普罗赛斯，赛莱德斯和约伯斯

——《深入解析 Windows 操作系统》翻译后记

范德成

(Robbie Fan)

2015 年 3 月

缘起

首先说明一下，这个标题完全是音译。本来是想叫《普罗赛斯，赛莱德斯和乔布斯》的，但后来想还是不要惊动乔帮主了。为什么叫这个名字呢，那得从我参与翻译的那一部分说起。

我本人无论是在上大学的时候，还是大学毕业以后，都一直对操作系统很有兴趣。而 Windows 又是每天都用的操作系统，自然更加让我兴致盎然。这本《Windows Internals》也是我一直感兴趣的书籍。所以当好友高博给我打电话说有机会参与《深入解析 Windows 操作系统》第六版的翻译时，我感到非常欣喜。

《深入解析 Windows 操作系统》的前一个中文版是潘爱民老师翻译的第四版。在高博的引荐下，我在上海和潘老师见了一面。潘老师那天正好从北京来上海出差，所以我们有幸见到。潘老师比我们年长十岁左右。他曾在北京大学教书，也在微软研究院工作过，而当我见到他的时候，他已经加盟盛大创新院了。他是权威的操作系统专家。潘老师和蔼而深沉。他说的每一句话都掷地有声。他说起，Windows Internals 一书是 David A. Solomon 和 Mark E. Russinovich 合著的。这系列书的第一本是 Helen Custer 写的 Inside Windows NT。之后出版的 Inside Windows 2000（算作第三版）、Windows Internals 第四版直至第六版等都是 David 和 Mark 编写的。这次第六版的翻译，潘老师已经做完前四章。其余的章节，我们将在潘老师第四版文本的基础上进行更新。我翻译的第一个章节是上册第五章，标题是 Processes, Threads and Jobs，后来就成了这篇博客的标题。

在我看来，Mark Russinovich 是黑客级的大师。和 C#语言之父 Anders Hejlsberg 这位计算机语言设计大师的风格不同，Mark 更像是在黑客世界中逐渐出名的。他在 Windows NT Magazine（NTMag；后来叫 Windows IT Pro 杂志）中显露出才华的时候，经常做一些修改或破坏 Windows 内部机制的程序，来研究 Windows NT 的内部机理并揭示其短处，乃至能让 Windows 蓝屏。因而，那时的他被微软员工称为“邪恶的 Mark Russinovich”。受到这些经历的影响，他发明了 LiveKd。NT 内核调试通常需要一台被调试的宿主机和一台通过 COM 端口连接的调试机，而利用 LiveKd，直接就可以方便地调试本地机器上的系统。

后来，他与 David Cutler 的小同事 David Solomon 合著了 Inside Windows 2000。David Solomon 和 David Cutler 有着很久的交情——David Cutler 在 DEC 公司做 project leader 的时候，David Solomon 加入 DEC，那时他才 16 岁；到 Windows 2000 发布的时候，已经十多年过去了。而那时 Mark 仍旧是非微软人员，他只能通过 WinDbg、KD 等调试工具来研究 Windows 的内部机理。而 David Solomon 虽然也不是微软人员，但由于他是 Windows NT 的讲师，因而被授权以进入微软大楼阅读 Windows 2000 的源代码。两人经常比赛，对某一个问题谁能更快地得到答案，而多数情况下却是 Mark 胜出。这让 David Solomon“不胜懊恼”。

在 2006 年，由于 Mark 参加了和微软员工一起的 Windows 内核培训的缘故，和微软内部人员混得挺熟。后来，微软收购了 Mark 和他的朋友 Bryce Cogswell 合开的公司 Winternals，从而 Mark 正式进入微软工作，成为微软的 Technical Fellow 之一。Technical Fellow 是微软中屈指可数的“领军人物”，它不是管理型职位，但却是技术人员最大的荣耀。成为了 Technical Fellow 也就成为了全微软数万员工的榜样。David Solomon 之前都无法想象，“邪恶的 Mark Russinovich”居然还能在微软的办公室里面这样进进出出，让人总感觉有些诡异。CSDN《程

序员》杂志 2007 年 6 月刊的《所罗门的宝藏》一文中讲到了这段有趣的历史。

Mark 不仅仅参与 Winternals 公司的商业软件的开发，同时他也开发了许多免费软件，以 Sysinternals 的名义发布，诸如 Process Explorer、Process Monitor、PageDefrag、PsTools、AccessEnum、Autoruns、NTFS-DOS、Rootkit Revealer 等著名的实用工具。他在 TechNet 博客上也不定时地发布一些有价值的系统除错和调试的经验。

需要注意的是，Windows Internals 第六版的作者中有一位 Alex Ionescu，他并非来自微软，而是开源操作系统 React OS 的著名黑客之一。React OS 是什么呢？该开源项目最初成立之时，其目标就是做一个开源的 Windows NT/2000！（其前任项目 FreeWin95 是要做一个开源的 Windows 95！）而当今它的目标则是做一个开源的 Windows XP 兼容系统出来。所以这个项目可谓是宏图大略，只是由于它的目标是兼容一个闭源操作系统的缘故，开源人士对此有所顾忌，导致参与的人不多，开发力量不够强大，因此到目前为止还处于 alpha 版（不稳定的测试版）阶段。记得在 2006 年时，React OS 中甚至连 Cache Manager（缓存管理器）都还没有实现。但 2014 年，它通过 Indiegogo 为 React OS 社区版拉了好多赞助，希望它能成功吧。

Windows 系统的发展背景

要理解 Windows，首先得了解这一系列操作系统诞生的背景。Windows 作为微软研发的操作系统，自然而然地继承了之前 DOS 操作系统的一些特性。DOS 是一个单任务、基于命令行的操作系统。它的特点是非常小巧，其核心的三个文件：IO.SYS、MSDOS.SYS 和 COMMAND.COM 总共不超过 300KB，并提供了所有核心的操作系统服务，包括文件系统（FAT12 和 FAT16）、内存管理、程序加载管理等多个方面，还支持 TSR（内存驻留程序）。所有的硬件驱动程序都由外部提供，特别是系统 BIOS，它提供了大量的硬件支持。并且，在 MS-DOS 系统的较靠后的几个版本里，已经加入了局域网的支持，虽然不及当年 Novell 网络那么强大，但已经是微软网络基础设施的一个雏形。1993 年，最后一个 MS-DOS 版本推出，版本号是 6.22。

这里简单介绍一下，什么是文件系统：它相当于磁盘上的一种树（或有向无环图）形数据结构，用于保存目录树和文件。FAT 文件系统即文件分配表，它有一个引导扇区，加上两个文件分配表（用于冗余备份），再加上一个根目录表，之后都是可供动态分配的磁盘空间。文件系统的分配单位叫做“簇”。每个簇包含多个扇区。它的文件分配表中的每个表项长度为 12 位或 16 位（所以称为 FAT12 和 FAT16），对应磁盘上的一个簇，而表项的内容则定义了文件在磁盘上的布局。表项之间采用链表的形式来连接，所以 FAT 是一种相当简单的文件系统。因为这个原因，继 DOS 和 Windows 9x 之后，它还被广泛应用于数码设备的文件系统格式，如 U 盘、数码相机的 SD 卡和 MP3 播放器等。

Windows 3.2 及更早的系统都不真正提供操作系统核心，而是重用了 MS-DOS，但对内存和任务管理作了扩展。以 Windows 3.1 为例，DOS 程序除非采用扩充内存、4G 线性地址空间（32 位保护模式）等手段，否则访问不到标准的 640KB 以上的内存，包括 640KB 到 1024KB 之间的上位内存块和 1024KB 以上的扩展内存。当年爱好 DOS 游戏的朋友也许还记得为了尽可能得到更多 640KB 范围内的内存而用 memmaker 和人工手段反复调校的日子。但 Windows 3.1 上的 Win16 程序却显式支持扩展内存。通过 GlobalAlloc、LocalAlloc 等函数，可以分配扩展内存中的块，并在使用时加以锁定（GlobalLock/LocalLock），从而在 16 位地址空间中访问

32 位线性地址（即虚拟地址）上的内容。除此之外，Windows 3.1 已支持虚拟内存，通过换页机制，可以虚拟出比物理内存更大的内存空间。虽然使用虚拟内存在空间上占了便宜，但在时间上却很吃亏：硬盘的速度比内存要慢许多。所以在实践当中，作为用户来说，不到万不得已，还是尽量不要让实际应用程序的内存消耗超过物理内存。

那么简单地说，为什么需要扩展内存呢？640KB 不够用吗？可以这样想：如果是一个像磁盘碎片整理程序（DEFRAG.EXE）这样复杂度的程序，它的大小约 100KB。一个计算机编程语言的解释器，如 QBASIC，它的大小约 200KB。一个股票走势分析程序，大小约 400KB。一张 640x480 的 24 位真彩色非压缩图片，大小约 900KB。随着计算任务的越来越复杂、应用程序的功能越来越多，640KB 内存显然捉襟见肘。这也就是为什么内存需要扩展到 1MB 以上的原因。当时的 386 地址空间从 1MB（20 位地址空间）提升到了 4GB（32 位地址空间）。如今看来，连 4GB 的地址空间都已经不够用了，硬件还在继续提高它们的规格，所以又有了 64 位的计算平台。以英特尔平台为例，自从 2007 年 64 位奔腾开始，酷睿、酷睿 2、i3、i5、i7、酷睿 M、64 位凌动等处理器都支持超过 4GB 的内存，而服务器 CPU 则更早就支持了超大内存的配置。

Windows 3.1 也加入了多任务。多任务，从技术上说，就是让本来必须从头运行到底、独占硬件资源的应用程序，能变得不再独占，能和其他应用程序共享资源，包括 CPU、显示器和网络等，从而让用户感觉好像许多程序在“同时”运行一样。比如，它让程序的用户界面以窗口的形式呈现，可以互相切换，以便共享显示器。

但它的 CPU 共享方式是协同式多任务——应用程序要通过从窗口过程（window procedure）中返回到系统，或者调用某些特定的 API 如 Sleep、GetMessage、I/O 函数等，来主动地把 CPU 的控制权交还给 Windows，然后 Windows 才能让下一个进程运行。这种多任务方式意味着诸多程序（尤其是网络应用程序）只能以异步的方式来编写，有时显得相当麻烦。况且 C 语言又并非真正支持协程（coroutine）或状态机（state machine，参见 C# 2.0）、生成器（generator，参见 Python）等机制，这样的麻烦避免不了。

Windows 3.1 的磁盘缓存依赖于 DOS 下的 SMARTDRV.EXE。此外，在 Windows 3.1 运行的时候不可以使用磁盘碎片整理程序。只有退出之后，在纯 DOS 环境下才能安全地执行碎片整理。

磁盘缓存，就是“智能”地把磁盘中的数据放到内存中，并利用内存中的数据，使得访问速度“变快”的技术。通常的思路是，把频繁读写的数据存放在内存中，以加速下一次读取；把要写入磁盘的数据“堆积”在内存中，按磁盘地址排好序、去除重叠内容以后写入，以减少数据量和寻道次数。碎片整理，是把单个文件在磁盘上存储成几个不同位置的块合并成一个连续块的操作，这样做之后，连续读或写这个文件时，磁盘的寻道次数可以减到最小，以加快速度（这条理由对于现今的固态硬盘由于页映射的关系已不再适用，但将来如果采用 NVRAM 了，也许会因为局域性 [locality] 的缘故而变得重新适用）。

Windows 95 终于造就了一次大的革新。和 Windows 3.1 不同，它摆脱了基于 MS-DOS 这样一种限制，而自己成为了独立的操作系统。其内部依然有一个称为 MS-DOS 7.0 的 DOS 系统，但一旦 Windows 启动起来，硬件管理等任务就完全交给 Windows 来执行。在进程调度方面，Windows 95 支持抢占式多任务，并支持线程。它把 CPU 时间分为时间片，每当一个线程的

连续运行时间超过时间片的大小时，Windows 95 会通过定时器中断，将 CPU 从它那里抢占过来，并调度下一个线程（在 NT 系统中是定时器或计数器中断处理例程往 DISPATCH_LEVEL 队列中添加一个 DPC 调用，之后这个 DPC 会在 DISPATCH_LEVEL 这个中断请求级别 [IRQL] 上调用线程调度器，来完成线程调度）。并且，Windows 95 支持优先级调度，只有当高优先级的线程空闲下来了，才会调度低优先级的线程。注意，优先级调度本身并不考虑公平性和饥饿（starvation）问题，而公平性方面 Windows Vista 已给出解决方案，饥饿问题在 Windows 中有包括优先级分离、临时提高到最高优先级等好几种手段来避免绝对的饥饿，也许这是在不违背“优先级”这一理念下的折衷方案，但在我看来，或许可以通过像阶梯截止期限调度器（Staircase Deadline scheduler）那样，通过 CPU 比例来定义优先级，能更自然地避免饥饿，也能提高程序的响应性能。有关这些信息，请参考《深入解析 Windows 操作系统（第六版）》第 5 章“进程、线程和作业”。

Windows 95 在文件系统上的一个改进是，支持 Unicode 编码的长文件名。之前的 Windows 版本由于基于 DOS 的缘故，只支持 8.3 形式的短文件名，即主文件名最长为 8 个 ASCII 字符，扩展名最长为 3 个 ASCII 字符。这样的支持让计算机难以使用。人们无法通过这么短的文件名来记住大量文件到底代表什么内容。Windows 95 将这一限制打破，支持了长达 255 个字符的文件名。为了向下兼容，这种支持在 FAT 文件系统上实现，并且老版本的 DOS 也能访问这样的磁盘卷，只不过无法显示或修改长文件名。为了避免破坏长文件名的存储格式，MS-DOS 7.0 会禁止不经过文件系统解释而直接访问磁盘的工具，除非用户显式使用 lock 命令将磁盘加锁。Windows 98 还进一步引入了 FAT32，它支持超过 2GB 大小的卷，并且单个卷上的簇的个数可以超过 65536（2 的 16 次方），外加可扩展长度的根目录。

Windows 95 也支持高级的磁盘缓存和内置的磁盘碎片整理程序。另外也有许多第三方的磁盘碎片整理程序，如 Norton SpeedDisk、Diskeeper 和 VOpt 等。

虽然 Windows 95 有了许多改进，但它还是有很多不足之处。首先，它的安全性只针对单用户——即使系统上设置了多个用户，仍然无法控制它们的权限——它们对计算机都拥有完全的管理权限。其次，它为了兼容老的 DOS 程序和 Win16 程序，Windows 95 保留了一些“硬伤”，包括 Win16 互斥量、USER/GDI 资源限制、可以部分写入的系统 2GB 虚拟内存空间等，导致它不稳定，易出错，而且在某些场合下多任务不流畅。例如在多设备 I/O 的情况下，同一时刻只有一个设备可以读写。它不支持多 CPU。另外，它并不原生地支持 Unicode。这些问题在 NT 系统中都得到了解决。

NT 系统的特点

下面，我们来看一下 Windows NT 系统的功能特点。

内核的设计

早期 Windows NT 的设计目标是要实现一个微内核的结构，后来由于性能问题，才逐渐把一些组件移到内核态。最后，在它的正式发布版，也就是 Windows NT 3.1 中，才把窗口和图形子系统移到内核态。NT 的设计之初就是针对不同的 CPU 架构的通用设计。一开始 NT 所试验的目标并非英特尔 i386 CPU，而是英特尔 i860，它使用的指令集与 386 完全不同，而是

一个 RISC（精简指令集）CPU。当时这样做的目的之一，就是不让系统过分依赖于 i386 平台。绝大部分操作系统代码都使用可移植的 C 语言代码编写，少量与 CPU 或硬件打交道，无法用 C 语言编写的代码采用平台相关的汇编语言编写。虽然 i860 的尝试由于市场的原因而被放弃了，但 NT 随后被移植到了 PowerPC、MIPS、Alpha 和 i386 这四个平台上，实现了硬件层面上的“一次编写，到处运行”的愿望。有关这段历史，请参考《观止》一书[6] 。

