我们能使操作系统安全可靠吗?

安德鲁·塔南鲍姆 卓立特·荷尔德 荷尔伯特·博斯

长期以来微内核体系结构由于其性能较单核结构逊色而被人忽视。如今微核结构有可能由于其潜在的高可靠性而重新风光，因为现在许多研究人员把可靠性看得比性能更为重要。

您什么时候碰到过一台电视机出现系统崩溃或者需要您去网络上下载什么紧急的软件升级补丁？除非那是一台极为老久的机器（即使如此也不会麻烦您去软件升级，只不过需要更换零部件罢了-译者注）。而一台新的电视机只不过是一个由一颗处理器、一个大屏幕监视器、一些解调无线电信号的模拟电子部件、一些特殊的（相对于一般计算机而言）读写（I/O）设备：遥控器、内置卡式录像带机或DVD驱动器以及ROM里的一些软件组成的计算机而已。

这一问，触及到计算机界人士讳莫如深的一个难堪的“秘密”，为什么电视、DVD录像机、MP3播放器、移动电话和其他软件驱动的电子设备都十分可靠、安全，而计算机却做不到呢？当然有许多“解释”，比如计算机是可变的、计算机可以允许用户改变软件、信息技术产业还不成熟等等。但是，当我们步入一个绝大部分的计算机用户是非专业技术人员的时代时，上述理由对于他们来说，就看起来越来越像是毫无说服力的借口。

消费者对于计算机的期待和对电视机一样：买了以后，插上电源，就能完美地工作10年。作为信息技术的专业人员，我们必须面对这个挑战，并且尽量使计算机像电视机一样可靠安全。

要做到可靠和安全，最大的障碍就是操作系统。虽然应用程序也存在着许多缺陷，但是如果操作系统没有毛病，应用程序中的缺陷只能产生有限的破坏。因此我们将集中讨论操作系统。

然而，在进入细节之前，有必要先说明一下可靠与安全之间的关系。这两个方面的问题一般都是由相同的原因造成的，即软件中的缺陷。一个缓冲区溢出错误会导致系统崩溃（可靠性问题），也会让精心编写的病毒或蠕虫接管计算机（安全性问题）。尽管我们主要关注可靠性，但在改善可靠性的同时也会提高安全性。

系统为什么不可靠?

当前的操作系统有两个特征使得它不可靠、不安全：一是规模巨大，二是故障隔离的性能很差。Linux内核拥有超过250万行的代码。Windows XP的内核则是前者的2倍还多。

一项关于软件可靠性的研究表明：每1000行可执行代码中会包含6～16个缺陷[1]，而另一项研究则估计可执行代码的缺陷密度（fault density）为每千行2～75个[2]。按照每1000行代码中有6个缺陷的保守估计，Linux内核大约存在15000个缺陷，而Windows XP中的缺陷至少是它的2倍。

更糟糕的是，通常大约70%的操作系统带有设备驱动程序，而这些驱动程序的失效密度是普通代码的3～7倍，所以上面提到的缺陷数可能在总体上被低估了[3]。显然，发现并修正所有的缺陷是根本不可行的，更何况缺陷的修补经常会引入新的缺陷。

当前操作系统的巨大规模意味着没有人能够理解其全部情况。显然，在没有人真正很好理解一个系统的情况下要将其工程化是极为困难的。

这给我们提出了第2个议题：故障隔离。没有哪个人能弄懂航空母舰的全部零件是如何工作的，但其子系统都实现了彼此很好地隔离，这样就不至于产生厕所阻塞影响到导弹发射子系统的问题。

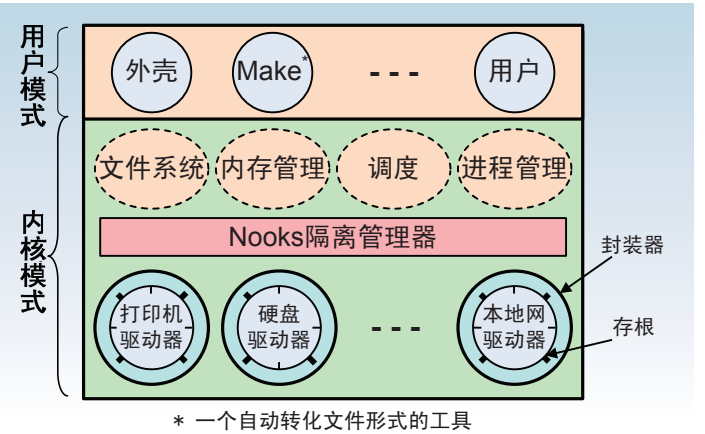
操作系统的各个部件没有这种隔离。现代操作系统包含成百上千个链接在一起的过程，作为一个独立的二进制程序以内核的形式运行。在这几百万行内核代码中，每一行都能够重写与之无关的组件使用的关键数据结构，从而以很难检测到的方式造成系统崩溃。此外，如果病毒或者蠕虫感染了一个内核过程，我们也无法阻止它快速蔓延至其他子程序并控制整台机器。

