于列表中没有列出的组的服务，最后一步执行不属于任何组的服务。

2. 启动错误

如果驱动器或服务对SCM启动命令回应一个错误，在服务注册键值中的 ErrorControl决定了

SCM如何处理。如果其值为 SERVICE_ERROR_IGNORE（0）或没有被指定，SCM只是简单地忽略此错误并继续服务的启动。如果其值为 SERVICE_ERROR_NORMAL（1），SCM写入一个事件到系统事件日志，描述为“ <服务名>服务由于下面的原因启动失败：”，记录在事件日志记录中，SCM包括了服务作为启动失败原因反馈给 SCM的文本Win32错误类型代号。

如果含有SERVICE_ERROR_SEVERE（2）或SERVICE_ERROR_CRITICAL（3）ErrorCtrol值的服务报告一个启动错误，SCM写入一个记录到事件日志并且调用内部函数 ScRevertToLastKnownGood。此函数切换系统注册配置到一种被称为last known good的版本，就是系统上一次成功启动所用的版本。然后 SCM用NtShutdownSystem系统服务重启系统，此服务在可执行程序中已经实现。如果系统已经使用last known good配置启动，那么系统只是简单地重启。

3. 接受最近一次成功引导的配置

除了启动服务，系统还管理SCM决定系统注册配置HKLM\SYSTEM\CurrentControlSet何时被存为last known good control控制设置。CurrentControlSet把服务主键作为一个子键包含，所以它包括了SCM数据库中的注册表，还包括了保存许多核心态和用户态子系统的配置信息的控制主键。缺省情况下，一次成功的启动包括所有自启服务的成功启动和一个用户的登录。如果系统在启动时因为设备驱动器崩溃或带有 SERVICE_ERROR_SEVERE 或 SERVICE_ERROR_ CRITICAL ErrorControl值的自启服务报告一个启动错误而导致异常终止，则此次为一次失败的启动。

SCM显然能确信它成功地完成了一次自启服务的启动。但是对于 Winlogon（\Winnt\System32\Winlogon.exe），则必须把一次成功的登录通知它。当用户登录时， Winlogon调用 NotifyBootConfigStatus函数，此函数发送一个信息给 SCM。在所有自启服务成功启动和收到来自NotifyBootConfigStatus函数的登录信息（任何最后到来的一个）后， SCM调用NtInitialize Registry保存当前的启动注册配置信息。

第三方软件开发者可以用他们自己的定义来取代 Winlogon的定义。举例来说，运行Microsoft SQL Server的系统可能直到SQL服务器能接受和处理事务时才确定启动为成功。开发者通过写一个启动验证程序并且把程序安装到保存在注册表主键值“ HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram ”中的磁盘地址来加入他们对系统成功启动的定义。另外，启动验证程序的安装必须通过设定“ HKLM\SOFTWARE\Microsoft\ Windows NT\CurrentVersion\Winlogon\ReportBootOk ”为0来禁用Winlogon对NotifyBootConfigStatus的调用。当启动验证程序被安装时， SCM在启动完所有自启服务后才运行它，并且在保存 last known good控制配置信息前等待程序调用 NotifyBootConfigStatus。

Windows 2000保留了多份CurrentControlSet的拷贝，而CurrentControlSet实际上只是一个指向一份拷贝的象征性的链接。控制配置在“ HKLM\SYSTEM\ControlSetnnn ”表单里都有名字，其中nnn是001、002这类的数字。HKLM\SYSTEM\Select键值包括了鉴别每一个控制配置的值。比方说，如果CurrentControlSet指向ControlSet001，那么当前选定的值为1。选定的LastKnownGood值包括了上一次成功启动时已知的正确的控制配置。在你的系统上另一个值可能是失败的，指向上一次没有成功启动并且使用last known good 控制设置启动的控制设置。

NtInitializeRegistry从last known good 控制设置中读取内容并且把它同 CurrentControlSet key

树同步起来。如果这是系统的第一次成功启动，last known good并不会存在，于是系统会为它创建一个新的控制设置，如果它存在，系统只是简单地把它同 CurrentControlSet不同的部分做更新。