前面讲到的图形子系统，除了位于内核模式的 Win32k.sys 中的部分之外，User32.dll（窗口用户界面 DLL）和 Gdi32.dll（图形设备接口 DLL）都是其用户模式部分。

由于图形子系统的一部分集成进了内核，Windows 的图形反应速度很快，用户感觉很流畅，这与 Linux 的 X Window 的有着可觉察延迟的情形形成了对比（当然，Linux 的 X Window 的性能更多地是受到其客户端/服务器模式的通信速度所限），也是 Windows 销路更好的原因之一。后来的 Windows Vista 系统才将图形子系统（指的是可以放进用户模式的部分，如 UI 和位图的绘制等，包括 DWM 和 DirectX）完全运行在用户模式下，以提高系统的稳定性。类似地，Android（安卓）的图形功能大部分运行在用户模式下，也有一部分运行在内核模式下，也就是 framebuffer 部分。而 Windows Vista 和安卓的图形性能却很高，这说明用户模式的图形模块并非性能降低的关键因素，关键还是要充分利用图形硬件的加速特性，减少物理内存中图形数据的非必要复制操作。

有关 Windows 内核的详细技术信息，请参考《深入解析 Windows 操作系统（第六版）》第 3 章“系统机制”。

程序的调度

前面说过，Windows 支持线程调度。进程是定义地址空间的单元，而线程则是调度的单元。早期的 CPU，单个 CPU 只有一个核心，单个核心同一时间也只能运行一个线程，所以调度多个线程的手段是把 CPU 时间切片，分配给不同的线程。多 CPU 的好处则是提高并行性（parallelism），以增加计算的速度和容量。Windows NT 的第一个版本就已经支持多 CPU。如今随着硬件的发展，已经出现了多核 CPU，且单个核心还支持“超线程”，即在同一个核心上可以“同时”跑两个线程（其原理是利用 CPU 流水线上不同运算部件的并行性；可参考胡越明著《计算机组成原理与系统结构》）。针对这些，Windows 也作了相应的支持。为了便于描述线程对 CPU 的使用，我们把最多能跑单个线程的 CPU 计算资源称为“逻辑处理器”。比如，在一台双核超线程的机器上，每个核能运行两个线程，我们就把它看作有 4 个逻辑处理器。

当线程在一个抢占式多任务操作系统中运行时，从编程者的角度来看，只需要假定 CPU 并非由这个线程独占，因此线程随时可能被挂起或恢复执行即可。至于线程是什么时候被挂起的，挂起的时间有多长，以及什么时候恢复运行等，都由操作系统来决定。Windows 操作系统是怎样处理的呢？有哪些规则来保证线程调度的公平性呢？优先级是怎样考虑的呢？这些内容已经超出了这篇文章的范围，请参考《深入解析 Windows 操作系统（第六版）》。

和 Linux 比较，Windows 调度器有它自己的特点。Linux 一度以来都用它的 O(1)调度器，无论有多少线程在运行，这个调度器的时间消耗都不会超过一个常数。后来在 2.6.16 等内核版本中，Con Kolivas 的 Staircase Deadline（SD，阶梯截止期限）调度器是一个他自己改写的版

本，但却没有被主干（main stream）调度器的维护者 Ingo Molnar 接受。后来 Ingo Molnar 基于 SD 调度器的思想，写出了 CFS（完全公平调度器）并加入主干版本 2.6.27。

为了了解各个调度器之间的区别，需要了解线程运行的特点。线程在运行时，它会暂时独占逻辑处理器一段时间。调度程序的关键目标是要让各个线程得到公平的调度，让它们得到公平的 CPU 时间份额。为了做到这一点，各个调度程序的实现方式是不同的。O(1)调度器的特点是，它会优先考虑 I/O 密集型程序。什么是 I/O 密集型程序呢？它是和“计算密集型程序”相对应的概念。一个程序可能花大量的时间来做计算，那么这样的程序只要有机会得到 CPU，就会运行尽可能长的时间，例如计算圆周率的程序（参见著名的用圆周率计算来测量 CPU 性能的程序 Super PI）。一个程序也有可能花费大量的时间来等待 I/O。这并不一定指那些往磁盘或网络上读写大量数据的程序，因为这里的关键是，它等待 I/O 所花费的时间多，换而言之就是占用 CPU 的时间少。除了磁盘碎片整理程序、网络浏览器这些程序之外，还有文本编辑器——因为它时常在等待用户按键，以及时钟程序——因为它通常会设置一个定时器并等待定时器被触发。可见，I/O 密集型线程占用的 CPU 时间相对较少。

根据这样的定义，调度程序首先判断，一个线程从当前的表现来看是“I/O 密集型”的？还是“计算密集型”的？然后给 I/O 密集型的线程赋以较高的动态优先级，让它在需要 CPU 时能抢占计算密集型线程手中的 CPU，从而提高公平性和响应性能。FreeBSD 调度器的做法是，观察线程最近 4 个时间片的使用情况，如果都没有用完就开始等待了（学术上叫做“被阻塞”），那么就给它最高的动态优先级，如果还有 3 个没用完则次之，以此类推。O(1)调度器的原理也类似。

而 Windows NT 的调度器，直到 Windows Server 2003 为止，本质上都是这样做的（详细请参见《深入解析 Windows 操作系统（第六版）》第 5 章）：以“时限单元”（quantum unit）为单位，来对线程的时间使用情况做计数。一个“时限”默认是 2 个时钟滴答，而一个时钟滴答则是 3 个时限单元。线程初始时被分配一个完整的时限（即 6 个时限单元）。每次线程被阻塞或者用完它的时间片（即“时限”）时，扣除它相应的时限单元数（不足一个时限单元的情况也算作一个时限单元）。直到所有就绪线程的时限单元用完之前，同一优先级的线程中只调度还有时限单元没用完的那些。这样带来的结果是，假设有一个线程与另一个线程互相通信，比如一个应用程序与它的数据库管理系统通过消息队列或信号量进行通信；第三个线程则一直在做计算。那么，即便线程一和线程二每次都在收到消息之后很快地做计算，再发消息并等待对方的回应，它们也会明显占劣势。

假设时限单元是 3 毫秒，这两个线程各计算 1 毫秒就结束，那么它们各被调度 1 毫秒，一共 2 毫秒。但此时这两个程序上已经被各记录了 3 毫秒。而那个计算密集型的线程则可以连续运行 18 毫秒，之后前两个线程会被再调度 4 次，把三个线程的时限都用完。于是计算密集型线程做了 18 毫秒的计算，另两个才分别计算了 6 毫秒，非常“吃亏”。在实际应用中，这一特点造成的影响表现在多个方面，当单核机器上运行一个计算密集型程序时，出现的情况有启动新进程特别慢、与图形 API 接口的应用程序速度明显变慢等等。因此使得在这些场景下 Windows 的响应速度很慢。

而我们将看到，从 Windows Vista 开始，线程的时间计算不再通过时限单元来表示，而是通过新一代 CPU 所具备的指令计数器，来做完全精确的计算[11] 。这样一来，线程调度的公平性不仅更高了，也超越了传统的 Linux O(1)调度器。

而 Linux 的 SD 或 CFS 调度器，则实现了另外一种特点。从 SD 调度器的名称来看，即阶梯截止期限，它所表示的是，高优先级的线程将不再是简单地优先于低优先级的线程。而是，高优先级的线程能比低优先级的线程获得更多的 CPU 百分比。相邻优先级的线程可以获得的 CPU 百分比从低往高是阶梯上升的，比如 1:2:4 等等。那么它主要解决了什么问题呢？就是高优先级计算密集型线程把系统拖死的情况。所以从这些角度来说，SD 或者说 CFS 带来的系统响应性比 Windows Vista 会更好。

Windows Vista 或 Windows 7 并非对高优先级线程一点制约手段也没有。它们也通过动态判断线程的情况来调整它们的优先级，以提高系统的响应性能。这一概念称为“优先级提升”，包括优先级分离、避免饥饿而提升等多种手段。优先级提升是如何实现的呢？请参考《深入解析 Windows 操作系统（第六版）》第 5 章。

比起单用户情况下的线程调度，多用户情况下的线程调度更为复杂。Windows 服务器版本支持“终端服务器”功能，也就是远程桌面服务。每一个用户可以有自己的远程桌面，能运行服务器上已安装的任意应用程序。针对这种典型的多用户情况，Windows Server 2008 R2（也就是 Windows 7 的服务器版本）提供了基于远程桌面会话的分布式公平份额调度器（Distributed Fair Share Scheduler，DFSS），来显著提高多用户情况下的线程调度公平性。详细信息，请参考《深入解析 Windows 操作系统（第六版）》第 5 章。

除了基于 x86 CPU ring0/ring1 隔离机制（或更新的采用 Intel VT/AMD-V 技术）实现的、能运行一个完整 32 位或 64 位操作系统的现代虚拟机（其早期版本，如 VMware、Virtual PC 等是在 2000 年左右实现的）之外[1]，早在 80386 时代，已存在为了兼容性而提供的运行老式程序的虚拟机。这就是 V86 模式。V86 模式可以称得上是英特尔 i386 CPU 引入的硬件虚拟化支持，但是，它虚拟化的只是 16 位的 8086 环境，而不是 80386 本身。当操作系统引导进入 386 特有的保护模式以后，DOS 和 Win16 程序就无法运行在实模式下面。此时，操作系统必须通过 V86 模式来虚拟出一个 DOS 和 Win16 环境。通过这样的虚拟化，虽然单个 DOS 程序仍被限制只能访问 640KB 内存，但多个命令行窗口中的 DOS 程序都分别运行在自己的 640KB 虚拟地址空间上，因而能够被动态映射到物理内存中的任意页面上。另外，DOS 环境中的扩展内存也通过 DPMI 接口得到了支持。支持 V86 模式的组件叫做 NTVDM（NT 虚拟 DOS 机），支持 Win16 环境的组件叫 WOW（Windows on Windows），因此有时在任务管理器里能看见 ntvdm.exe 和 wowexec.exe 这两个进程。

无独有偶，64 位 Windows 为 32 位应用程序虚拟一个运行环境的时候，这种技术也被称为 WOW。所不同的是，它的具体名称叫做 Wow6432[22]。

Linux 上用 V86 模式虚拟 DOS 运行环境的软件有 DOSEMU。由于利用了硬件虚拟化支持的缘故，它的运行速度也接近物理机。而另一个软件 DosBox 则类似于 Bochs 虚拟机，是将 DOS 程序的机器指令解释执行来模拟 DOS 的，所以速度比较慢。幸运的是，由于 DOS 流行的年代相当早，大多数 DOS 程序都是为 80486 或更老的 CPU 设计的，因此 DosBox 在绝大多数情况下已经足够快了。

为了适应多核时代的降临，Windows 7 调度器中的锁已经做了很大的优化。详细内容可以参考 MSDN Channel 9 上 Arun Kishan 的视频“Farewell to the Windows Kernel Dispatcher Lock”[13]。

这使得 Windows 7 在超过 4 核的情况下比 Windows Vista 表现得更卓越。而且，微软与客户积极协作，改进 Windows 7 在特定应用场景下的性能，也取得了不小的成效。

详细内容，可以参考《深入解析 Windows 操作系统（第六版）》第 5 章。第 5 章远不止提供了我这里讲到的这些内容，它还包含进程和线程的内核数据结构、进程的启动过程、线程的入口地址、线程池、NUMA 支持等诸多技术方面相当翔实的内容，是很好的技术参考。

内存管理器

顾名思义，内存管理器负责管理应用程序的内存使用[14] 。作为一个多任务操作系统，Windows 需要管理系统中同时运行的多个进程的内存使用。由于 i386 处理器在保护模式下支持段页式内存管理的缘故，Windows 也充分利用了这一特性。绝大多数情况下，Windows 采用页式内存管理。在这种模式下，应用程序和系统的大部分代码通过“线性地址”来访问内存。“线性地址”指的是 2^32 字节，也就是 4GB 的内存空间能直接通过 32 位地址来访问，而相对于 8086 的 16 位段号加 16 位偏移量这种非线性编址而言，它是“线性”的，所以称为“线性地址”。对于应用程序来说，只需要直接访问线性地址上的内存即可，但对于操作系统而言，它必须建立线性地址到物理内存地址的转译关系。通过这种转译，不同应用程序以及系统本身的内存可以被互相隔离和保护，而且还可以实现用磁盘来虚拟内存的效果。

在这里要科普一下，x86 的虚拟地址空间并不是把硬盘“虚拟”成内存，不是什么所有进程加上系统及空闲的物理内存放在一个空间里面，然后再跟着放上硬盘上的页面文件空间。这种“想象”可以工作，但完全和 x86 的设计不同。x86 里，以 Windows 为例，进程和它所有的 dll、内存映射文件等被映射到一块区域（比如 2GB 以下的空间），系统数据、硬件 I/O 地址等被映射到另一块区域（比如 2GB 以上的空间）。换一个进程，映射就会有很大变化。可以说映射本身（页目录、页表）就要占据不少空间。空间里也有好多“空洞”，一旦访问就会让进程崩溃（如果是内核模式代码访问，则会引起系统蓝屏或崩溃）。详细信息，可以参考《深入解析 Windows 操作系统（第六版）》第 10 章“内存管理”。

在理想的情况下，应用程序所需要访问的内存完全位于物理内存中，并且已经被映射到进程的线性地址空间内。这一线性地址空间就被称为“虚拟内存”，而被映射的物理页面集合叫做进程的工作集（working set）。简单地来说，当空闲的物理内存低于某一阈值时，内存管理器将会查询各个进程被映射的页面中最近没有被访问过的页面，并将其中的一部分“交换”到磁盘上，以便腾出空间来满足新的内存访问请求——Windows 会根据新访问页面的不同情况，创建它们或从磁盘上读取它们，再把它们映射到物理内存中。磁盘上交换内存页面用的文件通常被称为交换文件或页面文件。

线性地址空间中的内存被划分为“页面”，在 i386 体系结构里，它的大小是 4KB，且以 4KB 对齐。为了将物理内存空间与线性地址空间加以区分，物理内存空间中的 4KB 页被称为“页框”。一些别的 CPU 体系结构里，内存页面是 8KB 大小。后期的 x86 和 x64 还支持 2MB 的“大页面”。这些在《深入解析 Windows 操作系统（第六版）》第 10 章中都有介绍。

进程占用的物理内存既然被称为工作集，那么它所占用的虚拟内存被称为什么呢？在 Windows 任务管理器的“性能”标签页里面有一项“提交用量”（commit charge），它就是系统中所有虚拟内存占用量的总和。另外，一个进程的“私有字节数”（private bytes）表示

它的线性地址空间里不和其他进程共享的内存，通常这个值能反映一个进程所消耗的内存数量。我个人的一个经验是：如果“私有字节数”无法精确表示出一个进程实际消耗的内存量，比如由于它有很多共享的内存，但却没有真的被其他进程所共享的缘故。那么，一个办法就是先观察没有这个进程时的总虚拟内存消耗，然后启动这个进程，并把新的虚拟内存消耗与之前的做比较。这样就能大致看出这个进程消耗的虚拟内存数量了。

操作系统内核代码和驱动程序也需要使用内存。任务管理器中的“换页内存池”（paged pool）和“非换页内存池”（non-paged pool）表示出这种类型的内存使用，两者都位于内核的线性地址空间，其区别是一者可以换页而另一者不能。如果内核代码或驱动程序需要使用的某些内存总是在 DPC/Dispatch 这个中断请求级别（IRQL）以下（不包含该级别）被访问，那么就可以从换页内存池中分配。