回到舰船的例子。现代舰船的船壳内分成许多隔间。如果某一个隔间出现了裂缝，只有这个隔间漏水，不会影响整个船体。当前的操作系统就像隔间发明之前的船只，每一个漏缝都能使整只船沉没。

所幸研究人员正在研究构建更可靠的操作系统。下面我们根据方法的“彻底性”按照从最保守到最激进的次序介绍四种不同途径。

加装“装甲”的操作系统

最保守的方法是设计用于改善Windows和Linux等现有操作系统可靠性的Nooks技术[4]。Nooks 维持了在内核模式将成百上千个过程链接在同一地址空间的单内核结构，而把注意力集中于减少设备驱动程序带来的危害。前面已经提到，驱动程序是操作系统问题的核心。



* 一个自动转化文件形式的工具

图1 Nooks模型（每个驱动程序封装在保护软件层内。这个保护软件层可以监视驱动程序和内核之间的所有交互行为）

具体地说，如图1所示，Nooks技术通过将每个驱动程序封装在一个保护软件层内以形成一个轻量级保护域，来保护内核免受设备驱动程序漏洞的危害。这一技术有时称为沙箱。每个驱动程序的外壳都仔细地监视驱动器与内核之间的所有交互行为。这项技术也可以用于内核的其他扩展，比如可装载文件系统。但是为了简明起见，我们只讨论驱动程序。

Nooks技术的目标是：

- 当驱动程序发生故障时保护内核；
- 当驱动程序失效时自动恢复；
- 在执行上述任务的过程中，尽可能少地改变现有驱动程序和内核。

Nooks技术的目标并没有包括保护内核免受恶意驱动程序的破坏。这种技术最初是在Linux上实现的，但这种方法在其他现有内核中同样可得到很好的应用。

隔离

虚拟内存页表是用于防止驱动程序出错破坏内核数据结构的主要工具。当一个驱动程序运行时，除它之外的所有内存页就都变成只读的，这样就为每个驱动程序实现了一个单独的轻量级保护域。按这种方式，驱动程序能读取它所需要的内核数据结构，但是任何尝试直接改变内核数据结构的行为都将导致CPU异常而被Nooks隔离管理器捕获。驱动程序对其私有存储空间（用于存储堆栈、

堆、私有数据结构和内核对象拷贝）的访问则是可读写的。

插入

每类驱动程序都可以导出一组能够为内核调用的函数。例如，声音驱动程序可以提供一个调用函数来将一段音频样本写到声卡上，而另一个函数用于调整音量等。当加载驱动程序时，包含指向驱动程序函数的指针数组被赋值，这样内核就能据此找到每个函数。另外，驱动程序还会导入一组由内核提供的函数，例如，用于分配数据缓冲区的函数。

Nooks技术为导入和导出函数都提供了封装器（Wrapper）。当内核调用驱动程序函数或者驱动程序调用内核函数时，调用指令进入封装器。封装器将检查参数的正确性并管理调用。尽管封装器的存根（Stub）是从它们的函数原型里自动产生出来的（如图1中伸出和插入驱动程序的线段所示），开发者还是必须手工编写封装器本体。Nooks技术小组编写了455个封装器，其中329个用于内核函数的导出，126个用于设备驱动器函数的导出。

当一个驱动程序试图修改一个内核对象时，它的封装器会把这个对象拷贝到驱动程序的保护域，也就是其私有读写页面中。驱动程序然后修改这个拷贝。在成功完成这个请求之后，隔离管理器再将修改过的内核对象复制回内核。通过这种方式，即使在调用驱动程序时出现崩溃或者故障，也能将内核对象保持在一个有效的状态下。跟踪导入对象的方法是针对对象特定的，因此Nooks技术小组不得不手写代码来跟踪Linux驱动程序使用的43类对象。