Last known good在CurrentControlSet有变更的时候很有帮助，就像在 HKLM\SYSTEM\Control里改变了系统的performance-tuning值或者增加了一个服务或设备驱动器而导致后来的系统启动失败的情况。用户可以在系统引导进程的早期按下 F8来引出一个菜单，使他们可以使用last known good控制设置，把系统配置重置为上一次成功启动时的正确配置。

4. 服务失败

一个服务可以在它的注册表主键值里含有可选的失败处理动作和失败处理命令，在系统启动时，SCM记录这个注册表主键值。SCM向系统注册，所以系统在一个服务进程存在时通知 SCM。

服务可以为SCM配置的失败处理动作包括重启这个服务、运行一个程序或者重启计算机。此外，服务可以指定服务进程第一次失败、第二次失败和接下来的失败时的失败处理动作，并且如果服务被要求重启，还可以指定 SCM在服务重启前等待的延时。IIS服务管理器的失败处理动作将导致SCM运行IISReset应用程序，清除所有工作并且重启服务。你可以很方便地在 Services MMC snap-in里的服务属性对话框里的Recovery标签下管理上述的恢复操作。

5. 服务关闭

当被Winlogon调用时，Win32ExitWindowsEx函数发送一条消息给Win32子系统进程Csrss，调用Csrss的关闭例程。Csrss遍历所有活跃进程并且通知它们系统正在关闭。在通报下一个进程前，Csrss等待除了 SCM以外的每一个系统进程退出，等待的秒数在“ HKU\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout ”里指定（缺省值为20秒）。当Csrss遇到SCM进程时，它也通报SCM系统正在关闭，但是等待专为 SCM指定的超时。在系统初始化时， SCM通过Register ServicesProcess函数向Csrss注册它的进程ID，Csrss便通过使用SCM的进程ID（Csrss保存这个ID）辨认SCM。SCM的超时之所以与其他进程不同是因为Csrss知道SCM还要与需要在关闭时清除数据的服务进行通信，所以管理员可能仅需要调整 SCM的超时。SCM的超时值在“ HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout ”中，并且缺省状态设为20秒。

SCM的关闭处理程序有责任发送关闭通知给所有那些在 SCM初始化时申请需要关闭通知的服务。SCM的ScShutdownAllServices遍历SCM服务数据库寻找那些请求有关闭通知的服务并发送关闭通知给每一个这样的服务，并为每一个这样的服务记录服务的等待延时值，这个值在服务向SCM注册时就指定了。SCM明确它所收到的最大的等待延时。发送关闭通知后， SCM等待它所通知的服务退出或者一直等待到超过最大的等待延时为止。

如果服务在超过等待延时后仍没有退出，SCM就测定一个或多个它正在等待退出的服务是否已经发送了一个消息给 SCM，告诉SCM此服务在它的关闭进程上取得进展。如果至少一个服务有进展，SCM就在等待延时的范围内再等待一次。SCM持续执行它的等待循环直到所有的服务都已经退出或者它所等待的服务都没有在等待延时内发送给它取得进展的信息为止。

当SCM忙于通知服务关闭并且等待它们退出的时候，Csrss等待SCM退出。如果Csrss等待超时而SCM没有退出，Csrss简单转移，继续它的关闭进程。因而，在系统关闭时，那些在规定时限内没有成功关闭的服务只是简单地同 SCM一起执行。不幸的是，管理员无法知道他们是否应该

为那些在系统关闭前无法完全关闭的服务增加 WaitToKillServiceTimeout值。

6. 共享服务进程

在Windows 2000/XP自带的内部服务中，有些服务运行于它们自己的进程空间，而有些服务运行于与其他服务共享的进程空间。举例来说， SCM进程上运行了事件日志服务、文件服务器服务（LanmanServer）和LAN Manager name resolution服务。