与别的操作系统不同，Windows 的缓存管理器与内存管理器是紧密结合的[15] [9] 。Windows 中进程的内存，除了可以是通过堆管理器（HeapAlloc、GlobalAlloc、VirtualAlloc 等函数）分配出来的“匿名”内存之外，也可以是对文件的某一部分进行映射来得到的。内存映射文件（memory mapped files）在 Windows 内存管理器中是通过“内存区对象”来实现的。缓存管理器缓存文件的方式是，把文件要被访问的那一部分映射到系统地址空间里面，然后访问这块内存。比如在读取文件时，内存管理器会检查这块文件内容是否已经在物理内存中了，如果是，那就不必从磁盘上读取了。这样就自然成为了读取缓存。与之相似的是，普通应用程序的内存映射文件也可以被看作是带缓存的，只不过因为文件是被程序直接映射的，缓存管理器将不会参与到这个过程中。

已访问的文件数据被缓存管理器作为内存区对象映射到系统地址空间里，那么必然这部分系统地址空间也有其虚拟大小和工作集。在 32 位 Windows 上，这个虚拟大小的上限是 960MB。工作集显然不可能超过虚拟大小，但是这并不表示缓存的数据只有 960MB。原因在于，内存管理器有另外两个页面列表：备用列表（standby list）和已修改列表（modified list）。备用列表里保存了这样的物理页面：那些已经从进程工作集中去除的页面，但由于物理内存还没有那么紧张，它们还没有被用于其它目的，所以暂时仍保存着进程工作集之前的数据。已修改页面则是那些不在进程工作集，并且有数据已经被修改的页面；其中的数据会被平衡集管理器的“懒惰写”功能刷新到磁盘上。

备用列表和已修改列表合起来成为“转换列表（transition list）”。正因为这个列表的存在，缓存的实际大小是可能超过系统缓存虚拟地址空间大小的[10] 。

缓存管理器这样的特性会带来一些不方便，那就是无论是通过读、写文件 API 对文件做访问（考虑允许缓存的情况），还是通过内存映射文件方式（比如，一个可执行文件的 exe、dll 等文件就是被映射到虚拟内存中的）来读写页面，在性能监视器里这些操作都被显示为虚拟内存的换页操作。这样，很多时候发现内存换页厉害，就不知道是因为内存不够用还是有程序在读写磁盘了。

当然，缓存管理器还拥有显式预读、智能预读、延迟写等功能。详细信息请参考《深入解析 Windows 操作系统（第六版）》第 11 章“缓存管理器”。

从 Windows Vista 开始，磁盘缓存的机制也有所改进。Windows XP 的缓存管理器有一个不够

理想的特性，如果一个程序以随机访问方式或默认缓存方式打开一个几百兆这样的大文件，并将这个文件从头到底读一遍，那么缓存的工作集就会被访问几百兆的数据。而这些缓存的工作集又被当作进程工作集一样处理。进程工作集如果被这样大量地访问，自然有必要把它大范围扩大，原因是进程的数据往往会被再次访问。但对于磁盘缓存来说，往往数据只是被连续读取，即使被读取多遍，缓存在物理内存中也不会使性能提高太多，反而大量挤占了其他进程的工作集，影响系统总体性能。Windows Vista 已不再有这个性能问题。而且 Windows Vista 还有一个特性，那就是，即使应用程序指定不需要读缓存，对于读取的数据，它还是会以不影响缓存工作集大小这样一种不带来较大干扰的方式，把它缓存起来[3] 。这种现象在使用 Virtual PC 时特别明显，因为，Virtual PC 默认就是不使用读缓存的。

前面说过，Windows 内存管理器采用工作集机制来替换不经常使用的页面，这与 Linux 的 LRU（最近最少使用）不同。而且早期的 NT 内存管理器只在物理内存不够的时候才根据页面的“访问”位，修剪进程的工作集。此时的跟踪数据并不一定准确，因为它是进程积累了相当长一段时间之后使用过的页面。所以，这样的做法未必真正反映了进程的工作集。“工作集”这一名词是 Peter J. Denning 在 1968 年的论文《The Working Set Model for Program Behavior》中提出的。

工作集是一段时间里面进程访问过的页面。问题是，在实际的操作系统中，这个时间应该怎样确定。如果假设物理内存比较小，只能放得下一部分页面，那么操作系统应该优先把工作集的部分放进去。然后，随着时间的推移，进程可能访问原工作集之外的页面。

Denning 的论文指出，一般来说，进程的内存访问有着 locality（局域性）特性，但又符合一定的概率分布，就是工作集随着时间的增长，趋向于整个进程在其生命周期中所有要用到的页面。但这根曲线又是弯曲的（见 Figure 1），越往曲线开端（即时间越短）越陡峭，反之则越平缓。

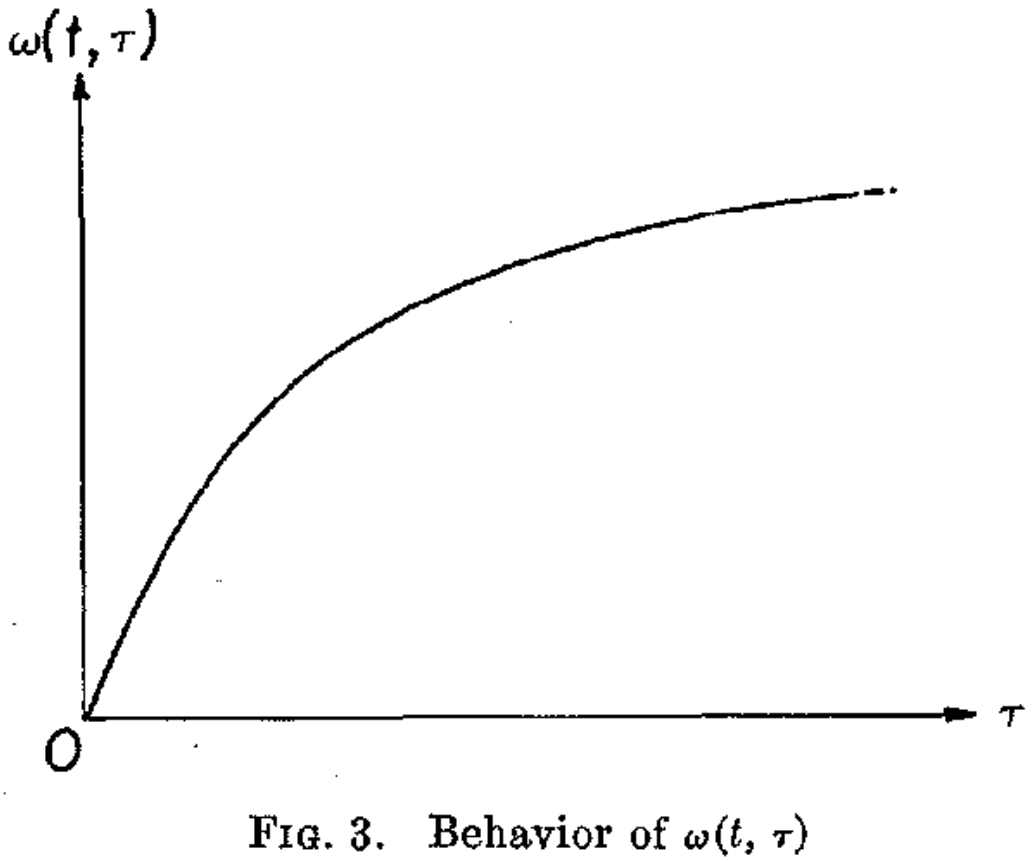

FIG. 3. Behavior of $\omega(t, \tau)$

Figure 1 - Denning's Working Set Theory

因此，我们可以设定一段时间，让这段时间所对应的进程工作集之外的所有页面在它们将来被访问时，用于加载、移除其他不用的页面的时间消耗总和小于这段指定的时间。这样，浪费在换页上的时间不会太多。

如果按上面这样定义的进程工作集能被放入物理内存，那么系统的状态就是健康的。如果上述定义的工作集太大，无法被放进物理内存，那么我们就说系统产生了颠簸（thrashing）。此时系统的性能将会特别差。

Windows XP/Server 2003 中的工作集概念虽然与 Denning 的模型有所不同，但在实践中其效果也还不错，不过，一旦发生“颠簸”，整个工作集的修剪会变得不太“理智”。其主要表现为，当进程的页面按“访问”位，把该保留的保留下，能移除的移除掉以后，如果物内存还是不够用，则会在已访问的页面中随机选择页面来移除。Linux 在这方面要表现得好很多。曾有一台 Linux 机器有 64MB 内存，跑得比较慢。后来把内存加到 128MB，用 vmstat 命令发现它实际需要的物理内存正接近 128MB。也就是说，当物理内存为系统中所有进程工作集一半大小的时候，Linux 还能用，虽然比较慢。而 Windows 遇到这种情况，会变得非常难以使用。Linux 和一些其他的*nix 系统在这方面采用的是 LRU 算法[16] 。我认为，比 LRU 更强大的缓存替换策略是老化-频率缓存[8] ；再进一步，则要用到人工智能了（马尔可夫链是其初级形式），但是，无论用什么方法，再好的方法都无法避免内存不够时的颠簸。然而，对于内存足够情况下的磁盘缓存来说，一个好的替换策略是会带来一定改善的。

前面提到过聚簇换页。它是 Windows NT 用来提高换页性能的一种方法。它在概念上有点类

似于预读缓存，利用的是磁盘连续读写的性能。需要指出它和文件缓存不同的一点是，文件中的页面是由文件系统分配、内存管理器无法移动的，而交换文件（页面文件）中哪些页面对应哪些地址的虚拟内存页面则是可以由内存管理器自己定义的。这样一来，写入页面时，就将页面在交换文件中的实际地址确定了下来，所以聚簇写是比较有效的优化。反过来，聚簇读则未必真的能让内存请求得到命中，特别是当应用程序的数据访问比较随机的时候。因此换入操作往往还是比较慢的。

Windows Vista 引入了 Superfetch 技术。Superfetch 并不是内存管理器的一部分，而是与内存管理器整合的一个组件。Superfetch 本身需要消耗一定的内存——在我的一台 1GB 内存的机器上，需要占用大约 60MB 内存。Superfetch 提供的功能是，它会在后台以较低的执行优先级把它预测到将有较大概率会被用到的内存页面放到物内存中去。它的预测机制涉及一些微软研究院发明的人工智能技术，比如，当一个程序被执行后，将有多大概率读取某一个文件，并以这样的分析来决定加载哪些页面。它也针对某些特定场合，比如中午吃完饭回来以后用户将会打开浏览器看看网页等情形做了优化。参见 Michael Fortin 的访谈[20] 。

Superfetch 在实现上还会区分页面的优先级。这样，它会把较高优先级的页面保留在物理内存中，而把低优先级的、物理内存无法容纳的页面换出到硬盘上。

除了 Superfetch 外，用户还可以手动启用 ReadyBoost 来提升性能。ReadyBoost 使用一个存储介质为闪存的缓存，来加速系统的读取速度。其原理是闪存的随机读取速度比硬盘要快得多，页面换入操作又是一个会影响应用程序响应性能的操作，所以，ReadyBoost 可以很好地缓解这一性能瓶颈。此外，再把页面换出操作尽可能做成聚簇换出页面，那么换出操作也会比较接近连续写，机械硬盘的性能优势也将得到发挥，从而从整体上提高了系统的性能。ReadyBoost 一旦打开，Superfetch 就会把中等优先级的页面放进 ReadyBoost 缓存，从而形成从物理内存到 ReadyBoost 缓存，再到机械硬盘的三层页面存储体系。

但是，ReadyBoost 的设计并不十分完美——我个人做了很多实验，结果表明 ReadyBoost 要提高性能是有先决条件的：空闲的物理内存必须相当充裕，至少要有 1/3 的空闲物理内存。如果做不到这一点，ReadyBoost 默认带来的性能提升将会非常有限。不过，我们却可以利用 Superfetch 的一个特性来让它提升性能。ReadyBoost 之所以性能不佳，原因只不过是当页面被换入时，缓存中没有这些页面，从而造成缓存不命中（记得吗，Superfetch 会把中等优先级的数据缓存在 ReadyBoost 中，那些高优先级的反而不一定在）。

要解决这个问题，可以利用 Superfetch 的一个特点：它在向物理内存填充“超级预读”数据的时候，同时也会向 ReadyBoost 缓存写入这些数据。那么怎么办呢？很简单，用一个程序，一下子占用大量物理内存（比如物理内存总量的一半），然后释放。这必然会带来大量的页面换出操作，同时 Superfetch 缓存也会被清空，留下大量的空闲内存。接下来，Superfetch 就会努力填充空闲内存，同时把这些页面也存入 ReadyBoost 缓存。填充的这些页面显然会是比进程工作集优先级低，但却比硬盘上的页面优先级高的页面。这样一来，在后续的时间里 ReadyBoost 就会很好地响应对这些页面的读取请求了，即使后来物理内存的使用量上升，也还能做出不错的响应，况且优先级更低的页面也早已缓存在 ReadyBoost 之内了。

Android 系统不支持用闪存来做虚拟内存，这使得它的内存空间被严重地限制在物理内存之中。不过，它采用内存压缩技术（起源于 Linux 中的 zRAM），在内存紧张的时候为应用程序

腾出内存空间。如果内存还是不够，就会强制关闭应用程序的界面部分，乃至杀服务进程。从这方面来说，Android 对物理内存容量的要求更高。这也就是为什么应用程序比 Android 还多的 Windows 8 平板所配置的内存通常是 2GB，不比流行的 Android 手机高，却也能流畅运行的原因。

有的系统上，提交的虚拟内存用量甚至可以达到物理内存的 1.5 倍之多。比如说，当物理内存为 4G 时，提交用量甚至可以达到 6GB 左右。这对于 Android 所使用的内存压缩技术而言，很难做到这么多。所以相对来说，同样大小物理内存的设备，Android 更容易内存不够用。

时下，有人担心平板或超级本所采用的 SSD 固态硬盘的闪存芯片的写入次数有限，因此频繁换页对它们的寿命有影响，所以把虚拟内存（页面文件）关闭。其实这在大多数情况下都是不必要的担心——假设 MLC 的 NAND 闪存能写入 10 万次数据，一个 256GB 的存储器，每天写入 256GB 的数据，都可以使用 200 年。考虑到写放大，就算有 3 倍吧，也能用 70 年。到时候电脑都早就换了吧。如果实在不放心，可以用性能监视器（运行 perfmon 命令，或者进入控制面板->管理工具打开它），观察页面换出（Pages Output/sec）量，再做区处。而且，即便换出量过大，应有的解决方案也是加大物理内存，或者关掉些应用程序，而不是关闭交换文件——因为如果那样做，应用程序反而会因为物理内存不够而退出，影响日常使用。

一个进程在启动时，内存管理器会为它分配内存。进程本身有一个可执行文件（通常扩展名为 exe），同时会加载一些动态链接库（dll）。可执行文件中的代码会调用动态链接库中的函数。而一个 exe 程序在编译、连接期间可以指定使用动态链接库的导入库（import library），这样，当可执行文件运行时，它就间接地能知道它所调用的动态链接库函数的内存地址。但是，动态链接库的加载地址并非固定的。为了适应这种动态加载的地址位置，操作系统会根据 exe 映像的全局函数指针表，根据 dll 的实际加载位置调整这些函数指针的值，这一过程叫做 rebasing（基地址调整）。

