

一些Win32调用使用路径名,然而相关的NT内核调用使用句柄。所以这些封装流程包括打开文件,调用NT内核,最后关闭句柄。封装流程同时包括把Win32 API从ANSI编码变成Unicode编码。在图11-10的Win32函数里使用字符串作参数的实际上是两套API,例如参数CreateProcessW和CreateProcessA。当这些参数要传递到下一个API时,这些字符串必须翻译成Unicode编码,因为NT内核调用只认识Unicode。

因为已经存在的Win32接口很少随着操作系统的改变而改变,所以从理论上说能在前一个版本系统上运行的程序也能正常地在新版本的系统上运行。可在实际情况中,依然经常存在新系统的兼容性问题。Windows太复杂了以至于有些表面上不合逻辑的改动会导致应用程序运行失败。应用程序本身也有问题,例如,它们也经常做细致的操作系统版本检查或者本身就有潜在的问题只不过是新系统上暴露出来了。然而,微软依旧尽力在每个版本上测试不同的兼容性问题,并且力图提供特定的解决办法。

Windows支持两种特殊环境,一种叫WOW。WOW32通过映射16位字符串到32位,来在32位x86系统用16位Windows 3.x应用程序。同样,WOW64允许32位的程序在x64架构的系统上运行。

Windows API体系不同于UNIX体系。对于后者来说,操作系统函数很简单,只有很少的参数以及很少的方法来执行同样的操作,从而可以有很多途径来完成同样的操作。Win32提供了非常广泛的接口和参数,常常能通过三四种方法来做同样的事情,同时把低级别和高级别的函数混和到一起,例如CreateFile和CopyFile。

这意味着Win32提供了一组非常多的接口,但是这也增加了复杂度,原因是在同一个API中糟糕的系统分层以及高低级别函数的混和。为了学习操作系统,我们仅仅关注那些低级别的函数封装了相关的NT内核的API的Win32 API。

Win32 有创建和管理进程和线程的调用。Win32也有许多进程内部通信的调用,例如创建、销毁、互斥、信号、通信接口和其他IPC实体。

虽然大量的内存管理系统对程序员来说是看不见的,但是一个重要的特征是可见的:即一个进程把文件映射到虚拟内存的一块区域上。这样允许线程可以使用指针来读写部分文件,而不必执行在硬盘和内存之间具体的读写数据操作。通过内存映射,内存系统可以根据需求来执行I/O操作(要求分页)。

Windows 处理内存映射文件使用三种完全不同的手段。第一种,它提供允许进程管理它们自己虚拟空间的接口,包括预留地址范围为以后用。第二种,Win32支持一种称作文件映射的抽象,这用来代替可定位的实体,如文件(文件的映射在NT的层次中称作section)。通常,文件映射是使用文件句柄来关联文件。但有时候也用来指向分页系统中的私有页面。

第三种方法是把文件映射的视图映射到一个进程的地址空间。Win32仅仅允许为当前进程创建一个视图,但是NT潜在的手段更加通用,允许为任意你有权限句柄的进程创建视图。和UNIX中的mmap相比,要区分开创建文件映射和把文件映射到地址空间的操作。

在Windows中,文件映射的内核态实体被句柄所取代。就像许多句柄一样,文件映射能够被复制到其他进程中去。这些进程中的任意一个能够根据需求映射文件到自己的地址空间中。这对共享进程间的私有内存是非常有用的,而且不必再创建文件来实现。在NT层,文件的映射(sections)也和NT名字空间保持一致,能够通过文件名来访问。

对许多程序来说,一个重要的领域是文件I/O操作。在Win32基本视图中,一个文件仅仅是一组有顺序的字节流。Win32提供超过60种调用来创建和删除文件和目录、打开关闭文件、读写文件、提取设置文件属性、锁定字节流范围以及更多基础操作的功能,这些功能基于文件系统的组织以及文件的各自访

Win32 调用	本地NT API调用
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

图11-10 Win32 API调用以及它们所包含的本地NT API调用示例

问权限。

还有更高级的处理文件数据的方法。除了主要的文件流,存在NTFS文件系统上的文件可以拥有额外的文件流。文件(甚至包括整个卷)可以被加密。文件可以被压缩成为一组相对稀疏的字节流,从而节省磁盘空间。不同硬盘的文件系统的卷可以通过使用不同级别的RAID存储而组织起来。修改文件或者目录可以通过一种直接通知的方式来实现,或者通过读NTFS为每个卷维护的日志来实现。

每个文件系统的卷默认挂载在NT的名字空间里,根据卷的名字来排列,因此,一个文件 \foo\bar 可以命名成 \Device\HarddiskVolume\foo\bar。对于NTFS的卷来说,挂载点(Windows称作再分解点)和符号链接用来帮助组织卷。

低级别的Windows I/O模式基本上是异步的。一旦一个I/O操作开始,系统调用将允许线程对I/O操作进行初始化并且开始I/O操作。Windows支持取消操作,以及一系列的不同机制来支持线程和I/O操作完成之后的同步。Windows也允许程序规定在文件打开时I/O操作必须同步,许多库函数,例如C库和许多Win32调用,也规定I/O的同步已支持兼容性或者简化编程模型。在这些情况下,执行体会在返回到用户态前和I/O操作结束时进行同步。

Win32提供的另一些调用是安全性相关的。每个线程将和一个内核对象进行捆绑,称作令牌(token),这个令牌提供关于该线程的身份和权限相关的信息。每个目标可以有ACL(访问权限控制列表),这个列表详细描述了哪种用户有权限访问并且对其进行操作。这种方式通过了一种细粒度的安全机制,可以指定具体哪些用户可以或者禁止访问特定的对象。这种安全模式是可以扩展的,允许应用程序添加新的安全规则,例如限制访问时间。

Win32的名字空间不同于前面描述的NT内核名字空间。NT内核空间仅仅只有一部分对Win32 API函数可见(即使整个NT名字空间可以通过Win32使用特殊字符串来访问,如“\\.”)。在Win32中,文件访问权限和驱动器号相关。NT目录\DosDevices里包含了对一个从驱动器号到实际设备对象的数个符号链接。例如,\DosDevices\C:是指向\Device\HarddiskVolume1。这个目录同样也包含了其他Win32设备的链接,如COM1:、LPT1:和NUL:(端口号和打印端口,以及非常重要的空设备)。\DosDevices是一个真正指向\??的链接,这样有利于提高效率。另外一个NT文件夹,\BaseNamedObjects用来存储各种各样的内核对象,这些文件可以通过Win32 API来访问。这些对象包括用来同步的对象,如信号、共享内存、定时器以及通信端口,MS-DOS和设备名称。

对于底层系统接口,我们额外说一下,Win32 API也支持许多GUI操作,包括系统所有图形接口的调用。有对窗口的创建、摧毁、管理和使用的调用,以及支持菜单、工具条、状态栏、滚动条、对话框、图标和许多在屏幕上显示的元素。Win32还提供调用来画几何图形、填充、使用调色板、处理文字以及在屏幕上放置图标等。也支持对键盘鼠标和其他输入设备的响应,如音频、打印等其他输出设备。

GUI操作直接使用win32k.sys驱动,这个驱动使用特殊的函数从用户态去访问内核态的接口。因为这些调用不包含NT操作系统中的系统调用,我们将不会详细讨论。

11.2.3 Windows 注册表

名字空间的根在内核中维护。存储设备,如系统的卷,附属于名字空间中。因为名字空间会因为系统的每次启动重新构建,那么系统怎么知道系统配置的细节呢?答案就是Windows会挂载一种特殊的文件系统(为小文件做了优化)到名字空间。这个文件系统称作注册表(registry)。注册表被组织成了不同的卷,称作蜂巢(hive)。每个蜂巢保存在一个单独文件中(在启动卷的目录C:\Windows\system32\config\下)。当Windows系统启动时,一个叫做SYSTEM的特殊蜂巢被装入了内存,这是由同样的装载内核和其他启动文件(例如位于启动盘的驱动程序)的程序来完成。

Windows在系统蜂巢里面保存了大量的重要信息,包括驱动程序去驱使什么设备工作,什么软件进行初始化,以及什么变量来控制操作系统的操作等。这些信息甚至被启动程序自己用来决定哪些驱动程序是用于启动的驱动,哪些必须立即需要启动。这些驱动包括操作系统自身来识别文件系统和磁盘驱动的程序。

其他配置蜂巢用在系统启动后,描述系统安装的软件的信息,特别是用户和用户态下安装在系统上的COM(Component Object-Model)。本地用户的登录信息保存在SAM(安全访问管理器)中。网络用

户的信息保存在lsass服务中，和网络服务器文件夹一起，用户可以通过上述两种配置拥有一个访问网络的用户名和密码。Windows Vista的储巢列表在图11-11中显示。

储巢文件	挂载名称	使用
SYSTEM	HKLM\SYSTEM	OS配置信息，供内核使用
HARDWARE	HKLM\HARDWARE	记录探测到的设备的内存储巢
BCD	HKLM\BCD*	启动配置数据库
SAM	HKLM\SAM	本地用账户信息
SECURITY	HKLM\SECURITY	lass的账号和其他信息
DEFAULT	HKEY_USERS\DEFAULT	新用户的默认储巢
NTUSER.DAT	HKEY_USERS<user id>	用户相关的储巢，保存在home目录
SOFTWARE	HKLM\SOFTWARE	COM注册的应用类
COMPONENTS	HKLM\COMPONENTS	SYS组件的清单和依赖

图11-11 Windows Vista中的注册表储巢。HKLM是HKEY_LOCAL_MACHINE的缩写

在引入注册表之前，Windows的配置信息保存在大量的.ini文件里，分散在硬盘的各个地方。注册表则把这些文件集中存储，使得这些文件可以在系统启动的过程中引用。这对Windows热插拔功能是很重要的。但是，随着Windows的发展，注册表已经变得无序。有些关于配置的信息的协议定义得很差，而且很多应用程序采取了特殊的方法。许多用户、应用程序以及所有驱动程序在运行时具有私有权限，而且经常直接更改注册表的系统参数——有时候会妨碍其他程序导致系统不稳定。

注册表是位于数据库和文件系统之间的一个交叉点，但是和每一个都不像。有整本描写注册表的书 (Born, 1998; Hipson, 2000; Ivens 1998)。有很多公司开发了特殊的软件去管理复杂的注册表。

regedit能够以图形窗口的方式来浏览注册表，这个工具允许你查看其中的文件夹（称作键）和数据项（称作值）。微软的新PowerShell脚本语言对于遍历注册表的键和值是非常有用的，它把这些键和值以类似目录的方式来看待。Procmon是一个比较有趣的工具，可以从微软工具网站：www.microsoft.com/technet/sysinternals中找到它。

Procmon监视系统中所有对注册表的访问。有时，一些程序可能会重复访问同一个键达数万次之多。

正如名字所显示的那样，注册表编辑器允许用户对注册表进行编辑，但是一旦你这么做了就必须非常小心。它很容易造成系统无法引导或损坏应用软件的安装，因此没有一些专业技巧就不要去修改它。微软承诺会在以后发布时清理注册表，但现在它仍是庞杂的一堆——比UNIX保留的配置信息复杂得多。

微软Windows Vista已经引入了一个基于事务管理的内核，用来支持对跨越文件系统和注册表操作的事务进行协调。微软计划在未来使用该功能以避免由于软件非完全正确安装而在系统目录和注册表储巢中留下当时局部状态信息所造成的元数据讹用问题。

Win32程序员通过函数调用可以很方便地访问注册表，包括创建、删除键、查询键值等。如图11-12所示。

当系统关闭时，大部分的注册表信息被存储在硬盘储巢中。因为极其严格的完整性要求使得需要纠正系统功能，自动实现备份，将元数据冲写入硬盘以防止在发生系统崩溃时所造成的损坏。注册表损坏需要重新安装系统上的所有软件。

Win32 API 函数	描述
RegCreateKeyEx	创建一个新的注册表键
RegDeleteKey	删除一个注册表键
RegOpenKeyEx	打开一个键并获得句柄
RegEnumKeyEx	列举某个键的下级副键
RegQueryValueEx	查询键内的数据值

图11-12 一些使用注册表的Win32 API 调用

11.3 系统结构

前面的章节从用户态下程序员写代码的角度研究了Windows Vista系统。现在我们将观察系统是如何组织的，不同的部件承担什么工作以及它们彼此间或者和用户程序间是如何配合的。这是实现底层用户态代码的程序开发人员所能看见的操作系统部分，类似于子系统和本地服务，以及提供给设备驱动程序开发者的系统视图。

尽管有很多关于Windows使用方面的书籍，但很少有书讲述它是如何工作的。不过，查阅《Microsoft Windows Internals, 4th ed》(Russionvich 和 Solomon, 2004) 是其中最好的选择之一。该书描述的虽然是Windows XP，但大部分的描述还是准确的。就内部机制而言，Windows XP和Windows Vista是非常相近的。

而且，微软通过Windows学术计划为大学教员和学生提供对其有帮助的Windows内核信息。该计划会发布大部分Windows Server 2003内核源代码、Cutler团队的原始NT设计文档和一大套源自Windows Internals 书籍的表述资料。另外，Windows驱动工具也会提供大量内核工作信息，因为设备驱动器不仅使用I/O设备，还需要使用进程、线程、虚拟内存和进程间的通信等。

11.3.1 操作系统结构

Windows Vista 操作系统包括很多层，如图11-6所示。在以下章节我们将研究操作系统中工作于内核态的最底层层次。其中心就是NOTS内核层自身，当Windows启动时由ntoskrnl.exe加载。NTOS包括两层，executive（执行体）提供大部分的服务，另一个较小的层称为内核（kernel），负责实现基础线程计划和同步抽象，同时也执行陷入句柄中断以及管理CPU的其他方面。

将NTOS分为内核和执行体体现了NT 的VAX/VMS根源。VMS操作系统也是由Cutler团队设计的，可分为4个由硬件实施的层次：用户、管理程序、执行体和内核，与VAX处理机结构提供的4种保护模式一致。Intel CPU也支持这4种保护环，但是一些早期的NT处理机对此不支持，因此内核和执行体表现了由软件实施的抽象，同时VMS在管理者模式下提供的功能，如假脱机打印，NT是作为用户态服务提供的。

NT的内核态层如图11-13所示。NTOS的内核层在执行体层之上，因为它实现了从用户态到内核态转换的陷入和中断机制。图11-13所示的最顶层是系统库ntdll.dll，它实际工作于用户态。系统库包括许多为编译器运行提供的支持功能以及低级库，类似于UNIX中的libc。Ntdll.dll也包括了特殊码输入指针以支持内核初始化线程、分发异常和用户态的异步过程调用（Asynchronous Procedure Calls, APC）等。因为系统库对内核运行是必需的，所以每个由NTOS创建的用户态进程都具有相同固定地址描绘的ntdll。当NTOS初始化系统时，会创建一个局部目标并且记录下内核使用的ntdll输入指针地址。

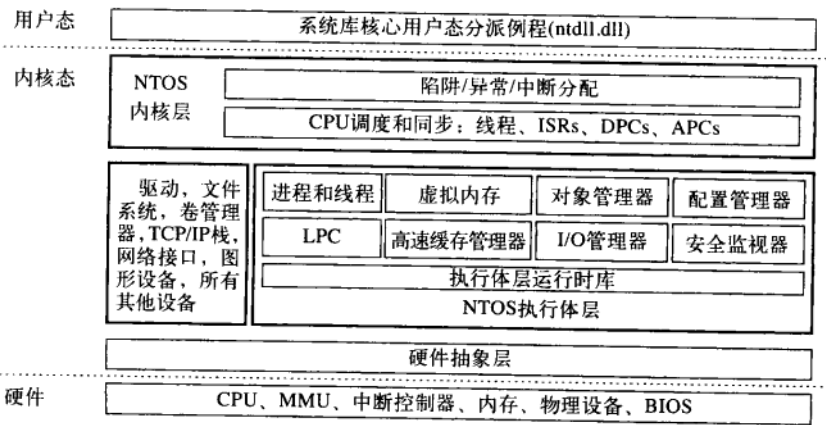


图11-13 Windows 内核态组织结构

在NTOS内核和执行体层之下是称为硬件抽象层（Hardware Abstraction Layer, HAL）的软件，该软件对类似于设备寄存器存取和DMA操作之类的底层硬件信息进行抽象，同时还就BIOS固件是如何表述配置信息和处理CPU芯片上的不同（如各种中断控制器）进行抽象。BIOS可以从很多公司获得，并且被集成成为计算机主板上的永久内存。

内核态下另一个主要部件就是设备驱动器。Windows内核态下任何非NTOS或HAL的设备都会用到设备驱动器，包括文件系统、网络协议栈和其他如防病毒程序、DRM软件之类的内核扩展，以及与硬件总线接口的管理物理设备驱动器等。

I/O和虚拟内存部件协作加载设备驱动程序至内核存储器并将它们连接到NTOS和HAL层。I/O管理器提供发现、组织和操作设备的接口,包括安排加载适当的设备驱动程序等。大多数管理设备和驱动器的配置信息都保留在注册表的系统储巢中。I/O管理器的即插即用下层部件保留硬件储巢内检测出的硬件信息,该储巢是保留在内存中的可变储巢而非存在于硬盘中,系统每次引导都会重新创建。

以下将详细介绍操作系统的不同部件。

1. 硬件抽象层

正如之前发布的基于NT的Windows系统一样,Windows Vista的目标之一是使得操作系统在不同的硬件平台之间具有可移植性。理想情况下,如果需要在一种新型计算机系统中运行该操作系统,仅仅需要在首次运行时使用新机器编译器重新编译操作系统即可。但实际上并没有那么简单。操作系统各层有大量部件具有很好的可移植性(因为它们主要处理支持编程模式的内部数据结构和抽象,从而支持特定的编成模式),其他层就必须处理设备寄存器、中断、DMA以及机器与机器间显著不同的其他硬件特征。

大多数NTOS内核源代码由C语言编写而非汇编语言(x86中仅2%是汇编语言,比x64少1%)。然而,所有这些C语言代码都不能简单地从x86系统中移植到一个SPARC系统,然后重新编译、重新引导,因为与不同指令集无关并且不能被编译器隐藏的处理机结构及其硬件有很多不同。像C这样的语言难以抽象硬件数据结构和参数,如页表输入格式、物理存储页大小和字长等。所有这些以及大量的特定硬件的优化即使不用汇编语言编写,也将不得不手工处理。

大型服务器的内存如何组织或者何种硬件同步基元是可获得的,与此相关的硬件细节对系统较高层都有比较大的影响。例如,NT的虚拟内存管理器和内核层了解涉及内存和内存位置的硬件细节。在整个系统中,NT使用的是比较和交换同步基元,对于没有这些基元的系统是很难移植上去的。最后,系统对字内的字节分类系统存在很多相关性。在所有NT原来移植到的平台上,硬件是设置为小端(little-endian)模式的。

除了以上这些影响便携性的较大问题外,不同制造商的不同主板还存在大量的小问题。CPU版本的不同会影响同步基元的实现方式。各种支持芯片组也会在硬件中断的优先次序、I/O设备寄存器的存取、DMA转换管理、定时器和实时时钟控制、多处理器同步、BIOS设备(如ACPI)的工作等方面产生差异。微软尝试通过最下端的HAL层隐藏对这些设备类型的依赖。HAL的工作就是对这些硬件进行抽象,隐藏处理器版本、支持芯片集和其他配置变更等具体细节。这些HAL抽象展现为NTOS和驱动可用的独立于机器的服务。

使用HAL服务而不直接写硬件地址,驱动器和内核在与新处理器通信时只需要较小改变,而且在多数情况下,尽管版本和支持芯片集不同但只要具有相同的处理器结构,系统中所有部件均无需修改就可运行。

HAL对诸如键盘、鼠标、硬盘等特殊的I/O设备或内存管理单元不提供抽象或服务。这种抽象功能广泛应用于整个内核态的各部件,如果没有HAL,通信时即使硬件间很小的差异也会造成大量代码的重大修改。HAL自身的通信很简单,因为所有与机器相关的代码都集中在一个地方,移植的目标就很容易确定:即实现所有的HAL服务。很多版本中,微软都支持HAL扩展工具包,允许系统制造者生产各自的HAL从而使得其他内核部件在新系统中无需更改即可工作,当然这要在硬件更改不是很大的前提下。

通过内存映射I/O与I/O端口的对比可以更好地了解硬件抽象层是如何工作的。一些机器有内存映射I/O,而有的机器有I/O端口。驱动程序是如何编写的呢?是不是使用内存映射I/O?无需强制做出选择,只需要判断哪种方式使驱动程序可独立于机器运行即可。硬件抽象层为驱动程序编写者分别提供了三种读、写设备寄存器的程序:

```
uc=READ_PORT_UCHAR(port); WRITE_PORT_UCHAR(port,uc);
us=READ_PORT_USHORT(port); WRITE_PORT_USHORT (port,us);
ul=READ_PORT_ULONG(port); WRITE_PORT_ULONG(port,ul);
```

这些程序各自在指定端口读、写无符号8、16、32位整数,由硬件抽象层决定这是否需要内存映射I/O。这样,驱动程序可以在设备寄存器实现方式有差异的机器间使用而不需要修改。

驱动程序会因为不同目的而频繁存取特定的I/O设备。在硬件层,一个设备在确定的总线上有一个或多个地址。因为现代计算机通常有多个总线(ISA、PCI、PCI-X、USB、1394等),这就可能造成不

同总线上的多个设备有相同的地址,因此需要一些方法来区别它们。HAL把与总线相关的设备地址映射为系统逻辑地址并以此来区分设备。这样,驱动程序就无需知道何种设备与何种总线相关联。这种机制也保护了较高层避免进行总线结构和地址规约的交替。

中断也存在相似的问题——总线依赖性。HAL 同样提供服务在系统范围内命名中断,并且允许驱动程序将中断服务程序附在中断内而无需知道中断向量与总线的关系。中断请求管理也受HAL控制。

HAL提供的另一个服务是在设备无关方式下建立和管理DMA转换,对系统范围和专用I/O卡的DMA引擎进行控制。设备由其逻辑地址指示。HAL实现软件的散布/聚合(从不相邻的物理内存块的地方写或者读)。

HAL也是以用一种可移植的方式来管理时钟和定时器的。定时器是以100纳秒为单位从1601年1月1日开始计数的,因为这是1601年的第一天,简化了闰年的计算。(一个简单测试:1800年是闰年吗? 答案:不是。)定时器服务和驱动程序中的时钟运行的频率是解耦合的。

有时需要在底层实现内核部件的同步,尤其是为了防止多处理机系统中的竞争环境。HAL提供基元管理同步,如旋转锁,此时一个CPU等待其他CPU释放资源,比较特殊的情况是资源被几个机器指令占有。

最终,系统引导后,HAL和BIOS通信,检查系统配置信息以查明系统所包含的总线、I/O设备及其配置情况,同时该信息被添加进注册表。HAL工作情况摘要如图11-14所示。

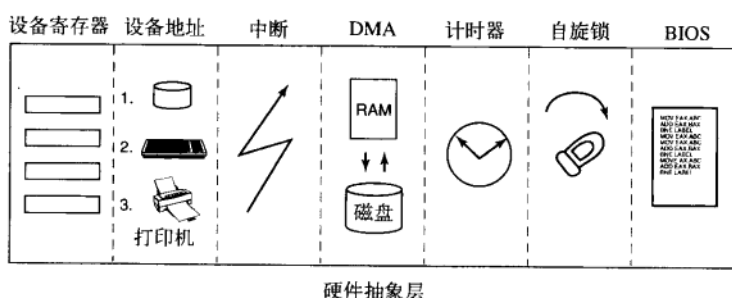


图11-14 一些HAL管理相关的硬件功能

2. 内核层

在硬件抽象层之上是NTOS,包括两层:内核和执行体。“内核”在Windows中是一个易混淆的术语。它可以指运行在处理机内核态下的所有代码,也可以指包含了Windows操作系统核心NTOS的ntoskrnl.exe文件,还可以指NTOS里的内核层,在本章中我们使用这个概念。此外,“内核”甚至用来命名用户态下提供本地系统调用的封装器的Win32库:kernel32.dll

Windows操作系统的内核层(如图11-13所示,执行体之上)提供了一套管理CPU的抽象。最核心的抽象是线程,但是内核也实现了异常处理、陷阱以及各种中断。支持线程的数据结构的创建和终止是在执行体实现的。核心层负责调度和同步线程。在一个单独的层内支持线程,允许执行体在用户态下,可以通过使用用来编写并行代码且相同优先级的多线程模型来执行,但同步原语的执行更专业。

内核线程调度程序负责决定哪些线程执行在系统的每一个CPU上。线程会一直执行,直到产生了一个定时器中断,或者是当线程需要等待一些情况,比如等待一个I/O读写完成或是一个锁定被释放,或者是更高优先级的线程等待运行而需要CPU,这时正在执行的线程会切换到另一个线程(量子过期)。当一个线程向另一个线程转换时,调度程序会在CPU上运行,并确保寄存器及其他硬件状态已保存。然后,调度程序会选择另一个线程在CPU上运行,并且恢复之前所保存的最后一个线程的运行状态。

如果下一个运行的线程是在一个不同的地址空间(例如进程),调度程序也必须改变地址空间。详细的调度算法我们将在本章内谈到进程和线程时讨论。

除了提供更高级别的硬件抽象和线程转换机制,核心层还有另外一项关键功能:提供对下面两种同步机制低级别的支持:control对象和dispatcher对象。Control对象,是核心层向执行体提供抽象的CPU管理的一种数据结构。它们由执行体来分配,但由核心层提供的例程来操作。Dispatcher对象是一种普通执行对象,使用一种公用的数据结构来同步。

3. 延迟过程调用

Control对象包括线程、中断、定时器、同步、调试等一些原语对象，和两个用来实现DPC和APC的特殊对象。DPC（延迟过程调用）对象是用来减少执行ISR（中断服务例程）所需要的时间，以响应从特定的设备来的中断。

系统硬件为中断指定了硬件优先级。在CPU进行工作时也伴随着一个优先级。CPU只响应比当前更高优先级的中断。通常的优先级是0，包括所有用户态下的优先级。设备中断发生在优先级3或更高，让一个设备中断的ISR以同一优先级的中断来执行是防止其他不重要的中断影响它正在进行的重要中断。

如果ISR执行得太长，提供给低优先级中断的服务将被推迟，可能造成数据丢失或减缓系统的I/O吞吐量。多ISR可以在任何同一时刻处理，每一个后续的ISR是由于产生了更高优先级的中断。

为了减少处理ISR所花费的时间，只有关键的操作才执行，如I/O操作结果的捕捉和设备重置。直到CPU的优先级降低，且没有其他中断服务阻塞，才会进行下一步的中断处理。DPC对象用来表示将要要做的工作，ISR调用核心层排列DPC到特定处理器上的DPC队列。如果DPC在队列的第一个位置，内核会登记一个特殊的硬件请求让CPU在优先级2产生中断（NT下称为DISPATCH级别）。当最后一个执行的ISR完成后，处理器的中断级别将回落到低于2，这将解开DPC处理中断。服务于DPC中断的ISR将会处理内核排列好的每一个DPC对象。

利用软中断延迟中断处理是一种行之有效的减少ISR延迟时间的方法。UNIX和其他系统在20世纪70年代开始使用延迟处理，以处理缓慢的硬件和有限的缓冲串行连接终端。ISR负责处理从硬件提取字符并排列它们。在所有高级别的中断处理完成以后，软中断将执行一个低优先级的ISR做字符处理，比如通过向终端发送控制字符来执行一个退格键，以抹去最后一个显示字符并向后移动光标。

在当前的Windows操作系统下，类似的例子是键盘设备。当一个键被敲击以后，键盘ISR从寄存器中读取键值，然后重新使键盘中断，但并不对下一步的按键进行及时处理。相反，它使用一个DPC去排队处理键值，直到所有优先的设备中断已处理完成。

因为DPC在级别2上运行，它们并不干涉ISR设备的执行，在所有排队中的DPC执行完成并且CPU的优先级低于2之前，它们会阻止任何线程的运行。设备驱动和系统本身必须注意不要运行ISR或DPC太长时间。因为在运行它们的时候不能运行线程，ISR或DPC的运行会使系统出现延迟，并且可能在播放音乐时产生不连续，因为拖延了线程对声卡的音乐缓冲区的写操作。DPC另一个通常的用处是运行程序以响应定时器中断。为了避免线程阻塞，要延长运行时间的定时器事件需要向内核维持后台活动的线程工作池做排队请求。这些线程有调度优先级12、13或15。我们会在线程调度部分看到，这些优先级意味着工作项目将会先于大多数线程执行，但是不会打断实时线程。

4. 异步过程调用

另一个特殊的内核控制对象是APC（异步过程调用）对象。APC与DPC的相同之处是它们都是延迟处理系统例行程序，不同之处在于DPC是在特定的CPU上下文中执行，而APC是在一个特定的线程上下文中执行。当处理一个键盘敲击操作时，DPC在哪个上下文中运行是没有关系的，因为一个DPC仅仅是处理中断的另一部分，中断只需要管理物理设备和执行独立线程操作，例如在内核空间的一个缓冲区记录数据。

当原始中断发生时，DPC例程运行在任何线程的上下文中。它利用I/O系统来报告I/O操作已经完成，I/O系统排列一个APC在线程的上下文中运行从而做出原始的I/O请求，在这里它可以访问处理输入的线程的用户态地址空间。

在下一个合适的时间，内核层会将APC移交给线程而且调度线程运行。一个APC被设计成看上去像一个非预期的程序调用，有些类似于UNIX中的信号处理程序。不过在内核态下，内核态的APC为了完成I/O操作，而在完成初始化I/O操作的线程的上下文中执行。这使APC既可以访问内核态的缓冲区，又可以访问用户态下，属于包含线程的进程的地址空间。一个APC在什么时候被移交，取决于线程已经在做什么，以及系统的类型是什么。在一个多处理器系统中，甚至是在DPC完成运行之前，接收APC的线程才可以开始执行。

用户态下的APC也可以用来把用户态的I/O操作已经完成的信息，通知给初始化I/O操作的线程。但

只有当内核中的目标线程被阻塞和被标示为准备接收APC时,用户态下的APC才可调用用户态下的应用程序。但随着用户态堆栈和寄存器的修改,为了执行在ntdll.dll系统库中的APC调度算法,内核将等待中的线程中断,并返回到用户态。APC调度算法调用和I/O操作相关的用户态应用程序。除了一些I/O完成后,作为一种执行代码方法的用戶态下的APC外,Win32 API中的QueueUserAPC允许将APC用于任意目的。

执行体也使用除了I/O完成之外的一些APC操作。由于APC机制精心设计为只有当它是安全的时候才提供APC,它可以用来安全地终止线程。如果这不是一个终止线程的好时机,该线程将宣布它已进入一个临界区,并延期交付APC直至得到许可。在获得锁或其他资源之前,内核线程会标记自己已进入临界区并延迟APC,这时,它们不能被终止,并仍然持有资源。

5. 调度对象

另一种同步对象是调度对象。这是常用的内核态对象(一种用户可以通过句柄处理的类型),它包含一个称为dispatcher_header的数据结构,如图11-15所示。

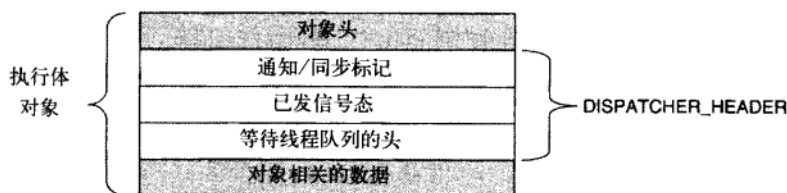


图11-15 执行对象中嵌入的dispatcher_header数据结构

它们包括信号器、互斥体、事件、可等待定时器和一些可以等待其他线程同步执行的对象。它们还包括表示打开的文件的对象、进程、线程和IPC端口。调度数据结构包含了表示对象状态的标志,和等待被标记的对象的线程队列。

同步原语,如信号器,是标准的调度对象。另外定时器、文件、端口线程和进程使用调度对象机制去通知。当一个定时器开启、一个文件I/O完成、一个端口正在传输数据或是一个线程或进程终止时,相关的调度对象会被通知,并唤醒所有等待该事件的线程。

由于Windows使用了一个单一的标准机制去同步内核态对象,一些专门的API就无需再等待事件,例如在UNIX中用来等待子进程的wait3。而通常情况下,线程要一次等待多个事件。在UNIX中,通过“select”系统调用,一个进程可以等待任何一个64位网络接口可以获得的数据。在Windows中亦有一个类似的API WaitForMultipleObjects,但是它允许一个线程等待任何类型的有句柄的调度对象。超过64个句柄可以指定WaitForMultipleObjects,以及一个可选择的超时值。线程随时准备运行任何一个和句柄标记相关的事件或发生超时。

内核使用两个不同的程序使得线程等待调度对象运行。发出一个通知对象信号使每一个等待的线程可以运行。同步对象仅使第一个等待的线程可以运行,用于调度对象,实施锁元,如互斥体。当一个线程等待一个锁再次开始运行,它做的第一件事就是再次尝试请求锁。如果一次仅有一个线程可以保留锁,其他所有可运行的线程可能立刻被阻塞,从而产生许多不必要的现场交换。使用同步机制和使用通知机制的分派对象(dispatcher object)之间的差别是dispatcher_header结构中的一个标记。

另外,在Windows代码中互斥体称为“变体”(mutant)。因为当一个线程保留一个出口时,它们需要执行OS/2语义中的非自动解锁,看来这是Cutler奇特的考虑。

6. 执行体

如图11-13所示,在NTOS的内核层以下是执行体。执行体是用C语言编写的,在结构上最为独立(内存管理是一个明显的例外),并且经过少量的修改已经移植到新的处理器上(MIPS、x86、PowerPC、Alpha、IA64和x64)。执行体包括许多不同的组件,所有的组件都通过内核层提供的抽象控制器来运行。

每个组件分为内部和外部的数据结构和接口。每个组件的内部方法是隐藏的,只有组件自己可以调用,而外部方法可以由执行体的所有其他组件调用。外部接口的一个子集由一个ntoskrnl.exe提供,而且设备驱动可以链接到它们就好像执行体是一个库。微软称许多执行体组件为“管理器”,因为每一个组

件管理操作系统的一部分，例如I/O、内存、进程、对象等。

对于大多数操作系统而言，许多功能在Windows上执行就像库的编码。除非在内核方式下运行，它的数据结构可以被共享和保护，以避免用户态下的编码访问，因此它具有硬件状态的访问权限，例如MMU控制寄存器。但是另一方面，执行体只是代表它的调用者简单执行操作系统的函数，因此它运行在它的调用者的线程中。

当任何执行控制操作阻塞等待与其他线程同步时，用户态线程也会阻塞。这在为一个特殊的用户态线程工作时是有意义的，但是在做一些相关的内务处理任务时是不公平的。当执行体认为一些内务处理线程是必须的时候，为了避免劫持当前的线程，一些内核态线程就会具体于特定的任务而产生，例如确保更改了的页面被回写到硬盘上。

对于可预见的低频率任务，会有一个线程一秒运行一次而且由一个长的项目单来处理。对于不可预见的工作，有一个之前曾经提到的高优先级的辅助线程池，通过将队列请求和发送辅助线程等待的同步事件信号，可以用来运行有界任务。

对象管理器管理在执行体使用的大部分内核态对象，包括进程、线程、文件、信号、I/O设备及驱动、定时器等。就像之前提到的，内核态对象仅仅是内核分配和使用的数据结构。在Windows中，内核数据结构有许多共同特点，即它们在管理标准功能中特别有用。

这些功能由对象管理器提供，包括管理对象的内存分配和释放，配额计算，支持通过句柄访问对象，为内核态指针引用保留引用计数，在NT名字空间给对象命名，为管理每一个对象的生命周期提供可扩展的机制。需要这些功能的内核数据结构是由对象管理器来管理的。其他数据结构，例如内核层使用的控制对象，或仅仅是内核态对象的扩展对象，不由对象管理器管理。

对象管理器的每一个对象都有一个类型用来指定这种类型的对象的生命周期怎样被管理。这些不是面向对象意义中的类型，而仅仅是当对象类型产生时的一个指定参数集合。为了产生一个新的类型，一个操作元件只需要调用一个对象管理器API即可。对象在Windows的函数中很重要，在下面的章节中将会讨论有关对象管理器的更多细节。

I/O管理器为实现I/O设备驱动提供了一个框架，同时还为设备上的配置、访问和完成操作提供一些特定的运行服务。在Windows中，设备驱动器不仅仅管理硬件设备，它们还为操作系统提供可扩展性。在其他类型的操作系统中被编译进内核的功能是被Windows内核动态装载和链接的，包括网络协议栈和文件系统。

最新的Windows版本对在用户态上运行设备驱动程序有更多的支持，这对新的设备驱动程序是首选的模式。Windows Vista有超过100万不同的设备驱动程序，工作着超过了100万不同的设备。这就意味着要获取正确的代码。漏洞导致设备在用户态的进程中崩溃而不能使用，这比造成对系统进行检测错误要好得多。错误的内核态设备驱动是导致Windows 可怕的BSOD（蓝屏死机）的主要来源，它是Windows检测到致命的内核态错误并关机或重新启动系统。蓝屏死机可以类比为UNIX系统中的内核恐慌。

在本质上，微软现在已经正式承认那些在microkernels研究领域的如MINIX 3和L4的研究员多年来都知道的结果：在内核中有更多的代码，那么内核中就有更多缺陷。由于设备驱动程序占了70%的内核代码，更多的驱动程序可以进入用户态进程，其中一个bug只会触发一个单一驱动器的失败（而不是降低整个系统）。从内核到用户态进程的代码移动趋势将在未来几年加速发展。

I/O管理器还包括即插即用和电源管理设施。当新设备在系统中被检测到，即插即用就开始工作。该即插即用设备的子模块首先被通知。它与服务一起工作，即用户态即插即用管理器，找到适当的设备驱动程序并加载到系统中。找到合适的设备驱动程序并不总是很容易，有时取决于先进的匹配具体软件设备特定版本的驱动程序。有时一个单一的设备支持一个由不同公司开发的多个驱动程序所支持的标准接口。

电源管理能降低能源消耗，延长笔记本电脑电池寿命，保存台式电脑和服务器能量。正确使用电源管理是具有挑战性的，因为在把设备和buses连接到CPU和内存时有许多微妙的依赖性。电力消耗不只是由设备供电时的影响，而且还由CPU的时钟频率影响，这也是电源管理在控制。

我们会在11.7节对I/O进一步研究和以及在11.8节中介绍最重要的NT文件系统NTFS。

进程管理管理着进程和线程的创建和终止，包括建立规则和参数指导它们。但是线程运行方面由核

心层决定,它控制着线程的调度和同步,以及它们之间相互控制的对象,如APC。进程包含线程、地址空间和一个可以用来处理进程指定内核态对象的句柄表。进程还包括调度器进行地址空间交换和管理进程中的具体硬件信息(如段描述符)所需要的信息。我们将在11.4节研究进程和线程的管理。

执行内存管理器实现了虚拟内存架构的需求分页。它负责管理虚拟页映射到物理页帧,管理现有的物理帧,和使用备份管理磁盘上页面文件,这些页面文件是用来备份那些不再需要加载到内存中的虚拟页的私有实例。该内存管理器还为大型服务器应用程序提供了特殊功能,如数据库和编程语言运行时的组件,如垃圾收集器。我们将在11.5节中研究内存管理。

内存管理器优化I/O的性能,文件系统内核虚拟地址空间保持一个内存的文件系统页。内存管理器使用虚拟的地址进行缓存,也就是说,按照它们文件所在位置来组织缓存页。这不同于物理块内存,例如在UNIX中,系统为原始磁盘卷保持一个物理地址块的内存。

内存的管理是使用内存映射文件来实现的。实际的缓存是由内存管理器完成。内存管理器需要关心的只是文件的哪些部分需要内存,以确保缓存的数据即时地刷新到磁盘中,并管理内核虚拟地址映射缓存文件页。如果一个页所需的I/O文件在缓存中没有,该页在使用内存管理器时将会发生错误。我们会在11.6节中学习内存管理器。

安全引用监视器(security reference monitor)执行Windows详细的安全机制,以支持计算机安全要求的国际标准的通用标准(Common Criteria),一个由美国国防部的橘皮书的安全要求发展而来的标准。这些标准规定了一个符合要求的系统必须满足的大量规则,如登录验证、审核、零分配的内存等更多的规则。一个规则要求,所有进入检查都由系统中的一个模块进行检查。在Windows中此模块就是内核中的安全监视器。我们将在11.9节中更详细地学习安全系统。

执行体中包括其他一些组件,我们将简要介绍。如前所述,配置管理实现注册表的执行组件。注册表中包含系统配置数据的文件的系统文件称为储巢(hive)。最关键的储巢是系统启动时加载到内存的系统储巢。只有在执行体成功地初始化其主要组件,包括了系统磁盘的I/O驱动程序,之后才是文件系统中储巢关联的内存中的储巢副本。因此,如果试图启动系统时发生不测,磁盘上的副本是不太可能被损坏的。

LPC的组成部分提供了运行在同一系统的进程之间的高效内部通信。这是一个基于标准的远程过程调用(RPC)功能,实现客户机/服务器的处理方式的数据传输。RPC还使用命名管道和TCP/IP作为传输通道。

在Windows Vista(现在称为ALPC、高级LPC)中LPC大大加强了对RPC新功能的支持,包括来自内核态组件的RPC,如驱动。LPC是NT原始设计中的一个重要的组成部分,因为它被子系统层使用,实现运行在每个进程和子系统进程上库存例程的通信,这实现了一个特定操作系统的个性化功能,如Win32或POSIX。

Windows NT 4.0中的许多代码与Win32进入内核的图形界面相关,因为当时的硬件无法提供所需的性能。该代码以前位于csrss.exe子系统进程,执行Win32接口。以内核为基础的图形用户界面的代码位于一个专门的内核驱动win32k.sys中。这一变化预计将提高Win32的性能,因为额外的用户态/内核态的转换和转换地址空间的成本经由LPC执行通信是被清除的。但并没能像预期的那样取得成功,因为运行在内核中的代码要求是非常严格的,运行在内核态上的额外消耗抵消了因减少交换成本获得的收益。

7. 设备驱动程序

最后一部分图11-13是设备驱动程序的组成。在Windows中的设备驱动程序的动态链接库是由NTOS装载。虽然它们主要是用来执行特定硬件的驱动程序,如物理设备和I/O总线,设备驱动程序的机制也可作为内核态的一般可扩展性的机制。如上所述,大部分的Win32子系统是作为一个驱动程序被加载。

I/O管理器组织的数据按照一定的路线流经过每个设备实例,如图11-16。这个路线称为设备栈,由分配到这条路线上的内核设备对象的私有实例组成。设备堆栈中的每个设备对象与特定的驱动程序对象相关联,其中包含日常使用的I/O请求的数据包流经该设备堆栈的表。在某些情况下,堆栈中的设备驱动程序表示其唯一的目的是在某一特定的设备上过滤I/O操作目标、总线或网络驱动器。过滤器的使用是有一些原因的。有时预处理或后处理I/O操作可以得到更清晰的架构,而其他时候只是以实用为出发

点,因为没有修改驱动的来源和权限,过滤器是用来解决这个问题的。过滤器还可以全面执行新的功能,如把磁盘分区或多个磁盘分成RAID卷。

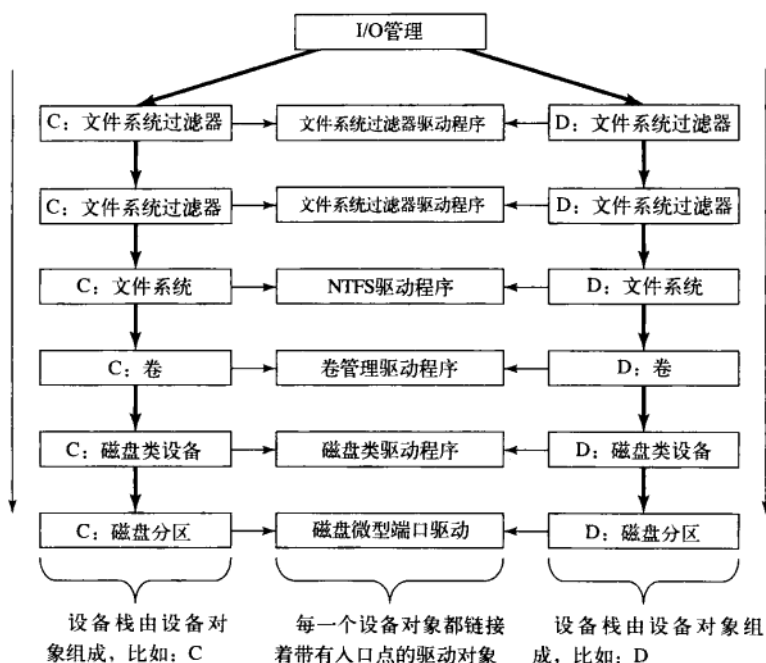


图11-16 简单描绘两个NTFS文件卷的设备栈。I/O请求包由上往下通过栈。每一级堆栈中的相关驱动中的适当程序被调用。该设备栈由分配给每个堆栈的设备对象组成

文件系统作为驱动程序被加载。每个文件系统卷的实例,有一个设备对象创建,并作为该设备堆栈卷的一部分。这是设备对象将与驱动对象的文件系统适当的卷格式发生关联。特别过滤驱动程序,称为文件系统过滤驱动程序,可以在插入设备对象之前,文件系统设备对象将功能应用于被发送到每个卷的I/O请求,如数据读取或写入的病毒检查。

网络协议也作为使用I/O模型的驱动被装载起来,例如Windows Vista整合的IPv4/IPv6 TCP/IP实现。对于老的基于MS-DOS的Windows操作系统,TCP/IP驱动实现了一个特殊的Windows I/O模型网络接口上的协议。还有其他一些驱动也执行这样的安排,其中的Windows小型端口。共享功能是在一个类驱动程序中。例如,SCSI或IDE磁盘或USB设备通用功能是作为一类驱动提供的,这一类驱动为这些设备的每个特定类型提供微端口驱动程序连接为一个库。

我们在本章不讨论任何特定的设备驱动,但是在11.7节中将更为详细地介绍有关I/O管理器如何与设备驱动互动。

11.3.2 启动Windows Vista

使用操作系统需要运行几个步骤。当电脑打开时,CPU初始化硬件。然后开始执行内存中的一个程序。但是,唯一可用的代码是由计算机制造商初始化的某些非易失性的CMOS内存形式(有时被用户更新,在一个进程中称为闪存)。在大多数PC机中,最初的初始化程序是BIOS(基本输入/输出系统),它知道如何在一台PC机上找到设备的标准类型。BIOS提供了Windows Vista在磁盘驱动器分区开始时首先装载的小引导程序。

引导程序知道如何在根目录的文件系统卷之外阅读足够的信息去发现独立的Windows BootMgr程序。BootMgr确定系统是否已经处于休眠或待机模式(特别省电模式,系统不需要重启就可以重新打开)。如果是,BootMgr加载和执行WinResume.exe。否则加载和执行WinLoad.exe执行新的启动。WinLoad加

载系统启动组件到内存中: 内核/执行体(通常是Ntoskrnl.exe)、HAL(hal.dll), 该文件包含系统储巢, Win32k.sys驱动包含Win32子系统的内核态部分, 以及任何其他在系统储巢中作为启动驱动程序列出的驱动程序的镜像, 这就意味着在系统启动时, 它们是必需的。

一旦Windows启动组件加载到内存中, 控制就转移给NTOS中的低级代码, 来完成初始化HAL、内核和执行体、链接驱动像、访问/更新系统配置中的数据等操作。所有内核态的组件初始化后, 第一个用户态进程被创建, 使用运行着的smss.exe程序(如同UNIX系统中的/etc/init)。

Windows启动程序在遇到系统启动失败时, 有专门处理常用问题的逻辑。有时安装一个坏的设备驱动程序, 或运行一个像注册表一样的程序(能导致系统储巢损坏), 会阻止系统正常启动。系统提供了一种功能来支持忽略最近的变化并启动到最近一次的系统正确配置。其他启动选项包括安全启动, 它关闭了许多可选的驱动程序。还有故障恢复控制台, 启动cmd.exe命令行窗口, 它提供了一个类似UNIX的单用户态。

另一个常见的问题, 用户认为, 一些Windows系统偶尔看起来很不可思议, 经常有系统和应用程序的(看似随机)崩溃。从微软的在线崩溃分析程序得到的数据, 提供了许多崩溃是由于物理内存损坏导致的证据。所以Windows Vista启动进程提供了一个运行广义上的内存诊断的选项。也许未来的PC硬件将普遍支持ECC(或者部分)内存, 但是今天的大多数台式机和笔记本电脑系统很容易受到攻击, 即便是在它们所包含的数十亿比特的内存中的单比特错误。

11.3.3 对象管理器的实现

对象管理器也许是Windows可执行过程中一个最重要的组件, 这也是为什么我们已经介绍了它的许多概念。如前所述, 它提供了一个统一的和一致的接口, 用于管理系统资源和数据结构, 如打开文件、进程、线程、内存部分、定时器、设备、驱动程序和信号。更为特殊的对象可以表示一些事物, 像内核的事务、外形、安全令牌和由对象管理器管理的Win32桌面。设备对象和I/O系统的描述联系在一起, 包括提供NT名字空间和文件系统卷之间的链接。配置管理器使用一个Key类型的对象与注册配置相链接。对象管理器自身有一些对象, 它用于管理NT名字空间和使用公共功能来实现对象。在这些目录中, 有象征性的联系和对象类型的对象。

由对象管理器提供的统一性有不同的方面。所有这些对象使用相同的机制, 包括它们是如何创建、销毁以及定额分配值的占有。它们都可以被用户态进程通过使用句柄访问。在内核的对象上有一个统一的协议管理指针的引用。对象可以从NT的名字空间(由对象管理器管理)中得到名字。调度对象(那些以信号事件相关的共同数据结构开始的对象)可以使用共同的同步和通知接口, 如WaitForMultipleObjects。有一个共同的安全系统, 其执行了以名称来访问的对象的ACL, 并检查每个使用的句柄。甚至有工具帮助内核态开发者, 在使用对象的过程中追踪调试问题。

理解对象的关键, 是要意识到一个(执行)对象仅仅是内核态下在虚拟内存中可以访问的一个数据结构。这些数据结构, 常用来代表更抽象的概念。例如, 执行文件对象会为那些已打开的系统文件的每一个实例而创建。进程对象被创建来代表每一个进程。

一种事实上的结果是, 对象只是内核数据结构, 当系统重新启动时(或崩溃时)所有的对象都将丢失。当系统启动时, 没有对象存在, 甚至没有对象类型描述。所有对象类型和对象自身, 由对象管理器提供接口的执行体的其他组件动态创建。当对象被创建并指定一个名字, 它们可以在以后通过NT名字空间被引用。因此, 建立对象的系统根目录还建立了NT名字空间。

对象结构, 如图11-17所示。每个对象包含一个对所有类型的所有对象的某些共性信息头。在这个头的领域内包括在名字空间内的对象的名称, 对象目录, 并指向安全描述符代表的ACL对象。

对象的内存分配来自由执行体保持的两个堆(或池)的内存之一。在有(像内存分配)效用函数的执行体中, 允许内核态组件不仅分配分页内核内存, 也分配无分页内核内存。对于那些需要被具有CPU 2级以及更高优先级的对象访问的任何数据结构和内核态是对象, 无分页内存都是需要的。这包括ISR和DPC(但不包括APC)和线程调度本身。该pagefault处理也需要由无分页内核内存分配的数据结构, 以避免递归。

大部分来自内核堆管理器的分配, 是通过使用每个处理器后备名单来获得的, 这个后备名单中包含分配大小一致的LIFO列表。这些LIFO优化不涉及锁的运作, 可提高系统的性能和可扩展性。

指令的结果，都是储存在内核内存的进程私有句柄表的一个64位句柄表入口。表中句柄逻辑位置的32位索引返回给用户用于随后的指令。内核的64位句柄表入口包含两个32位字节。一个字节包含29位指针指向包头。其后的3位作为标志（例如，表示句柄是否被它创建的进程继承）。这3位在指针就位以前是被屏蔽掉的。其他的字节包含一个32位正确掩码。这是必需的因为只有当对象创建或打开的时候许可校验才会进行。如果一个进程对某对象只有只读的权限，那在表示其他在掩码中的权限位都为0，从而让操作系统可以拒绝除读之外对对象进行任何其他的操作。

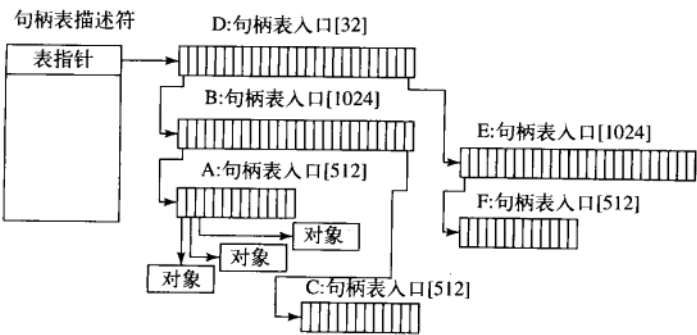


图11-19 最多达到1600万个句柄的句柄表数据结构

2. 对象名字空间

进程可以通过由一个进程把到对象的句柄复制给其他进程来共享对象。但是这需要复制的进程有到其他进程的句柄，而这样在多数情况中并不适用，例如进程共享的对象是无关的或被其他进程保护的。在其他情况下，对象即使在不被任何进程调用的时候仍然保持存在是非常重要的，例如表示物理设备的对象，或用户实现对象管理器和他自己的NT名字空间的对象。为了地址的全面分享和持久化需求，对象管理允许随意的对象在被创建的时候就给定其NT名字空间中的名字。然而，是由执行部件控制特定类型的对象来提供接口，以使用对象管理器的命名功能。

NT名字空间是分级的，借由对象管理器实现目录和特征连接。名字空间也是可扩展的，通过提供一个叫做Parse的进程程序允许任何对象类型指定名字空间扩展。Parse程序是一个可以提供给每一个对象类型的对象创建时使用的程序，如图11-20所示。

程序	使用时候	备注
Open	用于每个新的句柄	很少使用
Parse	用于扩展名字空间的对象类型	用于文件和档案密钥
Close	最后句柄关闭	清除可见结果
Delete	最后一个指针撤销	对象将被删除
Security	得到或设置对象的安全描述符	保护
QueryName	得到对象名称	外核很少使用

图11-20 用于指定一个新对象类型的对象语句

Open语句很少使用，因为默认对象管理器的行为才是必需的，所以程序为所有基本对象类型指定为NULL。

Close和Delete语句描述对象完成的不同阶段。当对象的最后一个句柄关闭，可能会有必要的动作清除状态，这些由Close语句来执行，当最后的指针参考从对象移除，使用Delete语句，从而对象可以准备被删除并使其内存可以重用。利用文件对象，这两个语句都实现为I/O管理器里面的回调，I/O管理器是声明了对象类型的组件。对象管理操作使得由设备堆栈发送的I/O操作能够与文件对象关联上，而大多数这些工作由文件系统完成。

Parse语句用来打开或创建对象，如文件和登录密码，以及扩展NT名字空间。当对象管理器试图通

过名称打开一个对象并遭遇其管理的名字空间树的叶结点，它检查该叶结点对象类型是否指定了一个 Parse 语句。如果有，它会引用该语句，将路径名中未用的部分传给它。再以文件对象为例，叶子结点是一个表现特定文件系统卷的设备对象。Parse 语句由 I/O 管理器执行，并发起在对文件系统的 I/O 操作，以填充一个指向文件的公开实例到该文件对象，这个文件是由路径名指定的。我们将在以后逐步探索这个特殊的实例。

QueryName 语句是用来查找与对象关联的名字。Security 语句用于得到、设置或删除该安全描述符的对象。对于大多数类型的对象，此程序在运行的安全引用监视器组件里提供一个标准的切入点。

注意，在图 11-20 里的语句并不执行每种对象类型最感兴趣的操作。相反，这些程序提供给对象管理器正确实现功能所需要的回调函数，如提供对对象的访问和对对象完成时的清理工作。除了这些回调，对象管理器还提供了一套通用对象例程，例如创建对象和对象类型，复制句柄，从句柄或者名字获得引用指针，并增加和减去对象头部的参考计数。

对象感兴趣的操作都是在本地 NT API 系统调用，如 NtCrestProcess、NtCreateFile 或 NtClose（关闭句柄所有类型的通用操作），如图 11-9 所示。

虽然对象名字空间对整个运作的系统是至关重要的，但却很少有人知道它的存在，因为没有特殊的浏览工具的话它对用户是不可见的。winobj 就是一个这样的浏览工具，在 www.microsoft.com/technet/sysinternals 可免费获得。在运行时，此工具描绘的对象的名字空间通常包含对象目录，如图 11-21 列出来的及其他一些。

一个被奇怪地命名为 \?? 的目录包含用户的所有 MS-DOS 类型的设备名称，如 A：表示软驱，C：表示第一块硬盘。这些名称其实是在设备对象活跃的地方链接到目录 \ 装置的符号。使用名称 \?? 是因为其按字母顺序排列第一，以加快查询从驱动器盘符开始的所有路径名称。其他的对象目录的内容应该是自解释的。

目录	内容
??	查找类似 C: 的 MS-DOS 设备的查找起始位置
DosDevices	目录 ??
Device	所有 I/O 设备
Driver	每个加载的设备驱动对应的对象
ObjectTypes	如图 11-22 中列出的类型的对象
Windows	发送消息到所有 Win32 GUI 窗口的对象
BaseNamedObjects	用户创建的 Win32 对象，如信号量、互斥量等
Arcname	由启动装载器发现的分区名称
NLS	National 语言支持对象
FileSystem	文件系统驱动对象和文件系统识别对象
Security	安全系统的对象
KnownDLLs	较早开启和一直开启的关键共享库

图 11-21 在对象名字空间中的一些典型目录

如上所述，对象管理器保持一个单独的句柄为每个对象计数。这个计数是从来不会大于指针引用计数，因为每个有效的句柄对象在它的句柄表入口有一个引用指针。使用单独句柄计数的理由是，当最后一个用户态的引用消失的时候，许多类型的对象可能需要清理自己的状态，尽管它们尚未准备好让它们的内存删除。

以一个文件对象为例表示一个打开文件的实例。Windows 系统中文件被打开以供独占访问。当文件对象的最后一个句柄被关闭，重要的是在那一刻就应该删除专有访问，而不是等待任何内核引用最终消失（例如，在最后一次从内存冲洗数据之后。）否则，从用户态关闭并重新打开一个文件可能无法按预期的方式工作，因为该文件看来仍然在使用中。

虽然对象管理器在内核具有全面的管理机制来管理内核中的对象生命周期，不论是 NT API 或 Win32

API的都没有提供一个引用机制来处理在用户态的并行多线程之间的句柄使用。从而多线程并发访问句柄会带来竞争条件 (race condition) 和bug, 例如, 可能发生一个线程在别的线程使用完特定的句柄之前就把它关闭了。或者多次关闭一个句柄。或者关闭另一个线程仍然在使用的句柄, 然后重新打开它指向不同的对象。

也许Windows的API应该被设计为每个类型对象带有一个关闭API, 而不是单一的通用NTClose操作。这将会至少会减少由于用户态线程关闭了错误的处理而发生错误的频率。另一个解决办法可能是在句柄表中的指针之外再添加一个序列域。

为了帮助程序开发人员在他们的程序中寻找这类的问题, Windows有一个应用程序验证, 软件开发商能够从Microsoft下载。我们将在11.7节介绍类似的驱动程序的验证器, 应用程序验证器通过大量的规则检查来帮助程序员寻找可能通过普通测试无法发现的错误。它也可以为句柄释放列表启用先进先出顺序, 以便句柄不会被立即重用 (即关闭句柄表通常采用效果较好的LIFO排序)。防止句柄被立即重用的情况发生, 在这些转化的情况下操作可能错误地使用一个已经关闭的句柄, 这是很容易检测到的。

该设备对象是执行体中一个最重要的和贯穿内核态的对象。该类型是由I/O管理器指定的, I/O管理器和设备驱动是设备对象的主要使用者。设备对象和驱动程序是密切相关的, 每个设备对象通常有一个链接指向一个特定的驱动程序对象, 它描述了如何访问设备驱动程序所对应的I/O处理例程。

设备对象代表硬件设备、接口和总线, 以及逻辑磁盘分区、磁盘卷甚至文件系统、扩展内核, 例如防病毒过滤器。许多设备驱动程序都有给定的名称, 这样就可以访问它们, 而无需打开设备的实例的句柄, 如在UNIX中。我们将利用设备对象以说明Parse程序是如何被使用的, 如图11-22所示。

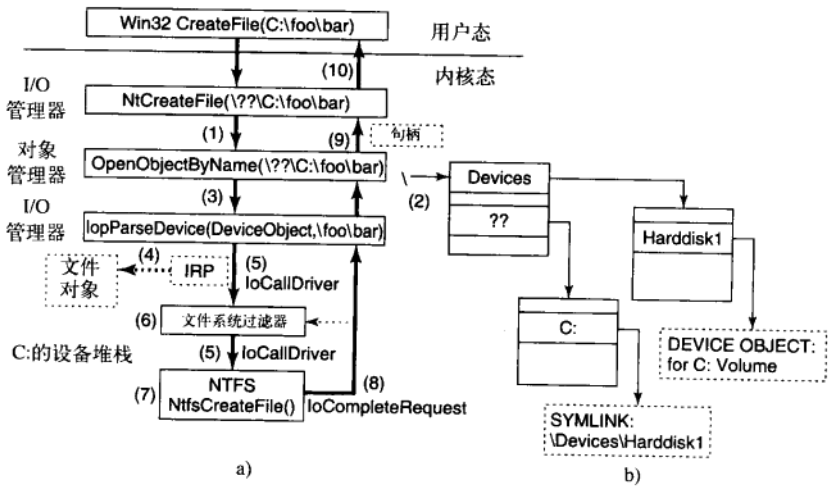


图11-22 I/O和对象管理器创建/打开文件并返回文件句柄的步骤

- 1) 当一个执行组件, 如实现了本地系统调用NtCreateFile的I/O管理器, 在对象管理器中称之为ObOpenObjectByName, 它发送一个NT名字空间的Unicode路径名, 例如\\?\\C:\\foo\\bar。
- 2) 对象管理器通过目录和符号链接表搜索并最终认定 \\?\\C: 指的是设备对象 (I/O管理器定义的一个类型)。该设备对象在由对象管理器管理的NT名字空间中一个叶节点。
- 3) 然后对象管理器为该对象类型调用Parse程序, 这恰好是由I/O管理器实现的IoParseDevice。它不仅传递一个指针给它发现的设备对象 (C:), 而且还把剩下的字符串\\foo\\bar也发送过去。
- 4) I/O管理器将创建一个IRP (I/O请求包), 分配一个文件对象, 发送请求到由对象管理器确定的设备对象发现的I/O设备堆栈。
- 5) IRP是在I/O堆栈中逐级传递, 直到它到达一个代表文件系统C: 实例的设备对象。在每一个阶段, 控制是通过一个与这一等级设备对象相连的切入点传递到驱动对象内部。切入点用在这种情况下, 是为了支持CREATE操作, 因为要求是创建或打开一个名为\\foo\\bar 的文件。

6) 该设备对象中遇到指向文件系统的IRP可以表示为文件系统筛选驱动程序,这可能在操作到达对应的文件系统设备对象之前修改I/O操作。通常情况下这些中间设备代表系统扩展,例如反病毒过滤器。

7) 文件系统设备对象有一个链接到文件系统驱动程序对象,叫NTFS。因此,驱动对象包含NTFS内创建操作的地址范围。

8) NTFS将填补该文件中的对象并将它返回到I/O管理器, I/O管理器备份堆栈中的所有设备,直到lopParseDevice返回对象管理器(如11.8节所述)。

9) 在对象管理器以其名字空间中的查找结束。它从Parse程序收到一个初始化对象(这正好是一个文件对象,而不是原来对象发现的设备对象)。因此,对象管理器为文件对象在目前进程的句柄表里创建了一个句柄,并对需求者返回句柄。

10) 最后一步是返回用户态的调用者,在这个例子里就是Win32 API CreateFile,它会把句柄返回给应用程序。

可执行组件能够通过调用ObCreateObjectType接口给对象管理器来动态创建新的类型。由于每次发布都在变化,所以没有一个限定的对象类型定义表。图11-23列出了在Windows Vista中非常通用的一些对象类型,供快速参考。

类 型	描 述
Process	用户进程
Thread	进程里的线程
Semaphore	进程内部同步的信号量
Mutex	用来控制进入关键区域的二进制信号量
Event	具有持久状态(已标记信号/未标记信号)的同步对象
ALPC Port	内部进程消息传递的机制
Timer	允许一个线程固定时间间隔休眠的对象
Queue	用来完成异步I/O通知的对象
Open file	关联到某个打开的文件的对象
Access token	某个对象的安全描述符
Profile	描述CPU使用情况的数据结构
Section	表述映射的文件的对象
Key	注册表关键字,用于把注册信息关联到某个对象管理名字空间
Object directory	对象管理器中一组对象的目录
Symbolic link	通过路径名引用到另一个对象管理器对象
Device	物理设备、总线、驱动或者卷实例的I/O设备对象
Device driver	每一个加载的设备驱动都有它自己的一个对象