有一种称为Service Host的进程包含了多个服务。多个 SvcHost的实例能被运行于不同的进程空间。运行在SvcHost进程的服务包括Telephony（TapiSrv）、Remote Procedure Call（RpcSs）和Remote Access Connection Manager（RasMan）。Windows 2000以DLLs的形式执行在SvcHost里运行的服务并且在服务的注册表主键值中包括了一个 ImagePath的定义。服务的注册表主键值也必须在子键值中包括名为 ServiceDll的键值，指向服务的DLL文件。

所有共享一个SvcHost进程的服务指定一个同样的变量，所以它们在 SCM的数据库中只有一个条目。在服务启动时，如果 SCM遇到第一个含有特殊参数的 SvcHost ImagePath的服务，它创建一个新的数据库条目并且使用这个参数执行 SvcHost 进程。新的SvcHost进程采用这个参数并且在HKLM\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost下寻找与它同名的值，SvcHost读取这个值的内容，将它解释为一串服务名的列表，并且在 SvcHost向SCM注册时通知SCM它正在提供这些Service Host服务。

当SCM在服务启动时遇到SvcHost服务，并且这个服务在启动时带有与 image数据库匹配条目的ImagePath时，它并不启动另一个进程而只是为服务发送一个启动命令给带有那个 Imagepath值的已经启动的 SvcHost。于是，这个SvcHost进程在服务注册键值中读出 ServiceDll参数并且装载此DLL到进程来启动服务。

7. 服务控制程序

服务控制程序都是标准 Win32应用程序，使用 CreateService、OpenService、StartService、ControlService、QueryServiceStatus和DeleteService SCM 函数。为了使用这些SCM函数，SCP必须首先通过调用 OpenSCMManager函数打开一个同SCM通信的通道。在调用的时候， SCP必须指定它需要执行的动作类型。比方说，如果 SCP只是简单地列举SCM数据库里的当前服务，它在调用OpenSCMManager的时候就请求列举服务。在初始化的时候， SCM创建了一个内部的对象代表SCM数据库，并且用Windows 2000/XP的安全函数通过安全描述符来保护这个对象，安全描述符里指定了帐号的安全权限。

SCM执行于服务自身的安全环境，当 SCP通过CreateService函数创建一个服务时，它指定一个安全描述符，此描述符是同 SCM服务数据库中服务的条目有内在关联的。 SCM保存在服务注册主键值中的安全描述符为安全值，在初始化时期，当它浏览服务注册表主键值的时候读取这个值，使得重新启动期间安全设置一直持续。同样的， SCP必须在调用OpenSCManager时指定它将要对SCM数据库访问的访问类型， SCP必须在调用 OpenService中告诉SCM它将对服务进行的访问类型。SCP请求的访问类型包括查询服务的状态，配置、停止和启动一个服务。

大家最熟悉的 SCP可能是Windows 2000里的MMC snap-in服务，它在“ \Winnt\System32\Filemgr.dll ”中。SCP有时在SCM执行之上把服务策略分层。一个很好的例子就是在手动启动一个

服务时MMC snap-in执行的超时服务。Snap-in给出一个进度条来显示服务启动的进度。服务通过设定它们的配置状态来间接地同SCP交互，它们的配置状态反映了服务对SCM类似于启动命令的命令的响应进展。SCP通过QueryServiceStatus函数查询配置状态。它们可以告诉服务：在服务处于挂起状态时动态地更新这些状态，并且SCM可以采取适当的动作通知用户服务当前的工作情况。

2.7 Windows 2000/XP的管理机制

Windows 2000/XP通常利用事件管理器报告错误信息和诊断信息。而事件查看工具则能使管理员详细掌握本地机和远程机器上的事件动态。与之相似的是，性能评估机制使得应用程序和操作系统组件能将性能表现的统计数据传送给性能监视器（Performance Monitor）。