进程在启动后，程序逻辑不仅需要用到全局变量和栈上的函数局部变量，时常还需要在堆上分配内存空间以供使用。此时，系统的堆管理器就会发挥作用。堆管理器提供了 HeapAlloc 和 HeapFree 函数以提供堆内存分配和释放的功能。许多 C/C++开发环境的 malloc 函数都最终调用了 HeapAlloc 函数。堆管理器有一些独特的特性，使得它能胜任堆管理的任务。首先，所有堆内存的分配和释放都只需要简单地调用 HeapAlloc 和 HeapFree 即可，这些操作中所有隐含的虚拟内存的分配和释放操作（即 VirtualAlloc/VirtualFree）都由堆管理器来完成，应用程序不需要担心。第二，堆管理器支持多个堆，应用程序可以创建自己的堆。第三，对于小于一定大小的内存分配（在 Windows XP SP2 上是 1KB），堆管理器能以 O(1)时间复杂度来分配。其具体实现使用了十字链表，可参考“HeapAlloc 内部算法”[17] 一文。第四，Windows Server 2003 开始，默认启用了低碎片堆（Low Fragmentation Heap），能有效减少堆的内部碎片，节省虚拟内存。第五，堆管理器拥有一定的容错机制，对于行为异常的应用程序，它可以检测并修正一部分堆内存使用上的错误，特别是双重释放，以及继续使用已释放的内存块等。

于 2003 年，微软首席软件架构师 Bill Gates 呼吁全公司关注软件的安全性。历来微软操作系统最为大众所诟病的一点就是它的安全性。无论是天生不安全的 Windows 95/98/ME，还是集成了安全性控制的 Windows NT 系列，都害怕真正被用作互联网上的服务器，其原因正是真正的安全性还没有做到位（2000 年之前，微软自己 microsoft.com 网站的服务器用的还都

是 FreeBSD）。众所周知，C/C++语言的特点是运行效率高，但它们天生是不安全的，其安全性完全掌控在程序员的手里。一个 C/C++程序如果设计编写得不好，可以产生很多安全隐患，包括未初始化的变量、缓冲区溢出、空指针访问、野指针访问、访问已经释放的内存块、重复释放、内存泄漏、释放原本就不是在堆上分配的内存等等。

于是，Windows XP SP2 中引入了数据执行保护（DEP，Data Execution Prevention），用于保护最典型的缓冲区溢出漏洞。攻击者可能期望将一个过长的字符串通过网络协议传递给 Windows 服务器上的服务，并假定该字符串将被存储在一个局部数组中，该局部数组位于函数的栈上。在英特尔 x86 架构上，栈是从高地址向低地址生长的。一旦缓冲区溢出，靠近栈底的内容将被覆盖。函数的返回地址又恰恰在局部变量之前被压栈。该字符串如果成功覆盖了函数的返回地址，就能够改写该返回地址，使其指向任意函数。但是，光让服务器进程执行一个现有函数是很难达到攻击目的的，因为多数现有函数对攻击者来说也只能做很有限的事。于是，更进一步，攻击者在另一个（可以是没有溢出的）字符串里放进一个自己编写的函数的机器码，然后让返回地址指向这个函数，就可以成功地实现攻击。

数据执行保护正是针对这种类型的攻击的。由于攻击者必须把他想执行的代码放进一段数据中，只要让这段数据无法被执行，就能解除威胁。DEP 的完整实现需要能够标记“不可执行”位的 CPU（基本上 Pentium 4 及更高版本的 x86 CPU 都支持），操作系统会把程序的数据页（包括局部变量、全局变量和堆）标记为“不可执行”，如果有指令跳转到这样的页面，将会触发一个陷阱，操作系统将处理这一事件，比如把进程终止。于是就可以防范这种类型的攻击。

虽然 DEP 能够防范数据被当作代码来执行，但它不能防止函数返回值被篡改。为了消除这种攻击带来的危害，Visual C++编译器支持 stack cookie（栈小数据）的编译选项（/GS）。当函数被调用时，编译器插入的代码会在靠近栈顶且邻近函数返回地址的地方新增一个栈小数据。这样，如果攻击者要修改函数返回地址，就必然要改写栈小数据，然后，编译器插入的代码在函数返回的时候检查这个小数据，就能知道返回地址可能已经被篡改，从而终止程序的运行。栈小数据还被用于加密返回地址（叫做“指针编码”，pointer encoding），使得这种攻击难度更高。

除了 DEP 之外，Windows Vista 还支持地址空间布局随机化（ASLR，Address Space Layout Randomization）。在 DEP 的基础上，ASLR 进一步把 dll 加载的位置打乱，这样可以让攻击者更难以找到系统函数的地址，更难发起有效的攻击。关于这些技术的细节，请参考《深入解析 Windows 操作系统（第六版）》第 10 章“内存管理”。

安全性

Windows NT 从设计伊始就引入了对象管理器、文件系统和注册表的安全性概念。对象管理器负责管理所有“已创建或打开的”对象的安全性，比如一个进程、一个打开的文件、一个软硬件设备等等。文件系统，主要是指 NTFS，负责管理文件系统对象的安全性，如文件、目录等。注册表则实现了注册表键上的安全性。

Windows 内核对象和执行体对象都由对象管理器负责管理，每一个对象都有它的安全性信息——哪些用户可以访问它们，以及以怎样的方式访问它们——读取、写入、执行，以及更多随着对象类型的不同而不同的权限等等。比如，新建一个进程时，就可以通过安全描述符指

定哪些用户对它有怎样的访问权限。

NTFS 通过 ACL（Access Control List，访问控制列表）来定义目录和文件的安全性[5] 。Windows 2000 开始引入了 NTFS 3.0。之前 Windows NT 4 所使用的版本叫 NTFS 1.2。NTFS 3.0 引入了安全性继承概念，这样，一个目录的安全性可以由子目录继承。由此在实践中也可以节省大量用于存储子目录安全性所需要的磁盘空间——子目录和文件只需要引用（而不是拷贝）父目录的 ACL 即可继承其安全性，只有当自己需要定义额外的安全性时才需要记录 ACL。

SE Linux 也支持 ACL。但许多 Linux 发行版默认安装不打开 SE Linux。然而，即便这样，它仍能实现与 ACL 接近的安全性设置。怎么做到这一点呢？ACL 指定了一组而不是单个用户/组对一个文件系统对象的访问权限。Linux 上一个文件系统对象可以属于一个用户，并属于一个组。另外，一个用户又可以属于多个组，那么方法就是把需要有访问该文件权限的用户加到中一个组，让文件属于这个组，再给予所需要的权限（读、写、执行）。但是，这样做的权限设置仍然受限于只有一个组能关联到一个文件这一事实，无法实现两个组两种权限的设置。

Windows Vista 引入了 UAC 和 UIPI 这两种新的安全特性。UAC 是 User Account Control（用户账户控制）的缩写。在 Windows Vista 里面，不仅普通用户（受限用户，通常属于 Users 组）运行于受限的权限下，连系统管理员启动的程序默认也运行在受限的权限下。这是什么意思呢？举例来说，Windows XP 中的管理员账户执行的程序都能写入 Windows、Program Files 等目录，而 Windows Vista 上，默认启动的程序是不允许这样做的。那么怎样启动的程序才能这样做呢？程序启动时，它本身或者它的启动者必须显式请求“权限提升”。比如，在资源管理器中右键点击可执行文件并选择“以管理员方式运行”。以这种方式启动程序后，如果当前账户是管理员，那么 Windows 会把桌面切换到 Winlogon 安全桌面，并提示用户确认。如果当前用户不是管理员，那么在切换到安全桌面以后，Windows 会提示用户选择一个管理员账户并输入密码以作确认。这种形式的确认之后，启动的程序就已经具有真正的管理员权限了。某些程序，如安装程序，需要运行在这样的权限下。详细内容请阅读《深入解析 Windows 操作系统（第六版）》第 6 章“安全性”。

UIPI（User Interface Privilege Isolation，用户界面特权隔离）是随着 UAC 应运而生的技术。有了 UAC 以后默认启动的程序不具备管理员权限，而要让程序得到权限提升来取得管理员权限，用户必须经过安全桌面来确认该行为。到此为止一切都是安全的（前提是该权限提升的应用程序并非恶意软件）。那么，接下来，该应用程序将运行在用户的普通桌面上。万一用户桌面上别的程序向这个具备管理员权限的程序发消息，岂不是不安全了？UIPI 正是用来解决这个问题的。UIPI 机制让管理员权限的应用程序在默认情况下受到完全的保护——非管理员权限的进程无法访问它的 UI，也无法给它发消息。管理员权限下的应用程序可以显式向 UIPI 指定它需要处理来自非管理员进程的怎样的消息。

UAC 和 UIPI 这种类型的保护，在业界称为“强制访问控制”（Mandatory Access Control；MAC）。它们通过把应用程序分在不同的诚信级别（integrity level，或者叫完整性级别），让更高“诚信”的进程不被更低“诚信”的程序随意访问，从而降低了更低诚信级别程序可能造成的破坏。除了 Windows 系统提供了这种保护之外，这一级别也被 Internet Explorer（7 或更高版本）用在了“保护模式 IE”功能里（当时由 Robert Gu 领导实现）。在 IE 的“安全”标签页，我们能看到“启用保护模式”选项。这个模式下，IE 的标签页代码是运行在“低”诚信级别

的，这样，一旦网页利用 IE 的漏洞攻击入系统，也只有很低的权限，无法修改用户文件，也无法修改系统。

继 Windows Vista 引入 UAC 之后，为了兼容在编写时没有考虑 Windows 安全模型的老的应用程序，UAC 提供了虚拟化支持。有些老版本的应用程序不仅在安装的时候会向 Windows、Program Files 等公共目录写入数据，而且在用户使用的过程中也会做这些事情。而在企业环境里面，或者从正式的安全性设置角度来看，普通用户是没有权限写入这些公共目录的。于是，当这些程序在普通用户账户下运行时，就访问不了这些目录，通常会导致程序不能正常运行。UAC 虚拟化了文件系统和注册表。通过 UAC 虚拟化，向这些公共目录中的写入会被重定向到用户配置文件目录下的 AppData\Local\Microsoft\Windows\VirtualStore 子目录中，从而在保护了老程序兼容性的前提下，隔离了不同用户的数据。

与 UAC 虚拟化有点类似、但功能强大许多的一个产品是现时流行的 Docker。其概念是“容器”。这个容器包装的内容有 CPU、内存管理、文件系统、运行中的进程树、用户 ID 和网络，提供这个环境让进程和线程运行。这个环境里的名字空间隔离（如进程树和文件系统）是通过 Linux 内核的 namespaces 功能来实现的。资源分配则是通过内核的 cgroups 来实现的。Wikipedia 上的 Operating System-level Virtualization 列举了包括 Docker 在内的多种操作系统级别的虚拟化技术。

Docker 这种级别的虚拟化，在微软对应到一个尚未发布的产品，叫 Drawbridge。MSDN Channel 9 上有关于 Drawbridge 的介绍[32] 。

另外值得一提的是，微软 Application Virtualization（App-V，应用虚拟化）的效果，也是为应用程序提供了一个虚拟的文件系统和注册表环境，使得应用程序不必被真正安装在目标机器上，就可以使用默认的设置来运行，大大减少了应用程序的配置损坏或不同应用程序互相之间配置冲突带来的问题。

作为对比，我们来看一下 Android 等移动操作系统的安全性是如何实现的。Android、iOS 都不允许全局的键盘钩子，也不允许应用程序在后台截屏，只允许显式的截屏和拍照，应用在没有得到授权的情况下也不能拨打电话和发送短信。默认情况下，Android 不允许一个应用访问别的应用的数据，除非那个被访问的应用显式地将数据指定为公开。从这些方面来看，Android 和 iOS 天生要比传统 Windows 安全得多，和新的 WinRT 处于同一安全级别。

在 Android 系统上，不允许键盘钩子、不允许后台截屏等功能都是通过 Android Sandbox（沙箱）来实现的。每个应用都运行在它自己独立的 Linux 用户账户下（虽然 Android 本身是一个单用户的系统，但却使用了多个 Linux 用户账户来运行应用）。应用程序通常通过 API 来访问系统功能，而这些功能最终都将转换为系统调用。系统调用会被内核代码处理。某些操作，如文件系统操作等，是简单的系统调用，都由操作系统内核代码直接完成。而某些操作，如拍照等功能，需要用到 IPC 之类的系统调用，它们会通过另一个协作进程来完成——这个时候就需要目标进程来判断，是否允许源进程做这样的操作。源进程如果没有相应权限，操作就会被终止。文件系统的访问权限，则是通过 SE Linux 机制，对文件和目录设置权限，以避免文件对所有应用的用户账户公开。Android 的总体安全实现就类似这样，它的沙箱对 Java 和原生（native）的应用程序都有效。详细信息，请参考 Android Security 一文[18] 。

当然，除了在上述方面的安全性的大大加强之外，Android 也有它的局限性。目前（以安卓 4.2 来说），它还缺乏 iOS 系统所拥有的“程序允许访问哪些功能”这样的控制——一个程序在安装时申请了所有功能；安装时和安装后都无法更改或取消访问这些功能的权限。另一方面，从文件系统来说，SD 卡上的内容是所有应用程序都可以访问的，所以，它缺乏 Windows 所拥有的自主访问控制（Discretionary Access Control，DAC）。它也缺乏安全审核机制（security auditing），也就是系统访问控制（System Access Control，SAC）。这样，我们既无法限制应用程序读写某些文件，也无法知道哪些应用程序读取了哪些文件。所以，我们还是要在 Android 设备上避免安装恶意软件（如只通过可信渠道下载软件），否则，如果有软件把我们的敏感文件读取并通过网络发送出去，我们也不知道。

从我个人的角度来看，一个安全的操作系统可以怎样设计呢？以前我读书时候接触了 Windows 95 和 Windows 98，发现这两个系统和 DOS 一样，对计算机病毒的抵御能力很弱。即便到了后来的 Windows XP 乃至 Windows 8，事实上对于病毒的抵御能力依然不强。Android 系统要好很多。而我在 Windows 98 时代的想法是，一则我们要保证操作系统本身无法被病毒感染。就好比现在的 Android 系统，其本身是不会被病毒所感染的，除非系统被 root，那是另当别论。二则要保证，不同的应用程序访问同一个用户的数据的时候，采用的是不同的角色，从而我们可以精确地控制哪些程序可以访问哪些文件或目录（读、写内容，删除，读、写安全性等），并可以对这些访问做审核（最好还可以方便地通过程序来过滤审核日志，以发现期望之外的访问）。对于一些特别敏感的数据，还要求应用程序向用户询问密码才能得到访问权。这个密码询问对话框需要通过系统来呈现，以避免应用程序伪造。为了识别出对话框的真假，用户可以通过一些手段，比如 Ctrl+Alt+Delete 组合键来确认这个对话框是系统提供的而不是应用程序提供的（有点类似于 Windows 的 UAC 权限提升对话框）。

上面的安全模型有很大一部分可以参考 Android 和 Windows。此外，Android 的安全性带来的一个不方便之处是，应用程序本身的目录，包括程序目录和数据目录，都无法被用户查看，其中的文件无法被复制出来。而我们如果要做一个更实用的系统，可以允许应用程序的目录被用户以只读方式查看和读取。但用户必须通过一个系统内置的程序来做到这件事。这样，应用程序互相之间就依然保有私密性，而用户则拥有了更大的权力来访问这些细节。应用程序可以再有一个安全存储区，用于存放一些不希望被用户直接看到的数据，如软件的证书密钥等，但是，系统最好能让用户看见这些文件占据了多少磁盘空间，以便用户根据需要来做系统的清理。

另外要讨论一个话题，就是现有的 Windows 系统怎样改变可以变得更安全？鉴于现有 Windows 系统需要兼容所有传统的桌面应用程序，同时又要运行最新的 WinRT 应用，Windows 也必须在安全性上提升一个层次，才能与 Android 和 iOS 相竞争。首先，驱动程序无法完全做到向下兼容——一些内核 ABI、API 的改变必然要求重新编译乃至重新编写设备驱动程序。但这一点在硬件厂商的配合下，应该还是不难做到的。其次，老的应用程序所受到的最大困扰无非是病毒、木马和流氓软件等恶意软件。自 Windows Vista x64 起，设备驱动程序的数字签名已经成为安装驱动程序时必要的元素，因此通过驱动程序引入恶意软件已经变得不那么容易。但从应用层面引入恶意软件仍然是比较容易的，原因在于所有非绿色软件都要求在管理员权限下运行安装程序，而只要安装程序中加载了恶意软件的成分并加以运行（比如通过安装一个 Windows 服务，或通过计划任务），这些恶意软件就可以侵占所有用户的桌面，还能影响其他软件乃至系统本身的稳定性。

