可扩展操作系统设计方法*)

A Design Methodology for Extensible Operating Systems

施笑安　周兴社　林　奕　王继晖

(西北工业大学计算机科学与工程系　西安710072)

Abstract Operating systems research has traditionally consisted of adding functions to the operating system or inventing and study new implementations. Regardless of the research goal, the single constant has been that the size and complexity of operating system increase over time. As a result, operating systems become the most single complex piece of software in a computer system. Today's operating system research is aimed at finding new ways to construct the operating system in order to increase its flexibility, allowing it to adapt to change in designing such extensible systems and the array of choices facing the operating system designer. We discuss such question for describing extensible operating systems and then advance four key design point for extensible operating systems, we analyze and argue each point that relate to different extensible technology. Provide scientific methodology for Operating systems research or development.

Keywords Operating system, Extensible, Kernel, Flexibility

1. 研究背景

大约每隔十年,操作系统就会经历一段重新构造,如50年代的批处理系统,60年代的分时操作系统,70年代的可移植操作系统(如 UNIX)。它最初是一个小而简单的系统,随着时间的推移,它已经成为一个大而复杂的系统。为了改进系统结构,80年代出现了微内核结构,微内核技术具有许多优点,如:使操作系统的结构更加模块化;系统可靠性和容错性好;系统具良好的可扩展性、可裁剪性[4]。

但是人们在研究和使用微核心操作系统的过程中,发现微内核操作系统普遍存在一个严重问题:即性能不好、效率不高。例如,Mach3.0的性能不如4.3BSD UNIX 这种传统单一大核心结构操作系统[2]。这一点已经引起了普遍关注。为更好地提高操作系统的性能和柔性,导致90年后期突出为可扩展操作系统研究。可扩展操作系统由一套接口和实现组成,可以根据应用的需要而灵活改变,它缓解了核心提供的服务不能满足应用需求的矛盾。

本文在对可扩展操作系统深入研究的基础上,提出了可扩展操作系统的四个关键设计点,并深入分析和论证了每个设计点中各种具体的可扩展技术。

2. 可扩展操作系统设计目标

(1)提高操作系统的性能

提高性能的目的是使系统能更好地满足应用需求,并提供更多的功能。传统操作系统有很多问题影响了系统的性能。例如:在传统操作系统中,应用将数据从网络转移到本地的磁盘上,是通过从网络拷贝数据到应用再返回操作系统的文件系统中,这在用户/内核边界导致大量不合理数据移动。解决该问题的方法通常是允许应用下载一个内核的扩展,它产生一个网络到磁盘的直接通道,而不需要再对数据进行额外的拷贝,提高系统的性能。此外,上下文切换次数也会造成操作系统性能下降,可以采用将代码移入内核减少上下文切换次数来提高系统的性能[4]。

(2)提高操作系统的柔性

提高柔性是为使操作系统适应应用需求的不可预测性或特殊目的的功能[3]。一类应用如数据库管理系统、多媒体系统、实时应用系统,常规的系统不能对它们提供优质的服务[5]。在大多数情况下,这些应用不得不采用较难的办法以避免使用操作系统提供的服务,或使用一个已经对特定应用进行了优化的特殊用途的系统,如实时操作系统。另外,当一类新应用出现时,毫无疑问,它需要操作系统提供新的功能支持,但当操作系统不断加入新的功能,必然使系统太大而难以维护,且可靠性、适应性更差。

为了支持可扩展性,一个可扩展操作系统要提供一套允许应用根据特定的要求裁剪系统行为的机制和接口。例如,一个可扩展操作系统的文件组件必须允许应用定制高速缓存置换算法。在进行一些特定应用处理(如数据压缩或搜索)时,请求处理在唯一的一个文件数据块上时,一个磁盘的驱动器能够被一个智能的磁盘驱动器所代替[9]。

3. 传统操作系统结构

一个传统的操作系统主要由内核、库和服务器构成。内核是执行在特权模式下的一段代码,在硬件的支持下可直接访问物理资源。应用运行在用户级,操作系统给应用提供服务。接口为应用提供操作系统的服务。应用程序与系统的接口(API)定义易于使用的系统高级抽象。如 POSIX 是一个通用 API,用来开发标准的 UNIX 类接口。内核对用户级程序提供系统编程接口(SPI),通过系统调用进行访问。有时,内核也需要使用一个服务调用进入应用/服务器。SPI 应该为有效、可配置和安全的 API 提供必要的支持。