图11-23 对象管理器管理的一些通用可执行对象类型

进程(process)和线程(thread)是明显的。每个进程和每个线程都有一个对象来表示,这个对象包含了管理进程或线程所需的主要属性。接下来的三个对象:信号量、互斥体和事件,都可以处理进程间的同步。信号量和互斥体按预期方式工作,但都需要额外的响铃和警哨(例如,最大值和超时设定)。事件可以在两种状态之一:已标记信号或未标记信号。如果一个线程等待事件处于已标记信号状态,线程被立即释放。如果该事件是未标记信号状态,它会一直阻塞直到一些其他线程信号释放所有被阻止的线程(通知事件)的活动或只是第一个被阻止的线程(同步事件)。也可以设置一个事件,这样一种信号成功等待后,它会自动恢复到该未标记信号的状态而不是处在已标记信号状态。

端口、定时器和队列对象也与通信和同步相关。端口是进程之间交换LPC消息的通道。定时器提供一种为特定的时间区间内阻塞的方法。队列用于通知线程已完成以前启动的异步 I/O 操作,或一个端口有消息等待。(它们被设计来管理应用程序中的并发的水平,以及在使用高性能多处理器应用中使用,如SQL)。

当一个文件被打开时,Open file对象将会被创建。没打开的文件,并没有对象由对象管理器管理。访问令牌是安全的对象。它们识别用户,并指出用户具有什么样的特权,如果有的话。配置文件是线程

的用于存储程序计数器的正在运行的周期样本的数据结构,用以确定程序线程的时间是花在哪些地方了。

段用来表示内存对象,这些内存对象可以被应用程序向内存管理器请求,将应用程序的地址空间映射到这个区域中来。它们记录表示磁盘上的内存对象的页的文件(或页面文件)的段。键表示的是象管理名字空间的注册表名字空间的加载点。通常只有一个名为`REGISTRY`关键对象,负责链接到注册表键值和`NT`名字空间的值。

对象目录和符号链接完全是本地对象管理器的`NT`名字空间的一部分。它们是类似于和它们对应的文件系统部分:目录允许要收集一些相关的对象。符号链接允许对象名字空间来引用一个对象名字空间的不同部分中的对象的一部分的名称。

每个已知的操作系统的设备有一个或多个设备对象包含有关它的信息,并且由系统引用该设备。最后,每个已加载设备驱动程序在对象空间中有一个驱动程序对象。驱动程序对象被所有那些表示被这些驱动控制的设备的实例共享。

其他没有介绍的对象有更多特别的目的,如同内核事务的交互或`Win32`线程池的工作线程工厂交互。

11.3.4 子系统、DLL 和用户态服务

回到图11-6,我们可以看到 Windows Vista 操作系统是由内核态中的组件和用户态的组件组成的。现在我们已经介绍完了我们的内核态组件,因此,我们接下来看看用户态组件。其中对于 Windows 有三种组件尤为重要:环境子系统、DLL 和服务进程。

我们已介绍 Windows 子系统模型,所以这里不作更多详细介绍,而主要是关注原始设计的`NT`,子系统被视为一种利用内核态运行相同底层软件来支持多个操作系统个性化的方法。也许这是试图避免操作系统竞争相同的平台,例如在`DEC`的`VAX`上的`VMS`和`Berkeley UNIX`。或者也许在微软没有人知道`OS/2`是否会成为一个成功的编程接口,他们加上了他们的投注。结果,`OS/2`成为无关的后来者,而`Win32 API`设计为与 Windows 95结合并成为主导。

Windows 用户态设计的第二个重要方面是在动态链接库(DLL),即代码是在程序运行的时候完成的链接,而非编译时。共享的库不是一个新的概念,最现代化的操作系统使用它们。在 Windows 中几乎所有库都是DLL,从每一个进程都装载的系统库`ntdll.dll`到旨在允许应用程序开发人员进行代码通用的功用函数的高层程序库。

DLL通过允许在进程之间共享通用代码来提高系统效率,保持常用代码在内存中,处理减少从程序磁盘到内存中的加载时间。并允许操作系统的库代码进行更新时无需重新编译或重新链接所有使用它的应用程序,从而提高系统的使用能力。

此外,共享的库介绍版本控制的问题,并增加系统的复杂性,因为为帮助某些特定的应用而引入的更改可能会给其他的一些特定的应用带来可能的错误,或者因为实现的改变而破坏了一些其他的应用——这是一个在 Windows 世界称为DLL黑洞的问题。

DLL 的实现在概念上是简单的。并非直接调用相同的可执行映像中的子例程的代码,一定程度的间接性引用被编译器引入: IAT (导入地址表)。当可执行文件被加载时,它查找也必须加载的DLL的列表(这将是一个图结构,因为这些DLL本身会指定它们所需要的其他的DLL列表)。所需的DLL被加载并且填写好它们的IAT。

现实是更复杂的。另一个问题是代表DLL之间的关系图可以包含环,或具有不确定性行为,因此计算要加载的DLL列表可以导致不能运行的结果。此外,在 Windows 中DLL代码库有机会来运行代码,只要它们加载到了进程中或者创建一个新线程。通常,这是使它们可以执行初始化,或为每个线程分配存储空间,但许多DLL在这些附加例程中执行大量的计算。如果任何函数调用的一个附加例程需要检查加载的DLL列表,死锁可能会发生在这个过程。

DLL用于不仅仅共享常见的代码。它们还可以启用一种宿主的扩展应用程序模型。`Internet Explorer`可以下载并链接到 DLL 调用 `ActiveX` 控件。另一端互联网的 Web 服务器也加载动态代码,以为它们所显示的网页产生更好的Web体验。像`Microsoft Office`的应用程序允许链接并运行DLL,使得Office可以类似一个平台来构建新的应用程序。`COM`(组件对象模型)编程模式允许程序动态地查找和加载编写来提供特定发布接口的代码,这就导致几乎所有使用 `COM`的应用程序都以`in-process`的方式来托管 DLL。

所有这类动态加载的代码，为操作系统造成了更大的复杂性，因为程序库的版本管理不是只为可执行体匹配对应版本的 DLL，而是有时把多个版本的同一个 DLL 加载到进程中——Microsoft 称之为肩并肩 (side-by-side)。单个的程序可以承载两个不同的 DLL，每个可能要加载同一个 Windows 库——但对该库的版本有不同要求。

较好的解决方案是把代码放到独立的进程里。而在进程外承载的代码结果具有较低的性能，并在很多情况下会带来一个更复杂的编程模型。微软尚未提供在用户态下来处理这种复杂度的一个好的解决办法。但这让人对相对简单的内核态产生了希望。

该内核态具有较少的复杂性，是因为它相对于用户态提供了更少的对外部设备驱动模型的支持。在 Windows 中，系统功能的扩展是通过编写用户态服务来实现的。这对于子系统运行得很好，并且在只有很少更新的时候，而不是整个系统的个性化的情况下，能够取得更好的性能。在内核实现的服务和在用户态进程实现的服务之间只有很少的功能性差异。内核和过程都提供了专用地址空间，可以保护数据结构和服务请求可以被审议。

但是，可能会与服务的用户态处理内核中服务有重大的性能差异。通过现代的硬件从用户态进入内核是很慢的，但是也比不上要来回切换两次的更慢，因为还需要从内存切换出来进入另一个进程。而且跨进程通信带宽较低。

内核态代码（非常仔细地）可以把用户态处理的数据作为参数传递给其系统调用的方式来访问数据。通过用户态的服务，数据必须被复制到服务进程或由映射内存等提供的一些机制（Windows Vista 中的 ALPC 功能在后台处理）。

将来跨地址空间的切换代价很可能会越来越小，保护模式将会减少，或甚至成为不相关。在 Singularity 中，微软研究院（Fandrich 等人，2006 年）使用运行时技术，类似 C# 和 Java，用来做一个完全软件问题的保护。这就要求地址空间的切换或保护模式下没有硬件的切换代价。

Windows Vista 利用用户态的服务进程极大地提升了系统的性能。其中一些服务是同内核的组件紧密相关的，例如 lsass.exe 这个本地安全身份验证服务，它管理了表示用户身份的令牌 (token) 对象，以及文件系统用来加密的密钥。用户态的即插即用管理器负责确定要使用新的硬件设备所需要的正确的驱动程序来安装它，并告诉内核加载它。系统的很多功能是由第三方提供的，如防病毒程序和数字版权管理，这些功能都是作为内核态驱动程序和用户态服务的组合方式实现的。

在 Windows Vista 中 taskmgr.exe 有一个选项卡，标识在系统上运行的服务。（早期版本的 Windows 将显示服务使用 net start 命令的列表）。很多服务是运行在同一进程 (svchost.exe) 中的。Windows 也利用这种方式来处理自己启动时间的服务，以减少启动系统所需的时间。服务可以合并到相同的进程，只要它们能安全地使用相同的安全凭据。

在每个共享的服务进程内，个体服务是以 DLL 的形式加载的。它们通常利用 Win32 的线程池的功能来共享一个进程池，这样对于所有的服务，只需要运行最小数目的线程。

服务是系统中常见的安全漏洞的来源，因为它们经常是可以远程访问的（取决于 TCP/IP 防火墙和 IP 安全设置），且不是所有程序员都是足够仔细的，他们很可能没有验证通过 RPC 传递的参数和缓冲区。

一直在 Windows 中运行的服务的数目是令人惊讶的。但这些服务中的很少一部分不断收到单个请求，如果有，那这样的进程看起来就像是远程的攻击者试图找到系统的漏洞。结果是越来越多的服务在 Windows 中被默认认为是关闭的，特别是 Windows Server 的相关版本。

11.4 Windows Vista 中的进程和线程

Windows 具有大量的管理 CPU 和资源分组的概念。以下各节中，我们将检查这些有关的 Win32 API 调用的讨论，并介绍它们是如何实现的。

11.4.1 基本概念

在 Windows Vista 中的进程是程序的容器。它们持有的虚拟地址空间，以及指向内核态的对象的线程的句柄。作为线程的容器，它们提供线程执行所需要的公共资源，例如配额结构的指针、共享的令牌对象以及用来初始化线程的默认参数——包括优先次序和调度类。每个进程都有用户态系统数据，称为

PEB (进程环境块)。PEB包括已加载的模块 (如EXE 和 DLL) 列表, 包含环境字符串的内存、当前的工作目录和管理进程堆的数据——以及很多随着时间的推移已添加的Win32 cruft。

线程是在Windows中调度CPU的内核抽象。优先级是基于进程中包含的优先级值来为每个线程分配的。线程也可以通过亲和处理只在某些处理器上运行。这有助于显式分发多处理器上运行的并发程序的工作。每个线程都有两个单独调用堆栈, 一个在用户态执行, 另一个在内核态执行。也有TEB (线程环境块) 使用户态数据指定到线程, 包括每个线程存储区 (线程本地存储区) 和Win32字段、语言和文化本地化以及其他专门的字段, 这些字段都被各种不同的功能添加上了。

除了PEB与TEB外, 还有另一个数据结构, 内核态与每个进程共享的, 即用户共享数据。这个可以由内核写的页, 但是每个用户态进程只能读。它包含了一系列的由内核维护的值, 如各种时间、版本信息、物理内存和大量的被用户态组件共享的标志, 如COM、终端服务和调试程序。有关使用此只读的共享页, 纯粹是出于性能优化的目的, 因为值也能获得通过系统调用到内核态获得。但系统调用是比一个内存访问代价大很多, 所以对于大量由系统维护的字段, 例如时间, 这样的处理就很有意义。其他字段, 如当前时区更改很少, 但依赖于这些字段的代码必须查询它们往往只是看它们是否已更改。

1. 进程

进程创建是从段对象创建的, 每个段对象描述了磁盘上某个文件的一个内存对象。在创建一个过程时创建的进程将接收一个句柄, 这个句柄允许它通过映射段、分配虚拟内存、写参数和环境变量数据、复制文件描述符到它的句柄表、创建线程来修改新的进程。这非常不同于在UNIX中创建进程的, 反映了Windows与UNIX 初始设计目标系统的不同。

正如11.1节所描述, UNIX 是为16 位单处理器系统设计的, 而这样的单处理器系统是用于在进程之间交换共享内存的。这样的系统中, 进程作为并发的单元, 并且使用像fork这样的操作来创建进程是一个天才般的设计主意。如果要在很小的内存中运行一个新的进程, 并且没有硬件支持的虚拟内存, 那么在内存中的进程就不得不换出到磁盘以创建空间。UNIX操作系统 (一种多用户的计算机操作系统) 最初仅仅通过简单的父进程交换技术和传递其物理内存给它的子进程来实现fork。这种操作和运行几乎是没有任何代价的。

相比之下, 在Cutler小组开发NT的时代, 当时的硬件环境是32位多处理器系统与虚拟内存硬件共享1~16兆字节的物理内存。多处理器为部分程序并行运行提供了可能, 因此NT使用进程作为共享内存和数据资源的容器, 并使用线程作为并发调度单元。

当然, 随后几年里的系统就完全不同于这些环境了。例如拥有64位地址空间并且一个芯片上集成十几个 (乃至数百个) CPU内核, 存储体系结构中若干GB大小的物理内存以及闪存设备和其他非易失存储设备的加入, 更广泛虚拟化、普适网络的支持, 以及例如事件型内存 (transactional memory) 这类同步技术的创新。Windows和UNIX操作系统无疑将继续适应现实中新的硬件, 但我们更感兴趣的是, 会有哪些新的操作系统会基于新硬件而被特别设计出来。

2. 作业和纤程

Windows可以将进程分组为作业, 但作业抽象并不足够通用。原因是其专为限制分组进程所包含的线程而设计, 如通过限制共享资源配额、强制执行受限令牌 (restricted token) 来阻止线程访问许多系统对象。作业最重要的特性是一旦一个进程在作业中, 该进程创建的进程、线程也在该作业中, 没有特例。就像它的名字所示, 作业是为类似批处理环境而非交互式计算环境而设计的。

一个进程最多属于一个作业。这是有道理的, 因为很难去定义一个进程必须服从多个共享配额或限制令牌的情况。但这也意味着, 如果有多个系统服务尝试使用作业来管理同一部分进程, 则会产生冲突。例如, 如果进程首先将自己加入到了一个作业中, 或者一个安全的工具已经将其加入了带有一定受限令牌的作业中, 则当一个管理工具试图将进程加入其他作业以限制其资源时将会失败。因此在Windows中很少使用作业。

图11-24显示了作业、进程、线程和纤程之间的关系。作业包含进程, 进程包含线程, 但是线程不包含纤程。线程与纤程通常是多对多的关系。

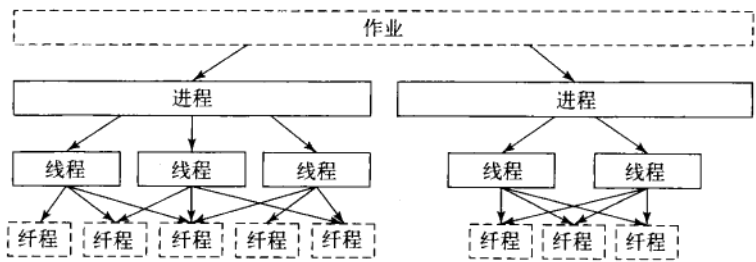


图11-24 作业、进程、线程、纤程之间的关系。作业和纤程是可选的，并不是所有的进程都在作业中或者包含纤程

纤程通过分配栈与用来存储纤程相关寄存器和数据的用户态纤程数据结构来创建。线程被转换为纤程，但纤程也可以独立于线程创建。这些新创建的纤程直到一个已经运行的纤程显式地调用 SwitchToFiber 函数才开始执行。由于线程可以尝试切换到一个已经在运行的纤程，因此，程序员必须使用同步机制以防止这种情况发生。

纤程的主要优点在于纤程之间的切换开销要远远小于线程之间的切换。线程切换需要进出内核而纤程切换仅需要保存和恢复几个寄存器。

尽管纤程是协同调度的，如果有多个线程调度纤程，则需要非常小心地通过同步机制以确保纤程之间不会互相干扰。为了简化线程和纤程之间的交互，通常创建和能运行它们的内核数目一样多的线程，并且让每个线程只能运行在一套可用的处理器甚至只是一个单一的处理器上。

每个线程可以运行一个独立的纤程子集，从而建立起线程和纤程之间一对多的关系来简化同步。即便如此，使用纤程仍然有许多困难。大多数的 Win32 库是完全不识别纤程的，并且尝试像使用线程一样使用纤程的应用会遇到各种错误。由于内核不识别纤程，当一个纤程进入内核时，其所属线程可能阻塞。此时处理器会调度任意其他线程，导致该线程的其他纤程均无法运行。因此纤程很少使用，除非从其他系统移植那些明显需要纤程提供功能的代码。图11-25总结了上面提到的这些抽象。

名称	描述	注释
作业	一组共享时间配额和限制的进程	很少使用
进程	持有资源的容器	
线程	内核调度的实体	
纤程	在用户空间管理的轻量级线程	很少使用

图11-25 CPU和资源管理所使用的基本概念

3. 线程

通常每一个进程是由一个线程开始的，但一个新的进程也可以动态创建。线程是CPU调度的基本单位，因为操作系统总是选择一个线程而不是进程来运行。因此，每一个线程有一个调度状态（就绪态、运行态、阻塞态等），而进程没有调度状态。线程可以通过调用指定了在其所属进程地址空间中的开始运行地址的 Win32 库函数动态创建。

每一个线程均有一个线程ID，其和进程ID取自同一空间，因此单一的ID不可能同时被一个线程和一个进程使用。进程和线程的ID是4的倍数，因为它们实际上是通过用于分配ID的特殊句柄表来执行分配的。该系统复用了如图11-18和图11-19所示的可扩展句柄管理功能。句柄表没有对象的引用，但使用指针指向进程或线程，使通过ID查找一个进程或线程非常有效。最新版本的 Windows 采用先进先出顺序管理空闲句柄列表，使ID无法马上重复使用。ID马上被重复使用的问题将在本章的最后问题部分再讨论。

线程通常在用户态运行，但是当它进行一个系统调用时，就切换到内核态，并以其在用户态下相同的属性以及限制继续运行。每个线程有两个堆栈，一个在用户态使用，而另一个在内核态使用。任何时候当一个线程进入内核态，其切换到内核态堆栈。用户态寄存器的值以上下文（context）数据结构的形式保存在该内核态堆栈底部。因为只有进入内核态的用户态线程才会停止运行，当它没有运行时该上下文数据结构中总是包括了其寄存器状态。任何拥有线程句柄的进程可以查看并修改这个上下文数据结构。

线程通常使用其所属进程的访问令牌运行，但在某些涉及客户机/服务器计算的情况下，一个服务器线程可能需要模拟其客户端，此时需要使用基于客户端令牌的临时令牌标识来执行客户的操作。（一

般来说服务器不能使用客户端的实际令牌,因为客户端和服务器的可运行于不同的系统。)

I/O处理也经常需要关注线程。当执行同步I/O时会阻塞线程,并且异步I/O相关的未完成的I/O请求也关联到线程。当一个线程完成执行,它可以退出,此时任何等待该线程的I/O请求将被取消。当进程中最后一个活跃线程退出时,这一进程将终止。

需要注意的是线程是一个调度的概念,而不是一个资源所有权的概念。任何线程可以访问其所属进程的所有对象,只需要使用句柄值,并进行合适的Win32调用。一个线程并不会因为一个不同的线程创建或打开了一个对象而无法访问它。系统甚至没有记录是哪一个线程创建了哪一个对象。一旦一个对象句柄已经在进程句柄表中,任何在这一进程中的线程均可使用它,即使它是在模拟另一个不同的用户。

正如前面所述,除了用户态运行的正常线程,Windows有许多只能运行在内核态的系统线程,而其与任何用户态进程都没有联系。所有这一类型的系统线程运行在一个特殊的称为系统进程的进程中。该进程没有用户态地址空间,其提供了线程在不代表某一特定用户态进程执行时的环境。当学到内存管理的时候,我们将讨论这样的一些线程。这些线程有的执行管理任务,例如写脏页面到磁盘上,而其他形成了工作线程池,来分配并执行部件或驱动程序需要系统进程执行的工作。

11.4.2 作业、进程、线程和线程管理API调用

新的进程是由Win32 API函数CreateProcess创建的。这个函数有许多参数和大量的选项,包括被执行文件的名称,命令行字符串(未解析)和一个指向环境字符串的指针。其中也包括了控制诸多细节的令牌和数值,这些细节包括了如何配置进程和第一个线程的安全性,调试配置和调度优先级等。其中一个令牌指定创建者打开的句柄是否被传递到新的进程中。该函数还接受当前新进程的工作目录和可选的带有关于此进程使用GUI窗口的相关信息的数据结构。Win32对新进程和其原始线程都返回ID和句柄,而非只为新进程返回一个ID号。

大量的参数揭示了Windows和UNIX在进程创建的开发设计上的诸多的不同之处。

1) 寻找执行程序的实际搜索路径隐藏在Win32的库代码里,但UNIX中则显式地管理该信息。

2) 当前工作目录在UNIX操作系统里是一个内核态的概念,但是在Windows里是用户态字符串。Windows为每个进程都打开当前目录的一个句柄,这导致了和UNIX一样的麻烦:除了碰巧工作目录是跨网络的情况下可以删除它,其他工作目录都是不能删除的。

3) UNIX解析命令行,并传递参数数组;而Win32需要每个程序自己解析参数。其结果是,不同的程序可能采用不一致的方式处理通配符(如*.txt)和其他特殊字符。

4) 在UNIX中,文件描述符是否可以被继承是句柄的一个属性。不过在Windows中,其同时是句柄和进程创建参数的属性。

5) Win32是面向图形用户界面的,因此新进程能直接获得其窗口信息,而在UNIX中,这些信息是通过参数传递给图形用户界面程序的。

6) Windows中的可执行代码没有SETUID位属性,不过一个进程也可以为另一个用户创建进程,只要其能获得该用户的信用标识。

7) Windows返回的进程、线程句柄可以用在很多独立的方法中修改新进程/线程,例如复制句柄、在新进程中设置环境变量等。UNIX则只在fork和exec调用时修改新进程。

这些不同有些是来自历史原因和哲学原因。UNIX的设计是面向命令行的,而不是像Windows那样面向图形用户界面的。UNIX的用户相比来说更高级,同时也懂得像PATH环境变量的概念。Windows Vista继承了很多MS-DOS中的东西。

这种比较也有点偏颇,因为Win32是一个用户态下的对NT本地进程执行的包装器,就像UNIX下的系统库函数fork/exec的封装。实际的NT中创建进程和线程的系统调用NtCreateProcess和NtCreateThread比Win32版本简单得多。NT进程创建的主要参数包括代表所要运行的程序文件句柄、一个指定新进程是否默认继承创建者句柄的标志,以及有关安全模型的相关参数。由于用户态下的代码能够使用新建进程的句柄能对新进程的虚拟地址空间进行直接的操作,所有关于建立环境变量、创建初始线程的细节就留给用户态代码来解决。

为了支持POSIX子系统,本地进程创建有一个选项可以指定,通过拷贝另一个进程的虚拟地址空间

来创建一个新进程，而不是通过映射一个新程序的段对象来新建进程。这种方式只用在实现POSIX的fork，而不是Win32的。

线程创建时传给新线程的参数包括：CPU的上下文信息（包括栈指针和起始指令地址）、TEB模板、一个表示线程创建后马上运行或以挂起状态创建（等待有人对线程句柄调用NtResumeThread函数）的标志。用户态下的栈的创建以及argv/argc参数的压入需要由用户态下的代码来解决，必须对进程句柄调用本地NT的内存管理API。

在Windows Vista的发行版中，包含了一个新的关于进程操作方面的本地API，这个接口将原来许多用户态下的步骤转移到了内核态下执行，同时将进程创建与起始线程创建绑定在一起进行。作这种改变的原因是支持通过进程划分信任边界。一般来说，所有用户创建的进程被同等信任，由用户决定信任边界在哪里。在Windows Vista中的这个改变，允许进程也可以提供信任边界，但是这意味着对于新进程句柄来说，创建者进程没有足够的权利在用户态下实现进程创建的细节。

1. 进程间通信

线程间可以通过多种方式进行通信，包括管道、命名管道、邮件槽、套接字、远程过程调用（RPC）、共享文件等。管道有两种模式：字节管道和消息管道，可以在创建的时候选择。字节模式的管道的工作方式与UNIX下的工作方式一样。消息模式的管道与字节模式的管道大致相同，但会维护消息边界。所以写入四次的128字节，读出来也是四个128字节的消息，而不会像字节模式的管道一样读出的是一个512字节的消息。命名管道在Vista中也是有的，跟普通的管道一样都有两种模式，但命名管道可以在网络中使用，而普通管道只能在单机中使用。

邮件槽是OS/2操作系统的特性，在Windows中实现只是为了兼容性。它们在某种方式上跟管道类似，但不完全相同。首先，它们是单向的，而管道则是双向的。而且，它们能够在网络中使用但不提供有保证的传输。最后，它们允许发送进程将消息广播给多个接收者而不仅仅是一个接收者。邮件槽和命名管道在Windows中都是以文件系统的形式实现，而非可执行的功能函数。这样做就可以通过现有的远程文件系统协议在网络上访问到它们。

套接字也与管道类似，只不过它们通常连接的是不同机器上的两个进程。例如，一个进程往一个套接字里面写入内容，远程机器上的另外一个进程从这个套接字中读出来。套接字同样也可以被用在同一台机器上的进程通信，但是因为它们比管道带来了更大的开销，所以一般来说它们只被用于网络环境下的通信。套接字原来是为伯克利UNIX而设计的，它的实现代码很多都是可用的，正如Windows发布日志里面所写的，Windows代码中使用了一些伯克利的代码及数据结构。

远程过程调用（RPC）是一种进程A命令进程B调用进程B地址空间中的一个函数，然后将执行结果返回给进程A的方式。在这个过程中对参数的限制很多。例如，如果传递的是个指针，那么对于进程B来说这个指针毫无意义，因此必须把数据结构打包起来然后以进程无关的方式传输。实现RPC的时候，通常是把它作为传输层之上的抽象层来实现。例如对于Windows来说，可以通过TCP/IP套接字、命名管道、ALPC来进行传输。ALPC的全称是高级本地过程调用（Advanced Local Procedure Call），它是内核态下的一种消息传递机制，为同一台机器中的进程间通信作了优化，但不支持网络间通信。基本的设计思想是可以发送有回复的消息，以此来实现一个轻量级的RPC版本，提供比ALPC更丰富的特性。ALPC的实现是通过拷贝参数以及基于消息大小的临时共享内存分配。

最后，进程间可以共享对象，如段对象。段对象可以同时被映射到多个进程的虚拟地址空间中，一个进程执行了写操作之后，其他进程可以也可以看见这个写操作。通过这个机制，在生产者消费者问题中用到的共享缓冲区就可以轻松地实现。

2. 同步

进程间也可以使用多种形式的同步对象。就像Windows Vista中提供了多种形式的进程间通信机制一样，Vista也提供了多种形式的同步机制，包括信号量、互斥量、临界区和事件。所有的这些机制只在线程上工作，而非进程。所以当一条线程由于一个信号量而阻塞时，同一个进程的其他线程（如果有的话）会继续运行而并不会被影响。

使用Win32的API函数CreateSemaphore可以创建一个信号量，可以将它初始化为一个给定的值，

同时也可以指定最大值。信号量是一个内核态对象，因此拥有安全描述符和句柄。信号量的句柄可以通过使用DuplicateHandler来进行复制，然后传递给其他进程使得多个进程可以通过相同的信号量来进行同步。在Win32的名字空间中一个信号量也可以被命名，可以拥有一个ACL集合来保护它。有些时候通过名字来共享信号量比通过拷贝句柄更合适。

对up和down的调用也是有的，只不过它们的函数名看起来比较奇怪：ReleaseSemaphore (up)和WaitForSingleObject (down)。可以给WaitForSingleObject一个超时时间，使得尽管此时信号量仍然是0，调用它的线程仍然可以被释放（尽管定时器重新引入了竞态）。WaitForSingleObject和WaitForMultipleObject是将在11.3节中讨论的分发者对象的常见接口。尽管有可能将单个对象的API封装成看起来更加像信号量的名字，但是许多线程使用多个对象的版本，这些对象可能是各种各样的同步对象，也可能是其他类似进程或线程结束、I/O结束、消息到达套接字和端口等事件。

互斥量也是用于同步的内核态对象，但是比信号量简单，因为互斥量不需要计数器。它们其实是锁，上锁的函数是WaitForSingleObject，解锁的函数是ReleaseMutex。就像信号量句柄一样，互斥量的句柄也可以复制，并且在进程间传递，从而不同进程间的线程可以访问同一个互斥量。

第三种同步机制是临界区，实现的是临界区的概念。临界区在Windows中与互斥量类似，但是临界区相对于主创建线程的地址空间来说是本地的。因为临界区不是内核态的对象，所以它们没有显式的句柄或安全描述符，而且也不能在进程间传递。上锁和解锁的函数分别是EnterCriticalSection和LeaveCriticalSection。因为这些API函数在开始的时候只是在用户空间中，只有当需要阻塞的时候才调用内核函数，它们比互斥量快得多。在需要的时候，可以通过合并自旋锁（在多处理器上）和内核同步机制来优化临界区。在许多应用中，大多数的临界区几乎不会被竞争或者只被锁住很短的时间，以至于没必要分配一个内核同步对象，这样会极大地节省内核内存。

我们讨论的最后一种同步机制叫事件，它使用内核态对象。就像我们前面描述的，有两类的事件——通知事件和同步事件。一个事件的状态有两种：收到信号和没收到信号。一个线程通过调用WaitForSingleObject来等待一个事件被信号通知。如果另一个线程通过SetEvent给事件发信号，会发生什么取决于这个事件的类型。对于通知事件来说，所有等待线程都会被释放，并且事件保持在set状态，直到手工调用ResetEvent进行清除；对于同步事件来说，如果有一个或多个线程在等待，那么有且仅有一个线程会被唤醒并且事件被清除。另一个替换的操作是PulseEvent，像SetEvent一样，除了在没有人等待的时候脉冲会丢失，而事件也被清除。相反，如果调用SetEvent时没有等待的线程，那么这个设置动作依然会起作用，被设置的事件处于被信号通知的状态，所以当后面的那个线程调用等待事件API时，这个线程将不会等待而直接返回。

Win32的API中关于进程、线程、纤程的个数将近100个，其中大量的各种形式的处理IPC的函数。对上面讨论的总结和另一些比较重要的内容可以参见图11-26。

可以注意到不是所有的这些都是系统调用。其中有一些是包装器，有一些包含了重要的库代码，这些库代码将Win32的接口映射到本地NT接口。另外一些，例如纤程的API，全部都是用户态下的函数，因为就像我们之前提到的，Windows Vista的内核态中根本没有纤程的概念，纤程完全都是由用户态下的库来实

Win32 API 函数	描 述
CreateProcess	创建一个新的进程
CreateThread	在已存在的进程中创建一个新的线程
CreateFiber	创建一个新的纤程
ExitProcess	终止当前进程及其全部线程
ExitThread	终止该线程
ExitFiber	终止该纤程
SwichToFiber	在当前线程中运行另一纤程
SetPriorityClass	设置进程的优先级类
SetThreadPriority	设置线程的优先级
CreateSemaphore	创建一个新的信号量
CreateMutex	创建一个新的互斥量
OpenSemaphore	打开一个现有的信号量
OpenMutex	打开一个现有的互斥量
WaitForSingleObject	等待一个单一的信号量、互斥量等
WaitForMultipleObjects	等待一系列已有句柄的对象
PulseEvent	设置事件激活，再变成未激活
ReleaseMutex	释放互斥量使其他线程可以获得它
ReleaseSemaphore	使信号量增加1
EnterCriticalSection	得到临界区的锁
leaveCriticalSection	释放临界区的锁

图11-26 一些管理进程、线程以及纤程的一些Win32调用

现

现的。

11.4.3 进程和线程的实现

本节将用更多细节来讲述Windows如何创建一个进程。因为Win32是最具文档化的接口，因此我们将从这里开始讲述。我们迅速进入内核来理解创建一个新进程的本地API调用是如何实现的。这里有很多细节我们都将略过，比如在创建一个路径的时候，WOW16 和WOW64 有怎样专用的代码，以及系统如何提供特定应用的修补来修正应用程序中的小的不兼容性和延迟错误。我们主要集中在创建进程时执行的主代码路径，以及看一看我们已经介绍的知识之间还欠缺的一些细节。

当用一个进程调用Win32 `CreateProcess`系统调用的时候，则创建一个新的进程。这种调用使用`kernel32.dll`中的一个（用户态）进程来分几步创建新进程，其中会使用多次系统调用和执行其他的一些操作。

1) 把可执行的文件名从一个Win32 路径名转化为一个NT 路径名。如果这个可执行文件仅有一个名字，而没有一个目录名，那么就在默认的目录里面查找（包括，但不限于，那些在PATH环境变量中的）。

2) 绑定这个创建过程的参数，并且把它们和可执行程序完全路径名传递给本地API `NtCreateUserProcess`。（这个API 被增加到Window Vista使得创建进程的细节可以在内核态里处理，从而让进程可以在可信的边界内使用。之前介绍的那些API仍然是存在的，只是不再被Win32的`CreateProcess`调用使用。）

3) 在内核态里运行，`NtCreateUserProcess`执行参数，然后打开这个进程的映像，创建一个内存区对象（section object），它能够用来把程序映射到新进程的虚拟地址空间。

4) 进程管理器分配和初始化进程对象。（对于内核和执行层，这个内核数据结构就表示一个进程。）

5) 内存管理器通过分配和创建页目录及虚拟地址描述符来为新进程创建地址空间。虚拟地址描述符描述内核态部分，包括特定进程的区域，例如自映射的页目录入口可以为每一个进程在内核态使用内核虚拟地址来访问它整个页表中的物理页面。

6) 一个句柄表为新的进程所创建。所有来自于调用者并允许被继承的句柄都被复制到这个句柄表中。

7) 共享的用户页被映射，并且内存管理器初始化一个工作集的数据结构，这个数据结构是在物理内存缺少的时候用来决定哪些页可以从一个进程里面移出。可执行映像中由内存区对象表示的部分会被映射到新进程的用户态地址空间。

8) 执行体创建和初始化用户态的进程环境块（PEB），这个PEB为用户态和内核用来维护进程范围的状态信息，例如用户态的堆指针和可加载库列表（DLL）。

9) 虚拟内存是分配在（ID表）新进程里面的，并且用于传递参数，包括环境变量和命令行。

10) 一个进程ID从特殊的句柄表（ID表）分配，这个句柄表是为了有效地定位进程和线程局部唯一的ID。

11) 一个线程对象被分配和初始化。在分配线程环境块（TEB）的同时，也分配一个用户态栈。包含了线程的为CPU寄存器保持的初始值（包括指令和栈指针）的CONTEXT记录也被初始化了。

12) 进程对象被放入进程全局列表中。进程和线程对象的句柄被分配到调用者的句柄表中。ID表会为初始线程分配一个ID。

13) `NtCreateUserProcess`向用户态返回新建的进程，其中包括处于就绪并被挂起的单一线程。

14) 如果NT API失败，Win32代码会查看进程是否属于另一子系统，如WOW64。或者程序可能设置为在调试状态下运行。以上特殊情况由用户态的`CreateProcess`代码处理。

15) 如果`NtCreateUserProcess`成功，还有一些操作要完成。Win32 进程必须向Win32子系统进程`csrss.exe`注册。`Kernel32.dll`向`csrss.exe`发送信息——新的进程及其句柄和线程句柄，从而进程可以自我复制了。进程和线程加入子系统列表中，从而它们拥有了所有Win32的进程和线程的完整列表。子系统此时就显示一个的带沙漏光标表明系统正运行，但光标还能使用。当进程首次调用GUI函数，通常是创建新窗口，光标将消失（如果没有调用到来，2秒后就会超时）。

16) 如果进程受限，如低权限的Internet Explorer，令牌会被改变，限制新进程访问对象。

17) 如果应用程序被设置成需要与当前Windows版本加垫层（shim）地兼容运行，则特定的垫层将运行（垫层通常封装库调用以稍微修改它们的行为，例如返回一个假的版本号或者延迟内存的释放）。

18) 最后, 调用NtResumeThread挂起线程, 并把这个结构返回给包含所创建的进程和线程的ID、句柄的调用者。

调度

Windows内核没有任何中央调度线程。所以, 当一个线程不能够再执行时, 线程将进入内核态, 调度线程再决定转向的下一个线程。在下面这些情况下, 当前正在执行的线程会执行调度程序代码:

1. 当前执行的线程发生了信号量、互斥、事件、I/O等类型的阻塞。
2. 线程向一个对象发信号 (如发一个信号或者是唤醒一个事件) 时。
3. 配额过期。

第一种情况, 线程已经在内核态运行并开始对调度器或输入输出对象执行操作了。它将不能继续执行, 所以线程会请求调度程序代码寻找装载下一个线程的CONTEXT记录去恢复其执行。

第二种情况, 线程也是在内核中运行。但是, 在向一些对象发出信号后, 它肯定还能够继续执行, 因为发信号对象从来没有受到阻塞。然而, 线程必须请求调度程序, 来观测它的执行结果是否释放了一个具有更高调度优先级的正准备运行的线程。如果是这样, 因为Windows完全是可抢占式的, 所以就会发生一个线程切换 (例如, 线程切换可以发生在任何时候, 不仅仅是在当前线程结束时)。但是, 在多处处理器的情况下, 处于就绪状态的线程会在另一个CPU上被调度, 那么, 即使原来线程拥有较低的调度优先级, 也能在当前的CPU上继续执行。

第三种情况, 内核态发生中断, 这时线程执行调度程序代码找到下一个运行的线程。由于取决于其他等待的线程, 可能会选择同样的线程, 这样线程就会获得新的配额, 可以继续执行。否则发生线程切换。

在另外两种情况下, 调度程序也会被调度:

- 1) 一个输入输出操作完成时。
- 2) 等待时间结束时。

在第一种情况下, 线程可能处于等待输入输出时被释放然后执行。如果不保证最小执行时间, 必须检查是否可以事先对运行的线程进行抢占。调度程序不会在中断处理程序中运行 (因为那使中断关闭保持太久)。相反, 中断处理发生后, DPC会排队等待一会儿。第二种情况下, 线程已经对一个信号量进行了down操作或者因一些其他对象而被阻塞, 但是定时器已经过期。对于中断处理程序来说, 有必要让DPC再一次排队等待, 以防止它在定时器中断处理程序时运行。

如果一个线程在这个时刻已到就绪, 则调度程序将会被唤醒并且如果新的可运行线程有较高的优先级, 那么和情形1的情况类似, 当前的线程会被抢占。

现在让我们来看看具体的调度算法。Win32 API提供两个API来影响线程调度。首先, 有一个叫SetPriorityClass的用来设定被调用进程中所有线程的优先级。其等级可以是: 实时、高、高于标准、标准、低于标准和空闲的。优先级决定进程的先后顺序。(在Vista系统中, 进程优先级等级也可以被一个进程用来临时地把它自己标记为后台运行(background)状态, 即它不应该被任何其他的活动进程所干扰。)注意优先级是对进程而言的, 但是实际上会在每个线程被创建的时候通过设置每个线程开始运行的基本优先级可以影响进程中每条线程的实际优先级。

第二个就是SetThreadPriority。它根据进程的优先级类来设定进程中每个线程的相对优先级 (可能地, 但是不必然地, 调用线程)。可划分如下等级: 紧要的、最高的、高于标准的、标准的、低于标准的、最低的和休眠的。时间紧急的线程得到最高的非即时的调度优先, 而空闲的线程不管其优先级类别都得到最低的优先级。其他优先级的值依据优先级的等级来定, 依次为 (+2, +1, 0, -1, -2)。进程优先级等级和相对线程优先级的使用使得能够更容易地确定应用程序的优先级。

调度程序按照下列方式进行调度。系统有32个优先级, 从0到31。依照图11-27的表格, 进程优先级和相对线程优先级的组合形成32个绝对线程优先级。在表格的数字决定了线程的基本优先级 (base priority)。除此之外, 每条线程都有当前优先级 (current priority), 这个当前的优先级可能会高于 (但是不低于) 前面提到的基本优先级, 关于这一点我们稍后将会讨论。

		Win32进程类优先级					
Win32线程优先级		实时	高	高于标准	标准	低于标准	空闲
	时间紧急	31	15	15	15	15	15
	最高	26	15	12	10	8	6
	高于标准	25	14	11	9	7	5
	标准	24	13	10	8	6	4
	低于标准	23	12	9	7	5	3
	最低	22	11	8	6	4	2
	空闲	16	1	1	1	1	1

图11-27 Win32优先级到Windows优先级的映射

为了使用这些优先级进行调度，系统维护一个包含32个线程列表的队列，分别对应图11-27中的0~31的不同等级。每个列表包含了就绪线程对应的优先级。基本的调度算法是从优先级队列中从31到0的从高优先级到低优先级的顺序查找。一旦一个非空的列表被找到，等待队首的线程就运行一个时间片。如果时间配额已用完，这个线程排到其优先级的队尾，而排在前面的线程就接下来运行。换句话说，当在最高的优先级有多条线程处于就绪状态，它们就按时间片轮转法来调度。如果没有就绪的线程，那么处理器空闲，并设置成低功耗状态来等待中断的发生。

值得注意的是，调度取决于线程而不是取决于线程所属的进程。因此调度程序并不是首先查看进程然后再是进程中的线程。它直接找到线程。调度程序并不考虑哪个线程属于哪个进程，除非进行线程切换时需要做地址空间的转换。

为了改进在具有大量处理器的多处理器情况下的调度算法的可伸缩性，调度管理器尽力不给全局的优先级表的数组加上一个全局的锁来实现同步访问控制。相反地，对于一个准备到CPU的线程来说，若是处理器已就位，则可以让它直接进行，而不必进行加锁操作。

对于每一个进程，调度管理器都维护了一个理想处理器（ideal processor）记录，它会在尽可能的时候让线程在这个理想处理器上运行。这改善了系统的性能，因为线程所用到的数据驻留在理想处理器的内存中。调度管理器可以感知多处理器的环境，并且每一个处理器有自己的内存，可以运行需要任意大小内存空间的程序——但是如果内存不在本地，则会花费较大的时间开销。这些系统被认为是NUMA（非统一内存地址）设备。调度管理器努力优化线程在这类计算机上的分配。当线程出现缺页错误时，内存管理器努力把属于理想处理器的NUMA节点的物理页面分配给线程。

队首的队列在图11-28中表示。这个图表明实际上有四类优先级：实时级、用户级、零页和空闲级，即当它为-1时有效。这些值得我们深入讨论。优先级16~31属于实时级的一类，用来为构建满足实时性约束的系统。处于实时级的线程优先于任何动态分配级别的线程，但是不先于DPC和ISR。如果一个实时级的应用程序想要在系统上运行，它就要求设备驱动不能运行DPC和ISR更多的额外时间，因为这样可能导致这些实时线程错过它们的截止时间。

用户态下不能运行实时级的线程。如果一个用户级线程在一个高优先级运行，比如说，键盘或者鼠标线程进入了一个死循环，键盘或者鼠标永远得不到运行从而系统被有效地挂起。把优先级设置为实时级的权限，需要启用进程令牌中相应的特权。通常用户没有这个特权。

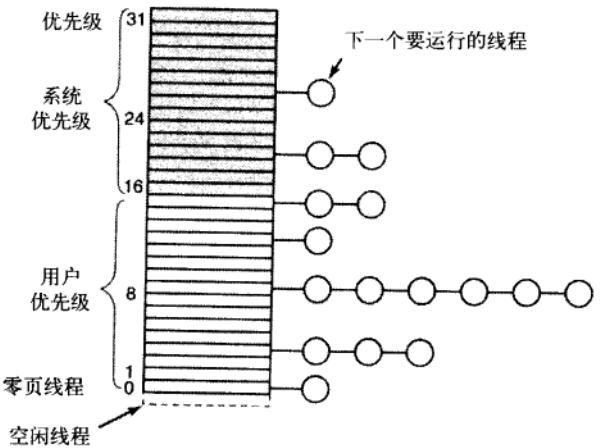


图11-28 Windows Vista为线程支持32个优先级

如果用户态下不能运行实时级的线程。如果一个用户级线程在一个高优先级运行，比如说，键盘或者鼠标线程进入了一个死循环，键盘或者鼠标永远得不到运行从而系统被有效地挂起。把优先级设置为实时级的权限，需要启用进程令牌中相应的特权。通常用户没有这个特权。

应用程序的线程通常在优先级1~15上运行。通过设定进程和线程的优先级,一个应用程序可以决定哪些线程得到偏爱(获得更高优先级)。ZeroPage系统线程运行在优先级0并且把所有要释放的页转化为全部包含0的页。每一个实时的处理器都有一个独立的ZeroPage线程。

每个线程都有一个基于进程优先级的基本优先级和一个线程自己的相对优先级。用于决定一个线程在32个列表中的哪一个列表进行排队的优先级取决于当前优先级,通常是得到和当前线程的基本优先级一样的优先级,但并不总是这样。在特定的情况下,非实时线程的当前优先级被内核一下子提到尽可能高的优先级(但是不会超过优先级15)。因为图11-28的排列以当前的优先级为基础,所以改变优先级可以影响调度。对于实时优先级的线程,没有任何的调整。

现在让我们看看一个线程在什么样的时机得到提升。首先,当输入输出操作完成并且唤醒一个等待线程的时候,优先级一下子被提高,给它一个快速运行的机会,这样可以使更多的I/O可以得到处理。这里保证I/O设备处于忙碌的运行状态。提升的幅度依赖于输入输出设备,典型地磁盘片对应于1级,串行总线对应于2级,6级对应于键盘,8级对应于声卡。

其次,如果一个线程在等待信号量,互斥量同步或其他的事件,当这些条件满足线程被唤醒的时候,如果它是前台的进程(该进程控制键盘输入发送到的窗口)的话,这个线程就会得到两个优先级的提升,其他情况则只提升一个优先级。这倾向于把交互式的进程优先级提升到8级以上。最后,如果一个窗口输入就绪使得图形用户接口线程被唤醒,它的优先级同样会得到大幅提升。

提升不是永远的。优先级的提升是立刻发生作用的,并且会引起处理器的再次调度。但是如果一个线程用完它的时间分配量,它就会降低一个优先级而且排在新优先级队列的队尾。如果它两次用完一个完整的时间配额,它就会再降一个优先级,如此下去直到降到它的基本优先级,在基本优先级得到保持不会再降,直到它的优先级再次得到提升。

还有一种情况就是系统变动(fiddle)优先级。假设有二个线程正在一个生产者-消费者类型问题上一起协同工作。生产者的工作需要更多的资源,因此,它得到高的优先级,例如说12,而消费者得到的优先级为4。在特定的时刻,生产者已经把共享的缓冲区填满,信号量发生阻塞,如图11-29a所示。

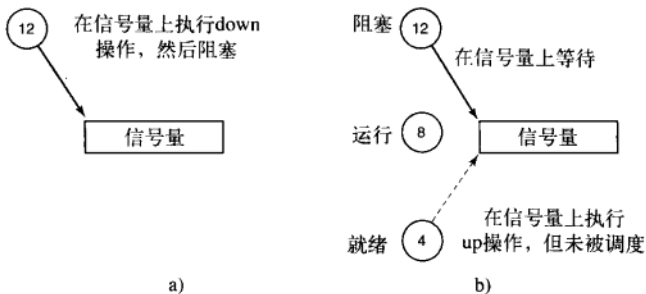


图11-29 优先级转置的示例

如图11-29b所示,在消费者得到调度再次运行之前,一个无关的线程在优先级8已就绪得到调度运行。只要这个线程想要运行,它将会一直运行,因为这个线程的优先级高于消费者的优先级,而比它优先级高的生产者由于阻塞也不能够运行。在这种情况下,直到优先级为8的线程运行完毕,生产者才有机会再次运行。

Windows通过一个称为大hack来解决此类问题的。系统记录一个已就绪的线程自从上次得到运行后距离当前的时间有多久。如果它超过一个特定的阈值,它就被提升到15级的优先级并得到两个时间配额的运转。这就可能解决生产者阻塞的情况。在两个时间配额用完之后,它的优先级一下子又回到原来的优先级而不是逐级地缓慢下降到原来的优先级。或许较好的解决方法是把那些用完时间配额的线程的优先级不断地降低。毕竟,问题不是由饥饿的线程所引起的,而是由贪婪线程造成的。这一问题广为人知地称作优先级倒转(priority inversion)。

在优先为16条线程获得互斥量却长时间得不到调度的时候会发生一个类似的问题,致使更重要的系统线程由于等待互斥量而不能运行发生饥饿。这一问题在操作系统里通过在那些只需要短时间拥有互斥

量的线程在很忙时禁用调度来解决。(在一个多处理器上, 一个Spin锁应被使用。)

在离开调度的主题之前, 关于时间配额值得再讨论一下。在Windows客户端系统上, 默认值是20毫秒。在Windows服务器系统上, 它是180毫秒。短的时间配额在交互性上会更好些, 然而长的时间配额能减少切换提高效率。如果需要, 时间配额可以手动地设置成默认值的2倍、4倍或6倍。

最后对调度算法来说, 当新窗口变成前台窗口的时候, 它的全部在窗口中注册的线程都会得到一个较长的时间配额。这一个变化给它们较多的处理器时间, 从而为这些窗口刚刚转移到前台的应用程序带来了更好的用户体验。

11.5 内存管理

Windows Vista有一个极端复杂的虚拟内存系统。这一系统包括了大量Win32函数, 这些函数通过内存管理器(NTOS执行层最大的组件)来实现。在下面章节中, 我们将依次了解它的基本概念、Win32的API调用以及它的实现。

11.5.1 基本概念

在Windows Vista系统中, 每个用户进程都有它自己的虚拟地址空间。对于x86机器, 虚拟地址是32位的; 因此, 每个进程拥有4GB大小的虚拟地址空间。其中用户态进程的虚拟地址大小为2GB(在服务器系统中, 用户态进程的虚拟地址大小可以配置成3GB)。另外的2GB(或1GB)空间为内核进程所用。对于运行在64位上的x64机器而言, 地址可以是32位的也可以是64位的。32位地址是为了应用那些“需要通过WOW64来运行在64位系统上的32位进程”而保留的。由于内核拥有大量可用的地址空间, 如果需要的话, 32位进程可以使用全部4GB大小的地址空间。对于x86和x64机器, 虚拟地址空间需要分页, 并且页的大小一般都是固定在4KB——虽然在有些情况下每页的大小也可被分为4MB(通过只使用页目录而忽略掉页表)。

图11-30表示了三个x86进程的虚拟地址空间。每个进程的底部和顶端 64KB 的虚拟地址空间通常保留不用。这种做法是为了辅助发现程序错误而设置的。无效的指针通常标志为0或者-1, 使用这样的指针会导致立即陷入中断, 而不会读取垃圾信息、甚至写入错误的内存地址。

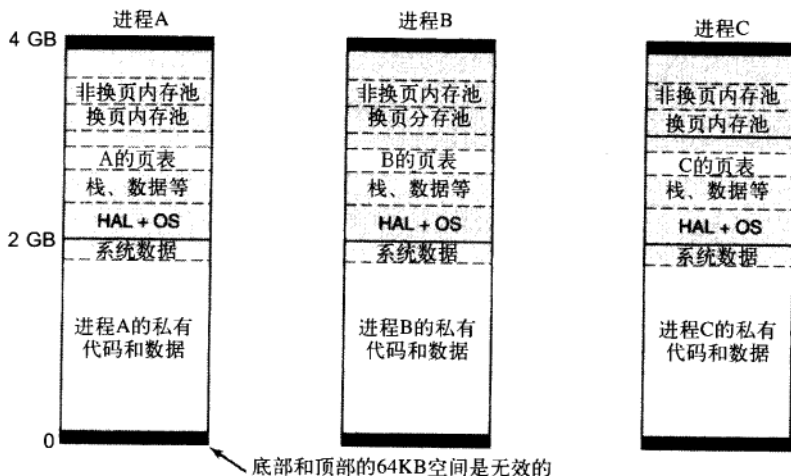


图11-30 x86三个用户进程的虚拟地址空间。白色的区域为每个进程私有的。阴影的区域为所有的进程共享

从64KB开始为用户私有的代码和数据。这些空间可以扩充到几乎2个GB。而最顶端的2GB包含了操作系统部分, 包括代码、数据、换页内存池和非换页内存池。除了每一进程的虚拟内存数据(像页表和工作集的列表), 上面的2GB全部作为内核的虚拟内存、并在所有的用户进程之中共享。内核虚拟内存仅在内核态才可以访问。共享进程在内核部分的虚拟内存的原因是: 当一个线程进行系统调用的时候, 它

陷入内核态之后不需要改变内存映射。所有要做的只是切换到线程的内核栈。由于进程在用户态下的页面仍然是可访问的,内核态下的代码在读取参数和访问缓冲时,就不用在地址空间之间来回切换、或者临时将页面进行两次映射。这里的权衡是通过用较小的进程私有地址空间,来换取更快的系统调用。

当运行在内核态的时候,Windows允许线程访问其余的地址空间。这样该线程就可以访问所有用户态的地址空间,以及对该进程来说通常不可访问的内核地址空间中的区域,例如页表的自映射区域。在线程切换到用户态之前,必须切换到它最初的地址空间。

1. 虚拟地址分配

虚拟地址的每页处于三种状态之一:无效、保留或提交。无效页面(invalid page)是指一个页面没有被映射到一个内存区对象(section object),对它的访问会引发一个相应的页面失效。一旦代码或数据被映射到虚拟页面,就说一个页面处于提交(committed)状态。在提交的页上发生页面失效会导致如下情况:将一个包含了引起失效的虚拟地址的页面映射到这样的页面——由内存区对象所表示,或被保存于页面文件之中。这种情况通常发生在需要分配物理页面,以及对内存区对象所表示的文件进行I/O来从硬盘读取数据的时候。但是页面失效的发生也可能是页表正在更新而造成的,即物理页面仍在内存的高速缓存中,这种情况下不需要进行I/O。这些叫做软异常(soft fault),稍后我们会更详细地讨论它们。

虚拟页面还可以处于保留的(reserved)状态。保留的虚拟页是无效的,但是这些页面不能被内存管理器用于其他目的而分配。例如,当创建一个新线程时,用户态栈空间的许多页保留于进程的虚拟地址空间,仅有一个页面是提交的。当栈增长时,虚拟内存管理器会自动提交额外的页面,直到保留页面耗尽。保留页面的功效是:可以保证栈不会太长而覆盖其他进程的数据。保留所有的虚拟页意味着栈最终可以达到它的最大尺寸;而栈所需要的连续虚拟地址空间的页面,也不会有用于其他用途的风险。除了无效、保留、提交状态,页面还有其他的属性:可读、可写及可运行(在AMD64兼容的处理器下)。

2. 页面文件

关于后备存储器的分配有一个有趣的权衡,已提交页面没有被映射于特定文件。这些页使用了页面文件(pagefile)。问题是该如何以及何时把虚拟页映射到页面文件的特定位置。一个简单的策略是:当一个页被提交时,为虚拟页分配一个硬盘上页面文件中的页。这会确保对于每一个有必要换出内存的已提交页,都有一个确定的位置写回去。

Windows使用一个适时(just-in-time)策略。直到需要被换出内存之前,在页面文件中的具体空间不会分配给已提交的页面。硬盘空间当然不需要分配给永远不换出的页面。如果总的虚拟内存比可用的物理内存少,则根本不需要页面文件。这对基于Windows的嵌入式系统是很方便的。这也是系统启动时的方式,因为页面文件是在第一个用户态进程smss.exe启动之后才初始化的。

在预分配策略下,用于私有数据(如栈、写时复制代码页)的全部虚拟内存受到页面文件大小的限制。通过适时分配的策略,总的虚拟内存大小是物理内存和页面文件大小的总和。既然相对物理内存来说硬盘足够大与便宜,提升性能的需求自然比空间的节省更重要。

有关请求调页,需要马上进行初始化从硬盘读取页的请求——因为在页入(page-in)操作完成之前,遇到页面失效的线程无法继续运行下去。对于失效页面的一个可能的优化是:在进行一次I/O操作时预调入一些额外的页面。然而,对于修改过的页写回磁盘和线程的执行一般并不是同步的。对于分配页面文件空间的适时策略便是利用这一点,在将修改过的页面写入页面文件时提升性能:修改过的页面被集中到一起,统一进行写入操作。由于只有当页面被写回时页面文件的空间才真正被分配,可以通过排列使页面文件中的页面较为接近甚至连续,来对大批写回页面时的寻找次数进行优化。

当存储在页面文件中的页被读取到内存中时,直到它们第一次被修改之前,这些页面一直保持它们在页面文件中的位置。如果一个页面从没被修改过,它将会进入到一个空闲物理页面的列表中去——这个表称作后备链表(standby list),这个表中的页面可以不用写回硬盘而再次被使用。如果它被修改,内存管理器将会释放页面文件中的页,并且内存将保留这个页的惟一副本。这是内存管理器通过把一个加载后的页标识为只读来实现的。线程第一次试图写一个页时,内存管理器检测到它所处的情况并释放页面文件中的页,再授权写操作给相应的页,之后让线程再次进行尝试。

Windows支持多达16个页面文件,通常覆盖到不同的磁盘来达到较高的I/O带宽。每一个页面文件

都有初始的大小和随后依需要可以增长到的最大空间，但是在系统安装时就创建这些文件达到它的最大值是最好的。如果当文件系统非常满却需要增长页面文件时，页面文件的新空间可能会由多个碎片所组成，这会降低系统的性能。

操作系统通过为进程的私有页写入映射信息到页表入口，或与原页表入口相对应的共享页的内存区对象，来跟踪虚拟页与页面文件的映射关系。除了被页面文件保留的页面外，进程中的许多页面也被映射到文件系统上的普通文件。

程序文件中的可执行代码和只读数据（例如EXE或DLL）可以映射到任何进程正在使用的地址空间。因为这些页面无法被修改，它们从来不需要换出内存，然而在页表映射全部被标记为无效后，可以立即重用物理页面。当一个页面在今后再次需要时，内存管理器将从程序文件中将其读入。

有时候页面开始时为只读但最终被修改。例如，当调试进程时在代码中设定中断点，或将代码重定向为进程中不同的地址，或对于开始时为共享的数据页面进行修改。在这些情况下，像大多数现代操作系统一样，Windows支持写时复制（copy-on-write）类型的页面。这些页面开始时像普通的被映射页面一样，但如果试图修改任何部分页面，内存管理器将会建立一份私有的、可写的副本。然后它更新虚拟页面的页表，使之指向那个私有副本，并且使线程重新进行写操作——这一次将会成功。如果这个副本之后需要被换出内存，那么它将被写回到页面文件而不是原始文件中。

除了从EXE和DLL文件映射程序代码和数据，一般的文件都可以映射到内存中，使得程序不需要进行显式的读写操作就可以从文件引用数据。I/O操作仍然是必要的，但它们由内存管理器通过使用内存区对象隐式提供，来表示内存中的页面和磁盘中的文件块的映射。

内存区对象并不一定和文件相关。它们可以和匿名内存区域相关。通过映射匿名内存区对象到多个进程，内存可以在不分配磁盘文件的前提下共享。既然内存区可以在NT名字空间给予名字，进程可以通过用名字打开内存区对象、或者复制进程间的内存区对象句柄的方式来进行通信。

3. 大物理内存寻址

多年前，当16位（或20位）的地址空间还作为标准的时候，机器已有兆字节的物理内存，人们努力想出各种技术使得程序可以使用更多的物理内存、而不是去适应有限的地址空间。这些技术通常基于存储器组转换（bank switching），使得一个程序可以突破16或者20位的限制，替换掉自己的一些内存块。在刚引入32位计算机时，大多数桌面计算机只有几个兆的物理内存。然而随着内存集成电路上变得更加密集，可用内存开始迅速增长。这推动了服务器的发展，因为服务器上的应用程序往往需要更多的内存。英特尔的Xeon芯片支持物理地址扩展（PAE），物理内存寻址空间从32位变为36位，意味着一个单一的系统可以支持高达64GB的物理内存。这远远大于2G或者3G——单个进程可以在32位的用户模式寻址的虚拟地址空间，然而一些像SQL数据库这样的大型应用软件恰恰被设计为运行在一个单个进程的寻址空间中，因此存储器组转换已经过时了，取代它的是地址窗口扩展（Address Windowing Extensions, AWE）。这种机制允许程序（以正确的特权级运行）去请求物理内存的分配。进程可以保留所需的虚拟地址，并请求操作系统进行虚拟地址与物理地址间的映射。在所有的服务器应用64位寻址方式前，AWE一直充当权宜之计的角色。

11.5.2 内存管理系统调用

Win32 API 包含了大量的函数来支持一个进程显式地管理它自己的虚拟内存，其中最重要的函数如图11-31所示。它们都是在包含一个单独的页或一个由两个或多个在虚拟地址空间中连续页的序列的区域上进行操作的。

前四个API函数是用来分配、释放、保护和查询虚拟地址空间中的区域的。被分配的区域总是从64KB的边界开始，以尽量减少移植到将来的体系结构的问题（因为将来的体系结构可能使用比当前使用的页更大的页）。实际分配的地址空间可以小于64KB，但是必须是一个页大小的整数倍。接下来的两个API给一个进程把页面固定到内存中以防止它们被替换到外存以及撤销这一性质的功能。举例来说，一个实时程序可能需要它的页面具有这样的性质以防止在关键操作上发生页面失效。操作系统强加了一个限制来防止一个进程过于“贪婪”：这些页面能够移出内存，但是仅仅在整个进程被替换出内存的时候才能这么做。当该进程被重新装入内存时，所有之前被指定固定到内存中的页面会在任何线程开始运

行之前被重新装入内存。尽管没有从图11-31中体现出来，Windows Vista还包含一些原生API函数来允许一个进程访问其他进程的虚拟内存。前提是该进程被给予了控制权，即它拥有一个相应的句柄。

Win32 API 函数	描 述
VirtualAlloc	保留或提交一个区域
VirtualFree	释放或解除提交一个区域
VirtualProtect	改变在一个区域上的读/写/执行保护
VirtualQuery	查询一个区域的状态
VirtualLock	使一个区域常驻内存（即不允许被替换到外存）
VirtualUnlock	使一个区域以正常的方式参与页面替换策略
CreateFileMapping	创建一个文件映射对象并且可以选择是否赋予该对象一个名字
MapViewOfFile	映射一个文件（或一个文件的一个部分）到地址空间中
UnmapViewOfFile	从地址空间中删除一个被映射的文件
OpenFileMapping	打开一个之前创建的文件映射对象

图11-31 Windows中用来管理虚拟内存的主要的Win32 API函数

列出的最后四个API函数是用来管理内存映射文件的。为了映射一个文件，首先必须通过调用CreateFileMapping来创建一个文件映射对象（见图11-23）。这个函数返回一个文件映射对象（即一个内存区对象）的句柄，并且可以选择是否为该操作添加一个名字到Win32地址空间中，从而其他的进程也能够使用它。接下来的两个函数从一个进程的虚拟地址空间中映射或取消映射内存区对象之上的视图。最后一个API能被一个进程用来映射其他进程通过调用CreateFileMapping创建并共享出来的映射，这样的映射通常是为了映射匿名内存而建立的。通过这样的方式，两个或多个进程能够共享它们地址空间中的区域。这一技术允许它们写内容到相互的虚拟内存的受限的区域中。

11.5.3 存储管理的实现

运行在x86处理器上的Windows Vista操作系统为每个进程都单独提供了一个4GB大小的按需分页（demand-paged）的线性地址空间，不支持任何形式的分段。从理论上说，页面的大小可以是不超过64KB的2的任何次幂。但是在Pentium处理器上，页面正常情况下固定地设置成4KB大小。另外，操作系统可以使用4MB的页来改进处理器存储管理单元中的快表（Translation Lookaside Buffer, TLB）的效率。内核以及大型应用程序使用了4MB大小的页面以后，可以显著地提高性能。这是因为快表的命中率提高了，并且访问页表以寻找在快表中没有找到的表项的次数减少了。

调度器选择单个线程来运行而不太关心进程，存储管理器则不同，它完全是在处理进程而不太关心线程。毕竟，是进程而非线程拥有地址空间，而地址空间正是存储管理器所关心的。当虚拟地址空间中的一片区域被分配之后，就像图11-32中进程A被分配了4片区域那样，存储管理器会为它创建一个虚拟地址描述符（Virtual Address Descriptor, VAD）。VAD列出了被映射地址的范围，用来表示作为后备存储的文件以及文件被映射区域起始位置的节区以及权限。当访问第一个页面的时候，创建一个页目录并且把它的物理地址插入进程对象中。一个地址空间被一个VAD的列表所完全定义。VAD被组织成平衡树的形式，从而保证一个特定地址的描述符能够被快速地找到。这个方案支持稀疏的地址空间。被映射的区域之间未使用的地址空间不会使用任何内存中或磁盘上的资源，从这个意义上说，它们是“免费”的。