为了对抗恶意软件，Windows 8 之中有两个功能可以使用：一个是 Refresh（刷新），一个是 System Restore（系统还原）。系统还原能将系统还原到之前一个时间点的状态，而不改变用户自己的数据，但它最大的不方便之处是，还原点到当前时间之间的所有已安装的程序，一经还原，全都会被卸载。刷新功能则更为激进，它是在保留数据的前提下把系统重装。所有 WinRT 应用的安装和数据会被自动恢复，但传统桌面程序将会是全部没有安装的状态。所以这两种手段并非确保 Windows 安全性的理想手段。

而且，Windows 桌面应用程序可以安装全局的键盘钩子，从而可以截获用户在浏览器等应用中输入的密码。这是当初为什么支付宝要推出安全支付控件的原因——通过它特制的驱动程序来获取密码，而不会被键盘钩子截获。Windows 桌面应用的这种全局性资源访问还体现在文件系统上面，比如前面说到的 UAC 虚拟化就是为了避免部分这种全局性访问的副作用。

我个人的想法是，可以通过指定哪些应用可以进行这些全局性访问，哪些程序不能进行这些全局性访问，以及访问的方式（比如某些应用可以全局读、局部写），来限制应用程序的权限。同时，必须对所有没有全局访问权限的应用程序进行虚拟化——当它们访问全局资源时，将它们的访问虚拟化，让它们不至于失败，又不让它们访问到它们不该访问的位置。这样一来，即使是需要全局安装的程序，也可以虚拟化为局部安装的程序。即使是流氓软件，只要是做了局部安装，一旦发现以后，也只需要将其强行卸载即可。

但是这里需要注意几个问题：一是如何识别单个安装程序。二是如何定义“局部”的范围。三是安装程序使用的方便性上的问题。要做到第一点，可以在沙箱中先安装一下这个程序。安装过程中可能有用户交互。根据程序的安装目录，可以区分不同的程序。其中会有很多细节需要处理和把握，比如有的程序会直接安装在 Windows 目录下，有的程序会安装到一个已存在的同公司软件的目录中。如果把同公司软件分开做局部虚拟化，虽然也可以运行，但使用体验上可能会有所不同。所以这个问题是个复杂的问题。

如何定义“局部”的范围呢？对于程序文件来说，“局部”可以是单个程序的目录（如 C:\Program Files\Tencent\QQ），也可以是该公司程序的目录（如 C:\Program Files\Tencent），而对于一些直接安装在公共目录如 Windows 目录的程序来说，可以是程序相关文件（如所有安装时被安装到 C:\Windows 目录下的文件）。但具体如何定，难道让“小白”用户来选择吗？用户已经够累了，不希望再为这些选择所困扰。所以这些信息，一种方式是通过 Microsoft 公司，像发放 Smart Screen 不安全站点列表一样，公开地通过离线或在线方式向 Windows 系统发放这些数据。对于已知的流氓软件，默认就将其通过“局部”方式安装即可。对于读取来说，除了上述程序文件之外，外加系统文件。对于用户数据来说，“局部”的含义可以是让程序写入所需要存储的非跨程序共享数据（叫做“应用数据”，application data，比如 C:\Users\Robbie\AppData\Roaming\Microsoft\HTML Help，就是 HTML Help 这个程序的应用数据）。“全局读”则外加能读取用户的个人文件夹数据，“全局写”也类似。

安装程序使用的方便性上的问题，前面略有提及，主要问题是，用户可能需要判断是“局部”安装一个程序，还是“全局”安装一个程序。另外，最好能把安装好以后的程序连同程序和数据打个包，下次重装系统时，只要一解包就行。但这样做有着很严格的前提，那就是系统不能暴露任何会随着系统安装的不同而不同的全局信息给应用程序。否则，应用程序一旦换了一个系统环境，可能运行不起来。这对于传统桌面程序来说可能比较困难，除非采用 App-V 技术。还有就是强制卸载一个程序——如果一个程序是“局部”安装的，那么可以强行把整

个“局部”都卸载掉。这应该不难做，只需要把所有虚拟化的局部文件和注册表数据删除即可。这样，无论是病毒还是流氓软件，都将很难有机可乘。

最后，通过资源管理器，应该能够访问所有数据，包括虚拟化和非虚拟化的数据。用户可以指定访问时的“虚拟化上下文”，来指定通过哪一个应用的“角度”来访问数据。

从实用的角度来说，这些设计无疑增加了系统的复杂度，无论是客户，还是技术支持，亦或是开发者本人，都不会喜欢这么复杂的设计。所以，最好的平衡，就是 Windows 8 现在走的路——用新平台 WinRT 运行新的安全模式下的程序，用老平台运行老的可信任的程序。对于最终用户来说，最好还能方便地安装上一个安卓虚拟机，运行安卓程序。这样就完整了。

文件系统

磁盘或其他形式的持久性存储介质是一台计算机用于保存长期数据的设备。这是计算机自身特点决定的。计算机的物理内存（RAM，Random Access Memory，又称随机访问存储器）无法在断电后保留数据。另一方面，目前还没有速度达到物理内存速度，又能在断电后持久保留数据的存储器（NVRAM）。在这些持久性存储介质上，文件系统就是里面最典型的数据结构。

在记忆中可追溯到的 80286 时代，5.25 英寸软盘（5 寸盘）是当时最流行的格式。后来又出现了 3.5 英寸软盘（3 寸盘）。5 寸盘有 360KB 和 1200KB 两种，3 寸盘有 720KB 和 1440KB 两种。软盘上使用的文件系统是 FAT12。以 1440KB 的 3 寸盘为例，它一共有 2880 个扇区，分别由 2 个面，每面 80 个磁道，每磁道 18 个扇区组成。每扇区有 512 字节。FAT12 文件系统占据了整个软盘，其中 1 个引导扇区，9 个 FAT 扇区，9 个备份 FAT 扇区（当两个 FAT 内容不一致时，可以用其中较新的一个恢复较旧的那个），14 个根目录扇区。其余空间用于存放文件和子目录。

FAT12 的元数据信息，比如它的 FAT 的个数、每个 FAT 的扇区数等信息都保存在引导扇区的 BPB（BIOS Parameter Block，BIOS 参数块）中。FAT12 的关键数据结构在于其根目录、子目录和 FAT。FAT 的内容就是一个链表，其中有许多条不交叉的链，每条链代表一个文件或一个目录在磁盘上的分布。每个链表表项在 FAT 中占据固定大小的字节数，如 FAT12 中，每个表项占据 1.5 个字节，也就是 12 比特，因此这种 FAT 被称为 FAT12。同时，每个表项又对应磁盘相应位置的簇。那么，有了文件的链之后，根目录和子目录里面的项就被用来表示文件的起始位置。这样就形成了整个磁盘的数据结构。

前面讲了整个软盘全部用于存储单个 FAT 文件系统。硬盘的容量比软盘大得多。为了让硬盘的使用更有灵活性，特别是为了多重启动、便于系统重装、隔离文件碎片等目的，人们可能会把一个硬盘划分成多个分区。每个分区可以包含一个文件系统。这样，一个硬盘上就可能有多个不同的文件系统，包括 FAT、NTFS、ReFS、ext2、ext3、ext4、reiser3、reiser4、zfs、btrfs 等等。

前面讲到分区表。分区表实际上也是一个链表的结构。在 MBR（Master Boot Record，主引导记录）格式的硬盘上。由于 MBR 格式最大只能表示 2TB 的硬盘，所以后来又有了 GPT（GUID Partition Table，GUID 分区表）硬盘，能支持超过 2TB 的硬盘。在我的主页上，我针对 MBR

的格式做了一些简介和讨论[19] 。

Windows 通常使用 NTFS 文件系统。该文件系统是自 Windows NT 3.1 被引入的。NTFS 自设计伊始就把许多重要特性纳入考虑范围，因此，不像 Linux 主流文件系统自 1991 年到 2014 年期间经历了 minixfs、ext、ext2、ext3、reiser3、ext4、zfs、btrfs 这样繁复的变化，NTFS 自 1993 年推出 Windows NT 之后，直到最近的 Windows 8.1 为止，都只有一些版本升级和功能改进，在设计上没有本质的变化。当然，2012 年微软设计开发了新的文件系统 ReFS[21] ，它的特点是，能用于特大容量的存储器，并能配合 Storage Space 功能，将许多磁盘组合成存储池，并以虚拟磁盘的形式展现给应用程序。

NTFS 具有以下特点：

1. 它使用了 64 位的对象指针，因此具有高度的可伸缩性。它的 1.0 版本是随着 1993 年第一版 Windows NT（NT 3.1）的推出而推出的。直到 2013 年 Windows 8.1，基本的结构依然保持不变。1993 年的时候，那时流行的硬盘大小也才 200~300MB，而 NTFS 已经超前地设计了最大 16TB 容量硬盘的兼容性(就具体的实现而言,在 Windows XP SP1 之前是支持到 128GB，自那个版本往后则支持 16TB)，并且单个文件的大小也能达到 16TB（在簇大小为默认值，即 4KB 的情况下）。
2. 可恢复性。NTFS 通过记录日志（$LogFile）来实现断电或系统崩溃时文件系统元数据的恢复，确保文件系统的一致性（有时会留下一些次要的不一致，但不再需要每次断电都运行 chkdsk 来检查和修复文件系统)。如果应用程序采用了 Windows Vista 及更高版本中的事务型文件系统，那么还可以确保断电时用户数据的完整性。
3. 安全性。前面已经说过，NTFS 支持基于 ACL 的安全性。
4. MFT（Master File Table，主文件表）的特性。首先，它具备冗余性。这一特性可以保证在 MFT 范围内出现坏块的情况下，依然可以保持文件系统能被访问。其次，NTFS 为 MFT 预先保留了尽可能连续的磁盘空间，这样当 MFT 扩大时，可以避免磁盘碎片。
5. 长达 32767 个 Unicode 字符的文件名。但是 Windows API 的 ANSI 版本并不允许这样长的文件名，而在 Unicode 版本的 API 中，必须采用\\?\表示法显式使用长路径；路径中的每个文件名或目录名也不能超过 GetVolumeInformation 函数返回的 lpMaximumComponentLength 所指示的长度（通常是 255 个字符）[33] 。
6. 性能。NTFS 的单个目录的存储是通过 B 树来实现的。因此，目录里的文件在按名称查找时，速度非常快，时间复杂度为 O(log n)。这对于单个目录包含很多文件的情况是很有利的。

NTFS 所有的元数据、文件和目录都通过两种存储方式之一来存储：一、MFT 记录；二、非驻留存储库（non-resident repository)。MFT 记录位于 MFT 之中，大小单位为 1KB（随着 4KB 扇区硬盘的流行，这个大小也可以是 4KB)，而非驻留存储库则位于 MFT 之外的簇中，大小单位为簇大小[23] 。

MFT 中的每一条记录，可能单独表示一个广义的文件（这包括元数据文件、目录，以及普通文件)，也可能属于表示某个广义文件的一系列记录中的某一条。NTFS 有很多元数据文件，包括$MFT（MFT 本身)、$MFTMirr（MFT 前 4 条记录的备份)、$LogFile（文件系统事务日志)、$Volume（卷信息)、$AttrDef（映射属性类型的数字标识到文字)、.（根目录，它是一个目录，不是普通的元数据文件)、$Bitmap（NTFS 卷的分配位图)、$Boot（引导扇区)、$BadClus（坏簇)、$Secure（ACL [访问控制列表]、安全描述符等安全信息)、$UpCase（大小写映射

表）、$Extend（扩展元数据的目录）、$Extend\$Quota（磁盘配额）、$Extend\$ObjId（链接跟踪信息）、$Extend\$Reparse（重解析点）。

MFT 记录有一个记录头，之后跟着各个属性。记录头里保存了一些信息，包括重用序列号、日志序列号（LSN，用于$LogFile 恢复机制）、更新序列号（USN）、更新序列、基本文件记录的文件引用（当这条文件记录是扩展记录的时候会用到，参见后面关于$ATTRIBUTE_LIST 的解释）、硬链接的数量。

在文件系统元数据中，包括目录以及$ATTRIBUTE_LIST（后面会讲）等，会用到一种叫做“文件引用”的 8 字节表示方式来引用 MFT 记录。文件引用包含一个 6 字节的 MFT 文件记录号和一个 2 字节的记录重用序列号。文件记录号就是 MFT 中以记录条数为单位的偏移量，例如，0 号记录就是$MFT。记录重用序列号是在每条记录被弃用并重用的时候，递增的一个序列号。它的目的是，当系统崩溃后重启进行恢复时，表示出文件记录是否已被更新，以便验证指向已删除文件的文件引用（包含老的序列号）是否指向一个已经被重用的（包含新的序列号）文件记录。

MFT 记录中的属性都有一个属性头，其中记录着属性类型码，另外可以有一个属性名。注意区别属性类型名和属性名。属性类型名是在$AttrDef 中定义的，与属性类型码相对应，而属性名则是在属性本身之内定义的，比如文件的可选数据流（alternate data stream）的名字就被存储为$DATA 属性的属性名。属性头还包含：属性大小、属性是否驻留、标志位。

对于驻留属性，还有属性长度、属性偏移量。对于非驻留属性，则包含：起始 VCN（后面会讲）、结束的 VCN、数据行串（data run）列表的偏移量、压缩单元的大小、属性分配的总大小（MFT 之外分配的簇的字节数）。属性的值则可以被抽象地看作是一个字节流（byte stream）。驻留属性的值是在 MFT 记录的属性内部表示的。非驻留属性是在 MFT 外分配的簇中表示的。注意，有些类型的属性是既能驻留又能不驻留的，对于具体的某一个属性，它是驻留还是非驻留则需要从“非驻留”标志位来判断。

NTFS 的目录是一个 B 树索引。在表示目录的 MFT 记录中有一个$INDEX_ROOT 属性。它总是驻留的。当目录中的项比较少时，所有的项都能被$INDEX_ROOT 容纳。$INDEX_ROOT 由一个根（root）信息和一个索引头（index header）开始，后面跟着多个索引项（index entry）。可以把$INDEX_ROOT 看作一个 B 树结点。每个索引项能表示索引值（文件名，以及一个指向 MFT 记录的文件引用）和结点指针（后面会讲到）。

当目录中的项多到无法在$INDEX_ROOT 中容纳时，NTFS 将使用一个叫做$INDEX_ALLOCATION 的非驻留属性，并在磁盘簇（也就是在 MFT 之外）为该属性分配空间。$INDEX_ALLOCATION 在 MFT 记录中的部分包含了这些簇的 VCN（虚拟簇号，亦即这些簇的相对位置）到 LCN（逻辑簇号，就是 NTFS 卷里簇的位置，相对于卷起始位置的偏移量，例如，引导扇区对应的 LCN 为 0）映射。

VCN 是什么意思呢？比如一个目录占了 3 个簇，那么 VCN 就会是 0、1、2。数据行串（run）指的是$INDEX_ALLOCATION 所采用的 run length（起始位置/长度）编码，也就是当连续的 VCN 对应连续的 LCN 时，只需要一条记录就可以表示了。比如 VCN 为 0 时，LCN 为 256，而 VCN 1 对应 LCN 257，一直到 VCN 9 对应 LCN 265。$INDEX_ALLOCATION 就会记录 VCN 0，LCN

256，长度 10。这样就能表示整个对应关系了。这里说的记录信息是理论上的，实际上，信息被压缩成了：头部（1 字节）、LCN 相对偏移量（动态长度，长度在头部指定）、VCN 个数（动态长度，长度在头部指定）。其中 LCN 相对偏移量在第一条记录中是绝对簇号，而在后续记录中则是相对于前一条记录中的 LCN 的偏移量。

