集成保护和分散式抽象的可扩展操作系统*)

施笑安 周兴社 林 奕 李 晖
(西北工业大学计算机科学与工程系 西安710072)

Integrated Protection and Decentralizing Abstractions with Extensible Operating Systems

SHI Xiao-An ZHOU Xing-She LIN Yi LI Hui
(Dept. of Computer Science and Engineering, Northwestern Polytechnical University, Shanxi, Xian 710072)

Abstract Protection and abstraction are traditional advantages of operating system that provide for application-programmer. How to change Protection and abstraction in order to ensure the new Extensible Operating Systems having high performance and good flexibility is the most crucial problem to build Extensible Operating Systems. We bring forward a new approach--Protected Decentralizing abstractions, integrating Decentralizing OS abstractions and Protected abstractions, and use it to implement a prototype library operating system -- EXLinux /LibOS. A novel way to safely share user-level abstractions in the prototype, enables unprivileged and untrusted applications to be defined and securely shared generic abstractions at run-time, in which the same abstraction is invoked repeatedly.

Keywords Decentralizing abstractions, Protect, Library operating systems, Kernel

1 概述

传统操作系统为程序员的开发应用提供了两个特别的好处:抽象和保护。抽象提供了一个很方便的硬件接口,使用抽象来虚拟资源以支持共享,使用户产生了没有共享资源的错觉,提高了系统的一致性并增强了功能。选择哪一种内核抽象是很重要的,通常每种类型只有一种单一的抽象支持,并且该抽象通常是难以扩展或替代的;为了支持系统的完整性,防止错误和恶意的任务,保护限制物理资源的访问。保护通常内含在系统提供的抽象中,在多任务共享资源时是最影响操作系统性能的因素[3]。如何对现有操作系统的抽象和保护方法进行改造,以保证新一代的可扩展操作系统构造具有高性能和更好的柔性,是当前可扩展操作系统设计时所面临的主要问题。

可扩展操作系统由一套接口和实现组成[1],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。可扩展操作系统设计目标主要是:提高操作系统的性能,使系统能更好地满足应用需求,并提供更多的功能;提高操作系统的柔性,使操作系统适应应用需求的不可预测性或特殊的功能[7]。

本文提供一种新的集成保护和分散式抽象的方法设计可扩展操作系统,分四个部分进行了论述。首先对传统的操作系统抽象进行了讨论,其次,具体介绍了可扩展操作系统中使用的分散式操作系统抽象方法。接着我们针对分散式抽象设计了一种保护分散式抽象的方法。最后使用该方法设计了一个可扩展操作系统的原型——Exlinux/LibOS。

2 传统操作系统抽象方法

传统操作系统(包括微内核)的抽象被分为四个部分:进程抽象、虚拟内存抽象、文件系统抽象和通讯抽象。这四部分抽象包含了七个抽象语意:保护、安全、并发、一致性、命名、原子性和持久性,无论在什么机制中实现抽象都必须考虑这些特性。

但传统操作系统在简化操作系统抽象实现的同时也限制了灵活性和性能。只提供一组抽象从而限制了灵活性。性能也因为不灵活和不柔性而受到限制[6]。如标准操作系统对系统的抽象设置所有相关的状态到特权级地址空间,使用户级的程序只能够通过一个固定的系统调用接口进行存取[9]。这个集中模型使它容易确保共享状态的一致性,并保证非授权的访问不能发生。然而,粗糙的接口迫使任务使用预定义抽象目标来预先决定策略,这明显地导致性能的降低[6,9]。

这些问题可以通过分散操作系统各部分来解决。使用适当的分布式控制,应用程序可以根据它们的需要更好地使用抽象。

3 分散式操作系统的抽象方法

利用分布式的固有特性可实现操作系统抽象及其特性。为更好地解释在分布式下如何实现抽象及其语义,下面对分布环境下使用的分散式抽象机制及其实现进行了具体讨论。

分散式抽象的可扩展操作系统基本特征如图1所示,它有如下特征:

·分离:和传统操作系统相反,它对每一个进程的抽象状态分别进行维护。这使进程可以拥有并保护自己的抽象状态。就象在微内核中一样,实际效果是操作系统代码中的错误不再和系统冲突,仅和与其相关的应用程序有关。

·最小化授权:最小化授权遵循最少权限原则。每一个抽象都能对其内部状态进行控制。任何对其它抽象状态进行的修改必须通过明确的接口来实现。

*)本课题得到国防预研基金项目(98J15.15.HK0321),十五国防基础研究项目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、实时嵌入式操作系统、自适应计算。周兴社 博士导师,教授,研究方向为网络与分布式计算。林 奕 博士生,研究分布式计算。李 晖 硕士生,研究分布式计算。

·局部信息:可以使分布式操作系统的大部分工作将使用局部信息.这样具有可靠性和可升级的优点.抽象将更少地依赖于中央信息库.