1. 页面失效处理

当在Windows Vista上启动一个进程的时候，很多映射了程序的EXE和DLL映像文件的页面可能已经在内存中，这是因为它们可能被其他进程共享。映像中的可写页面被标记成写时复制（copy-on-write），使得它们能一直被共享，直到内容要被修改的那一刻。如果操作系统从一次过去的执行中认出了这个EXE，它可能已经通过使用微软称之为超级预读取（SuperFetch）的技术记录了页面引用的模式。超级预读取技术尝试预先读入很多需要的页面到内存中，尽管进程尚未在这些页面上发生页面失效。这一技术通过重叠从磁盘上读入页面和执行映像中的初始化代码，减小了启动应用程序所需的延时。同时，它改进了磁盘的吞吐量，因为使用了超级预读取技术以后，磁盘驱动器能够更轻易地组织对磁盘的读请求

来减少所需的寻道时间。进程预约式页面调度（prepaging）技术也用到了系统启动、把后台应用程序移到前台以及休眠之后重启系统当中。

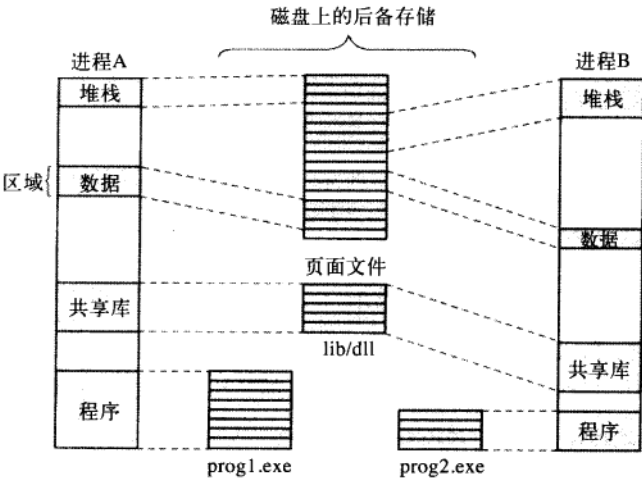


图11-32 被映射的区域以及它们在磁盘上的“影子”页面。lib.dll 文件被同时映射到两个地址空间中

存储管理器支持预约式页面调度，但是它被实现成系统中一个单独的组件。被读入到内存的页面不是插入到进程的页表中，而是插入到后备列表中，从而使得在需要时可以不访问磁盘就将它们插入到进程中。

未被映射的页面稍微有些不同。它们没有被通过读取文件来初始化。相反，一个未被映射的页面第一次被访问的时候，存储管理器会提供一个新的物理页面，该页面的内容被事先清零（为了安全方面的原因）。在后续的页面失效处理过程中，未被映射的页面可能会被从内存中找到，否则的话，它们必须被从页面文件中重新读入内存。

存储管理器中的按需分页是通过页面失效来驱动的。在每次页面失效发生的时候，会发生一次到内核的陷入。内核将建立一个说明发生了什么事情的机器无关的描述符，并把该描述符传递给存储管理器相关的执行部件。存储管理器接下来会检查引发页面失效的内存访问的有效性。如果发生页面失效的页面位于一个已提交的区域内，存储管理器将在VAD列表中查找页面地址并找到（或创建）进程页表项。对于共享页面的情况，存储管理器使用与内存区对象关联的原始页表项来填写进程页表中的新页表项。

不同处理器体系结构下的页表项的格式可能会不同。对于x86和x64，一个被映射页面的页表项如图11-33所示。如果一个页表项被标记为有效，它的内容会被硬件读取并解释，从而虚拟地址能够转换成正确的物理地址。未被映射的页面也有对应的页表项，但是这些页表项被标记成无效，硬件将忽略这些页表项除该标记之外的部分。页表项的软件格式与硬件格式有所不同，软件格式由存储管理器决定。例如，对于一个未映射的页面，它必须在使用前分配和清零，这一点可以通过页表项来表明。

页表项中有两个重要的位是直接由硬件更新的，它们是访问位（access bit）和脏位（dirty bit）。这两个位跟踪了什么时候一个特定的页面映射用来访问该页面以及这个访问是否以写的方式修改了页面的内容。这确实很有助于提高系统性能。因为存储管理器可以使用访问位来实现LRU（Least-Recently Used，最近最少使用）类型的页面替换策略。LRU原理是，那些最长时间没有被使用过的页面有最小的可能性在不久的将来被再次使用。访问位使存储管理器知道一个页面被访问过了，脏位使存储管理器知道一个页面被修改了，或者更重要的是，一个页面没有被修改。如果一个页面自从从磁盘上读到内存后没有被修改过，存储管理器就没有必要在将该页面用到其他地方之前将页面内容写回磁盘了。

正如表11-33所示，x86体系结构通常使用32位大小的页表项，而x64体系结构使用64位大小的页表项。在域上面的唯一区别是x64的物理页号域是30位，而不是20位。然而，现今存在的任何x64处理器所支持的物理页面的数量都要远小于x64体系结构所能表示的数量。x86体系结构也支持一种特殊的物理地址扩展（Physical Address Extension, PAE）。PAE模式允许处理器访问超过4GB的物理内存，附加的物理页框位要求PAE模式下的页表项也是64位。

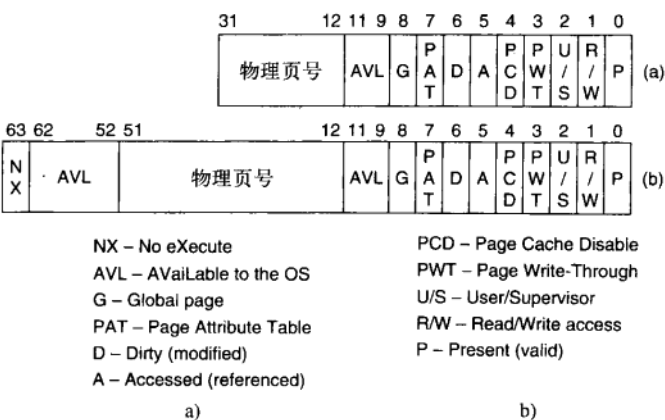


图11-33 一个a) Intel x86体系结构和b) AMD x64体系结构上的已映射页面的页表项 (PTE)

每个页面失效都可以归入以下五类中的一类：

- 1) 所引用的页面没有提交。
- 2) 尝试违反权限的页面访问。
- 3) 修改一个共享的写时复制页面。
- 4) 需要扩大栈。
- 5) 所引用的页已经提交但是当前没有映射。

第一种和第二种情况是由于编程错误引起。如果一个程序试图使用一个没有一个有效映射的地址或试图进行一个称为访问违例 (access violation) 的无效操作 (例如试图写一个只读的页面)，通常的结果是，这个进程会被终止。访问破坏的原因通常是坏指针，包括访问从进程释放的和被解除映射的内存。

第三种情况与第二种情况有相同的症状 (试图写一个只读的页面)，但是处理方式是不一样的。因为页面已经标记为写时复制，存储管理器不会报告访问违例，相反它会为当前进程产生一个该页面的私有副本，然后返回到试图写该页面的线程。该线程将重写操作，而这次的写操作将会成功完成而不会引发页面失效。

第四种情况在线程向栈中压入一个值，而这个值会被写到一个还没有被分配的页面的情况下发生。存储管理器程序能够识别这种特殊情况。只要为栈保留的虚拟页面还有空间，存储管理器就会提供一个新的物理页面，将该页面清零，最后把该页面映射到进程地址空间。线程在恢复执行的时候会重试上次引发页面失效的内存访问，而这次该访问会成功。

最后，第五种情况就是常见的页面失效。这种异常包含下述几种情况。如果该页是由文件映射的，内存管理器必须查找该页与内存区对象结合在一起的原型页表等类似的数据结构，从而保证在内存中不存在该页的副本。如果该页的副本已经在内存中，即在另一个进程的页面链表已经存在该页面的副本，或者在后备、已修改页链表中，则只需要共享该页即可。否则，内存管理器分配一个空闲的物理页面，并安排从磁盘复制文件页。

如果内存管理器能够从内存中找到需要的页而不是去磁盘查找从而响应页面失效，则称为软异常 (soft fault)。如果需要从磁盘进行复制，则称为硬异常 (hard fault)。软异常同硬异常相比开销更小，对于应用程序性能的影响很小。软异常出现在下面场景中：一个共享的页已经映射到另一个进程；请求一个新的全零页，或所需页面已经从进程的工作集移除，但是还没有重用。

当一个物理页面不再映射到任何进程的页表，将进入以下三种状态之一：空闲、修改或后备。内存管理器会立刻释放类似那些已结束进程的栈页面这样不再会使用的页面。根据判断映射页面的页表项中的上次从磁盘读出后的脏位是否设置，页面可能会再次发生异常，从而进入已修改链表或者后备链表 (standby list)。已修改链表中的页面最终会写回磁盘，然后移到后备链表中。

内存管理器可以根据需要从空闲链表或者后备链表中分配页面。它在分配页面并从磁盘复制之前，

总是在已修改链表和后备链表中检查该页面是否已经在内存中。Windows Vista中的预约式调页机制通过读入那些未来可能会用到的页面并把它们插入后备链表的方式将硬异常转化为软异常。内存管理器通过读入成组的连续页面而不是仅仅一个页面来进行一定数量的普通预约式调页。多余调入的页面立刻插入后备链表。而由于内存管理器的开销主要是进行I/O操作引起的,因而预约式调页并不会带来很大的浪费。与读入一簇页面相比,仅读入一个页面的额外开销是可以忽略的。

图11-33中的页表项指的是物理页号,而不是虚拟页号。为了更新页表(以及页目录)项,内核需要使用虚拟地址。Windows使用如图11-34所示的页目录表项中的自映射(self-map)表项将当前进程的页表和页目录映射到内核虚拟地址空间。通过映射页目录项到页目录(自映射),就具有了能用来指向页目录项(图11-34a)和页表项(图11-34b)的虚拟地址。每个进程的自映射占用4MB内核地址空间(x86上)。幸运的是,该4MB地址空间是同样一块地址空间。

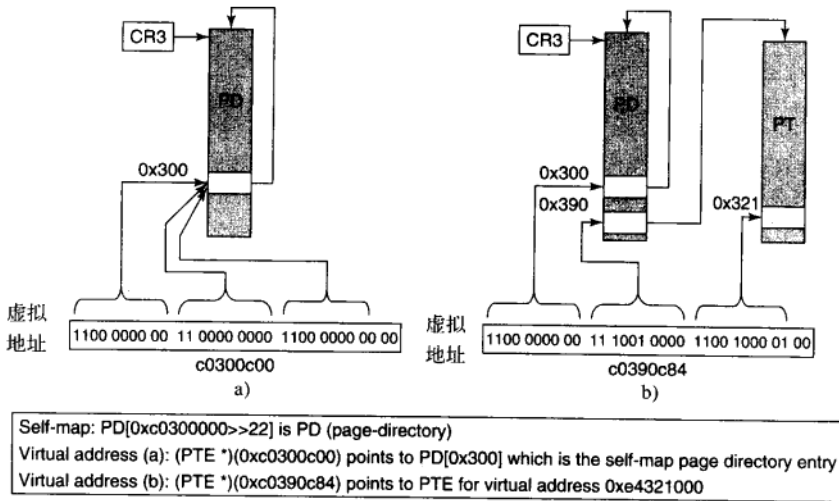


图11-34 x86上, Windows用来映射页表和页目录的物理页面到内核虚拟地址的自映射表项

2. 页面置换算法

当空闲物理页面数量降得较低时,内存管理器开始从内核态的系统进程以及用户态进程移走页面。目标就是使得最重要的虚拟页面在内存中,而其他的在磁盘上。决定什么是重要的需要技巧。Windows通过大量使用工作集来解决这一问题。工作集处在内存中,不需要通过页面失效即可使用的映射入内存的页面。当然,工作集的大小和构成随着从属于进程的线程运行来回变动。

每个进程的工作集由两个参数描述:最小值和最大值。这两个参数并不是硬性边界,因而一个进程在内存中可能具有比它的工作集最小值还小的页面数量(在特定的环境下),或者比它的工作集最大值还大得多的页面数量。每个进程初始具有同样的最大值和最小值的工作集,但这些边界随着时间的推移是可以改变的,或是由包含在作业中的进程的作业对象决定。根据系统中的全部物理内存大小,这个默认的初始最小值的范围是20~50个页面,而最大值的范围是45~345个页面。系统管理员可以改变这些默认值。尽管一般的家庭用户很少去设置,但是服务器端程序可能需要设置。

只有当系统中的可用物理内存降得很低的时候工作集才会起作用。其他情况下允许进程任意使用它们选择的通常远远超出工作集最大值的内存。但是当系统面临内存压力的时候,内存管理器开始将超出工作集上限最大的进程使用的内存压回到它们的工作集范围内。工作集管理器具有三级基于定时器的周期活动。新的活动会加入到相应的级别。

1) **大量的可用内存**: 扫描页面,复位页面的访问位,并使用访问位的值来表示每个页面的新旧程度。在每个工作集内保留使用一个估算数量的未使用页面。

2) **内存开始紧缺**: 对每个具有一定比例未用页面的进程,停止为工作集增加页面,同时在需要增加

一个新的页面的时候换出最旧的页面。换出的页面进入后备或者已修改链表。

3) 内存紧缺：消减（也即减小）工作集，通过移除最旧的页面从而降低工作集的最大值。

平衡集管理器（balance set manager）线程调用工作集管理器，使得其每秒都在运行。工作集管理器抑制一定数量的工作从而不会使得系统过载。它同时也监控要写回磁盘的已修改链表上的页面，通过唤醒ModifiedPageWriter线程使得页面数量不会增长得过快。

3. 物理内存管理

上面提到了物理页面的三种不同链表，空闲链表、后备链表和已修改链表。除此以外还有第四种链表，即全部被填零的空闲页面。系统会频繁地请求全零的页面。当为进程提供新的页面，或者读取一个文件的最后部分不足一个页面时，需要全零页面。将一个页面写为全零是需要时间的，因此在后台使用低优先级的线程创建全零页是一个较好的方式。另外还有第五种链表存放有硬件错误的页面（即通过硬件错误检测）。

系统中的所有页面要么由一个有效的页表项索引，要么属于以上五种链表中的一种，它们的全体称为页框号数据库（PFN数据库）。图11-35表明PFN数据库的结构。该表格由物理页框号索引。表项都是固定长度的，但是不同类型的表项使用不同的格式（例如共享页面相对于私有页面）。有效的表项维护页面的状态以及指向该页面数量的计数。工作集中的页面指出哪个表项索引它们。还有一个指向该页的进程页表的指针（非共享页），或者指向原型页表的指针（共享页）。

此外还有一个指向链表中下一个页面的指针（如果有的话），以及其他的若干诸如正在进行读和写的域以及标志位等。这些链表链接在一起，并且通过下标指向下一个单元，不使用指针，从而达到节省存储空间的目的。另外用物理页面的表项汇总在若干指向物理页面的页表项中找到的脏位（即由于共享页面）。表项还有一些别的信息用来表示内存页面的不同，以便访问那些内存速度更快的大型服务器系统上（即NUMA-非均衡存储器访问的机器）。



图11-35 一个有效的页面在页框数据库上的一些主要域

工作集管理器和其他的系统线程控制页面在工作集和不同的链表间移动。下面对这些转变进行研究。当工作集管理器将一个页面从某个工作集中去掉，则该页面按照自身是否修改的状态进入后备或已修改链表的底部。这一转变在图11-36的（1）中进行了说明。

这两个链表中的页面仍然是有效的页面，当页面失效发生的时候需要它们中的一个页，则将该页移回工作集而不需要进行磁盘I/O操作（2）。当一个进程退出，该进程的非共享页面不能通过异常机制回到以前的工作集，因此该进程页表中的有效页面以及挂起和已修改链表中的页面都移入空闲链表（3）。任何该进程的页面文件也得到释放。

其他的系统调用会引起别的转变。平衡集管理器线程每4秒运行一次来查找那些所有的线程都进入空闲状态超过一定秒数的进程。如果发现这样的进程，就从物理内存去掉它们的内核栈，这样的进程的

页面也如(1)一样移动到后备链表或已修改链表。

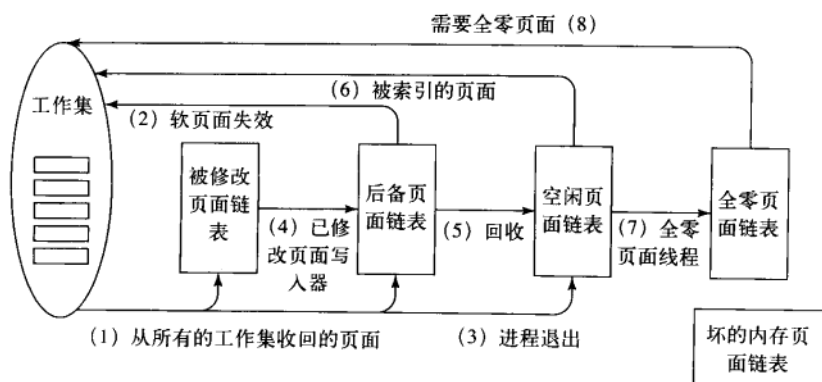


图11-36 不同的页面链表以及它们之间的转变

两个系统线程——映射页面写入器 (mapped page writer) 和已修改页面写入器 (modified page writer)，周期性地被唤醒来检查是否有足够的干净页面。如果没有，这两个线程从已修改链表的顶部取出页面，写回到磁盘，然后将这些页面插入后备链表(4)。前者处理对于映射文件的写，而后者处理页面文件的写。这些写的结果就是将已修改(脏)页面移到后备(干净)链表中。

之所以使用两个线程是因为映射文件可能会因为写的结果增长，而增长的结果就需要对磁盘上的数据结构具有相应的权限来分配空闲磁盘块。当一个页面被写入时如果没有足够的内存，就会导致死锁。另一个线程则是解决向页面文件写入页时的问题。

下面说明图11-36中另一个转换。如果进程解除页映射，该页不再和进程相关从而进入空闲链表(5)，当该页是共享的时候例外。当页面失效会请求一个页框给将要读入的页，此时该页框会尽可能从空闲链表中取下(6)。由于该页会被全部重写，因此即使有机密的信息也没有关系。

栈的增长则是另一种情况。这种情况下，需要一个空的页框，同时安全规则要求该页全零。由于这个原因，另一个称为零页面线程 (ZeroPage thread) 的低优先级内核线程 (参见图11-28) 将空闲链表中的页面写全零并将页面放入全零页链表(7)。全零页面很可能比空闲页面更加有用，因此只要当CPU空闲且有空闲页面，零页面线程就会将这些页面全部写零，而在CPU空闲的时候进行这一操作也是不增加开销的。

所有这些链表的存在导致了一些微妙的策略抉择。例如，假设要从磁盘载入一个页面，但是空闲链表是空的，那么，要么从后备链表中取出一个干净页(虽然这样做稍后有可能导致缺页)，要么从全零页面链表中取出一个空页(忽略将该页清零的代价)，系统必须在上述两种策略之间做出选择。哪一个更好呢？

内存管理器必须决定系统线程把页面从已修改链表移动到后备链表的积极程度。有干净的页面后备总比有脏页后备好得多(因为如有需要，干净的页可以立即重用)，但是一个积极的净化策略意味着更多的磁盘I/O，同时一个刚刚净化的页面可能由于缺页中断重新回到工作集中，然后又成为脏页。通常来讲，Windows通过算法、启发、猜测、历史、经验以及管理员可控参数的配置来做权衡。

总而言之，内存管理需要一个拥有多种数据结构、算法和启发性的十分复杂、重要的构件。它尽可能地自我调整，但是仍然留有很多选项使系统管理员可以通过配置这些选项来影响系统性能。大部分的选项和计数器可以通过工具浏览，相关的各种工具包在前面都有提到。也许在这里最值得记住的就是，在真实的系统里，内存管理不仅仅是一个简单的时钟或老化的页面算法。

11.6 Windows Vista的高速缓存

Windows高速缓存(cache)通过把最近和经常使用的文件片段保存在内存中的方式来提升文件系统的性能。高速缓存管理器管理的是虚拟寻址的数据块，也就是文件片段，而不是物理寻址的磁盘块。这种方法非常适合NTFS文件系统，如11.8节所示。NTFS把所有的数据作为文件来存储，包括文件系统的元数据。

高速缓存的文件片段称为视图 (view), 这是因为它们代表了被映射到文件系统的文件上的内核虚拟地址片段。所以, 在高速缓存中, 对物理内存的管理实际上是由内存管理器提供的。高速缓存管理器的作用是视图管理内核虚拟地址的使用, 命令内存管理器在物理内存中钉住页面, 以及为文件系统提供接口。

Windows高速缓存管理工具在文件系统中被广泛地共享。这是因为高速缓存是根据独立的文件来虚拟寻址的, 高速缓存管理器可以在文件的基础上很轻易地实现预读取。访问高速缓存数据的请求来自于每个文件系统。由于文件系统不需要先把文件的偏移转换成物理磁盘号然后再请求读取高速缓存的文件页, 所以虚拟缓存非常方便。类似的转换发生在内存管理器调用文件系统访问存储在磁盘上的页面的时候。

除了对内核虚拟地址和用来缓存的物理内存资源的管理外, 考虑到视图的一致性, 大批量磁盘回写, 以及文件结束标志的正确维护 (特别是当文件扩展的时候), 高速缓存管理器还必须与文件系统协作。在文件系统、高速缓存管理器和内存管理器之间管理文件最困难的方面在于文件中最后一个字节的偏移, 即有效数据长度。如果一个程序写出了文件末尾, 则越过的磁盘块都需要清零, 同时为了安全的原因, 在文件的元数据中记录的有效数据长度不应该允许访问未经初始化的磁盘块, 所以全零磁盘块在文件元数据更新为新的长度之前必须写回到磁盘上。然而, 可以预见的是, 如果系统崩溃, 一些文件的数据块可能还没有按照内存中的数据进行更新, 还有一些数据块可能含有属于其他文件的数据, 这都是不能接受的。

现在让我们来看看高速缓存管理器是如何工作的。当一个文件被引用时, 高速缓存管理器映射一块大小为256KB的内核虚拟地址空间给文件。如果文件大于256KB, 那么每次只有一部分文件被映射进来。如果高速缓存管理器耗尽了虚拟地址空间中大小为256KB的块, 那么, 它在映射一个新文件之前必须释放一个旧的文件。文件一旦被映射, 高速缓存管理器通过把内核虚拟地址空间复制到用户缓冲区的方式来满足对该数据块的请求。如果要复制的数据块不在物理内存当中, 会发生缺页中断, 内存管理器会按照通常的方式处理该中断。高速缓存管理器甚至不知道一个数据块是不是在内存当中。复制总是成功的。

除了在内核和用户缓冲区之间复制的页面, 高速缓存管理器也为映射到虚拟内存的页面和依靠指针访问的页面服务。当一个线程访问某一映射到文件中的虚拟地址但发生缺页的时候, 内存管理器在大多数情况下能够使用软中断处理这种访问。如果该页面已经被高速缓存管理器映射到内存当中, 即该页面已经在物理内存当中, 那么就不需要去访问磁盘了。

高速缓存不一定适合所有的应用程序。大型企业应用程序, 如SQL, 希望自己来管理高速缓存和I/O。Windows允许文件绕开高速缓存管理器, 以未缓冲I/O的方式打开。从历史上看, 这类应用程序使用一个可增长的用户态虚拟地址空间来替代操作系统提供的高速缓存, 因此, 系统应支持一种配置, 使得重新启动后能给应用程序提供其所需的3GB的用户态地址空间, 而只使用1GB的地址空间用于内核态来代替2-GB/2-GB的传统分割。这种运行模式 (启动选项启用后, 称为3GB模式) 在一些允许以多种粒度来调整用户/内核地址空间分割的操作系统上不太灵活。当Windows运行在3GB模式下时, 只有一半数量的内核虚拟地址可用。高速缓存管理器通过映射更少的文件来进行调整, 这正是SQL所喜欢的。

Windows Vista在系统中引入了一种全新的、有别于高速缓存管理器的缓存技术, 称为ReadyBoost。用户可以在USB接口或其他端口插入闪存, 并命令操作系统使用闪存作为一个通写缓存。闪存引入了一种新的存储层次, 这对于增加磁盘读缓存的数量特别有用。虽然比不上作为普通内存的动态RAM (DRAM), 但是从闪存读取数据还是相当快的。结合高速的DRAM和相对廉价的闪存, Vista系统以少量的DRAM, 使得不必开启机箱就可以获得更高的性能。

ReadyBoost压缩数据 (通常为2倍), 并加密。ReadyBoost使用一个过滤驱动程序来处理文件系统发送到卷管理器的I/O请求。名为ReadyBoot的类似技术, 通过使用闪存缓存数据来加速Windows Vista系统的启动时间。但是这些技术对拥有1GB或更多内存的系统影响较小。在只有512MB内存的系统上尝试运行Windows Vista才是它们真正有帮助的地方。内存容量将近1GB的系统拥有足够的内存, 页面请求非常罕见, 使得磁盘I/O能够满足大多数使用场景。

通写方式对闪存被拔除时减少数据丢失很重要, 但未来的PC硬件可能在主板上直接集成闪存。这样, 闪存不用通写方式也可以使用, 从而缓存系统故障时也需要继续存在的关键数据, 而无需旋转磁盘。这不仅带来了性能的提升, 而且还可以降低能耗 (从而提高笔记本电脑的电池寿命), 因为磁盘旋转少了。现在一些笔记本电脑一直在致力于使用大量的闪存来代替机电磁盘。

11.7 Windows Vista的输入/输出

Windows I/O管理器提供了灵活的、可扩展的基础框架,以便有效地管理非常广泛的I/O设备和服务,支持自动的设备识别和驱动程序安装(即插即用)及用于设备和CPU的电源管理——以上均基于异步结构使得计算可以与I/O传输重叠。大约有数以十万计的设备在Windows Vista上工作。一大批常用设备甚至不需要安装驱动程序,因为Windows操作系统已附带其驱动程序。但即使如此,考虑到所有的版本,也有将近100万种不同的驱动程序在Windows Vista上运行。以下各节中,我们将探讨一些I/O相关的问题。

11.7.1 基本概念

I/O管理器与即插即用管理器紧密联系。即插即用背后的基本思想是一条可枚举总线。许多总线的设计,包括PC卡、PCI、PCI-x、AGP、USB、IEEE 1394、EIDE和SATA,都支持即插即用管理器向每个插槽发送请求,并要求每个插槽上的设备表明身份。即插即用管理器发现设备的存在以后,就为其分配硬件资源,如中断等级,找到适当的驱动程序,并加载到内存中。每个驱动程序加载时,就为其创建一个驱动程序对象(driver object)。每个设备至少分配一个设备对象。对于一些总线,如SCSI,枚举只发生在启动时间,但对于其他总线,如USB,枚举可以在任何时间发生,这就需要即插即用管理器,总线驱动程序(确实在枚举的总线),和I/O管理器之间的密切协作。

在Windows中,所有与硬件无关的程序,如文件系统、反病毒过滤器、卷管理器、网络协议栈,甚至内核服务,都是用I/O驱动程序来实现的。系统配置必须设置成能够加载这些驱动程序,因为在总线上不存在可枚举相关的设备。其他如文件系统,在需要时由特殊代码加载,例如文件系统识别器查看裸卷,以及辨别文件系统格式的时候。

Windows的一个有趣的特点是支持动态磁盘(dynamic disk)。这些磁盘可以跨越多个分区,或多个磁盘,甚至无需重新启动在使用中就可以重新配置。通过这种方式,逻辑卷不再被限制在一个单一的分区或磁盘内,一个单一的文件系统也可以透明地跨越多个驱动器。

从I/O到卷可被一个特殊的Windows驱动程序过滤产生卷阴影副本(Volume Shadow Copies)。过滤驱动程序创建一个可单独挂载的,并代表某一特定时间点的卷快照。为此,它会跟踪快照点后的变化。这对恢复被意外删除的文件或根据定期生成的卷快照查看文件过去的状态非常方便。

阴影副本对精确备份服务器系统也很有价值。在该系统上运行服务器应用程序,它们可以在合适的时机制作一个干净的持久备份。一旦所有的应用程序准备就绪,系统初始化卷快照,然后通知应用程序继续执行。备份由卷快照组成。这与备份期间不得不脱机相比,应用程序只是被阻塞了很短的时间。

应用程序参与快照过程,因此一旦发生故障,备份反映的是一个非常易于恢复的状态。否则,就算备份仍然有用,但抓取的状态将更像是系统崩溃时的状态。而从崩溃点恢复系统更加困难,甚至是不可能的,因为崩溃可能在应用程序执行过程的任意时刻发生。墨菲定律说,故障最有可能在最坏的时候发生,也就是说,故障可能在应用程序的数据正处于不可恢复的状态时发生。

另一方面,Windows支持异步I/O。一个线程启动一个I/O操作,然后与该I/O操作并行执行。这项功能对服务器来说特别重要。有各种不同的方法使线程可以发现该I/O操作是否已经完成。一是启动I/O操作的同时指定一个事件对象,然后等待它结束。另一种方法是指定一个队列,当I/O操作完成时,系统将一个完成事件插入到队列中。三是提供一个回调函数,I/O操作完成时供系统调用。四是在内存中开辟一块区域,当I/O操作完成时由I/O管理器更新该区域。

我们要讨论的最后一个方面,是由Windows Vista提出的I/O优先级。I/O优先级是由发起I/O操作的线程来确定的,或者也可以明确指定。共有5个优先级别,分别是:关键、高、正常、低、非常低。关键级别为内存管理器预留,以避免系统经历极端内存压力时出现死锁现象。低和非常低的优先级为后台进程所使用,例如磁盘碎片整理服务、间谍软件扫描器和桌面搜索,以免干扰正常操作。大部分I/O操作的优先级是正常级别,但是为避免小故障,多媒体应用程序也可标记它们的I/O优先级为高。多媒体应用可有选择地使用带宽预留模式获得带宽保证以访问时间敏感的文件,如音乐或视频。I/O系统将给应用程序提供最优的传输大小和显式I/O操作的数目,从而维持应用程序向I/O系统请求的带宽保证。

11.7.2 输入/输出API调用

由I/O管理器提供的API与大多数操作系统提供的API并没有很大的不同。基本操作有open、read、

write、ioctl和close, 以及即插即用和电源操作、参数设置、刷新系统缓冲区等。在Win32层, 这些API被包装成接口, 向特定的设备提供了更高一级的操作。在底层, 这些API打开设备, 并执行这些基本类型的操作。即使是对一些元数据的操作, 如重命名文件, 也没有用专门的系统调用来实现。它们只是特殊的ioctl操作。在我们解释了I/O设备栈和I/O管理器使用的I/O请求包 (IRP) 之后, 读者将对上面的陈述更有体会。

I/O系统调用	描 述
NtCreateFile	打开一个新的或已存在的文件或设备
NtReadFile	从一个文件或设备上读取数据
NtWriteFile	把数据写到一个文件或设备
NtQueryDirectoryFile	请求关于一个目录的信息, 包括文件
NtQueryVolumeInformationFile	请求关于一个卷的信息
NtSetVolumeInformationFile	修改卷信息
NtNotifyChangeDirectoryFile	当任何在此目录中或其子目录树中的文件被修改时执行完成
NtQueryInformationFile	请求关于一个文件的信息
NtSetInformationFile	修改文件信息
NtLockFile	给文件中一个区域加锁
NtUnlockFile	解除区域锁
NtFsControlFile	对一个文件进行多种操作
NtFlushBuffersFile	把内存文件缓冲刷新到磁盘
NtCancelIoFile	取消文件上未完成的I/O操作
NtDeviceIoControlFile	对一个设备的特殊操作

图11-37 执行I/O的原生NT API调用

保持了Windows一贯的通用哲学, 原生NT I/O系统调用带有很多参数并包括很多变种。图11-37列出了I/O管理器中主要的系统调用接口。NtCreateFile用于打开已经存在的或者新的文件。它为新创建的文件提供了安全描述符和一个对被请求的访问权限的详细描述, 并使得新文件的创建者拥有了一些如何分配磁盘块的控制权。NtReadFile和NtWriteFile需要文件句柄、缓冲区和长度等参数。它们也需要一个明确的文件偏移量的参数, 并且允许指定一个用于访问文件锁定区域字节的钥匙。正如上面提到的, 大部分的参数都和指定哪一个函数来报告 (很可能是异步) I/O操作的完成有关。

NtQuerydirectoryFile是一个在执行过程中访问或修改指定类型对象信息的标准模式的一个例子, 在这种模式中存在多种不同的查询API。在本例中, 指定类型的对象是指与某些目录相关的一些文件对象。一个参数用于指定请求什么类型的信息, 比如目录中的文件名列, 或者是经过扩展的目录列表所需要的每个文件的详细信息。由于它实际上是一个I/O操作, 因此它支持所有的报告I/O操作已完成的标准方法。NtQueryVolumeInformationFile很像是目录查询操作, 但是与目录查询操作不同的是, 它有一个参数是打开的卷的文件句柄, 不管这个卷上是否有文件系统。与目录不同的是, 卷上有一些参数可以修改, 因此这里有了单独用于卷的API NtSetVolumeInformationFile。

NtNotifyChangeDirectoryFile是一个有趣的NT范式的例子。线程可以通过I/O操作来确定对象是否发生了改变 (对象主要是文件系统的目录, 就像在此例中; 也可能是注册表键)。因为I/O操作是异步的, 所以线程在调用I/O操作后会立即返回并继续执行, 并且只有在修改对象之后线程才会得到通知。未处理的请求作为一个外部的I/O操作, 使用一个I/O请求包 (IRP) 被加入到文件系统的队列中等待。如果想从系统移除一个文件系统卷, 给执行过未处理I/O操作的线程的通知就会出问题, 因为那些I/O操作正在等待。因此, Windows提供了取消未处理I/O操作的功能, 其中包括支持文件系统强行卸载有未处理I/O操作的卷的功能。

NtQueryInformationFile是一个用于查询目录中指定文件的信息的系统调用。还有一个与它相对应的系统调用: NtSetInformationFile。这些接口用于访问和修改文件的各种相关信息, 如文件名, 类似于加密、压缩、稀疏等文件特征, 其他文件属性和详细资料, 包括查询内部文件ID或给文件分配一个唯一的二进制名称 (对象ID)。

这些系统调用本质上是特定于文件的ioctl的一种形式。这组操作可以用来重命名或删除一个文件。但是请注意，它们处理的并不是文件名，所以要重命名或删除一个文件之前必须先打开这个文件。它们也可以被用来重新命名NTFS上的交换数据流（见11.8节）。

存在独立的API（NtLockFile和NtUnlockFile）用来设置和删除文件中字节域的锁。通过使用共享模式，NtCreateFile允许访问被限制的整个文件。另一种是这些锁API，它们用来强制访问文件中受限制的字节域。读操作和写操作必须提供一个与提供给NtLockFile的钥匙相符合的密钥，以便操作被锁定的区域。

UNIX中也有类似的功能，但在UNIX中应用程序可以自由决定是否认同这个区域锁。NtFsControlFile和前面提到的查询和设置操作很相像，但它是一个旨在处理特定文件的操作，其他的API并不适合处理这种文件。例如，有些操作只针对特定的文件系统。

最后，还有一些其他的系统调用，比如NtFlushBuffersFile。像UNIX的sync系统调用一样，它强制把文件系统数据写回到磁盘。NtCancelIoFile用于取消对一个特定文件的外部I/O请求，NtDeviceIoControlFile实现了对设备的ioctl操作。它的操作清单实际上比ioctl更长。有一些系统调用用于按文件名删除文件，并查询特定文件的属性——但这些操作只是由上面列出的其他I/O管理器操作包装而成的。在这里，我们虽然列出，但并不是真的要把它们实现成独立的系统调用。还有一些用于处理I/O完成端口的系统调用，Windows的队列功能帮助多线程服务器提高使用异步I/O操作的效率，主要通过按需准备线程并降低在专用线程上服务I/O所需要的上下文切换数目来实现。

11.7.3 I/O实现

Windows I/O系统由即插即用服务、电源管理器、I/O管理器和设备驱动模型组成。即插即用服务检测硬件配置上的改变并且为每个设备创建或拆卸设备栈，也会引起设备驱动程序的装载和卸载。功耗管理器会调节I/O设备的功耗状态，以在设备不用的时候降低系统功耗。I/O管理器为管理I/O内核对象以及如IoCallDrivers和IoCompleteRequest等基于IRP的操作提供支持。但是，支持Windows I/O所需要的大部分工作都由设备驱动程序本身实现。

1. 设备驱动程序

为了确保设备驱动程序能和Windows Vista的其余部分协同工作，微软公司定义了设备驱动程序需要符合的WDM（Windows驱动程序模型）。WDM被设计成能在Windows 98系统上运行，也能在从Windows 2000开始的基于NT的系统上运行。WDM允许开发人员编写与两类系统都兼容的驱动程序。微软公司还提供了用于帮助驱动程序开发人员编写符合模型的驱动程序的开发工具箱（Windows 驱动程序开发工具箱）。大部分Windows驱动程序的开发过程都是先复制一份合适的简单的驱动程序，然后修改它。

微软公司也提供一个驱动程序验证器，用以验证驱动程序的多个行为以确保驱动程序符合Windows驱动程序模型的结构要求和I/O请求的协议要求、内存管理等。操作系统中带有此验证器，管理员可能通过运行verifier.exe来控制驱动程序验证器，验证器允许管理员配置要验证哪些驱动程序以及在怎样的范围（多少资源）内验证这些驱动程序。

即使有所有的驱动程序开发和验证支持，在Windows中写一个简单的驱动程序仍然是非常困难的事情，因此微软建立了一个叫做WDF（Windows驱动程序基础）的包装系统，它运行在WDM顶层，简化了很多更普通的需求，主要和驱动程序与电源管理和即插即用操作之间的正确交互有关。

为了进一步简化编写驱动程序，也为了提高系统的健壮性，WDF包含UMDF（用户模式驱动程序架构），使用UMDF编写的驱动程序作为在进程中执行的服务。还有KMDF（内核模式驱动程序架构），使用KMDF编写的驱动程序作为在内核中执行的服务，但是也使得WDM中的很多细节变得不可预料。由于底层是WDM，并且WDM提供了驱动程序模型，因此，本节将主要关注WDM。

在Windows中，设备是由设备对象描述的。设备对象也用于描述硬件（例如总线），软件抽象（例如文件系统、网络协议），还可以描述内核扩展（例如病毒过滤器驱动程序）。上面提到的这些设备对象都是由Windows中的设备栈来组织的，见前面的图11-16。

I/O操作从I/O管理器调用可执行API IoCallDriver程序开始，IoCallDriver带有指向顶层设备对象和

描述I/O请求的IRP的指针。这个例程可以找到与设备对象联合在一起的驱动程序。在IRP中指定操作类型通常都符合前面讲过的I/O管理器系统调用，例如创建、读取和关闭。

图11-38表示的是一个设备栈在单独一层上的关系。驱动程序必须为每个操作指定一个进入点。IoCallDriver从IRP中获取操作类型，利用在当前级别的设备栈中的设备对象来查找指定的驱动程序对象，并且根据操作类型索引到驱动程序分派表去查找相应驱动程序的进入点。最后会把设备对象和IRP传递给驱动程序并调用它。

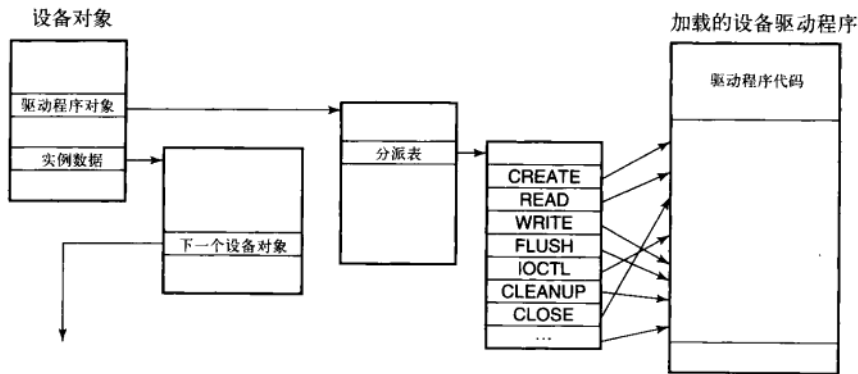


图11-38 设备栈中的单独一层

一旦驱动程序完成处理IRP描述的请求后，它将有三种选择。第一，驱动程序可以再一次调用IoCallDriver，把IRP和设备栈中的下一个设备对象传递给相应的驱动程序。第二，驱动程序也可以声明I/O请求已经完成并返回到调用者。第三，驱动程序还可以在内部对IRP排队并返回到调用者，同时声明I/O请求仍未处理。后一种情况下，如果栈上的所有驱动都认可挂起行为且返回各自的调用者，则会引起一次异步I/O操作。

2. I/O请求包

图11-39表示的是IRP中的主要的域。IRP的底部是一个动态大小的数组，包含那些被设备栈管理请求的域，每个驱动程序都可以使用这些域。在完成一次I/O请求的时候，这些设备栈的域也允许驱动程序指定要调用哪个例程。在完成请求的过程中，按倒序访问设备栈的每一级，并且依次调用由每个应用程序指定的完成例程。在每一级，驱动程序可以继续执行以完成请求，也可以因为还有更多的工作要做从而决定让请求处于未处理状态并且暂停I/O的完成。

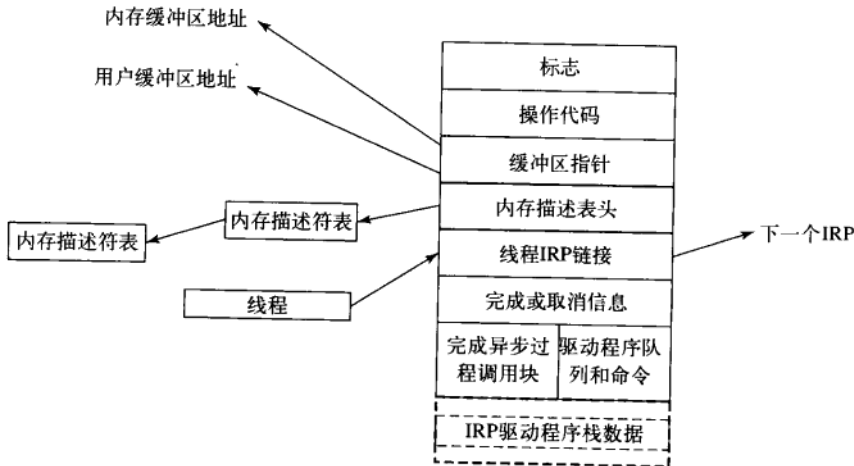


图11-39 I/O 请求包的主要域

当I/O管理器分配一个IRP时，为了分派一个足够大的IRP，它必须知道这个设备栈的深度。在建立设备栈的时候，I/O管理器会在每一个设备对象的域中记录栈的深度。注意，在任何栈中都没有正式地定义下一个设备对象是什么。这个信息被保存在栈中当前驱动程序的私有数据结构中。事实上这个栈实际上并不一定是一个真正的栈。在每一层栈中，驱动程序都可以自由地分配新的IRP，或者继续使用原来的IRP，或者发送一个I/O操作给另一个设备栈，或者甚至转换到一个系统工作线程中继续执行。

IRP包含标志位、索引到驱动程序分派表的操作码、指向内核与用户缓冲区的指针和一个MDL（内存描述符列表）列表。MDL用于描述由缓冲区描述的物理内存框，也就是用于DMA操作。有一些域用于取消和完成操作。当I/O操作已经完成后，在处理IRP时用于排列这个IRP到设备中的域会被重用。目的是给用于在原始线程的上下文中调用I/O管理器的完成例程的APC控制对象提供内存。还有一个连接域用于连接所有的外部IRP到初始化它们的线程。

3. 设备栈 (Device Stack)

Windows Vista中的驱动程序可以自己完成所有的任务，如图11-40所示的打印机驱动程序。另一方面，驱动程序也可以堆叠起来，即一个请求可以在一组驱动程序之间传递，每个驱动程序完成一部分工作。图11-40也给出了两个堆叠的驱动程序。

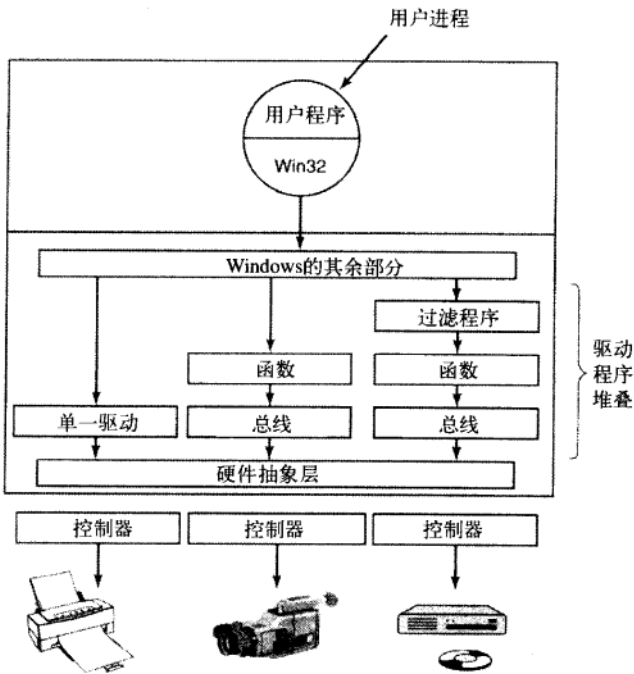


图11-40 Windows允许驱动程序堆叠起来操作设备。这种堆叠是通过设备对象 (Device Object) 来表示的

堆叠驱动程序的一个常见用途是将总线管理与控制设备的功能性工作分离。因为要考虑多种模式和总线事务，PCI总线上的总线管理相当复杂。通过将这部分工作与特定于设备的部分分离，驱动程序开发人员就可以从学习如何控制总线中解脱出来了。他们只要在驱动栈中使用标准总线驱动程序就可以了。类似地，USB和SCSI驱动程序都有一个特定于设备的部分和一个通用部分。Windows为其中的通用部分提供了公共的驱动程序。

堆叠设备驱动程序的另一个用途是将过滤器驱动程序 (filter driver) 插入到驱动栈中。我们已经讨论过文件系统过滤器驱动程序的使用了，该驱动程序插入到文件系统之上。过滤器驱动程序也用于管理物理硬件。在IRP沿着设备栈 (Device Stack) 向下传递的过程中，以及在完成操作 (completion operation) 中IRP沿着设备栈中各个设备驱动程序指定的完成例程 (completion routine) 向上传递的过

程中, 过滤器驱动程序会对所要进行的操作进行变换。例如, 一个过滤器驱动程序能够在将数据存放到磁盘上时对数据进行压缩, 或者在网络传输前对数据进行加密。将过滤器放在这里意味着应用程序和真正的设备驱动程序都不必知道过滤器的存在, 而过滤器会自动对进出设备的数据进行处理。

内核态设备驱动程序是影响Windows的可靠性和稳定性的严重问题。Windows中大多数内核崩溃都是由设备驱动程序出错造成的。因为内核态设备驱动程序与内核及执行层使用相同的地址空间, 驱动程序中的错误可能破坏内核数据结构, 甚至更糟。其中的有些错误之所以产生, 部分原因是为Windows编写的设备驱动程序的数量极其庞大, 部分原因是设备驱动程序由缺乏经验的开发者编写。当然, 为了编写一个正确的驱动程序而涉及的大量设备细节也是造成驱动程序错误的原因。

I/O模型是强大而且灵活的, 但是几乎所有的I/O都是异步的, 因此系统中会大量存在竞态条件(race condition)。从Win9x系统到基于NT技术的Windows系统, Windows 2000首次增加了即插即用(的功能)和电源管理设施。这对要正确地操纵在处理I/O包过程中涉及的驱动器的驱动程序提出了很多要求。PC机用户常常插上/拔掉设备, 把笔记本电脑合上盖子装入公文包, 而通常不考虑设备上那个小绿灯是否仍然亮着(表示设备正在与系统交互)。编写在这样的环境下能够正确运行的设备驱动程序是非常具有挑战性的, 这也是开发WDF(Windows Driver Foundation)以简化Windows驱动模型的原因。

电源管理器集中管理整个系统的电源使用。早期的电源管理包括关闭显示器和停止磁盘旋转以降低电源消耗。但是, 我们需要延长笔记本电脑在电池供电情况下的使用时间。我们还会涉及长时间无人看管运行的桌面计算机的电源节约, 以及节省为现今存在的巨大的服务器群提供能源的昂贵花费(像微软、Google这样的公司将服务器群建在水电站附近以降低费用)。当我们面临以上问题时, 情况迅速变得复杂起来。

更新一些的电源管理设施可以在系统没有被使用的时候, 通过切换设备到后备状态甚至通过使用软电源开关(soft power switch)将设备完全关闭来降低部件功耗。在多处理器中, 可以通过关闭不需要的CPU和降低正在运行的CPU的频率来减少功耗。当一个处理器空闲的时候, 由于除了等待中断发生之外, 该处理器不需要做任何事情, 它的功耗也相应减少了。

Windows支持一种特殊的关机模式——休眠, 该模式将物理内存复制到磁盘, 然后把电力消耗降低到很低的水平(笔记本电脑在休眠状态下可以运行几个星期), 电池的消耗也变得十分缓慢。因为所有的内存状态都被写入磁盘, 我们甚至可以在笔记本电脑休眠的时候为其更换电池。从休眠状态重新启动时, 系统恢复已保存的内存状态并重新初始化设备。这样计算机就恢复到休眠之前的状态, 而不需要重新登录, 也不必重新启动所有休眠前正在运行的应用程序和服务。尽管Windows设法优化这个过程(包括忽略在磁盘中已备份而在内存中未被修改的页面及压缩其他内存页面以减少对I/O操作的需求), 对于一个有几个GB内存的笔记本电脑或桌面机来说, 仍然需要花费数秒钟的时间来进入休眠状态。

另一种可选择的模式是待机模式, 电源管理器将整个系统降到最低的功率状态, 仅使用足够RAM刷新的功率。因为不需要将内存复制到磁盘, 进入待机状态比进入休眠状态的速度更快。但是待机状态不像休眠状态那么可靠。因为如果在待机状态遇到桌面机掉电, 笔记本电脑更换电池, 或者由于驱动程序故障使得设备切换到低功耗状态后无法重新初始化等情况, 系统将无法恢复到待机前的状态。在开发Windows Vista的过程中, 微软和很多硬件设备厂商合作, 花费了极大的努力改进待机模式的操作。他们也终止了允许应用软件禁止系统进入待机模式这一习惯(有时疏忽的用户没有等到指示灯熄灭就把笔记本电脑放进公文包, 从而导致笔记本电脑过热)。

有很多关于WDM(Windows Driver Model)和WDF(Windows Driver Foundation)的有用的书(Cant, 2005; Oney, 2002; Orwick & Smith, 2007; Viscarola 等人, 2007)。

11.8 Windows NT 文件系统

Windows Vista支持若干种文件系统, 其中最重要的是FAT-16、FAT-32和NTFS(NT文件系统)。FAT-16是MS-DOS文件系统, 它使用16位磁盘地址, 这就限制了它使用的磁盘分区不能大于2GB。现在, 这种文件系统基本上仅用来访问软盘。FAT-32使用32位磁盘地址, 最大支持2TB的磁盘分区。FAT32没有任何安全措施, 现在我们只在可移动介质(如闪存)中使用它。NTFS是一个专门为Windows NT开发的文件系统。从Windows XP开始, 计算机厂商把它作为默认安装的文件系统, 这极大地提升了

Windows的安全性和功能。NTFS使用64位磁盘地址并且（理论上）能够支持最大 2^{64} 字节的磁盘分区，尽管还有其他因素会限制磁盘分区大小。

因为NTFS文件系统是一个带有很多有趣的特性和创新设计的现代文件系统，在本章中我们将针对NTFS文件系统进行讨论。NTFS是一个大而且复杂的文件系统；由于篇幅所限，我们不能讨论其所有的特性，但是接下来的内容会使读者对它印象深刻。

11.8.1 基本概念

NTFS限制每个独立的文件名最多由255个字符组成；全路径名最多有32 767个字符。文件名采用Unicode编码，允许非拉丁语系国家的用户（如希腊、日本、印度、俄罗斯和以色列）用他们的母语为文件命名。例如， $\phi\tau\lambda\epsilon$ 就是一个完全合法的文件名。NTFS完全支持区分大小写的文件名（所以foo与Foo和FOO是不同的）。Win32 API不完全支持区分大小写的文件名，并且根本不支持区分大小写的目录名。为了保持与UNIX系统的兼容，当运行POSIX子系统时，Windows提供区分大小写的支持。Win32不区分大小写，但是它保持大小写状态，所以文件名可以包含大写字母和小写字母。尽管区分大小写是一个UNIX用户非常熟悉的特性，但是对一般用户而言，这是很不方便的。例如，现在的互联网在很大程度上是不区分大小写的。

与FAT32和UNIX文件不同，NTFS文件并不只是字节的一个线性序列，而是一个文件由很多属性组成，每个属性由一个字节流表示。大部分文件都包含一些短字节流（如文件名和64位的对象ID），和一个包含数据的未命名的长字节流。当然，一个文件也可以有两个或多个数据流（即长字节流）。每个流有一个由文件名、一个冒号和一个流名组成的名字，例如，foo:stream1。每个流有自己的大小，并且相对于所有其他的流都是可以独立锁定的。一个文件中存在多个流的想法在NTFS中并不新鲜。苹果Macintosh的文件系统为每个文件使用两个流，一个数据分支（data fork）和一个资源分支（resource fork）。NTFS中多数据流的首次使用是为了允许一个NT文件服务器为Macintosh用户提供服务。多数据流也用于表示文件的元数据，例如Windows GUI中使用的JPEG图像的缩略图。但是，多数据流很脆弱，并且在传输文件到其他文件系统，通过网络传输文件甚至在文件备份和后来恢复的过程中都会丢失文件。这是因为很多工具都忽略了它们。

与UNIX文件系统类似，NTFS是一个层次化的文件系统。名字的各部分之间用“\”分隔，而不是“/”，这是从MS-DOS时代与CP/M相兼容的需求中继承下来的。与UNIX中当前工作目录的概念不同的是，作为文件系统设计的一个基础部分的链接到当前目录（.）和父目录（..）的硬连接，在Windows是作为一种惯例来是实现的。系统仅在其中的POSIX子系统里支持硬连接，正因为这样，NTFS支持对目录的遍历检查（UNIX中的“x”权限）。

从Windows Vista开始，NTFS才开始支持符号链接。为了避免如Spoofing这样的安全问题（当年在UNIX 4.2BSD第一次引入符号链接时就遇到过），通常只允许系统管理员来创建符号链接。在Vista中符号链接的实现用到一个叫再解析点（reparse points）的NTFS特性（将在本节后续部分讨论）。另外，NTFS也支持压缩、加密、容错、日志和稀疏文件。我们马上就会探讨这些特性及其实现。

11.8.2 NTFS文件系统的实现

NTFS文件系统是专门为NT系统开发的，用来替代OS/2中的HPFS文件系统的。它是一个具有很高复杂性和精密性的文件系统。NT系统的大部分是在陆地上设计的。从这方面看，NTFS与NT系统其他部分相比是独一无二的，因为它的很多最初设计都是在一艘驶出普吉特湾的帆船的甲板上完成的（严格遵守上午工作，下午喝啤酒的作息协议）。

接下来，我们将从NTFS结构开始，探讨一系列NTFS特性，包括文件名查找、文件压缩、日志和加密。

1. 文件系统结构

每个NTFS卷（如磁盘分区）都包含文件、目录、位图和其他数据结构。每个卷被组织成磁盘块的一个线形序列（在微软的术语中叫“簇”），每个卷中块的大小是固定的。根据卷的大小不同，块的大小从512字节到64KB不等。大多数NTFS磁盘使用4KB的块，作为有利于高效传输的大块和有利于减少内部碎片的小块之间的折中办法。每个块用其相对于卷起始位置的64位偏移量来指示。

每个卷中的主要数据结构叫MFT（主文件表，Master File Table），该表是以1KB为固定大小的记录

的线形序列。每个MFT记录描述一个文件或目录。它包含了如文件名、时间戳、文件中的块在磁盘上的地址的列表等文件属性。如果一个文件非常大,有时候会需要两个或更多的MFT记录来保存所有块的地址列表。这时,第一个MFT记录叫做基本记录(base record),该记录指向其他的MFT记录。这种溢出方案可以追溯到CP/M,那时每个目录项称为一个范围(extent)。用一个位图记录哪个MFT表项是空闲的。

MFT本身就是一个文件,可以被放在卷中的任何位置,这样就避免了在第一磁道上出现错误扇区引起的问题。而且MFT可以根据需要变大,最大可以有 2^{48} 个记录。

图11-41是一个MFT。每个MFT记录由数据对(属性头,值)的一个序列组成。每个属性由一个说明了该属性是什么和属性值有多长的头开始。一些属性值是变长的,如文件名和数据。如果属性值足够短能够放到MFT记录中,那么就把它放到记录里。这叫做直接文件(immediate file, [Mullender and Tanenbaum, 1984])。如果属性值太长,它将被放在磁盘的其他位置,并在MFT记录里存放一个指向它的指针。这使得NTFS对于小的域(即那些能够放入MFT记录中的域)非常有效。

最开始的16个MFT记录为NTFS元数据文件而预留,如图11-41所示。每一个记录描述了一个正常的具有属性和数据块的文件,就如同其他文件一样。这些文件中每一个都由“\$”开始表明它是一个元数据文件。第一个记录描述了MFT文件本身。它说明了MFT文件的块都放在哪里以确保系统能找到MFT文件。很明显,Windows需要一个方法找到MFT文件中第一个块,以便找到其余的文件系统信息。找到MFT文件中第一个块的方法是查看启动块,那是卷被格式化为文件系统时地址所存放的位置。

记录1是MFT文件早期部分的复制。这部分信息非常重要,因此拥有第二份拷贝至关重要以防MFT的第一块坏掉。记录2是一个Log文件。当对文件系统做结构性的改变时,例如,增加一个新目录或删除一个现有目录,动作在执行前就记录在Log里,从而增加在这个动作执行时出错后(比如一次系统崩溃)被正确恢复的机会。对文件属性做的改变也会记录在这里。事实上,唯一不会记录的改变是对用户数据的改变。记录3包含了卷的信息,比如大小、卷标和版本。

上面提到,每个MFT记录包含一个(属性头,值)数据对的序列。属性在\$AttrDef文件中定义。这个文件的信息在MFT记录4里。接下来是根目录,根目录本身是一个文件并且可以变为任意长度。MFT记录5用来描述根目录。

卷里的空闲空间通过一个位图来跟踪。这个位图本身是一个文件,它的磁盘地址和属性由MFT记录6给出。下一个MFT记录指向引导装载程序。记录8用来把所有的坏块链接在一起来确保不会有文件使用它们。记录9包含安全信息;记录10用于大小写映射。对于拉丁字母A-Z,映射是非常明确的(至少是对说拉丁语的人来说)。对于其他语言的映射,如希腊、亚美尼亚或乔治亚,就对于讲拉丁语的人不太明确,因此这个文件告诉我们如何做。最后,记录11是一个目录包含杂项文件用于磁盘配额、对象标识符、再解析点,等等。最后四个MFT记录被留作将来使用。

每个MFT记录由一个记录头和后面跟着的(属性头,值)对组成。记录头包含一个幻数用于有效性检查,一个序列号(每次当记录被一个新文件再使用时就被更新),文件引用记数,记录实际使用的字节数,基本记录(仅用于扩展记录)的标识符(索引,序列号),和其他一些杂项。

NTFS定义了13个属性能够出现在MFT记录中。图11-42列出了这些属性。每个属性头标识了属性,给出了长度,值字段的位置,一些各种各样的标记和其他信息。通常,属性值直接跟在它们的属性头后面,但是如果一个值对于一个MFT记录太长的话,它可能被放在不同的磁盘块中。这样的属性称作非常

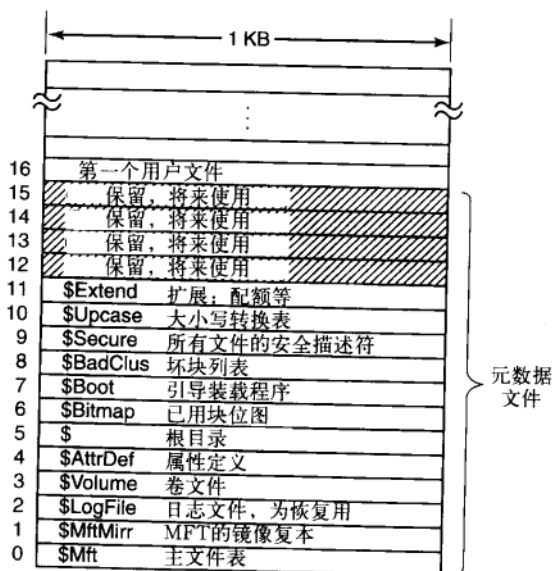


图11-41 NTFS主文件表

驻属性，数据属性很明显就是这样一个属性。一些属性，像名字，可能出现重复，但是所有属性必须在MFT记录中按照固定顺序出现。常驻属性头有24个字节长；非常驻属性头会更长，因为它们包含关于在磁盘上哪些位置能找到这些属性的信息。

标准的信息域包含文件所有者、安全信息、POSIX需要的时间戳、硬连接计数、只读和存档位，等等。这些域是固定长度的，并且总是存在的。文件名是一个可变长度Unicode编码的字符串。为了使具有非MS-DOS文件名的文件可以访问老的16位程序，文件也可以有一个符合8+3规则的MS-DOS短名字。如果实际文件名符合8+3命名规则，第二个MS-DOS文件名就不需要了。

在NT4.0中，安全信息被放在一个属性中，但在Windows 2000及以后的版本中，安全信息全部都放在一个单独的文件中使得多个文件可以共享相同的安全描述。由于安全信息对于每个用户的许多文件来说是相同的，于是这使得许多MFT记录和整个文件系统节省了大量的空间。

当属性不能全部放在MFT记录中时，就需要使用属性列表。这个属性就会说明在哪里找到扩展记录。列表中的每个条目在MFT中包含一个48位的索引来说明扩展记录在哪里，还包含一个16位的序号来验证扩展记录与基本记录是否匹配。

就像UNIX文件拥有一个i节点号一样，NTFS文件也有一个ID。文件可以依据ID被打开，但是由于ID是基于MFT记录的，并且可以因该文件的记录移动（例如，如果文件因备份被恢复）而改变，所以当ID必须保持不变时，这个NTFS分配的ID并不总是有用。NTFS允许有一个可以设置在文件上而且永远不需要改变的独立对象ID属性。举例来说，当一个文件被拷贝到一个新卷时，这个属性随着文件一起过去。

重解析点告诉分析文件名的过程来做特别的事。这个机制用于显式加载文件系统和符号链接。两个卷属性用于标示卷。随后三个属性处理如何实现目录——小的目录就是文件列表，大的目录使用B+树实现。日志工具流属性用来加密文件系统。

最后，我们关注最重要的属性：数据流（在一些情况下叫流）。一个NTFS文件有一个或多个数据流，这些就是负载所在。默认数据流是未命名的（例如，目录路径\文件名::\$DATA），但是替代数据流有自己的名字，例如：目录路径\文件名:流名::\$DATA。

对于每个流，流的名字（如果有）会在属性头中。头后面要么是说明了流包含哪些块的磁盘地址列表，要么是仅几百字节大小的流（有许多这样的流）本身。存储了实际流数据的数据流记录称为立即文件（Mullender和Tanenbaum，1984）。

当然，大多数情况下，数据放不进一个MFT记录中，因此这个属性通常是非常驻属性。现在让我们看一看NTFS如何记录特殊数据中非常驻属性的位置。

2. 存储分配

保持对磁盘上在可能的情况下，连续分配的块进行跟踪的模型，这是出于效率的原因。举例来说，如果一个流的第一个逻辑块放在磁盘上的块20，那么系统将努力把第二个逻辑块放在块21，第三个逻辑块放在块22，以此类推，实现这些行串的一个方法是尽可能一次分配许多磁盘块。

一个流中的块是通过一串记录描述的，每个记录描述了一串逻辑上连续的块，对于一个没有孔的流来说，只有唯一的一个记录。按从头到尾的顺序写的流都属于这一类。对于一个包含一个孔的流（例如，只有块0~49和块60~79被定义了），会有两个记录。这样的流会产生于先写入前50个块，然后找到逻辑上第60块，然后写其他20个块。当孔被读出时，用全零表示。有孔的文件称为稀疏文件。

每个记录始于一个头，这个头给出第一个块在流中偏移量。接着是没有被记录覆盖的第一个块的偏

属 性	描 述
标准信息	标志位，时间戳等
文件名	Unicode文件名，可能重复用做MS-DOS格式名
安全描述符	废弃了。安全信息现在用\$Extend \$Secure表示
属性列表	额外的MFT记录的位置，如果需要的话
对象ID	对此卷唯一的64位文件标识符
重解析点	用于加载和符号链接
卷名	当前卷的名字（仅用于\$Volume）
卷信息	卷版本（仅用于\$Volume）
索引根	用于目录
索引分配	用于很大的目录
位图	用于很大的目录
日志工具流	控制记录日志到\$LogFile
数据	数据流，可以重复