*)基金项目:本课题得到国防预研基金项目(98J15.15.HK0321)和西北工业大学博士创新基金资助。施笑安　讲师,博士生,主要研究方向为网络与分布式软件、嵌入式系统、自适应计算。周兴社　博士导师,教授,主要研究方向为网络与分布式计算。林　奕　博士生,研究方向为网络与分布式计算。王继晖　博士生,研究方向为网络与分布式计算。

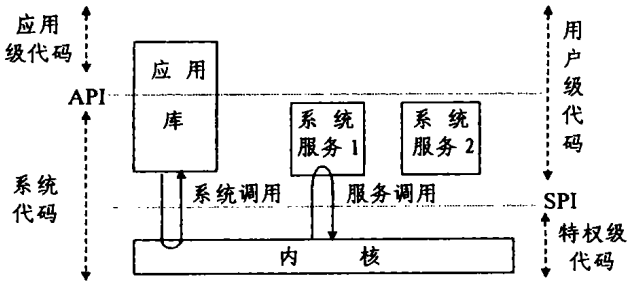

图1 传统操作系统的结构

如何对传统的操作系统进行改造和设计，提高其性能和柔性，面临以下的主要问题：(1)怎样扩展并如何有效地使用扩展?(2)怎样保证扩展的安全?(3)扩展系统是公平的吗?(4)对扩展如何编程?

通过对当前的可扩展操作系统进行的分析，找出了以下关键设计技术，解决可扩展系统的性能和柔性。

4. 可扩展技术

可扩展操作系统设计技术由四个关键设计点构成，包括“扩展位置”、“程序生成”、“保护”和“扩展粒度”。其中“扩展位置”和“生成程序”，是衡量系统性能和柔性的尺度；“保护”解决系统可能由扩展引发的冲突，是系统正确扩展的基础；“扩展粒度”是系统可扩展的单位。表1总结了以上讨论的设计点。

表1 可扩展操作系统设计方法分类

设计点	属性
扩展位置	内核
	库
	服务器
程序生成	选择
	替换
	推理
保护	硬件
	软件
	解释
扩展粒度	粗粒度
	细粒度
	受限的细粒度

4.1 扩展位置

(1)内核 将专用的模块作为内核扩展驻留在操作系统的地址空间，虽然可确保按需要安全地分配物理资源，但会使系统丧失柔性[6]。此外它也增加了执行在特权模式里的代码，这会增加因程序错误而引起灾难性风险。然而，将代码包含进内核具有一个性能优势：在内核里它响应系统调用比它在服务器中快，因为服务内核事件比服务器调用更在系统的内部。

亟待解决的问题是，我们不仅将代码载入内核，还要提供安全性。如OSF/1系统利用超级用户动态地将可执行代码或一个文件系统载入内核，实现新的系统调用。但其不能约束下载的代码以保证内核的安全[12]。

对于经常执行的系统调用，Synthesis系统通过内核动态生成内核代码，实现快速特定的执行。它采用了代码评估，存储和内过程调用等优化技术。例如，当一个文件以读方式打开，内核代码产生指向该文件特定读数据的文件描述符。这种方法虽然没有改变系统功能，但加速了现存的系统调用——应用不需要通过生成的代码来控制系统，超越系统以执行它期望的策略。SPIN和Exokernel系统也都是运用将代码载入内核来实现扩展的[1]。

(2)系统库 每一任务能与它自己选择的库相连，柔性地设置系统代码到一个库中，提供了最大的可扩展性选择。事实上，每一种类型的应用都有其特定的资源需要，诸如数据库系统和科学计算，能够用适当的策略连接一整体系统服务的方式获益。

许多系统，象OSF/1，允许动态的库连接，它扩充了应用的适应性和移植性。如一个应用在OSF/1编译时能使用动态连接库运行在MACH上。另外，这些库例程的代码段能够在运行应用的时间进行共享，减少了内存的开销。

因为库例程能够替换系统调用和进程间通信(IPC)，所以可以提供大部分高效的选择，库能够让后续的调用使用以前调用的存储信息。

但不幸的是，在资源请求变化或存储一个设备时，库不能为服务器提供足够的能力。

(3)服务器 单个的应用不可能控制服务器里的策略实现，达到用库中代码所能实现的柔性。虽然如此，服务器比内核更较容易改变，例如超级用户可以关闭一个服务器，修改它并重新启动它。一个服务器能仿真现存的API，或者提供一个特定的功能，如一个文件系统。提供不同功能的服务器集合允许多个应用混合或按照它们自己的需要匹配不同策略。

服务器在多个计算应用间控制资源分配十分有用，内核可以信任确定的服务器(通常，有超级用户运行这些服务器)以对应用提供公平的服务，甚至保护应用。在V++系统里，一个服务器可以支持一个被分配给它的应用预定的物理页池；内核在服务器间依次分配资源。