恢复

当出现故障后，用户模式恢复代理开始运行并查询配置数据库，以便确定接下来要做什么。在许多情况下，释放占有的资源并重新启动驱动程序就足够了，因为除了最耗时的和不常见的错误，大多数普通的算法错误都能在测试过程中发现。

虽然这项技术能够恢复系统，但可能无法继续运行之前的应用程序。因此，在后来的工作中[5]Nooks技术小组提出了影子驱动程序这个概念，以在出现驱动程序失败之后能继续运行之前的应用程序。

简而言之，在正常操作中，影子驱动程序会记录在恢复应用程序运行时可能会需要的每个驱动程序与内核之间的通信。在驱动程序重启之后，影子驱动程序会将日志中的信息输入到重启的驱动程序。例如，通过重复读写控制（I/O control，IOCTL）系统调用来设置参数（例如音量）到出错前状态，内核不会察觉这一过程。一旦该过程完成，驱动程序就开始处理新的请求。

局限性

实验表明，尽管Nooks技术可以截获99%的驱动程序致命错误和55%的非致命错误，但这并不理想。例如，驱动程序仍可能会执行它们本不应执行的特权指令，写到错误的I/O端口，进入无限循环。此外，Nooks技术小组不得不手写大量的封装代码。这些代码有可能包含许多错误。Nooks没有对驱动程序重新获得对所有内存的写权限采取防范措施。尽管如此，Nooks技术仍是一种潜在的提高传统操作系统内核可靠性的有效途径。

半虚拟机（Paravirtual Machines）[1]

第二种方法建立在虚拟机概念之上。虚拟机概念的出现可以追溯到20世纪60年代[6]。简言之，这种思想是在硬件上而不是在操作系统上运行一个叫做虚拟机监视器的特别控制程序。虚拟机可创建真实机器的多个实例。每个实例可运行能在裸机上运行的任何软件。

[1] 虚拟化的一种方式，基于host/guest结构，在这种模式下，虚拟机监视器实际上对子操作系统（guest OS，亦有译作客户操作系统）的代码进行了修改。

这种技术允许两个或更多的操作系统，比如Linux和Windows系统，同时运行在同一个硬件上，而且每个操作系统都认为自己独占整个机器。虚拟机素以有较好的故障隔离而著称。毕竟，如果任何虚拟机都不知道其他虚拟机的存在，那么其中一台机器上的问题就不可能蔓延到其它机器上去。

这里所做的研究是将这种思想应用到单个操作系统的保护中去，而不是在不同操作系统之间[7]。此外，因为奔腾芯片不能完全虚拟化，在虚拟机上运行一个无修改的操作系统这一想法也不得不适当修正，即允许修改操作系统，以确保其不去做任何无法被虚拟化的事情。为了区别这种技术和真正的虚拟化，我们称其为半虚拟化。

在20世纪90年代，卡尔斯鲁厄（Karlsruhe）大学的一个研究小组编写了L4微内核[8]。他们能够在L4微内核之上以可以看成是虚拟机的方式运行一种稍加修改的Linux版本（L4 Linux）[9]。研究人员后来认识到在L4上不仅可以运行一个Linux拷贝，还可以同时运行多个拷贝。如图2所示，这一认识产生了一个想法：在一个Linux虚拟机上运行应用程序，而在一个或多个其他Linux虚拟机上运行设备驱动程序。

通过让设备驱动程序运行在与主虚拟机互相分离的一个或多个虚拟机上，主虚拟机运行剩下的操作系统程序和应用程序，就能做到如果一个设备驱动程序崩溃，只有和它相关的虚拟机会受影响，而主虚拟机仍然继续运行。这种方法的另一个优点是不必修改设备驱动程序，因为它们运行在一个标准的Linux内核环境中。当然，为了实现半虚拟化，必须修改Linux内核，但这是一次性的修改，无需为每个设备驱动程序重复工作。

因为设备驱动程序运行在硬件的用户模式中，一个重要问题是它们如何进行读写（I/O）操作和中断处理。物理层上的读写是通过在Linux内核中增加约3000行代码，使运行在Linux内核上的驱动程序被允许通过利用L4服务进行读写，而不是自行操作。另外还有约5000行代码用于处理磁盘、网卡和PCI[2]总线这三个相互隔离的驱动程序与运行应用程序的虚拟机之间的通信。