图11-42 MFT记录中使用的属性

移量。在上面的例子中，第一个记录有一个 (0, 50) 的头，并会提供这50个块的磁盘地址。第二个记录有一个 (60, 80) 的头，会提供其他20个块的磁盘地址。

每个记录的后面跟着一个或多个对，每个对给出了磁盘地址和持续长度。磁盘地址是该磁盘块离本分区起点的偏移量；游程在行串中块的数量。在一段行串记录中需要有多少对就可以有多少对。图 11-43描述了用这种方式表示的三段、9块的流。

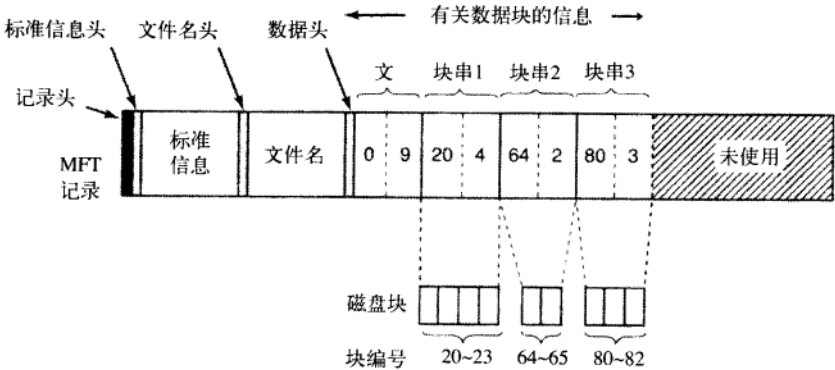


图11-43 有3个连续空间、9个块的短流的一条MFT记录

在这个图中，有一个9个块（头，0~8）的短流的MFT记录。它由磁盘上三个行串连续块组成。第一段是块20~23，第二段是块64~65，第三段是80~82。每一个行串被记录在MFT记录中的一个（磁盘地址，块计数）对中。有多少行串是依赖于当流被创建时磁盘块分配器在找连续块的行串时做的有多好。对于一个n块的流，段数可能是从1到n的任意值。

有必要在这里做几点说明：

首先，用这种方法表达的流的大小没有上限限制。在地址不压缩的情况下，每一对需要两个64位数表示，总共16字节。然而，一对能够表示100万个甚至更多的连续的磁盘空间。实际上，20M的流包含20个独立的包含100万个1KB的块的行串，每个都可以轻易地放在一个MFT记录中，然而一个60KB的被分散到60个不同的块的流却不行。

其次，表示每一对的直截了当的方法会占用2×8个字节，有压缩方法可以把一对的大减小到低于16字节。许多磁盘地址有多个高位0字节。这些可以被忽略。数据头能告诉我们要多少个高位0字节被忽略了，也就是说，在一个地址中实际上有多少个字节被用。也可以用其他的压缩方式。实际上，一对经常只有4个字节。

第一个例子是比较容易的：所有的文件信息能容纳在一个MFT记录中，如果文件比较大或者是高度碎片化以至于信息不能放在一个MFT记录当中，这时会发生什么呢？答案很简单：用两个或更多的MFT记录。从图11-44可以看出，一个文件的首MFT记录是102，对于一个MFT记录而言它有太多的行串，因而它会计算需要多少个扩展的MFT记录。比如说两个，于是会把它们的索引放到首记录中，首记录剩余的空间用来放前k个行串。

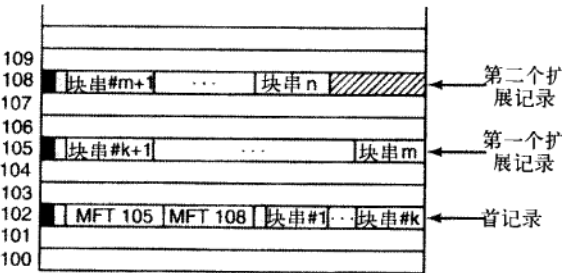


图11-44 需要三个MTF记录存储其所有行串的文件

注意，图11-44包含了一些多余的信息。理论上不需要指出一串行串的结尾，因为这些信息可以从行串中对中计算出来。列出这些信息是为了更有效地搜索：找到在一个给定文件偏移量的块，只需要去检查记录头，而不是行串对。

当MFT记录102中所有的空间被用完后，剩余的行串继续在MFT记录105中存放，并在这个记录中

放入尽可能多的项。当这个记录也用完后，剩下的行串放在MFT记录108中。这种方式可以用多个MFT记录去处理大的分段存储文件。

有可能出现这样的问题：如果文件需要的MFT记录太多，以至于首个MTF记录中没有足够的空间去存放所有的索引。解决问题的方法是：使扩展的MFT记录列表成为非驻留的（即：存放在其他的硬盘区域而不是在首MFT记录中），这样它就能根据需要而增大。

图11-45表示一个MFT表项如何描述一个小目录。这个记录包含若干目录项，每一个目录项可以描述一个文件或目录。每个表项包含一个定长的结构体和紧随其后的不定长的文件名。定长结构体包含该文件对应的MFT表项的索引、文件名长度以及其他的属性和标志。在目录中查找一个目录项需要依次检查所有的文件名。

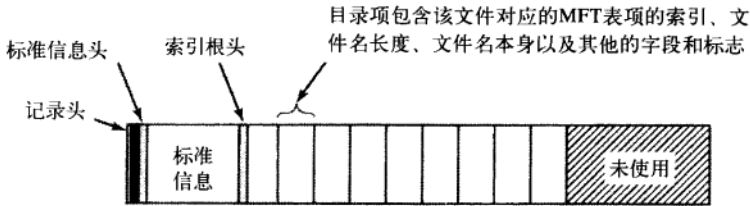


图11-45 描述小目录的MFT记录

大目录采用一种不同的格式，即用B+树而不是线性结构来列出文件。通过B+树可以按照字母顺序查找文件，并且更容易在目录的正确位置插入新的文件名。

现在有足够的信息去描述使用文件名对文件\??\C:\foo\bar的查找是如何进行的。从图11-22可以知道Win32、原生NT系统调用、对象和I/O管理器如何协作通过向C盘的NTFS设备栈（device stack）发送I/O请求打开一个文件。I/O请求要求NTFS为剩余的路径名\foo\bar填写一个文件对象。

NTFS从C盘根目录开始分析\foo\bar路径，C盘的块可以在MFT中的第五个表项中找到（参考图11-41）。然后在根目录中查找字符串“foo”，返回目录foo在MFT中的索引，接着再查找字符串“bar”，得到这个文件的MFT记录的引用。NTFS通过调用安全引用管理器来实施访问检查，如果所有的检查都通过了，NTFS从MFT记录中搜索得到::\$DATA属性，即默认的数据流。

找到文件bar后，NTFS在I/O管理器返回的文件对象上设置指针指向它自己的元数据。元数据包括指向MFT记录的指针、压缩和范围锁、各种关于共享的细节等。大多数元数据包含在一些数据结构中，这些数据结构被所有引用这个文件的文件对象共享。有一些域是当前打开的文件特有的，比如当这个文件被关闭时是否需要删除。一旦文件成功打开，NTFS调用IoCompleteRequest，它通过把IPR沿I/O栈向上传递给I/O和对象管理器。最终，这个文件对象的句柄被放进当前进程的句柄表中，然后回到用户态。之后调用ReadFile时，应用程序能够提供句柄，该句柄表明C:\foo\bar文件对象应该包含在传递到C：设备栈给NTFS的读请求中。

除了支持普通文件和目录外，NTFS支持像UNIX那样的硬连接，也通过一个叫做重解析点的机制支持符号链接。NTFS支持把一个文件或者目录标记为一个重解析点，并将其和一块数据关联起来。当在文件名解析的过程中遇到这个文件或目录时，操作就会失败，这块数据被返回到对象管理器。对象管理器将这块数据解释为另一个路径名，然后更新需要解析的字符串，并重启I/O操作。这种机制用来支持符号链接和挂载文件系统，把文件搜索重定向到目录层次结构的另外一个部分甚至到另外一个不同的分区。

重解析点也用来为文件系统过滤器驱动程序而标记个别文件。在图11-22中显示了文件系统过滤器如何安装到I/O管理器和文件系统之间。I/O请求通过调用IoCompleteRequest来完成，其把控制权转交给在请求发起时设备栈上每个驱动程序插入到IRP中的完成例程。需要标记一个文件的驱动程序首先关联一个重解析标签，然后监控由于遇到重解析点而失败的打开文件操作的完成请求。通过用IRP传回的数据块，驱动程序可以判断出这是否是一个驱动程序自身关联到该文件的数据块。如果是，驱动程序将停止处理完成例程而接着处理原来的I/O请求。通常这将引发一个打开请求，但这时将有一个标志告诉NTFS忽略重解析点并同时打开文件。

3. 文件压缩

NTFS支持透明的文件压缩。一个文件能够以压缩方式创建,这意味着当向磁盘中写入数据块时NTFS会自动尝试去压缩这些数据块,当这些数据块被读取时NTFS会自动解压。读或写的进程完全不知道压缩和解压在进行。

压缩流程是这样的:当NTFS写一个有压缩标志的文件到磁盘时,它检查这个文件的前16个逻辑块,而不管它们占用多少个项,然后对它们运行压缩算法,如果压缩后的数据能够存放在15个甚至更少的块中,压缩数据将写到硬盘中;如果可能的话,这些块在一个行串里。如果压缩后的数据仍然占用16个块,这16个块以不压缩方式写到硬盘中。之后,去检查第16-31块看是否能压缩到15个甚至更少的块,以此类推。

图11-46a显示一个文件。该文件的前16块被成功地压缩到了8个,对第二个16块的压缩没有成功,第三个16块也压缩了50%。这三个部分作为三个行串来写,并存储于MFT记录中。“丢失”的块用磁盘地址0存放在MFT表项中,如图11-46b所示。在图中,头(0,48)后面有五个二元组,其中,两个对应着第一个(被压缩)行串,一个对应没有压缩的行串,两个对应最后一个(被压缩)行串。

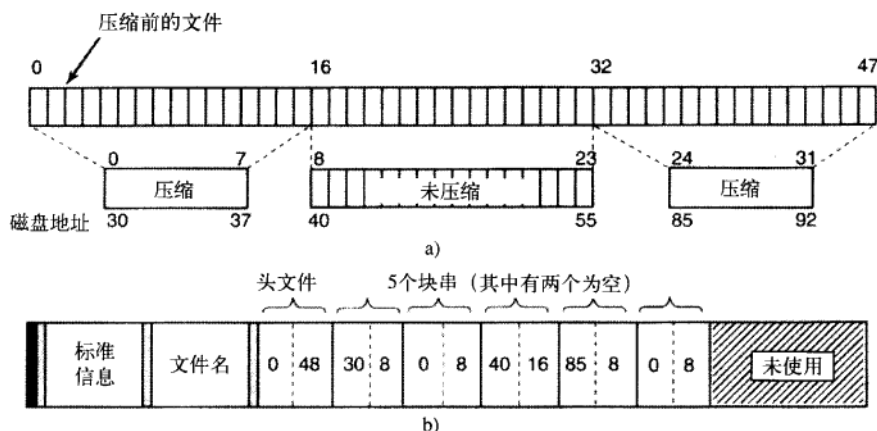


图11-46 a) 一个占48块的文件被压缩到32块的例子; b) 被压缩后文件对应的MFT记录

当读文件时,NTFS需要分辨某个行串是否被压缩过,它可以根据磁盘地址进行分辨,如果其磁盘地址是0,表明它是16个被压缩的块的最后部分。为了避免混淆,磁盘第0块不用于存储数据。同时,因为卷上的第0块包含了引导扇区,用它来存储数据也是不可能的。

随机访问压缩文件也是可行的,但是需要技巧。假设一个进程寻找图11-46中文件的第35块,NTFS是如何定位一个压缩文件的第35块区的呢?答案是NTFS必须首先读取并且解压整个行串,获得第35块的位置,之后就可以将该块传给读取它的进程。选择16个块作为压缩单元是一个折衷的结果,短了会影响压缩效率,长了则会使随机访问开销过大。

4. 日志

NTFS支持两种让程序探测卷上文件和目录变化的机制。第一种机制是调用名为NtNotifyChange Directory File的I/O操作,传递一个缓冲区给系统,当系统探测到目录或者子目录树变化时,该操作返回。这个I/O操作的结果是在缓冲区里填上变化记录的一个列表。缓冲区应该足够大,否则填不下的记录会被丢弃。

第二种机制是NTFS变化日志。NTFS将卷上的目录和文件的变化记录保存到一个特殊文件中,程序可以使用特殊文件系统控制操作来读取,即调用API NtFsControlFile并以FSCTL_QUERY_USN_JOURNAL为参数。日志文件通常很大,而且日志中的项在被检查之前重用的可能性非常小。

5. 文件加密

如今,计算机用来存储很多敏感数据,包括公司收购计划、税务信息、情书,数据的所有者不想把这些信息暴露给任何人。但是信息的泄漏是有可能发生的,例如笔记本电脑的丢失或失窃;使用MS-

DOS软盘重起桌面系统来绕过Windows的安全保护；或者将硬盘从计算机里移到另一台安装了不安全操作系统的计算机中。

Windows提供了加密文件的选项来解决这些问题，因此当电脑的失窃或用MS-DOS重启时，文件内容是不可读的。Windows加密的通常方式是将重要目录标识为加密的，然后目录里的所有文件都会被加密，新创建或移动到这些目录来的文件也会被加密。加密和解密不是NTFS自己管理的，而是由EFS (Encryption File System) 驱动程序来管理，EFS作为回调向NTFS注册。

EFS为特殊文件和目录提供加密。在Windows Vista中还有另外一个叫做BitLocker的加密工具，它加密了卷上几乎所有的数据。只要用户使用强密钥来发挥这种机制的优势，任何情况下它都能帮助用户保护数据。考虑到系统丢失或失窃的数量，以及身份泄露的强烈敏感性，确保机密被保护是非常重要的。每天都有惊人数量的笔记本电脑丢失；仅考虑纽约市，华尔街大部分公司平均一周在出租车上丢失一台笔记本电脑。

11.9 Windows Vista中的安全

看过加密后，该从总体上探讨安全问题了。NT的最初设计符合美国国防部 C2 级安全需求 (DoD 5200.28-STD)，该橘皮书是安全的DoD系统必需满足的标准。此标准要求操作系统必需具备某些特性才能认定对特定类型的军事工作是足够安全的。虽然Windows Vista并不是专为满足C2兼容性而设计的，但它从最初的NT安全设计中继承了很多安全特性，包括下面的几个：

- 1) 具有反欺骗措施的安全登录。
- 2) 自主访问控制。
- 3) 特权化访问控制。
- 4) 对每个进程的地址空间保护。
- 5) 新页被映射前必需清空。
- 6) 安全审计。

让我们来简要地回顾一下这些条目。

安全登录意味着系统管理员可以要求所有用户必需拥有密码才可以登录。欺骗是指一个恶意用户编写了一个在屏幕上显示登录提示的程序然后走开以期望一个无辜的用户会坐下来并输入用户名和密码。用户名和密码被写到磁盘中并且用户被告知登陆失败。Windows Vista通过指示用户按下CTRL-ALT-DEL登录来避免这样的攻击。键盘驱动总是可以捕获这个键序列，并随后调用一个系统程序来显示真正的登录屏幕。这个过程可以起作用是因为用户进程无法禁止键盘驱动对CTRL-ALT-DEL的处理。但是NT可以并且确实在某些情况下禁用了CTRL-ALT-DEL安全警告序列。这种想法来自于Windows XP和Windows 2000，用来使NT系统对从Windows 98切换过来的用户保持更多的兼容性。

自主访问控制允许文件或者其他对象的所有者指定谁能以何种方式使用它。特权化访问控制允许系统管理员（超级用户）无需覆盖上述权限设定。地址空间保护仅仅意味着每个进程自己的受保护的虚拟地址空间不能被其他未授权的进程访问。下一个条目意味着当进程的堆增长时被映射进来的页面被初始化为零，这样它就找不到页面以前的所有者所存放的旧信息（参见在图 11-36中为此目的而提供的清零页的列表）。最后，安全审计使得管理员可以获取某些安全相关事件的日志。

橘皮书没有指定当笔记本电脑被盗时将发生什么事情，然而在一个大型组织中每星期发生一起盗窃是很常见的。于是，Windows Vista提供了一些工具，当笔记本被盗或者丢失时，谨慎的用户可以利用它们最小化损失。当然，谨慎的用户正是那些不会丢失笔记本的人——这种麻烦是其他人引起的。

下一章将描述在Windows Vista中基本的安全概念，以及关于安全的系统调用。最后，我们将看看安全是怎样实现的。

11.9.1 基本概念

每个Windows Vista用户（和组）用一个SID (Security ID, 安全ID) 来标识。SID是二进制数字，由一个短的头部后面接一个长的随机部分构成。每个SID都是世界范围内唯一的。当用户启动进程时，进程和它的线程带有该用户的SID运行。安全系统中的大部分地方被设计为确保只有带有授权SID的线程才可以访问对象。

每个进程拥有一个指定了SID和其他属性的访问令牌。该令牌通常由winlogon创建,就像后面说的那样。图11-47展示了令牌的格式。进程可以调用GetTokenInformation来获取令牌信息。令牌的头部包含了一些管理性的信息。过期时间字段表示令牌何时不再有效,但当前并没有使用该字段。组字段指定了进程所隶属的组。POSIX子系统需要该字段。默认的DACL (Discretionary Access Control List, 自主访问控制列表) 会赋给被进程创建的对象,如果没有指定其他ACL的话。用户的SID表示进程的拥有者。受限SID使得不可信的进程以较少的权限参与到可信进程的工作中,以免造成破坏。

最后,权限字段,如果有的话,赋予进程除普通用户外特殊的权利,比如关机和访问本来无权访问的文件的权利。实际上,权限域将超级用户的权限分成几种可独立赋予进程的权限。这样,用户可被赋予一些超级用户的权限,但不是全部的权限。总之,访问令牌表示了谁拥有这个进程和与其关联的权限及默认值。

头部	过期时间	组	默认DACL	用户SID	组SID	受限SID	权限	身份模拟级别	完整度级别
----	------	---	--------	-------	------	-------	----	--------	-------

图11-47 访问令牌结构

当用户登录时, winlogon赋予初始的进程一个访问令牌。后续的进程一般会将这个令牌继承下去。初始时,进程的访问令牌会被赋予其所有的线程。然而,线程在运行过程中可以获得一个不同的令牌,在这种情况下,线程的访问令牌覆盖了进程的访问令牌。特别地,一个客户端线程可以将访问权限传递给服务器线程,从而使得服务器可以访问客户端的受保护的文件和其他对象。这种机制叫做身份模拟 (impersonation)。它是由传输层 (比如ALPC、命名管道和TCP/IP) 实现的、被RPC用来实现从客户端到服务器的通信。传输层使用内核中安全引用监控器组件的内部接口提取出当前线程访问令牌的安全上下文,并把它传送到服务器端来构建用于服务器模拟客户身份的令牌。

另一个基本的概念是安全描述符 (security descriptor)。每个对象都关联着一个安全描述符,该描述符描述了谁可以对对象执行何种操作。安全描述符在对象被创建的时候指定。NTFS文件系统和注册表维护着安全描述符的持久化形式,用以为文件和键对象 (对象管理器中表示已打开的文件和键的实例) 创建安全描述符。

安全描述由一个头部和其后带有一个或多个访问控制入口 (Access Control Entry, ACE) 的DACL组成。ACE主要有两类: 允许项和拒绝项。允许项含有一个SID和一个表示带有此SID的进程可以执行哪些操作的位图。拒绝项与允许项相同,不过其位图表示的是谁不可以执行那些操作。比如, Ida拥有一个文件,其安全描述符指定任何人都可读, Elvis不可访问, Cathy可读可写, 并且Ida自己拥有完全的访问权限。图11-48描述了这个简单的例子。Everyone这个SID表示所有的用户, 但该表项会被任何显式的ACE覆盖。

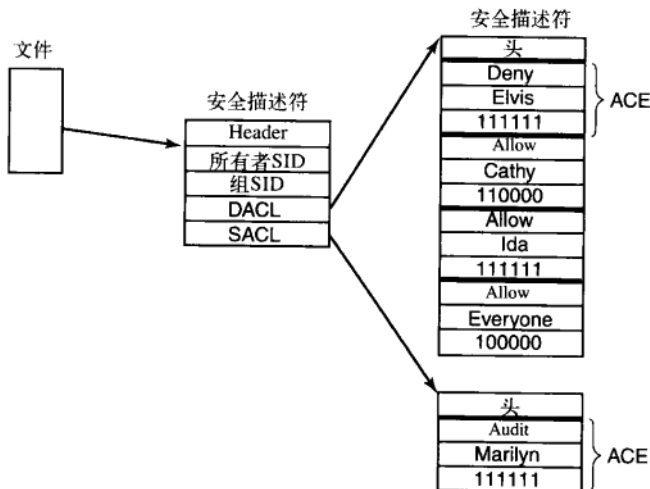


图11-48 文件的安全描述符示例

除DACL外，安全描述符还包含一个系统访问控制列表（System Access Control List，SACL）。SACL跟DACL很相似，不过它表示的并不是谁可以使用对象，而是哪些对象访问操作会被记录在系统范围内的安全事件日志中。在图11-48中，Marilyn对文件执行的任何操作都将会被记录。SACL还包含完整度级别字段，我们将稍后讨论它。

11.9.2 安全相关的API调用

Windows Vista的访问控制机制大都基于安全描述符。通常情况下进程创建对象时会将一个安全描述符作为参数提供给CreateProcess、CreateFile或者其他对象创建调用。该安全描述符就会附属在这个对象上，就如在图11-48中看到的那样。如果没有给创建对象的函数调用提供安全描述符，调用者的访问令牌中默认的安全设置（参见图11-47）将被使用。

大部分Win32 API安全调用跟安全描述符的管理相关，因此在这里主要关注它们。图11-49列出了那些最重要的调用。为了创建安全描述符，首先要分配存储空间，然后调用Initialize Security Descriptor初始化它。该调用填充了安全描述符的头部。如果不知道所有者的SID，可以根据名字用LookupAccountSid来查询。随后SID被插入到安全描述符中。对组SID也一样，如果有的话。通常，这些SID会是调用者自己的SID和它的某一个组SID，不过系统管理员可以填充任何SID。

Win32 API 函数	描 述
InitializeSecurityDescriptor	准备一个新的安全描述符
LookupAccountSid	查询指定用户名的SID
SetSecurityDescriptorOwner	设置安全描述符中的所有者的SID
SetSecurityDescriptorGroup	设置安全描述符中的组SID
InitializeAcl	初始化DACL或者SACL
AddAccessAllowedAce	向DACL或者SACL添加一个允许访问的新ACE
AddAccessDeniedAce	向DACL或者SACL添加一个拒绝访问的新ACE
DeleteAce	从DACL或者SACL删除ACE
SetSecurityDescriptorDacl	使DACL依附到一个安全描述符

图11-49 Win32中基本的安全调用

这时可调用InitializeAcl初始化安全描述符的DACL（或者SACL）。ACL入口项可通过AddAccess AllowedAce和AddAccessDeniedAce。可多次调用这些函数以添加任何所需的ACE入口项。可调用DeleteAce来删除一个入口项，这用来修改已存在的ACL而不是构建一个新的ACL。SetSecurity DescriptorDacl可以把一个准备就绪的ACL与安全描述符关联到一起。最后，当创建对象时，可将新构造的安全描述符作为参数传送使其与这个对象相关联。

11.9.3 安全性的实现

在独立的Windows Vista系统中，安全由大量的组件来实现，我们已经看过了其中大部分组件（网络是完全不同的事情，超出了本书的讨论范围）。登录和认证分别由winlogon和lsass来处理。登录成功后会获得一个带有访问令牌的GUI shell程序（explorer.exe）。这个进程使用注册表中的SECURITY 和 SAM表项。前者设置一般性的安全策略，而后者包含了针对个别用户的安全信息，如11.2.3节讨论的那样。

一旦用户登录成功，每当打开对象进行访问就会触发安全操作。每次OpenXXX调用都需提供正要被打开的对象的名字和所需的权限集合。在打开的过程中，安全引用监控器会检查调用者是否拥有所需的权限。它通过检查调用者的访问令牌和跟对象关联的DACL来执行这种检查。安全监控管理器依次检查ACL中的每个ACE。一旦发现入口项与调用者的SID或者调用者所隶属的某个组相匹配，访问权限即可确定。如果调用者拥有所需的权限，则打开成功；否则打开失败。

正如已经看到的那样，除允许项外，DACL还包括拒绝项。因此，通常把ACL中的拒绝访问的项置于赋予访问权限的项之面，这样一个被特意拒绝访问的用户不能通过作为拥有合法访问权限的组的成员这样的后门获得访问权。

对象被打开后,调用者会获得一个句柄。在后续的调用中,只需检查尝试的操作是否在打开时所申请的集合内,这样就避免了调用者为了读而打开文件然后对该文件进行写操作。另外,正如SACL所要求的那样,在句柄上进行的调用可能会导致产生审计日志。

Windows Vista增加了另外的安全设施来应对使用ACL保护系统的共同问题。进程的令牌中含有新增加的必需的完整性级别(Integrity-Level) SID字段并且对象在SACL中指定了一个完整性级别ACE。完整性级别阻止了对对象的写访问,不管DACL中有何种ACE。特别地,完整性级别方案用来保护系统免受被攻击者控制的Internet Explorer进程(可能用户接受了不妥的建议而从未知的网站下载代码)的破坏。低权限的IE,运行时的完整性级别被设置为低。系统中所有的文件和注册表中的键拥有中级的完整性级别,因此低完整性级别的IE不能修改它们。

近年来Windows增加了很多其他的安全特性。对于Windows XP Service Pack 2来说,系统的大部分在编译时使用了可对多种栈缓冲区溢出漏洞进行验证的选项(/GS)。另外,在AMD64体系结构中一种叫做NX的设施可限制执行栈上的代码。即使在x86模式下处理器中的NX位也是可用的。NX代表不可执行(no execute),它可以给页面上加标记使得其上的代码不能被执行。这样,即使攻击者利用缓冲区溢出漏洞向进程插入代码,跳转到代码处开始执行也不是一件容易的事情。

Windows Vista引入了更多的安全特性来阻止攻击者。加载到内核态的代码要经过检查(这在x64系统中是默认的)并且只有被正确签名的代码才可以被加载。在每个系统中,DLL和EXE的加载地址连同栈分配的地址都经过了有意的混排,这使得攻击者不太可能利用缓冲区溢出漏洞跳转到一个众所周知的地址然后执行一段被特意编排的获得权限提升的代码。会有更小比例的系统受到依赖于标准地址处的二进制数据的攻击。在受到攻击时系统更加可能只是崩溃掉,将一个潜在的权限升级攻击转化为危险性更小的拒绝服务攻击。

在微软公司称为用户账户控制(User Account Control, UAC)的引入是另一个改变。这用来解决大部分用户以管理员身份运行系统这个长期的问题。Windows的设计并不需要用户以管理员身份使用系统,但在很多发布版本中对此问题的忽视使得如果你不是管理员就不可能顺利地使用Windows。始终以管理员身份使用系统是危险的。用户的错误会轻易地毁坏系统,而且如果用户由于某种原因被欺骗或攻击了而去运行可能危害系统的代码,这些代码将拥有管理员的访问权限并且可能会把其自身深深埋藏在系统中。

如果有UAC,当尝试执行需要管理员访问权限的操作时,系统会显示一个重叠的特殊桌面并且接管控制权,使得只有用户的输入可以授权这次访问(与C2安全中CTRL-ALT-DEL的工作方式类似)。当然,攻击者不需要成为管理员也可以破坏用户所真正关心的,比如他的个人文件。但UAC确实可阻止现有类型的攻击,并且如果攻击者不能修改任何系统数据或文件,那受损的系统恢复起来也比较容易。

Windows Vista中最后的一个安全特性已经提到过了。这就是对具有安全边界的受保护进程(protected process)的支持。通常,在系统中用户(由令牌对象代表)定义了权限的边界。创建进程后,用户可通过任意数目的内核设施来访问进程以进行进程创建、调试、获取路径名和线程注入等。受保护进程关掉了用户的访问权限。这个设施在Vista中的唯一用处就是允许数字版权管理软件更好地保护内容。对受保护进程的使用在未来的发布版本中可能会用于对用户更加友好的目的,比方说保护系统以应对攻击者而不是保护内容免受系统所有者的攻击。

由于世界范围内越来越多的针对Windows系统的攻击,近年来微软公司加大了提高Windows安全性的努力。其中某些攻击非常成功,使得整个国家和主要公司的计算机都宕掉了,导致了数十亿美元的损失。这些攻击大都利用了编码中的小错误,这些错误可导致缓冲区溢出,从而使得攻击者可以通过重写返回地址、异常处理指针和其他数据来控制程序的执行。使用类型安全的语言而不是C和C++可避免许多此类的问题。即使使用这些不安全的语言,如果让学生更好地理解参数和数据验证中的陷阱,许多漏洞也可以避免。毕竟,许多在Microsoft编写代码的软件工程师在几年前也还是学生,就像正在阅读此实例研究的你们中的许多人一样。有许多关于在基于指针的语言中可被利用的编码上的小错误的类型以及如何避免的书籍(比如,Howard和LeBlank, 2007)。

11.10 小结

Windows Vista中的内核态由HAL、NTOS的内核和执行体层以及大量实现了从设备服务到文件系统、

从网络到图形的设备驱动程序组成。HAL对其他组件隐藏了硬件上的某些差别。内核层管理CPU以支持多线程和同步,执行体实现大多数的内核态服务。

执行体基于内核态的对象,这些对象代表了关键的执行体数据结构,包括进程、线程、内存区、驱动程序、设备以及同步对象等。用户进程通过调用系统服务来创建对象并获得句柄的引用以用于后续对执行体组件的调用。操作系统也创建一些内部对象。对象管理器维护者一个名字空间,对象可以插入该名字空间以备后续的查询。

Windows系统中最重要的对象是进程、线程和内存区。进程拥有虚拟地址空间并且是资源的容器。线程是执行的单元并被内核层使用优先级算法调度执行,该优先级算法使优先级最高的就绪线程总在运行,并且如有必要可抢占低优先级线程。内存区表示可以映射到进程地址空间的像文件这样的内存对象。EXE和DLL等程序映像用内存区来表示,就像共享内存一样。

Windows支持按需分页虚拟内存。分页算法基于工作集的概念。系统维护着几种类型的页面列表来优化内存的使用。这些页面列表是通过调整工作集来填充的,调整过程使用了复杂的规则试图重用在规定时间内没有被引用的物理页面。缓存管理器管理内核中的虚拟地址并用它将文件映射到内存,这提高了许多应用程序的I/O性能,因为读操作不用访问磁盘就可被满足。

设备驱动程序遵循Windows驱动程序模型,并执行输入/输出。每个驱动程序开始先初始化一个驱动程序对象,该对象含有可被系统调用以操控设备的过程的地址。实际的设备用设备对象来代表,设备对象可以根据系统的配置描述来创建,或者由即插即用管理器按照它在枚举系统总线时所发现的设备创建。设备组织成一个栈,I/O请求包沿着栈向下传递并被每个设备的驱动程序处理。I/O具有内在的异步性,驱动程序通常将请求排队以便后续处理然后返回到调用者。文件系统卷作为I/O系统中的设备实现。

NTFS文件系统基于一个主文件表,每个文件或者目录在表中有一条记录。NTFS文件系统的所有元数据本身是NTFS文件的一部分。每个文件含有多个属性,这些属性或存储在MFT记录中或者不在其中(存储在MFT外部的块中)。除此之外,NTFS还支持Unicode、压缩、日志和加密等。

最后,Windows Vista拥有一个基于访问控制列表和完整性级别的成熟的安全系统。每个进程带有一个令牌,此令牌表示了用户的标识和进程所具有的特殊权限。每个对象有一个与其相关联的安全描述符。安全描述符指向一个自主访问控制列表,该列表中包含允许或者拒绝个体或者组访问的访问控制入口项,Windows在最近的发行版本中增加了大量的安全特性,包括用BitLocker来加密整个卷,采用地址空间随机化,不可执行的堆栈以及其他措施使得缓冲区溢出攻击更加困难。

习题

1. HAL可以跟踪从1601年开始的所有时间。举一个例子,说明这项功能的用途。
2. 在11.3.2节,我们介绍了在多线程应用程序中一个线程关闭了句柄而另一个线程仍然在使用它们所造成的问题。解决此问题的一种可能性是插入序列域。请问该方法是如何起作用的?需要对系统做哪些修改?
3. Win32 系统没有信号功能。如果要引入此功能,我们可以将信号设置为进程所有,线程所有,两者都有或者两者都没有。试着提出一项建议,并解释为什么。
4. 另一种使用DLL的方式是静态地将每个程序链接到它实际调用到那些库函数,既不多也不少。在客户端机器或者服务器机器上引入此方法,哪个更合理?
5. 在Windows中线程拥有独立的用户态栈和内核态栈的原因是哪些?
6. TLB对性能有重大的影响。为了提高TLB的有效性,Windows使用了大小为4MB的页,这是什么?
7. 在一个执行体对象上可定义的不同操作的数量有没有限制?如果有,这个限制从何而来?如果没有,请说明为什么。
8. Win32 API的调用WaitForMultipleObjects以一组同步对象的句柄为参数,使得线程被这组同步对象阻塞。一旦它们中的任何一个收到信号,调用者线程就会被释放。这组同步对象是否可以包含两个信号灯、一个互斥体和一个临界区?理由是什么?提示:这不是一个恶作剧的问题,但确实有必要认真考虑一番。
9. 给出三个可能会终止线程的原因。
10. 如11.4节所述,有一个特殊的句柄表用于为进程和线程分配ID。句柄表的算法通常是分配第一个可用的句柄(按照后进先出的顺序维护空

闲链表)。在最新发布的Windows 版本中,该算法变成了ID表总是以先进先出的顺序跟踪空闲链表。使用后先进出顺序分配进程线ID有什么潜在的问题?为什么.UX操作系统没有这个问题?

11. 假设时间片配额被设置为20 毫秒,当前优先级为24的线程在配额开始的时候刚开始执行。突然一个I/O操作完成了并且一个优先级为28的线程变成就绪状态。这个线程需要等待多久才可以使用CPU?
12. 在Windows Vista 中,当前的优先级总是大于或等于基本的优先级。是否在某些情况下当前的优先级低于基本的优先级也是有意义的?若有,请举例。否则请说明原因。
13. 在Windows中很容易实现一些设施将运行在内核中的线程临时依附到其他进程的地址空间。为什么在用户态却很难实现?这样做有何目的?
14. 即使有很多空闲的可用内存而且内存管理器也不需要调整工作集,分页系统仍然会经常对磁盘进行写操作。为什么?
15. 为什么用来访问进程页目录和页表的物理页面的自身映射数据总是占用同一片4MB的内核虚拟地址空间(在x86上)?
16. 如果保留了一段虚拟地址空间但是没有提交它,你认为系统会为其创建一个VAD吗?请证明你的答案。
17. 在图11-36中,哪些转移是由策略决定的,而不是由系统事件(例如,一个进程退出并释放其页面)所强迫的转移?
18. 假设一个页面被共享并且同时存在于两个工作集中。如果它从一个工作集移出,在图11-36中它将会到哪里去?当它从第二个工作集移出时会发生什么?
19. 当进程取消对一个页面的映射时,干净的页面进行图11-36中的转移(5),那脏的栈页怎样处理呢?为什么脏的栈页面被取消映射时不会被转移到已修改列表中呢?
20. 假设一个代表某种类型互斥锁(比如互斥对象)的分发对象被标记为使用通知事件而不是同步事件来声明锁被释放。为什么这样是不好的?你的回答在多大程度上依赖于锁被持有的时间、时间片配额的长度和系统是否为多处理器的?

21. 一个文件存在如下映射。请给出MFT的行串。

偏移	0	1	2	3	4	5	6	7	8	9	10
磁盘地址	50	51	52	22	24	25	26	53	54	-	60

22. 考虑图11-43中的MFT记录。假设该文件增长了并且在文件的末尾添加了第10个块。新块的序号是66。现在MFT记录会是什么样子?
23. 在图11-46b中,最先的两个行串的长度都为8个块。你觉得它们长度相等只是偶然的,还是跟压缩的工作方式有关?请解释理由。
24. 假如您想创建Windows Vista的精简版。在图11-47中可以取消哪些字段而不削弱系统的安全性?
25. 由许多程序(Web浏览器、Office、COM 服务器)使用的一个扩展模型是对程序所包含的DLL添加钩子函数来扩展其底层功能。只要在加载DLL前仔细模拟客户的身份,该模型对基于RPC的服务来说就是合理的,是这样的吗?为什么不是?
26. 在NUMA机器上,不管何时Windows内存管理器需要分配物理内存来处理页面失效,它总尝试从当前线程的理想的处理器的NUMA节点中获取。为什么?如果线程正运行在其他处理器上呢?
27. 系统崩溃时,应用程序可以轻易地从基于卷的影子副本的备份中恢复,而不是从磁盘状态中恢复。请给出几个这样的例子。
28. 在某些情况下为了满足安全性的要求需要为进程提供全零的页面,在11.9节中向进程的堆提供内存就是这样的一种情况。请给出一个或者多个其他需要对页面清零的虚拟内存操作。
29. 在当前所有的Windows 发行版本中,regedit命令可用于导出部分或全部注册表到一个文本文件。在一次工作会话中保存注册表若干次,看看有什么变化。如果您能够在Windows中安装软件或硬件,请找出安装或卸载程序或设备时注册表有何变化。
30. 写一个UNIX程序,模拟用多个流来写一个NTFS文件。它应能接受一个或多个文件作为参数,并创建一个输出文件,该文件的一个流包含所有参数的属性,其他的流包含每个参数的内容。然后再写一个程序来报告这些属性和流并提取出所有的组成成分。

第12章 实例研究3: Symbian操作系统

在前面的两章里,我们已经介绍了两种在台式机以及笔记本电脑上通用的操作系统:Linux以及Windows Vista。但实际上,超过90%的CPU都并非用于台式机或笔记本电脑,而是用于嵌入式系统,例如手机、PDA、数码相机、便携式摄像机、游戏机、iPod、MP3播放器、CD播放器、DVD刻录机、无线路由器、电视机、GPS接收器、激光打印机、汽车,以及其他许多消费产品。它们大多使用现代的32位或64位芯片,几乎全部安装有成熟的操作系统。但是很少有人意识到这些操作系统的存在。在这一章里,我们将研究嵌入式系统中十分通用的一个操作系统:Symbian操作系统。

Symbian操作系统是一个运行在一些厂商的智能手机平台上的操作系统。智能手机因其运行功能齐全的操作系统的特性而得名。Symbian操作系统用来作为很多厂商的多种智能手机的基础,通过精心设计,专门运行在智能手机平台上,即那些CPU、内存以及存储容量有限、主要针对通信的通用计算机。

针对Symbian操作系统的探讨将从它的历史开始。随后给出这个系统的概况,大致介绍它是怎样设计的以及实现什么样的功能。然后如前两章那样,介绍Symbian操作系统设计的各个方面,包括进程、内存管理、I/O、文件系统以及安全性。最后介绍Symbian操作系统怎样处理智能手机中的通信问题。

12.1 Symbian操作系统的历史

UNIX操作系统有着很长的历史,几乎与计算机一样的久远。Windows操作系统也有较长的历史。而Symbian操作系统的历史相对较短。它起源于20世纪90年代研发的操作系统,首次出现则是在2001年。鉴于Symbian操作系统所依赖的智能手机平台也是近期才得到发展的,这一点应当并不令人惊讶。

Symbian操作系统起源于掌上设备,随后经历了几个版本的升级得到快速发展。

12.1.1 Symbian操作系统的起源:Psion和EPOC

Symbian操作系统继承于某些最初的掌上设备。20世纪80年代末,作为将台式设备的功能整合到小型的可移动装置中的一个手段,掌上设备得到发展。对掌上电脑的初次尝试并没有引起太多的注意。Apple Newton是一个设计良好的掌上电脑设备,但只在少数使用者中间流行。虽然开始很缓慢,但20世纪90年代中期发展的掌上电脑则已经针对用户以及人们使用移动设备的方式进行了更好的修改。掌上电脑最初设计为PDA,是电子规划员的个人数码助手,不断地发展并具有了多种功能。随着它们的发展,在功能上已经趋向于台式机,也相应地有了与台式机同样的需求。它们需要多任务的处理方式;需要增加多种形式的存储能力;需要在输入输出上更加灵活。

掌上设备也逐渐包含了通信功能。随着个人设备的发展,个人通信也同样在发展着。移动电话的使用在20世纪90年代末期有了飞速的发展。因此,将掌上设备与移动电话相结合形成智能手机是一件很自然的事情。而随着这种合并的产生,在掌上设备里运行的操作系统也不得不发展。

在20世纪90年代,Psion电脑公司制造了PDA设备。1991年,Psion生产了Series 3——一个配有小尺寸单色显示屏的小型电脑,小到可以放入口袋中。在Series 3之后,1996年又制造了具有红外功能的Series 3c,1998年又生产了具有更快处理器速度以及更多内存容量的Series 3mx。它们各自均获得了成功,而它们的成功主要源自其良好的功耗管理以及与包括个人电脑和掌上设备在内的其他设备之间的互通性。程序是用C语言实现的,利用面向对象设计,并采用了“应用引擎”——Symbian操作系统发展中的一个重要部分。这种引擎方案功能强大。它借鉴了微内核的设计,从而强调类似于服务器般的引擎的功能性——通过回应来自各应用程序的请求进行功能管理。这种方式使得它可以拥有标准化的API以及利用对象的抽象来使得应用程序编程者免于诸如数据格式等令人麻烦的细节问题。

1996年,Psion开始设计一种新型的32位操作系统,它支持触摸屏上的定位设备,采用多媒体技术,

并且具有更丰富的通信功能。这个新的系统同时也更加面向对象,并且可以移植到不同的体系结构和设备设计上。Psion的付出所得到的结果是系统EPOC版本1的推出。EPOC由C++编程实现,并且是彻底的面向对象的设计。它依然使用了引擎方案,并将这个设计理念扩展到协同访问系统服务和外部设备的一系列服务器。EPOC扩展了通信能力,开发了多媒体,引入了新的针对触摸屏等接口的平台,并通用化了硬件接口。

之后EPOC又继续发展了两个版本:EPOC版本3 (ER3)和EPOC版本5 (ER5)。它们在新的Psion Series 5及Series 7的电脑平台上运行。

Psion同时也试图强调它的操作系统可以适用于其他硬件平台。在2000年左右,新的掌上设备发展的最大机会在手机业务,而在这方面,众多厂商一直都在为它的下一代设备寻找一个新的先进的操作系统。为了利用这些机会,Psion与手机业的巨头,包括Nokia、Ericsson、Motorola以及Matsushita (Panasonic),成立了一个合资项目Symbian,用来控制EPOC操作系统核心的所有权并使其继续发展。这一新的内核设计现在称为Symbian操作系统

12.1.2 Symbian操作系统版本6

鉴于EPOC的最后一个版本为ER5,因此Symbian操作系统在2001年以版本6首发。它利用了EPOC的灵活特性,并主要面向几个不同的通用平台。其设计非常灵活,从而满足了发展各种高级移动设备以及手机的需要,同时允许众多厂商具有区别各自产品的能力。

同时,Symbian操作系统将会积极采用现代最先进的、成熟的关键技术。这更强化了对于面向对象以及客户机-服务器结构的选择,正如它们在台式机以及网络世界的愈加广泛的使用。

Symbian操作系统版本6被它的设计者们称为“开放的”。这个“开放”不同于UNIX以及Linux那样的开源特性。这里,“开放”指的是这个操作系统的结构是公开的并且是大家均可获得的。另外,所有的系统接口也都公开,从而鼓励第三方软件的开发。

12.1.3 Symbian操作系统版本7

Symbian操作系统版本6在设计和功能上很像EPOC以及版本6以前的版本。它的设计主要着眼于移动电话。此后,随着越来越多的厂商设计了移动电话,即使是EPOC的灵活性也不能够应付如此众多的移动电话对Symbian操作系统的使用需求。

Symbian操作系统版本7保持了EPOC的台式机功能,但是大部分系统内部被重写了以包含多种智能手机功能。操作系统内核以及操作系统服务从用户界面中分离出来。相同的操作系统现在可以在众多不同的智能手机平台上运行了,它们各自拥有着不同的用户界面系统。Symbian操作系统现在可以扩展以处理新的不可预期的信息格式或者用在使用不同的电话技术的智能手机上。Symbian操作系统版本7发布于2003年。

12.1.4 今天的Symbian操作系统

Symbian操作系统版本7是一个重要的版本,因为它将抽象性以及灵活性带入了操作系统。然而,这种抽象是有代价的。操作系统的性能不久便成为一个需要解决的问题。

于是重写操作系统的工程又开始了,这次主要着眼于性能。这个新的操作系统设计旨在保持Symbian操作系统版本7的灵活性的同时提高其性能,并增强其安全性。Symbian操作系统版本8,发布于2004年,提高了Symbian操作系统的性能,尤其是在其实时功能上。Symbian操作系统版本9发布于2005年,增加了基于性能的安全以及看门机制安装的概念。如同Symbian操作系统版本7增加软件的灵活性那样,Symbian操作系统版本9增加了针对硬件的灵活性。一个新的二进制模型得到了开发,从而使硬件开发者可以使用Symbian操作系统,而不必重新设计硬件使其适应某一特定的结构模型。

12.2 Symbian操作系统概述

前一节介绍过,Symbian操作系统是由一个掌上设备操作系统发展成为一个以实时性能作为目标的用在智能手机平台上的操作系统。这一节里我们将对Symbian操作系统设计中蕴含的概念作简单的介绍。这些概念与如何使用这个操作系统息息相关。

Symbian是一个独特的操作系统,因为它是以智能手机作为目标平台的。它不是将一般的操作系统

硬装入智能手机（有很大的难度），也不是使较大的操作系统适应于较小的平台。然而，它确实包含了许多其他大型操作系统所具有的特性，从多任务到内存管理再到安全问题。

Symbian操作系统继承了其前身的最佳的特性，具有由EPOC传承下来的面向对象特性。并且如版本6中所引入的，使用了微内核的设计方案，最小化了内核的开销，将不必要的功能移到了用户层进程。它模仿EPOC中应用的引擎模型，使用了客户机/服务器结构。它支持多种台式机功能，包括多任务和多线程，以及可扩展存储系统。它还继承了EPOC中强调的多媒体与通信。

12.2.1 面向对象

面向对象是一个意味着抽象的术语。在一个面向对象的设计中，针对某个系统成分的数据和功能，建立一个抽象的实体，称为对象。一个对象提供了具体的数据以及功能，但隐藏了具体实现。一个合理实现的对象可以被移除并被另外一个不同的对象代替，只要系统其他部分对这个对象的使用（也即其接口）保持不变。

当面向对象应用到操作系统设计中时，就意味着所有的系统调用以及内核端功能的使用均要通过接口，而不能直接获取实际数据或依靠其他类型的实现。一个面向对象的内核的实现通过对象来提供内核功能。使用内核端对象通常意味着一个应用程序具有一个对象的句柄，也就是对对象的一个引用，然后通过这个句柄来获得对该对象接口的访问。

Symbian操作系统采用了面向对象的设计。系统功能的实现是隐藏的，系统数据的使用通过系统对象已定义的接口完成。在Linux等操作系统中，构建一个文件描述符，并将这个描述符作为open调用的参数；而在Symbian操作系统中则会创建一个文件对象，然后调用该对象的open方法。举例来说，在Linux操作系统中，正如大家所知道的，文件描述符对应于系统内存中文件描述符表的索引的整数表示；而在Symbian操作系统中，文件系统表的实现是未知的，而所有的文件操作是通过一个特定的文件类的对象来实现的。

需要注意的是Symbian操作系统与其他在设计中运用了面向对象理念的操作系统不同。例如，许多操作系统设计使用了抽象数据类型，人们甚至可以说系统调用整个理念就是通过将系统实现细节对用户程序隐藏起来而实现了抽象。而对于Symbian操作系统，整个操作系统的结构均是面向对象设计的。操作系统功能以及系统调用都是与系统对象相联系的。资源分配以及保护则是对应于对象的分配，而不是系统调用的实现。

12.2.2 微内核设计

具有面向对象内在特性的Symbian操作系统的内核结构是微内核设计。内核中包括最小限度的系统功能以及数据，许多系统功能被放到了用户空间服务器端。服务器端通过获得系统对象的句柄并对这些对象进行必要的系统调用来完成各自的服务。用户空间应用程序与这些服务器端进行交互而不是采取系统调用。

典型的基于微内核的操作系统初始化引导时占用较少的内存，并且其结构也更加动态。当需要时可以启动服务器，而在启动时并不需要全部的服务器。微内核大多为可插拔结构，允许当需要时加载系统模块并插入到内核中。因此，微内核结构十分灵活：支持新功能的代码（例如，新硬件驱动程序）可以随时加载和插入。

Symbian操作系统被设计为基于微内核的操作系统。通过打开与资源服务器端的连接访问系统资源，资源服务器随后协同访问资源本身。Symbian操作系统支持对于新的实现的可插拔结构。对于系统功能的新的实现可以设计为系统对象，并动态插入到内核中。例如，可以实现新的文件系统并且在操作系统运行的同时添加到内核中。

这种微内核的设计也带来了一些需要探讨的话题。在传统的操作系统中一个系统调用便已足够时，微内核使用消息传递。性能可能会由于对象间通信所增加的花费而受到影响。在传统操作系统中位于内核的那些功能被移到用户空间时效率可能会降低。举例来说，与可以直接访问内核数据结构的Windows内核中的进程调度相比，进程调度的多函数调用的开销降低了性能。由于在用户空间与内核空间对象中传递消息，会经常发生特权级切换，这就更降低了它的性能。最后，在传统设计方案中只用到了一个地址空间的系统调用，而这种消息传递以及优先级转换意味着至少需要用到两个地址空间来完成一个微内核服务请求。

这些性能问题使得Symbian操作系统（以及其他基于微内核的操作系统）的设计者们对于设计以及实现细节给予了极大关注。设计的重点是最小化的、紧凑的集中服务。

12.2.3 Symbian操作系统纳核

Symbian操作系统的设计者们在操作系统设计的核心采用了一种纳核的结构来处理微内核所具有的问题。正如在微内核结构中，某些系统功能被移到了用户空间服务器端，Symbian操作系统将需要复杂实现的功能分离到内核中，而只将最基本的功能放在系统核心的纳核中。

在Symbian操作系统中，纳核提供部分最基本的功能。在纳核中，运行在特权级别的简单线程完成着十分初级的功能。在这一层的实现中包括调度同步操作、中断处理和同步对象，如互斥变量以及信号量。这一层中的实现功能大多是可抢占的，而且是非常初级的（所以它们可以很快）。例如，动态内存分配对于纳核就是过于复杂的功能。

这种纳核的设计需要一个二级层次来实现较为复杂的内核功能。Symbian操作系统内核层提供了操作系统所需要的其他较为复杂的内核功能。每个在Symbian操作系统内核层的操作都是特权级的操作，并与纳核层的初级操作一起来完成更加复杂的内核工作。复杂的对象服务、用户态线程、进程调度以及上下文切换、动态内存、动态库加载、复杂的同步、对象及进程间通信只是在这层实现的部分操作。这层是完全可抢占式的，并且中断可以使其对任何一部分的执行进行重新调度，即使是在上下文转换的过程中也可以。

图12-1展示了一个完整的Symbian操作系统内核的结构。

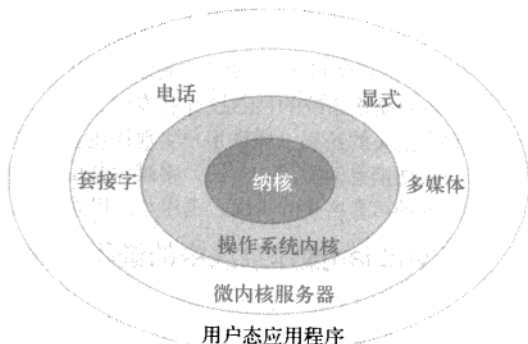


图12-1 Symbian操作系统内核结构具有多层

12.2.4 客户机/服务器资源访问

正如我们所提到的那样，Symbian操作系统使用微内核设计并使用客户机/服务器模型来访问系统资源。需要访问系统资源的应用是客户端，操作系统中运行着的用来协调资源访问的程序是服务器端。在Linux中，人们可能需要调用open来打开一个文件；在Windows中，需要利用Microsoft API来创建一个窗口；而在Symbian操作系统中的过程均是相同的：首先建立一个到服务器端的连接，服务器端需要确认这个连接，然后对服务器端发出实现某个操作的请求。因此打开一个文件表示找到文件服务器端，调用connect建立与服务器端的连接，然后发送给服务器端一个附有某特定文件名字的open请求。

这样做对于保护资源有着几点好处。首先，它符合操作系统的面向对象以及微内核的设计。其次，这种结构对于管理多任务、多线程系统中所需要的资源多重访问十分有效。最后，每个服务器都可以专注于它必须管理的资源，并能方便地进行升级以及替换为新的设计。

12.2.5 较大型操作系统的特点

尽管Symbian操作系统所针对的目标电脑规模较小，但它有着许多大型系统的特点。你可以在Symbian操作系统上找到大型操作系统（如Linux以及Windows）的各种特性，只是以另一种形式出现。Symbian操作系统与较大型的操作系统有一些共同的特性。

- 进程与线程：Symbian操作系统是一个多任务多线程的操作系统。许多进程可以同时运行，相互间可以进行通信，也可以在各进程内运行多个线程。
- 常见文件系统支持：Symbian操作系统利用一个文件系统模型来管理对系统存储空间的访问，正如大型操作系统一样。它具有一个与Windows兼容的默认文件系统（默认使用FAT-32文件系统），通过使用插件式接口支持其他文件系统。Symbian操作系统支持几种不同类型的文件系统，包括FAT-16、FAT-32、NTFS，以及许多存储卡格式（例如JFFS）。
- 网络：Symbian操作系统支持TCP/IP网络以及其他的通信接口，例如串行、红外和蓝牙。

- 内存管理：尽管Symbian操作系统不使用（也没有相应的设备）虚拟内存映射，但它通过按页管理实现对内存访问，并支持页的置换，也就是说支持页面换入，但不支持页面换出。

12.2.6 通信与多媒体

Symbian操作系统以多种方式协助通信。我们很难简单地对其进行概述而不提及通信特点。通信的模型遵循面向对象、微内核以及客户机/服务器结构。Symbian操作系统的通信结构是以模块形式建立的，从而允许新的通信机制方便地接入操作系统。从用户层接口到新的协议实现，到新的设备驱动，模块可以实现任何功能。由于这样的微内核实现，可以引入新的模块并动态地加载到系统操作中。

由于Symbian操作系统只针对智能手机平台，因此有许多独特的特点。它具有一个可插拔的消息结构，可以引入新的消息类型，并可以通过消息服务器动态加载的模块来实现。消息系统被设计为层次结构，各层由特定类型的对象来实现。例如，消息传递对象不同于消息类型对象。一种形式的消息传递，例如手机无线传递（类似于CDMA）可以传送几种不同类型的消息（标准文本消息类型、SMS类型或者如BIO消息等的系统指令）。通过实现新的对象并将其加载到内核中可以引入新的传递方法。

Symbian操作系统的核心设计有专门针对多媒体的各种API。多媒体设备以及上下文由特殊的服务器和用户自定义的结构（允许用户实现描述新的或现存的上下文以及怎样对上下文进行处理的模块）来处理。与实现消息的方法相类似，多媒体是由多种形式相互作用的对象来实现。声音播放的方式被设计为一个与各种声音格式的实现方式相互作用的对象。

12.3 Symbian操作系统中的进程和线程

Symbian操作系统是一个多任务操作系统，像其他操作系统那样，使用了进程和线程的概念。然而，Symbian操作系统的内核结构以及它对资源稀缺性的处理方式影响了它看待这些多任务对象的方式。

12.3.1 线程和纳线程

对于多任务，Symbian操作系统更倾向于线程，并且是建立在线程概念上的，而不是把进程作为多任务的基础。线程构成了多任务的中心单元。操作系统简单地把一个进程看成是具有一个进程控制块和某个内存空间的线程的集合。

Symbian操作系统对于线程的支持是基于纳线程的纳核。纳核仅提供简单的线程支持，每个线程是由一个基于纳核的纳线程来支持的。纳核为纳线程提供调度、同步（线程间通信）以及计时服务。纳线程运行在特权模式下，需要一个栈来存储它们的运行时刻环境数据。纳线程不能运行在用户态。这就意味着操作系统能够对每个纳线程保持紧密的控制。每个纳线程需要一个数据的极小集来运行：实质上就是它的栈以及栈的大小。操作系统保持对其他一切的控制，比如每个线程使用的代码，以及在运行时刻的栈上存储线程的上下文。

同进程具有状态一样，纳线程也具有线程状态。Symbian操作系统的纳核使用的模型在基本模型中增加了一些状态。除了基本状态以外，纳线程还可以处于如下状态：

- 挂起。这就是当一个线程挂起另一个线程时的状态，与等待状态不同，在等待状态下一个线程是被某个上层对象阻塞（例如，一个Symbian操作系统线程）。
- 快速信号量等待。处于这个状态的线程正在等待一个快速信号量（哨兵变量的一种）得到信号通知。快速信号量是纳核级别的信号量。
- DFC等待。处于这种状态的线程正在等待一个延迟的函数调用或者要被加入到DFC队列中的DFC。DFC用在设备驱动实现中。它们代表对于内核的调用，可被Symbian操作系统内核层排入队列并且调度执行。
- 休眠。休眠线程正在等候特定长度的时间过去。
- 其他。还有一种通用状态，是当开发人员为纳线程实现额外的状态时使用的。当开发人员为新的手机平台（称作个性层）扩展纳核功能时使用该状态。进行这个工作的开发人员也必须实现这些状态与他们的扩展实现之间的来回跳转。

下面将纳线程思想与传统进程思想作比较。纳线程实际上是一个完全轻量级的进程。它具有极小的

上下文,当纳线程进出处理器时进行切换。每个纳线程和进程一样具有一个状态。对于纳线程来说,关键是纳核对它们的紧密控制,以及构成每个纳线程上下文的极小数据集。

Symbian操作系统线程依赖于纳线程,内核增加除纳核提供的功能之外的支持。标准程序使用的用户模式线程由Symbian操作系统线程执行。每个Symbian操作系统线程包含一个纳线程并且添加自己的运行时时刻栈到纳线程使用的栈中。Symbian操作系统线程可以通过系统调用在内核模式下进行操作。Symbian操作系统也能为执行增加例外处理以及退出信号。

Symbian操作系统线程在纳线程实现之上实现自己的状态集。由于Symbian操作系统线程将一些功能性增加到纳线程实现中,因此新的状态反映了构成Symbian操作系统线程的新的思想。Symbian操作系统添加了Symbian操作系统线程可以进入的新的七种状态,来关注Symbian操作系统线程可能出现的特殊阻塞条件。这些特殊状态包括在信号量上的等待和挂起(正常的)、互斥变量以及条件变量。由于Symbian操作系统的实现处于纳线程之上,因此这些状态从某种方面上来说是由纳线程状态实现的,通常都是用不同的方式使用挂起的纳线程状态。

12.3.2 进程

Symbian操作系统的进程,就是在一个单一的进程控制块结构下,具有一个单一存储空间的,归于一类的Symbian操作系统的线程组。可能只有一个执行的线程,或者一个进程控制块下有很多线程。Symbian操作系统线程和纳线程已经定义了进程状态和进程调度的概念。因此,调度一个进程实际上是通过调度一个线程以及初始化数据需要使用的正确的进程控制块来完成的。

Symbian操作系统线程通过几种方式,在一个单一进程的组下工作在一起。首先,有一个主线程被标志为进程的起始点。其次,线程共享调度参数。也就是说,进程通过一种调度方法——改变进程参数,来改变所有线程的参数。第三,线程共享包括设备和其他对象描述符的存储空间对象。最后,当一个进程终止时,内核终止该进程的所有线程。

12.3.3 活动对象

活动对象是线程的特有形式,用这种方式实现以便减轻它们带给操作环境的负担。Symbian操作系统的设计者意识到,应用中的线程在很多情况下可能会发生阻塞。由于Symbian操作系统致力于通信工具方面,因此许多应用程序具有类似的执行模式:它们向一个通信套接字写数据或者通过管道发送信息,然后在等待接收者的响应时阻塞。这样设计活动对象,是为了当它们从这种阻塞状态返回时,具有进入被调用代码的单一入口点,这简化了它们的实现。由于活动对象运行在用户空间,因此它们具有Symbian操作系统线程的特性。它们本身具有自己的纳线程,并且能够加入Symbian操作系统的其他线程构成操作系统的一个进程。

假若活动对象仅仅是Symbian操作系统线程,有人就会问操作系统从这种简化的线程模型中得到了什么益处。活动对象的关键点体现在调度上。所有的活动对象在等待事件的时候驻留在一个单一进程中,对系统而言可以作为一个单一的线程。内核不必连续地检查每一个活动对象是否被解除阻塞。因此,单一进程中的活动对象,能够由在一个单一线程中执行的单一调度器来协调。通过将其他方面作为多线程执行的代码结合到一个线程中,通过构建固定的入口点进入代码,以及通过使用一个单一调度器来协调它们的执行,活动对象构成了标准线程的一种高效、轻量版本。

认识到活动对象和Symbian操作系统进程结构在何处融合成为一体是很重要的。当一个传统线程通过系统调用进入等待状态从而阻塞自己的运行时,操作系统仍然需要检查这个线程。在上下文切换期间,操作系统需要花费时间检查处于等待状态的阻塞进程,决定是否需要将其移动到就绪状态。活动对象把自己放入等待状态以等待特定的事件,因此,操作系统不需要去检查它们,而只是在特定的事件发生后移动它们。结果就是更少的线程检测以及更好的性能。

12.3.4 进程间通信

在类似Symbian操作系统的多线程环境中,进程间通信对系统性能是至关重要的。线程,特别是系统服务形式的线程经常通信。

套接字是Symbian操作系统使用的基本通信模型。它是两个端点之间抽象的通信管道。这一抽象是

用来隐藏端点之间的传输方法和数据管理。Symbian操作系统使用套接字的概念在客户端和服务端之间、线程到设备之间以及线程之间进行通信。

套接字模型也构成了设备I/O的基础。抽象再次成为使这一模型更加有效的关键。同一个设备进行数据交换的所有机制不是由应用程序管理的，而是由操作系统管理的。例如，网络环境中工作于TCP/IP上的套接字可以很容易地通过改变套接字使用的类型参数而适应于蓝牙环境。这种变换下，其他大部分的数据交换工作都是由操作系统完成的。

Symbian操作系统实现了通用操作系统上使用的标准同步原语。操作系统中广泛地使用了信号量和互斥量的一些形式。这些为进程和线程提供同步能力。

12.4 内存管理

诸如Linux和Windows系统中的内存管理使用了很多我们前面讲过的关于实现内存资源管理的概念。例如，从物理内存框架构建的虚拟内存页面、按需分页的虚拟内存以及动态页面置换，这些概念共同给出近乎无限的内存资源形象。这里物理内存是由诸如硬盘空间等支持和扩展的。

Symbian操作系统和实际的通用操作系统一样，也必须提供内存管理模式。然而，由于智能手机上的存储容量非常有限，内存模型受到限制，而且进行内存管理的时候不能使用虚拟内存/交换空间模型。但正是如此，Symbian操作系统使用了我们讨论过的内存管理的大多数机制，包括硬件MMU。

12.4.1 没有虚拟内存的系统

许多计算机系统没有提供成熟的支持按需分页的虚拟内存的设备。在这些平台上操作系统可以获得的惟一的存储设备就是内存，它们没有硬盘设备。正因为这样，大多数较小的系统，从PDA到智能手机，再到更高层次的掌上设备，都不支持按需分页的虚拟内存。

下面考虑大多数小的平台设备上使用的内存空间。这些系统一般都有两种类型的存储介质：RAM和闪存。RAM存储操作系统代码（当系统启动时使用），闪存用作操作内存和永久性（文件）存储介质。通常，可以为一个设备（比如安全数据卡）增添额外的闪存，这些存储空间专门用作永久性存储。

没有支持按需分页的虚拟内存不代表缺少内存管理。实际上，大多数较小的平台构建在包含许多较大系统的管理特征的硬件上。这些管理特征包含诸如分页、地址翻译以及虚拟/物理地址抽象。没有虚拟内存仅仅意味着页面不能从内存交换出去并存储在外部设备上，而内存页的抽象仍然在使用。页面被替换了，但是它们也只是被丢弃了。也就是说只有代码页可以被置换，因为只有它们备份在闪存上。