应用通过IPC与服务器进行交互，如Mach系统通过重新指向系统调用支持对UNIX可执行程序的兼容。其它的系统也对应用与服务器间的通信明确地提供了IPC支持。在任何情况下，一个对服务器的调用和它后来的返回会引起两个域(内核与用户域)的交叉并对变量的通行带来了额外的开销。

现在，人们正在研究使系统中的服务由共享库与专用服务器共同提供的方法。其中共享库完成服务中与性能密切相关的部分，而服务器完成应用程序所无法完成的、不要求高性能的工作[6]。这一结构具有显著优点：首先，由于服务是由库提供的，应用程序可以自由使用共享库，应用程序和共享库之间可以结合得较为紧密。在请求服务时直接访问共享库，避免了与远地服务器进行通信的开销。其次，由于共享库可以和特定应用密切相关，这样便可以利用与特定应用有关的知识来优化策略，例如可以高效地实现资源管理。

4.2 程序生成

(1)选择方式 是应用对操作系统提供的选项按自己的需要进行定制。Hydra是一个较早的例子，其内核设计目标是允许应用级软件定义策略机制。因为保护域相交带来很高开销，在每一个策略的决策上用应用级软件进行调用是不切实际的；所以Hydra执行参数化策略以便应用级软件能够选择适当的参数，并将应用定制策略的实施留给内核。这种方法的成功取决于对未来的应用恰当的估计及其策略的适应能力。

(2)替换方式 应用在操作系统内替换一个模块。在许多系统中，应用希望扩展系统要么增加它们自己的专用组件，要么保证适当的专用组件已经预先安装了。增加一个特定组件的典型方法是任务将代码下载到内核中。这种方法是底层的

和明确的，因为应用明白它们所需要的特殊变化，它们宁愿靠提供内核代码来进行系统扩展，而不用一个高级描述方式。在这样的系统里，调整操作系统性能的责任由应用完成。

虽然该方法是低级的，但明确的定制允许按任务的需要精确地进行调整。它们也有一些问题，首先，定制需求要求构造特殊的内核代码，意味着将在任务级需要一个高级的内核专门技术。其次，单个的应用没必要用全系统来执行专门的组件，这主要表现在定制与其它应用的冲突上。因此，替换方式不适用于有许多应用的大系统。第三，因为它需要应用对系统动态的变化进行响应，所以用它支持自适应的操作系统是困难的；最后，明确的定制不能支持不愿和不能对调整操作系统承担责任的应用。

(3)*推理方式*　代替明确地定制的一个可供选择的办法是推理定制。操作系统替换它自己拥有的模块，决策对应用是透明的。操作系统支持推理定制生成和动态地选择适当的特定组件，并且通过通常的系统调用接口自动地使用信息。这样的系统对含糊的应用提供了一些支持；然而用于驱动定制的有限信息意味着失去了最优化的机会。Synthetix 系统的内核就是一个早期的基于推理定制例子。

4.3　保护

保护对操作系统的扩展部分进行控制和限制，保证整个系统安全和可靠。每一项扩展技术都是提供了不同层次的安全，并利用一个不同的信任层次，主要的信任模式有以下三种。

(1)*硬件保护*　为利用硬件提供的保护优势，将扩展放在内核地址空间的外部是最简单的、可能也是确保内核安全的最可靠方法[7]。当内核想进行一个扩展时，它用服务调用进入用户级代码；当代码返回后，内核继续运行该扩展。伴随着内存保护，内核能对扩展时间分片以确信它没有独占 CPU。如果运行时间太长，内核能够主动放弃它。这种模式的主要缺点是有一个与服务调用相关的开销；对小的扩展，每次的调用开销与运行扩展相比可能非常大。

(2)*软件保护*　软件保护能够提供三种选择中最高的性能，其放置扩展代码到与内核同样地址空间里，但对扩展部分进行约束。靠控制语言(如 Modula－3)、编译器或可执行代码补丁的扩展，我们能控制扩展指令。这允许我们确保扩展不能对它范围外读或写，或跳到任意的内核代码段[11]。另外，我们能确保它不能发出放弃中断或初始化 I/O 操作的指令。