Windows NT事件监视器和性能监视器是有一定局限性的。举例来说，程序接口千差万别，这种差异无疑增加了应用程序运用事件和性能管理器收集数据的复杂程度。而性能评估机制体现不出它们之间的细微差别，尤其是通过网络的时候，因为这些性能统计数据往往经过系统的修改，所以不能直接反映你所关心的对象的情况。Windows NT 4监视系统中最大的弱点就是它们几乎不具备可扩展性，而且日志和性能数据的收集程序也没有给用户提供交互中所必需的编程接口。应用程序必须以预先定义好的格式提供数据，而程序没有办法收到与性能评测相关的事件的提示消息，而且事件管理器所提供的事件的消息也无法让程序知道这些事件的确切类型和来源。最后，这些性能数据评估程序实际上无法通过事件管理器或是应用程序接口与事件或性能数据的提供程序进行通信。

为了消除这些局限，同时也为其他的数据源提供管理和分析的工具，Windows 2000/XP引入了一种新的机制：Windows Management Instrumentation（WMI）。WMI是基于网络的企业管理（Web-Based Enterprise Management，WBEM）标准的实现，这个标准是由分布式管理工作小组（Distributed Management Task Force，DMTF）所制订的。WBEM标准中包括了事务数据收集和管理的设计，这种设计具有很强的伸缩性和扩展性，这两种特性正是管理可能由多种不同组件组成的本地机和远程机器所必需的。

2.7.1 WMI的体系结构

WMI包含4个主要的组件（如图2-16所示）：管理程序、WMI基础设施、数据生产者和被管理的对象集合。管理程序处理和显示应用程序从被管理的对象获取的数据。一个简单的例子是性能评测工具使用WMI标准而不是API来收集性能相关的统计数据，而更复杂的例子是，一个基于WMI标准的企业级的管理工具可以使管理员方便地管理每台软件与硬件的配置。

开发者往往将管理应用程序的目标定位于从指定的对象获取数据。一个对象可能代表一个组件，如一个网络适配器，也可能代表一系列的组件，如一台计算机。数据生产者需要确定和输出管理应用程序所关心的对象在系统中代表什么。比如，一个网络适配器的厂商可能会将一些适配器的属性加入WMI所支持的适配器中，这样应用程序可以通过WMI根据自己的需要查询和设置适配器的状态和行为。在一些应用（如设备驱动程序）中，微软公司提供了一个带有编程接口的数据生产者，使开发者用最少的代码设计出拥有自定义的管理对象的数据生产者。

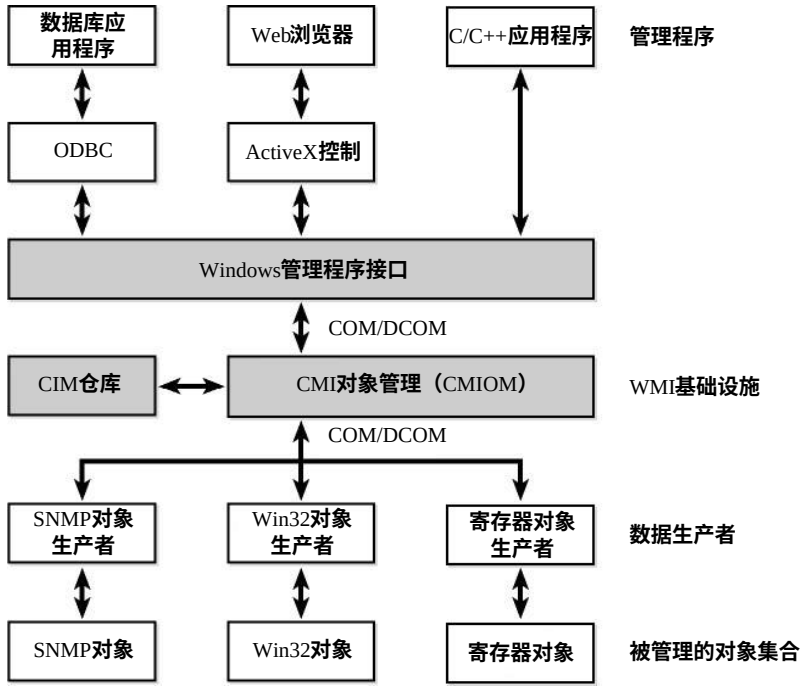


图2-16 WMI的体系结构

WMI基础设施是通用信息模型对象管理器（CIMOM）的核心，它是联系管理应用程序和数据生产者的桥梁。WMI将数据以一个名为CIMOM Object Repository的数据库的形式存储在磁盘上。作为这个基础设施的一部分，WMI同时也支持一些API，通过它们，管理应用程序就可以访问对象以及数据生产者所提供的数据和类的定义。