理论上，这种方法会提供比单个操作系统更高的可靠性，因为当包含一个或多个驱动程序的虚拟机发生崩溃时，虚拟机可以重新启动，驱动程序也会返回到它们的初始状态，而不是像Nooks技术那样试图让驱动程序回到崩溃前的状态。例如，如果一个音频驱动程序崩溃了，它会将音量恢复为默认值，而不是崩溃前的值。

性能测试表明，采用这种方式的半虚拟机系统的实际开销所占比例大约为3%～8%。

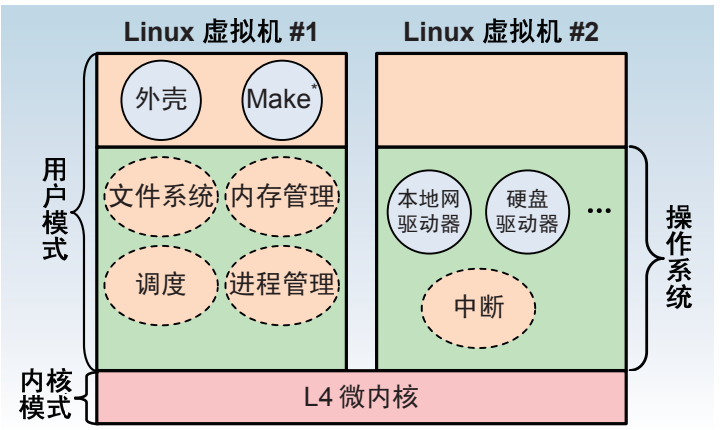


图2 虚拟机（其中一个Linux虚拟机运行应用程序，其他一台或多台虚拟机运行设备驱动程序）

多服务器操作系统

前两种方法主要关注的是为传统的操作系统打补丁，接下来的两种方式主要是针对未来的操作系统。

其中一种方法直接颠覆了把整个操作系统当作内核模式中的一个单一的庞大的二进制

[2] Peripheral Component Interconnect，外设部件互连标准

程序来运行，这样一个操作系统设计的核心理念。取而代之的是，只有很小的微内核在内核模式中运行，而操作系统的其余部分则作为一组完全隔离的用户模式的服务器和驱动进程来运行。

这种思想已经存在20多年了，但是因为其性能稍低于单内核，所以最初它并没有得到充分研究。在20世纪80年代，性能代表了一切，可靠性和安全性还没有进入人们关注的视野。当然，那时，航空工程师也没有花很多精力去担心每英里耗费的汽油量以及驾驶员座舱门承受武装袭击的能力。现在时代改变了，人们关注的重点也在改变。

多服务器体系结构

通过关注一个现代的实例会有助于更清楚地了解多服务器操作系统的思想。如图3所示，在Minix3中，由微内核处理中断，并且为进程管理提供基本机制，实现进程间的通信以及进程调度。微内核也提供一小部分内核调用给授权的驱动程序和服务器，比如读取指定用户地址空间的特定部分，或者向经授权的I/O端口写入。时钟驱动程序共享微内核的地址空间，但作为一个分开的进程进行调度。除此以外，没有其他的驱动程序运行在内核模式中。