·分离改变:和传统操作系统的单点修改相反,分布式操作系统分离改变.这意味着一个进程抽象实现的改变并不直接影响其它进程.这使得一个抽象可以根据具体使用来优化,给灵活性和性能的改善留有余地.

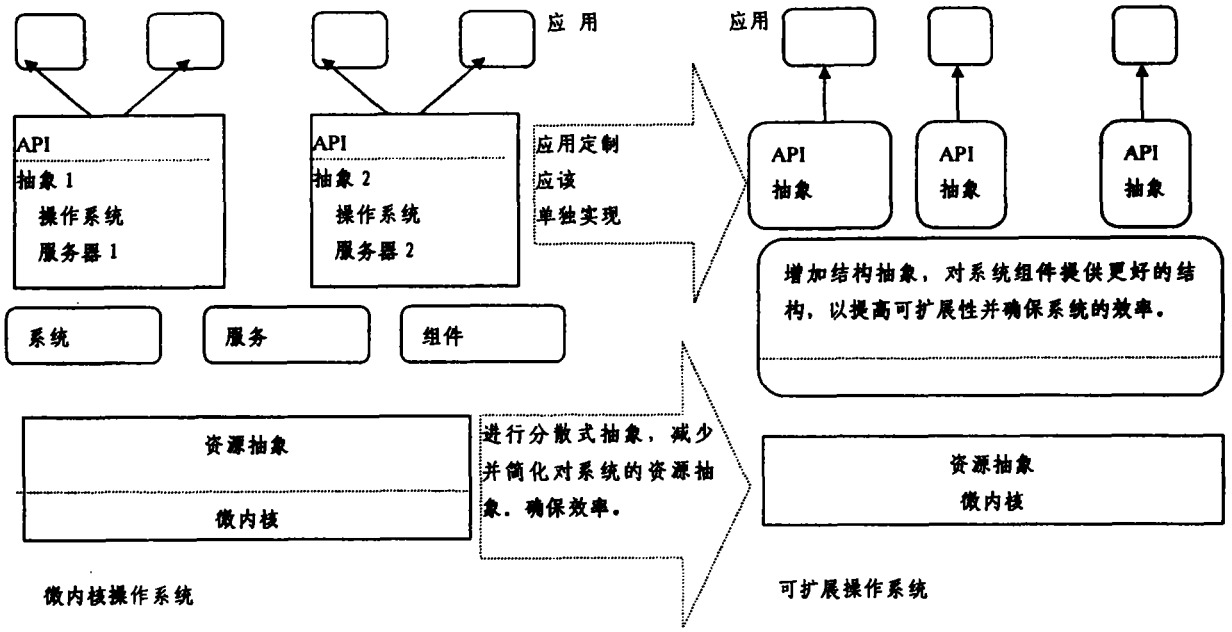

图1 分散式抽象的可扩展操作系统基本特征

3.1 分散式系统抽象的特性

有了这些共享状态的方法,可以实现传统操作系统的一些重要特性:

·保护 把抽象状态存储在不同的保护域可以实现保护.或者对抽象使用分离的保护域,或者进程拥有多个保护域,每个域包含抽象的一部分[4].在同一抽象中提供了错误分离.

·并发性控制 并发性控制通过标准分布式算法或者通过传统的操作系统的临界区、非封锁同步等单个点上的串行化来实现.

·访问控制 访问控制可以通过最小权限实现.每一进程或抽象都有对自己状态访问的权利.如果需要修改或作用于另一抽象,必须通过明确的接口并且拥有信用证才能实现,这个方法提供了比传统操作系统更好的安全性,因为错误的影响只限于相关的抽象.

·一致性 获得一致性的最简单办法就是仅保留一份数据备份.或者把所有数据放在一个位置,或者使用注册表把数据放在不同位置.当一个应用程序需要访问新数据时,它将检查注册表.注册表指向数据的拥有者或其位置,或者对不知道的信息做出反应.