它需要可控的和信任的语言翻译工具来处理扩展源。生成正确的代码保证内核的安全性；如果翻译工具能够通过欺骗产生不安全代码，内核的安全性就不能够保障。例如，在特定的环境下，一个编译器的数组范围检查可被破坏，能允许写代码重写自己的堆栈和围绕内核的安全机制。许多的现代语言，诸如 Modula－3和 ML，比 C 语言更加安全。在一个安全类型和指针安全性语言里，构造一个指针指向一个任意内存地址和留下一个"不确定"的指针到已经释放的内存，都是不可能的事。如果类型构造是可利用的，它与编译—时间分析或一个例程类型检查相结合以保证构造是有效的。同样地，数组的范围在访问时被检查以确保一个程序不能存取数组之外的数据。

虽然定义一个语言可以确保类型和指针的安全，语言工具对编译和运行语言可能不会正确地实现定义。使用现在的技术，很难证明一个 Modula－3编译器的正确性，因此，我们不能确信一个编译器不会产生错误代码。即使翻译工具能正确地工作，内核仍然需要证明将代码载入内核是可靠的，靠一个信任的语言进行实际处理，靠执行翻译，或靠标记代码(用一个加密检验和)在装载时就能完成，前一项技术在扩展装载时花费高，后一项技术意味着我们能够加入一个秘密的关键值在语言工具里。并且用关键值标出生成的代码。

软件灾难隔离(SFI)的目标是使它更加有效，以确保使用软件进行内存访问比用硬件更有效[13]。一类 SFI，如沙箱模型，确保一个内存地址的高位与沙箱分配功能或模块的区域相匹配。用这种方法，一个模块能在最坏情况下，用指针陷入或跳到它自己的代码里重写它自己的数据，沙箱模型可在编译时被执行或在目标文件当作一个处理过的段执行。允许从编译器分离沙箱工具。在装载时，一个线形时间算法能用来保证在一段目标代码里所有的内存引用已经被正确地沙箱化了。

(3)*解释*　我们能构造一个虚拟机或解释器来运行扩展。对一个实际的或虚拟的机器，源语言(C、Modula－3，或其它语言)能编译生成中间语言或机器代码。内核将包含一个解释器来运行该代码，确保它的安全性。靠在解释器里执行唯一的安全操作方式，允许对扩展的性能进行全部控制。

中间代码也能被用作一个运行时代码生成器的输入。一个快速的解释器也比编译的代码运行慢10到100倍。但是，使用增量代码生成技术，性能将会达到编译的代码性能。

对于 Java 虚拟机，Java 被编译为一个压缩字节代码，当作解释器运行时，Java 解释器是相当快的，另外，Sun 计划在未来对 Java 发行一个例程代码生成器，使被编译过的 Java 将与编译过的 C 或 C++运行在同样的速度上。

另一项用来建造解释语言的技术是不进行源到中间格式的转换，直接解释该语言，这项技术用在 awk、sh 和 Tel 中，用每条语句的一个较高的开销，来实现一个较小的启动时间。

4.4　粒度

扩展的粒度与扩展密切相关，它分为两个层面：内核接口提供给扩展的一个功能(内部接口)，通过该接口应用存取扩展的接口(访问接口)。内部接口指示应该如何自约束扩展，访问接口只是合理地实现一个扩展到功能类型。

内部接口允许访问：应用可利用接口(如，系统调用)，一个内部内核接口的子集，或所有的内核接口。当暴露的内核进入点数增加，扩展与操作系统间集成更紧密，减少了扩展的大小和复杂度，但要求在扩展部分的设计上更复杂，对扩展的增加潜力与未来内核版本构成矛盾，并且增加了为防止错误扩展行为的保护开销。

通常采用三种方法修改访问接口：替换或者过载现存的内核进入点，替换或过载内部内核进入点，或使用通常的消息来与一个扩展进行通信。

(1)*粗粒度*　粗粒度扩展是粗糙的模块式扩展，因为每一个扩展是自约束的，只使用可为应用利用的内核接口。一个常规的微内核是一个粗粒度扩展系统的例子，在微内核结构中，一个完整的服务器可被替换，但这个服务器必须是一个自约束的单元，只利用为另外的用户级进程可用到的调用。

(2)*细粒度可扩展性*　是靠提供使一个扩展能覆盖任何内部进入点的机制来支持。例如，当操作系统作为一个用户级应用库(如 Exokernel)被实现时，一个应用能替换任何它所拥有的一个单独的内核模块。所以，它可能用应用定制的模块对内部的和外部的接口都进行替换。因为扩展能被定义在任意小的单元里，因此我们称这种结构为细粒度的可扩展性。

(3)受限的细粒度可扩展性 当一个系统替换内核里的扩展并使用软件保护,细粒度的可扩展性常常是不切实际的,因为它意味着在内核里的每一个例程必须被保护。如果任何的程序能够被替换,那么任何内核对预想的安排在调用一个程序前或者在调用一个程序后都是无效的。结果是,软件保护系统通使用一个细粒度可扩展性的特殊情况——受限的细粒度可扩展性,它只允许内核功能的一个子集可扩展。在这样的系统里,难题是识别所有有用的可扩展点并确保它们的替换不会使系统的预想无效。