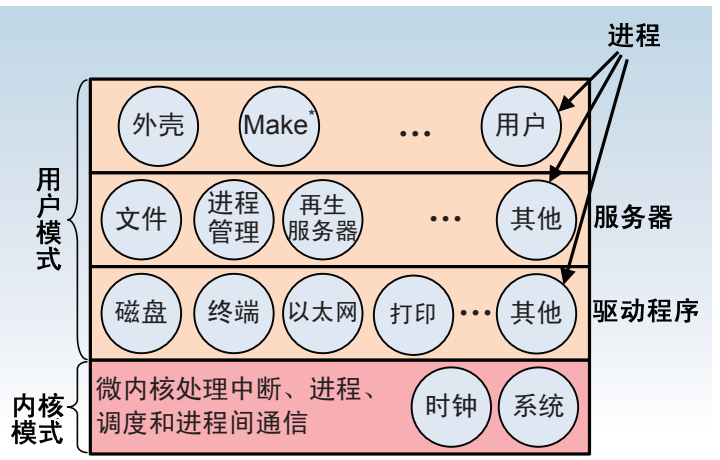


图3 Minix 3的结构（微内核处理中断，为进程管理提供了基本机制，实现进程间的通信以及进行进程调度）

在微内核之上是设备驱动层[10]，其中每一个I/O设备都有自己的驱动程序。这些程序作为彼此分立的进程在私有地址空间中运行，并受内存管理单元（Memory Management Unit，MMU）硬件的保护。设备驱动层包含了磁盘、终端设备（键盘和显示器）、以太网、打印机和音频等设备的驱动程序进程。这些驱动程序运行在用户模式中，不能执行特权指令或者读写电脑的I/O端口。它们必须通过内核调用来获得这些服务。虽然带来了一些开销，但这种设计提高了可靠性。

在设备驱动层的上面是服务层。其中的文件服务器是一个小程序（只有4500行的可执行代码），接受来自用户进程的请求处理针对与文件相关的POSIX[3]系统调用，例如read（读）、write（写）、lseek（随机访问文档）和stat（获取文件属性）等指令，并执行这些系统调用。同在这一层的还有进程管理器，负责处理进程和内存的管理，并执行POSIX和其他的系统调用，例如fork（生成子进程）、exec（装入和运行程序）和brk（改变进程数据段）等。

一个有些与众不同的方面是再生服务器（reincarnation server）。它是所有其他服务器和所有驱动程序的父进程。如果一个驱动程序或服务器崩溃、结束或未能响应周期探测请求，再生服务器会在必要时将其杀死，然后根据磁盘或者随机存储器上的拷贝进行重新启动。虽然驱动程序可以按这个方式重新启动，但是目前只有那些不维护太多内部状态的服务器能做到这点。

其他的服务器包括网络服务器（包含全部的TCP/IP[4]堆栈）、数据存储（供其他服务器使用的简单命名服务器）以及用来辅助调试的消息服务器。

最后，位于服务层之上的是用户进程。它和其他Unix系统的

[3] Portable Operating System Interface of Unix，可移植操作系统接口

[4] TCP（Transmission Control Protocol，传输控制协议），IP（Internet Protocol，网际协议）

唯一区别是，那些用于读、写以及其他系统调用的库函数是通过向服务器发消息来完成工作的。除了这个隐藏在系统库中的区别以外，它们还是能使用POSIX的应用程序接口（API[5]）的普通的用户进程。

进程间通信

因为允许所有进程相互协作，所以进程通信（IPC[6]）在多服务器操作系统中至关重要。然而，由于Minix3中的服务器和驱动程序都作为物理隔离的进程运行，因而它们不能直接调用彼此的函数或者共享数据结构。因此Minix3利用rendezvous原则（直接地址、固定长度、不经过缓存、没有附加拷贝）传递固定长度的消息来实现进程间通信。当发送方和接收方都准备就绪时，系统直接将这个消息从发送方拷贝给接收方。此外，还存在一种异步事件通知机制。那些不能发送的事件在进程表中会被标记为一个挂起的位图。

Minix3巧妙地将中断与消息传输系统结合在一起。中断处理器使用通知机制来发出读写完成的信号。这个机制允许中断处理器在驱动程序的“挂起中断”位图中设置一个比特位，然后无阻碍地继续执行。当驱动程序准备好接收中断时，内核将把挂起的内容转变成正常消息。

可靠性特征

Minix3的可靠性来自于多个方面。首先，由于只有大约4000行代码运行在内核中，因此按照每1000行代码有6个缺陷的保守估计，Minix3的内核中缺陷总数估计大约仅有24个。相比之下，Linux有15000个，Windows则更多。由于除时钟之外的设备驱动程序都是用户进程，因此外来代码不会以内核模式运行。内核规模小也会使采用手工方式或形式化方式验证代码成为可能。

Minix 3的进程间通信设计不要求消息进行排列或缓冲，因而消除了对内核缓冲区管理的需求。此外，由于进程间通信是功能强大的结构，因此每个服务器和驱动程序的进程间通信的能力都是严格限定的。对于每个进程，可用的进程间通信原语、允许的目标以及用户事件通知都是有限制的。例如，用户进程只能依据rendezvous原则并且只能发送给POSIX服务器。