Win32程序使用底层的WMI-COM接口直接与WMI通信。其他接口都是在这层接口之上的，包括一个为Microsoft Access数据库应用程序设计的开放数据库连接（ODBC）适配器。数据库程序开发者用WMI ODBC适配器将数据源嵌入开发着的数据库中，这样就可以通过对基于WMI的数据的查询生成数据报表。WMI ActiveX控件支持另一个层次的编程接口，网络程序开发者可以用这些控件构造出访问WMI数据的交互界面。微软公司还提供还有一种WMI脚本API，用于那些基于脚本的应用程序和Microsoft Visual Basic程序；实际上微软公司所有的编程语言都支持这种脚本API。

同样在与数据生产者交互的过程中，WMI COM接口组成了WMI最基本的API。但不同的是，管理应用程序是COM的客户端，而数据生产者则是COM或DCOM的服务器端（也就是说，WMI中的COM对象是由数据生产者实现的）。WMI数据生产者的可能表现形式有：WMI管理进程加载的动态链接库、独立的Win32应用程序以及独立的Win32服务进程。微软公司提供了一套内建的数据生产者，它们提供了很常用的数据源，如性能 API、注册表、事件管理器、活动目录、SNMP和基于WDM（Windows Driver Model）的设备驱动程序。同时，WMI的开发包使得我们可以开发第三方的WMI数据生产者。

2.7.2 数据生产者

WBEM的核心是基于DMTF 设计的CIM。CIM规定了管理系统如何表示从计算机传递到应用程序或设备的信息。它还从系统级的层次上确定了管理系统的每一部分的描述，大至整个系统，小到一个应用程序或一个设备。这样，开发者就可以用 CIM来定义自己所写的管理程序中所需的组件，通常用管理对象格式（Managed Object Format，MOF）语言来描述。

除了为对象定义类之外，数据生产者还必须为 WMI提供这些对象的编程接口。 WMI根据数据生产者提供接口的不同将它们分成若干类。注意，一个数据生产者可能同时具有多个特征，因此它可以同时是类和事件的生产者。为了说明这一点，看一个同时具有若干个特征的数据生产者——事件日志数据生产者，它定义了多个对象，包括一个事件日志计算机、一个日志记录和一个日志文件。因为它在定义这些对象时用到了类定义，所以它是一个类类型的数据生产者，必须将这些类的定义传给 WMI。同时它又是一个实例类型的数据生产者，因为它为自己所属的好几个类定义了多个实例，其中一个类是事件日志文件类，这个数据生产者用这个类为每个系统日志（系统事件日志、应用程序事件日志和安全事件日志）都创建了一个实例。

事件日志数据生产者定义了实例数据，这样管理应用程序就可以列举它们。为了让应用程序用WMI存储和备份这些事件日志文件，数据生产者为日志文件对象提供了实现这些功能的方法。这样做使得这个数据生产者又是一个方法类型的生产者。最后，可以要求在每一个新的记录写入日志文件时，数据生产者都要给应用程序发送消息，这样这个数据生产者实际又有事件类型生产者的属性，因为它用到了 WMI的事件响应机制。

2.7.3 通用信息模型和管理对象格式语言

CIM紧随着面向对象语言（如 C++、Java）的步伐，这些语言以类的表示形式设计模板。类的运用使开发人员得以使用强大的继承和合成建模技术。子类可以继承父类的特性，并且可以增加它们自己的特性和重载它们从父类继承来的特性。类还可以组合，开发者可以创建一个包含其他类的类。