·原子性 原子性的实现同传统的实现类似.在传统操作系统中,通过把抽象状态隐藏起来直到原子操作完成来实现原子性.如果原子操作被取消,可以通过全局信息和扩展的权限来回滚任何抽象状态.在分布式操作系统下缺乏这些特性,这使得修改多个抽象时实现原子性变得困难.

·持久性 通过集中的方法或令牌的形式可以实现状态的持久性.集中的方法,即通过状态的生存期或者在一个没有其它进程访问的点上,服务器保存稳固的状态.

·命名 可以在一个中心位置进行名字解析,或者可以传递请求直到解析出名字.使用一个传统的名字服务器,注册不同进程及抽象能解析的名字.相反,可以只注册根,它将把不能解析的名字通过服务器传递给其它进程来解析.

从上面可以看出传统的操作系统过于严格,不允许用户级任务修改操作系统的抽象,分散式抽象操作系统结构通过将保护与管理相分离的方法,安全地使不信任的任务有效地控制硬件和软件的资源.

3.2 分散式抽象的可扩展操作系统

该结构通过提供给不信任任务足够的对资源的控制来解决这个问题,它划分了传统操作系统界面的任务,将保护和管理分离:它们保护资源的同时把管理授权给任务.例如,每一个任务管理它自己的磁盘块高速缓存,但是可扩展操作系统允许在缓存中的页能被所有的任务安全地共享.因而,可扩展操作系统保护页和磁盘块而任务管理它们.

当然,并不是所有的任务需要用户化的资源管理,而是直接和可扩展操作系统交换信息,我们希望大多数程序和资源库联系起来,这些资源库包含了在传统操作系统抽象之下的底层资源.但是,与传统的系统抽象使用方法不同,资源库的使用不是基于特权的,因而能随时对它进行修改和替换.我们称不带特权资源库为库操作系统(LIBRARY OPERATING SYSTEM,LibOS).我们希望,可扩展操作系统的管理方式有助于操作系统的革新——面向任务的程序员比操作系统使用者在人数上多出几个数量级,每一个程序员能够自定义一个LibOS 而不会影响到系统的其它部分.LibOS 也允许增量地、有选择地吸收操作系统新的特征——任务与 LibOS 相连并从中获取资料,这样操作系统新的功能性被有效地传播给任务.

然而分散式控制,使它对互相不信任的任务共享系统的抽象非常困难,同时,在分散式抽象系统中,这种操作系统抽象用库来实现,并将它们的状态存放在共享内存中.这些抽象不能损害用户保护的内存区.不幸的是,这种机制常常不能提供所需的性能[2].下面我们提出设计和实现保护分散式抽象

机制的一种方法，在分散式抽象系统里共享用户级的抽象，通过提供存取方式接口保证它们的状态安全，任务能安全地共享高级的抽象。

4 保护分散式抽象方法

保护分散式抽象是一种新的安全方式，在分散式抽象系统里共享用户级的抽象。能够使非特权、不信任的任务在运行时进行定义并安全地共享生成的抽象。

4.1 保护分散式抽象思想

保护分散式抽象机制接近与数据封装的概念，基于面向对象的程序设计语言，一个抽象构成一个对象，它的状态位于一个保护域，只能靠抽象的访问/保护方法被读或写。然而，保护分散式抽象的接口及其状态都能在运行时定义。该机制提供了较好的柔性，并消除了在内核里设置性能临界抽象的需要。保护抽象的状态使用分散式抽象的分级命名能力保证，一个抽象的状态与一个能力相连，它将允许用户的任务随时调用抽象接口里的一个方法。没有这样的能力，进程不能存取抽象的状态。保护方式对状态拥有全部的权限并能毫无阻碍地执行它们的任务，内核保证保护方法的控制传递。使系统中除了这些方法的其它的实体也能在拥有该状态能力时运行。

4.2 保护分散式抽象基本设计

保护分散式抽象机制在分散式抽象的硬件保护机制之上构成一个软件保护层。硬件保护层确保硬件抽象，而保护方法保证软件抽象。一个保护的软件抽象由两部分组成：

·状态——对内存页、磁盘块等资源，抽象的定义希望能被系统中别的进程安全地共享，安全地共享意味着在内核提供的保证下进行共享，关于该状态的一套确定的常量将被保持。这些常量被抽象的方法明确定义。

·保护方法——一组进程，能查询和更新状态，这些进程是对保持常量的响应，它们定义抽象的组成并对保护的抽象提供一个完全的接口。

保护分散式抽象使一个抽象的状态只能用于该抽象的方法，这保证了状态的一致性，在更新时和存取时规则被严格遵守。当使用一个抽象调用一个保护方法，内核接受该进程状态的能力并使用将能力控制传递到请求的方法。该方法运行在调用的上下文中，并能更新所需的状态。保护方法可消除能力并取消绑定抽象的状态，这确保了只有保护方法有能力存取抽象的状态。