另外，所有的内核数据结构都是静态的。所有的这些特性都极大地简化了代码，也消除了缓冲区溢出、内存泄漏、异常中断、不可信内核代码以及其他的原因带来的内核缺陷。当然，将操作系统的大部分功能转变为用户模式并不能消除驱动程序和服务器中不可避免的缺陷，但能大大降低这些缺陷的危害。一个内核缺陷有可能破坏重要的数据结构，也有可能在磁盘中写入垃圾数据，等等；而大多数驱动程序和服务器中的缺陷不会再造成这么大的破坏，因为这些进程被严格地分隔开，且活动范围十分受限。

用户模式下的驱动程序和服务器并不作为超级用户运行。它们不能访问自身地址空间以外的内存，除非是通过内核调用（此时内核检验其合法性）。更严格的是，内核进程表中的位图和范围以按各个进程具体规定的方式控制着得到认可的内核调用集合、进程间通信的能力以及可用I/O端口。例如，内核可以阻止打印机向用户地址空间写入、访问磁盘I/O端口或者给音频驱动程序发送消息。在传统的单内核系统中，随便一个驱动程序能够做任何事情。

另一个与可靠性有关的特性是将指令空间和数据空间分开。假如程序错误或者病毒让驱动程序或者服务器的缓冲区溢出，并在数据空间中写入外来代码，这些注入的代码无法通过跳转或者过程返回到其上而执行，因为内核不会运行在对应进程的（只读）指令空间以外的代码。

在其他为提高可靠性而设计的专门特性中，最重要的是自治愈功能。如果驱动程序通过一个非法的指针进行存储，陷入无限循

[5] Application Programming Interface

[6] interprocess communication

环或者出现其他的异常行为，那么再生服务器会在不影响到正在运行着的进程情况下，自动替换掉这个驱动程序。

重启一个出现逻辑错误的驱动程序并不能消除错误，然而在实际情形中，微小的定时错误或其它类似错误会导致许多问题，而通常重启驱动程序能修复系统。此外，这个机制能够修复由攻击引起的，诸如“死亡ping（ping of death）”通过发送错误格式的IP包导致计算机崩溃这样的程序错误。

性能考虑

几十年来，研究者们因性能问题而对建立在微内核基础上的多服务器体系结构持批评态度。然而，许多项目证明模块设计实际上可以提供具有竞争力的性能。虽然Minix3没有经过性能优化，但是它运行得相当快。与内核模式驱动程序相比，用户模式驱动程序造成的性能损失不超过10%。在2.2GHz的超威速龙（Athlon）处理器上，系统能够在不到6秒的时间内完成自我创建，包括内核、通用驱动程序以及全部服务器（112个编译器和11个链接器）等内容。

多服务器的体系结构使得以很小的性能开销为代价，构建高可靠性的类Unix环境成为可能，这使其成为实际可行的技术。针对奔腾（Pentium）处理器设计的Minix3可以在遵守伯克利许可证（Berkeley license）的前提下从www.minix3.com免费下载。面向其他硬件结构和嵌入式系统的端口也正在开发中。

基于语言的保护

这种最彻底，最激进的方法出人意料地来自微软研究院。事实上，微软的这种方法摒弃了原有的概念，即以内核模式运行的单一程序加上一些以用户模式运行的用户进程集合，取而代之的是由新的类型安全（type-safe）语言写成的系统。在这种类型安全语言中，不存在指针或其它与C或C++语言有关的问题。和前面的两种技术一样，这种技术也已存在了几十年。

伯勒斯（Burroughs）B5000计算机使用过这种技术。那时唯一可用的语言是Algol[7]。它的保护不是通过内存管理单元（那时还没有这种单元），而是通过Algol语言的编译器拒绝产生“危险”代码来实现的。面向21世纪，微软研究院改进了这一思想。

概况

这一名为Singularity的系统几乎全部是用Sing#语言（一种新的类型安全语言）写成的。这种语言在C#语言基础上，增加了消息传递原语，其语义由正式的书面协议定义。因为语言安全性严格限制了系统和用户的进程，所以所有的进程可一起运行在一个单一的虚拟地址空间上。这种设计是安全的，因为编译器不允许一个进程访问另一进程的数据，而且也是高效的，因为它取消了内核陷阱和上下文转换。

此外，Singularity的设计也是很灵活的，因为每个进程都是闭合的实体，因而可以有自己的代码、数据结构、内存布局、运行时系统、库和内存回收器。系统使用了内存管理单元，但只用于映射页面而不是为每个进程建立单独的保护域。