DMTF提供了多样的类作为 WBEM标准的一部分。这些类是 CIM的基本语言，代表了适用于所有管理领域的对象，是 CIM核心模块的一部分。一个核心类的例子是 CIM_ManagedSystem Element，这个类包含了很少的基本属性，基本属性标志了物理部件（如硬件设备），还标志了逻辑部件（如进程、文件）。这些属性包含了标题、描述、安装日期和状态。因而， CIM_ LogicalElement 和CIM_PhysicalElement类继承了CIM_ManagedSystemElement的属性，这两个类也是CIM核心模块的一部分。 WBEM标准称这些类为抽象类，因为它们只为其他类继承而存在（也就是说，没有一个抽象类的对象实例存在）。我们由此可以把抽象类看作是定义了其他类使用的属性的模板。

第二种类代表了那些对于管理领域特殊但又并非特殊实现的对象。这些类组成了通用模块并被认为是核心模块的扩展。一个通用模块类的例子是 CIM_FileSystem类，它继承了类 CIM_ LogicalElement的属性。因为实际上每一个操作系统，包括 Windows 2000/XP、Linux和其他版本

的UNIX，都依靠文件系统存储，所以CIM_FileSystem类是通用模块中一个合适的要素。

最后一种类给普通模型附加了一些面向具体技术的成分。Windows 2000定义了一大组这种类来代表Win32环境特殊的对象。由于所有的操作系统在文件中保存数据，因此CIM通用模块包括了CIM_LogicalFile类。CIM_DataFile类继承CIM_LogicalFile类，而且Win32增加了Win32_PageFile和Win32_ShortcutFile文件类给这些Win32文件类型。

事件日志供应者使用了扩展继承属性。时间日志文件是一个数据文件，含有增加的特殊事件日志属性，比如一个日志文件名（LogfileName）和一个文件所包含记录的统计数字（NumberOfRecords）。类浏览器显示的类树揭示了Win32_NTEventlogFile是基于几个继承层次的，其中，CIM_DataFile从CIM_LogicalFile中派生得来，而CIM_LogicalFile从CIM_LogicalElement派生得来，CIM_LogicalElement又是从CIM_ManagedSystemElement派生得来的。

在MOF中构造一个类后，WMI开发者可以通过几种方法提供WMI对类定义的支持。WDM开发者把MOF文件编译成一个二进制MOF文件（BMF）——一个比MOF文件更简洁的表示法——并且把这个BMF文件交给WDM基础设施。另一种方法是编译MOF文件并且使用WMI COM APIs把类定义送给WMI基础设施。最后，生产者可以使用MOF编译工具（Mofcomp.exe）来直接交给WMI基础设施一个类编译好的代表。

2.7.4 WMI名字空间

类定义了对象的属性，而在系统中对象是类的一个实例。WMI使用一个名字空间来组织对象，名字空间由几个子名字空间组成，WMI层次地排列这些子空间。一个管理应用程序在名字空间里访问类之前必须先连接上这个名字空间。

WMI命名名字空间的根目录为root。所有WMI有四个在root下预先定义好的名字空间：CIMV2、Default、Security和WMI。这些名字空间中的一些名字空间包括了其他的名字空间。比如，CIMV2包括了Applications和ms_409 namespaces作为它的子名字空间。生产者有时也定义它们自己的名字空间。

与文件系统的名字空间不同，文件系统是由目录和文件的分级结构构成，而WMI名字空间只有一个层次。WMI使用对象的属性而不是像文件系统那样使用名字，它定义这些属性鉴别对象。管理应用程序用关键字名指定类名以定位一个名字空间中的特殊对象。因此，每一个类的实例都必须通过关键字的值唯一确定。比如，事件日志生产者使用Win32_NTLogEvent类来描述事件日志中的记录。这个类有两个关键字：Logfile，一个字符串；RecordNumber，一个无符号整数。向WMI查询事件日志记录实例的管理程序从标志此记录的那对关键字得到它们。应用程序使用下面示例中对象路径名的语法来访问记录。名字开始的部分（\\PICKLES）标志了对象所在的计算机，第二部分（\CIMV2）是对象所处的名字空间。类名紧跟在冒号后面，关键字名和它相关的值跟在其后，关键字值用逗号隔开。