4.3 保护分散式抽象的系统支持

传统的操作系统通过保护许多资源的方式提供它们的高层抽象。而可扩展操作系统允许任务直接进入底层资源，可扩展操作系统必须能提供象 UNIX 一样的保护，包括在需要安全保护的高层实体上提供进入控制。应用保护分散式抽象思想，系统抽象的关键是它们既不妨碍硬件资源的底层进入，也不过分强调抽象保护的条件。机制如下：

·它对所有的资源采用同样的方法进行控制。

·可扩展操作系统将软件抽象与硬件资源绑定在一起。

·可扩展操作系统向抽象中载入代码，而不采用映射硬件抽象的方法。

4.3.1保护共享 底层的可扩展操作系统界面提供给 LibOS 对硬件足够的控制权，有利于使用传统操作系统抽象。库使用方式最大的好处是，它们能够相信与它们相连的任务，而不用防备恶意的使用。但是，LibOS 没有必要信任其它所有的 LibOS。当 LibOS 要保证它们的抽象不变时，它们必须注意抽象中包括了哪些资源，其它哪些进程已经进入这些资源，和它们设置在这些进程中的信任级别。可扩展操作系统提供四种机制保证 LibOS 在使用共享抽象时能保持不变。

·软件的作用区域：只能通过系统调用对内存区域进行读和写操作，内核提供子页保护和出错隔离。

·允许等级性命名的方式，并要求在每一个系统调用中明确声明。这样，当子进程偶然错误地申请写父进程的页或软件区域时将会产生权限错误，要求被拒绝。

·采用预唤醒技术：采用内核代码载入技术，当一个进程所需的特殊条件成熟时，立即唤醒它。这保证了有问题的或冲突的进程不会挂起正确的进程。

·在进程作用区域之间提供健壮的临界管理，它禁止软件中断而且代价低。用它我们就不用考虑是否解除与其它进程的互相信任。

4.3.2信任机制 采用下述三级信任机制将最优实现共享抽象。

·共同的信任：任务在共享资源，互相之间放置大量的共同的信任。

·单向的信任：有时在进程之间会发生如下情况：它们共享资源，其中一个信任另一个，但是这种信任不是共同的。

·对共同的不信任采用防御性策略：对于能够被互相不信任的进程共享的操作系统抽象，LibOS 必须采取防御性措施并对其它可能对其采取行动的进程作出合理的解释[6]。

5 系统实现

基于保护分散式抽象的可扩展操作系统为各种应用提供了比较多的原始硬件接口，并允许应用提供各自的操作系统抽象和接口。它的设计特点也就是给予应用尽可能多的对硬件资源的控制权限。这种设计去除了大部分的传统的内核的抽象，而由用户层的库操作系统(LibOS)来提供这些抽象。每一个应用都与开发者享用的 LibOS 相连并成为用户空间代码的一部分。

采用库操作系统方法，我们在对 Linux 源码改造的基础上设计和开发了一个可扩展操作系统原型——EXLinux/LibOS，它将资源的管理和保护分离，以前保留在操作系统中的抽象和策略被移入 LibOS 库，操作系统只保护资源，极大地提高了操作系统的性能。并允许应用制定自己的操作系统抽象，同时，对相同的抽象重复调用时，消除了上下文切换的需求并对通常情况进行优化，可获得较好的柔性。

EXLinux 管理绝大多数的标准机器资源：CPU、内存、网络和磁盘，它的缺省的库操作系统是 LibOS。EXLinux 保护各种硬件资源，包括环境、保护域、时隙、硬件页起始、网络设备、块、注册和控制台。这些资源通过使用层次命名能力而被保护。层次命名包括：性能和名字。性能定义了如合法性，修改/删除，分配，写和名字长度等特性。能力有可变长名字，这些名字被用来定义一个能力间的重域关联[6]。

总结 在该系统中，集成了分散式操作系统抽象和保护抽象的方法，使用这种新的安全方式在操作系统里共享用户级的抽象，能够使非特权、不信任的任务在运行时定义和安全地共享生成的抽象。它在对相同的抽象重复调用时，消除了上下文切换的需求并对通常情况进行了优化，可获得较好的柔性。保护分散式抽象的设计强调简单并正确性证明，易于理解