内存管理包含如下的任务：

- 应用程序大小的管理：应用程序的大小（所有的代码和数据）对如何使用内存有很大的影响。创建小的软件需要技巧和规则。使用面向对象的设计在这里成为一种阻碍（更多的对象意味着更多的动态内存分配，而这需要更大的堆尺寸）。大多数针对较小平台的操作系统非常不鼓励任何模块的静态链接。
- 堆的管理：堆（用来进行动态内存分配的空间）在较小的平台上必须严格地管理。堆空间在较小的平台上一旦划定边界，以便程序员尽可能地回收和重用。冒险越界会导致内存分配的错误。
- 就地执行：没有磁盘设备的平台通常支持就地执行。这就是说闪存被映射到虚拟内存地址空间，程序可以直接从闪存上执行，而不需要首先复制到RAM上。这样做使加载时间减小为零，允许应用程序迅速启动，而且也不需要使用的稀缺的RAM。
- 加载动态链接库：什么时候加载动态链接库的选择会明显影响系统性能。例如，当应用程序第一次加载到内存就加载所有的动态链接库，比在执行中不定时发生的加载更加容易接受。比起执行时应用程序发生延迟，用户更加能够接受启动它时有一些滞后。注意动态链接库可能并不需要加载，如果它们已经在内存中或者它们包含在外部闪存中（在这种情况下，它们可以就地执行）就是这种情况。
- 卸下内存管理给硬件：如果有MMU，尽可能地使用它。实际上，将越多的功能放入MMU，系统的性能越好。

即使使用就地执行的规则，较小的平台仍然需要保留一部分内存用作操作系统操作。这些内存与永

久性存储介质共享，并且通常以两种方法中的一种进行管理。首先，一些操作系统采用一种十分简单的方法，内存根本不分页。在这些类型的系统中，上下文切换意味着分配操作空间（比如堆空间），同时所有进程间共享这些操作空间。这种方法在进程的存储区域几乎没有保护，信任进程间可以很好地工作。Palm操作系统使用这种简单的方式进行内存管理。第二种方法是使用一种更加有规则的方法，内存被切分成为页，这些页按照需要分配。操作系统管理一个空闲列表来保存页，按照需要分配给操作系统和用户进程。在这种方法中（由于没有虚拟内存），当页的空闲列表用光时，系统就会没有内存，从而不会再有分配发生。Symbian操作系统是第二种方法的例子。

12.4.2 Symbian操作系统的寻址方式

由于Symbian操作系统是32位系统，因此寻址范围可以达到4GB。它与更大的系统一样使用同样的抽象方式：程序必须使用由操作系统映射到物理地址的虚拟地址。和大多数系统一样，Symbian操作系统把内存划分为虚拟页面和物理页框。页框的大小通常是4KB，但也是可变的。

因为最大具有4GB的地址空间，因此4KB的页框大小就意味着具有超过100万条目的页表。Symbian操作系统只有有限的内存，因此不能拿出1MB内存专用于页表。而且，对这么大的一张表的搜索和访问对系统都是很大的负担。为了解决这个问题，Symbian操作系统采用2级页表方式，如图12-2所示。称作页面目录的第一级提供一个到第二级的链接，可以使用虚拟地址的一部分进行检索（前12位）。该目录驻留在内存中，由TTBR（转换表基址寄存器）指向。每个目录条目指向第二级，也就是页表的集合。这些页表提供到某一内存中特定页的链接，由虚拟地址的一部分检索（中间8位）。最后，虚拟地址的低12位索引检索页的字。在这一虚拟-物理地址映射计算中，硬件起辅助作用。尽管Symbian操作系统不能假定任何辅助硬件的存在，但是在大多数体系中这一映射转换都是由MMU完成的。比如ARM处理器就具有扩展的MMU，带有转换后备缓冲器来辅助地址计算。

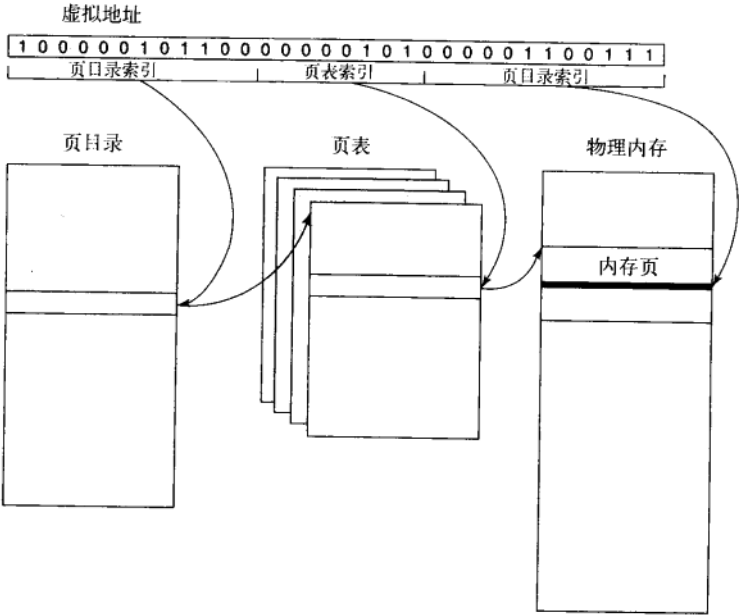


图12-2 Symbian操作系统使用2级页表来减少页表访问时间和占用空间

当一个页面不在内存中时，就会出现错误状态，这是因为当一个应用程序启动的时候所有的应用程序内存页面都应该已经被加载（没有请求页面调度）。链接到可执行应用中的小的桩代码显式地把动态加载库加载到内存中，而不是通过页失效方式。

Symbian操作系统中尽管没有页交换，但内存却不可思议地是动态的。应用程序通过内存进行上下文切换，同时正如上面所说的，当应用程序开始执行时将它们所需要的存储空间加载到内存中。每个应

用程序需要的内存页面能够从操作系统中静态请求进入内存。对于堆（更确切地说是动态空间）是有界限的，因此静态请求也可以由动态空间来实现。从一个空闲页框的列表中分配页框给页面；如果没有空闲页框，那么就会出现错误。正在使用的页框不能被刚到的应用程序的页面替换，即使该页框是针对当前没有执行的应用程序的。这是由于Symbian操作系统中没有页交换，同时因为十分有限的闪存空间只给用户文件使用，也就没有空间来复制被收回空间的页面。

实际上Symbian操作系统使用的存储实现模型有四种不同的版本。每种模型都是为了特定类型的硬件配置。一个简要的列表如下：

- 移动模型：该模型是为早期的ARM体系结构设计的。移动模型中的页目录是4KB长，每个条目4字节，给出一块16KB大小的目录。通过与页框相关的存取位和使用域来标志存储器访问的方式保护存储页面。域信息记录在页表目录，MMU为每个域实现访问权限。尽管没有明确使用分段，但是在内存布局上有如下结构：有用户分配数据的数据区，也有内核分配数据的内核区。
- 复合模型：该模型是为ARM6或者之后的体系开发的模式。这些版本中的MMU与以前版本使用的不同。例如，由于页表目录能分成两部分，每个部分索引页表的不同部分，因此需要不同的处理方式。这两部分分别用作用户页表和内核页表。ARM体系中新的版本修订并增强了每个页框的访问位，但是不赞成使用域的概念。
- 直接模型：该模型假定根本就没有MMU。这一模型很少使用，并且在智能手机上禁止使用。没有MMU会导致严重的性能问题。由于某些原因，MMU被禁止的一些场合中该模式比较有用。
- 仿真模型：该模型是为了支持基于windows宿主机的Symbian操作系统仿真器。仿真器与实际的目标CPU几乎没有区别。仿真器作为一个单独的Windows进程运行，因此地址空间被限定为2GB，而不是4GB。为仿真器提供的所有内存可以被任何Symbian操作系统进程访问，因此不具有内存保护。Symbian操作系统库以Windows格式的动态链接库形式提供，因此Windows处理内存的分配和管理。

12.5 输入和输出

Symbian操作系统的输入/输出结构仿照其他操作系统的设计。本节会指出其中一些Symbian操作系统特有的基于自己目标平台的性质。

12.5.1 设备驱动

在Symbian操作系统中，设备驱动作为具有内核权限的代码运行，从而赋予用户级别的代码对系统保护资源的访问能力。同Linux与Windows一样，设备驱动程序代表软件去访问硬件。

Symbian操作系统中的设备驱动分为两层：一个是逻辑设备驱动（LDD），一个是物理设备驱动（PDD）。LDD为上层软件提供一个接口，而PDD直接与硬件进行交互。在这种模型下，LDD可以为某一类特定的设备使用相同的实现，而PDD随着不同的设备改变。Symbian操作系统支持许多标准的LDD。有时，如果硬件非常标准或者常用，Symbian操作系统也提供PDD。

考虑串行设备的一个例子。Symbian操作系统定义了一个通用的串行LDD，该LDD定义了访问串行设备的程序接口。LDD给PDD提供一个接口，PDD提供串行设备访问接口。PDD实现有助于调节CPU和串行设备之间速度差异所必需的缓冲和流控制机制。一个单一的LDD（用户那边）可以连接任何用来运行串行设备的PDD。在某个特定的智能手机上，这些PDD可能包括一个红外端口或者一个RS-232端口。这两个是非常好的例子，它们使用相同的串行LDD，但是使用不同的PDD。

当LDD和PDD不在内存中时，它们可以由用户程序动态地加载进内存。程序编制工具能够检查是否需要加载。

12.5.2 内核扩展

内核扩展就是Symbian操作系统在引导时刻加载的驱动程序。由于它们是在引导时刻加载的，因此是与标准的设备驱动区别对待的特殊情况。

内核扩展与标准的设备驱动不同。大多数设备驱动是由LDD同成对的PDD实现的，在用户空间程序需要的时候加载。内核扩展在引导时刻加载，针对特定的设备，通常没有成对的PDD。

内核扩展是引导过程的一部分。这些特殊的设备驱动在调度器启动之后加载并且启动。它们执行对于操作系统非常重要的功能: DMA服务、显示管理、对外设的总线控制(例如USB总线)。之所以提供它们有两个原因。首先, 它与我们已经看作是微内核设计特征的面向对象设计抽象相称。其次, 它允许Symbian操作系统所处的不同平台运行专门的设备驱动, 从而不需要重新编译内核而使用硬件。

12.5.3 直接存储器访问

设备驱动经常使用DMA, Symbian操作系统支持DMA硬件的使用。DMA硬件包含一个控制一系列DMA通道的控制器。每个通道提供内存和设备间的单一方向的通信, 因此, 数据的双向传输需要两个DMA通道。至少有一对DMA通道是专用于显示LCD控制器的。此外, 大多数平台提供一定数量的常规DMA通道。

一旦一个设备把数据传送到内存, 就会激发一个系统中断。PDD为了传输设备使用DMA硬件提供的DMA服务, 这里传输设备是指与硬件接口的设备驱动的一部分。在PDD与DMA控制器之间, Symbian操作系统实现两层软件: 一个软件的DMA层, 一个与DMA硬件接口的内核扩展。DMA层把自身分成平台独立层和平台相关层。作为内核扩展, DMA层在引导进程中是内核启动的第一批设备驱动的一个。

由于特殊的原因, 对DMA的支持是比较复杂的。Symbian操作系统支持许多不同的硬件配置, 但是没有提供缺省的DMA配置。与DMA硬件的接口是标准化的, 由平台无关层来提供。平台相关层和内核扩展由生产厂商提供, 这样Symbian操作系统就如对其他设备一样处理DMA硬件: 在LDD和PDD构件中具有设备驱动。由于DMA硬件本身是一个设备, 并且它并行了Symbian操作系统支持所有设备的方式, 因此这种实现支持的方式是合理的。

12.5.4 特殊情况: 存储介质

Symbian操作系统中存储介质驱动是PDD的一种特殊形式, 文件服务器排他地使用它们来实现对存储介质设备的访问。因为智能手机既可以容纳固定的存储介质也可以容纳移动的存储介质, 所以存储介质驱动必须识别和支持多种形式的存储介质。Symbian操作系统对介质的支持包括一个标准的LDD和为用户提供的接口API。

Symbian操作系统中的文件服务器能够同时支持多达26个不同的设备。本地设备, 像在Windows中一样, 通过驱动器号来区分。

12.5.5 阻塞I/O

Symbian操作系统通过活动对象处理阻塞I/O。设计者认识到等待I/O事件的所有线程的负荷会影响系统中的其他线程这一事实。活动对象使得阻塞I/O调用可以由操作系统来处理, 而不是进程自身。活动对象由一个调度器进行协调并且在一个单独的线程中执行。

当活动对象使用一个阻塞I/O调用时, 它用信号通知操作系统并且把自身挂起。当调用完成时, 操作系统唤醒挂起的进程, 该进程如同带有数据返回的函数一样继续执行。区别只是对于活动对象的一个观点: 它不能调用一个函数并期待一个返回值; 它必须调用一个特殊的函数并且使该函数设置阻塞I/O, 但是立刻返回。操作系统接管等待过程。

12.5.6 可移动存储器

可移动存储器带给操作系统设计人员一个有趣的两难处境。当往读取槽插入一张安全数据(Secure Digital, SD)卡时, 该卡就同其他设备一样成为一个设备。它需要一个控制器、一个驱动、一种总线结构, 而且很有可能通过DMA与CPU进行通信。然而, 对这类模型移除存储介质是一个很严重的问题: 操作系统怎样检测插入和移除? 这一模型如何适应一张介质卡的不存在? 还有更加复杂的情况, 一些设备槽能够兼容不止一种类型的设备。例如, 一张SD卡, 一张miniSD卡(带有适配器), 以及一张多媒体卡都使用同一类插槽。

Symbian操作系统使用可移动存储器的很多共同性来实现对它们的支持。每种可移动存储器通常具有如下特点:

- 1) 所有的设备必须支持插入和移除。

- 2) 所有的可移动存储器能够“热”拔,也就是正在使用时被拔下。
- 3) 每种介质都能报告它自己的容量。
- 4) 必须拒绝不适配的卡。
- 5) 每种卡都需要电源。

为了支持可移动存储器, Symbian操作系统提供控制每种支持卡片的软件控制器。这些控制器和设备驱动工作在一起,这在软件层面上也是一样的。当一张卡插入时,就创建了一个套接字对象,该套接字对象构成数据流动过程中的通道。为了适应卡状态的变化, Symbian操作系统提供了一系列的当状态改变发生时的事件。设备驱动像活动对象一样被配置用来监听这些事件并作出反应。

12.6 存储系统

和所有面向用户的操作系统一样, Symbian操作系统有一个文件系统。下面我们来对其进行描述。

12.6.1 移动设备文件系统

就文件系统和存储而言,手机操作系统有很多和台式机操作系统相同的需求。多数的这类系统都现在32位硬件平台上;允许用户以任意的名字命名文件;大量存储文件,需要一定的组织结构。这意味着我们需要一个分层的、基于目录的文件系统。而且,手机操作系统设计人员有很多文件系统可以选择时,一个很重要的特性影响了他们的选择:大多数手机存储介质可以和Windows环境共享使用。

如果手机系统中没有可移动存储器件,则任一种文件系统都是可以使用的。但是,对于使用闪存作为存储的系统来说,还有特殊的情况需要考虑。存储块一般都是512字节到2048字节,但闪存不能直接修改数据记录,而需要先擦除数据,然后才能进行写入。另外,擦除的操作很不精确,每次擦除不能只擦除一个字节,而必须擦除整个块。擦除速度相对比较慢。

为了顺应这些特征,并且使闪存工作效率最高,需要文件系统能够把写操作分散到整个器件,以及解决较长的擦除时间问题。一个基本的概念是,当文件被更新时,文件系统会将文件的更新副本写入空闲的存储块并修改文件指针,而在有空闲时间时再进行旧数据块的回收操作。

最早的闪存文件系统之一是微软公司在20世纪90年代初为MS-DOS使用的FFS2文件系统。在1994年PCMCIA工业组织通过了关于闪存的闪存传输层(Flash Translation Layer)标准后,闪存器件可以被识别为一个FAT文件系统。Linux同时也专门为闪存设计了JFFS (Journaling Flash File System) 和YAFFS (Yet Another Flash Filing System) 两种文件系统。

但是,移动平台必须和其他计算机共享存储介质,这就要求必须有一定的兼容措施。FAT文件系统是最常用到的。而且,由于与FAT-32相比, FAT-16有着较小的分配表以及长文件的简化用法,所以FAT-16的使用更为广泛。

12.6.2 Symbian操作系统文件系统

作为智能手机操作系统, Symbian OS至少需要实现FAT-16文件系统。实际上,它的确支持FAT-16,并在大多数存储介质上使用。

但是, Symbian操作系统文件服务器是建立在一个类似Linux的虚拟文件系统的抽象层上的。面向对象技术允许多种文件系统的实现代码作为文件服务器的插件使用,于是允许同时使用多种文件系统。多种文件系统的实现代码可以在一个文件服务器中共存。

Symbian操作系统也支持NFS和SMB文件系统。

12.6.3 文件系统安全和保护

智能手机安全是通用计算机安全的一个有趣的变体。有很多侧面特征使得智能手机安全更富有挑战。Symbian操作系统在设计选择上有很多与通用计算平台和其他智能手机平台不同的地方。在这里我们只关注和文件系统安全有关的特征,其他方面将在下一节中进行讨论。

考虑到智能手机的环境,它们属于单用户设备,不需要在使用前进行用户认证。一个手机用户可以执行应用程序、拨打电话、访问网络,全都不需要用户认证。在这样的环境下,使用基于权限的安全措施是很有挑战性的,因为缺乏认证机制意味着只有一组权限可以使用,即所有人使用同样的一组权限。

除了权限,安全经常受益于其他形式的信息。在Symbian操作系统版本9或更新的版本中,应用程

序在安装时就被指定了其行为能力(授予一个应用程序权限的机制将在下一节涉及)。一个应用程序在请求执行某项行为时,其行为能力集将被检查。如果这种访问在行为能力集中存在,访问被许可,否则被拒绝。行为能力检查会造成一些系统开销——每次涉及到访问资源的系统调用都需要进行检查——但检查一个文件的所有者是否匹配的开销会更长。这个折中在Symbian操作系统中效果很好。

Symbian操作系统中还有一些其他形式的文件安全。在Symbian操作系统的存储器件中有特定的区域,需要有特定权限的应用程序才能访问。这种特定的权限只将安装程序赋予了应用程序。这样做的效果是,新安装的应用程序在安装完成后即被保护,不受任何非系统的访问(意味着非系统的恶意程序,如病毒,不能感染已经安装的程序)。另外,文件系统预留了专门保存应用程序产生的特殊数据的区域(这被称作数据锁定,见下一节)。

对Symbian操作系统来说,权限的使用和文件所有者在保护文件访问上的效果是相当的。

12.7 Symbian操作系统的安全

智能手机提供的环境很难保证安全。像我们之前提到的,它们属于单用户设备,不需要在使用基本功能前进行用户认证。更复杂的功能(如应用软件安装)需要授权,但不需认证。然而,智能手机上执行的复杂操作系统中,有很多途径进行数据的交换(以及执行程序)。在这样的环境进行安全防护变得很复杂。

Symbian操作系统很好地体现了这一安全难度。用户期望基于Symbian操作系统的智能手机允许不经认证即可任意使用——没有登录和身份鉴别。但是,你肯定经历过,一个和Symbian操作系统同样复杂的操作系统很容易受到病毒、蠕虫和其他恶意软件的影响。在Symbian操作系统版本9以前的版本中,操作系统提供了一个守门人式的安全功能:系统询问用户是否允许安装每一个应用程序。这种设计的思维是,只有用户自己安装的程序会造成系统毁坏,一个被告知的用户会知道他所要安装的哪些软件是恶意软件。用户会理智地使用它们。

守门人式设计有很多优点。例如:一个新的没有用户自己安装的应用程序的智能手机是一个可以无故障运行的系统。只安装用户认为不是恶意软件的程序,即可保证系统的安全。这种设计的问题是,用户并不总是知道安装一个应用程序的全部后果。存在伪装成有用的应用程序的病毒,在提供有用功能的同时静默地安装恶意代码。普通用户无法验证所有软件的可信度。

Symbian操作系统版本9的信任验证机制提升到了一个新设计的平台上。这个版本的操作系统保留原有的守门人式机制,但是在用户之外提供了对安装软件进行验证的机制。每个软件开发者现在需要负责通过数字签名技术来验证一个软件是由其编写的。不是所有的软件都必须有这样的验证,只有需要访问特定系统资源的软件需要。当一个应用软件需要数字签名时,需要如下几个步骤:

- 1) 软件开发者需要从可信的第三方获得一个厂商ID,这些可信的第三方由Symbian来进行鉴定。
- 2) 当一个开发者开发了一个程序包并希望发布时,他必须将其提交到可信的第三方进行验证。开发者提交其厂商ID、应用程序以及该应用程序访问系统的方式列表。
- 3) 可信第三方验证所提供的访问类型列表是完全的,而且没有其他类型的访问发生。如果该可信第三方可以进行此验证,该软件即由可信第三方进行签名。这意味着安装包中会包含一些特殊的信息,详细地描述该软件会对Symbian操作系统做出什么操作。
- 4) 该安装包被送回到软件开发人员处,并可以发放给用户。需要注意的是,这个方法依赖于应用程序如何访问系统资源。在Symbian操作系统中,应用程序必须拥有访问一个资源的能力,才会允许使用相应的资源。这种行为能力的机制建立在Symbian操作系统的内核中。当一个进程被创建时,该进程的进程控制块的一部分用来记录该进程被授予的权限。当进程试图使用它不能使用的权限时,该访问将被内核阻止。

这个看起来复杂的机制使得我们可以在Symbian操作系统中建立一个自动的守门人式机制,来验证要安装的软件。安装过程检查安装包中的标识。如果该标识是有效的,该应用程序被授予的权限将记录下来,同时可以在执行时通过内核的检查。

图12-3中的图描述了Symbian操作系统版本9中的信任关系。需要注意的是,系统中内置了多个信任等级。有些应用软件不访问任何系统资源,故而不需要签名。一个例子是只在屏幕上显示内容的简单

应用。这些应用软件不被也不需被信任。下一个信任级别是用户级签名应用程序级。这些应用程序只被授予其需要的权限。第三个信任等级由系统服务组成。同用户级应用程序一样，这些服务只需要特定的权限以便完成其任务。在一个如同Symbian操作系统的微内核体系结构中，这些服务运行在用户态，并像用户程序一样被信任。最后，有一类程序需要系统的完全信任。这组程序拥有修改整个系统的能力，并由内核代码组成。

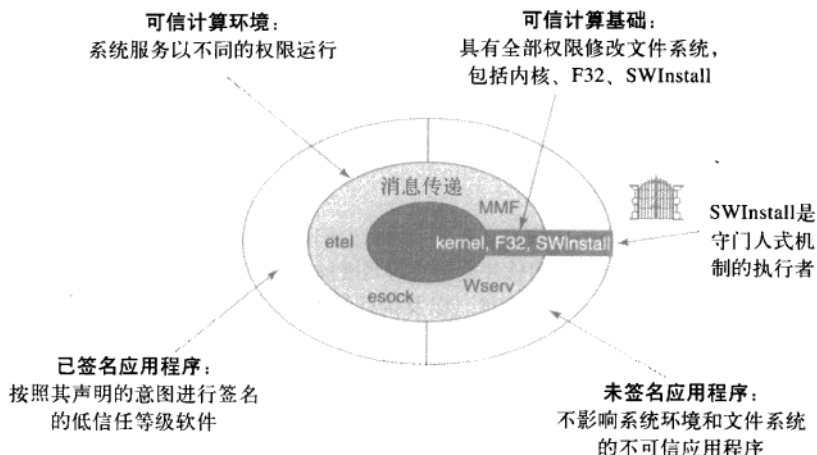


图12-3 Symbian操作系统通过信任关系来保证安全

在这个系统中有若干个方面看起来值得质疑。例如，这样复杂的机制真的有必要吗（尤其是需要花费金钱来制作的情况下）？结论是肯定的：Symbian签名系统代替用户来对软件进行完整性验证，并且该验证必须被执行。这一机制看起来可能会带来开发上的难度。是否每次在真实物理设备上进行测试都需要一个新的签名的安装包？为了解决这个问题，Symbian操作系统识别开发人员的特殊签名。一个开发人员必须获得一个有时效限制（通常是6个月）的证书和一个特殊的智能手机，即可使用自己的数字证书来创建安装包。

除了这样的守门人式机制外，Symbian操作系统版本9同时采用数据锁定（Data Caging）技术，来组织特定目录下的数据。比如，可执行代码只存在一个目录中，而该目录只对软件安装程序可写。另外，应用程序只能在一个目录中进行写操作，它们各自的数据不能被其他程序访问。

12.8 Symbian操作系统中的通信

Symbian操作系统按照特殊的标准设计，并使用客户机/服务器机制和基于栈的配置，以事件驱动型的通信为特色。

12.8.1 基本基础结构

Symbian操作系统的通信系统基础结构建立在基本构件之上。考虑如图12-4中所示的一个非常通用的模式。考虑把这个图作为一个可组织模型的起点。在这个栈的底层是物理设备，以一定方式链接到计算机。这个设备可以是集成在通信设备中的手机调制解调器或是一个蓝牙无线电装置。在此，我们不关心底层的硬件实现，而是把这个物理设备当做一个会以合适的方式响应软件发出的命令的抽象设备。

下一层，即我们需要关心的第一层，是设备驱动层。我们已经指出了设备驱动的结构；这一层的软件直接通过LDD和PDD结构与硬件配合工作。这一层的软件是硬件相关的，每个新型号的硬件设备都需要一个软件的设备驱动为其衔接。不同的硬件需要不同的设备驱动，但它们都为上层提供同样的接口。协议层期望无论什么样的硬件都具有相同的接口。

下一层就是协议实现层，包含了Symbian操作系统所支持的各种协议的实现。这些实现承担了下层的设备驱动接口，并向上面的应用层提供了一个单一、统一的接口。这就是提供诸如蓝牙和TCP/IP协议的各种协议的部分。

最后,应用程序层是最高的一层。该层包含了需要利用通信基础结构的应用程序。应用程序不清楚通信是怎样实现的,但是,该应用程序需要通知操作系统它需要使用哪个具体设备。一旦设备就位,应用程序不直接访问设备,而是依赖协议实现层的API来驱动真实硬件。

12.8.2 更仔细地观察基础结构

Symbian操作系统的一个更详细的结构如图12-5所示。这个图基于图12-4的通用结构,原图中的层被细分为Symbian操作系统中使用的可操作单元。

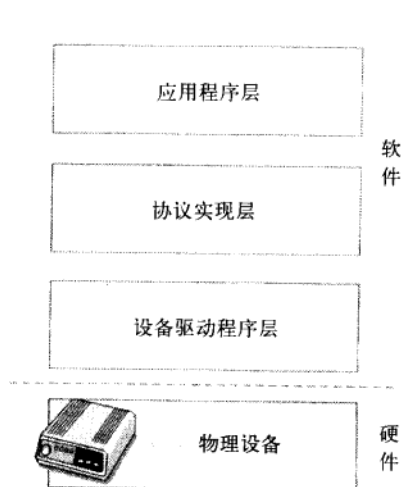


图12-4 Symbian操作系统中面向块的通信结构

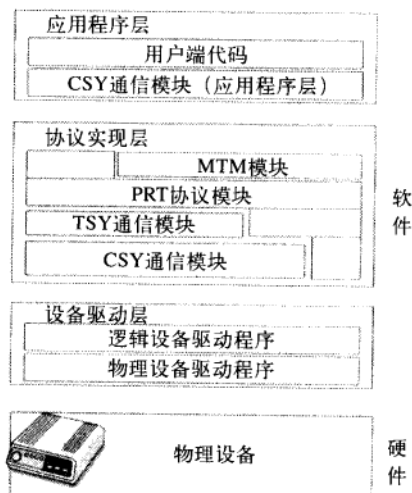


图12-5 Symbian操作系统中的通信设施具有一组丰富的功能

1. 物理设备

首先需要注意的是,物理设备层没有变化。如我们之前所述,Symbian操作系统并不直接控制硬件。所以,它兼容所有符合该层的API设计的硬件,但不需指定硬件本身是如何设计和建造的。这一点对Symbian操作系统和其开发人员都有益处。通过将硬件看作抽象结构并通过这一抽象进行通信,Symbian操作系统的设计人员保证了Symbian操作系统可以广泛地兼容现有设备,同时适应未来的硬件。

2. 设备驱动层

如图12-5所示,设备驱动层被分为两层。如我们之前所述,PPD层通过硬件端口直接与硬件设备进行交互。而LDD层与协议实现层交互,实现了Symbian操作系统中与硬件相关的策略。这些策略包括输入输出缓冲、中断机制和流控制。

3. 协议实现层

在图12-5中,协议实现层分为了若干子层。在协议实现层中使用了四种模块,列举在下面:

- CSY模块: 协议实现层最底层是通信服务,即CSY模块。一个CSY模块直接通过设备驱动程序的PDD部分与硬件通信,实现了协议的许多底层特征。例如,一个协议可能需要向硬件设备传递原始数据,或者需要在传输过程中使用7位或8位的缓存。这些工作模式会被CSY模块处理。
- TSY模块: 电话中包含了一大部分通信基础结构,这些功能需要由特殊的模块来实现。电话服务(TSY)模块实现了这些功能。基本的TSY可能在很多的硬件上支持标准的TSY,例如拨打和切断电话。更高级的TSY模块可以支持更高级的硬件,比如支持GSM功能。
- PRT模块: 协议实现层的核心模块是协议模块(PRT模块)。该模块由服务器用来实现具体的协议。一个服务器在试图使用协议的时候创建一个PRT模块的实例。例如,TCPIP.PRT模块中实现了TCP/IP相关的协议。蓝牙协议在BT.PRT模块中实现。
- MTM: 由于Symbian操作系统被设计用来处理短信息,设计人员专门为处理的所有类型的短信息

建立了相应的机制，而这些专门的模块称为信息类型模块（MTM模块）。短信息处理包括多个方面，MTM模块需要处理所有这些需求。用户界面类MTM模块需要实现多种供用户查看和处理短信息的方式，包括如何阅读短信息，如何被告知短信息发送进度等。客户端MTM模块处理寻址、创建、回复短信息；而服务器端MTM模块需要实现面向服务器的相关短信息管理功能，如目录管理、特定信息的管理等。

根据所使用通信类型的不同，这些模块以不同的方式彼此依赖。例如，实现使用蓝牙的协议，我们只需要物理器件上层的PRT模块即可。某些IrDA协议也是如此。而基于PPP的TCP/IP实现则需要使用PRT模块、TSY模块和CSY模块；不基于PPP的TCP/IP协议则不需要TSY模块和CSY模块，但是其PRT模块需要直接连接到网络设备驱动上。

4. 结构模块化

基于模块化的思想在这样一个栈式的模型实现中是很有用的。在这个分层的设计中，从例子中可以看出，抽象带来的优势是很明显的。考虑TCP/IP协议的实现。一个PPP连接既可以直接使用CSY模块，也可以选择GSM或普通调制解调器的TSY实现，后者实际底层仍由CSY模块来实现。未来新的电话技术出现后，当前的结构仍然可以起作用，我们只需要为新的电话实现添加一个TSY模块。另外，细调TCP/IP协议栈不需要修改任何其依赖的模块，只需要简单地调整TCP/IP PRT模块。这样广泛的模块化意味着在已有结构上很容易添加新代码、丢弃旧代码，当前代码的修改不会对整个系统带来巨大的变化，也不需要大量的重新安装。

最后，图12-5在应用层添加了子层。应用程序通过CSY模块和协议实现层中的协议模块进行交互。虽然我们可以认为这些模块属于协议实现层的一部分，但更清晰的表示是，这些模块在协助应用程序进行操作。例如，在使用红外接口将短信息发送到手机的过程中，应用程序会在应用程序中使用IRCOMM CSY模块，通过协议实现层的短信息实现模块来完成。同样，在这样一个过程中，模块化带来了很大的优势，应用程序可以关注实现其擅长的功能，而不是通过程序。

12.9 小结

Symbian操作系统是一个为智能手机平台设计的面向对象的操作系统。它的微内核设计只提供了很小的纳核，只实现了最快和最简单的内核功能。Symbian操作系统通过客户机/服务器的体系结构，将对系统资源的访问分配给用户态的服务器。Symbian操作系统虽然是为智能手机设计的，但其也拥有很多通用操作系统的特性：进程和线程、内存管理、文件系统支持、丰富的通信支持。同时，Symbian操作系统也实现了一些独特的特性，比如，活动对象使等待外部事件更为迅速、没有虚拟内存使得内存管理更富有挑战性、支持面向对象的设备驱动程序采用双层抽象结构。

习题

1. 对下列的每一个服务，描述其在如Symbian操作系统这样的微内核操作系统中，是在用户态还是内核态执行。
 - 调度线程的执行。
 - 打印一个文档。
 - 应答蓝牙搜索信号。
 - 管理线程对屏幕的访问。
 - 在短信息到达时发出声音。
 - 中断当前执行并接听电话。
2. 列举微内核设计带来的三个效率提升。
3. 列举微内核设计带来的三个效率问题。
4. Symbian操作系统将其内核分割为纳核和Symbian内核两部分。如动态内存管理之类的服务被认为过于复杂而不能进入纳核。描述动态内存管理中的复杂模块，解释为什么不能将其放进微内核。
5. 我们讨论过，活动对象使得I/O操作更有效率。你认为应用程序是否能够同时使用多个活动对象？系统在多个I/O事件发生时如何响应？
6. Symbian操作系统中的安全是否关注软件安装和应用程序的Symbian签名？这是否足够安全？是否会有某个场景，应用程序可以不必安装即被运行？（提示：考虑手机数据输入的所有可能方式）
7. 在Symbian操作系统中，广泛应用了基于服务的对共享资源的保护。列举三种在微内核环境下，这种方式协调资源的优势。思考这些优势对不同体系结构的影响。

第13章 操作系统设计

在过去的12章中，我们讨论了许多话题并且分析了许多与操作系统相关的概念和实例。但是研究现有的操作系统不同于设计一个新的操作系统。在本章中，我们将简述操作系统设计人员在设计与实现一个新系统时必须考虑的某些问题和权衡。

在系统设计方面，关于什么是好，什么是坏，存在着一定数量的民间传说在操作系统界流传，但是令人吃惊的是这些民间传说很少被记录下来。最重要的一本书可能是Fred Brooks的经典著作*The Mythical Man Month*（中文译名《人月神话》）。在这本书中，作者讲述了他设计与实现IBM OS/360系统时的经历。该书的20周年纪念版修订了某些素材并且新增加了4章（Brooks, 1995）。

有关操作系统设计的三篇经典论文是“Hints for Computer System Design”（计算机系统设计的忠告，Lampson, 1984）、“On Building Systems that Will Fail”（论建造将要失败的系统，Corbat6, 1991）和“End-to-End Arguments in System Design”（系统设计中端到端的论据，Saltzer等人，1984）。与Brooks的著作一样，这三篇论文都极其出色地经历了岁月的考验，其中的大多数真知灼见在今天仍然像文章首次发表时一样有效。

本章吸收了这些资料来源，另外加上了作者作为三个系统的设计者或合作设计者的个人经历，这三个系统是：Amoeba（Tanenbaum等人，1990）、MINIX（Tanenbaum和Woodhull, 1997）和Globe（Van Steen等人，1999a）。由于操作系统设计人员在设计操作系统的最优方法上没有达成共识，因此与前面各章相比，本章更加主观，也无疑更具有争议。

13.1 设计问题的本质

操作系统设计与其说是精确的科学，不如说是一个工程项目。设置清晰的目标并且满足这些目标非常困难。我们将从这些观点开始讨论。

13.1.1 目标

为了设计一个成功的操作系统，设计人员对于需要什么必须有清晰的思路。缺乏目标将使随后的决策非常难于做出。为了明确这一点，看一看两种程序设计语言PL/I和C会有所启发。PL/I是IBM公司在20世纪60年代设计的，因为在当时必须支持FORTRAN和COBOL是一件令人讨厌的事，同时令人尴尬的是，学术界背地里嚷嚷着Algol比这两种语言都要好。所以IBM设立了一个委员会来创作一种语言，该语言力图满足所有人的需要，这种语言就是PL/I。它具有一些FORTRAN的特点、一些COBOL的特点和一些Algol的特点。但是该语言失败了，因为它缺乏统一的洞察力。它只是彼此互相竞争的功能特性的大杂烩，并且过于笨重而不能有效地编译。

现在考察C语言。它是一个人（Dennis Ritchie）为了一个目的（系统程序设计）而设计的。C语言在所有的方面都取得了巨大的成功，因为Ritchie知道他需要什么，不需要什么。结果，在面世几十年之后，C语言仍然在广泛使用。对于需要什么要有一个清晰的洞察力是至关重要的。

操作系统设计人员需要什么？很明显，不同的系统会有所不同，嵌入式系统就不同于服务器系统。然而，对于通用的操作系统而言，需要留心4个基本的要素：

- 1) 定义抽象概念。
- 2) 提供基本操作。
- 3) 确保隔离。
- 4) 管理硬件。

下面将描述这些要素。

一个操作系统最重要但可能最困难的任务是定义正确的抽象概念。有一些抽象概念，例如进程和文

件，多年以前就已经提出来了，似乎比较显而易见。其他一些抽象概念，例如线程，还比较新鲜，就不那么成熟了。例如，如果一个多线程的进程有一个线程由于等待键盘输入而阻塞，那么由这个进程通过调用fork函数创建的新进程是否也包含一个等待键盘输入的线程？其他的抽象概念涉及同步、信号、内存模型、I/O的建模以及其他领域。

每一个抽象概念可以采用具体数据结构的形式实例化。用户可以创建进程、文件、信号量等。基本操作则处理这些数据结构。例如，用户可以读写文件。基本操作以系统调用的形式实现。从用户的观点来看，操作系统的核心是由抽象概念和其上的基本操作所构成的，而基本操作则可通过系统调用加以利用。

由于多个用户可以同时登录到一台计算机，操作系统需要提供机制将他们隔离。一个用户不可以干扰另一个用户。为了保护的目的，进程概念广泛地用于将资源集合在一起。文件和其他数据结构一般也是受保护的。确保每个用户只能在授权的数据上执行授权的操作是系统设计的关键目标。然而，用户还希望共享数据和资源，因此隔离必须是选择性的并且要在用户的控制之下。这就使问题更加复杂化了。电子邮件程序不应该弄坏Web浏览器程序，即使只有一个用户，不同的进程也应该隔离开来。

与这一要点密切相关的是需要隔离故障。如果系统的某一部分崩溃（最为一般的是一个用户进程崩溃），不应该使系统的其余部分随之崩溃。系统设计应该确保系统的不同部分良好地相互隔离。从理想的角度看，操作系统的各部分也应该相互隔离，以便使故障独立。

最后，操作系统必须管理硬件。特别地，它必须处理所有低级芯片，例如中断控制器和总线控制器。它还必须提供一个框架，从而使设备驱动程序得以管理更大规模的I/O设备，例如磁盘、打印机和显示器。

13.1.2 设计操作系统为什么困难

摩尔定律表明计算机硬件每十年改进100倍，但却没有一个定律宣称操作系统每十年改进100倍。甚至没有人能够宣称操作系统每十年在某种程度上会有所改善。事实上，可以举出事例，一些操作系统在很多重要的方面（例如可靠性）比20世纪70年代的UNIX版本7还要糟糕。

为什么会这样？大部分责任常常归咎于惯性和渴望向后兼容，不能坚持良好的设计原则也是问题的根源。但是还不止这些。操作系统在特定的方面根本不同于计算机商店以49美元销售的小型应用程序。我们下面就看一看使设计一个操作系统比设计一个应用程序要更加困难的8个问题。

第一，操作系统已经成为极其庞大的程序。没有一个人能够坐在一台PC机前在几个月内匆匆地完成一个严肃的操作系统。UNIX的所有当前版本都超过了300万行代码，Windows Vista有超过500万行的内核代码（全部代码超过7亿行）。没有一个人能够理解300万到500万行代码，更不必说7亿行代码。当你拥有一件产品，如果没有一名设计师能够有望完全理解它时，结果经常远没有达到最优也就不难预料了。

操作系统不是世界上最复杂的系统，例如，航空母舰就要复杂得多，但是航空母舰能够更好地分成相互隔离的部分。设计航空母舰上的卫生间的人员根本不必关心雷达系统，这两个子系统没有什么相互作用。而在操作系统中，文件系统经常以意外的和无法预料的方式与内存系统相互作用。

第二，操作系统必须处理并发。系统中往往存在多个用户和多个设备同时处于活动状态。管理并发自然要比管理单一的顺序活动复杂得多。竞争条件和死锁只是出现的问题中的两个。

第三，操作系统必须处理可能有敌意的用户——想要干扰系统的用户或者做不允许做的事情（例如偷窃另一个用户的文件）的用户。操作系统需要采取措施阻止这些用户不正当的行为，而字处理程序和照片编辑程序就不存在这样的问题。

第四，尽管事实上并非所有的用户都相信其他用户，但是许多用户确实希望与经过选择的其他用户共享他们的信息和资源。操作系统必须使其成为可能，但是要以确保怀有恶意的用户不能妨害的方式。而应用程序就不会面对类似这样的挑战。

第五，操作系统已经问世很长时间了。UNIX已经历了四分之一世纪，Windows面世也已经超过二十年并且还没有消退的迹象。因此，设计人员必须思考硬件和应用程序在遥远的未来可能会发生的变化，并且考虑为这样的变化做怎样的准备。紧密地局限于世界的一个特定视野的系统通常不会存世太久。

第六，操作系统设计人员对于他们的系统将怎样被人使用实际上并没有确切的概念，所以他们需要提供相当程度的通用性。UNIX和Windows在设计时都没有把电子邮件或Web浏览器放在心上，然而许多运行这些系统的计算机却很少做其他的事情。人们在告诉一名轮船设计师建造一艘轮船时，却会指明他

想要的是渔船、游船还是战舰，并且当产品生产出来之后鲜有人会改变产品的用途。

第七，现代操作系统一般被设计成可移植的，这意味着它们必须运行在多个硬件平台上。它们还必须支持上千个I/O设备，所有这些I/O设备都是独立设计的，彼此之间没有关系。这样的差异可能会导致问题，一个例子是操作系统需要运行在小端机器和大端机器上。第二个例子经常在MS-DOS下看到，用户试图安装一块声卡和一个调制解调器，而它们使用了相同的I/O端口或者中断请求线。除了操作系统以外，很少有程序必须处理由于硬件部件冲突而导致的这类问题。

第八，也是最后一个问题，是经常需要与某个从前的操作系统保持向后兼容。以前的那个系统可能在字长、文件名或者其他方面有所限制，而在设计人员现在看来这些限制都是过时的，但是却必须坚持。这就像让一家工厂转而去生产下一年的汽车而不是这一年的汽车的同时，继续全力地去生产这一年的汽车。

13.2 接口设计

到现在读者应该清楚，编写一个现代操作系统并不容易。但是人们要从何处开始呢？可能最好的起点是考虑操作系统提供的接口。操作系统提供了一组抽象，主要是数据类型（例如文件）以及其上的操作（例如read）。它们合起来形成了对用户的接口。注意，在这一上下文中操作系统的用户是指编写使用系统调用的代码的程序员，而不是运行应用程序的人员。

除了主要的系统调用接口，大多数操作系统还具有另外的接口。例如，某些程序员需要编写插入到操作系统中的设备驱动程序。这些驱动程序可以看到操作系统的某些功能特性并且能够发出某些过程调用。这些功能特性和调用也定义了接口，但是与应用程序员看到的接口完全不同。如果一个系统要取得成功，所有这些接口都必须仔细地设计。

13.2.1 指导原则

有没有指导接口设计的原则？我们认为有的。简而言之，原则就是简单、完备和能够有效地实现。

原则1：简单

一个简单的接口更加易于理解并且更加易于以无差错的方式实现。所有的系统设计人员都应该牢记法国先驱飞行家和作家Antoine de St. Exupéry的著名格言：

不是当没有东西可以再添加，而是当没有东西可以再裁减时，才能达到尽善尽美。

这一原则说的是少比多好，至少在操作系统本身中是这样。这一原则的另一种说法是KISS原则：Keep It Simple, Stupid（保持简朴无华）。

原则2：完备

当然，接口必须能够做用户需要做的一切事情，也就是说，它必须是完备的。这使我们想起了另一条著名的格言，Albert Einstein（阿尔伯特·爱因斯坦）说过：

万事都应该尽可能简单，但是不能过于简单。

换言之，操作系统应该不多不少准确地做它需要做的事情。如果用户需要存储数据，它就必须提供存储数据的机制；如果用户需要与其他用户通信，操作系统就必须提供通信机制；如此等等。1991年，CTSS和MULTICS的设计者之一Fernando Corbató在他的图灵奖演说中，将简单和完备的概念结合起来并且指出：

首先，重要的是强调简单和精练的价值，因为复杂容易导致增加困难并且产生错误，正如我们已经看到的那样。我对精练的定义是以机制的最少化和清晰度的最大化实现指定的功能。

此处重要的思想是机制的最少化（minimum of mechanism）。换言之，每一个特性、功能和系统调用都应该尽自己的本分。它应该做一件事情并且把它做好。当设计小组的一名成员提议扩充一个系统调用或者添加某些新的特性时，其他成员应该问这样的问题：“如果我们省去它会不会发生可怕的事情？”如果回答是：“不会，但是有人可能会在某一天发现这一特性十分有用”，那么请将其放在用户级的库中，而不是操作系统中，尽管这样做可能会使速度慢一些。并不是所有的特性都要比高速飞行的子弹还要快。目标是保持Corbató所说的机制的最少化。

让读者简略地看一看我亲身经历的两个例子：MINIX（Tanenbaum和Woodhull，2006）和Amoeba（Tanenbaum等人，1990）。实际上，MINIX具有三个系统调用：send、receive和sendrec。系统是作为一组进程的集合而构造的，内存管理、文件系统以及每个设备驱动程序都是单独的可调度的进程。作为

首要的近似,内核所做的全部工作只是调度进程以及处理在进程之间传递的消息。因此,只需要两个系统调用:send发送一条消息,而receive接收一条消息。第三个调用sendrec只是为了效率的原因而做的优化,它使得仅用一次内核陷阱就可以发送一条消息并且请求应答。其他的一切事情都是通过请求某些其他进程(例如文件系统进程或磁盘驱动程序)做相应的工作而完成的。

Amoeba甚至更加简单。它仅有一个系统调用:执行远程过程调用。该调用发送一条消息并且等待一个应答。它在本质上与MINIX的sendrec相同。其他的一切都建立在这一调用的基础上。

原则3:效率

第三个指导方针是实现效率。如果一个功能特性或者系统调用不能够有效地实现,或许就不值得包含它。对于程序员来说,一个系统调用的代价有多大也应该在直觉上是显而易见的。例如,UNIX程序员会认为lseek系统调用比read系统调用要代价低廉,因为前者只是在内存中修改一个指针,而后者则要执行磁盘I/O。如果直觉的代价是错误的,程序员就会写出效率差的程序。

13.2.2 范型

一旦确定了目标,就可以开始设计了。一个好的起点是考虑客户将怎样审视该系统。最为重要的问题之一是如何将系统的所有功能特性良好地结合在一起,并且展现出经常所说的体系结构一致性(architectural coherence)。在这方面,重要的是区分两种类型的操作系统“客户”。一方面,是用户,他们与应用程序打交道;另一方面,是程序员,他们编写应用程序。前者主要涉及GUI,后者主要涉及系统调用接口。如果打算拥有遍及整个系统的单一GUI,就像在Macintosh中那样,设计应该在此处开始。然而,如果打算支持许多可能的GUI,就像在UNIX中那样,那么就应该首先设计系统调用接口。首先设计GUI本质上是自顶向下的设计。这时的问题是GUI要拥有什么功能特性,用户将怎样与它打交道,以及为了支持它应该怎样设计系统。例如,如果大多数程序在屏幕上显示图标然后等待用户在其上点击,这暗示着GUI应该采用事件驱动模型,并且操作系统或许也应该采用事件驱动模型。另一方面,如果屏幕主要被文本窗口占据,那么进程从键盘读取输入的模式可能会更好。

首先设计系统调用接口是自底向上的设计。此时的问题是程序员通常需要哪些种类的功能特性。实际上,并不是需要许多特别的功能特性才能支持一个GUI。例如,UNIX窗口系统X只是一个读写键盘、鼠标和屏幕的大C程序。X是在UNIX问世很久以后才开发的,但是并不要求对操作系统做很多修改就可以使它工作。这一经验验证了这样的事实:UNIX是十分完备的。

1. 用户界面范型

对于GUI级的接口和系统调用接口而言,最重要的方面是有一个良好的范型(有时称为隐喻),以提供观察接口的方法。台式计算机的许多GUI使用我们在第5章讨论过的WIMP范型。该范型在遍及接口的各处使用定点-点击、定点-双击、拖动以及其他术语,以提供总体上的体系结构一致性。对于应用程序常常还有额外的要求,例如要有一个具有文件(FILE)、编辑(EDIT)以及其他条目的菜单栏,每个条目具有某些众所周知的菜单项。这样,熟悉一个程序的用户就能够很快地学会另一个程序。

然而,WIMP用户界面并不是惟一可能的用户界面。某些掌上型计算机使用一种程式化的手写界面。专用的多媒体设备可能使用像VCR一样的界面。当然,语音输入具有完全不同的范型。重要的不是选择这么多的范型,而是存在一个单一的统领一切的范型统一整个用户界面。

不管选择什么范型,重要的是所有应用程序都要使用它。因此,系统设计者需要提供库和工具包给应用程序开发人员,使他们能够访问产生一致的外观与感觉的过程。用户界面设计非常重要,但它并不是本书的主题,所以我们现在要退回到操作系统接口的主题上。

2. 执行范型

体系结构一致性不但在用户层面是重要的,在系统调用接口层面也同样重要。在这里区分执行范型和数据范型常常是有益的,所以我们将讨论两者,我们以前者为开始。

两种执行范型被广泛接受:算法范型和事件驱动范型。算法范型(algorithmic paradigm)基于这样的思想:启动一个程序是为了执行某个功能,而该功能是事先知道的或者是从其参数获知的。该功能可能是编译一个程序、编制工资册,或者是将一架飞机飞到旧金山。基本逻辑被硬接线到代码当中,而程序则时常发出系统调用获取用户输入、获得操作系统服务等。图13-1a中概括了这一方法。

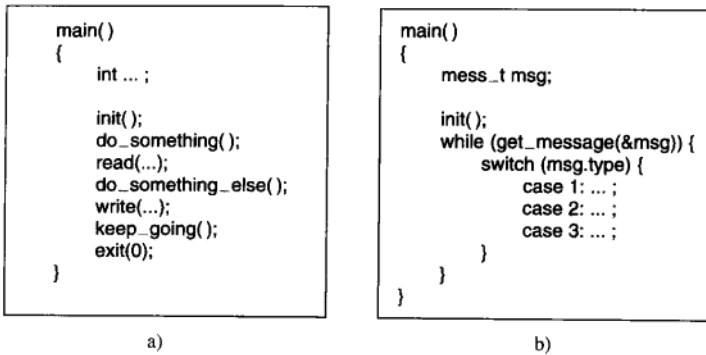


图13-1 a) 算法代码；b) 事件驱动代码

另一种执行范型是图13-1b所示的事件驱动范型 (event-driven paradigm)。在这里程序执行某种初始化 (例如通过显示某个屏幕)，然后等待操作系统告诉它第一个事件。事件经常是键盘敲击或鼠标移动。这一设计对于高度交互式的程序是十分有益的。

这些做事的每一种方法造就了其特有的程序设计风格。在算法范型中，算法位居中心而操作系统被看作是服务提供者。在事件驱动范型中，操作系统同样提供服务，但是这一角色与作为用户行为的协调者和被进程处理的事件的生产者相比就没那么重要了。

3. 数据范型

执行范型并不是操作系统导出的惟一范型，同等重要的范型是数据范型。这里关键的问题是系统结构和设备如何展现给程序员。在早期的FORTRAN批处理系统中，所有一切都是作为连续的磁带而建立模型。用于读入的卡片组被看作输入磁带，用于穿孔的卡片组被看作输出磁带，并且打印机输出被看作输出磁带。磁盘文件也被看作磁带。对一个文件的随机访问是可能的，只要将磁带倒带到对应的文件并且再次读取就可以了。

使用作业控制卡片可以这样来实现映射：

```
MOUNT(TAPE08, REEL781)
```

```
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

第一张卡片指示操作员去从磁带架上取得磁带卷781，并且将其安装在磁带驱动器8上。第二张卡片指示操作系统运行刚刚编译的FORTRAN程序，映射INPUT（意指卡片阅读机）到逻辑磁带1，映射磁盘文件MYDATA到逻辑磁带2，映射打印机（称为OUTPUT）到逻辑磁带3，映射卡片穿孔机（称为PUNCH）到逻辑磁带4，并且映射物理磁带驱动器8到逻辑磁带5。

FORTRAN具有读写逻辑磁带的语法。通过读逻辑磁带1，程序获得卡片输入。通过写逻辑磁带3，输出随后将会出现在打印机上。通过读逻辑磁带5，磁带卷781将被读入，如此等等。注意，磁带概念只是集成卡片阅读机、打印机、穿孔机、磁盘文件以及磁带的范型。在这个例子中，只有逻辑磁带5是一个物理磁带，其余的都是普通的（假脱机）磁盘文件。这只是一个原始的范型，但它却是正确方向上的一个开端。

后来，UNIX问世了，它采用“所有一切都是文件”的模型进一步发展了这一思想。使用这一范型，所有I/O设备都被看作是文件，并且可以像普通文件一样打开和操作。C语句

```
fd1 = open("file1", O_RDWR);
```

```
fd2 = open("/dev/tty", O_RDWR);
```

打开一个真正的磁盘文件和用户终端。随后的语句可以使用fd1和fd2分别读写它们。从这一时刻起，在访问文件和访问终端之间并不存在差异，只不过在终端上寻道是不允许的。

UNIX不但统一了文件和I/O设备，它还允许像访问文件一样通过管道访问其他进程。此外，当支持映射文件时，一个进程可以得到其自身的虚拟内存，就像它是一个文件一样。最后，在支持/proc文件系统的UNIX版本中，C语句

```
fd3 = open("/proc/501", O_RDWR);
```

允许进程（尝试）访问进程501的内存，使用文件描述符fd3进行读和写，这在某种程度上是有益的，例如对于一个调试器。

Windows Vista更进一步，它试图使所有一切看起来像是一个对象。一旦一个进程获得了一个指向文件、进程、信号量、邮箱或者其他内核对象的有效句柄，它就可以在其上执行操作。这一范型甚至比UNIX更加一般化，并且比FORTRAN要一般化得多。

统一的范型还出现在其他上下文中，其中在这里值得一提的是Web。Web背后的范型是充满了文档的超空间，每一个文档具有一个URL。通过键入一个URL或者点击被URL所支持的条目，你就可以得到该文档。实际上，许多“文档”根本就不是文档，而是当请求到来时由程序或者命令行解释器脚本生成的。例如，当用户询问一家网上商店关于一位特定艺术家的CD清单时，文档由一个程序即时生成；在查询未做出之前该文档的确并不存在。

至此我们已经看到了4种事例，即所有一切都是磁带、文件、对象或者文档。在所有这4种事例中，意图是统一数据、设备和其他资源，从而使它们更加易于处理。每一个操作系统都应该具有这样的统一数据范型。

13.2.3 系统调用接口

如果一个人相信Corbat6的机制最少化的格言，那么操作系统应该提供恰好够用的系统调用，并且每个系统调用都应该尽可能简单（但不能过于简单）。统一的数据范型在此处可以扮演重要的角色。例如，如果文件、进程、I/O设备以及更多的东西都可以看作是文件或者对象，那么它们就都能够用单一的read系统调用来读取。否则，可能就有必要具有read_file、read_proc以及read_tty等单独的系统调用。

在某些情况下，系统调用可能看起来需要若干变体，但是通常更好的实现是具有处理一般情况的一个系统调用，而由不同的库过程向程序员隐藏这一事实。例如，UNIX具有一个系统调用exec，用来覆盖一个进程的虚拟地址空间。最一般的调用是：

```
exec(name, argv, envp);
```

该调用加载可执行文件name，并且给它提供由argv所指向的参数和envp所指向的环境变量。有时明确地列出参数是十分方便的，所以库中包含如下调用的过程：