2.7.5 类联合

许多对象类别都通过一些途径同其他对象类别相关联。WMI使供应者构造一个类联合来代表

两个类之间的逻辑连接。类联合把一个类同另一个类关联起来，所以它只有两个属性：一个类名和Ref modifier。给定一个对象，管理应用程序可以查询被关联的对象。通过这种途径，生产者定义了一个对象层次。

典型的Win32系统组件把它们的对象放在CIMV2名字空间里。对象浏览器首先定位Win32_ComputerSystem对象实例DSOLOMON，它代表了计算机，然后对象浏览器获得与此对象关联的其他对象并在DSOLOMON下展示出来。对象浏览器用户界面用一个双向箭头文件夹图标来展示关联对象，被关联的对象在这个目录下展示。

2.7.6 WMI对象浏览器

在对象浏览器里，你可以发现事件日志生产者关联类Win32_NTLogEventComputer在DSOLOMON的下面并且有众多的Win32_NTLogEvent实例存在。在对象浏览器里选择一个Win32_NTLogEvent类的实例将会在右边的面板的属性标签中展示类的属性。微软公司计划使用对象浏览器来帮助WMI开发者检查他们的对象，但是一个管理应用程序可以完成同样的操作并且可以以更易理解的方式展示或收集类的属性信息。

2.7.7 WMI执行

WMI基础设施主要在“\Winnt\System32\Wbem\Winmgmt.exe”文件中实现。这个文件以Win32服务方式运行，在一个管理应用程序或WMI生产者试图第一次访问WMI APIs时启动它。大多数WMI组件缺省驻留于“\Winnt\System32 and \Winnt\System32\Wbem”，包括Win32 MOF文件、内建的生产者动态链接库和管理应用程序WMI动态链接库。

“\Winnt\System32\Wbem”下的目录保存库、记录文件和第三方MOF文件。WMI执行库——称为CIMOM库——“\Winnt\System32\Wbem\Repository\Cim.rep”文件。Winmgmt认可众多同库关联的的注册表设定（包括不同的内在性能参数，比方说CIMOM备份的地址和时间间隔），库的“HKLM\SOFTWARE\Microsoft\WBEM\CIMOM”注册表主键值保存了这些注册设定。

设备驱动器用特殊的接口来提供数据和接受从WMI来的命令——调用WMI系统控制命令。因为这些接口是跨平台的，所以它们归属于“\root\WMI”名字空间下。

2.7.8 WMI安全

WMI在名字空间层上执行安全措施。如果一个管理应用程序成功地连接上了名字空间，应用程序可以查看和访问所有处于那个名字空间的对象的属性。管理员可以使用WMI控制应用程序来控制何种用户可以访问一个名字空间。

习题

2.1 操作系统作为一个软件，具有什么样的特点？你能否举出相应的例子说明这些特点？

2.2 在设计操作系统的时候要考虑哪些因素的影响，为什么？

2.3 一个优秀的操作系统设计应该具备什么样的特点，如何理解这些设计目标？你认为

Windows 2000/XP是否具备这些特点，请举例说明。

2.4 操作系统设计的过程包括哪些阶段，在每个阶段都要考虑什么问题。

2.5 有哪些常见的操作系统的体系结构，它们各自都有什么特点？ Windows 2000/XP采用了什么样的体系结构，有什么优缺点？

2.6 下列哪些指令应在核心态运行：

(1) 关中断

(2) 读实时时钟

(3) 设置实时时钟

(4) 改变内存页映射

(5) 设置处理器运行状态

2.7 你认为下列操作系统采用了哪种体系结构或者具有哪些体系结构的特点：

(1) MSDOS

(2) Linux

(3) Windows 95

(4) Windows NT

(5) VM/370

(6) BSD UNIX

(7) Mach

(8) Minix

2.8 你认为各种操作系统的体系结构分别适合什么样的应用场合？

2.9 客户/服务器的操作系统体系结构在分布式系统中使用非常广泛，你认为它能否用于单机环境？Windows 2000/XP具有很多这种体系结构的特征，那么在这些方面 Windows 2000/XP对原有的模型做了哪些调整，你认为这些调整是否有用？