Singularity设计的一个关键原则是禁止动态的进程扩展。这样做的结果之一就是不允许可加载模块（比如，设备驱动程序和浏览器插件）的存在，因为它们有可能引入未经验证的外来代码。这些代码有可能破坏其父进程。取而代之的是，这种扩展必须作为独立的进程运行，以便完全隔离并且按照标准的进程间通信机制通信。

微内核

Singularity操作系统由一个微内核进程和一组用户进程组成。它们通常都运行在同一个虚拟地址空间中。微内核可以控制对硬件

[7] 也被称为国际代数语言，是计算机发展史上首批产生的高级语言，当时还是晶体管计算机流行的时代，由于ALGOL语句和普通语言表达式接近，更适于数值计算，所以ALGOL多用于科学计算机。

的访问、分配和清空内存，创建、销毁和调度线程，利用互斥机制处理线程同步，利用通道处理进程间的同步以及监视读写操作。每一个设备驱动程序都作为一个单独进程来运行。

虽然大部分微内核代码是用Sing#编写的，但是还有一小部分是用C#、C++或者汇编语言编写的，而且这部分必须是可信的，原因在于它不能被验证。这些可信代码包含了硬件抽象层和内存回收器。硬件抽象层通过隐藏一些概念（如，I/O端口、中断请求行、直接内存访问通道和定时器）向系统屏蔽低层硬件，以便向操作系统其他部分提供与机器无关的抽象。

进程间通信

用户进程利用点到点的双向通道，通过发送强类型消息给微内核来获得系统服务。事实上，所有进程到进程的通信都利用这些通道。与其他的消息传递系统在某个函数库中拥有“SEND（发送）”和“RECEIVE（接收）”的功能不同，Sing#完全通过语言自身来支持这个通道，包括形式化的类型系统和协议规范。

为了弄清楚这一点，可参看如下这个通道规范：

```
contract C1 {
ain message Request(int x) requires x > 0;
aout message Reply(int y);
aout message Error();

state Start:
aRequest? -> Pending;
state Pending: one {
aReply! -> Start;
aError! -> Stopped;
}
state Stopped: ;
```

上述协议表明，通道允许通过3条消息：请求、应答和报错：第1条消息以正整数作为参数；第2条消息以任意整数作为参数；第3条消息没有参数。当需要使用该通道向服务器发消息时，请求消息便从客户端发送到服务器，其余两条消息则反向发送。用一个状态机指明有关通道的协议。

在起始状态，客户端发送请求消息，通道进入挂起状态。这时，服务器可以要么发送答复消息，要么发送出错消息来响应请求。答复消息让通道重新恢复到起始状态，继续通信。出错消息则让通道进入停止状态，结束通道上的通信。

堆

如果所有的数据，如从磁盘读入的文件块，都必须经过通道，那么系统就会很慢。因此，在基本规则中加入“异常”。这个规则规定每个进程的数据都是完全私有和内部的。虽然Singularity支持公共的对象堆，但堆中的每个对象在每一时刻都属于一个单个进程。不过，堆对象的所有权可以在通道上传递。

以I/O设备为例，看看堆是如何运作的。当磁盘驱动程序在一个数据块中进行读操作时，程序会把这个块放到堆上，然后系统将这个块的处理权传递给提出数据请求的用户，继续维护单个所有者的原则，但允许数据无需拷贝就可从磁盘直接传到用户。

文件系统

Singularity操作系统为每个服务器维护一个单一的分级命名空间。由一个根命名服务器处理树顶端的事务，而其他命名服务器可挂载在它的结点上。文件系统作为一个进程挂载在/fs上，所以类似于/fs/users/Linda/foo的名字可能是一个用户文件。文件以B-树[8]的形式实现，以数据块号为关键字。当用户进程请求一个文件时，文件系统会要求磁盘驱动程序将被请求的数据块放到堆上。然后就如前面描述的一样转移所有权。

[8] 一种多路平衡查找树，适合在磁盘等直接存取设备上组织动态的查找表。

验证

每个系统组件都有元数据，用以描述它的附属结构、出口、资源和行为。这些元数据是用来验证的。系统映像由微内核、驱动程序、运行系统所需要的应用程序以及它们的元数据组成。外部验证器可以在系统执行映像之前对其执行多项检查，例如，确保驱动程序没有发生资源冲突。