目录的另一个属性$BITMAP 记录了所有簇的使用情况——可能为一个目录分配了簇，用来存放 B 树的结点，但是之后又释放了这个结点。目录未必要马上收缩，此时可以用这个$BITMAP 来表示哪些结点是被占用的，哪些是空闲的。

那么这些$INDEX_ALLOCATION 簇，每一个簇就是一个索引块，也就是 B 树中的结点，其中包含一个索引头，后面跟着多个索引项。B 树是一种平衡搜索树[28]，它可以有多层，除了根结点之外，其余的每层有多个结点。每个结点有多个子节点，父子结点间用指针链接。根据一个结点是否有子结点，可以把结点分为非叶结点和叶结点。根结点就是$INDEX_ROOT。通过分析 NTFS 的结构可以肯定，它是一个略有自定义的 B 树结构，因为$INDEX_ROOT 与其他结点的大小并不相同，它可能会小一点。除此之外，其他特性和 B 树一模一样。它肯定不是 B+树。欲知更详细的分析，请参见我的 ntfs_fact_analysis.txt 一文[31]。

不仅是目录，$Secure 也使用了类似的 B 树索引。目录的索引类型叫做$I30，它是作为$INDEX_ROOT 和$INDEX_ALLOCATION 的属性名出现的。有一次，我看见一个 NTFS 卷在系统重启之后，chkdsk 自动运行，检查出了一些目录数据的不一致性，就显示为$I30 index。

当$INDEX_ROOT、$INDEX_ALLOCATION、$BITMAP 等属性大到一定程度时，一条 MFT 记录将无法保存这些信息。类似的，文件记录或者其他类型的记录中，属性总和也可能大到在一条记录里存放不下。此时，NTFS 会创建$ATTRIBUTE_LIST 属性，用来表示扩展记录（extension record，相对于基本记录 [base record] 而言）。该属性保存了每条属性：类型码、本条记录的长度、属性名、起始 VCN（后面会讲）、文件引用。它会列出各种其他属性，但不包括标准信息（$STANDARD_INFORMATION）。如果$ATTRIBUTE_LIST 属性本身太长了，使得标准信息加上它超出了一条记录的总长度，那么该属性就变成非驻留属性，其内容将在 MFT 外的簇中分配。该非驻留属性的所有行串（run）映射都必须在当前 MFT 记录中表示。整个属性的长度默认情况下不允许超过 256KB，即 64 个簇，因此完全可以在一条 MFT 记录中表示。

$ATTRIBUTE_LIST 属性表示那些不位于本条 MFT 记录中的属性时，它会为目标属性每次在不同记录中的出现都保存一个表项。对于非驻留属性，“起始 VCN”值会保存该属性在那条 MFT 记录中所记录的行串的起始 VCN。比如，假设一个文件的$DATA 属性非驻留且特别长，它的第二段分配在记录 1000 中，里面保存了 VCN 2000 起始的值，那么就会有两条$DATA 记录存在于$ATTRIBUTE_LIST 中。第一条的文件引用指向$DATA 的基本记录，“起始 VCN”为 0。第二条的文件引用指向$DATA 的扩展记录 1000，“起始 VCN”为 2000。

NTFS 的恢复。NTFS 利用日志文件，来保证当系统因为某种原因崩溃或被强行关机时，文件系统在下次启动时能够自动修复。前面讲过，日志文件的名称叫$LogFile。其中保存了两个区域：重启区和日志记录。重启区分成两半，内容一样，都保存了一些上下文信息，它们描述了一旦系统立即崩溃，下次重启时需要从哪里开始分析恢复动作。日志记录有三种：重做记录、撤销记录和检查点记录。具体细节请参见《深入解析 Windows 操作系统（第六版）》第 12 章“文件系统”。

关于 NTFS 基于日志的崩溃恢复功能，《深入解析 Windows 操作系统（第六版）》中详细讲述了整整十页，但是其原理之复杂，让读者难以快速把握其脉络。为了容易理解，我们先设想一下，如果我们自己来设计这样一个文件系统，需要实现怎样的机制。我们从需求出发来考虑这个问题。首先，我们的需求是，如果系统在执行到某个状态时崩溃或者断电，下次重新启动时，文件系统要能够把磁盘上的文件系统元数据恢复到一个一致状态，这个状态要尽可能接近最新的状态。然后，我们的设计包括哪些数据：一个循环使用的日志，在实际应用中永远不可能达到上限的 LSN（64 位，在通常 NTFS 文件系统的生命周期中难以达到；比如说，需要几万年才能达到）。日志中有一个重启区域（实际上有两个，但至少得有一个）。每个磁盘写操作先在内存中执行，同时在内存中追加相应的成对出现的“重做”和“撤销”记录。每次 LFS（日志文件服务）需要向磁盘上刷新数据时，它会在内存中先追加一个 checkpoint（检查点）记录，然后通知缓存管理器，先把日志刷新到磁盘，再把元数据和文件数据刷新到磁盘。最后，把重启区域中的数据（包含检查点的 LSN）刷新到磁盘。

在这个基础上，所有文件系统的原子操作都被设计成“事务”。比如创建文件、删除文件、改名、扩展文件长度、缩短文件长度、设置文件信息、设置文件安全性等。每个“重做”和“撤销”记录都关联到一个具体的事务。在我们的假想设计中，我们可以为每个事务赋予一个“永不重复”的编号（比如 64 位长；这是除了 LSN 之外的另一个编号），在每条重做撤销记录中包含这个编号。同时，如果同一事务的多条记录能够按顺序串起来，无疑也会对恢复带来不少便利，因此可以用两个指针，把这些记录串成双向链表。此外，为了进一步确保数据准确性，可以在每一条记录上加一个校验和，在重启区域中也加一个校验和。

恢复时，先扫描重启区域所指向的检查点之后的所有记录，形成一个内存中的事务表。可能后面还有检查点（比如那个检查点被刷新以后，相应的重启区域数据还没被刷新，系统就崩溃了），但重启区域中的记录显然是更准确的，因为它是在元数据之后被刷新到磁盘的。然后，这些事务有的已经提交，有的还没有提交，对于已提交的事务，按照检查点之后的重做记录来依次重做这些事务，最后这些事务的状态就能达到提交时的状态。对于未提交的事务，按照这些事务的撤销记录来撤销修改，这些记录不必限定在检查点之后，因为整个事务都需要被撤销。甚至可以在重启区域中专门记录最早的尚未提交的事务的 LSN，以加速这一查找过程。可以画一张图来帮助理解：

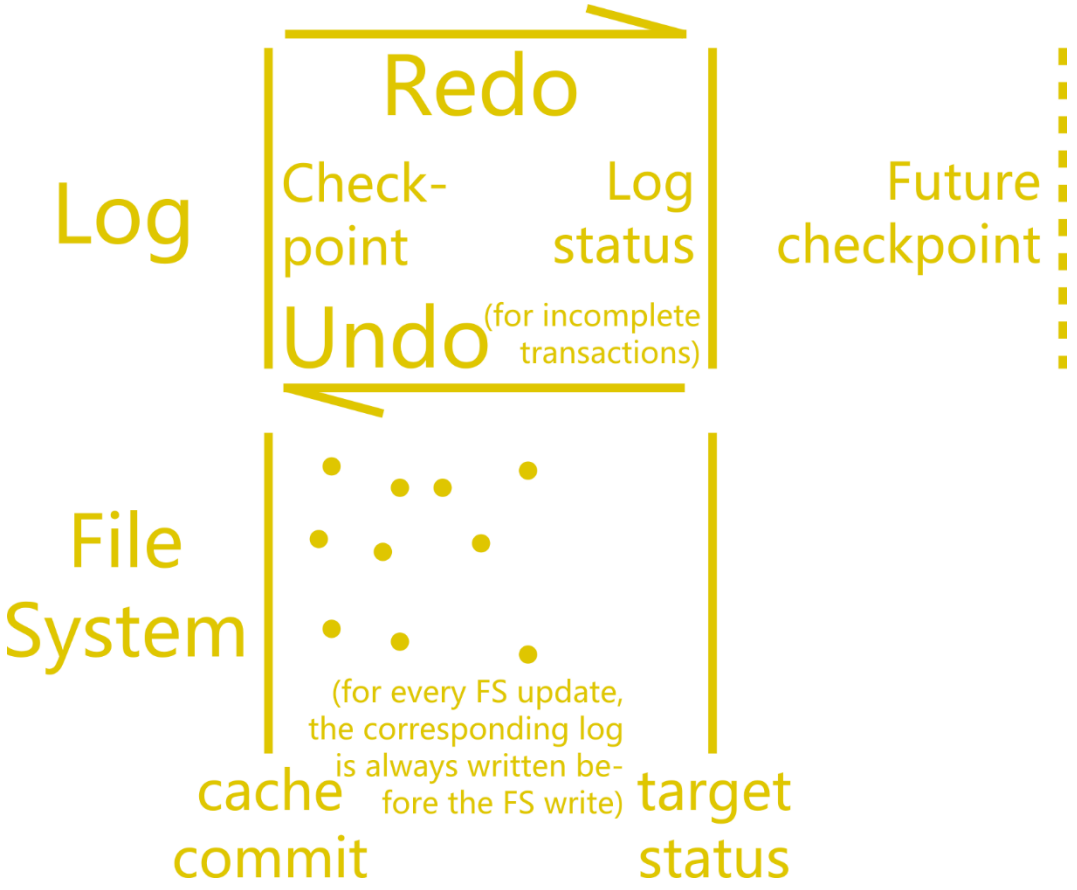

Figure 2 - NTFS recovery

和数据库事务一样，事务的原子性、一致性、隔离性和持久性（ACID）都需要被保证。持久性和一致性已经通过上述日志方式得到了保证，而原子性和隔离性则需要更加深入地在运行时加以保证。作为文件系统来说，这些元数据操作的耗时并不长，对于并行化的要求不高，我们可以通过在共享的数据结构上加锁来保证原子性。如果我们设计这样的一套锁机制，那么在操作一个文件的时候，比如扩展文件的长度或者设置文件的信息的时候，我们就在文件上加一个“独占锁”。但是，光这样还不够，因为，如果在操作过程中，文件的父目录或者父目录的父目录被删除了，这是会引起问题的。所以，我们还需要在所有的父目录路径上加一个“独占意向锁”（参考 SQL Server 的用法）。这样就可以实现原子性和隔离性了。相应的，元数据在被读取时，共享锁和“共享意向锁”也是必要的。

由于 NTFS 只在日志中记录了文件系统元数据的修改，而没有记录文件数据的修改，其实在恢复过程中，如果不加以判断，直接执行重做或撤销记录，是有可能覆盖文件数据的。举个例子，比如在系统运行的状态 A 中，元数据占据了簇 02、03，文件数据占据了簇 01、04；之后的状态 B 中，元数据占据了簇 01、02、03，文件数据占据了簇 04；状态 C 中，元数据占据了簇 02、03，文件数据占据了簇 01、04；状态 D 中，元数据占据了簇 02、03、04，文件数据占据了簇 01。假设在状态 D 发生了崩溃。检查点指向状态 A。但实际上，可能缓存已经被部分刷新为状态 D。如果直接重做，那么就可能把簇 01 改写，这样，文件数据可能被覆盖。解决方法之一是，在系统运行的情况下，每次都在所有数据（包括日志数据、元数据和文件数据）被完全刷新到磁盘上之后，才告诉用户进程刷新完毕。这样，用户进程将不能假设在数据被部分刷新的情况下，磁盘上的数据仍是一致的，从而保证了文件系统对用户进程的约定。不过，这种情况会导致元数据信息泄露，后面会讲一个具体的例子。

如果一个文件被删除，然后该文件被删除前所占据的空间被重新分配，又被写入了数据，此时系统崩溃了，怎么办呢？这种情况对于应用程序来说也没有问题，因为这三个操作是按顺序执行的，在其中任何一点上日志刷新都是没问题的，只有当重新分配空间的事务提交并刷新到磁盘上之后，数据才会被写到磁盘上。但是，有另一个问题，如果实际数据还没有被写入就发生了崩溃，那么所分配的磁盘块将包含原有位置的旧数据，如果这些数据属于另一个用户，那就发生了数据泄露。这将导致安全性问题。于是还必须保证，当分配磁盘块的“扩展文件”事务的提交记录被写入日志前，磁盘块必须被清零。

还有更微妙的一种情形：有一个目录，先在其中创建一个文件（于是有条日志记录写入它的目录项），然后，删除这两者，最后扩展另一个文件，这个文件占据了被删除的目录的空间。如果在日志被刷新到磁盘上之后，元数据和数据被刷新到磁盘上之前，系统崩溃了，那么重启之后将会重做元数据操作。问题是，这将导致最后的那个文件的数据里含有那个创建过一个文件的目录的元数据，这无疑也是一种信息泄露。想一想，除了前面讲的分配空间之前必须清零的方法外，这个问题还可以怎样解决？（提示：答案在 Stephen Tweedie 的演讲里。）

其他文件系统的实现可能不同。比如 Stephen Tweedie 设计开发的 ext3[27]，它只实现重做日志记录。这是因为它只把已完成的事务（对应于 NTFS 中的已提交事务）所修改的元数据或文件数据扇区刷新到磁盘（NTFS 则会既刷新已提交事务的数据，也刷新尚未提交的事务的数据）。需要注意的是，ext3 的“提交”指的是批量事务更新到磁盘的过程，因此它对应于 NTFS 的“刷新”或“检查点”，而 ext3 的“事务句柄”才对应于 NTFS 的“事务”。

ext3 的删除操作非常 tricky：由于一个删除操作可能涉及大量磁盘块上的元数据（这个和 NTFS 的数据行串不同，行串总是紧密地存储在连续的空间里的，而删除单个文件时，文件碎片再多，涉及的日志数据最多不过 MB 级别），因此日志可能存不下所有的元数据更新。那么怎么办呢？ext3 会把所有这些数据组织到一个叫做孤立文件列表的结构，并把这种复杂的删除操作分解为多次缩短文件长度和最后一次删除文件的事务日志，同时在操作伊始就把该文件加到孤立文件列表中。如果下次重启时，发现孤立文件列表里有该文件，就继续把它删除，无论当时缩短到什么程度，都可以确保完成。

ext3 的日志不仅支持文件系统元数据的恢复，也可选地支持文件数据的恢复。

NTFS 支持透明压缩，也就是文件可以被直接压缩，压缩后仍然能和普通文件一样被程序直接读写。它所使用的压缩算法是基于 Lempel-Ziv 算法 LZ77 的变种，LZNT1。然而，需要注意的是，NTFS 的单个文件的碎片个数不能超过 1.5M（也就是最坏情况下，150 万个 4KB 的簇，就是 5.7GB 左右）。随之而来的一个问题是，透明压缩的文件经常会有很多碎片，平均 16~32KB 左右就会有一个碎片，因此透明压缩的单个文件超过 10GB 的时候就要小心了，弄不好就会达到文件系统限制值。这是我有一次把一个服务器共享目录上的 10GB 的 VHD 文件压缩以后，一位微软总部的工程师告诉我的。

其实，Linux 上的文件系统发展也不是一帆风顺的。在 Linux 开发的早期，于 1993 年，Linux 内核引入了两个文件系统：Xia 和 Ext2。最初 Linux 的文件系统采用的是 MinixFS。Minix 文件系统是 Minix 操作系统附带的文件系统。它有很多限制，比如文件系统最大只有 64MB，文件名最多 14 个字符等等。ext 则打破了这一限制，文件系统最大可达 2GB，而且支持 255

字符的长文件名。当时和 ext 竞争的另一个 Linux 文件系统是 XiaFS，后者在 MinixFS 的基础上予以扩展，但仍保留了 MinixFS 的诸多限制。结果就是 ext 文件系统以最终的流行而胜出。后来的 ext2 文件系统进一步打破了 ext 文件系统的限制，最大的分区大小支持到 4TB。ext3 则在 ext2 的基础上不需要转换原有格式就支持了恢复日志。