2.10 在客户/服务器的体系结构中，进程服务将决定谁被调度。系统的调度机制将在操作系统的核心内完成，而调度的策略则在进程服务中完成。

(1) 进程描述表应存放在什么地方？是核心还是进程服务的私有地址空间？

(2) 我们知道调度时的切换动作一定是在核心内完成的，那么核心是如何知道切换到哪里去呢？

2.11 Windows 2000/XP包括哪些主要的组成部分，各自的作用是什么？

2.12 Windows 2000/XP是通过什么方式支持各种不同的硬件平台实现可移植的目标？

2.13 Windows 2000/XP的设备驱动程序是做什么的，有哪些类型的设备驱动程序？

2.14 环境子系统的主要作用是什么，它是怎样在 Windows 2000/XP中发挥它的作用的？

2.15 通过进程观察器，你能看见哪些重要的系统进程，它们各自都有什么作用？试着用进程观察器终止一个重要的系统进程（例如 WINLOGON），看一看会有什么结果。在DDK中有一个实用程序kill，它可以绕过系统的检查而杀掉这些进程，试着结束 SMSS或者CSRSS，看一看有什么结果。

2.16　Windows 2000/XP是怎样看待中断和异常的，它提供了什么样的机制去处理它们？

2.17　DPC、APC有什么区别？

2.18　LPC对RPC做了什么样的优化？

2.19　Windows 2000/XP的核心资源管理采用了对象的方式，这有什么好处？在 Windows 2000/XP中有哪些类型的对象，都有什么作用？对象管理器是怎样把这些对象组织起来的？

2.20　Windows 2000/XP提供了哪些同步互斥的机制？（包括单处理器的情况和对称多处理器的情况）。

2.21　用Regedit.exe实用程序查看Windows 2000/XP的注册表，了解注册表的构成。

2.22　什么是Windows 2000/XP服务，Windows 2000/XP是如何操纵服务的？

2.23　请简述WMI的作用和构成情况。

2.24　查阅一些有关 Linux的资料，看看 Linux的体系结构是什么样子的。为了提高性能，Linux在体系结构上做了那些权衡折中。

2.25　查阅资料，比较Windows 2000/XP、Windows NT 4、Windows 9x和Windows CE在体系结构上的共同点和差异。想一想为什么会有这样的差异。

2.26　使用资源工具包和 DDK的工具查看系统状况：① 子系统的启动；② 造成一次系统崩溃，并用调试工具查看故障转储文件（最小转储 64K）；③ 窥视核心的非文档化接口；④ 抓一个系统快照，查看当前系统的内存、页表、进程、对象等情况；⑤ 使用性能监视器查看各种不同负载情况下的性能情况；⑥ 研究对象管理器，查看系统对象及其属性；⑦ 查看系统服务活动；⑧ 查看系统的启动日志以及注册表的相关部分，了解Windows 2000/XP的启动机制。

2.27　编写一个简单程序（helloworld），使用调试器在汇编级别跟踪它的运行情况。

2.28　编写一个最简单的核心态设备驱动程序，安装到你的 Windows 2000/XP中，并编写一个用户程序请求该设备驱动程序的功能。试着使用调试工具进行跟踪。调试器可以用Windows 2000/XP的kernel debugger做远程调试或者使用 Soft ICE。有关的编程指南可以查看Windows 2000/XP的DDK文档和《Windows Device Drivers》一书。

2.29　Windows 2000/XP的服务是要编程实现的，查阅 MSDN的相应文章了解这一过程。