验证过程分为3个步骤：

■ 编译器检查类型安全性、对象所有权和通道协议等。

■ 编译器生成微软中间语言（Microsoft Intermediate Language，MSIL），一种可移植的类似Java虚拟机（JVM[9]）的字节码，以供验证器检查。

■ 微软中间语言经后端编译器编译成x86代码，可以在这种代码中插入运行时检查（尽管现有编译器不进行这项检查）。

这些冗余验证的目的是在验证器中捕获错误。

上面介绍的4种方式都是把注意力集中在通过防止存在错误的设备驱动程序引起系统崩溃，试图以此来提高操作系统的可靠性。

在Nooks技术中，每个驱动程序都是手工单个封装在软件外壳之中，以便于仔细监控它与操作系统其他部分的交互，但所有驱动程序都运行在内核中。半虚拟机方法不仅在这一点上有了进一步的扩展，而且还将驱动程序转移到一个或多个虚拟机上。这些虚拟机与主虚拟机隔离，进一步削弱了驱动程序的权限。Nooks技术和超虚拟机方法都着眼于提高现有（或传统）操作系统的可靠性。

与此相对，另外2个方法是用更可靠和更安全的操作系统代替现有的操作系统。多服务器操作系统让驱动程序和操作系统组件运行在隔离的用户进程中，同时允许它们之间使用微内核的进程间通信机制来通信。最后，Singularity操作系统——最彻底的方法，通过类型安全语言、单一地址空间以及形式化协议来仔细限定每个模块的功能。

上述4个项目中的3项研究成果（基于L4的超虚拟机、Minix3和Singularity操作系统）使用的都是微内核。目前还无法预料哪种技术（如果有的话）会在未来发展中得到广泛应用。不管怎样，一个有趣的问题值得我们注意：长期以来由于性能不如单内核而为人们所摒弃的微内核技术可能因为它们潜在的高可靠性而卷土重来，现在许多人认为可靠性比性能更重要。微内核技术的轮回已经启动了。

编译自美国电气与电子工程师学会计算机学会《IEEE Computer》2006年第5期“Can We Make Operating Systems keliable and Secure”
作者：Andrew S. Tanenbaum、Jorrit N. Herder、Herbert Bos
译者：贾春福、钟安鸣、刘昕海

贾春福
博士，南开大学教授，主要研究方向为信息安全与可信计算，恶意软件的发现与防治技术。

钟安鸣
博士，中国民用航空大学教师，研究方向为信息安全技术。

刘昕海
南开大学信息技术科学学院硕士研究生，研究方向为信息安全技术。

参考文献

[1] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Comm.ACM, Jan. 1984, pp. 42-52

[2] T.J. Ostrand and E.J. Weyuker, The Distribution of Faults in a Large Industrial Software System, Proc. Int'l Symp. Software Testing and Analysis, ACM Press, 2002, pp. 55-64

[9] Java Virtual Machine，Java虚拟机

[3] A. Chou et al., "An Empirical Study of Operating System Errors," Proc. 18th ACM Symp. Operating System Principles, ACM Press, 2001, pp. 73-88
[4] M. Swift, B. Bershad, and H. Levy, "Improving the Reliability of Commodity Operating Systems," ACM Trans. Computer Systems, vol. 23, 2005, pp. 77-110
[5] M. Swift et al., "Recovering Device Drivers," Proc. 6th Symp. Operating System Design and Implementation, ACM Press, 2003, pp. 1-16
[6] R.P. Goldberg, "Architecture of Virtual Machines," Proc. Workshop Virtual Computer Systems, ACM Press, 1973, pp. 74-112
[7] J. LeVasseur et al., "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," Proc. 6th Symp. Operating System Design and Implementation, 2004, pp. 17-30
[8] J. Liedtke, "On Microkernel Construction," Proc. 15th ACM Symp. Operating System Principles, ACM Press, 1995, pp. 237-250
[9] H. Hartig et al., "The Performance of Microkernel-Based Systems," Proc. 16th ACM Symp. Operating System Principles, ACM Press, 1997, pp. 66-77
[10] J.N. Herder et al., "Modular System Programming in MINIX 3," Usenix; www.usenix.org/publications/login/2006-04/openpdfs/herder.pdf