ReiserFS 以其能非常高效地存储小文件而著称，但又由于它比较容易将磁盘上的文件碎片化而被批评。ReiserFS 的设计师 Hans Reiser 是美籍德裔。2006 年时，他原本已经在开发 ReiserFS 的新版本——Reiser4 了，里面将会用到一种叫 Dancing Tree 的 B 树变种，能更有效地处理小文件的存储，同时不影响系统性能。Reiser4 也将支持事务、透明压缩、插件等多种高级功能。但是，因为他谋杀妻子 Nina 的事情东窗事发，于 2008 年他被捕入狱。然后，Reiser4 的开发也就变缓许多，尽管仍有人坚持在做这个项目。不过，近年来新出现并逐渐流行起来的 SSD 却又不怕文件碎片，因为它本身就是能够被随机访问的半导体集成电路，这未免对 ReiserFS 是一种鼓舞。另一个好消息是，比 Reiser4 更晚研发的 btrfs 文件系统从 Reiser4 的设计中吸取了好几点精髓，而 btrfs 已经被纳入 Linux 内核的主流版本（main stream）。希望这件事能让狱中的 Hans Reiser 开心点吧。

Windows 里面，如果要编写第三方文件系统驱动程序，可以通过 Installable File System（可加载文件系统，IFS）来实现。这些驱动程序和 NTFS 一样，必须运行在内核模式。这使得只有 C 语言等不依赖于用户模式运行环境的编程平台才能被用来编写这些驱动程序。另外，Linux 则在 2005 年的 2.6.14 内核引入了 FUSE（Filesystem in Userspace，用户空间的文件系统）。有了 FUSE 之后，就可以编写用户模式下的文件系统驱动程序了。这提供了很多可能性，比如编写者可以不需要懂很多内核编程知识，文件系统可以与用户模式的其他程序交互，可以访问 Web Service，甚至，只要把文件系统接口封装好了，就能用 Python 等其他语言编写文件系统驱动程序了。

NT 4.0 有一种特别的防碎片文件分配策略。在 Windows NT 4.0 等早期版本的 NT 系统中并没有内置的磁盘碎片整理程序。在 Windows NT 4.0 的开发过程中，有一家名为 Executive Software 的英国公司与微软合作，为 NT 系统添加磁盘碎片整理的支持。说来有趣，做磁盘碎片整理的厂商很多，光是当年的大公司就有赛门铁克之流，更何况如今还有 VOpt、PerfectDisk、JKDefrag、SmartDefrag、Whitney Defrag、Puran Defrag 等多家竞争对手，为什么唯独 Execsoft 却成了这么重要的合作伙伴了呢？其肯定有一些其他的历史背景，但有一点事实不可忽略：Execsoft 早就研发过 VAX/VMS 小型机操作系统上的磁盘碎片整理程序，而其他的厂商大都不是这一路的。而 NT 的首席架构师 David Cutler 之前就是 VMS 开发团队的领军人物之一，而且，据 David Solomon 所说，NT 和 VMS 有着很大的相似性，由此可见 NT 和 VMS 是有着深厚渊源的。

于是，这一渊源促成了 Execsoft 和 Microsoft 的合作，他们开发的 Diskeeper 就成了一个重要的磁盘碎片整理产品。Windows 2000 自带的磁盘碎片整理程序实际上是 Diskeeper 的精简版。后来，Execsoft 总部前往美国，并更名为 Diskeeper Corporation。再后来，由于 SSD 的兴起，磁盘碎片整理的市场需求变少，公司就推出更多其他产品，如 Undelete、HyperFast、V-Locity 等，同时公司也更名为 Condusiv Technologies，以便推行更加面向企业 IT 的产品战略。

而 NT 4.0 为了预防文件碎片，它将文件尽可能均匀地分布在磁盘的各处，让每个文件的尾部都有一段连续的空闲磁盘空间。可以观察下面两张图片，两个系统都是虚拟机上新装好的

系统，每个系统都有数以千计的系统文件，其中有许多几十到几百 KB 的小文件。NT 4.0 的帮助手册中提到了哪些 DOS 命令有 NT 版本，哪些没有，而这里面就说明了 defrag 命令是不存在于 NT 中的。旁边写了一句附注：“Windows NT 自动优化磁盘使用”。我想，这句话也许就是指的这个预分布特性吧。

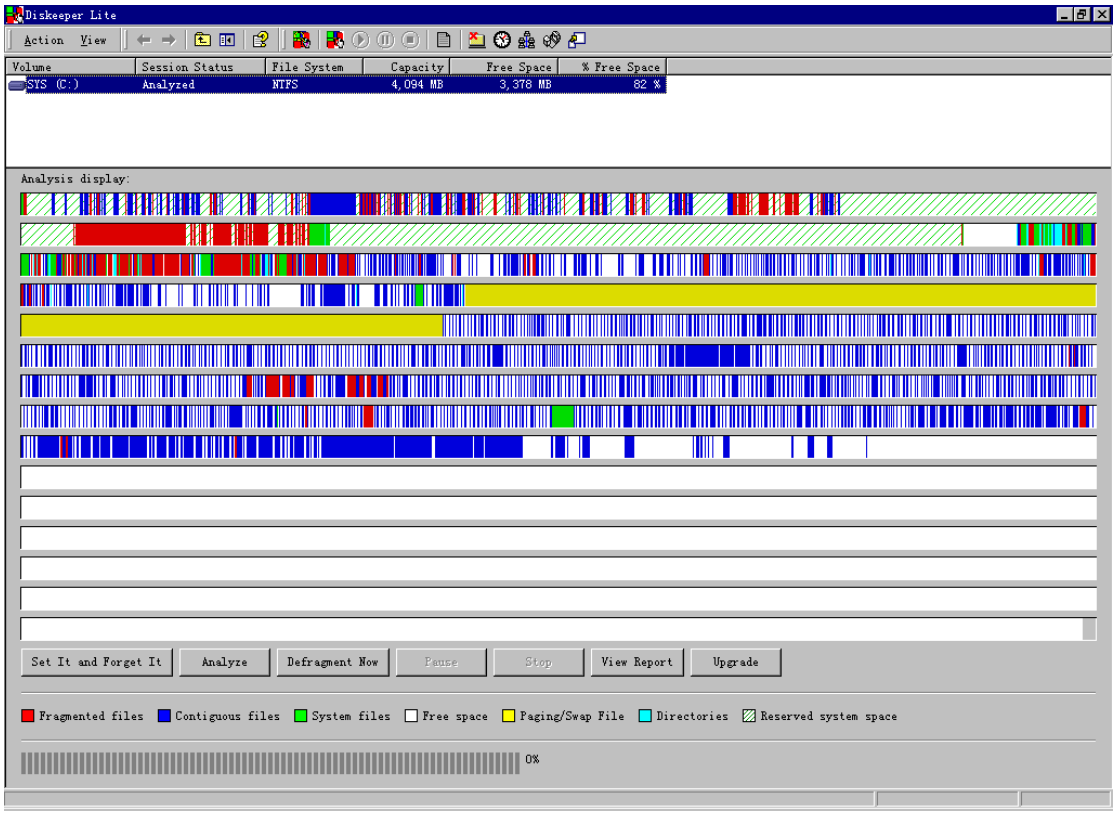

Figure 3 - Diskeeper showing a 4GB volume in Windows NT 4.0

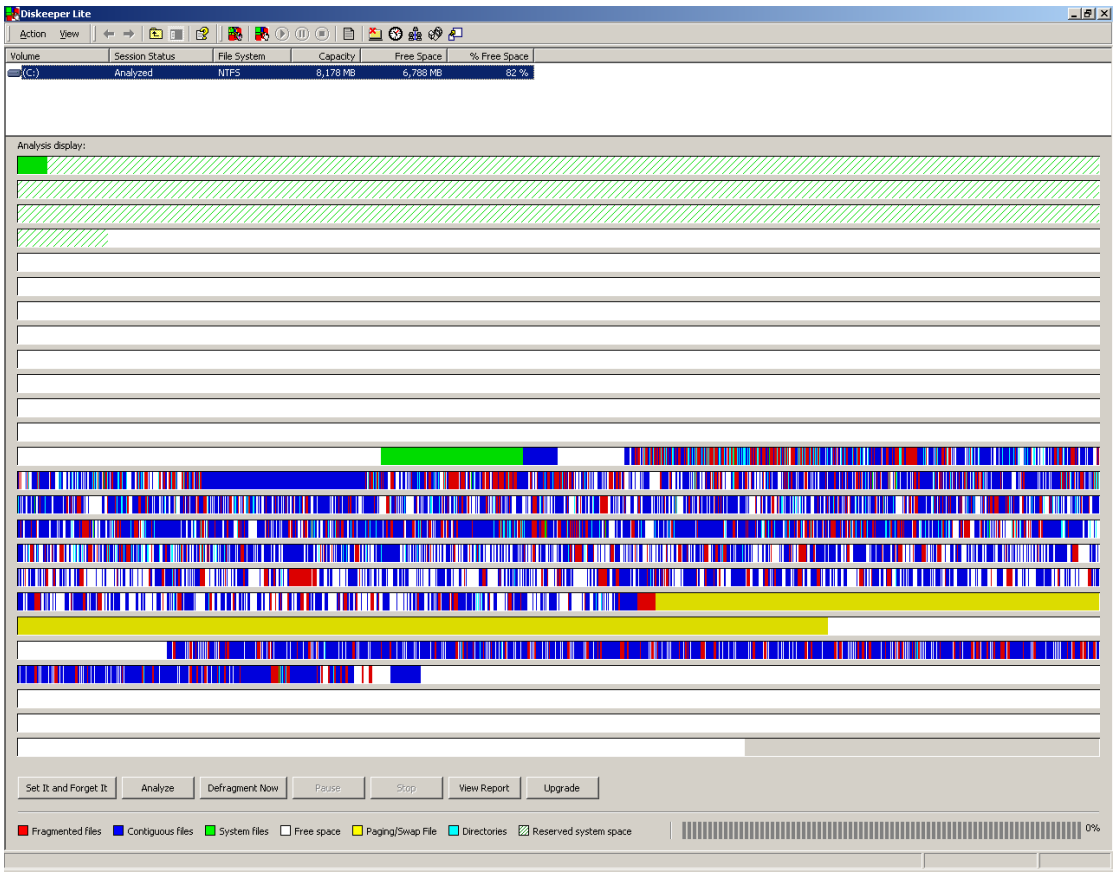

Figure 4 - Diskeeper showing an 8GB volume in Windows NT 2000

文件系统的功能方面，Windows Vista 引入了符号链接——终于，在 Windows 上要集成各处的文件资源虚拟成一棵目录树不再是难于上青天的事了。另外，Windows Vista 还提供了事务型文件系统和注册表，只要应用程序有需求，就能让文件数据和注册表数据都被事务化处理，以保证数据操作的 ACID 特性。

Windows 和 Linux 风格上的差异

所有的软件，它们在经历了真实生活的风吹雨打之后，都会逐渐成长起来。作为一个操作系统，由于需求因素，诸如向下兼容、有无相关编程需要等原因，其编程接口的一致性未必总能不断提高，但灵活性却往往逐渐提高。Windows 也不例外。

一直有一种说法，就是 Unix 操作系统有一个统一理念——所有对象都是文件。Windows 不具备这一理念，所以接口繁多。其实虽然 Unix 的接口相当一致，但实际做操作的时候，也必然需要精心设计才能有好的效果。比如在/proc 文件系统中，可以把进程的信息直接像读文本文件一样读出来，很方便，但是，如果要做成一个管理工具，则必须把这些文本好好地存到一个结构体中，然后再在 UI 上妥当地显示出来。而另一方面，Windows 也未必就没有统一理念。比如，DeviceIOControl 就是与所有设备驱动程序通信的统一接口，包括与文件系统通信都用它。

当然，由于需求不同，灵活性的提高程度也会不同。比如，Linux 需要适应从小到嵌入式设备，大到大型机的所有规模，所以它的灵活性要求必然会比较高。Windows 的策略有所不同。

嵌入式和移动设备上运行的是 Windows CE 和 Windows Mobile。

据我个人在 2005 年的经验（完全只是经验），Windows 的平衡集管理器每一秒把数据刷新到磁盘上，而 Linux 默认是 5 秒且可配置。因此 Linux 对硬盘的写操作要比 Windows 要少。另外，Windows XP 所允许的已修改页面数量相当少，而 Windows Vista 和 Server 2008 则要动态得多，这使得某些时常要写大量数据到文件的程序跑得更快。

虽然 Windows NT 早在 1993 年发布时就有了相当具伸缩性的前瞻性设计，但是，随后的好多年，其内核都相对没有在性能方面做大的优化，直到 2001 年开始研发的 Windows Vista 才引入了较大的变革，即便如此，Windows 在 I/O 方面的优化还是与 Linux 有差距。Windows Vista 的磁盘 I/O 优化过于偏向吞吐量。

假定你用 VMware 创建一个虚拟机，并为其虚拟硬盘立即分配空间（而不是采用自动扩展虚拟硬盘），那么就需要在磁盘上大量写入数据，比如 40GB 的数据。假定你用的物理磁盘是机械硬盘，此时 Windows Vista 会优化吞吐性能，尽量保证这个连续写入足够快，但这会引起响应性能的损失。机器会变得十分卡。

Linux 则不然，无论是 Elevator、Anticipatory 还是 CFQ，这些调度器都注重 responsiveness（响应性能）。Elevator 调度器就是操作系统教科书里常见的“电梯算法”调度，它是充分照顾了硬盘磁头移动的物理特性的调度器。它的性能不错，但是公平性欠佳——靠近中间的请求，磁头每来回一次都能得到两次服务，而处于边缘的请求有较大概率每次磁头来回只能得到一次满足，造成的后果就是当多个线程读写时，那些只读写靠近磁盘中间的文件的线程会跑得更快。

Cyclic elevator（循环电梯）是电梯算法的改进版本，它会一直沿着一个方向进行扫描，直到满足了所有前方的请求，然后直接返回，不在返回途中满足其他请求，直到返回到最靠后方的请求之后重新向前走。它的速度比电梯算法略慢，但公平性得到了保证。

Anticipatory 是“期待式”调度器，由 Nick Piggin 设计，它会在收到一个请求后，等待一小段时间（比如几个毫秒，一般这个数值要小于磁盘的寻道时间才有意义，否则就容易等得太久），期待有新的请求，此时会有一定概率得到一个附近的请求（比如，刚才读取数据的线程做了一下计算，然后继续读取下一段数据）。然后，看刚才有没有得到非常靠近的请求。如果有，就满足那个请求。否则，继续按照循环电梯算法前进。Andrew Morton 的性能测试结果表明，在通常的工作负载中，Anticipatory 会带来一些性能改进，节约的时间比例通常在 0%到 50%之间，但也有变得更慢些的。

CFQ 则在后来融入了 Anticipatory 的思想，并且它本来的设计就有一些更高级的功能。比如它支持 ionice（也就是 I/O 优先级），并为每一个进程准备一个 I/O 请求队列。每个请求队列被赋予各自的时间片，以使得不同请求队列中的请求互相都能得到公平的服务。

Windows Vista 的 I/O 系统也提供了优先级机制。我曾经参与微软 MadDog 项目时，遇到的一个 LuaTools 权限提升（是指 UAC elevation，不是指 privilege escalation）带来的自动化测试 bug，就是和 I/O 优先级有关。当时 MadDog 相关的某个组件在 Windows 登录时自动运行，并利用 LuaTools——一个测试工具来对实验机监视器的可执行程序进行无人工干预的权限