```
execl(name, argv, ..., argv, 0);  
execle(name, argv, ..., argv, envp);
```

所有这些过程所做的事情是将参数粘连在一个数组中，然后调用exec来做工作。这一安排达到了双赢目的：单一的直接系统调用使操作系统保持简单，而程序员得到了以各种方法调用exec的便利。

当然，试图拥有一个调用来处理每一种可能的情况很可能难以控制。在UNIX中，创建一个进程需要两个调用：fork然后是exec，前者不需要参数，后者具有3个参数。相反，创建一个进程的Win32 API调用CreateProcess具有10个参数，其中一个参数是指向一个结构的指针，该结构具有另外18个参数。

很久以前，有人曾经问过这样的问题：“如果我们省略了这些东西会不会发生可怕的事情？”诚实的回答应该是：“在某些情况下程序员可能不得不做更多的工作以达到特定的效果，但是最终的结果将会是一个更简单、更小巧并且更可靠的操作系统。”当然，主张10+18个参数版本的人可能会说：“但是用户喜欢所有这些特性。”对此的反驳可能会是：“他们更加喜欢使用很少内存并且从来不会崩溃的系统。”在更多功能性和更多内存代价之间的权衡是显而易见的，并且可以从价格上来衡量（因为内存的价格是已知的）。然而，每年由于某些特性而增加的崩溃次数是难于估算的，并且如果用户知道了隐藏的代价是否还会做出同样的选择呢？这一影响可以在Tanenbaum软件第一定律中做出总结：

添加更多的代码就是添加更多的程序错误。

添加更多的功能特性就要添加更多的代码，因此就要添加更多的程序错误。相信添加新的功能特性而不会添加新的程序错误的程序员要么是计算机的生手，要么就是相信牙齿仙女（据说会在儿童掉落在枕边的牙齿旁放上钱财的仙女）正在那里监视着他们。

简单不是设计系统调用时出现的惟一问题。一个重要的考虑因素是Lampson（1984）的口号：

不要隐藏能力。

如果硬件具有极其高效的方法做某事，它就应该以简单的方法展露给程序员，而不应该掩埋在某些其他抽象的内部。抽象的目的是隐藏不合需要的特性，而不是隐藏值得需要的特性。例如，假设硬件具有一种特殊的方法以很高的速度在屏幕上（也就是视频RAM中）移动大型位图，正确的做法是要有一个新的系统调用能够得到这一机制，而不是只提供一种方法将视频RAM读到内存中并且再将其写回。新的系统调用应该只是移动位而不做其他事情。如果系统调用速度很快，用户总可以在其上建立起更加方便的接口。如果它的速度慢，没有人会使用它。

另一个设计问题是面向连接的调用与无连接的调用。读文件的标准UNIX系统调用和Win32系统调用是面向连接的。首先你要打开一个文件，然后读它，最后关闭它。某些远程文件访问协议也是面向连接的。例如，要使用FTP，用户首先要登录到远程计算机上，读文件，然后注销。

另一方面，某些远程文件访问协议是无连接的，例如Web协议（HTTP）。要读一个Web页面你只要请求它就可以了；不存在事先建立连接的需要（TCP连接是需要的，但是这处于协议的低层；访问Web本身的HTTP协议是无连接的）。

任何面向连接的机制与无连接的机制之间的权衡在于建立连接的机制（例如打开文件）要求的额外开销，以及在后续调用（可能很多）中避免进行连接所带来的好处。对于单机上的文件I/O而言，由于建立连接的代价很低，标准的方法（首先打开，然后使用）可能是最好的方法。对于远程文件系统而言，两种方法都可以采用。

与系统调用接口有关的另一个问题是接口的可见性。POSIX强制的系统调用列表很容易找到。所有UNIX系统都支持这些系统调用，以及少数其他系统调用，但是完全的列表总是公开的。相反，Microsoft从未将Windows Vista系统调用列表公开。作为替代，Win32 API和其他API被公开了，但是这些API包含大量的库调用（超过10 000个），只有很少数是真正的系统调用。将所有系统调用公开的论据是可以让程序员知道什么是代价低廉的（在用户空间执行的函数），什么是代价昂贵的（内核调用）。不将它们公开的论据是这样给实现提供了灵活性，无须破坏用户程序就可以修改实际的底层系统调用，以便使其工作得更好。

13.3 实现

看过用户界面和系统调用接口后，现在让我们来看一看如何实现一个操作系统。在下面8个小节，我们将分析涉及实现策略的某些一般的概念性问题。在此之后，我们将看一看某些低层技术，这些技术通常是十分有益的。

13.3.1 系统结构

实现必须要做出的第一个决策可能是系统结构应该是什么。我们在1.7节分析了主要的可能性，在这里要重温一下。一个无结构的单块式设计实际上并不是一个好主意，除非可能是用于电冰箱中的微小的操作系统，但是即使在这里也是可争论的。

1. 分层系统

多年以来很好地建立起来的一个合理的方案是分层系统。Dijkstra的THE系统（图1-25）是第一个分层操作系统。UNIX和Windows Vista也具有分层结构，但是在这两个系统中分层更是一种试图描述系统的方法，而不是用于建立系统的真正的指导原则。

对于一个新系统，选择走这一路线的设计人员应该首先非常仔细地选择各个层次，并且定义每个层次的功能。底层应该总是试图隐藏硬件最糟糕的特异性，就像图11-7中HAL所做的那样。或许下一层应该处理中断、上下文切换以及MMU，从而在这一层的代码大部分是与机器无关的。在这一层之上，不同的设计人员可能具有不同的口味（与偏好）。一种可能性是让第3层管理线程，包括调度和线程间同步，如图13-2所示。此处的思想是从第4层开始，我们拥有适当的线程，这些线程可以被正常地调度，并且使用标准的机制（例如互斥量）进行同步。

在第4层，我们可能会找到设备驱动程序，每个设备驱动程序作为一个单独的线程而运行，具有自己的状态、程序计数器、寄存器等，可能（但是不必要）处于内核地址空间内部。这样的设计可以大大

简化I/O结构，因为当一个中断发生时，它就可以转化成在一个互斥量上的unlock，并且调用调度器以（潜在地）调度重新就绪的线程，而该线程曾阻塞在该互斥量之上。MINIX使用了这一方案，但是在UNIX、Linux和Windows Vista中，中断处理程序运行在一类“无主地带”中，而不是作为适当的线程可以被调度、挂起等。由于任何一个操作系统的大多数复杂性在于I/O之中，使其更加易于处理和封装的任何技术都是值得考虑的。

在第4层之上，我们预计会找到虚拟内存、一个或多个文件系统以及系统调用接口。如果虚拟内存处于比文件系统更低的层次，那么数据块高速缓存就可以分页出去，使虚拟内存管理器能够动态地决定在用户页面和内核页面（包括高速缓存）之间应该怎样划分实际内存。Windows Vista就是这样工作的。

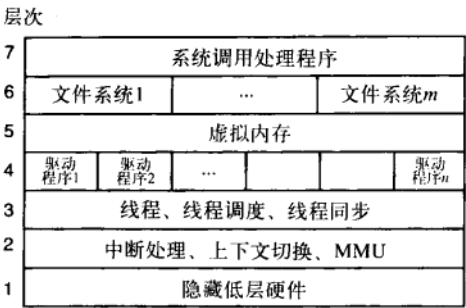


图13-2 现代分层操作系统的一种可能的设计

2. 外内核

虽然分层在系统设计人员中间具有支持者，但是还有另一个阵营恰恰持有相反的观点（Engler等人，1995）。他们的观点基于端到端的论据（end-to-end argument）（Saltzer等人，1984）。这一概念说的是，如果某件事情必须由用户程序本身去完成，在一个较低的层次做同样的事情就是浪费。

考虑该原理对于远程文件访问的一个应用。如果一个系统担心数据在传送中被破坏，它应该安排每个文件在写的时候计算校验和，并且校验和与文件一同存放。当一个文件通过网络从源盘传送到目标进程时，校验和也被传送，并且在接收端重新计算。如果两者不一致，文件将被丢弃并且重新传送。

校验比使用可靠的网络协议更加精确，因为除了位传送错误以外，它还可以捕获磁盘错误、内存错误、路由器中的软件错误以及其他错误。端到端的论据宣称使用一个可靠的网络协议是不必要的，因为端点（接收进程）拥有足够的信息以验证文件本身的正确性。在这一观点中，使用可靠的网络协议的惟一原因是为了效率，也就是说，更早地捕获与修复传输错误。

端到端的论据可以扩展到几乎所有操作系统。它主张不要让操作系统做用户程序本身可以做的任何事情。例如，为什么要有一个文件系统？只要让用户以一种受保护的方式读和写原始磁盘的一个部分就可以了。当然，大多数用户喜欢使用文件，但是端到端的论据宣称，文件系统应该是与需要使用文件的任何程序相链接的库过程。这一方案使不同的程序可以拥有不同的文件系统。这一论证线索表明操作系统应该做的全部事情是在竞争的用户之间安全地分配资源（例如CPU和磁盘）。Exokernel是一个根据端到端的论据建立的操作系统（Engler 等人，1995）。

3. 基于微内核的客户-服务器系统

在让操作系统做每件事情和让操作系统什么也不做之间的折衷是让操作系统做一点事情。这一设计导致微内核的出现，它让操作系统的大部分作为用户级的服务器进程而运行，如图13-3所示。在所有设计中这是最模块化和最灵活的。在灵活性上的极限是让每个设备驱动程序也作为一个用户进程而运行，从而完全保护内核和其他驱动程序，但是让设备驱动程序运行在内核会增加模块化程度。

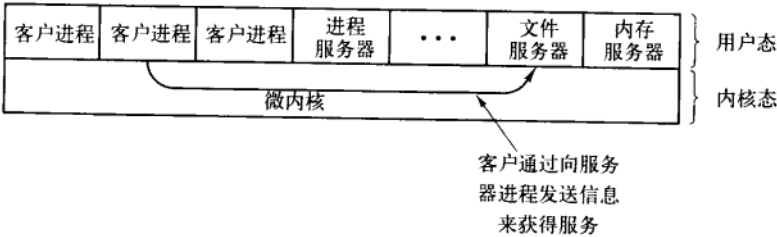


图13-3 基于微内核的客户-服务器计算

当设备驱动程序运行在内核态时，可以直接访问硬件设备寄存器，否则需要某种机制以提供这样的

访问。如果硬件允许，可以让每个驱动程序进程仅访问它需要的那些I/O设备。例如，对于内存映射的I/O，每个驱动程序进程可以拥有页面将它的设备映射进来，但是没有其他设备的页面。如果I/O端口空间可以部分地加以保护，就可以保证只有相应的正确部分对每个驱动程序可用。

即使没有硬件帮助可用，仍然可以设法使这一思想可行。此时需要的是一个新的系统调用，该系统调用仅对设备驱动程序进程可用，它提供一个（端口，取值）对列表。内核所做的是首先进行检查以了解进程是否拥有列表中的所有端口，如果是，它就将相应的取值复制到端口以发起设备I/O。类似的调用可以用一种受保护的方式读I/O端口。

这一方法使设备驱动程序避免了检查（并且破坏）内核数据结构，这（在很大程度上）是一件好事情。一组类似的调用可以用来让驱动程序进程读和写内核表格，但是仅以一种受控的方式并且需要内核的批准。

这一方法的主要问题，并且一般而言是针对微内核的主要问题，是额外的上下文切换导致性能受到影响。然而，微内核上的所有工作实际上是许多年前当CPU还非常缓慢的时候做的。如今，用尽CPU的处理能力并且不能容忍微小性能损失的应用程序是十分稀少的。毕竟，当运行一个字处理器或Web浏览器时，CPU可能有95%的时间是空闲的。如果一个基于微内核的操作系统将一个不可靠的3GHz的系统转变为一个可靠的2.5GHz的系统，可能很少有用户会抱怨。毕竟，仅仅在几年以前当他们得到具有1GHz的速度（就当时而言十分惊人）的系统时，大多数用户是相当快乐的。

4. 可扩展的系统

对于上面讨论的客户-服务器系统，思想是让尽可能多的东西脱离内核。相反的方法是将更多的模块放到内核中，但是以一种“受保护的”方式。当然，这里的关键字是“受保护的”。我们在9.5.6节中研究了某些保护机制，这些机制最初打算用于通过Internet引入小程序，但是对于将外来的代码插入到内核中的过程同样适用。最重要的是沙盒技术和代码签名，因为解释对于内核代码来说实际上是不可行的。

当然，可扩展的系统自身并不是构造一个操作系统的方法。然而，通过以一个只是包含保护机制的最小系统为开端，然后每次将受保护的模块添加到内核中，直到达到期望的功能，对于手边的应用而言一个最小的系统就建立起来了。按照这一观点，对于每一个应用，通过仅仅包含它所需要的部分，就可以裁剪出一个新的操作系统。Paramecium就是这类系统的一个实例（Van Doorn, 2001）。

5. 内核线程

此处，另一个相关的问题是系统线程，无论选择哪种结构模型。有时允许存在与任何用户进程相隔离的内核线程是很方便的。这些线程可以在后台运行，将脏页面写入磁盘，在内存和磁盘之间交换进程，如此等等。实际上，内核本身可以完全由这样的线程构成，所以当用户发出系统调用时，用户的线程并不是在内核模式中运行，而是阻塞并且将控制传给一个内核线程，该内核线程接管控制以完成工作。

除了在后台运行的内核线程以外，大多数操作系统还要启动许多守护进程。虽然这些守护进程不是操作系统的组成部分，但是它们通常执行“系统”类型的活动。这些活动包括接收和发送电子邮件，并且对远程用户各种各样的请求进行服务，例如FTP和Web网页。

13.3.2 机制与策略

另一个有助于体系结构一致性的原理是机制与策略的分离，该原理同时还有助于使系统保持小型和良好的结构。通过将机制放入操作系统而将策略留给用户进程，即使存在改变策略的需要，系统本身也可以保持不变。即使策略模块必须保留在内核中，如果可能，它也应该与机制相隔离，这样策略模块中的变化就不会影响机制模块。

为了使策略与机制之间的划分更加清晰，让我们考虑两个现实世界的例子。第一个例子，考虑一家大型公司，该公司拥有负责向员工发放薪水的工资部门。该部门拥有计算机、软件、空白支票、与银行的契约以及更多的机制，以便准确地发出薪水。然而，策略——确定谁将获得多少薪水——是完全与机制分开的，并且是由管理部门决定的。工资部门只是做他们被吩咐做的事情。

第二个例子，考虑一家饭店。它拥有提供餐饮的机制，包括餐桌、餐具、服务员、充满设备的厨房、与信用卡公司的契约，如此等等。策略是由厨师长设定的，也就是说，厨师长决定菜单上有什么。如果厨师长决定撤掉豆腐换上牛排，那么这一新的策略可以由现有的机制来处理。

现在让我们考虑某些操作系统的例子。首先考虑线程调度。内核可能拥有一个优先级调度器,具有 k 个优先级。机制是一个数组,以优先级为索引,如图10-11或图11-19所示。每个数组项是处于该优先级的就绪线程列表的表头。调度器只是从最高优先级到最低优先级搜索数组,选中它找到的第一个线程。策略是设定优先级。系统可能具有不同的用户类别,每个类别拥有不同的优先级。它还可能允许用户进程设置其线程的相对优先级。优先级可能在完成I/O之后增加,或者在用完时间配额之后降低。还有众多的其他策略可以遵循,但是此处的中心思想是设置策略与执行之间的分离。

第二个例子是分页。机制涉及到MMU管理,维护占用页面与空闲页面的列表,以及用来将页面移入磁盘或者移出磁盘的代码。策略是当页面故障发生时决定做什么,它可能是局部的或全局的,基于LRU的或基于FIFO的,或者是别的东西,但是这一算法可以(并且应该)完全独立于实际管理页面的机制。

第三个例子是允许将模块装载到内核之中。机制关心的是它们如何被插入、如何被链接、它们可以发出什么调用,以及可以对它们发出什么调用。策略是确定允许谁将模块装载到内核之中以及装载哪些模块。也许只有超级用户可以装载模块,也许任何用户都可以装载被适当权威机构数字签名的模块。

13.3.3 正交性

良好的系统设计在于单独的概念可以独立地组合。例如,在C语言中,存在基本的数据类型,包括整数、字符和浮点数,还存在用来组合数据类型的机制,包括数组、结构和联合。这些概念独立地组合,允许拥有整数数组、字符数组、浮点数的结构和联合成员等。实际上,一旦定义了一个新的数据类型,如整数数组,就可以如同一个基本数据类型一样使用它,例如作为一个结构或者一个联合的成员。独立地组合单独的概念的能力称为正交性(orthogonality),它是简单性和完整性原理的直接结果。

正交性概念还以各种各样的伪装出现在操作系统中, Linux的clone系统调用就是一个例子,它创建一个新线程。该调用有一个位图作为参数,它允许单独地共享或复制地址空间、工作目录、文件描述符以及信号。如果复制所有的东西,我们将得到一个进程,就像调用fork一样。如果什么都不复制,则是在当前进程中创建一个新线程。然而,创建共享的中间形式同样也是可以的,而这在传统的UNIX系统中是不可能的。通过分离各种特性并且使它们正交,是可以做到更好地控制自由度的。

正交性的另一个应用是Windows Vista中进程概念与线程概念的分离。进程是一个资源容器,既不多也不少。线程是一个可调度的实体。当把另一个进程的句柄提供给一个进程时,它拥有多少个线程都是没有关系的。当一个线程被调度时,它从属于哪个进程也是没有关系的。这些概念是正交的。

正交性的最后一个例子来自UNIX。在UNIX中,进程的创建分两步完成:fork和exec。创建新的地址空间与用新的内存映像装载该地址空间是分开的,这就为在两者之间做一些事情提供了可能(例如处理文件描述符)。在Windows Vista中,这两个步骤不能分开,也就是说,创建新的地址空间与填充该地址空间的概念不是正交的。Linux的clone加exec序列是更加正交的,因为存在更细粒度的构造块可以利用。作为一般性的规则,拥有少量能够以很多方式组合的正交元素,将形成小巧、简单和精致的系统。

13.3.4 命名

操作系统使用的最长久的数据结构具有某种类型的名字或标识符,通过名字或标识符就可以引用这些数据结构。显而易见的例子有注册名、文件名、设备名、进程ID等。在操作系统的设计与实现中,如何构造和管理这些名字是一个重要的问题。

为人们的使用而设计的名字是ASCII或Unicode形式的字符串,并且通常是层次化的。目录路径,例如usr/ast/books/mos2/chap-12,显然是层次化的,它指出从根目录开始搜索的一个目录序列。URL也是层次化的。例如, www.cs.vu.nl/~ast/表示一个特定国家(nl)的一所特定大学(vu)的一个特定的系(cs)内的一台特定的机器(www)。斜线号后面的部分指出的是目标机器上的一个特定的文件,在这种情形中,按照惯例,该文件是ast主目录中的www/index.html。注意URL(以及一般的DNS地址,包括电子邮件地址)是“反向的”,从树的底部开始并且向上走,这与文件名有所不同,后者从树的顶部开始并且向下走。看待这一问题的另一种方法是从头写这棵树是从左开始向右走,还是从右开始向左走。

命名经常在外部和内部两个层次上实现。例如,文件总是具有字符串名字供人们使用。此外,几乎总是存在一个内部名字由系统使用。在UNIX中,文件的实际名字是它的i节点号,在内部根本就不使用ASCII名字。实际上,它甚至不是惟一的,因为一个文件可能具有多个链接指向它。在Windows Vista中,相仿的

内部名字是MFT中文件的索引。目录的任务是在外部名字和内部名字之间提供映射，如图13-4所示。

在许多情况下（例如上面给出的文件名的例子），内部名字是一个无符号整数，用作进入一个内部表格的索引。表格—索引名字的其他例子还有UNIX中的文件描述符和Windows Vista中的对象句柄。注意这些都没有任何外部表示，它们严格地被系统和运行的进程所使用。一般而言，对于当系统重新启动时就会丢失的暂时的名字，使用表格索引是一个很好的主意。

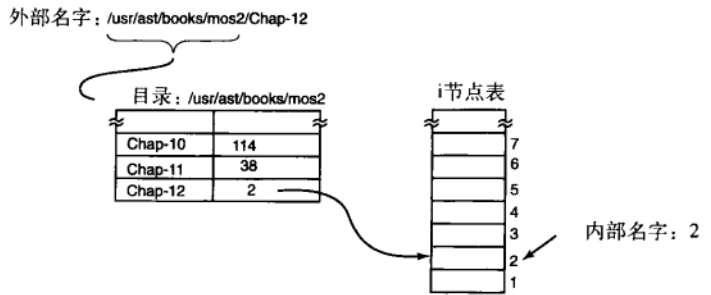


图13-4 目录用来将外部名字映射到内部名字上

操作系统经常支持多个名字空间，既在内部又在外部。例如，在第11章我们了解了Windows Vista支持的三个外部名字空间：文件名、对象名和注册表名（并且还有我们没有考虑的活动目录名）。此外，还存在着使用无符号整数的数不清的内部名字空间，例如对象句柄、MFT项等。尽管外部名字空间中的名字都是Unicode字符串，但是在注册表中查寻一个文件名是不可以的，正如在对象表中使用MFT索引是不可以的。在一个良好的设计中，相当多的考虑花在了需要多少个名字空间，每个名字空间中名字的语法是什么，怎样分辨它们，是否存在抽象的和相对的名字，如此等等。

13.3.5 绑定的时机

正如我们刚刚看到的，操作系统使用多种类型的名字来引用对象。有时在名字和对象之间的映射是固定的，但是有时不是。在后一种情况下，何时将名字与对象绑定可能是很重要的。一般而言，早期绑定（early binding）是简单的，但是不灵活，而晚期绑定（late binding）则比较复杂，但是通常更加灵活。

为了阐明绑定时机的概念，让我们看一看某些现实世界的例子。早期绑定的一个例子是某些高等学校允许父母在婴儿出生时登记入学，并且预付当前的学费。以后当学生长大到18岁时，学费已经全部付清，无论此刻学费有多么高。

在制造业中，预先订购零部件并且维持零部件的库存量是早期绑定。相反，即时制造要求供货商能够立刻提供零部件，不需要事先通知。这就是晚期绑定。

程序设计语言对于变量通常支持多种绑定时机。编译器将全局变量绑定到特殊的虚拟地址，这是早期绑定的例子。过程的局部变量在过程被调用的时刻（在栈中）分配一个虚拟地址，这是中间绑定。存放在堆中的变量（这些变量由C中的malloc或Java中的new分配）仅仅在它们实际被使用的时候才分配虚拟地址，这便是晚期绑定。

操作系统对大多数数据结构通常使用早期绑定，但是偶尔为了灵活性也使用晚期绑定。内存分配是一个相关的案例。在缺乏地址重定位硬件的机器上，早期的多道程序设计系统不得不在某个内存地址装载一个程序，并且对其重定位以便在此处运行。如果它曾经被交换出去，那么它就必须装回到相同的内存地址，否则就会出错。相反，页式虚拟内存是晚期绑定的一种形式。在页面被访问并且实际装入内存之前，与一个给定的虚拟地址相对应的实际物理地址是不知道的。

晚期绑定的另一个例子是GUI中窗口的放置。在早期图形系统中，程序员必须为屏幕上的所有图像设定绝对屏幕坐标，与此相对照，在现代GUI中，软件使用相对于窗口原点的坐标，但是在窗口被放置在屏幕上之前该坐标是不确定的，并且以后，它甚至是可能改变的。

13.3.6 静态与动态结构

操作系统设计人员经常被迫在静态与动态数据结构之间进行选择。静态结构总是简单易懂，更加容易编程并且用起来更快；动态结构则更加灵活。一个显而易见的例子是进程表。早期的系统只是分配一个固定的数组，存放每个进程结构。如果进程表由256项组成，那么在任意时刻只能存在256个进程。试图创建第257个进程将会失败，因为缺乏表空间。类似的考虑对于打开的文件表（每个用户的和系统范

围的)以及许多其他内核表格也是有效的。

一个替代的策略是将进程表建立为一个小型表的链表,最初只有一个表。如果该表被填满,可以从全局存储池中分配另一个表并且将其链接到前一个表。这样,在全部内核内存被耗尽之前,进程表不可能被填满。

另一方面,搜索表格的代码会变得更加复杂。例如,在图13-5中给出了搜索一个静态进程表以查找给定PID, pid的代码。该代码简单有效。对于小型表的链表,做同样的搜索则需要更多的工作。

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

图13-5 对于给定PID搜索进程表的代码

当存在大量的内存或者当表的利用可以猜测得相当准确时,静态表是最佳的。例如,在一个单用户系统中,用户不太可能立刻启动64个以上的进程,并且如果试图启动第65个进程失败了,也并不是一个彻底的灾难。

还有另一种选择是使用一个固定大小的表,但是如果该表填满了,就分配一个新的固定大小的表,比方说大小是原来的两倍。然后将当前的表项复制到新表中并且把旧表返回空闲存储池。这样,表总是连续的而不是链接的。此处的缺点是需要某些存储管理,并且现在表的地址是变量而不是常量。

对于内核栈也存在类似的问题。当一个线程切换到内核模式,或者当一个内核模式线程运行时,它在内核空间中需要一个栈。对于用户线程,栈可以初始化成从虚拟地址空间的顶部向下生长,所以大小不需要预先设定。对于内核线程,大小必须预先设定,因为栈占据了某些内核虚拟地址空间并且可能存在于许多栈。问题是:每个栈应该得到多少空间?此处的权衡与进程表是类似的。

另一个静态-动态权衡是进程调度。在某些系统中,特别是在实时系统中,调度可以预先静态地完成。例如,航空公司在班机启航前几周就知道它的飞机什么时候要出发。类似地,多媒体系统预先知道何时调度音频、视频和其他进程。对于通用的应用,这些考虑是不成立的,并且调度必须是动态的。

还有一个静态-动态问题是内核结构。如果内核作为单一的二进制程序建立并且装载到内存中运行,情况是比较简单的。然而,这一设计的结果是添加一个新的I/O设备就需要将内核与新的设备驱动程序重新链接。UNIX的早期版本就是以这种方式工作的,在小型计算机环境中它相当令人满意,那时添加新的I/O设备是十分罕见的事情。如今,大多数操作系统允许将代码动态地添加到内核之中,随之而来的则是所有额外的复杂性。

13.3.7 自顶向下与自底向上的实现

虽然最好是自顶向下地设计系统,但是在理论上系统可以自顶向下或者自底向上地实现。在自顶向下的实现中,实现者以系统调用处理程序为开端,并且探究需要什么机制和数据结构来支持它们。接着编写这些过程等,直到触及硬件。

这种方法的问题是,由于只有顶层过程可用,任何事情都难于测试。出于这样的原因,许多开发人员发现实际上自底向上地构建系统更加可行。这一方法需要首先编写隐藏底层硬件的代码,特别是图11-6中的HAL。中断处理程序和时钟驱动程序也是早期就需要的。

然后,可以使用一个简单的调度器(例如轮转调度)来解决多道程序设计问题。在这一时刻,测试系统以了解它是否能够正确地运行多个进程应该是可能的。如果运转正常,此时可以开始仔细地定义贯穿系统的各种各样的表格和数据结构,特别是那些用于进程和线程管理以及后面内存管理的表格与数据结构。I/O和文件系统在最初可以等一等,用于测试和调试目的的读键盘与写屏幕的基本方法除外。在某些情况下,关键的低层数据结构应该得到保护,这可以通过只允许经由特定的访问过程来访问而实现——实际上这是面向对象的程序设计思想,不论采用何种程序设计语言。当较低的层次完成时,可以彻底地测试它们。这样,系统自底向上推进,很像是建筑商建造高层办公楼的方式。

如果有一个大型团队可用,那么替代的方法是首先做出整个系统的详细设计,然后分配不同的小组编写不同的模块。每个小组独立地测试自己的工作。当所有的部分都准备好时,可以将它们集成起来并

加以测试。这一设计方式存在的问题是，如果最初没有什么可以运转，可能难于分离出一个或多个模块是否工作不正常，或者一个小组是否误解了某些其他模块应该做的事情。尽管如此，如果有大型团队，还是经常使用该方法使程序设计工作中的并行程度最大化。

13.3.8 实用技术

我们刚刚了解了系统设计与实现的某些抽象思想，现在将针对系统实现考察一些有用的具体技术。这方面的技术很多，但是篇幅的限制使我们只能介绍其中的少数技术。

1. 隐藏硬件

许多硬件是十分麻烦的，所以只好尽早将其隐藏起来（除非它要展现能力，而大多数硬件不会这样）。某些非常低层的细节可以通过如图13-2所示的HAL类型的层次得到隐藏。然而，许多硬件细节不能以这样的方式来隐藏。

值得尽早关注的一件事情是如何处理中断。中断使得程序设计令人不愉快，但是操作系统必须对它们进行处理。一种方法是立刻将中断转变成别的东西，例如，每个中断都可以转变成即时弹出的线程。在这一时刻，我们处理的是线程，而不是中断。

第二种方法是将每个中断转换成在一个互斥量上的unlock操作，该互斥量对应正在等待的驱动程序。于是，中断的惟一效果就是导致某个线程变为就绪。

第三种方法是将一个中断转换成发送给某个线程的消息。低层代码只是构造一个表明中断来自何处的消息，将其排入队列，并且调用调度器以（潜在地）运行处理程序，而处理程序可能正在阻塞等待该消息。所有这些技术，以及其他类似的技术，都试图将中断转换成线程同步操作。让每个中断由一个适当的线程在适当的上下文中处理，比起在中断碰巧发生的随意上下文中运行处理程序，前者要更加容易管理。当然，这必须高效率地进行，而在操作系统内部深处，一切都必须高效率地进行。

大多数操作系统被设计成运行在多个硬件平台上。这些平台可以按照CPU芯片、MMU、字长、RAM大小以及不能容易地由HAL或等价物屏蔽的其他特性来区分。尽管如此，人们高度期望拥有单一的一组源文件用来生成所有的版本，否则，后来发现的每个程序错误必须在多个源文件中修改多次，从而有源文件逐渐疏远的危险。

某些硬件的差异，例如RAM大小，可以通过让操作系统在引导的时候确定其取值并且保存在一个变量中来处理。内存分配器可以利用RAM大小变量来确定构造多大的数据块高速缓存、页表等。甚至静态的表格，如进程表，也可以基于总的可用内存来确定大小。

然而，其他的差异，例如不同的CPU芯片，就不能让单一的二进制代码在运行的时候确定它正在哪一个CPU上运行。解决一个源代码多个目标机的问题的一种方法是使用条件编译。在源文件中，定义了一定的编译时标志用于不同的配置，并且这些标志用来将独立于CPU、字长、MMU等的代码用括号括起。例如，设想一个操作系统运行在Pentium和UltraSPARC芯片上，这就需要不同的初始化代码。可以像图13-6a中那样编写init过程的代码。根据CPU的取值（该值定义在头文件config.h中），实现一种初始化或其他的初始化过程。由于实际的二进制代码只包含目标机所需要的代码，这样就不会损失效率。

<pre>#include "config.h" init() { #if (CPU == PENTIUM) /*此处是Pentium的初始化*/ #endif #if (CPU == ULTRASPARC) /*此处是UltraSPARC的初始化*/ #endif }</pre>	<pre>#include "config.h" #if (WORD_LENGTH == 32) typedef int Register; #endif #if (WORD_LENGTH == 64) typedef long Register; #endif Register R0, R1, R2, R3;</pre>
a)	b)

图13-6 a) 依赖CPU的条件编译；b) 依赖字长的条件编译

第二个例子，假设需要一个数据类型Register，它在Pentium上是32位，在UltraSPARC上是64位。这可以由图13-6b中的条件代码来处理（假设编译器产生32位的int和64位的long）。一旦做出这样的定义（可能是在别的什么地方的头文件中），程序员就可以只需声明变量为Register类型并且确信它们将具有正确的长度。

当然，头文件config.h必须正确地定义。对于Pentium处理器，它大概是这样的：

```
#define CPU_PENTIUM
#define WORD_LENGTH 32
```

为了编译针对UltraSPARC的系统，应该使用不同的config.h，其中具有针对UltraSPARC的正确取值，它或许是这样的：

```
#define CPU_ULTRASPARC
#define WORD_LENGTH 64
```

一些读者可能奇怪为什么CPU和WORD_LENGTH用不同的宏来处理。我们可以很容易地用针对CPU的测试而将Register的定义用括号括起，对于Pentium将其设置为32位，对于UltraSPARC将其设置为64位。然而，这并不是一个好主意。考虑一下以后当我们将系统移植到64位Intel Itanium处理器时会发生什么事情。我们可能不得不为了Itanium而在图13-6b中添加第三个条件。通过像上面那样定义宏，我们要做的全部事情是在config.h文件中为Itanium处理器包含如下的代码行：

```
#define WORD_LENGTH 64
```

这个例子例证了前面讨论过的正交性原则。那些依赖CPU的细节应该基于CPU宏而条件编译，而那些依赖字长的细节则应该使用WORD_LENGTH宏。类似的考虑对于许多其他参数也是适用的。

2. 间接

人们不时地说在计算机科学中没有什么问题不能通过另一个层次间接得到解决。虽然有些夸大其词，但是其中的确存在一定程度的真实性。让我们考虑一些例子。在基于Pentium的系统上，当一个键被按下时，硬件将生成一个中断并且将键的编号而不是ASCII字符编码送到一个设备寄存器中。此外，当此键后来被释放时，第二个中断生成，同样伴随一个键编号。间接为操作系统使用键编号作为索引检索一张表格以获取ASCII字符提供了可能，这使得处理世界上不同国家使用的许多键盘十分容易。获得按下与释放两个信息使得将任何键作为换挡键成为可能，因为操作系统知道键按下与释放的准确序列。

间接还被用在输出上。程序可以写ASCII字符到屏幕上，但是这些字符被解释为针对当前输出字体的一张表格的索引。表项包含字符的位图。这一间接使得将字符与字体相分离成为可能。

间接的另一个例子是UNIX中主设备号的使用。在内核内部，有一张表格以块设备的主设备号作为索引，还有另一张表格用于字符设备。当一个进程打开一个特定的文件（例如/dev/hd0）时，系统从i节点提取出类型（块设备或字符设备）和主副设备号，并且检索适当的驱动程序表以找到驱动程序。这一间接使得重新配置系统十分容易，因为程序涉及的是符号化的设备名，而不是实际的驱动程序名。

还有另一个间接的例子出现在消息传递的系统中，该系统命名一个邮箱而不是一个进程作为消息的目的地。通过间接使用邮箱（而不是指定一个进程作为目的地），能够获得相当可观的灵活性（例如，让一位秘书处理她的老板的消息）。

在某种意义上，使用诸如

```
#define PROC_TABLE_SIZE 256
```

的宏也是间接的一种形式，因为程序员无须知道表格实际有多大就可以编写代码。一个好的习惯是为所有的常量提供符号化的名字（有时-1、0和1除外），并且将它们放在头文件中，同时提供注释解释它们代表什么。

3. 可重用性

在略微不同的上下文中重用相同的代码通常是可行的。这样做是一个很好的想法，因为它减少了二进制代码的大小并且意味着代码只需要调试一次。例如，假设用位图来跟踪磁盘上的空闲块。磁盘块管理可以通过提供管理位图的过程alloc和free得到处理。

在最低限度上,这些过程应该对任何磁盘起作用。但是我们可以比这更进一步。相同的过程还可以用于管理内存块、文件系统块高速缓存中的块,以及i节点。事实上,它们可以用来分配与回收能够线性编号的任意资源。

4. 重入

重入指的是代码同时被执行两次或多次的能力。在多处理器系统上,总是存在着这样的危险:当一个CPU执行某个过程时,另一个CPU在第一个完成之前也开始执行它。在这种情况下,不同CPU上的两个(或多个)线程可能在同时执行相同的代码。这种情况必须通过使用互斥量或者某些其他保护临界区的方法进行处理。

然而,在单处理器上,问题也是存在的。特别地,大多数操作系统是在允许中断的情况下运行的。否则,将丢失许多中断并且使系统不可靠。当操作系统忙于执行某个过程 P 时,完全有可能发生一个中断并且中断处理程序也调用 P 。如果 P 的数据结构在中断发生的时刻处于不一致的状态,中断处理程序就会注意到它们处于不一致的状态并且失败。

可能发生这种情况的一个显而易见的例子是 P 是调度器。假设某个进程用完了它的时间配额,并且操作系统正将其移动到其队列的末尾。在列表处理的半路,中断发生了,使得某个进程就绪,并且运行调度器。由于队列处于不一致的状态,系统有可能会崩溃。因此,即使在单处理器上,最好是操作系统的大部分为可重入的,关键的数据结构用互斥量来保护,并且在中断不被允许的時刻禁用中断。

5. 蛮力法

使用蛮力法解决问题多年以来获得了较差的名声,但是依据简单性它经常是行之有效的方法。每个操作系统都有许多很少会调用的过程或是具有很少数据的操作,不值得对它们进行优化。例如,在系统内部经常有必要搜索各种表格和数组。蛮力算法只是让表格保持表项建立时的顺序,并且当必须查找某个东西时线性地搜索表格。如果表项的数目很少(例如少于1000个),对表格排序或建立散列表的好处不大,但是代码却复杂得多并且很有可能在其中存在错误。

当然,对处于关键路径上的功能,例如上下文切换,使它们加快速度的一切措施都应该尽力去做,即使可能要用汇编语言编写它们。但是,系统的大部分并不处于关键路径上。例如,许多系统调用很少被调用。如果每隔1秒有一个fork调用,并且该调用花费1毫秒完成,那么即便将其优化到花费0秒也不过仅有0.1%的获益。如果优化过的代码更加庞大且有更多错误,那就不必多此一举了。

6. 首先检查错误

由于各种各样的原因,许多系统调用可能潜在地会失败:要打开的文件属于他人,因为进程表满而创建进程失败;或者因为目标进程不存在而使信号不能被发送。操作系统在执行调用之前必须无微不至地检查每一个可能的错误。

许多系统调用还需要获得资源,例如进程表的空位、i节点表的空位或文件描述符。一般性的建议是在获得资源之前,首先进行检查以了解系统调用能否实际执行,这样可以省去许多麻烦。这意味着,将所有的测试放在执行系统调用的过程的开始。每个测试应该具有如下的形式:

```
if (error_condition) return(ERROR_CODE);
```

如果调用通过了所有严格的测试,那么就可以肯定它将会取得成功。在这一时刻它才能获得资源。

如果将获得资源的测试分散开,那么就意味着如果在这一过程中某个测试失败,到这一时刻已经获得的所有资源都必须归还。如果在这里发生了一个错误并且资源没有被归还,可能并不会立刻发生破坏。例如,一个进程表项可能只是变得永久地不可用。然而,随着时间的流逝,这一差错可能会触发多次。最终,大多数或全部进程表项可能都会变得不可用,导致系统以一种极度不可预料且难以调试的方式崩溃。

许多系统以内存泄漏的形式遭受了这一问题的侵害。典型地,程序调用malloc分配了空间,但是以后忘记了调用free释放它。逐渐地,所有的内存都消失了,直到系统重新启动。

Engler等人(2000)推荐了一种有趣的方法在编译时检查某些这样的错误。他们注意到程序员知道许多定式而编译器并不知道,例如当你锁定一个互斥量的时候,所有在锁定操作处开始的路径都必须包含一个解除锁定的操作并且在相同的互斥量上没有更多的锁定。他们设计了一种方法让程序员将这一事实告诉编译器,并且指示编译器在编译时检查所有路径以发现对定式的违犯。程序员还可以设定已分配

的内存必须在所有路径上释放，以及设定许多其他的条件。

13.4 性能

所有事情都是平等的，一个快速的操作系统比一个慢速的操作系统好。然而，一个快速而不可靠的操作系统还不如一个慢速但可靠的操作系统。由于复杂的优化经常会导致程序错误，有节制地使用它们是很重要的。尽管如此，在性能是至关重要的地方进行优化还是值得的。在下面几节我们将看一些一般的技术，这些技术在特定的地方可以用来改进性能。

13.4.1 操作系统为什么运行缓慢

在讨论优化技术之前，值得指出的是许多操作系统运行缓慢在很大程度上是操作系统自身造成的。例如，古老的操作系统，如MS-DOS和UNIX版本7在几秒钟内就可以启动。现代UNIX系统和Windows Vista尽管运行在快1000倍的硬件上，可能要花费几分钟才能启动。原因是它们要做更多的事情，有用的或无用的。看一个相关的案例。即插即用使得安装一个新的硬件设备相当容易，但是付出的代价是在每次启动时，操作系统都必须检查所有的硬件以了解是否存在新的设备。这一总线扫描是要花时间的。

一种替代的（并且依作者看来是更好的）方法是完全抛弃即插即用，并且在屏幕上包含一个图标标明“安装新硬件”。当安装一个新的硬件设备时，用户可以点击图标开始总线扫描，而不是在每次启动的时候做这件事情。当然，当今的系统设计人员是完全知道这一选择的。但是他们拒绝这一选择，主要是因为他们假设用户太过愚笨而不能正确地做这件事情（尽管他们使用了更加友好的措辞）。这只是一个例子，但是还存在更多的事例，期望让系统“用户友好”（或者“防傻瓜”，取决于你的看法）却使系统始终对所有用户是缓慢的。

或许系统设计人员为改进性能可以做的最大的一件事情，是对于添加新的功能特性更加具有选择性。要问的问题不是“用户会喜欢吗？”而是“这一功能特性按照代码大小、速度、复杂性和可靠性值不计代价吗？”只有当优点明显地超过缺点的时候，它才应该被包括。程序员倾向于假设代码大小和程序错误计数为0并且速度为无穷大。经验表明这种观点有些过于乐观。

另一个重要因素是产品的市场销售。到某件产品的第4或第5版上市的时候，真正有用的所有功能特性或许已经全部包括了，并且需要该产品的大多数人已经拥有它了。为了保持销售，许多生产商仍然继续生产新的版本，具有更多的功能特性，正是这样才可以向现有的顾客出售升级版。只是为了添加新的功能特性而添加新的功能特性可能有助于销售，但是很少会有助于性能。

13.4.2 什么应该优化

作为一般的规则，系统的第一版应该尽可能简单明了。惟一的优化应该是那些显而易见要成为不可避免的问题的事情。为文件系统提供块高速缓存就是这样的一个例子。一旦系统引导起来并运行，就应该仔细地测量以了解时间真正花在了什么地方。基于这些数字，应该在最有帮助的地方做出优化。

这里有一个关于优化不但不好反而更坏的真实故事。作者的一名学生编写了MINIX的mkfs程序。该程序在一个新格式化的磁盘上布下一个新的文件系统。这名学生花了大约6个月的时间对其进行优化，包括放入磁盘高速缓存。当他上交该程序时，它不能工作，需要另外几个月进行调试。在计算机的生命周期中，当系统安装时，该程序典型地在硬盘上运行一次。它还对每块做格式化的软盘运行一次。每次运行大约耗时2秒。即使未优化的版本耗时1分钟，花费如此多的时间优化一个很少使用的程序也是相当不值的。

对于性能优化，一条相当适用的口号是：

足够好就够好了。

通过这条口号我们要表达的意思是：性能一旦达到一个合理的水平，榨出最后一点百分比的努力和复杂性或许并不值得。如果调度算法相当公平并且在90%的时间保持CPU忙碌，它就尽到了自己的职责。发明一个改进了5%但是要复杂得多的算法或许是一个坏主意。类似地，如果缺页率足够低到不是瓶颈，克服重重难关以获得优化的性能通常并不值得。避免灾难比获得优化的性能要重要得多，特别是针对一种负载的优化对于另一种负载可能并非优化的情况。

13.4.3 空间-时间的权衡

改进性能的一种一般性的方法是权衡时间与空间。在一个使用很少内存但是速度比较慢的算法与一个使用很多内存但是速度更快的算法之间进行选择，这在计算机科学中是经常发生的事情。在做出重要的优化时，值得寻找通过使用更多内存加快了速度的算法，或者反过来通过做更多的计算节省了宝贵的内存的算法。

一种常用而有益的技术是用宏来代替小的过程。使用宏消除了通常与过程调用相关联的开销。如果调用出现在一个循环的内部，这种获益尤其显著。例如，假设我们使用位图来跟踪资源，并且经常需要了解在位图的某一部分中有多少个单元是空闲的。为此，我们需要一个过程bit_count来计数一个字节中值为1的位的个数。图13-7a中给出了简单明了的过程。它对一个字节中的各个位循环，每次计数它们一次。

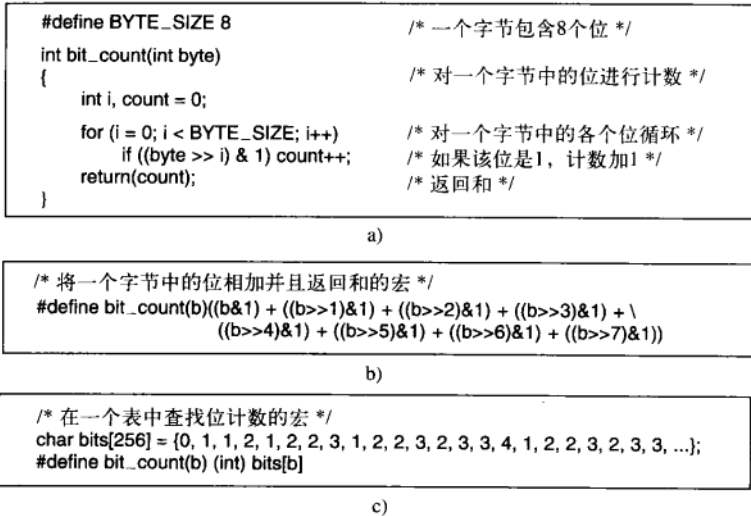


图13-7 a) 对一个字节中的位进行计数的过程；b) 对位进行计数的宏；c) 在表中查找位计数

该过程有两个低效的根源。首先，它必须被调用，必须为它分配栈空间，并且必须返回。每个过程调用都有这个开销。第二，它包含一个循环，并且总是存在与循环相关联的某些开销。

一种完全不同的方法是使用图13-7b中的宏。这个宏是一个内联表达式，它通过对参数连续地移位，屏蔽除低位以外的其他位，并且将8个项相加，这样来计算位的和。这个宏决不是一件艺术作品，但是它只在代码中出现一次。当这个宏被调用时，例如通过