5. 扩展操作系统方法的归纳

表2是我们对几个典型的可扩展操作系统用四个关键设计点进行的技术分析总结。由于可扩展操作系统在具体实现时是多种多样的,可能同时采取多种设计方法和技术,对具体分析带来了无比的复杂性,因此,为突出重点,这里只列出了它所采用的主要方法。

表2 典型的可扩展操作系统设计

系统	位置	程序生成	保护	粒度
Apertos	内核	替换	软件	细粒度
	库	替换	硬件	细粒度
Exokernel	内核	替换	软件	程序上受限的
	库	替换	硬件	细粒度
Cache Kernel	服务器	替换	硬件	粗粒度
Choices	内核	替换	解释	粗粒度
Hydra	内核	选择	硬件	粗粒度
Scout	内核	选择	软件	细粒度
SPIN	内核	替换	解释	受限的细粒度
Spring	服务器	替换	软件	粗粒度
Synthetix	内核	推理	解释	细粒度
VINO	内核	替换	软件	受限的细粒度

结束语 虽然可扩展操作系统的设计方法不尽相同,但它们一般都能归纳到本文提出的可扩展操作系统四个关键设计点中,对于特定的应用需求,需要在继承传统操作系统优秀成果的基础上,通过综合考虑和利用这四个关键设计点,解决操作系统的性能和柔性问题。但是,今天的可扩展操作系统在实际应用中还面临着一系列的难题,最主要是系统的自适应性和可靠性,这都是今后操作系统研究中需要重点解决的问题。

参 考 文 献

1 Bershad B, et al. Extensibility, Safety, and Performance in the SPIN Operating System. In: Proc. of the Fifteenth Symposium on Operating Systems Principles, Copper Mountain, CO, Dec. 1995. 267～284

2 Chen B, et al. The Measured Performance of Personal Computer Operating Systems. ACM Transactions on Computer Systems, Feb. 1996. 3～40

3 Cowan C, et al. Adaptable Operating Systems. reference to paper in this same collection

4 Cao P, et al. Application-controlled File Caching Policies. In: Proc. of the 1994 Summer Usenix Technical Conf. Boston, MA, June 1994. 171～182

5 Campbell R, Tan S M. mChoices: An Object-Oriented Multimedia Operating System. In: Proc. of HotOS V, Orcas Island, WA, May 1995. 90～94

6 Engler D, Kaashoek F, OToole J. Exokernel: An Operating System Architecture for Application-level Resource Management. In: Proc. of the Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain, CO, Dec. 1995. 251～266

7 Fall K, Pasquale J. Exploiting In-Kernel Data Paths to Improve Throughput and CPU Availability. In: Proc. of the 1993 Winter Usenix Technical Conf San Diego, CA 1993, 327～334

8 Fiuczynski M, Bershad B. An Extensible Protocol Architecture for Application-Specific Networking. In: Proc. of the 1996 USENIX Technical Conf. San Diego, CA, 1996. 55～64

9 Ganger G R, Kaashoek M F. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In: USENIX 1997 Annual Technical Conf. Jan. 1997

10 Golub W, et al. Mach: A New Kernel Foundation for UNIX Development. In: Proc. of the Summer 1986 USENIX Conf. July 1986. 93～12

11 Liskov B, et al. Theta Reference Manual. MIT LCS Programming-Methodology Group Memo 88, Feb. 1995

12 Seltzer M I, Endo Y, Small C, Smith K A. Dealing with disaster: Surviving misbehaved kernel extensions. In: Proc. 2nd Symp. on Operating Systems Design and Implementation, Seattle, WA, Oct. 1996

13 Wahbe R, Lucco S, Anderson T E, Graham S L. Efficient software-based fault isolation. In: Proc. of the Fourteenth ACM Symposium on Operating Systems Principles, 1993. 203～216

(上接第153页)

4 Kinny D, Georgeff M, Rao A. A methodology and modelling technique for systems of BDI agents. Proceed-ings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038). Springer-Verlag, Berlin, Germany, 1996

5 Bauer B, et al. Agent UML: A formalism for specifying multiagent software systems. In: Proc. of the First Intl. Workshop (AOSE-2000). Springer-Verlag, Berlin, Germany, 2000

6 刘大有, 杨鲲, 陈建中. Agent 研究现状与发展趋势. 软件学报 2000(11)