提升。权限提升之后，I/O 的速度却异常缓慢。据了解，这是因为 Windows Vista 在登录时，资源管理器将那些自动启动的程序的 I/O 优先级设为“低”，以减少对系统本身 I/O 的影响，在一分钟之后再把这些进程的 I/O 优先级设回来。没想到，LuaTools 把那个新进程的权限提升了，新进程继承了源进程的 I/O 优先级。而资源管理器本身又没有管理员权限，因此无法把它的 I/O 优先级设回来。最后的解决方案是，等待一分钟以后再用 LuaTools 提升权限。

前面说起过 Windows 的磁盘缓存有预读优化。而 Linux 的磁盘缓存的预读优化也比 Windows 考究。吴峰光博士在 2007 的论文《Linux readahead: less tricks for more》[29] 就提出了一个重要的简化 Linux 预读机制并提升其算法灵活性的方案。该方案非常简洁，这将允许将来的算法改进变得更加容易。并且，它对原预读机制所未能解决的“交织读”提供了一定的支持，为将来的改进提供了启示。在吴峰光的 2007 年 Linux 文件预读的 PDF 中[30] ，更是分析了很多种典型的预读优化所面对的场景：交织读、串行与并行矩阵运算、稀疏顺序读、文件服务器响应乱序网络请求的乱序读、顺序+随机混合读、聚簇随机读等等。有兴趣的读者可以进一步阅读这些资料。

不要说 Windows 8 为了兼容传统桌面程序而不如安卓安全，或者 Windows 8 的 RT 应用程序不够丰富而不如 iOS。其实 Linux 也有好多照顾向下兼容性的时候。UTF-8 是 Plan 9 系统研发过程中，Ken Thompson（Unix 之父）和 Rob Pike 发明的，它就是为了让好多不知道多字节字符集的应用程序能兼容而存在的。UTF-8 让应用程序得到的懒惰程度甚至比那些支持 GB2312 的 MS-DOS 程序都高。为什么呢？

UTF-8 要求每个低于 128 的非扩展 ASCII 字符直接就等于其 UTF-8 编码，这个和 GB2312 一样。但是 UTF-8 要求所有其他 Unicode 字符用大于等于 128 的多个字节表示，并且，首字节的范围是 0xC0 到 0xFF，后续字节的范围是 0x80 到 0xBF，整个字符的字节个数由首字节二进制最高位连续的 1 的个数表示，所以一个字符里面的真子字节串不可能成为另一个字符。这一举避免了 GB2312/GBK 的兼容性困境——许多程序，例如 strstr 函数，都不需要修改，更不会有 GBK 那样一个繁体字里面会隐含着一个反斜杠字节（路径分隔符）的窘境——这对于 DOS 程序是很致命的。

另外，GBK 编码下，在字符串中指定位置向前查找一个字符的函数_mbsdec 最坏情况下所需的时间复杂度是 O(n)。这个在 GB2312 下都是不需要的（尽管 GB2312 支持的中文字符相比 GBK 要少得多），而 UTF-8 更是直接避免了这个问题——可以确保 O(1)的操作。

Go 语言的三位发明人，Ken Thompson 和 Rob Pike 就是其中两位。Go 语言也是相当具有前瞻性的。Google 虽然不以桌面软件见长，但是 Go 语言却是其重磅炸弹之一，这一服务器领域的编程语言虽然不可能完全取代 Java 和 C#的客户端地位，但足以通过其高效、实用、超高并发等特性在高负荷服务器编程市场上引起大的震动。不过，个人感觉 Go 语言的 Simple Stupid 程度比 C 要差，比如不带括号的 if 语句反而带来不少困惑，另外初始化列表里最后一项的逗号要求不加，灵活性不如 C 和 Python。要是有 Dennis Ritchie 参与优化语法就好了，可惜他老人家已经仙逝多年了。

趣事

话说，我在翻译第 6 章的时候，为了利用上下班在地铁上的时间，我是在 Android 手机上用

终端版 vim 和百度输入法（五笔）做初译，然后放回电脑上自审的。有次参加一个活动，正好在地铁上遇到我的一个在微软工作的同学，我跟他说，你看，我用 Google 的 Android 系统给你们微软撑市面，不错吧？他笑了。

不过等到 2013 年接近尾声的时候，我的翻译也剩最后一章，此时我却已经购买了一台微软自家的产品 Surface RT，加上一个雷柏的带触摸板的键盘来辅助我的翻译进程。不过，还是有一些工作必须在 PC 上完成，特别是矢量图的处理，和一些实验程序的运行。但是，Surface RT 在我上下班乘坐地铁的时间里还是帮了我的大忙。

书里的实验用到了各种工具，其中做内核调试用的就是 WinDBG。说到 WinDBG，有一位人物不得不提。那就是 John Robbins。曾经的 NuMega 员工，WinIce/SoftIce 开发者，历年来 MSDN 杂志 Bugslayer 专栏的作者。后来自己开设一家咨询公司为别人讲授如何调试程序，如何避免 C/C++编程中的错误，以及排查程序中的疑难 bug。我曾在微软工作过一段时间，当时有一位同事对我很好，那次 John 到微软来讲 Debugging C/C++ programs in Windows，那位同事特意提醒我向经理申请参加。于是我有幸参与，除了听到了三天宝贵且昂贵的课程（数千美元）之外，也一睹大侠尊容。

John Robbins 讲的精彩课程：Windows 中的调试。他曾开玩笑地说：我 20 年前就干这一行了，那时候你们还没有出生呢，哈，肯定是这样！还有就是 WinDBG 中的 bang command。在《深入解析 Windows 操作系统（第六版）》这本书里，多处出现 WinDBG 或 KD 的命令，其中有些命令是以感叹号开头的，这些命令是调试器扩展带来的命令。John Robbins 就把它们叫做 bang command，嗯，就是 The Big Bang Theory 里面的这个 Bang！更多有关大侠的趣事，请参见我在 CSDN 上的个人博客中的文章[7] 。

话说，高博同学非常热爱技术翻译和技术教育。他做出了好几本高质量的翻译作品，包括 The C++ Gotchas（《C++：99 种常见错误》）、The Design of Design（《设计原本》）、The Information（《信息简史》）。这次 Windows Internals 也很荣幸得到他的贡献。他在计算机方面兴趣广泛，他早期比较有趣的就是高中时用 Pascal 写了一个 shell，长达一万多行，并且在高三毕业前用 Word 6.0 写了长达 50 页的 DOS 介绍文章。但当我对他说，“你好牛啊，写了个 shell 出来。”他却谦虚地说，“当时写代码完全没有章法，好多复制粘贴。”

当我发现 2005 年的台式机已经再也无法安装 NT 4.0（安装执行到某个步骤就蓝屏）时（此事实在 2008 年单位的电脑上，我给高博做了实验演示），我为了怀旧，也会偶尔在虚拟机里安装一下 NT 4。在我的 NT 4 安装里，我通常会把我在 1997 年到 2004 年之间收集的软件往里面装一装，体验一下那时的感觉。这个对于现在的 90 后乃至 00 后来说，够怀旧了吧。但是，高博同学更进一步——我不仅见到过他把 Windows 7 安装在 MacBook Air 之上，更有把 Word 6.0 安装在 Windows NT 3.51 上，并通过图形版 telnet 上交大 BBS。嗯，这个图形版 telnet 只有在 Windows 95、98、NT 3.x、4.0 上面才有；后面版本的 Windows 都只有命令行的 telnet 了，你如果明白我在说些什么的话。

不过高博在某些方面却比我“高富帅”得多。2008 年有一次在单位，他要做一张 DOS 启动光盘，用来刷系统 BIOS。也许是刻录程序参数设置错误的缘故，反复刻了好几张光盘，没一张能用的。我看他这样下去可不是一盒光盘全都浪费掉了！于是我上前用 U 盘做了一张 DOS 启动盘（好像是用惠普的某写入工具），交到他手上，帮他节省了剩下的半盒光盘。其

实在 U 盘容量越来越大的今天，这是很不错的方法；不仅是 DOS 启动盘，有时我安装 Linux 也从 U 盘启动。

在单位里，除了工作之外，与高博还有一些特别的合作。有一次，和包括他在内的三个同事一起为了一个单位内部项目加班到凌晨 1 点，办公楼的大门已关，四个人被锁在楼里。保安室里也没有人。只能打电话给已经回家的保安求助。等候期间，高博来了一个 Steve Ballmer 卖 Windows 1.0 广告的模仿秀：How much do you think this advanced operating environment is worth? 500? 1000? Even more? No! It's just 99 dollars. 然后我就指着我的文曲星，说，it's just 99 dollars！高博于是给我拍了张照片，这张照片后来就成了我 Flickr 相册的封面。

在上册翻译的过程中，他多次和我一起在淮海路逛万得城电器城，以及带着 Thinkpad 边做翻译边喝茶，也使我有了不少灵感，让我更好地完成了上册的翻译。他还请我吃了两次正宗的上海本帮菜，并开着他的小汽车带我和凌杰大侠一起兜风，从闵行到南京东路苹果店，还有外滩。这些都成了我人生中的有趣经历。

和高博同学的故事还有好多，说也说不完。总之他是我很要好的朋友，和他合作的几个项目都很愉快。他博学多才，和他交流，无论是物理、数学还是计算机，乃至文学、外语，他都能讲得头头是道，时不时能和他碰撞出思维的火花。

谨以此文感谢：潘爱民老师——没有潘老师就没有这本书的中文版，这本书的翻译有一半是基于潘老师的第四版内容，而且，所有工作的完成，离不开潘老师的鼓励和支持。

高博——是他向潘老师推荐我，让我有了这次宝贵的机会，十分感谢他。在与出版社的沟通方面，他做出了很大贡献。

本书也献给我默默无闻的父母，没有他们作为坚强后盾，我根本无法以这样的速度完成本书的翻译。

本博客献给我的亲爱的太太杨爽——在她的鼓励下，才让这篇长篇大论的博客成文。否则将长期被我束之高阁。

另外，本文也献给一位曾在上海群硕交流过的姓龙的朋友——我曾经在面试时与他进行过交流，他当时讲起了他对 Windows NT 技术的兴趣，讲到他读过这本书。这样的朋友并不太多——希望第六版也能给他带来乐趣和动力！

微软的宗旨是，发挥客户的潜力。那么，在此我也预祝这本书能让读者您也发挥您的潜力！

有关本文，如果你有什么问题，请给我发邮件，邮箱地址是：范德成（的拼音） at gmail.com。

Your potential. Our passion.

范德成
2015 年 3 月 3 日于上海

作者简介

范德成，2004 年毕业于上海交通大学。英文名 Robbie Fan，网名 Robbie Mosaic。在微软和 SAP 公司有多年项目经验。对操作系统、数据库、算法与数据结构、计算机体系结构、程序设计语言、软件测试方法学等方面具有浓厚兴趣。对 Python、Firefox、Linux 等开源技术感兴趣，并构建了微型开源项目 Robbie's Shell for Windows。

参考资料：

[1] 范德成（2012/12）：计算机软件的硬件支持
http://blog.csdn.net/r_mosaic/article/details/8298793
[2] 范德成（2012/07）：Windows 2000 SP4 内核调试初窥
http://blog.csdn.net/r_mosaic/article/details/7772644
[3] 范德成（2012/05）：Windows Vista: Cached Reading of a File Opened with No Cache Enabled
http://blog.csdn.net/r_mosaic/article/details/7581541
[4] 范德成（2012/04）：ReadyBoost--Robbie 的基准测试
http://www.fandecheng.com/archives/430
[5] 范德成（2011/10）：NTFS 访问控制列表
http://blog.csdn.net/r_mosaic/article/details/6840176
[6] Pascal Zachary 著，张银奎译（2009）：《观止——微软创建 NT 和未来的夺命狂奔》
http://book.douban.com/subject/3699395/
[7] 范德成（2011/08）：John Robbins 上课时讲的趣事
http://blog.csdn.net/r_mosaic/article/details/6665934
[8] 范德成（2011/05）：afcastro（老化-频率缓存）的设计
http://www.fandecheng.com/personal/interests/programming/afcastro_design.htm
[9] 范德成（2009/11）：Windows NT Disk Cache Overview
http://www.fandecheng.com/personal/interests/ewindows/advanced_windows/winnt_cache.htm
[10] 范德成（2007/06）：Why Windows Cache Can Exceed Its Virtual Size
http://blog.csdn.net/r_mosaic/article/details/1667174
[11] (MSDN Channel 9, 2007/03) Mark Russinovich: Windows Vista kernel improvements
http://channel9.msdn.com/Shows/Going+Deep/Mark-Russinovich-From-Winternals-to-Microsoft-On-Windows-Security-Windows-CoreArch
[12] (MSDN Channel 9, 2007/12) Mark Russinovich: About Server 2008 and MinWin
http://channel9.msdn.com/Shows/Going+Deep/Mark-Russinovich-On-Working-at-Microsoft-Windows-Server-2008-Kernel-MinWin-vs-ServerCore-HyperV
[13] (MSDN Channel 9, 2009/08) Arun Kishan: Farewell to the Windows Kernel Dispatcher Lock
http://channel9.msdn.com/shows/Going+Deep/Arun-Kishan-Farewell-to-the-Windows-Kernel-Dispatcher-Lock/
[14] (MSDN Channel 9, 2010/09) Landy Wang: Windows Memory Manager
http://channel9.msdn.com/shows/Going+Deep/Landy-Wang-Windows-Memory-Manager
[15] (MSDN Channel 9, 2005/04) Molly Brown: Windows Cache Manager
http://channel9.msdn.com/shows/Going+Deep/Windows-NT-Cache-Manager-Molly-Brown/
[16] Paul Murphy (2004/03): What Differentiates Linux from Windows?
http://www.linuxinsider.com/story/33089.html
[17] ZERO（2005/12）：HeapAlloc 内部算法 http://multi-crash.com/?p=92 （或
http://blog.csdn.net/ZERO2046/article/details/559742）
[18] Android 安全性概述 https://source.android.com/devices/tech/security/
[19] 范德成（2011/10）：分区表和主引导记录
http://www.fandecheng.com/personal/interests/pwindows/msdos_functional/mbr.htm
[20] Michael Fortin (2006/10): Windows Vista Superfetch

http://channel9.msdn.com/shows/Going+Deep/The-Advancement-of-Windows-Michael-Fortin-Windows-Vista-SuperFetch/

[21] (MSDN 2012/01) Building the next generation file system for Windows: ReFS http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx

[22] 范德成（2011/02）：Windows 7 and the 64-bit computing era http://www.fandecheng.com/personal/interests/ewindows/advanced_windows/windows7.htm

[23] NTFS on Wikipedia http://en.wikipedia.org/wiki/NTFS

[24] (Microsoft TechNet 2003/03) How NTFS Works http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx

[25] Computer Forensics (2005/04): NTFS Structure http://www.cse.scu.edu/~tschwarz/coen152_05/PPtPre/NTFSFS.ppt

[26] Linux NTFS project (2004/03): NTFS document http://sourceforge.net/projects/linux-ntfs/files/NTFS%20Documentation/0.5/

[27] Stephen Tweedie (2000/07): EXT3, Journaling Filesystem http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html

[28] B-Tree on Wikipedia http://en.wikipedia.org/wiki/B-tree

[29] 吴峰光等（2007/06）：Linux readahead: less tricks for more https://www.kernel.org/doc/ols/2007/ols2007v2-pages-273-284.pdf

[30] 吴峰光、邹南海（2007/10）：Linux 文件预读

[31] 范德成（2014/12）：NTFS Fact Analysis http://www.fandecheng.com/personal/interests/ewindows/advanced_windows/ntfs_fact_analysis.zip

[32] (MSDN Channel 9, 2011/10) Galen Hunt, Reuben Olinsky, Jon Howell: Drawbridge: An Experimental Library Operating System http://channel9.msdn.com/Shows/Going+Deep/Drawbridge-An-Experimental-Library-Operating-System

[33] (MSDN) Naming Files, Paths, and Namespaces https://msdn.microsoft.com/en-us/library/aa365247(VS.85).aspx#maxpath

全文完