```
sum = bit_count(table[i]);
```

这个宏调用看起来与过程调用等同。因此，除了定义有一点凌乱以外，宏中的代码看上去并不比过程中的代码要差，但是它的效率更高，因为它消除了过程调用的开销和循环的开销。

我们可以更进一步研究这个例子。究竟为什么计算位计数？为什么不在一个表中查找？毕竟只有256个不同的字节，每个字节具有0到8之间的惟一的值。我们可以声明一个256项的表bits，每一项（在编译时）初始化成对应于该字节值的位计数。采用这一方法在运行时根本就不需要计算，只要一个变址操作就可以了。图13-7c中给出了做这一工作的宏。

这是用内存换取计算时间的明显的例子。然而，我们还可以再进一步。如果需要整个32位字的位计数，使用我们的bit_count宏，每个字我们需要执行四次查找。如果将表扩展到65 536项，每个字查找两次就足够了，代价是更大的表。

在表中查找答案可以用在其他方面。例如，在第7章中，我们看到了JPEG图像压缩是怎样工作的，它使用了相当复杂的离散余弦变换。另一种压缩技术GIF使用表查找来编码24位RGB图像。然而，GIF只对具有256种颜色或更少颜色的图像起作用。对于每幅要压缩的图像，构造一个256项的调色板，每一项包含一个24位的RGB值。压缩过的图像于是包含每个像素的8位索引，而不是24位颜色值，增益因子

为3。图13-8中针对一幅图像的一个4×4区域说明了这一思想。原始未压缩的图像如图13-8a所示，该图中每个取值是一个24位的值，每8位给出红、绿和蓝的强度。GIF图像如图13-8b所示，该图中每个取值是一个进入调色板的8位索引。调色板作为图像文件的一部分存放，如图13-8c所示。实际上，GIF算法的内容比这要多，但是思想的核心是表查找。

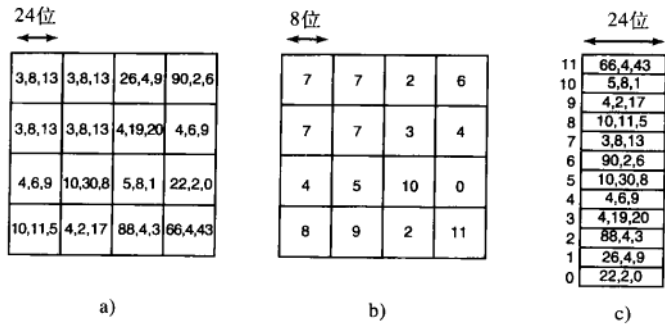


图13-8 a) 每个像素24位的未压缩图像的局部；b) 以GIF压缩的相同局部，每个像素8位；c) 调色板

存在减少图像大小的另一种方法，并且这种方法说明了一种不同的权衡。PostScript是一种程序设计语言，可以用来描述图像。（实际上，任何程序设计语言都可以描述图像，但是PostScript专为这一目的进行了调节。）许多打印机具有内嵌的PostScript解释器，能够运行发送给它们的PostScript程序。

例如，如果在一幅图像中存在一个像素矩形块具有相同的颜色，用于该图像的PostScript程序将携带指令，用来将一个矩形放置在一定的位置并且用一定的颜色填充该矩形。只需要少数几个位就可以发出此命令。当打印机接收图像时，打印机中的解释器必须运行程序才能绘制出图像。因此，PostScript以更多的计算为代价实现了数据压缩，这是与表查找不同的一种权衡，但是当内存或带宽不足时是颇有价值的。

其他的权衡经常牵涉数据结构。双向链表比单向链表占据更多的内存，但是经常使得访问表项速度更快。散列表甚至更浪费空间，但是要更快。简而言之，当优化一段代码时要考虑的重要事情之一是：使用不同的数据结构是否将产生最佳的时间-空间平衡。

13.4.4 高速缓存

用于改进性能的一种众所周知的技术是高速缓存。在任何相同的结果可能需要多次的情况下，高速缓存都是适用的。一般的方法是首先做完整的工作，然后将结果保存在高速缓存中。对于后来的尝试，首先要检查高速缓存。如果结果在高速缓存中，就使用它。否则，再做完整的工作。

我们已经看到高速缓存在文件系统内部的运用，在高速缓存中保存一定数目最近用过的磁盘块，这样在每次命中时就可以省略磁盘读操作。然而，高速缓存还可以用于许多其他目的。例如，解析路径名就代价高昂得令人吃惊。再次考虑图4-35中UNIX的例子。为了查找/usr/ast/mbox，需要如下的磁盘访问：

- 1) 读入根目录的i节点（i节点1）。
- 2) 读入根目录（磁盘块1）。
- 3) 读入/usr的i节点（i节点6）。
- 4) 读入/usr目录（磁盘块132）。
- 5) 读入/usr/ast的i节点（i节点26）。
- 6) 读入/usr/ast目录（磁盘块406）。

只是为了获得文件的i节点号就需要6次磁盘访问。然后必须读入i节点本身以获得磁盘块号。如果文件小于块的大小（例如1024字节），那么需要8次磁盘访问才读到数据。

某些系统通过对（路径，i节点）的组合进行高速缓存来优化路径名的解析。对于图4-35的例子，在解析/usr/ast/mbox之后，高速缓存中肯定会保存图13-9的前三项。最后三项来自解析其他路径。

路 径	i节点号
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

图13-9 图4-35的i节点高速缓存的局部

当必须查找一个路径时，名字解析器首先查阅高速缓存并搜索它以找到高速缓存中存在的最长的子字符串。例如，如果存在路径/usr/ast/grants/stw，高速缓存会返回/usr/ast/是i节点26这样的事实，这样搜索就可以从这里开始，消除了四次磁盘访问。

对路径进行高速缓存存在的一个问题是，文件名与i节点号之间的映射并不总是固定的。假设文件/usr/ast/mbox从系统中被删除，并且其i节点重用于不同用户所拥有的不同的文件。随后，文件/usr/ast/mbox再次被创建，并且这一次它得到i节点106。如果不对这件事情进行预防，高速缓存项现在将是错误的，并且后来的查找将返回错误的i节点号。为此，当一个文件或目录被删除时，它的高速缓存项以及（如果它是一个目录的话）它下面所有的项都必须从高速缓存中清除。

磁盘块与路径名并不是能够高速缓存的惟一项目，i节点也可以被高速缓存。如果弹出的线程用来处理中断，每个这样的线程需要一个栈和某些附加的机构。这些以前用过的线程也可以被高速缓存，因为刷新一个用过的线程比从头创建一个新的线程更加容易（为了避免必须分配内存）。难于生产的任何事物几乎都能够被高速缓存。

13.4.5 线索

高速缓存项总是正确的。高速缓存搜索可能失败，但是如果找到了一项，那么这一项保证是正确的并且无需再费周折就可以使用。在某些系统中，包含线索（hint）的表是十分便利的。这些线索是关于答案的暗示，但是它们并不保证是正确的。调用者必须自行对结果进行验证。

众所周知的关于线索的例子是嵌在Web页上的URL。点击一个链接并不能保证被指向的Web页就在那里。事实上，被指向的网页可能10年前就被删除了。因此包含URL的网页上面的信息只是一个线索。

线索还用于连接远程文件。信息是提示有关远程文件某些事项的线索，例如文件存放的位置。然而，自该线索被记录以来，文件可能已经被移动或者被删除了，所以为了明确线索是否正确，总是需要进行检查。

13.4.6 利用局部性

进程和程序的行为并不是随机的，它们在时间上和空间上展现出相当程度的局部性，并且可以以各种方式利用该信息来改进性能。空间局部性的一个常见例子是这样的事实：进程并不是在其地址空间内部随机地到处跳转的。在一个给定的时间间隔内，它们倾向于使用数目比较少的页面。进程正在有效地使用的页面可以被标记为它的工作集，并且操作系统能够确保当进程被允许运行时，它的工作集在内存中，这样就减少了缺页的次数。

局部化原理对于文件也是成立的。当一个进程选择了一个特定的工作目录时，很可能将来许多文件引用将指向该目录中的文件。通过在磁盘上将每个目录的所有i节点和文件就近放在一起，可能会获得性能的改善。这一原理正是Berkeley快速文件系统的基础（McKusick等人，1984）。

局部性起作用的另一个领域是多处理器系统中的线程调度。正如我们在第8章中看到的，在多处理器上一种调度线程的方法是试图在最后一次用过的CPU上运行每个线程，期望它的某些内存块依然还在内存的高速缓存中。

13.4.7 优化常见的情况

区分最常见的情况和最坏可能的情况并且分别处理它们，这通常是一个好主意。针对这两者的代码常常是相当不同的。重要的是要使常见的情况速度快。对于最坏的情况，如果它很少发生，使其正确就足够了。

第一个例子，考虑进入一个临界区。在大多数时间中，进入将是成功的，特别是如果进程在临界区内部不花费很多时间的话。Windows Vista提供的一个Win32 API调用EnterCriticalSection就利用了这一期望，它自动地在用户态测试一个标志（使用TSL或等价物）。如果测试成功，进程只是进入临界区并且不需要内核调用。如果测试失败，库过程将调用一个信号量上的down操作以阻塞进程。因此，在通常情况下是不需要内核调用的。

第二个例子，考虑设置一个警报（在UNIX中使用信号）。如果当前没有警报待完成，那么构造一个警报并且将其放在定时器队列上是很简单的。然而，如果已经有一个警报待完成，那么就必須找到它并且从定时器队列中删除。由于alarm调用并未指明是否已经设置了一个警报，所以系统必须假设最坏的

情况,即有一个警报。然而,由于大多数时间不存在警报待完成,并且由于删除一个现有的警报代价高昂,所以区分这两种情况是一个好主意。

做这件事情的一种方法是在进程表中保留一个位,表明是否有一个警报待完成。如果这一位为0,就好办了(只是添加一个新的定时器队列项而无须检查)。如果该位为1,则必须检查定时器队列。

13.5 项目管理

程序员是天生的乐观主义者。他们中的大多数认为编写程序的方式就是急切地奔向键盘并且开始击键,不久以后完全调试好的程序就完成了。对于非常大型的程序,事实并非如此。在下面几节,关于管理大型软件项目,特别是大型操作系统项目,我们有一些看法要陈述。

13.5.1 人月神话

经典著作《人月神话》的作者Fred Brooks是OS/360的设计者之一,他后来转向了学术界。在这部经典著作中,Fred Brooks讨论了建造大型操作系统为什么如此艰难的问题(Brooks, 1975, 1995)。当大多数程序员看到他声称程序员在大型项目中每年只能产出1000行调试好的代码时,他们怀疑Brooks教授是否生活在外层空间,或许是在臭虫星(Planet Bug——此处Bug为双关语)上。毕竟,他们中的大多数在熬夜的时候一个晚上就可以产出1000行程序。这怎么可能是任何一个IQ大于50的人一年的产出呢?

Brooks指出的是,具有几百名程序员的大型项目完全不同于小型项目,并且从小型项目获得的结果并不能放大到大型项目。在一个大型项目中,甚至在编码开始之前,大量的时间就消耗在规划如何将工作划分成模块、仔细地说明模块及其接口,以及试图设想模块将怎样互相作用这样的事情上。然后,模块必须独立地编码和调试。最后,模块必须集成起来并且必须将系统作为一个整体来测试。通常的情况是,每个模块单独测试时工作得十分完美,但是当所有部分集成在一起时,系统立刻崩溃。Brooks将工作量估计如下:

- 1/3规划
- 1/6编码
- 1/4模块测试
- 1/4系统测试

换言之,编写代码是容易的部分,困难的部分是断定应该有哪些模块并且使模块A与模块B正确地交互。在由一名程序员编写的小型程序中,留待处理的所有部分都是简单的部分。

Brooks的书的标题来自他的断言,即人与时间是不可互换的。不存在“人月”这样的单位。如果一个项目需要15个人花2年时间构建,很难想像360个人能够在1个月内构建它,甚至让60个人在6个月内做出它或许也是不可能的。

产生这一效应有三个原因。第一,工作不可能完全并行化。直到完成规划并且确定了需要哪些模块以及它们的接口,甚至都不能开始编码。对于一个2年的项目,仅仅规划可能就要花费8个月。

第二,为了完全利用数目众多的程序员,工作必须划分成数目众多的模块,这样每个人才能有事情做。由于每个模块可能潜在地与每个其他模块相互作用,需要将模块-模块相互作用的数目看成随着模块数目的平方而增长,也就是说,随着程序员数目的平方而增长。这一复杂性很快就会失去控制。对于大型项目而言,人与月之间的权衡远不是线性的,对63个软件项目精细的测量证实了这一点(Boehm, 1981)。

第三,调试工作是高度序列化的。对于一个问题,安排10名调试人员并不会加快10倍发现程序错误。事实上,10名调试人员或许比一名调试人员还要慢,因为他们在相互沟通上要浪费太多的时间。

对于人员与时间的权衡,Brooks将他的经验总结在Brooks定律中:

对于一个延期的软件项目,增加人力将使它更加延期。

增加人员的问题在于他们必须在项目中获得培训,模块必须重新划分以便与现在可用的更多数目的程序员相匹配,需要开许多会议来协调各方面的努力等。Abdel-Hamid和Madnick (1991)用实验方法证实了这一定律。用稍稍不敬的方法重述Brooks定律就是:

无论分配多少妇女从事这一工作,生一个孩子都需要9个月。

13.5.2 团队结构

商业操作系统是大型的软件项目,总是需要大型的人员团队。人员的质量极为重要。几十年来人们

已经众所周知的是，顶尖的程序员比拙劣的程序员生产率要高出10倍（Sackman等人，1968）。麻烦在于，当你需要200名程序员时，找到200名顶尖的程序员非常困难，对于程序员的质量你不得不有所将就。

在任何大型的设计项目（软件或其他）中，同样重要的是需要体系结构的一致性。应该有一名才智超群的人对设计进行控制。Brooks引证兰斯大教堂[⊖]作为大型项目的例子，兰斯大教堂的建造花费了几十年的时间，在这一过程中，后来的建筑师完全服从于完成最初风格的建筑师的规划。结果是其他欧洲大教堂无可比拟的建筑结构的一致性。

在20世纪70年代，Harlan Mills把“一些程序员比其他程序员要好很多”的观察结果与对体系结构一致性的需要相结合，提出了首席程序员团队（chief programmer team）的范式（Baker, 1972）。他的思想是要像一个外科手术团队，而不是像一个杀猪屠夫团队那样组织一个程序员团队。不是每个人像疯子一样乱砍一气，而是由一个人掌握着手术刀，其他人在那里提供支持。对于一个10名人员的项目，Mills建议的团队结构如图13-10所示。

头 衔	职 责
首席程序员	执行体系结构设计并编写代码
副手	辅助首席程序员并为其提供咨询
行政主管	管理人员、预算、空间、设备、报告等
编辑	编辑文档，而文档必须由首席程序员编写
秘书	行政主管和编辑各需要一名秘书
程序文书	维护代码和文档档案
工具师	提供首席程序员需要的任何工具
测试员	测试首席程序员的代码
语言律师	兼职人员，他可以就语言向首席程序员提供建议

图13-10 Mills建议的10人首席程序员团队的分工

自从提出这一建议并付诸实施，30年过去了。一些事情已经变化（例如需要一个语言层——C比PL/I更为简单），但是只需要一名才智超群的人员对设计进行控制仍然是正确的。并且这名才智超群者在设计和编程上应该能够100%地起作用，因此需要支持人员。尽管借助于计算机的帮助，现在一个更小的支持人员队伍就足够了。但是在本质上，这一思想仍然是有效的。

任何大型项目都需要组织成层次结构。底层是许多小的团队，每个团队由首席程序员领导。在下一层，必须由一名经理人对一组团队进行协调。经验表明，你所管理的每一个人将花费你10%的时间，所以每10个团队的一个小组就需要一名全职的经理人。这些经理人也必须被管理。

Brooks观察到，坏消息不能很好地沿着树向上传播。麻省理工学院的Jerry Saltzer将这一效应称为坏消息二极管（bad-news diode）。因为存在着在两千年前将带来坏信息的信使斩首的古老传统，所以首席程序员或经理人都愿意告诉他的老板项目延期了4个月，并且无论如何都没有满足最终时限的机会。因此，顶层管理者就项目的状态通常不明就里。当不能满足最终时限的情况变得十分明显时，顶层管理者的响应是增加人员，此时Brooks定律就起作用了。

实际上，大型公司拥有生产软件的丰富经验并且知道如果它随意地生产会发生什么，这样的公司趋向于至少是试图正确地做事情。相反，较小的、较新的公司，匆匆忙忙地希望其产品早日上市，不能总是仔细地生产他们的软件。这经常导致远远不是最优化的结果。

Brooks和Mills都没有预见到开放源码运动的成长。尽管该运动取得了某些成功，但是一旦新鲜感消失，它是否还是生产大量高质量软件的切实可行的模型还有待观察。回想早年无线电广播是由业余无线电操作人员占据支配地位的，但是很快就让位于商业无线电台和后来的商业电视台。值得注意的是，最为成功的开放源码软件项目显然使用了首席程序员模型，有一名才智超群者控制着体系结构设计（例如，Linus Torvalds控制着Linux内核，而Richard Stallman控制着GNU C编译器）。

⊖ 兰斯（Reims）——法国东北部城市。——译者注

13.5.3 经验的作用

拥有丰富经验的设计人员对于一个操作系统项目来说至关重要。Brooks指出，大多数错误不是在代码中，而是在设计中。程序员正确地做了吩咐他们要做的事情，而吩咐他们要做的事情是错误的。再多测试软件都无法弥补糟糕的设计说明书。

Brooks的解决方案是放弃图13-11a的经典开发模型而采用图13-11b的模型。此处的想法是首先编写一个主程序，它仅仅调用顶层过程，而顶层过程最初是哑过程。从项目的第一天开始，系统就可以编译和运行，尽管它什么都做不了。随着时间的流逝，模块被插入到完全的系统中。这一方法的成效是系统集成测试能够持续地执行，这样设计中的错误就可以更早地显露出来。实际上，拙劣的设计决策导致的学习过程在软件生命周期中应该更早就开始。

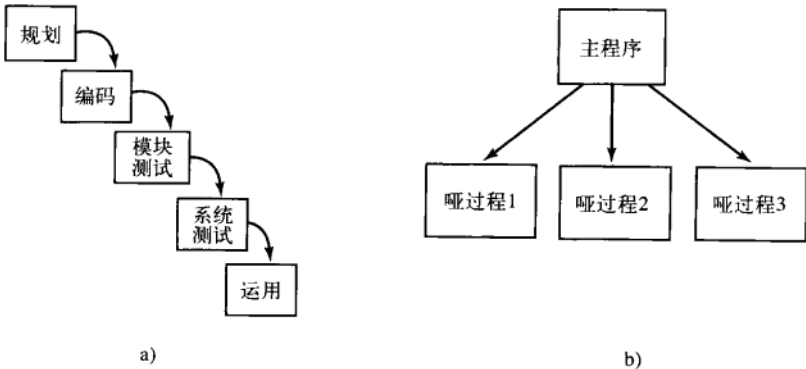


图13-11 a) 传统的分阶段软件设计过程；b) 另一种设计在第一天开始就产生一个（什么都不做的）工作系统

缺乏知识是一件危险的事情。Brooks注意到被他称为第二系统效应（second system effect）的现象。一个设计团队生产的第一件产品经常是最小化的，因为设计人员担心它可能根本就不能工作。结果，他们在加入许多功能特性方面是迟疑的。如果项目取得成功，他们会构建后续的系统。由于被他们自己的成功所感动，设计人员在第二次会包含所有华而不实的东西，而这些都是他们在第一次有意省去的。结果，第二个系统臃肿不堪并且性能低劣。第二个系统的失败使他们在第三次冷静下来并且再次小心谨慎。

就这一点而言，CTSS和MULTICS这一对系统是一个明显的例子。CTSS是第一个通用分时系统并且取得了巨大的成功，尽管它只有最小化的功能。它的后继者MULTICS过于野心勃勃并因此而吃尽了苦头。MULTICS的想法是很好的，但是由于存在太多新的东西所以多年以来系统的性能十分低劣并且绝对不是一个重大的商业成功。在这一开发路线中的第三个系统UNIX则更加小心谨慎并且更加成功。

13.5.4 没有银弹

除了《人月神话》，Brooks还写了一篇有影响的学术论文，称为“*No Silver Bullet*”（没有银弹）（Brooks, 1987）。在这篇文章中，他主张在十年之内由各色人等兜售的灵丹妙药中，没有一样能够在软件生产率上产生数量级的改进。经验表明他是正确的。

在建议的银弹中，包括更好的高级语言、面向对象的程序设计、人工智能、专家系统、自动程序设计、图形化程序设计、程序验证以及程序设计环境。或许在下一个十年将会看到一颗银弹，或许我们将只好满足于逐步的、渐进的改进。

13.6 操作系统设计的趋势

做预测总是困难的——特别是关于未来。例如，1899年美国专利局局长Charles H. Duell请求当时的总统McKinley（麦金利）取消专利局（以及他的工作！），因为他声称“每件能发明的事物都已经发明了”（Cerf and Navasky, 1984）。然而，Thomas Edison（托马斯·爱迪生）在几年之内就发明了几件新的物品，包括电灯、留声机和电影放映机。让我们将新电池装入我们的水晶球中，并且冒险猜测一下在最近的未来操作系统将走向何方。

13.6.1 虚拟化

虚拟化重回时代。它第一次出现在1967年的IBM CP/CMS系统中，现在它重回奔腾平台。最近许多计算机在裸机上运行管理程序，如图13-12所示。管理程序会创建多个虚拟机，每个虚拟机有单独的操作系统。有些计算机利用一个虚拟机为遗产应用创建Windows系统，利用几个虚拟机为当前应用运行Linux系统，或许也会在其他虚拟机上运行若干实验性操作系统。这种现象在第8章已经讨论，并且是未来的发展趋势。

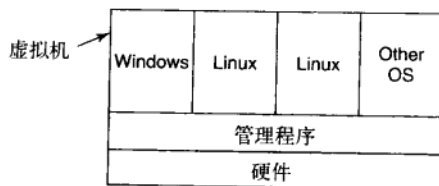


图13-12 运行4个虚拟机的管理程序

13.6.2 多核芯片

多核芯片已经出现，但即使是双核，针对它们的操作系统还没有很好地利用其能力，更不用提64核。这些核会做什么事情？它们需要哪些软件？这些目前都是未知的。起初，人们试图通过对当前操作系统打补丁的方法来支持多核，但锁表问题和其他软件资源的问题使得这种方法不太可能成功，因此需要全新的思路来解决这些问题。

虚拟化和多核芯片的结合创造了一个全新的环境，这里CPU的数目是可编程的。对于8核芯片，软件可以在下列情况做同样的事情：只利用1个CPU而忽略其他7个；使用全部8个CPU；利用双道虚拟化获得16个虚拟CPU；利用四道虚拟化获得32个虚拟CPU；或更多其他组合。程序可以在启动时指定所需CPU数目，由操作系统来保证程序需求的满足。

13.6.3 大型地址空间操作系统

随着计算机从32位地址空间转向64位地址空间，操作系统设计中的重大转变成为可能。32位地址空间并不大。如果你通过给地球上的每个人提供他或她自己的字节来试图分割 2^{32} 个字节，那么将没有足够的字节可以提供。相反， 2^{64} 大约是 2×10^{19} 。现在每个人可以得到他或她个人的3GB大的一块。

对于 2×10^{19} 字节的地址空间我们能做什么呢？首先，可以淘汰文件系统概念。作为替代，所有文件在概念上可以始终保存在（虚拟）内存中。毕竟在那里存在足够的空间，可以放下超过10亿部全长的电影，每一部压缩到4GB。

另一个可能的用途是永久对象存储。对象可以在地址空间中创建，并且保存在那里直到所有对它们的引用消失，在此时它们可以自动被删除。这样的对象在地址空间中是永久的，甚至是在关机和重新启动计算机的时候。有了64位的地址空间，在用光地址空间之前，可以用每秒100MB的速率创建对象长达5000年。当然，为了实际存储这么大量的数据，需要许多磁盘存储器用于分页交换，但是在历史上这是第一次限制因素是磁盘，而不是地址空间。

由于大量数目的对象在地址空间中，允许多个进程同时在相同的地址空间中运行，以便以一般的方式共享对象就变得十分有趣了。这样的设计显然会通向与我们现在所使用的操作系统完全不同的操作系统。有关这一概念的某些思想包含在参考文献（Chase等人，1994）中。

就64位地址而言，另一个必须重新思考的操作系统问题是虚拟内存。对于 2^{64} 字节的虚拟地址空间和8KB的页面，我们有 2^{51} 个页面。常规的页表不能很好地按比例变换到这样的大小，所以需要别的东西。反转的页表是可行的，但是也有人提出了其他的想法（Talluri等人，1995）。无论如何，64位操作系统为新的研究提供了大量的余地。

13.6.4 联网

当前的操作系统是为独立的计算机而设计的。联网是事后添加的，并且一般通过特殊的程序访问，例如Web浏览器、FTP或telnet。在将来，联网或许将会是所有操作系统的基础。不具备网络连接的独立的计算机就像是没有连接到电话网的电话机一样罕见。并且很可能几Gbps的连接是标准的速率。

操作系统将不得不改变以适应这一范型的转变。本地数据与远程数据的区别可能会模糊到这样的程度：实际上没有人知道或者关心数据存放在什么地方。任何地方的计算机能够像本地数据一样处理任何地方的数据。在一个有限的范围内，对于NFS而言这已经成为现实，但是它很可能变得更加普遍并且更好地集成。

对于Web的访问现在需要特殊的程序(浏览器),将来可能会以一种无缝的方式完全集成到操作系统中。存储信息的标准方式可能会变为Web页面,并且这些页面可能包含各种各样的非文本项目,包括音频、视频、程序以及其他,它们全部作为操作系统的基本数据而管理。

13.6.5 并行系统与分布式系统

另一个活跃的领域是并行系统与分布式系统。当前的多处理器操作系统和多计算机操作系统只是标准的单处理器操作系统对调度器进行了轻微的调整,以便对并行性处理得好一点。在将来,我们可能会看到这样的操作系统,其中并行性比现在更加处于中心地位。如果在多处理器配置下台式计算机很快拥有2个、4个或更多的CPU,这一效应将会大大地激发。这就可能导致许多应用程序为多处理器而设计,并且就要求操作系统对并发性要求提供更好的支持。

在未来几年,多计算机很可能在大规模科学与工程超级计算机中占据支配地位,但是它们的操作系统还相当原始。进程安置、负载平衡以及通信都需要做大量的工作。

目前分布式系统经常作为中间件来构建,因为现有的操作系统没有为分布式应用程序提供正确的设施。今后,操作系统的设计将会考虑到分布式系统,所以从一开始所有必要的功能特性在操作系统中就已经存在了。

13.6.6 多媒体

多媒体系统在计算机世界里显然是一颗正在兴起的明星。如果计算机、立体声音响、电视机和电话机全部合并在一起成为一个单一的设备,能够支持高质量的静止图像、音频和视频,并且连接到高速网络中,从而能够轻松地下载、交换和远程访问这些文件,可能不会有人感到吃惊。这些设备的操作系统,甚至是独立的音频和视频设备的操作系统,与现在的操作系统在本质上是不同的。特别地,实时保证是必须的,这将推动系统设计。此外,消费者完全不能容忍他们的数字电视时不时地崩溃,所以要求更好的软件质量和容错性。还有,多媒体文件倾向于非常长,所以必须改造文件系统以便能够有效地处理它们。

13.6.7 电池供电的计算机

功能强大的台式计算机(可能拥有64位地址空间、高带宽网络、多处理器以及高品质的音频和视频)无疑很快就会普及。它们的操作系统必然与目前的操作系统有重大的区别,以便处理所有这些需求。然而,市场上增长甚至更快的部分是电池供电的计算机,包括笔记本、掌上机、Webpad、100美元的膝上机以及智能手机。它们中的某些机种拥有与外部世界的无线连接,其他的机种当它们不在家中与基站对接时将运行在非连接的模式下。这就需要不同的操作系统,它们比当前的操作系统更加小巧、快速、灵活和可靠。在这里,各种各样的微内核与可扩展的系统可能形成这类操作系统的基础。

这些操作系统必须处理完全连接(也就是有线连接)、弱连接(也就是无线连接)和非连接操作,包括离线前的数据储藏和返回在线时的一致性分析,这些都要比当前的系统更好。它们还必须能比当前的系统更好地处理移动问题(例如找到一个激光打印机,登录到其上,并且通过无线电波把文件发送给它)。电源管理是必需的,这包括在操作系统与应用程序之间关于剩余多少电池电量以及电池如何最好利用的大量对话框。动态地改装应用程序以处理微小屏幕的局限可能变得十分重要。最后,新的输入和输出模式(包括手写和语音)可能需要操作系统的新技术以改善品质。电池供电、掌上无线、语音操作的计算机,与具有4个64位CPU的多处理器以及以GB为单位光纤网络连接的台式计算机,两者的操作系统不太可能有很多共同之处。当然,还存在无数的混交机种具有它们自己的需求。

13.6.8 嵌入式系统

新型操作系统将高速增长的最后领域是嵌入式系统。处于洗衣机、微波炉、玩具、晶体管收音机、MP3播放器、便携式摄像机、电梯以及心脏起搏器内部的操作系统将不同于上面的所有操作系统,并且很可能相互之间也不相同。每个操作系统或许都需要仔细地剪裁以适应其特定的应用,因为任何人都不大可能将一块PCI卡插入心脏起搏器将其变成一个电梯控制器。由于所有的嵌入式系统在设计时就知道它只运行有限数目的程序,所以对其进行优化是可能的,而这样的优化在通用系统中是做不到的。

对于嵌入式系统而言,一种有希望的思路是可扩展的操作系统(例如Paramacium和Exokernel)。这些操作系统可以随着应用程序的需要而被构建成轻量级的或重量级的,但是以一种应用程序间一致的方

式。因为嵌入式系统将以上亿的量级生产，所以对于新型操作系统而言这是一个主要的市场。

13.6.9 传感节点

虽然传感网络的市场并不大，但它们正被部署在很多环境中，从楼宇/边境监控到森林火险监测等。传感器是低成本、低功耗的，需要特别精简的操作系统，仅比运行时函数库复杂一些。随着功能强大的传感节点越来越便宜，我们会看到实际的操作系统运行其上，并尽可能针对其任务进行优化、尽可能节约功耗。一般来说，其电池寿命以月衡量，而无线传输是主要的电源消耗者，因此这些系统应该以节能为首要目标。

13.7 小结

操作系统的设计开始于确定它应该做什么。接口应该是简单的、完备的和高效的。应该拥有一个清晰的用户界面范型、执行范型和数据范型。

系统应该具有良好的结构，使用若干种已知技术中的一种，例如分层结构或客户-服务器结构。内部组件应该是相互正交的，并且要清楚地分离策略与机制。大量的精力应该投入到诸如静态与动态数据结构、命名、绑定时机以及模块实现次序这样的一些问题上。

性能是重要的，但是优化应该仔细地选择，从而使优化不致于破坏系统的结构。空间-时间权衡、高速缓存、线索、利用局部性以及优化常见的情况等技术通常都值得尝试。

两三个人编写一个系统与300个人生产一个大型系统是不同的。在后一种情况下，团队结构和项目管理对于项目的成败起着至关重要的作用。

最后，操作系统在未来几年必须进行变革以跟上新的趋势和迎接新的挑战。这些趋势和挑战包括基于管理程序的系统、多核系统、64位地址空间、大规模的网络连接、大规模多处理器、多媒体、掌上无线计算机、嵌入式系统及其传感节点。对于操作系统设计人员来说今后几年将十分令人激动。

习题

1. 摩尔定律 (Moore's law) 描述了一种指数增长现象，类似于将一个动物物种引入到具有充足食物并且没有天敌的新环境中生长。本质上，随着食物供应变得有限或者肉食动物学会了捕食新的被捕食者，一条指数增长曲线可能最终成为一条具有一个渐进极限的S形曲线。讨论可能最终限制计算机硬件改进速率的因素。
2. 图13-1显示了两种范型：算法范型和事件驱动范型。对于下述每一种程序，哪一范型可能更容易使用：
 - a) 编译器
 - b) 照片编辑程序
 - c) 工资单程序
3. 在某些早期的苹果Macintosh计算机上，GUI代码是在ROM中的。为什么？
4. Corbat6的格言是系统应该提供最小机制。这里是一份POSIX调用的列表，这些调用也存在于UNIX版本7中。哪些是冗余的？换句话说，哪些可以被删除而不损失功能性，因为其他调用的简单组合可以做同样的工作并具有大体相同的性能。access、alarm、chdir、chmod、chown、chroot、close、creat、dup、exec、exit、fcntl、fork、fstat、ioctl、kill、link、lseek、mkdir、mknod、open、pause、pipe、read、stat、time、times、umask、unlink、utime、wait和write。
5. 在一个基于微内核的客户-服务器系统中，微内核只做消息传递而不做其他任何事情。用户进程仍然可以创建和使用信号量吗？如果是，怎样做？如果不是，为什么不能？
6. 细致的优化可以改进系统调用的性能。考虑这样一种情况，一个系统调用每10ms调用一次，一次调用花费的平均时间是2ms。如果系统调用能够加速两倍，花费10s的一个进程现在要花费多少时间运行？
7. 请在零售商店的上下文中简要讨论一下机制与策略。
8. 操作系统经常在外部和内部这两个不同的层次上实现命名。这些名字就如下性质有什么区别？
 - a) 长度
 - b) 惟一性
 - c) 层次结构
9. 处理大小事先未知的表格的一种方法是将其大小固定，但是当表格填满时，用一个更大的表格取代它，并且将旧的表项复制到新表中，然后释放旧的表格。使新表的大小是原始表格大

- 小的2倍，与新表的大小只是原始表格大小的1.5倍相比，有什么优点和缺点？
10. 在图13-5中，标志found用于表明是否找到一个PID。忽略found而只是在循环的结尾处测试p以了解是否到达结尾，这样做可行吗？
 11. 在图13-6中，条件编译隐藏了Pentium与Ultra SPARC的区别。相同的方法可以用于隐藏拥有一块IDE磁盘作为惟一磁盘的Pentium与拥有一块SCSI磁盘作为惟一磁盘的Pentium之间的区别吗？这是一个好的思路吗？
 12. 间接是使一个算法更加灵活的一种方法。它有缺点吗？如果有的话，有哪些缺点？
 13. 可重入的过程能够拥有私有静态全局变量吗？讨论你的答案。
 14. 图13-7b中的宏显然比图13-7a中的过程效率更高。然而，它的一个缺点是难于阅读。它还存在其他缺点吗？如果有的话，还有哪些缺点？
 15. 假设我们需要一种方法来计算一个32位字中1的个数是奇数还是偶数。请设计一种算法尽可能快地执行这一计算。如果必要，你可以使用最大256KB的RAM来存放各种表。编写一个宏实现你的算法。附加分：编写一个过程通过在32个位上进行循环来做计算。测量一下你的宏比过程快多少倍。
 16. 在图13-8中，我们看到GIF文件如何使用8位的值作为索引检索一个调色板。相同的思路可以用于16位宽的调色板。在什么情况下（如果有的话），24位的调色板是一个好的思路？
 17. GIF的一个缺点是图像必须包含调色板，这会增加文件的大小。对于一个8位宽的调色板而言，达到收支平衡的最小图像大小是多少？对于16位宽的调色板重复这一问题。
 18. 在正文中，我们展示了对路径名进行高速缓存使得当查找路径名时可以显著地加速。有时使用的另一种技术是让一个守护程序打开根目录中的所有文件，并且保持它们永久地打开，为的是迫使它们的i节点始终处于内存中。像这样钉住i节点可以进一步改进路径查找吗？
 19. 即使一个远程文件自从记录了一个线索以来没有被删除，它也可能自从最后一次引用以来发生了改变。有哪些可能有用的其他信息要记录？
 20. 考虑一个系统，它将对远程文件的引用作为线索而储藏，例如形如（名字，远程主机，远程名字）。一个远程文件悄悄地被删除然后被取代是可能的。那么线索将取回错误的文件。怎样才能使这一问题尽可能少地发生？
 21. 我们在正文中阐述了局部性经常可以被用来改进性能。但是，考虑一种情况，其中一个程序从一个数据源读取输入并且连续地输出到两个或多个文件中。试图利用文件系统上的局部性在这里可能会导致效率的降低吗？存在解决这一问题的方法吗？
 22. Fred Brooks声称一名程序员每年只能编写1000行调试好的代码，然而MINIX的第一版（13 000行代码）是一个人在3年之内创作的。怎样解释这一矛盾？
 23. 使用Brooks每名程序员每年1000行的数字，估计生产Windows Vista花费的资金数量。假设一名程序员每年的成本是100 000美元（包括日常开销，例如计算机、办公空间、秘书支持以及管理开销）。你相信这一答案吗？如果不相信，什么地方有错误？
 24. 随着内存越来越便宜，可以设想一台计算机拥有巨大容量的电池供电的RAM来取代硬盘。以当前的价格，仅有RAM的低端PC成本是多少？假设1GB的RAM盘对于低端机器是足够的。这样的机器有竞争力吗？
 25. 列举某个装置内部的嵌入式系统中不需要用到的常规操作系统的某些功能特性。
 26. 使用C编写一个过程，在两个给定的参数上做双精度加法。使用条件编译编写该过程，使它既可以在16位机器上工作，也可以在32位机器上工作。
 27. 编写程序，将随机生成的短字符串输入到一个数组中，然后使用下述方法在数组中搜索给定的字符串：a) 简单的线性搜索（蛮力法），b) 自选的更加复杂的方法。对于从小型数组到你的系统所能处理的最大数组这样的数组大小范围重新编译你的程序。评估两种方法的性能。收支平衡点在哪里？
 28. 编写一个程序模拟在内存中的文件系统。

第14章 阅读材料及参考文献

在前13章中我们已经介绍了操作系统的许多内容。本章的目的在于向那些希望对操作系统进行进一步研究的读者提供一些帮助。14.1节列出了向读者推荐的阅读材料，14.2节按照字母顺序列出了本书中所引用的所有书籍和文章。

除了下面给出的参考书目以外，奇数年份举行的ACM操作系统原理学术会议（Symposium on Operating Systems Principles, SOSP）和偶数年份举行的USENIX操作系统设计与实现学术会议（Symposium on Operating Systems Design and Implementation, OSDI）也是了解目前操作系统领域研究工作的很好渠道。一年一度的Eurosys 200x 会议也有一流的文章。还可以在《ACM Transactions on Computer Systems》和《ACM SIGOPS Operating Systems Review》两份杂志中找到一些相关的文章。另外ACM、IEEE和USENIX的许多会议也涉及到有关的内容。

14.1 进行深入阅读的建议

在以下各小节中，我们给出一些深入阅读的建议。与本书中标题为“与……有关的研究工作”小节中引用的那些有关当前研究工作的文章不同，这些参考资料实际上多数属于入门和培训一类的。不过，可以把它们看作本书中所介绍内容的不同视角和不同的着重点。

14.1.1 简介及概要

Silberschatz et al., *Operating System Concepts with Java, 7th ed.*

关于操作系统的一本通用教材。它涵盖了进程、内存管理、存储管理、保护和安全、分布式系统和一些专用系统等方面的内容。书中给出了两个实例：Linux和Windows XP。书的封面全是恐龙，这与公元2000年的操作系统有什么关系不十分清楚。

Stallings, *Operating Systems, 5th ed.*

这是有关操作系统的另一本教科书。它涵盖了所有传统的内容，还包括少量分布式系统的内容。

Stevens and Rago, *Advanced Programming in the UNIX Environment*

该书叙述如何使用UNIX系统调用接口以及标准C库编写C程序。有基于System V第4版以及UNIX 4.4BSD版的例子。有关这些实现与POSIX的关系在书中有具体叙述。

Tanenbaum and Woodhull, “Operating Systems Design and Implementation”

一个通过动手实践来学习操作系统的方法。这本书主要介绍了一些常见的原理，另外详细介绍了一个真实的操作系统——MINIX3。并且附带了这个操作系统的清单。

14.1.2 进程和线程

Andrews and Schneider, “Concepts and Notations for Concurrent Programming”

这是一本关于进程和进程间通信的教程，包括忙等待、信号量、管程、消息传递以及其他技术。文章中同时也说明了这些概念是如何嵌入到不同编程语言中去的。这篇文章非常老，但是却经受住了时间的考验。

Ben-Ari, *Principles of Concurrent Programming*

这本书专门讨论了进程间的通信问题，其他章节则讨论了互斥性、信号量、管程以及哲学家就餐问题等。

Silberschatz et al., *Operating System Concepts with Java, 7th ed.*

该书的第4~6章讨论了进程和进程间通信，包括调度、临界区、信号量、管程以及经典的进程间通信问题。

14.1.3 存储管理

Denning, "Virtual Memory"

该文是一篇关于虚拟内存诸多特性的经典文章。作者Denning是该领域的先驱之一，正是他创立了工作集概念。

Denning, "Working Sets Past and Present"

该书很好地阐述了大容量存储器的管理和页面置换算法。书后附有完整的参考文献。虽然其中很多文章都非常老了，但是原理实际根本没有变化。

Knuth, *The Art of Computer Programming, Vol. 1*

该书讨论并比较了首次适配算法、最佳适配算法和其他一些存储管理算法。

Silberschatz et al., *Operating System Concepts with Java, 7th ed.*

该书第8章和第9章讨论了存储管理，包括交换、分页和分段等。书中提到了很多页面置换算法。

14.1.4 输入/输出

Geist and Daniel, "A Continuum of Disk Scheduling Algorithms"

该文给出了一个通用的磁盘臂调度算法，并给出了详细的模拟和实验结果。

Scheible, "A Survey of Storage Options"

现在存储的方法很多：DRAM，SRAM，SDRAM，闪存，硬盘，软盘，CD-ROM，DVD，还有磁带等。这篇文章对这些技术进行了评述，着重总结了它们的优缺点。

Stan and Skadron, "Power-Aware Computing"

能源问题始终是移动设备的主要问题，直到有人能设法将摩尔定律运用于电池技术为止。我们甚至有可能在不久就需要一个可感知温度的操作系统。这篇文章就是针对这些问题的一个综述，同时介绍对能源感知计算中的计算机这一特定问题的5篇文章。

Walker and Cragon, "Interrupt Processing in Concurrent Processors"

在超标量计算机中精确实现中断是一项具有挑战性的工作。其技巧在于将状态序列化并且快速地完成这项工作。文中讨论了许多设计问题以及相关的权衡考虑。

14.1.5 文件系统

McKusick et al., "A Fast File System for UNIX"

在4.2BSD环境下重新实现了UNIX的文件系统。该文描述了新文件系统的设计，并把重点放在其性能上。

Silberschatz et al., *Operating System Concepts with Java, 7th ed.*

该书第10~11章与文件系统有关，涉及文件操作、文件访问方式、目录以及实现和其他内容等。

Stallings, *Operating Systems, 5th ed.*

该书第12章包括许多有关安全环境的内容，特别是有关黑客、病毒以及其他威胁等内容。

14.1.6 死锁

Coffman et al., "System Deadlocks"

该文简要介绍了死锁、死锁的产生原因以及如何预防和检测。

Holt, "Some Deadlock Properties of Computer Systems"

该文围绕死锁进行了讨论。Holt引入了一个可用来分析某些死锁情况的有向图模型。

Isloor and Marsland, "The Deadlock Problem: An Overview"

这是关于死锁的入门教程，重点放在了数据库系统，也介绍了多种模型和算法。

Shub, "A Unified Treatment of Deadlock"

这是一部关于死锁产生和解决的简短综述，同时也给出了一些在教学时应当强调内容的建议。

14.1.7 多媒体操作系统

Lee, "Parallel Video Servers: A Tutorial"

许多组织想提供视频点播，这就需要拥有可扩展的、容错的并行视频服务器。文中介绍了怎样构建

这种服务器的主要问题,包括服务器的体系结构、条带化、布局策略、负载平衡、冗余、协议以及同步。

Leslie et al., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications"

许多多媒体实现的尝试都是在现有的操作系统上增加一些功能,另一个方向是全部重新开发,就像该文描述的一样,为多媒体构建一个新的操作系统,不需要为向下兼容而进行修改。这样做的结果是产生一个与传统操作系统截然不同的新系统。

Sitaram and Dan, "Multimedia Servers"

多媒体服务器与通常的文件服务器有很多不同之处。作者详细讨论了这些不同,特别是在调度、存储子系统和高速缓存这几个方面。

14.1.8 多处理机系统

Ahmad, "Gigantic Clusters: Where Are They and What Are They Doing?"

为了解大型多计算机系统的先进性,可以读这篇文章。它描述了这一思想,并且给出了对当前在使用的一些大型系统的概况介绍。根据摩尔定律可以合理推断,这里提到的规模大约每两年就会增长一倍。

Dubois et al., "Synchronization, Coherence, and Event Ordering in Multiprocessors"

该文是一个关于基于共享存储器多处理器系统中同步问题的指南,而且,其中的一些思想对于单处理器和分布式存储系统也是适用的。

Geer, "For Programmers, Multicore Chips Mean Multiple Challenges"

多核芯片的时代正在到来——不论软件界的人们是否准备好。实际上他们并没有准备好,而且为这些芯片编写程序往往是巨大的挑战,这包括选择合适的工具、将有关工作划分成小的部分,以及测试结果等。

Kant and Mohapatra, "Internet Data Centers"

Internet数据中心是一个被兴奋剂刺激起来的巨大多计算机。常常让成千上万台计算机为一个应用软件而工作。这里的主要问题就是可伸缩性、可维护性和能源。这篇文章既是对有关问题的一个介绍,也是对同一个问题的其他4篇文章的介绍。

Kumar et al., "Heterogeneous Chip Multiprocessors"

用在台式电脑上的多核芯片是对称的——每一个核是相同的。然而对一些应用软件来说,异构的多处理器(Chip multiprocessors, CMPS)是很普遍的,有的核用来计算、有的处理视频编码、有的处理音频编码等。这篇文章就讨论异构多处理器中的有关问题。

Kwok and Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"

如果提前知道所有作业的特性,就可能对多计算机系统或者多处理器进行优化作业调度。问题在于最优调度的计算时间会很长。在这篇论文中,作者讨论并且比较了用不同方法解决这个问题的27种著名的算法。

Rosenblum and Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends"

这篇文章从虚拟机监视器的历史谈起,然后讨论现有处理器、内存和I/O的虚拟化情况。文章涉及的领域不但包括上面三个方面,还包括将来如何用硬件来减少这些问题。

Whitaker et al., "Rethinking the Design of Virtual Machine Monitors"

多数计算机都有一些奇怪和难于虚拟化的方面。在这篇文章中Denali系统的作者就虚拟化进行了一些探讨,即如何修改客户操作系统以避免遇到一些奇怪的特性,由此来防止仿真这些特性。

14.1.9 安全

Bratus, "What Hackers Learn That the Rest of Us Don't"

是什么让黑客如此与众不同?他们所关注的,而一般程序员却忽略的,是什么?他们对API态度不同吗?支末问题重要吗?读者好奇吗?如果感兴趣,建议读一读。

Computer, Feb 2000

这一期Computer的主题是生物测量学。关于这个主题有6篇论文,从入门到专题,从各种特定技术

到法律和隐私问题，论文都有涉及。

Denning, *Information Warfare and Security*

信息已经变成了战争武器，既是军事武器也是军事配合武器。参与者不仅尝试攻击对方的信息系统，而且要防卫好自己的系统。在这本吸引人的书中，作者讨论了所有能想到的关于攻击策略和防卫策略的话，从数据欺骗到包窥探器。该书对于计算机安全有极大兴趣的读者来说是必读的。

Ford and Allen, "How Not to Be Seen"

病毒，间谍软件，rootkits和数字版权管理系统都对隐藏数据情有独钟。这篇文章对各种隐身的方法进行了简要的介绍。

Hafner and Markoff, *Cyberpunk*

书中介绍了由《纽约时报》曾经写过网络蠕虫故事（马尔可夫链）的计算机记者讲述的世界上关于年轻黑客破坏计算机的三种流传最广的故事。

Johnson and Jajodia, "Exploring Steganography: Seeing the Unseen"

隐身术具有悠久的历史，可以回到将信使的头发剃光，然后在剃光的头上纹上信息，然后在信使的头发长出来之后再将他送走的年代。尽管当前的技术很多，但是它们也是数字化的。本书对于想在这一主题彻底入门的读者来说是一个开端。

Ludwig, *The Little Black Book of Email Viruses*

如果想编写反病毒软件并且想了解在位级别（bit level）上这些病毒是怎么工作的，那么这本书很适合。每种病毒都有详细的讨论并且也提供了绝大多数的实际代码。但是，要求读者透彻掌握 Pentium 汇编语言编程知识。

Mead, "Who is Liable for Insecure Systems?"

很多有关计算机安全的措施都是从技术角度出发的，但是这并非惟一的角度。也许软件经销商应该对由于他们的问题软件而带来的损失负起责任。如果比现在更多地关注于安全，这会是经销商的机会吗？对这个提法感兴趣吗？可以读一下这篇文章。

Milojicic, "Security and Privacy"

安全性涉及很多方面，包括操作系统、网络、私密性表示等。在这篇文章中，6位安全方面的专家给出了他们各自关于这个主题的想法和见解。

Nachenberg, "Computer Virus-Antivirus Coevolution"

当反病毒的开发人员找到一种方法能够探测某种电脑病毒并且使其失效时，病毒的编写者已经在改进和开发更强的病毒。本书探讨了这种制造病毒和反病毒之间的“猫和老鼠”游戏。作者对于反病毒编写者能否取胜这场游戏并不持乐观态度，这对电脑用户来说也许不是一个好消息。

Pfleeger, *Security in Computing*, 4th ed.

尽管已经出版了很多关于计算机安全的书籍，但大多数却只关注网络安全性。本书不仅关注网络安全，还包含了讨论操作系统安全性、数据库安全性和分布式系统安全性的章节。

Sasse, "Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems"

作者讲述了他许多大机场所经历的瞳孔识别系统的体验。不是所有的体验都是正面的。

Thibadeau, "Trusted Computing for Disk Drives and Other Peripherals"

如果读者认为磁盘驱动器只是一个储存比特的地方，那么最好再考虑一下。现代的磁盘驱动器有非常强大的CPU，兆级的RAM，多个通信通道甚至有自己的启动ROM。简而言之，它就是一个完整的计算机系统，很容易被攻击，因此它也需要有自己的保护机制。这篇文章讨论的就是磁盘驱动器的安全问题。

14.1.10 Linux

Bovet and Cesati, *Understanding the Linux Kernel*

该书也许是对Linux内核整体知识讨论最好的一本书。它涵盖了进程、存储管理、文件系统和信号等内容。

IEEE, *Information Technology—Portable Operating System Interface (POSIX), Part 1: System*

Application Program Interface (API) [C Language]

这是一个标准。一些部分确实值得一读，特别是附录B，清晰阐述了为什么要这样做。参考标准的一个好处在于通过定义不会出现错误。例如，如果一个宏的名字中的排字错误贯穿了整个编辑过程，那么它将不再是一个错误，而成为一种正式标准。

Fusco, *The Linux Programmers' Toolbox*

这本书是为那些知道一些基本Linux知识，并且希望能够进一步了解Linux程序如何工作的中级读者们写作的。该书假定读者是一个C程序员。

Maxwell, *Linux Core Kernel Commentary*

该书的前400页给出了Linux的内核源代码的一个子集。后面的150页则是对这些代码的评述。与John Lions的经典书籍(1996)风格很相似。如果你了解Linux内核的很多细节，那么这是一个不错的起点，但是读40 000行C语言代码不是每个人都必需的。

14.1.11 Windows Vista

Cusumano and Selby, "How Microsoft Builds Software"

你是否曾经好奇过一个人如何能够写出29 000 000行代码(就像Windows 2000一样)，并且让它作为一个整体运转起来?希望探究微软是如何采用建造和测试循环来管理大型软件项目的读者，可以参看这篇论文。其过程相当有启发性。

Rector and Newcomer, *Win32 Programming*

如果想找一本1500页的书，告诉你如何编写Windows程序，那么读这本书是一个不错的开始。它涵盖了窗口、设备、图形输出、键盘和鼠标输入、打印、存储管理、库和同步等许多主题。阅读这本书要求读者具有C或者C++语言的知识。

Russinovich and Solomon, *Microsoft Windows Internals, 4th ed.*

如果想学习如何使用Windows，可能会有几百种相关的书。如果想知道Windows内部如何工作的，本书是读者最好的选择。它给出了很多内部算法和数据结构以及可观的技术细节。没有任何一本书可以替代。

14.1.12 Symbian操作系统

Cinque et al., "How do Mobile Phone Fail? A Failure Data Analysis of Symbian OS Smart Phones"

以前不论怎样，当计算机崩溃时，至少电话总是可以用的。而现在电话其实就是一个小屏幕的计算机，它们也会因为糟糕的软件而崩溃。本文讨论了可以导致Symbian手机或者终端崩溃的软件错误。

Morris, *The Symbian OS Architecture Sourcebook*

如果你一直在寻找关于Symbian操作系统的进一步细节，那么本书是一个很好的开始。它涉及了Symbian的体系结构和相当数量的各层细节，而且还给出了一些实例分析。

Stichbury and Jacobs, *The Accredited Symbian Developer Primer*

如果你对Symbian手机或者掌上电脑如何开发应用软件感兴趣的话，那么本书是一个不错的选择。它从所需要的C++语言讲起，逐步深入到系统结构、文件系统、网络管理、工具链和兼容性。

14.1.13 设计原则

Brooks, *The Mythical Man Month: Essays On Software Engineering*

Fred Brooks是IBM的OS/360的主要设计者之一。以其丰富的经验，他知道在计算机的设计中什么是可以运行的和什么是不能运行的。他在25年前写下这本诙谐且内涵丰富的书中给出的建议现在一样是可行的。

Cooke et al., "UNIX and Beyond: An Interview with Ken Thompson"

设计一个操作系统与其说是一门科学，不如说是一门艺术。因此，倾听该领域专家的谈话是一个学习这方面知识的有效途径。在操作系统领域中，没有谁比Ken Thompson更有发言权的了。在对这位UNIX、Inferno、Plan9操作系统的合作设计者的访问过程中，Ken Thompson阐明了在这个领域中我们从哪里开始和即将走向哪里等问题。

Corbató, "On Building Systems That Will Fail"

在获得图灵奖的演讲大会上，这位分时系统之父阐述了许多Brooks在《人月神话》中同样关注的问

题。他的结论是所有的复杂系统都将最终失败，为了设计一个成功的系统，避免复杂化、追求设计上的优雅风格和简单化原则是绝对重要的。

Crowley, *Operating Systems: A Design-Oriented Approach*

大多数介绍操作系统的教材仅仅是讲操作系统的基本概念（进程调度、虚拟内存等）和列举一些例子，对于如何设计一个操作系统却没有提及。该书独一无二的特点在于有4章是说明如何设计一个操作系统的。

Lampson, "Hints for Computer System Design"

Butler Lampson——世界上最主要的具有创新性的操作系统设计者之一，在他多年的设计经历中总结了许多设计方法、对设计的建议和一些指导原则并写下这篇诙谐的内涵丰富的文章，正如Brooks的书一样，对于有抱负的操作系统的的设计者来说，这本书一定不要错过。

Wirth, "A Plea for Lean Software"

Niklaus Wirth——著名的经验丰富的系统设计者，曾设计了面向网络的基于图形用户界面的操作系统Oberon，包括Oberon编译器和文本编辑器，只有200KB。通过讨论Oberon系统，他阐明软件应该基于简单的概念，使其简单明了，而不是商用化软件的复杂。

14.2 按字母顺序排序的参考文献

- AARAJ, N., RAGHUNATHAN, A., RAVI, S., and JHA, N.K.: "Energy and Execution Time Analysis of a Software-Based Trusted Platform Module," *Proc. Conf. on Design, Automation and Test in Europe*, IEEE, pp. 1128–1133, 2007.
- ABDEL-HAMID, T., and MADNICK, S.: *Software Project Dynamics: An Integrated Approach*, Upper Saddle River, NJ: Prentice Hall, 1991.
- ABDELHAFEZ, M., RILEY, G., COLE, R.G., and PHAMDO, N.: "Modeling and Simulation of TCP MANET Worms," *Proc. 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, IEEE, pp. 123–130, 2007.
- ABRAM-PROFETA, E.L., and SHIN, K.G.: "Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Comp. Syst.*, IEEE, pp. 66–75, 1998.
- ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., and YOUNG, M.: "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, USENIX, pp. 93–112, 1986.
- ADAMS, G.B. III, AGRAWAL, D.P., and SIEGEL, H.J.: "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks," *Computer*, vol. 20, pp. 14–27, June 1987.
- ADAMS, K., and AGESON, O.: "A Comparison of Software and Hardware Techniques for X86 Virtualization," *Proc. 12th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, pp. 2–13, 2006.
- ADYA, A., BOLOSKY, W.J., CASTRO, M., CERMAK, C., CHAIKEN, R., DOUCEUR, J.R., LORCH, J.R., THEIMER, M., and WATTENHOFER, R.P.: "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *Proc. Fifth Symp. on Operating System Design and Implementation*, USENIX, pp. 1–15, 2002.
- AGARWAL, R., and STOLLER, S.D.: "Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables," *Proc. 2006 Workshop on Parallel and Distributed Systems*, ACM, pp. 51–60, 2006.
- AGRAWAL, D., BAKTIR, S., KARAKOYUNLU, D., ROHATGI, P., and SUNAR, B.: "Trojan Detection Using IC Fingerprinting," *Proc. 2007 IEEE Symp. on Security and Privacy*, IEEE, pp. 296–310, May 2007.
- AHMAD, I.: "Gigantic Clusters: Where Are They and What Are They Doing?" *IEEE Concurrency*, vol. 8, pp. 83–85, April–June 2000.
- AHN, B.-S., SOHN, S.-H., KIM, S.-Y., CHA, G.-I., BAEK, Y.-C., JUNG, S.-I., and KIM, M.-J.: "Implementation and Evaluation of EXT3NS Multimedia File System," *Proc. 12th Annual Int'l Conf. on Multimedia*, ACM, pp. 588–595, 2004.
- ALBERS, S., FAVRHOLDT, L.M., and GIEL, O.: "On Paging with Locality of Reference," *Proc. 34th ACM Symp. of Theory of Computing*, ACM, pp. 258–267, 2002.
- AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A., and SUBRAHMANYAM, P.: "VMI: An Interface for Paravirtualization," *Proc. 2006 Linux Symp.*, 2006.

- ANAGNOSTAKIS, K.G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., and KEROMYTI, A.D.: "Deflecting Targeted Attacks Using Shadow Honey pots," *Proc. 14th USENIX Security Symp.*, USENIX, p. 9, 2005.
- ANDERSON, R.: "Cryptology and Competition Policy: Issues with Trusted Computing," *Proc. ACM Symp. on Principles of Distributed Computing*, ACM, pp. 3-10, 2003.
- ANDERSON, T.E.: "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. on Parallel and Distr. Systems*, vol. 1, pp. 6-16, Jan. 1990.
- ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D., and LEVY, H.M.: "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," *ACM Trans. on Computer Systems*, vol. 10, pp. 53-79, Feb. 1992.
- ANDREWS, G.R.: *Concurrent Programming—Principles and Practice*, Redwood City, CA: Benjamin/Cummings, 1991.
- ANDREWS, G.R., and SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.
- ANDREWS, G.R., and SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol. 15, pp. 3-43, March 1983.
- ARNAB, A., and HUTCHISON, A.: "Piracy and Content Protection in the Broadband Age," *Proc. S. African Telecomm. Netw. and Appl. Conf.*, 2006.
- ARNAN, R., BACHMAT, E., LAM, T.K., and MICHEL, R.: "Dynamic Data Reallocation in Disk Arrays," *ACM Trans. on Storage*, vol. 3, Art. 2, March 2007.
- ARON, M., and DRUSCHEL, P.: "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 223-246, 1999.
- ASRIGO, K., LITTY, L., and LIK, D.: "Using VMM-Based Sensors to Monitor Honey pots," *Proc. ACM/USENIX Int'l Conf. on Virtual Execution Environments*, ACM, pp. 13-23, 2006.
- BACHMAT, E., and BRAVERMAN, V.: "Batched Disk Scheduling with Delays," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 33, pp. 36-41, 2006.
- BAKER, F.T.: "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, pp. 1, 1972.
- BAKER, M., SHAH, M., ROSENTHAL, D.S.H., ROUSSOPOULOS, M., MANIATIS, P., GIULL, T.J., and BUNGALIE, P.: "A Fresh Look at the Reliability of Long-Term Digital Storage," *Proc. Eurosys 2006*, ACM, pp. 221-234, 2006.
- BALA, K., KAASHOEK, M.F., and WEIHL, W.: "Software Prefetching and Caching for Translation Lookaside Buffers," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 243-254, 1994.
- BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S.K., and USTUNER, A.: "Thorough Static Analysis of Device Drivers," *Proc. Eurosys 2006*, ACM, pp. 73-86, 2006.
- BARATTO, R.A., KIM, L.N., and NIEH, J.: "THINC: A Virtual Display Architecture for Thin-Client Computing," *Proc. 20th Symp. on Operating System Principles*, ACM, pp. 277-290, 2005.
- BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUBAUER, R., PRATT, I., and WARFIELD, A.: "Xen and the Art of Virtualization," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 164-177, 2003.
- BOEHM, B.: *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.
- BORN, G.: *Inside the Microsoft Windows 98 Registry*, Redmond, WA: Microsoft Press, 1998.
- BOVET, D.P., and CESATI, M.: *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, 2005.
- BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., and SCHIOBERG, H.: "Live Wide-Area Migration of Virtual Machines Including Local Persistent State," *Proc. ACM/USENIX Conf. on Virtual Execution Environments*, ACM, pp. 169-179, 2007.
- BRATUS, S.: "What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum," *IEEE Security and Privacy*, vol. 5, pp. 72-75, July/Aug. 2007.
- BRINCH HANSEN, P.: "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, pp. 199-207, June 1975.
- BRISOLARA, L., HAN, S., GUERIN, X., CARRO, L., REISS, R., CHAE, S., and JERRAYA, A.: "Reducing Fine-Grain Communication Overhead in Multithread Code Generation for Heterogeneous MPSoC," *Proc. 10th Int'l Workshop on Software and Compilers for Embedded Systems*, ACM, pp. 81-89, 2007.
- BROOKS, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*, Reading, MA: Addison-Wesley, 1975.
- BROOKS, F.P., Jr.: "No Silver Bullet—Essence and Accident in Software Engineering," *Computer*, vol. 20, pp. 10-19, April 1987.
- BROOKS, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary edition, Reading, MA: Addison-Wesley, 1995.
- BRUSCHI, D., MARTIGNONI, L., and MONGA, M.: "Code Normalization for Self-Mutating Malware," *IEEE Security and Privacy*, vol. 5, pp. 46-54, March/April 2007.
- BUGNION, E., DEVINE, S., GOVIL, K., and ROSENBLUM, M.: "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. on Computer Systems*, vol. 15, pp. 412-447, Nov. 1997.
- BULPIN, J.R., and PRATT, I.A.: "Hyperthreading-Aware Process Scheduling Heuristics," *Proc. Annual Tech. Conf. USENIX*, pp. 399-403, 2005.
- BURNETT, N.C., BENT, J., ARPACI-DUSEAU, A.C., and ARPACI-DUSEAU, R.H.: "Exploiting Gray-Box Knowledge of Buffer-Cache Management," *Proc. Annual Tech. Conf. USENIX*, pp. 29-44, 2002.
- BURTON, A.N., and KELLY, P.H.J.: "Performance Prediction of Paging Workloads Using Lightweight Tracing," *Proc. Int'l Parallel and Distributed Processing Symp.*, IEEE, pp. 278-285, 2003.
- BYUNG-HYUN, Y., HUANG, Z., CRANFIELD, S., and PURVIS, M.: "Homeless and Home-Based Lazy Release Consistency protocols on Distributed Shared Memory," *Proc. 27th Australasian Conf. on Computer Science*, Australian Comp. Soc., pp. 117-123, 2004.

- CANT, C.: *Writing Windows WDM Device Drivers: Master the New Windows Driver Model*, Lawrence, KS: CMP Books, 2005.
- CARPENTER, M., LISTON, T., and SKOUDIS, E.: "Hiding Virtualization from Attackers and Malware," *IEEE Security and Privacy*, vol. 5, pp. 62-65, May/June 2007.
- CARR, R.W., and HENNESSY, J.L.: "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. Eighth Symp. on Operating Systems Principles*, ACM, pp. 87-95, 1981.
- CARRIERO, N., and GELERNTER, D.: "The S/Net's Linda Kernel," *ACM Trans. on Computer Systems*, vol. 4, pp. 110-129, May 1986.
- CARRIERO, N., and GELERNTER, D.: "Linda in Context," *Commun. of the ACM*, vol. 32, pp. 444-458, April 1989.
- CASCAVAL, C., DUESTERWALD, E., SWEENEY, P.F., and WISNIEWSKI, R.W.: "Multiple Page Size Modeling and Optimization," *Int'l Conf. on Parallel Arch. and Compilation Techniques*, IEEE, 339-349, 2005.
- CASTRO, M., COSTA, M., and HARRIS, T.: "Securing Software by Enforcing Data-flow Integrity," *Proc. Seventh Symp. on Operating Systems Design and Implementation*, USENIX, pp. 147-160, 2006.
- CAUDILL, H., and GAVRILOVSKA, A.: "Tuning File System Block Addressing for Performance," *Proc. 44th Annual Southeast Regional Conf.*, ACM, pp. 7-11, 2006.
- CERF, C., and NAVASKY, V.: *The Experts Speak*, New York: Random House, 1984.
- CHANG, L.-P.: "On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems," *Proc. ACM Symp. on Applied Computing*, ACM, pp. 1126-1130, 2007.
- CHAPMAN, M., and HEISER, G.: "Implementing Transparent Shared Memory on Clusters Using Virtual Machines," *Proc. Annual Tech. Conf.*, USENIX, pp. 383-386, 2005.
- CHASE, J.S., LEVY, H.M., FEELEY, M.J., and LAZOWSKA, E.D.: "Sharing and Protection in a Single-Address-Space Operating System," *ACM Trans. on Computer Systems*, vol. 12, pp. 271-307, Nov. 1994.
- CHATTOPADHYAY, S., LI, K., and BHANDARKAR, S.: "FGS-MR: MPEG4 Fine Grained Scalable Multi-Resolution Video Encoding for Adaptive Video Streaming," *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- CHEN, P.M., NG, W.T., CHANDRAS, AYOCK, C., RAJAMANI, G., and LOWELL, D.: "The Rio File Cache: Surviving Operating System Crashes," *Proc. Seventh Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, pp. 74-83, 1996.
- CHEN, S., and THAPAR, M.: "A Novel Video Layout Strategy for Near-Video-on-Demand Servers," *Proc. Int'l Conf. on Multimedia Computing and Systems*, IEEE, pp. 37-45, 1997.
- CHEN, S., and TOWSLEY, D.: "A Performance Evaluation of RAID Architectures," *IEEE Trans. on Computers*, vol. 45, pp. 1116-1130, Oct. 1996.
- CHEN, S., GIBBONS, P.B., KOZUCH, M., LIASKOVITIS, V., AILAMAKI, A., BLELLOCH, G.E., FALSAFI, B., FIX, L., HARDAVELLAS, N., MOWRY, T.C., and WILKERSON, C.: "Scheduling Threads for Constructive Cache Sharing on CMPs," *Proc. ACM Symp. on Parallel Algorithms and Arch.*, ACM, pp. 105-115, 2007.
- CHENG, J., WONG, S.H.Y., YANG, H., and LU, S.: "SmartSiren: Virus Detection and Alert for Smartphones," *Proc. Fifth Int'l Conf. on Mobile Systems, Apps., and Services*, ACM, pp. 258-271, 2007.
- CHENG, N., JIN, H., and YUAN, Q.: "OMFS: An Object-Oriented Multimedia File System for Cluster Streaming Server," *Proc. Eighth Int'l Conf. on High-Performance Computing in Asia-Pacific Region*, IEEE, pp. 532-537, 2005.
- CHERTON, D.R.: "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *ACM SIGOPS Operating Systems Rev.*, vol. 18, pp. 12-20, Oct. 1984.
- CHERTON, D.R.: "The V Distributed System," *Commun. of the ACM*, vol. 31, pp. 314-333, March 1988.
- CHEKASOVA, L., and GARDNER, R.: "Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor," *Proc. Annual Tech. Conf.*, USENIX, pp. 387-390, 2005.
- CHERVENAK, A., VELLANKI, V., and KURMAS, Z.: "Protecting File Systems: A Survey of Backup Techniques," *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.
- CHIANG, M.-L., and HUANG, J.-S.: "Improving the Performance of Log-Structured File Systems with Adaptive Block Rearrangement," *Proc. 2007 ACM Symp. on Applied Computing*, ACM, pp. 1136-1140, 2007.
- CHILDS, S., and INGRAM, D.: "The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop," *Proc. Seventh IEEE Real-Time Tech. and Appl. Symp.*, IEEE, pp. 135-141, 2001.
- CHOU, A., YANG, J., CHELF, B., HALLEM, S., and ENGLER, D.: "An Empirical Study of Operating System Errors," *Proc. 18th Symp. on Operating Systems Design and Implementation*, ACM, pp. 73-88, 2001.
- CHOW, T.C.K., and ABRAHAM, J.A.: "Load Balancing in Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-8, pp. 401-412, July 1982.
- CINQUE, M., COTRONEO, D., KALBARCZYK, Z., IYER, and RAVISHANKAR K.: "How Do Mobile Phones Fail? A Failure Data Analysis of Symbian OS Smart Phones," *Proc. 37th Annual Int'l Conf. on Dependable Systems and Networks*, IEEE, pp. 585-594, 2007.
- COFFMAN, E.G., ELPHICK, M.J., and SHOSHANI, A.: "System Deadlocks," *Computing Surveys*, vol. 3, pp. 67-78, June 1971.
- COOKE, D., URBAN, J., and HAMILTON, S.: "Unix and Beyond: An Interview with Ken Thompson," *Computer*, vol. 32, pp. 58-64, May 1999.
- CORBATO, F.J.: "On Building Systems That Will Fail," *Commun. of the ACM*, vol. 34, pp. 72-81, June 1991.

- CORBATO, F.J., MERWIN-DAGGETT, M., and DALEY, R.C.: "An Experimental Time-Sharing System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 335-344, 1962.
- CORBATO, F.J., SALTZER, J.H., and CLINGEN, C.T.: "MULTICS—The First Seven Years," *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, pp. 571-583, 1972.
- CORBATO, F.J., and VYSOTSKY, V.A.: "Introduction and Overview of the MULTICS System," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 185-196, 1965.
- CORNELL, B., DINDA, P.A., and BUSTAMANTE, F.E.: "Wayback: A User-Level Versioning File System for Linux," *Proc. Annual Tech. Conf.*, USENIX, pp. 19-28, 2004.
- COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., and BARHAM, P.: "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th Symp. on Operating System Prin.*, ACM, pp. 133-147, 2005.
- COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L.: "Concurrent Control with Readers and Writers," *Commun. of the ACM*, vol. 10, pp. 667-668, Oct. 1971.
- COX, L.P., MURRAY, C.D., and NOBLE, B.D.: "Pastiche: Making Backup Cheap and Easy," *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 285-298, 2002.
- CRANOR, C.D., and PARULKAR, G.M.: "The UVM Virtual Memory System," *Proc. Annual Tech. Conf.*, USENIX, pp. 117-130, 1999.
- CROWLEY, C.: *Operating Systems: A Design-Oriented Approach*, Chicago: Irwin, 1997.
- CUSUMANO, M.A., and SELBY, R.W.: "How Microsoft Builds Software," *Commun. of the ACM*, vol. 40, pp. 53-61, June 1997.
- DABEK, F., KAASHOEK, M.F., KARGET, D., MORRIS, R., and STOICA, I.: "Wide-Area Cooperative Storage with CFS," *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 202-215, 2001.
- DALEY, R.C., and DENNIS, J.B.: "Virtual Memory, Process, and Sharing in MULTICS," *Commun. of the ACM*, vol. 11, pp. 306-312, May 1968.
- DALTON, A.B., and ELLIS, C.S.: "Sensing User Intention and Context for Energy Management," *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 151-156, 2003.
- DASIGENIS, M., KROUPIS, N., ARGYRIOU, A., TATAS, K., SOUDRIS, D., THANAILAKIS, A., and ZERVAS, N.: "A Memory Management Approach for Efficient Implementation of Multimedia Kernels on Programmable Architectures," *Proc. IEEE Computer Society Workshop on VLSI, IEEE*, pp. 171-177, 2001.
- DAUGMAN, J.: "How Iris Recognition Works," *IEEE Trans. on Circuits and Systems for Video Tech.*, vol. 14, pp. 21-30, Jan. 2004.
- DAVID, F.M., CARLYLE, J.C., and CAMPBELL, R.H.: "Exploring Recovery from Operating System Lockups," *Proc. Annual Tech. Conf.*, USENIX, pp. 351-356, 2007.
- DEAN, J., and GHEMAWAT, S.: "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137-150, 2004.
- DENNING, D.: "A Lattice Model of Secure Information Flow," *Commun. of the ACM*, vol. 19, pp. 236-243, 1976.
- DENNING, D.: *Information Warfare and Security*, Reading, MA: Addison-Wesley, 1999.
- DENNING, P.J.: "The Working Set Model for Program Behavior," *Commun. of the ACM*, vol. 11, pp. 323-333, 1968a.
- DENNING, P.J.: "Thrashing: Its Causes and Prevention," *Proc. AFIPS National Computer Conf.*, AFIPS, pp. 915-922, 1968b.
- DENNING, P.J.: "Virtual Memory," *Computing Surveys*, vol. 2, pp. 153-189, Sept. 1970.
- DENNING, P.J.: "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, vol. SE-6, pp. 64-84, Jan. 1980.
- DENNIS, J.B., and VAN HORN, E.C.: "Programming Semantics for Multiprogrammed Computations," *Commun. of the ACM*, vol. 9, pp. 143-155, March 1966.
- DIFFIE, W., and HELLMAN, M.E.: "New Directions in Cryptography," *IEEE Trans. on Information Theory*, vol. IT-22, pp. 644-654, Nov. 1976.
- DIJKSTRA, E.W.: "Co-operating Sequential Processes," in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.
- DIJKSTRA, E.W.: "The Structure of THE Multiprogramming System," *Commun. of the ACM*, vol. 11, pp. 341-346, May 1968.
- DING, X., JIANG, S., and CHEN, F.: "A buffer cache management scheme exploiting both Temporal and Spatial localities," *ACM Trans. on Storage*, vol. 3, Art. 5, June 2007.
- DUBOIS, M., SCHEURICH, C., and BRIGGS, F.A.: "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, vol. 21, pp. 9-21, Feb. 1988.
- EAGER, D.L., LAZOWSKA, E.D., and ZAHORJAN, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-12, pp. 662-675, May 1986.
- EDLER, J., LIPKIS, J., and SCHONBERG, E.: "Process Management for Highly Parallel UNIX Systems," *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, pp. 1-17, Sept. 1988.
- EFSTATHIOPOULOS, P., KROHN, M., VANDERBORG, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIERES, D., KAASHOEK, F., and MORRIS, R.: "Labels and Event Processes in the Asbestos Operating System," *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 17-30, 2005.
- EGAN, J.I., and TEIXEIRA, T.J.: *Writing a UNIX Device Driver*, 2nd ed., New York: John Wiley, 1992.
- EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., and SONG, D.: "Dynamic Spyware Analysis," *Proc. Annual Tech. Conf.*, USENIX, pp. 233-246, 2007.
- EGGERT, L., and TOUCH, J.D.: "Idle-time Scheduling with Preemption Intervals," *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 249-262, 2005.
- EL GAMAL, A.: "A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms," *IEEE Trans. on Information Theory*, vol. IT-31, pp. 469-472, July 1985.

- ELPHINSTONE, K., KLEIN, G., DERRIN, P., ROSCOE, T., and HEISER, G.: "Towards a Practical, Verified, Kernel," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 117-122, 2007.
- ENGLER, D.R., CHELF, B., CHOU, A., and HALLEM, S.: "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions," *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 1-16, 2000.
- ENGLER, D.R., GUPTA, S.K., and KAASHOEK, M.F.: "AVM: Application-Level Virtual Memory," *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 72-77, 1995.
- ENGLER, D.R., and KAASHOEK, M.F.: "Exterminat All Operating System Abstractions," *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 78-83, 1995.
- ENGLER, D.R., KAASHOEK, M.F., and O'TOOLE, J. Jr.: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 251-266, 1995.
- ERICKSON, J.S.: "Fair Use, DRM, and Trusted Computing," *Commun. of the ACM*, vol. 46, pp. 34-39, 2003.
- ETSION, Y., TSAFIR, D., and FEITELSON, D.G.: "Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes," *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 172-183, 2003.
- ETSION, Y., TSAFIR, D., and FEITELSON, D.G.: "Desktop Scheduling: How Can We Know What the User Wants?" *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, pp. 110-115, 2004.
- ETSION, Y., TSAFIR, D., and FEITELSON, D.G.: "Process Prioritization Using Output Production: Scheduling for Multimedia," *ACM Trans. on Multimedia, Computing, and Applications*, vol. 2, pp. 318-342, Nov. 2006.
- EVEN, S.: *Graph Algorithms*, Potomac, MD: Computer Science Press, 1979.
- FABRY, R.S.: "Capability-Based Addressing," *Commun. of the ACM*, vol. 17, pp. 403-412, July 1974.
- FAN, X., WEBER, W.-D., and BARROSO, L.-A.: "Power Provisioning for a Warehouse-Sized Computer," *Proc. 34th Annual Int'l Symp. on Computer Arch.*, ACM, pp. 13-23, 2007.
- FANDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J.R., and LEVI, S.: "Language Support for Fast and Reliable Message-Based Communication in Singularity OS," *Proc. Eurosys 2006*, ACM, pp. 177-190, 2006.
- FASSINO, J.-P., STEFANI, J.-B., LAWALL, J.J., and MULLER, G.: "Think: A Software Framework for Component-Based Operating System Kernels," *Proc. Annual Tech. Conf.*, USENIX, pp. 73-86, 2002.
- FEDOROVA, A., SELTZER, M., SMALL, C., and NUSSBAUM, D.: "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design," *Proc. Annual Tech. Conf.*, USENIX, pp. 395-398, 2005.
- FEENEY, M.J., MORGAN, W.E., PIGHIN, F.H., KARLIN, A.R., LEVY, H.M., and THEK-KATH, C.A.: "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 201-212, 1995.
- FELTEN, E.W., and HALDERMAN, J.A.: "Digital Rights Management, Spyware, and Security," *IEEE Security and Privacy*, vol. 4, pp. 18-23, Jan./Feb. 2006.
- FEUSTAL, E.A.: "The Rice Research Computer—A Tagged Architecture," *Proc. AFIPS Conf.*, AFIPS, 1972.
- FLINN, J., and SATYANARAYANAN, M.: "Managing Battery Lifetime with Energy-Aware Adaptation," *ACM Trans. on Computer Systems*, vol. 22, pp. 137-179, May 2004.
- FLORENCHIO, D., and HERLEY, C.: "A Large-Scale Study of Web Password Habits," *Proc. 16th Int'l Conf. on the World Wide Web*, ACM, pp. 657-666, 2007.
- FLUCKIGER, F.: *Understanding Networked Multimedia*, Upper Saddle River, NJ: Prentice Hall, 1995.
- FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., and SHIVERS, O.: "The Flux Oskit: A Substrate for Kernel and Language Research," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 38-51, 1997.
- FORD, B., HIBLER, M., LEPREAU, J., TULLMAN, P., BACK, G., CLAWSON, S.: "Microkernels Meet Recursive Virtual Machines," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 137-151, 1996.
- FORD, B., and SUSARLA, S.: "CPU Inheritance Scheduling," *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, pp. 91-105, 1996.
- FORD, R., and ALLEN, W.H.: "How Not To Be Seen," *IEEE Security and Privacy*, vol. 5, pp. 67-69, Jan./Feb. 2007.
- FOSTER, I.: "Globus Toolkit Version 4: Software for Service-Oriented Systems," *Int'l Conf. on Network and Parallel Computing*, IFIP, pp. 2-13, 2005.
- FOTHERINGHAM, J.: "Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store," *Commun. of the ACM*, vol. 4, pp. 435-436, Oct. 1961.
- FRANZ, M.: "Containing the Ultimate Trojan Horse," *IEEE Security and Privacy*, vol. 5, pp. 52-56, July-Aug. 2007.
- FRASER, K., and HARRIS, T.: "Concurrent Programming without Locks," *ACM Trans. on Computer Systems*, vol. 25, pp. 1-61, May 2007.
- FRIEDRICH, R., and ROJIA, J.: "Next Generation Data Centers: Trends and Implications," *Proc. 6th Int'l Workshop on Software and Performance*, ACM, pp. 1-2, 2007.
- FUSCO, J.: *The Linux Programmer's Toolbox*, Upper Saddle River, NJ: Prentice Hall, 2007.
- GAL, E., and TOLEDO, S.: "A Transactional Flash File System for Microcontrollers," *Proc. Annual Tech. Conf.*, USENIX, pp. 89-104, 2005.
- GANAPATHY, V., BALAKRISHNAN, A., SWIFT, M.M., and JHA, S.: "Microdrivers: a New Architecture for Device Drivers," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 85-90, 2007.

- GANESH, L., WEATHERSPOON, H., BALAKRISHNAN, M., and BIRMAN, K.: "Optimizing Power Consumption in Large-Scale Storage Systems," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 49-54, 2007.
- GARTINKEL, T., ADAMS, K., WARFIELD, A., and FRANKLIN, J.: "Compatibility is Not Transparency: VMM Detection Myths and Realities," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 31-36, 2007.
- GARTINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., BONEH, D.: "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 193-206, 2003.
- GAW, S., and FELTEN, E.W.: "Password Management Strategies for Online Accounts," *Proc. Second Symp. on Usable Privacy*, ACM, pp. 44-55, 2006.
- GEER, D.: "For Programmers, Multicore Chips Mean Multiple Challenges," *IEEE Computer*, vol. 40, pp. 17-19, Sept. 2007.
- GEIST, R., and DANIEL, S.: "A Continuum of Disk Scheduling Algorithms," *ACM Trans. on Computer Systems*, vol. 5, pp. 77-92, Feb. 1987.
- GELENIER, D.: "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems*, vol. 7, pp. 80-112, Jan. 1985.
- GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T.: "The Google File System," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 29-43, 2003.
- GLEESON, B., PICOVICI, D., SKEHILL, R., and NELSON, J.: "Exploring Power Saving in 802.11 VoIP Wireless Links," *Proc. 2006 Int'l Conf. on Commun. and Mobile Computing*, ACM, pp. 779-784, 2006.
- GNADY, C., BUTT, A.R., and HU, Y.C.: "Program-Counter Based Pattern Classification in Buffer Caching," *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 395-408, 2004.
- GONG, L.: *Inside Java 2 Platform Security*, Reading, MA: Addison-Wesley, 1999.
- GRAHAM, R.: "Use of High-Level Languages for System Programming," *Project MAC Report TM-13*, M.I.T., Sept. 1970.
- GREENAN, K.M., and MILLER, E.L.: "Reliability Mechanisms for File Systems using Non-Volatile Memory as a Metadata Store," *Proc. Int'l Conf. on Embedded Software*, ACM, pp. 178-187, 2006.
- GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*, Cambridge, MA: M.I.T. Press, 1994.
- GROSSMAN, D., and SILVERMAN, H.: "Placement of Records on a Secondary Storage Device to Minimize Access Time," *Journal of the ACM*, vol. 20, pp. 429-438, 1973.
- GUMMADI, K.P., DUNN, R.J., SARIOU, S., GRIBBLE, S., LEVY, H.M., and ZAHORJAN, J.: "Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload," *Proc. 19th Symp. on Operating Systems Principles*, 2003.
- GURUMURTHI, S.: "Should Disks Be Speed Demons or Brainiacs?" *ACM SIGOPS Operating Systems Rev.*, vol. 41, pp. 33-36, Jan. 2007.
- GURUMURTHI, S., SIVASUBRAMANIAN, A., KANDEMIR, M., and FRANKE, H.: "Reducing Disk Power Consumption in Servers with DRPM," *Computer*, vol. 36, pp. 59-66, Dec. 2003.
- HACKETT, B., DAS, M., WANG, D., and YANG, Z.: "Modular Checking for Buffer Overflows in the Large," *Proc. 28th Int'l Conf. on Software Engineering*, ACM, pp. 232-241, 2006.
- HAND, S.M.: "Self-Paging in the Nemesis Operating System," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 73-86, 1999.
- HAND, S.M., WARFIELD, A., FRASER, K., KOITSOVINOS, E., and MAGENHEIMER, D.: "Are Virtual Machine Monitors Microkernels Done Right?," *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 1-6, 2005.
- HAERTIG, H., HOHMUTH, M., LIETKE, J., and SCHONBERG, S.: "The Performance of Kernel-Based Systems," *Proc. 16th Symp. on Operating Systems Principles*, ACM, pp. 66-77, 1997.
- HAFNER, K., and MARKOFF, J.: *Cyberpunk*, New York: Simon and Schuster, 1991.
- HALDERMAN, J.A., and FELTEN, E.W.: "Lessons from the Sony CD DRM Episode," *Proc. 15th USENIX Security Symp.*, USENIX, pp. 77-92, 2006.
- HARI, K., MAYRON, L., CRISTODOULOU, L., MARQUES, O., and FURHT, B.: "Design and Evaluation of 3D Video System Based on H.264 View Coding," *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- HARSEN, J.J., and PEARLMAN, W.A.: "Capacity of Steganographic Channels," *Proc. 7th Workshop on Multimedia and Security*, ACM, pp. 11-24, 2005.
- HARRISON, M.A., RUZZO, W.L., and ULLMAN, J.D.: "Protection in Operating Systems," *Commun. of the ACM*, vol. 19, pp. 461-471, Aug. 1976.
- HART, J.M.: *Win32 System Programming*, Reading, MA: Addison-Wesley, 1997.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., and WEISER, M.: "Using Threads in Interactive Systems: A Case Study," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 94-105, 1993.
- HAVENDER, J.W.: "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, pp. 74-84, 1968.
- HEISER, G., UHLIG, V., and LEVASSEUR, J.: "Are Virtual Machine Monitors Microkernels Done Right?" *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 95-99, 2006.
- HENCHIRI, O., and JAPKOWICZ, N.: "A Feature Selection and Evaluation Scheme for Computer Virus Detection," *Proc. Sixth Int'l Conf. on Data Mining IEEE*, pp. 891-895, 2006.
- HERDER, J.N., BOS, H., GRAS, B., HOMBURG, P., and TANENBAUM, A.S.: "Construction of a Highly Dependable Operating System," *Proc. Sixth European Dependable Computing Conf.*, pp. 3-12, 2006.
- HICKS, B., RUEDA, S., JAEGER, T., and MCDANIEL, P.: "From Trusted to Secure: Building and Executing Applications That Enforce System Security," *Proc. Annual Tech. Conf.*, USENIX, pp. 205-218, 2007.

- HIGHAM, L., JACKSON, L., and KAWASH, J.: "Specifying Memory Consistency of Write Buffer Multiprocessors." *ACM Trans. on Computer Systems*, vol. 25, Art. 1, Feb. 2007.
- HILDEBRAND, D.: "An Architectural Overview of QNX." *Proc. Workshop on Microkernels and Other Kernel Arch.*, ACM, pp. 113–136, 1992.
- HIPSON, P.D.: *Mastering Windows 2000 Registry*, Alameda, CA: Sybex, 2000.
- HOARE, C.A.R.: "Monitors, An Operating System Structuring Concept." *Commun. of the ACM*, vol. 17, pp. 549–557, Oct. 1974; Erratum in *Commun. of the ACM*, vol. 18, p. 95, Feb. 1975.
- HOHMUTH, M., and HAERTIG, H.: "Pragmatic Nonblocking Synchronization in Real-Time Systems." *Proc. Annual Tech. Conf., USENIX*, pp. 217–230, 2001.
- HOHMUTH, M., PETER, M., HAERTIG, H., and SHAPIRO, J.: "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors." *Proc. 11th ACM SIGOPS European Workshop*, ACM, Art. 22, 2004.
- HOLT, R.C.: "Some Deadlock Properties of Computer Systems." *Computing Surveys*, vol. 4, pp. 179–196, Sept. 1972.
- HOM, J., and KREMER, U.: "Energy Management of Virtual Memory on Diskless Devices." *Compilers and Operating Systems for Low Power*, Norwell, MA: Kluwer, pp. 95–113, 2003.
- HOWARD, J.H., KAZAR, M.J., MENEES, S.G., NICHOLS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N., and WEST, M.J.: "Scale and Performance in a Distributed File System." *ACM Trans. on Computer Systems*, vol. 6, pp. 55–81, Feb. 1988.
- HOWARD, M., and LEBLANC, D.: *Writing Secure Code for Windows Vista*, Redmond, WA: Microsoft Press, 2006.
- HUANG, W., LIU, J., KOOP, M., ABALI, B., and PANDA, D.: QNOMAD: Migrating OS-Bypass Networks in Virtual Machines." *Proc. ACM/USENIX Int'l Conf. on Virtual Execution Environments*, ACM, pp. 158–168, 2007.
- HUANG, Z., SUN, C., PURVIS, M., and CRANFIELD, S.: "View-Based Consistency and False Sharing Effect in Distributed Shared Memory." *ACM SIGOPS Operating System Rev.*, vol. 35, pp. 51–60, April 2001.
- HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., and O'MALLEY, S.: "Logical vs. Physical File System Backup." *Proc. Third Symp. on Oper. Systems Design and Impl., USENIX*, pp. 239–249, 1999.
- IEEE: *Information Technology—Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, 1990.
- IN, J., SHIN, I., and KIM, H.: "Memory Systems: SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory." *Proc. 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools*, ACM, pp. 217–226, 2007.
- ISLOOR, S.S., and MARSLAND, T.A.: "The Deadlock Problem: An Overview." *Computer*, vol. 13, pp. 58–78, Sept. 1980.
- IVENS, K.: *Optimizing the Windows Registry*, Foster City, CA: IDG Books Worldwide, 1998.
- JAEGER, T., SAILER, R., and SREENIVASAN, Y.: "Managing the Risk of Covert Information Flows in Virtual Machine Systems." *Proc. 12th ACM Symp. on Access Control Models and Technologies*, ACM, pp. 81–90, 2007.
- JAYASIMHA, D.N., SCHWIEBERT, L., MANIVANNAN, and MAY, J.A.: "A Foundation for Designing Deadlock-Free Routing Algorithms in Wormhole Networks." *J. of the ACM*, vol. 50, pp. 250–275, 2003.
- JIANG, X., and XU, D.: "Profiling Self-Propagating Worms via Behavioral Footprinting." *Proc. 4th ACM Workshop in Recurring Malcode*, ACM, pp. 17–24, 2006.
- JOHNSON, N.F., and JAJODIA, S.: "Exploring Steganography: Seeing the Unseen." *Computer*, vol. 31, pp. 26–34, Feb. 1998.
- JONES, J.R.: "Estimating Software Vulnerabilities." *IEEE Security and Privacy*, vol. 5, pp. 28–32, July/Aug. 2007.
- JOO, Y., CHOI, Y., PARK, C., CHUNG, S., and CHUNG, E.: "System-level optimization: Demand paging for OneNAND Flash eXecute-in-place." *Proc. Int'l Conf. on Hardware Software Codesign, A ACM*, pp. 229–234, 2006.
- KABAY, M.: "Flashes from the Past." *Information Security*, p. 17, 1997.
- KAMINSKY, D.: "Explorations in Namespace: White-Hat Hacking across the Domain Name System." *Commun. of the ACM*, vol. 49, pp. 62–69, June 2006.
- KAMINSKY, M., DAVVIDES, G., MAZIERES, D., and KAASHOEK, M.F.: "Decentralized User Authentication in a Global File System." *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 60–73, 2003.
- KANG, S., WON, Y., and ROH, S.: "Harmonic Interleaving: File System Support for Scalable Streaming of Layer Encoded Objects." *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- KANT, K., and MOHAPATRA, P.: "Internet Data Centers." *Computer*, vol. 27, pp. 35–37, Nov. 2004.
- KARLIN, A.R., LI, K., MANASSE, M.S., and OWICKI, S.: "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor." *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 41–54, 1991.
- KARLIN, A.R., MANASSE, M.S., MCGEOCH, L., and OWICKI, S.: "Competitive Randomized Algorithms for Non-Uniform Problems." *Proc. First Annual ACM Symp. on Discrete Algorithms*, ACM, pp. 301–309, 1989.
- KAROL, M., GOLESTANI, S.J., and LEE, D.: "Prevention of Deadlocks and LiveLocks in Lossless Backpressed Packet Networks." *IEEE/ACM Trans. on Networking*, vol. 11, pp. 923–934, 2003.
- KAUFMAN, C., PERLMAN, R., and SPECINER, M.: *Network Security*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2002.
- KEETON, K., BEYER, D., BRAU, E., MERCHANT, A., SANTOS, C., and ZHANG, A.: "On the Road to Recovery: Restoring Data After Disasters." *Proc. Eurosys 2006*, ACM, pp. 235–238, 2006.

- KELEHER, P., COX, A., DWARKADAS, S., and ZWAENEPOEL, W.: "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. USENIX Winter 1994 Conf.*, USENIX, pp. 115-132, 1994.
- KERNIGHAN, B.W., and PIKE, R.: *The UNIX Programming Environment*, Upper Saddle River, NJ: Prentice Hall, 1984.
- KIENZLE, D.M., and ELDER, M.C.: "Recent Worms: A Survey and Trends," *Proc. 2003 ACM Workshop on Rapid Malcode*, ACM, pp. 1-10, 2003.
- KIM, J., BARATTO, R.A., and NIEH, J.: "pTHINC: A Thin-Client Architecture for Mobile Wireless Web," *Proc. 15th Int'l Conf. on the World Wide Web*, ACM, pp. 143-152, 2006.
- KING, S.T., and CHEN, P.M.: "Backtracking Intrusions," *ACM Trans. on Computer Systems*, vol. 23, pp. 51-76, Feb. 2005.
- KING, S.T., DUNLAP, G.W., and CHEN, P.M.: "Operating System Support for Virtual Machines," *Proc. Annual Tech. Conf.*, USENIX, pp. 71-84, 2003.
- KING, S.T., DUNLAP, G.W., and CHEN, P.M.: "Debugging Operating Systems with Time-Traveling Virtual Machines," *Proc. Annual Tech. Conf.*, USENIX, pp. 1-15, 2005.
- KIRSCH, C.M., SANVIDO, M.A.A., and HENZINGER, T.A.: "A Programmable Microkernel for Real-Time Systems," *Proc. 1st Int'l Conf. on Virtual Execution Environments*, ACM, pp. 35-45, 2005.
- KISSLER, S., and HOYT, O.: "Using Thin Client Technology to Reduce Complexity and Cost," *Proc. 33rd Annual Conf. on User Services*, ACM, pp. 138-140, 2005.
- KLEIMAN, S.R.: "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proc. USENIX Summer 1986 Conf.*, USENIX, pp. 238-247, 1986.
- KLEIN, D.V.: "Fooling the Cracker: A Survey of, and Improvements to, Password Security," *Proc. UNIX Security Workshop II*, USENIX, Summer 1990.
- KNUTH, D.E.: *The Art of Computer Programming. Vol. 1: Fundamental Algorithms*, 3rd Ed., Reading, MA: Addison-Wesley, 1997.
- KOCHAN, S.G., and WOOD, P.H.: *UNIX Shell Programming*, Indianapolis: IN, 2003.
- KONTOTHANASSIS, L., STETS, R., HUNT, H., RENCUZOGULLARI, U., ALTEKAR, G., DWARKADAS, S., and SCOTT, M.L.: "Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks," *ACM Trans. on Computer Systems*, vol. 23, pp. 301-335, Aug. 2005.
- KOTLA, R., ALVISEI, L., and DAHLIN, M.: "SafeStore: A Durable and Practical Storage System," *Proc. Annual Tech. Conf.*, USENIX, pp. 129-142, 2007.
- KRATZER, C., DITTMANN, J., LANG, A., and KUHN, T.: "WLAN 'Steganography: A First Practical Review," *Proc. Eighth Workshop on Multimedia and Security*, ACM, pp. 17-22, 2006.
- KRAVETS, R., and KRISHNAN, P.: "Power Management Techniques for Mobile Communication," *Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, pp. 157-168, 1998.
- KRIEGER, O., AUSLANDER, M., ROSENBERG, B., WISNIEWSKI, R.W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A., and UHLIG, V.: "K42: Building a Complete Operating System," *Proc. Eurosys 2006*, ACM, pp. 133-145, 2006.
- KRISHNAN, R.: "Timeshared Video-on-Demand: A Workable Solution," *IEEE Multimedia*, vol. 6, Jan-March 1999, pp. 77-79.
- KROEGER, T.M., and LONG, D.D.E.: "Design and Implementation of a Predictive File Prefetching Algorithm," *Proc. Annual Tech. Conf.*, USENIX, pp. 105-118, 2001.
- KRUEGEL, C., ROBERTSON, W., and VIGNA, G.: "Detecting Kernel-Level Rootkits Through Binary Analysis," *Proc. First IEEE Int'l Workshop on Critical Infrastructure Protection*, IEEE, pp. 13-21, 2004.
- KRUEGER, P., LAI, T.-H., and DIXIT-RADIYA, V.A.: "Job Scheduling is More Important Than Processor Allocation for Hypercube Computers," *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, pp. 488-497, May 1994.
- KUM, S.-U., and MAYER-PATEL, K.: "Intra-Stream Encoding for Multiple Depth Streams," *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- KUMAR, R., TULSEN, D.M., JOUPPI, N.P., and RANGANATHAN, P.: "Heterogeneous Chip Multiprocessors," *Computer*, vol. 38, pp. 32-38, Nov. 2005.
- KUMAR, V.P., and REDDY, S.M.: "Augmented Shuffle-Exchange Multistage Interconnection Networks," *Computer*, vol. 20, pp. 30-40, June 1987.
- KUPERMAN, B.A., BRODLEY, C.E., OZDOGANOLU, H., VIJAYKUMAR, T.N., and JALOTE, A.: "Detection and Prevention of Stack Buffer Overflow Attacks," *Commun. of the ACM*, vol. 48, pp. 50-56, Nov. 2005.
- KWOK, Y.-K., AHMAD, I.: "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *Computing Surveys*, vol. 31, pp. 406-471, Dec. 1999.
- LAI, A.M., and NIEH, J.: "On the Performance of Wide-Area Thin-Client Computing," *ACM Trans. on Computer Systems*, vol. 24, pp. 175-209, May 2006.
- LAMPOR, L.: "Password Authentication with Insecure Communication," *Commun. of the ACM*, vol. 24, pp. 770-772, Nov. 1981.
- LAMPSON, B.W.: "A Scheduling Philosophy for Multiprogramming Systems," *Commun. of the ACM*, vol. 11, pp. 347-360, May 1968.
- LAMPSON, B.W.: "A Note on the Confinement Problem," *Commun. of the ACM*, vol. 10, pp. 613-615, Oct. 1973.
- LAMPSON, B.W.: "Hints for Computer System Design," *IEEE Software*, vol. 1, pp. 11-28, Jan. 1984.
- LAMPSON, B.W., and STURGIS, H.E.: "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center Technical Report, June 1979.
- LANDWEHR, C.E.: "Formal Models of Computer Security," *Computing Surveys*, vol. 13, pp. 247-278, Sept. 1981.

- LE, W., and SOFFA, M.J.: "Refining Buffer Overflow Detection via Demand-Driven Path-Sensitive Analysis," *Proc. 7th ACM SIGPLAN-SOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM, pp. 63-68, 2007.
- LEE, J.Y.B.: "Parallel Video Servers: A Tutorial," *IEEE Multimedia*, vol. 5, pp. 20-28, April-June 1998.
- LESIELE, L., MAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., and HYDEN, E.: "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE J. on Selected Areas in Commun.*, vol. 14, pp. 1280-1297, July 1996.
- LEVASSEUR, J., UHLIG, V., STOFSS, J., and GOTZ, S.: "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *Proc. Sixth Symp. on Operating System Design and Implementation*, USENIX, pp. 17-30, 2004.
- LEVIN, R., COHEN, E.S., CORWIN, W.M., POLLACK, F.J., and WULF, W.A.: "Policy/Mechanism Separation in Hydra," *Proc. Fifth Symp. on Operating Systems Principles*, ACM, pp. 132-140, 1975.
- LEVINE, G.N.: "Defining Deadlock," *ACM SIGOPS Operating Systems Rev.*, vol. 37, pp. 54-64, Jan. 2003a.
- LEVINE, G.N.: "Defining Deadlock with Fungible Resources," *ACM SIGOPS Operating Systems Rev.*, vol. 37, pp. 5-11, July 2003b.
- LEVINE, G.N.: "The Classification of Deadlock Prevention and Avoidance Is Erroneous," *ACM SIGOPS Operating Systems Rev.*, vol. 39, pp. 47-50, April 2005.
- LEVINE, J.G., GRIZZARD, J.B., and OWEN, H.L.: "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security and Privacy*, vol. 4, pp. 24-32, Jan./Feb. 2006.
- LI, T., K.: "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. Thesis, Yale Univ., 1986.
- LI, K., and HUDAK, P.: "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. on Computer Systems*, vol. 7, pp. 321-359, Nov. 1989.
- LI, K., KUMPF, R., HORTON, P., and ANDERSON, T.: "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," *Proc. 1994 Winter Conf.*, USENIX, pp. 279-291, 1994.
- LI, T., ELLIS, C.S., LEBECK, A.R., and SORIN, D.J.: "Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution," *Proc. Annual Tech. Conf.*, USENIX, pp. 31-44, 2005.
- LIE, D., THEKKATH, C.A., and HOROWITZ, M.: Implementing an Untrusted Operating System on Trusted Hardware," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 178-192, 2003.
- LIEDTKE, J.: "Improving IPC by Kernel Design," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 175-188, 1993.
- LIEDTKE, J.: "On Micro-Kernel Construction," *Proc. 15th Symp. on Operating Systems Principles*, ACM, pp. 237-250, 1995.
- LIEDTKE, J.: "Toward Real Microkernels," *Commun. of the ACM*, vol. 39, pp. 70-77, Sept. 1996.
- LIN, G., and RAJARAMAN, R.: "Approximation Algorithms for Multiprocessor Scheduling under Uncertainty," *Proc. 19th Symp. on Parallel Algorithms and Arch.*, ACM, pp. 25-34, 2007.
- LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose, CA: Peer-to-Peer Communications, 1996.
- LIU, C.L., and LAYLAND, J.W.: "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. of the ACM*, vol. 20, pp. 46-61, Jan. 1973.
- LIU, J., HUANG, W., ABALL, B., and PANDA, B.K.: "High Performance VMM-Bypass I/O in Virtual Machines," *Proc. Annual Tech. Conf.*, USENIX, pp. 29-42, 2006.
- LO, V.M.: "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, pp. 30-39, 1984.
- LORCH, J.R., and SMITH, A.J.: "Reducing Processor Power Consumption by Improving Processor Time Management in a Single-User Operating System," *Proc. Second Int'l Conf. on Mobile Computing and Networking*, ACM, pp. 143-154, 1996.
- LORCH, J.R., and SMITH, A.J.: "Apple Macintosh's Energy Consumption," *IEEE Micro*, vol. 18, pp. 54-63, Nov./Dec. 1998.
- LU, P., and SHEN, K.: "Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache," *Proc. Annual Tech. Conf.*, USENIX, pp. 29-43, 2007.
- LUDWIG, M.A.: *The Giant Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 1998.
- LUDWIG, M.A.: *The Little Black Book of Email Viruses*, Show Low, AZ: American Eagle Publications, 2002.
- LUND, K., and GOEBEL, V.: "Adaptive Disk Scheduling in a Multimedia DBMS," *Proc. 11th ACM Int'l Conf. on Multimedia*, ACM, pp. 65-74, 2003.
- LYDA, R., and HAMROCK, J.: "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security and Privacy*, vol. 5, pp. 17-25, March/April 2007.
- MANIATIS, P., ROUSSOPOULOS, M., GIULLI, T.J., ROSENTHAL, D.S.H., and BAKER, M.: "The LOCS Peer-to-Peer Digital Preservation System," *ACM Trans. on Computer Systems*, vol. 23, pp. 2-50, Feb. 2005.
- MARKOWITZ, J.A.: "Voice Biometrics," *Commun. of the ACM*, vol. 43, pp. 66-73, Sept. 2000.
- MARSH, B.D., SCOTT, M.L., LEBLANC, T.J., and MARKATOS, E.P.: "First-Class User-Level Threads," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 110-121, 1991.
- MATTHUR, A., and MUNDUR, P.: "Dynamic Load Balancing Across Mirrored Multimedia Servers," *Proc. 2003 Int'l Conf. on Multimedia*, IEEE, pp. 53-56, 2003.
- MAXWELL, S.E.: *Linux Core Kernel Commentary*, 2nd ed., Scottsdale, AZ: Coriolis, 2001.

- MCDANIEL, T.: "Magneto-Optical Data Storage," *Commun. of the ACM*, vol. 43, pp. 57-63, Nov. 2000.
- McKUSICK, M.J., JOY, W.N., LEFFLER, S.J., and FABRY, R.S.: "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, vol. 2, pp. 181-197, Aug. 1984.
- McKUSICK, M.K., and NEVILLE-NEIL, G.V.: *The Design and Implementation of the FreeBSD Operating System*, Reading, MA: Addison-Wesley, 2004.
- MEAD, N.R.: "Who Is Liable for Insecure Systems?" *Computer*, vol. 37, pp. 27-34, July 2004.
- MEDINETS, D.: *UNIX Shell Programming Tools*, New York: McGraw-Hill, 1999.
- MELLOR-CRUMMEY, J.M., and SCOTT, M.I.: "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21-65, Feb. 1991.
- MENON, A., COX, A., and ZWANEPOEL, W.: "Optimizing Network Virtualization in Xen," *Proc. Annual Tech. Conf.*, USENIX, pp. 15-28, 2006.
- MILJOJIC, D.: "Operating Systems: Now and in the Future," *IEEE Concurrency*, vol. 7, pp. 12-21, Jan.-March 1999.
- MILJOJIC, D.: "Security and Privacy," *IEEE Concurrency*, vol. 8, pp. 70-79, April-June 2000.
- MIN, H., YI, S., CHO, Y., and HONG, J.: "An Efficient Dynamic Memory Allocator for Sensor Operating Systems," *Proc. 2007 ACM Symposium on Applied Computing*, ACM, pp. 1159-1164, 2007.
- MOFFIE, M., CHENG, W., KAEI, D., and ZHAO, Q.: "Hunting Trojan Horses," *Proc. First Workshop on Arch. and System Support for Improving Software Dependability*, ACM, pp. 12-17, 2006.
- MOODY, G.: *Rebel Code*, Cambridge, MA: Perseus Publishing, 2001.
- MOORE, J., CHASE, J., RANGANATHAN, P., and SHARMA, R.: "Making Scheduling 'Cool': Temperature-Aware Workload Placement in Data Centers," *Proc. Annual Tech. Conf.*, USENIX, pp. 61-75, 2005.
- MORRIS, B.: *The Symbian OS Architecture Sourcebook*, Chichester, UK: John Wiley, 2007.
- MORRIS, J.H., SATYANARAYANAN, M., CONNER, M.H., HOWARD, J.H., ROSENTHAL, D.S., and SMITH, F.D.: "Andrew: A Distributed Personal Computing Environment," *Commun. of the ACM*, vol. 29, pp. 184-201, March 1986.
- MORRIS, R., and THOMPSON, K.: "Password Security: A Case History," *Commun. of the ACM*, vol. 22, pp. 594-597, Nov. 1979.
- MOSHCHUK, A., BRAGIN, T., GRIBBLE, S.D., and LEVY, H.M.: "A Crawler-Based Study of Spyware on the Web," *Proc. Network and Distributed System Security Symp.*, Internet Society, pp. 1-17, 2006.
- MULLENDER, S.J., and TANENBAUM, A.S.: "Immediate Files," *Software Practice and Experience*, vol. 14, pp. 365-368, 1984.
- MUNISWARMY-REDDY, K.-K., HOLLAND, D.A., BRAUN, U., and SELTZER, M.: "Provenance-Aware Storage Systems," *Proc. Annual Tech. Conf.*, USENIX, pp. 43-56, 2006.
- MUTHITACHAROEN, A., CHEN, B., and MAZIERES, D.: "A Low-Bandwidth Network File System," *Proc. 18th Symp. on Operating Systems Principles*, ACM, pp. 174-187, 2001.
- MUTHITACHAROEN, A., MORRIS, R., GIL, T.M., and CHEN, B.: "Ivy: A Read/Write Peer-to-Peer File System," *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 31-44, 2002.
- NACHENBERG, C.: "Computer Virus-Antivirus Coevolution" *Commun. of the ACM*, vol. 40, pp. 46-51, Jan. 1997.
- NEMETH, E., SNYDER, G., SEEBASS, S., and HEIN, T.R.: *UNIX System Administration Handbook*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2000.
- NEWHAM, C., and ROSENBLATT, B.: *Learning the Bash Shell*, Sebastopol, CA: O'Reilly & Associates, 1998.
- NEWTON, G.: "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *ACM SIGOPS Operating Systems Rev.*, vol. 13, pp. 33-44, April 1979.
- NIEH, J., and LAM, M.S.: "A SMART Scheduler for Multimedia Applications," *ACM Trans. on Computer Systems*, vol. 21, pp. 117-163, May 2003.
- NIEH, J., VAILL, C., and ZHONG, H.: "Virtual-Time Round Robin: An O(1) Proportional Share Scheduler," *Proc. Annual Tech. Conf.*, USENIX, pp. 245-259, 2001.
- NIGHTINGALE, E.B., and FLINN, J.: "Energy-Efficiency and Storage Flexibility in the Blue File System," *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 363-378, 2004.
- NIKOLOPOULOS, D.S., AVGADE, E., PAPATHEODOROU, T.S., POLYCHRONOPOULOS, C.D., and LABARTA, J.: "The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms," *Proc. Int'l Conf. on Supercomputing*, ACM, pp. 23-37, 2001.
- NIST (National Institute of Standards and Technology): FIPS Pub. 180-1, 1995.
- OKI, B., PFLUEGL, M., SIEGEL, A., and SKEEN, D.: "The Information Bus—An Architecture for Extensible Distributed Systems," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 58-68, 1993.
- ONEY, W.: *Programming the Microsoft Windows Driver Model*, 2nd ed., Redmond, WA: Microsoft Press, 2002.
- ORGANICK, E.I.: *The Multics System*, Cambridge, MA: M.I.T. Press, 1972.
- ORWICK, P., and SMITH, G.: *Developing Drivers with the Windows Driver Foundation*, Redmond, WA: Microsoft Press, 2007.
- OSTRAND, T.J., and WEYUKER, E.J.: "The Distribution of Faults in a Large Industrial Software System," *Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, ACM, pp. 55-64, 2002.

- OSTERHOUT, J.K.: "Scheduling Techniques for Concurrent Systems," *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, pp. 22-30, 1982.
- PADIOLEAU, Y., LAWALL, J.L., and MULLER, G.: "Understanding Collateral Evolution in Linux Device Drivers," *Proc. Eurosys 2006*, ACM, pp. 59-72, 2006.
- PADIOLEAU, Y., and RIDOUX, O.: "A Logic File System," *Proc. Annual Tech. Conf.*, USENIX, pp. 99-112, 2003.
- PAI, V.S., DRUSCHEL, P., and ZWAENEPOEL, W.: "IO-Lite: A Unified I/O Buffering and Caching System," *ACM Trans. on Computer Systems*, vol. 18, pp. 37-66, Feb. 2000.
- PANAGIOTOU, K., and SOUZA, A.: "On Adequate Performance Measures for Paging," *Proc. 38th ACM Symp. on Theory of Computing*, ACM, pp. 487-496, 2006.
- PANKANTI, S., BOLLE, R.M., and JAIN, A.: "Biometrics: The Future of Identification," *Computer*, vol. 33, pp. 46-49, Feb. 2000.
- PARK, C., KANG, J.-U., PARK, S.-Y., KIM, J.-S.: "Energy Efficient Architectural Techniques: Energy-Aware Demand Paging on NAND Flash-Based Embedded Storages," *ACM*, pp. 338-343, 2004b.
- PARK, C., LIM, J., KWON, K., LEE, J., and MIN, S.: "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory," *Proc. 4th ACM Int'l Cong. on Embedded Software, September*, ACM, pp. 114-124, 2004a.
- PARK, S., JIANG, W., ZHOU, Y., and ADVE, S.: "Managing Energy-Performance Trade-offs for Multithreaded Applications on Multiprocessor Architectures," *Proc. 2007 Int'l Conf. on Measurement and Modeling of Computer Systems*, ACM, pp. 169-180, 2007.
- PARK, S., and OHM, S.-Y.: "Real-Time FAT File System for Mobile Multimedia Devices," *Proc. Int'l Conf. on Consumer Electronics*, IEEE, pp. 245-346, 2006.
- PATE, S.D.: *UNIX Filesystems: Evolution, Design, and Implementation*, New York: Wiley, 2003.
- PATTERSON, D., and HENNESSY, J.: *Computer Organization and Design*, 3rd ed., San Francisco: Morgan Kaufman, 2004.
- PATTERSON, D.A., GIBSON, G., and KATZ, R.: "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, ACM, pp. 109-166, 1988.
- PAUL, N., GURUMURTHI, S., and EVANS, D.: "Towards Disk-Level Malware Detection," *Proc. First Workshop on Code-based Software Security Assessments*, 2005.
- PEEK, D., NIGHTINGALES, E.B., HIGGINS, B.D., KUMAR, P., and FLINN, J.: "Sprockets: Safe Extensions for Distributed File Systems," *Proc. Annual Tech. Conf.*, USENIX, pp. 115-128, 2007.
- PERMANDIA, P., ROBERTSON, M., and BOYAPATI, C.: "A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language," *Proc. 2007 Conf. on Languages Compilers and Tools*, ACM, p. 10-19, 2007.
- PESERICO, E.: "Online Paging with Arbitrary Associativity," *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, ACM, pp. 555-564, 2003.
- PETERSON, G.L.: "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, pp. 115-116, June 1981.
- PETZOLD, C.: *Programming Windows*, 5th ed., Redmond, WA: Microsoft Press, 1999.
- PFLEGER, C.P., and PFLEGER, S.L.: *Security in Computing*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2006.
- PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., and WINTERBOTTOM, P.: "The Use of Name Spaces in Plan 9," *Proc. 5th ACM SIGOPS European Workshop*, ACM, pp. 1-5, 1992.
- PIZZO, F., and VITEK, J.: "An Empirical Evaluation of Memory Management Alternatives for Real-Time Java," *Proc. 27th IEEE Int'l Real-Time Systems Symp.*, IEEE, pp. 25-46, 2006.
- POPEK, G.J., and GOLDBERG, R.P.: "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. of the ACM*, vol. 17, pp. 412-421, July 1974.
- POPESCU, B.C., CRISPO, B., and TANENBAUM, A.S.: "Secure Data Replication over Untrusted Hosts," *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, 121-127, 2003.
- PORTOKALIDIS, G., and BOS, H.: "SweetBait: Zero-Hour Worm Detection and Containment Using Low- and High-Interaction Honeybots,"
- PORTOKALIDIS, G., SLOWINSKA, A., and BOS, H.: "ARGOS: An Emulator of Fingerprinting Zero-Day Attacks," *Proc. Eurosys 2006*, ACM, pp. 15-27, 2006.
- PRAHAKARAN, V., ARPACI-DUSSEAU, A.C., and ARPACI-DUSSEAU, R.H.: "Analysis and Evolution of Journaling File Systems," *Proc. Annual Tech. Conf.*, USENIX, pp. 105-120, 2005.
- PRASAD, M., and CHIUH, T.: "A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks," *Proc. Annual Tech. Conf.*, USENIX, pp. 211-224, 2003.
- PRECHET, L.: "An Empirical Comparison of Seven Programming Languages," *Computer*, vol. 33, pp. 23-29, Oct. 2000.
- PUSARA, M., and BRODLEY, C.E.: "DMSBC session: User Re-Authentication via Mouse Movements," *Proc. 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, ACM, pp. 1-8, 2004.
- QUYNH, N.A., and TAKEFUI, Y.: "Towards a Tamper-Resistant Kernel Rootkit Detector," *Proc. Symp. on Applied Computing*, ACM, pp. 276-283, 2007.
- RAJAGOPALAN, M., LEWIS, B.T., and ANDERSON, T.A.: "Thread Scheduling for Multicore Platforms," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 7-12, 2007.
- RANGASWAMI, R., DIMITRIJEVIC, Z., CHANG, E., and SCHAUER, K.: "Building MEMS-Storage Systems for Streaming Media," *ACM Trans. on Storage*, vol. 3, Art. 6, June 2007.

- ROZIER, M., ABBROSSIMOV, V., ARMAND, F., BOULE, L., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LEONARD, P., LANGLOIS, S., and NEUHAUSER, W.: "Chorus Distributed Operating Systems," *Computing Systems*, vol. 1, pp. 305-379, Oct. 1988.
- RUBINI, A., KROAH-HARTMAN, G., and CORBET, J.: *Linux Device Drivers*, Sebastopol, CA: O'Reilly & Associates, 2005.
- RUSSINOVICH, M., and SOLOMON, D.: *Microsoft Windows Internals*, 4th ed. Redmond, WA: Microsoft Press, 2005.
- RYCROFT, M.E.: "No One Needs It (Until They Need It): Implementing A New Desktop Backup Solutions," *Proc. 34th Annual SIGUCCS Conf. on User Services*, ACM, pp. 347-352, 2006.
- SACKMAN, H., ERIKSON, W.J., and GRANT, E.E.: "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Commun. of the ACM*, vol. 11, pp. 3-11, Jan. 1968.
- SAIDI, H.: "Guarded Models for Intrusion Detection," *Proc. 2007 Workshop on Programming Languages and Analysis for Security*, ACM, pp. 85-94, 2007.
- SAITO, Y., KARAMANOLIS, C., KARLSSON, M., and MAHALINGAM, M.: "Taming Aggressive Replication in the Pangea Wide-Area File System," *Proc. Fifth Symp. on Operating System Design and Implementation*, USENIX, pp. 15-30, 2002.
- SALTZER, J.H.: "Protection and Control of Information Sharing in MULTICS," *Commun. of the ACM*, vol. 17, pp. 388-402, July 1974.
- SALTZER, J.H., REED, D.P., and CLARK, D.D.: "End-to-End Arguments in System Design," *ACM Trans. on Computer Systems*, vol. 2, pp. 277-277, Nov. 1984.
- SALTZER, J.H., and SCHROEDER, M.D.: "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, pp. 1278-1308, Sept. 1975.
- SALUS, P.H.: "UNIX At 25," *Byte*, vol. 19, pp. 75-82, Oct. 1994.
- SANOK, D.J.: "An Analysis of how Antivirus Methodologies Are Utilized in Protecting Computers from Malicious Code," *Proc. Second Annual Conf. on Information Security Curriculum Development*, ACM, pp. 142-144, 2005.
- SARHAN, N.J., and DAS, C.R.: "Caching and Scheduling in NAD-Based Multimedia Servers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, pp. 921-933, Oct. 2004.
- SASSE, M.A.: "Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems," *IEEE Security and Privacy*, vol. 5, pp. 78-81, May/June 2007.
- SCHAFER, M.K.F., HOLLSTEIN, T., ZIMMER, H., and GLESNER, M.: "Deadlock-Free Routing and Component Placement for Irregular Mesh-Based Networks-on-Chip," *Proc. 2005 Int'l Conf. on Computer-Aided Design*, IEEE, pp. 238-245, 2005.
- SCHEIBLE, J.P.: "A Survey of Storage Options" *Computer*, vol. 35, pp. 42-46, Dec. 2002.
- SCHWARTZ, A., and GUERRAZZI, C.: "You Can Never Be Too Thin: Skinny-Client Technology," *Proc. 33rd Annual Conf. on User Services*, ACM, pp. 336-337, 2005.
- SCOTT, M., LEBLANC, T., and MARSH, B.: "Multi-model Parallel Programming in Psyche," *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pp. 70-78, 1990.
- SEAWRIGHT, L.H., and MACKINNON, R.A.: "VM/370—A Study of Multiplicity and Usefulness," *IBM Systems J.*, vol. 18, pp. 4-17, 1979.
- SHAH, M., BAKER, M., MOGUL, J.C., and SWAMINATHAN, R.: "Auditing to Keep Online Storage Services Honest," *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 61-66, 2007.
- SHAH, S., SOULES, C.A.N., GANGER, G.R., and NOBLE, B.N.: "Using Provenance to Aid in Personal File Search," *Proc. Annual Tech. Conf.*, USENIX, pp. 171-184, 2007.
- SHENOY, P.J., and VIN, H.M.: "Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers," *Perf. Eval. J.*, vol. 38, pp. 175-199, 1999.
- SHUB, C.M.: "A Unified Treatment of Deadlock," *J. of Computing Sciences in Colleges*, vol. 19, pp. 194-204, Oct. 2003.
- SILBERSCHATZ, A., GALVIN, P.B., and GAGNE, G.: *Operating System Concepts with Java*, 7th ed. New York: Wiley, 2007.
- SIMON, R.J.: *Windows NT Win32 API SuperBible*, Corte Madera, CA: Sams Publishing, 1997.
- SITARAM, D., and DAN, A.: *Multimedia Servers*, San Francisco: Morgan Kaufman, 2000.
- SMITH, D.K., and ALEXANDER, R.C.: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*, New York: William Morrow, 1988.
- SNIR, M., OTTO, S.W., HUSS-LEDERMAN, S., WALKER, D.W., and DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge, MA: M.I.T. Press, 1996.
- SON, S.W., CHEN, G., and KANDEMIR, M.: "A Compiler-Guided Approach for Reducing Disk Power Consumption by Exploiting Disk Access Locality," *Proc. Int'l Symp. on Code Generation and Optimization*, IEEE, pp. 256-268, 2006.
- SPAFFORD, E., HEAPHY, K., and FERBRACHE, D.: *Computer Viruses*, Arlington, VA: ADAPSO, 1989.
- STALLINGS, W.: *Operating Systems*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2005.
- STAN, M.R., and SKADRON, K.: "Power-Aware Computing," *Computer*, vol. 36, pp. 35-38, Dec. 2003.
- STEIN, C.A., HOWARD, J.H., and SELTZER, M.I.: "Unifying File System Protection," *Proc. Annual Tech. Conf.*, USENIX, pp. 79-90, 2001.
- STEIN, L.: "Stupid File Systems Are Better," *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, pp. 5, 2005.
- STENMETZ, R., and NAHRSTEDT, K.: *Multimedia: Computing, Communications and Applications*, Upper Saddle River, NJ: Prentice Hall, 1995.

- TANENBAUM, A.S., and WOODHULL, A.S.: *Operating Systems: Design and Implementation*, 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2006.
- TANG, Y., and CHEN, S.: "A Automated Signature-Based Approach against Polymorphic Internet Worms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 18, pp. 879–892, July 2007.
- TEORY, T.J.: "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, pp. 1–11, 1972.
- THIBAUDEAU, R.: "Trusted Computing for Disk Drives and Other Peripherals," *IEEE Security and Privacy*, vol. 4, pp. 26–33, Sept/Oct. 2006.
- THOMPSON, K.: "Reflections on Trusting Trust," *Commun. of the ACM*, vol. 27, pp. 761–763, Aug. 1984.
- TOLENTINO, M.E., TURNER, J., and CAMERON, K.W.: "Memory-Miser: A Performance-Constrained Runtime System for Power Scalable Clusters," *Proc. Fourth Int'l Conf. on Computing Frontiers*, ACN, pp. 237–246, 2007.
- TSAFIR, D., ETSION, Y., FEITELSON, D.G., and KIRKPATRICK, S.: "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications," *Proc. 19th Annual Int'l Conf. on Supercomputing*, ACM, pp. 303–312, 2005.
- TUCEK, J., NEWSOME, J., LU, S., HUANG, C., XANTHOS, S., BRUMLEY, D., ZHOU, Y., and SONG, D.: "Sweeper: a Lightweight End-to-End System for Defending Against Fast Worms," *Proc. Eurosys 2007*, ACM, pp. 115–128, 2007.
- TUCKER, A., and GUPTA, A.: "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 12th Symp. on Operating Systems Principles*, ACM, pp. 159–166, 1989.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S., and BROWN, R.: "Design Tradeoffs for Software-Managed TLBs," *ACM Trans. on Computer Systems*, vol. 12, pp. 175–205, Aug. 1994.
- ULUSKI, D., MOFFIE, M., and KAEHL, D.: "Characterizing Antivirus Workload Execution," *ACM SIGARCH Computer Arch. News*, vol. 33, pp. 90–98, March 2005.
- VAHALIA, U.: *UNIX Internals—The New Frontiers*, Upper Saddle River, NJ: Prentice Hall, 2007.
- VAN DOORN, L., HOMBURG, P., and TANENBAUM, A.S.: "Paramecium: An Extensible Object-Based Kernel," *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, pp. 86–89, 1995.
- VAN 'T NOORDE, G., BALOGH, A., HOFMAN, R., BRAZIER, F.M.T., and TANENBAUM, A.S.: "A Secure Jailing System for Confining Untrusted Applications," *Proc. Second Int'l Conf. on Security and Cryptography*, INSTICC, pp. 414–423, 2007.
- VASWANI, R., and ZAHORJAN, J.: "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors," *Proc. 13th Symp. on Operating Systems Principles*, ACM, pp. 26–40, 1991.
- VENKATACHALAM, V., and FRANZ, M.: "Power Reduction Techniques for Microprocessor Systems," *Computing Surveys*, vol. 37, pp. 195–237, Sept. 2005.
- STEVENS, R.W., and RAGO, S.A.: "Advanced Programming in the UNIX Environment," Reading, MA: Addison-Wesley, 2008.
- STICHBURY, J., and JACOBS, M.: *The Accredited Symbian Developer Primer*, Chichester, UK: John Wiley, 2006.
- STIEGLER, M., KARP, A.H., YEE, K.-P., CLOSE, T., and MILLER, M.S.: "Polaris: Virus-Safe Computing for Windows XP," *Commun. of the ACM*, col. 49, pp. 83–88, Sept. 2006.
- STOESS, J., LANG, C., and BELLOSA, F.: "Energy Management for Hypervisor-Based Virtual Machines," *Proc. Annual Tech. Conf.*, USENIX, pp. 1–14, 2007.
- STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*, New York: Doubleday, 1989.
- STONE, H.S., and BOKHARI, S.H.: "Control of Distributed Processes," *Computer*, vol. 11, pp. 97–106, July 1978.
- STOREY, M.W., GREENAN, K.M., MILLER, E.L., and VORUGANTI, K.: "POTSHARDS: Secure Long-Term Storage without Encryption," *Proc. Annual Tech. Conf.*, USENIX, pp. 143–156, 2007.
- SWIFT, M.M., ANNAMALAI, M., BERSHAD, B.N., and LEVY, H.M.: "Recovering Device Drivers," *ACM Trans. on Computer Systems*, vol. 24, pp. 333–360, Nov. 2006.
- TALLURI, M., HILL, M.D., and KHALIDI, Y.A.: "A New Page Table for 64-Bit Address Spaces," *Proc. 15th Symp. on Operating Systems Prin.*, ACM, pp. 184–200, 1995.
- TAM, D., AZIMI, R., and STUMM, M.: "Thread Clustering: Sharing-Aware Scheduling," *Proc. Eurosys 2007*, ACM, pp. 47–58, 2007.
- TAMAI, M., SUN, T., YASUMOTO, K., SHIBATA, N., and ITO, M.: "Energy-Aware Video Streaming with QoS Control for Portable Computing Devices," *Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2004.
- TAN, G., SUN, N., and GAO, G.R.: "A Parallel Dynamic Programming Algorithm on a Multi-Core Architecture," *Proc. 19th ACM Symp. on Parallel Algorithms and Arch.*, ACM, pp. 135–144, 2007.
- TANENBAUM, A.S.: *Computer Networks*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 2003.
- TANENBAUM, A.S.: *Structured Computer Organization*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2006.
- TANENBAUM, A.S., HERDER, J.N., and BOS, H.: "File Size Distribution on UNIX Systems: Then and Now," *ACM SIGOPS Operating Systems Rev.*, vol. 40, pp. 100–104, Jan. 2006.
- TANENBAUM, A.S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G.J., MULLENDER, S.J., JANSEN, J., and VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM*, vol. 33, pp. 46–63, Dec. 1990.
- TANENBAUM, A.S., and VAN STEEN, M.R.: *Distributed Systems*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 2006.

- VILLA, H.: "Liquid Colling: A Next Generation Data Center Strategy," *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, ACM, Art. 287, 2006.
- VINOSKI, S.: "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 35, pp. 46-56, Feb. 1997.
- VISCAROLA, P.G., MASON, T., CARIDDI, M., RYAN, B., and NOONE, S.: *Introduction to the Windows Driver Foundation Kernel-Mode Framework*, Amherst, NH: OSR Press, 2007.
- VOGELS, W.: "File System Usage in Windows NT 4.0," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 93-109, 1999.
- VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G.C., and BREWER, E.: "Capriccio: Scalable Threads for Internet Services," *Proc. 19th Symp. on Operating Systems Principles*, ACM, pp. 268-281, 2003.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S.C., SCHAUSER, K.E.: "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. on Computer Arch.*, ACM, pp. 256-266, 1992.
- VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A.C., VOELKER, G.M., and SAVAGE, S.: "Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm," *Proc. 20th Symp. on Operating Systems Principles*, ACM, pp. 148-162, 2005.
- WAGNER, D., and DEAN, D.: "Intrusion Detection via Static Analysis," *IEEE Symp. on Security and Privacy*, IEEE, pp. 156-165, 2001.
- WAGNER, D., and SOTO, P.: "Mimicry Attacks on Host-Based Intrusion Detection Systems," *Proc. Ninth ACM Conf. on Computer and Commun. Security*, ACM, pp. 255-264, 2002.
- WAHBE, R., LUCCO, S., ANDERSON, T., and GRAHAM, S.: "Efficient Software-Based Fault Isolation," *Proc. 14th Symp. on Operating Systems Principles*, ACM, pp. 203-216, 1993.
- WALDO, J.: "The Jini Architecture for Network-Centric Computing," *Commun. of the ACM*, vol. 42, pp. 76-82, July 1999.
- WALDO, J.: "Alive and Well: Jini Technology Today," *Computer*, vol. 33, pp. 107-109, June 2000.
- WALDSPURGER, C.A.: "Memory Resource Management in VMware ESX server," *ACM SIGOPS Operating System Rev.*, vol. 36, pp. 181-194, Jan. 2002.
- WALDSPURGER, C.A., and WEIHI, W.E.: "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 1-12, 1994.
- WALKER, W., and CRAGON, H.G.: "Interrupt Processing in Concurrent Processors," *Computer*, vol. 28, pp. 36-46, June 1995.
- WANG, A., KUENNING, G., REIHER, P., and POPEK, G.: "The Conquest File System: Better Performance through a Disk/Persistent-RAM Hybrid Design," *ACM Trans. on Storage*, vol. 2, pp. 309-348, Aug. 2006.
- WANG, L., and DASGUPTA, P.: "Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System," *Proc. 21st Int'l Conf. on Advanced Information Networking and Applications Workshops*, IEEE, pp. 583-589, 2007.
- WANG, L., and XIAO, Y.: "A Survey of Energy-Efficient Scheduling Mechanisms in Sensor Networks," *Mobile Networks and Applications*, vol. 11, pp. 723-740, Oct. 2006a.
- WANG, R.Y., ANDERSON, T.E., and PATTERSON, D.A.: "Virtual Log Based File Systems for a Programmable Disk," *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, pp. 29-43, 1999.
- WANG, X., LI, Z., XU, J., REITER, M.K., KIL, C., and CHOI, J.Y.: "Packet vaccine: Black-Box Exploit Detection and Signature Generation," *Proc. 13th ACM Conf. on Computer and Commun. Security*, ACM, pp. 37-46, 2006b.
- WEIL, S.A., BRANDT, S.A., MILLER, E.L., LONG, D.D.E., and MALTZAHN, C.: "Ceph: A Scalable, High-Performance Distributed File System," *Proc. Seventh Symp. on Operating System Design and Implementation*, USENIX, pp. 307-320, 2006.
- WEISER, M., WELCH, B., DEMERS, A., and SHENKER, S.: "Scheduling for Reduced CPU Energy," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 13-23, 1994.
- WHEELER, P., and FULP, E.: "A Taxonomy of Parallel Techniques of Intrusion Detection," *Proc. 45th Annual Southeast Regional Conf.*, ACM, pp. 278-282, 2007.
- WHITAKER, A., COX, R.S., SHAW, M., and GRIBBLE, S.D.: "Rethinking the Design of Virtual Machine Monitors," *Computer*, vol. 38, pp. 57-62, May 2005.
- WHITAKER, A., SHAW, M., and GRIBBLE, S.D.: "Scale and Performance in the Denali Isolation Kernel," *ACM SIGOPS Operating Systems Rev.*, vol. 36, pp. 195-209, Jan. 2002.
- WILLIAMS, A., THIES, W., and ERNST, M.D.: "Static Deadlock Detection for Java Libraries," *Proc. European Conf. on Object-Oriented Programming*, Springer, pp. 602-629, 2005.
- WIRES, J., and FEELEY, M.: "Secure File System Versioning at the Block Level," *Proc. Eurosys 2007*, ACM, pp. 203-215, 2007.
- WIRTH, N.: "A Plea for Lean Software," *Computer*, vol. 28, pp. 64-68, Feb. 1995.
- WOLF, W.: "The Future of Multiprocessor Systems-on-Chip," *Proc. 41st Annual Conf. on Design Automation*, ACM, pp. 681-685, 2004.
- WONG, C.K.: *Algorithmic Studies in Mass Storage Systems*, New York: Computer Science Press, 1983.
- WRIGHT, C.P., SPILLANE, R., SIVATHANU, G., and ZADOK, E.: "Extending ACID Semantics to the File System," *ACM Trans. on Storage*, vol. 3, Art. 4, June 2007.
- WU, M.-W., HUANG, Y., WANG, Y.-M., and KUO, S.Y.: "A Stateful Approach to Spyware Detection and Removal," *Proc. 12th Pacific Rim Int'l Symp. on Dependable Computing*, IEEE, pp. 173-182, 2006.

- Server," *Proc. 13th IEEE Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, pp. 337-342, 2004.
- ZARANDI, S., and THOMASIAN, A.: "Optimization of Online Disk Scheduling Algorithms," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 33., pp. 42-46, 2006.
- ZEKAUSKAS, M.J., SAWDON, W.A., and BERSHAD, B.N.: "Software Write Detection for a Distributed Shared Memory," *Proc. First Symp. on Operating System Design and Implementation*, USENIX, pp. 87-100, 1994.
- ZEIDOVICH, N., BOYD-WICKIZER, KOHLER, E., and MAZIERES, D.: "Making Information Flow Explicit in HiStar," *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, pp. 263-278, 2006.
- ZHANG, L., PARKER, M., and CARTER, J.: "Efficient Address Remapping in Distributed Shared-Memory Systems," *ACM Trans. on Arch. and Code. Optimization*, vol. 3, pp. 209-229, June 2006.
- ZHANG, Z., and GHOSE, K.: "HFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance," *Proc. Eurosys 2007*, ACM, pp. 175-187, 2007.
- ZHOU, Y., and LEE, E.A.: "A Causality Interface for Deadlock Analysis in Dataflow," *Proc. 6th Int'l Conf. on Embedded Software*, ACM/IEEE, pp. 44-52, 2006.
- ZHOU, Y., and PHILBIN, J.F.: "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," *Proc. Annual Tech. Conf.*, USENIX, pp. 91-104, 2001.
- ZOBEL, D.: "The Deadlock Problem: A Classifying Bibliography," *ACM SIGOPS Operating Systems Rev.*, vol. 17, pp. 6-16, Oct. 1983.
- ZUBER, K.M., PILLAI, P., and SHIN, K.G.: "EMERALDS: A Small-Memory Real-Time Microkernel," *Proc. 17th Symp. on Operating Systems Principles*, ACM, pp. 277-299, 1999.
- ZWICKY, E.D.: "Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not," *Proc. Fifth Conf. on Large Installation Systems Admin.*, USENIX, pp. 181-190, 1991.
- WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C., and POLLACK, F.J.: "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. of the ACM*, vol. 17, pp. 337-345, June 1974.
- YAHAV, I., RASCHID, L., and ANDRADE, H.: "Bid Based Scheduler with Backfilling for a Multiprocessor System," *Proc. Ninth Int'l Conf. on Electronic Commerce*, ACM, pp. 459-468, 2007.
- YANG, J., TWOHEY, P., ENGLER, D., and MUSUVATHI, M.: "Using Model Checking to Find Serious File System Errors," *ACM Trans. on Computer Systems*, vol. 24, pp. 393-423, 2006.
- YANG, L., and PENG, L.: "SecCMP: A Secure Chip-Multiprocessor Architecture," *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, ACM, pp. 72-76, 2006.
- YOON, E.J., RYU, E.-K., and YOO, K.-Y.: "A Secure User Authentication Scheme Using Hash Functions," *ACM SIGOPS Operating Systems Rev.*, vol. 38, pp. 62-68, April 2004.
- YOUNG, M., TEVANI, A., JR., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKEY, W., BLACK, D., and BARON, R.: "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. 11th Symp. on Operating Systems Principles*, ACM, pp. 63-76, 1987.
- YU, H., AGRAWAL, D., and EL ABBADI, A.: "MEMS-Based Storage Architecture for Relational Databases," *Vldb J.*, vol. 16, pp. 251-268, April 2007.
- YUAN, W., and NAHRSTEDT, K.: "Energy-Efficient CPU Scheduling for Multimedia Systems," *ACM Trans. on Computer Systems*, ACM, vol. 24, pp. 292-331, Aug. 2006.
- ZACHARY, G.P.: *Showstopper*, New York: Maxwell Macmillan, 1994.
- ZAHORJAN, J., LAZOWSKA, E.D., and EAGER, D.L.: "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 180-198, April 1991.
- ZAIA, A., BRUNEO, D., and PULIAFITO, A.: "A Scalable Grid-Based Multimedia

[General Information]

书名=现代操作系统 原书第3版

作者= (荷) 塔嫩鲍姆著; 陈向群, 马洪兵等译

页数= 582

出版社=北京市:机械工业出版社

出版日期= 2009.07

SS号= 12405282

DX号= 000006750201

URL= <http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000006750201&d=5BF2143D94BEBBA6AB595C91081F33DC>

第1章

引论

- 1.1 什么是操作系统
 - 1.1.1 作为扩展机器的操作系统
 - 1.1.2 作为资源管理者的操作系统
- 1.2 操作系统的历史
 - 1.2.1 第一代(1945~1955):真空管和穿孔卡片
 - 1.2.2 第二代(1955~1965):晶体管和批处理系统
 - 1.2.3 第三代(1965~1980):集成电路芯片和多道程序设计
 - 1.2.4 第四代(1980年至今):个人计算机
- 1.3 计算机硬件介绍
 - 1.3.1 处理器
 - 1.3.2 存储器
 - 1.3.3 磁盘
 - 1.3.4 磁带
 - 1.3.5 I/O设备
 - 1.3.6 总线
 - 1.3.7 启动计算机
- 1.4 操作系统大观园
 - 1.4.1 大型机操作系统
 - 1.4.2 服务器操作系统
 - 1.4.3 多处理器操作系统
 - 1.4.4 个人计算机操作系统
 - 1.4.5 掌上计算机操作系统
 - 1.4.6 嵌入式操作系统
 - 1.4.7 传感器节点操作系统
 - 1.4.8 实时操作系统
 - 1.4.9 智能卡操作系统
- 1.5 操作系统概念
 - 1.5.1 进程
 - 1.5.2 地址空间
 - 1.5.3 文件
 - 1.5.4 输入/输出
 - 1.5.5 保护
 - 1.5.6 shell
 - 1.5.7 个体重复系统发育
- 1.6 系统调用
 - 1.6.1 用于进程管理的系统调用
 - 1.6.2 用于文件管理的系统调用
 - 1.6.3 用于目录管理的系统调用
 - 1.6.4 各种系统调用
 - 1.6.5 Windows Win32 API
- 1.7 操作系统结构
 - 1.7.1 单体系统
 - 1.7.2 层次式系统
 - 1.7.3 微内核
 - 1.7.4 客户机-服务器模式
 - 1.7.5 虚拟机
 - 1.7.6 外核
- 1.8 依靠C的世界
 - 1.8.1 C语言
 - 1.8.2 头文件
 - 1.8.3 大型编程项目
 - 1.8.4 运行模型
- 1.9 有关操作系统的研究
- 1.10 本书其他部分概要
- 1.11 公制单位
- 1.12 小结

习题

第2章 进程与线程

- 2.1 进程
 - 2.1.1 进程模型
 - 2.1.2 创建进程
 - 2.1.3 进程的终止
 - 2.1.4 进程的层次结构
 - 2.1.5 进程的状态
 - 2.1.6 进程的实现
 - 2.1.7 多道程序设计模型
- 2.2 线程
 - 2.2.1 线程的使用

- 2.2.2 经典的线程模型
- 2.2.3 P O S I X 线程
- 2.2.4 在用户空间中实现线程
- 2.2.5 在内核中实现线程
- 2.2.6 混合实现
- 2.2.7 调度程序激活机制
- 2.2.8 弹出式线程
- 2.2.9 使单线程代码多线程化
- 2.3 进程间通信
 - 2.3.1 竞争条件
 - 2.3.2 临界区
 - 2.3.3 忙等待的互斥
 - 2.3.4 睡眠与唤醒
 - 2.3.5 信号量
 - 2.3.6 互斥量
 - 2.3.7 管程
 - 2.3.8 消息传递
 - 2.3.9 屏障
- 2.4 调度
 - 2.4.1 调度介绍
 - 2.4.2 批处理系统中的调度
 - 2.4.3 交互式系统中的调度
 - 2.4.4 实时系统中的调度
 - 2.4.5 策略和机制
 - 2.4.6 线程调度
- 2.5 经典的 I P C 问题
 - 2.5.1 哲学家就餐问题
 - 2.5.2 读者 - 写者问题
- 2.6 有关进程和线程的研究
- 2.7 小结

习题

第3章 存储管理

- 3.1 无存储器抽象
- 3.2 一种存储器抽象：地址空间
 - 3.2.1 地址空间的概念
 - 3.2.2 交换技术
 - 3.2.3 空闲内存管理
- 3.3 虚拟内存
 - 3.3.1 分页
 - 3.3.2 页表
 - 3.3.3 加速分页过程
 - 3.3.4 针对大内存的页表
- 3.4 页面置换算法
 - 3.4.1 最优页面置换算法
 - 3.4.2 最近未使用页面置换算法
 - 3.4.3 先进先出页面置换算法
 - 3.4.4 第二次机会页面置换算法
 - 3.4.5 时钟页面置换算法
 - 3.4.6 最近最少使用页面置换算法
 - 3.4.7 用软件模拟 L R U
 - 3.4.8 工作集页面置换算法
 - 3.4.9 工作集时钟页面置换算法
 - 3.4.10 页面置换算法小结
- 3.5 分页系统中的设计问题
 - 3.5.1 局部分配策略与全局分配策略
 - 3.5.2 负载控制
 - 3.5.3 页面大小
 - 3.5.4 分离的指令空间和数据空间
 - 3.5.5 共享页面
 - 3.5.6 共享库
 - 3.5.7 内存映射文件
 - 3.5.8 清除策略
 - 3.5.9 虚拟内存接口
- 3.6 有关实现的问题
 - 3.6.1 与分页有关的工作
 - 3.6.2 缺页中断处理
 - 3.6.3 指令备份
 - 3.6.4 锁定内存中的页面
 - 3.6.5 后备存储
 - 3.6.6 策略和机制的分离
- 3.7 分段
 - 3.7.1 纯分段的实现
 - 3.7.2 分段和分页结合：MULTICS
 - 3.7.3 分段和分页结合：Intel Pentium

3.8 有关存储管理的研究

3.9 小结

习题

第4章 文件系统

4.1 文件

4.1.1 文件命名

4.1.2 文件结构

4.1.3 文件类型

4.1.4 文件存取

4.1.5 文件属性

4.1.6 文件操作

4.1.7 使用文件系统调用的一个示例程序

4.2 目录

4.2.1 一级目录系统

4.2.2 层次目录系统

4.2.3 路径名

4.2.4 目录操作

4.3 文件系统的实现

4.3.1 文件系统布局

4.3.2 文件的实现

4.3.3 目录的实现

4.3.4 共享文件

4.3.5 日志结构文件系统

4.3.6 日志文件系统

4.3.7 虚拟文件系统

4.4 文件系统管理和优化

4.4.1 磁盘空间管理

4.4.2 文件系统备份

4.4.3 文件系统的一致性

4.4.4 文件系统性能

4.4.5 磁盘碎片整理

4.5 文件系统实例

4.5.1 C D - R O M文件系统

4.5.2 M S - D O S文件系统

4.5.3 U N I X V 7文件系统

4.6 有关文件系统的研究

4.7 小结

习题

第5章 输入/输出

5.1 I / O硬件原理

5.1.1 I / O设备

5.1.2 设备控制器

5.1.3 内存映射 I / O

5.1.4 直接存储器存取

5.1.5 重温中断

5.2 I / O软件原理

5.2.1 I / O软件的目标

5.2.2 程序控制 I / O

5.2.3 中断驱动 I / O

5.2.4 使用DMA的 I / O

5.3 I / O软件层次

5.3.1 中断处理程序

5.3.2 设备驱动程序

5.3.3 与设备无关的 I / O软件

5.3.4 用户空间的 I / O软件

5.4 盘

5.4.1 盘的硬件

5.4.2 磁盘格式化

5.4.3 磁盘臂调度算法

5.4.4 错误处理

5.4.5 稳定存储器

5.5 时钟

5.5.1 时钟硬件

5.5.2 时钟软件

5.5.3 软定时器

5.6 用户界面：键盘、鼠标和监视器

5.6.1 输入软件

5.6.2 输出软件

5.7 瘦客户机

5.8 电源管理

5.8.1 硬件问题

5.8.2 操作系统问题

5.8.3 应用程序问题

5.9 有关输入/输出的研究

5.10 小结

习题

第6章 死锁

6.1 资源

6.1.1 可抢占资源和不可抢占资源

6.1.2 资源获取

6.2 死锁概述

6.2.1 资源死锁的条件

6.2.2 死锁建模

6.3 鸵鸟算法

6.4 死锁检测和死锁恢复

6.4.1 每种类型一个资源的死锁检测

6.4.2 每种类型多个资源的死锁检测

6.4.3 从死锁中恢复

6.5 死锁避免

6.5.1 资源轨迹图

6.5.2 安全状态和不安全状态

6.5.3 单个资源的银行家算法

6.5.4 多个资源的银行家算法

6.6 死锁预防

6.6.1 破坏互斥条件

6.6.2 破坏占有和等待条件

6.6.3 破坏不可抢占条件

6.6.4 破坏环路等待条件

6.7 其他问题

6.7.1 两阶段加锁

6.7.2 通信死锁

6.7.3 活锁

6.7.4 饥饿

6.8 有关死锁的研究

6.9 小结

习题

第7章 多媒体操作系统

7.1 多媒体简介

7.2 多媒体文件

7.2.1 视频编码

7.2.2 音频编码

7.3 视频压缩

7.3.1 J P E G 标准

7.3.2 M P E G 标准

7.4 音频压缩

7.5 多媒体进程调度

7.5.1 调度同质进程

7.5.2 一般实时调度

7.5.3 速率单调调度

7.5.4 最早最终时限优先调度

7.6 多媒体文件系统范型

7.6.1 V C R 控制功能

7.6.2 近似视频点播

7.6.3 具有 V C R 功能的近似视频点播

7.7 文件存放

7.7.1 在单个磁盘上存放文件

7.7.2 两个替代的文件组织策略

7.7.3 近似视频点播的文件存放

7.7.4 在单个磁盘上存放多个文件

7.7.5 在多个磁盘上存放文件

7.8 高速缓存

7.8.1 块高速缓存

7.8.2 文件高速缓存

7.9 多媒体磁盘调度

7.9.1 静态磁盘调度

7.9.2 动态磁盘调度

7.10 有关多媒体的研究

7.11 小结

习题

第8章 多处理机系统

8.1 多处理机

8.1.1 多处理机硬件

8.1.2 多处理机操作系统类型

8.1.3 多处理机同步

8.1.4 多处理机调度

8.2 多计算机

8.2.1 多计算机硬件

8.2.2 低层通信软件

- 8.2.3 用户层通信软件
 - 8.2.4 远程过程调用
 - 8.2.5 分布式共享存储器
 - 8.2.6 多计算机调度
 - 8.2.7 负载均衡
 - 8.3 虚拟化
 - 8.3.1 虚拟化的条件
 - 8.3.2 I 型管理程序
 - 8.3.3 II 型管理程序
 - 8.3.4 准虚拟化
 - 8.3.5 内存的虚拟化
 - 8.3.6 I / O 设备的虚拟化
 - 8.3.7 虚拟工具
 - 8.3.8 多核处理机上的虚拟机
 - 8.3.9 授权问题
 - 8.4 分布式系统
 - 8.4.1 网络硬件
 - 8.4.2 网络服务和协议
 - 8.4.3 基于文档的中间件
 - 8.4.4 基于文件系统的中间件
 - 8.4.5 基于对象的中间件
 - 8.4.6 基于协作的中间件
 - 8.4.7 网格
 - 8.5 有关多处理机系统的研究
 - 8.6 小结
- 习题
- 第9章 安全
- 9.1 环境安全
 - 9.1.1 威胁
 - 9.1.2 入侵者
 - 9.1.3 数据意外遗失
 - 9.2 密码学原理
 - 9.2.1 私钥加密技术
 - 9.2.2 公钥加密技术
 - 9.2.3 单向函数
 - 9.2.4 数字签名
 - 9.2.5 可信平台模块
 - 9.3 保护机制
 - 9.3.1 保护域
 - 9.3.2 访问控制列表
 - 9.3.3 权能
 - 9.3.4 可信系统
 - 9.3.5 可信计算基
 - 9.3.6 安全系统的形式化模型
 - 9.3.7 多级安全
 - 9.3.8 隐蔽信道
 - 9.4 认证
 - 9.4.1 使用口令认证
 - 9.4.2 使用实际物体的认证方式
 - 9.4.3 使用生物识别的验证方式
 - 9.5 内部攻击
 - 9.5.1 逻辑炸弹
 - 9.5.2 后门陷阱
 - 9.5.3 登录欺骗
 - 9.6 利用代码漏洞
 - 9.6.1 缓冲区溢出攻击
 - 9.6.2 格式化字符串攻击
 - 9.6.3 返回 l i b c 攻击
 - 9.6.4 整数溢出攻击
 - 9.6.5 代码注入攻击
 - 9.6.6 权限提升攻击
 - 9.7 恶意软件
 - 9.7.1 特洛伊木马
 - 9.7.2 病毒
 - 9.7.3 蠕虫
 - 9.7.4 间谍软件
 - 9.7.5 r o o t k i t
 - 9.8 防御
 - 9.8.1 防火墙
 - 9.8.2 反病毒和抑制反病毒技术
 - 9.8.3 代码签名
 - 9.8.4 囚禁
 - 9.8.5 基于模型的入侵检测
 - 9.8.6 封装移动代码

9.8.7 Java安全性

9.9 有关安全性研究

9.10 小结

习题

第10章 实例研究1:Linux

10.1 UNIX与Linux的历史

10.1.1 UNICS

10.1.2 PDP-11 UNIX

10.1.3 可移植的UNIX

10.1.4 Berkeley UNIX

10.1.5 标准UNIX

10.1.6 MINIX

10.1.7 Linux

10.2 Linux概述

10.2.1 Linux的设计目标

10.2.2 到Linux的接口

10.2.3 shell

10.2.4 Linux应用程序

10.2.5 内核结构

10.3 Linux中的进程

10.3.1 基本概念

10.3.2 Linux中进程管理相关的系统调用

10.3.3 Linux中进程与线程的实现

10.3.4 Linux中的调度

10.3.5 启动Linux系统

10.4 Linux中的内存管理

10.4.1 基本概念

10.4.2 Linux中的内存管理系统调用

10.4.3 Linux中内存管理的实现

10.4.4 Linux中的分页

10.5 Linux中的I/O系统

10.5.1 基本概念

10.5.2 网络

10.5.3 Linux的输入/输出系统调用

10.5.4 输入/输出在Linux中的实现

10.5.5 Linux中的模块

10.6 Linux文件系统

10.6.1 基本概念

10.6.2 Linux的文件系统调用

10.6.3 Linux文件系统的实现

10.6.4 NFS:网络文件系统

10.7 Linux的安全性

10.7.1 基本概念

10.7.2 Linux中安全相关的系统调用

10.7.3 Linux中的安全实现

10.8 小结

习题

第11章 实例研究2:Windows Vista

11.1 Windows Vista的历史

11.1.1 20世纪80年代:MS-DOS

11.1.2 20世纪90年代:基于MS-DOS的Windows

11.1.3 21世纪:基于NT的Windows

11.1.4 Windows Vista

11.2 Windows Vista编程

11.2.1 内部NT应用编程接口

11.2.2 Win32应用编程接口

11.2.3 Windows注册表

11.3 系统结构

11.3.1 操作系统结构

11.3.2 启动Windows Vista

11.3.3 对象管理器的实现

11.3.4 子系统、DLL和用户态服务

11.4 Windows Vista中的进程和线程

11.4.1 基本概念

11.4.2 作业、进程、线程和纤程管理API调用

11.4.3 进程和线程的实现

11.5 内存管理

11.5.1 基本概念

11.5.2 内存管理系统调用

11.5.3 存储管理的实现

11.6 Windows Vista的高速缓存

11.7 Windows Vista的输入/输出

11.7.1 基本概念

11.7.2 输入/输出API调用

- 11.7.3 I/O实现
- 11.8 Windows NT文件系统
 - 11.8.1 基本概念
 - 11.8.2 NTFS文件系统的实现
- 11.9 Windows Vista中的安全
 - 11.9.1 基本概念
 - 11.9.2 安全相关的API调用
 - 11.9.3 安全性的实现
- 11.10 小结

习题

第12章 实例研究3：Symbian操作系统

- 12.1 Symbian操作系统的历史
 - 12.1.1 Symbian操作系统的起源：Psion和EPOC
 - 12.1.2 Symbian操作系统版本6
 - 12.1.3 Symbian操作系统版本7
 - 12.1.4 今天的Symbian操作系统
- 12.2 Symbian操作系统概述
 - 12.2.1 面向对象
 - 12.2.2 微内核设计
 - 12.2.3 Symbian操作系统纳核
 - 12.2.4 客户机/服务器资源访问
 - 12.2.5 较大操作系统的特点
 - 12.2.6 通信与多媒体
- 12.3 Symbian操作系统中的进程和线程
 - 12.3.1 线程和纳线程
 - 12.3.2 进程
 - 12.3.3 活动对象
 - 12.3.4 进程间通信
- 12.4 内存管理
 - 12.4.1 没有虚拟内存的系统
 - 12.4.2 Symbian操作系统的寻址方式
- 12.5 输入和输出
 - 12.5.1 设备驱动
 - 12.5.2 内核扩展
 - 12.5.3 直接存储器访问
 - 12.5.4 特殊情况：存储介质
 - 12.5.5 阻塞I/O
 - 12.5.6 可移动存储器
- 12.6 存储系统
 - 12.6.1 移动设备文件系统
 - 12.6.2 Symbian操作系统文件系统
 - 12.6.3 文件系统安全和保护
- 12.7 Symbian操作系统的安全
- 12.8 Symbian操作系统中的通信
 - 12.8.1 基本基础结构
 - 12.8.2 更仔细地观察基础结构
- 12.9 小结

习题

第13章 操作系统设计

- 13.1 设计问题的本质
 - 13.1.1 目标
 - 13.1.2 设计操作系统为什么困难
- 13.2 接口设计
 - 13.2.1 指导原则
 - 13.2.2 范型
 - 13.2.3 系统调用接口
- 13.3 实现
 - 13.3.1 系统结构
 - 13.3.2 机制与策略
 - 13.3.3 正交性
 - 13.3.4 命名
 - 13.3.5 绑定的时机
 - 13.3.6 静态与动态结构
 - 13.3.7 自顶向下与自底向上的实现
 - 13.3.8 实用技术
- 13.4 性能
 - 13.4.1 操作系统为什么运行缓慢
 - 13.4.2 什么应该优化
 - 13.4.3 空间-时间的权衡
 - 13.4.4 高速缓存
 - 13.4.5 线索
 - 13.4.6 利用局部性
 - 13.4.7 优化常见的情况
- 13.5 项目管理

- 13.5.1 人月神话
- 13.5.2 团队结构
- 13.5.3 经验的作用
- 13.5.4 没有银弹
- 13.6 操作系统设计的趋势
 - 13.6.1 虚拟化
 - 13.6.2 多核芯片
 - 13.6.3 大型地址空间操作系统
 - 13.6.4 联网
 - 13.6.5 并行系统与分布式系统
 - 13.6.6 多媒体
 - 13.6.7 电池供电的计算机
 - 13.6.8 嵌入式系统
 - 13.6.9 传感节点
- 13.7 小结

习题

第14章 阅读材料及参考文献

- 14.1 进行深入阅读的建议
 - 14.1.1 简介及概要
 - 14.1.2 进程和线程
 - 14.1.3 存储管理
 - 14.1.4 输入/输出
 - 14.1.5 文件系统
 - 14.1.6 死锁
 - 14.1.7 多媒体操作系统
 - 14.1.8 多处理机系统
 - 14.1.9 安全
 - 14.1.10 Linux
 - 14.1.11 Windows Vista
 - 14.1.12 Symbian操作系统
 - 14.1.13 设计原则
- 14.2 按字母顺序排序的参考文献