(下转第112页)

过程，当用户在某个图形元素上移动鼠标，并且按住鼠标左键时，图形元素随着鼠标的移动而移动。

UML 对面向对象设计的表述能力相当强大，可以较好地描述 FVI 模型的各个层面。FVI 的层次化和模块化结构可以较好地与 UML 各个粒度的设计工具相结合，较好地融入到整个软件的设计过程中。通过 UML 用例图可以很清晰地导出 View Model，Frame Controller 以及 Core Interface。利用 UML 的包 Package 对模型进行分解细化。

当然，在使用 UML 进行 GUI 设计时，也存在着一些局限和问题。比如，UML 用例图对用户界面需求的描述不够清晰和明确，这导致在由用例图导出界面模型时容易产生偏差。用例图与其他静态视图和动态视图的对应关系没有明确描述。UML 对并发过程控制的约束关系的描述能力较弱，在使用 UML 描述用 View 和 Frame Controller 内部的动态行为时，存在着设计者的隐含预定和假设。实际上，这些问题都是由于 GUI 设计的特殊性所造成的，为此众多学者也提出了各种方法和手段，如 Essential User Case 设计方法[7]，使用 Petri 网或形式化语言描述界面动态行为等[8]。但在实际应用中，UML 被认识是最有效、最直接的设计建模语言。

结束语 FVI 模型以视图 View 为设计核心，采用逐层递阶的思想，反映了在用户界面设计中自顶向下的设计思路，为 GUI 界面的需求分析、界面结构化和层次化设计，以及界面动态交互任务设计提供了逐步细化的面向对象的设计框架模型。FVI 模型将 GUI 设计的中心放在了 View Model 上，将 GUI 逐步分解为层次化和模块化的子视图，可以从各个层次和角度去描述 GUI 界面的层次、结构和动态交互。相应地把用户任务看作是由多个视图的内部行为的组合，给予可视化组件更多的自主权，适合于面向对象的系统设计和实现。

在实际应用中，使用 UML 设计 GUI 模型，为设计人员提供了有效的 GUI 设计语言，可以较好地弥合界面设计与核心应用之间的分歧，使设计人员可以使用统一的设计语言和工具对软件进行整体规划和设计，从而提高了软件设计的质量和效率，减少了设计人员对需求的理解偏差、设计人员之间的理解偏差，以及设计人员与实现人员之间的理解偏差，有利于软件体系结构的扩充和维护，从而提高软件开发的质量、健壮性、可测试性、可维护性和可扩展性。

GUI 设计中还包括大量的图形界面交互行为的设计，如图形控件交互设计、对话框交互设计等等。限于篇幅有限，本文未作详细讨论。实际应用中利用 UML 语言可以较好地描述界面构件的动态交互行为。

通过实际应用，我们感觉到使用 UML 实现面向对象的 FVI 设计模型，明确了 GUI 设计在软件体系结构中的位置，为设计人员提供了 GUI 设计的依据和方法，提高了软件设计文档的完整性和一致性。尽管使用 UML 描述 GUI 交互行为存在一定的局限性，但考虑到 UML 语言的特点以及 GUI 设计与核心应用设计的统一，UML 仍是最有效的 GUI 模型描述工具。对于 UML 在 GUI 设计上的局限，可以通过对 UML 扩展的方式加以弥补，以不断完善对 GUI 设计的支持。

参考文献

1 Myers B A, Hudson, Pausch. Past, Present, and Future of User Interface Software Tools. ACM Transactions on Computer-Human Interaction, 2000, 7(1): 3～28
2 Myers B A. Why are Human-Computer Interfaces Difficult to Design and Implement?: [Research Reports, CMU-CS-93-183]. Computer Science Department, Carnegie Mellon University, 1993
3 Bomsdorf B, Szwillus G. Tool Support for Task-Based User Interface Design. A CHI'99 Workshop, 1999
4 Coutaz J. Software Architecture Modeling for User Interfaces, in Encyclopedia on Soft-ware Engineering, Wiley, 1993
5 Hussey A, Carrington D. Comparing Two User-Interface Architecture: MVC and PAC. TECHNICAL REPORT NO. 95-33, Software Verification Research Centre, Dept. of Computer Science, The University of Queensland
6 涂序彦. 大系统控制论. 国防工业出版社, 2000
7 Constantine L L. Essential Modeling: Use Cases for User Interfaces. ACM Interactions, 1995, 2(2): 34～46
8 Elkoutbi M, Keller R K. User Interface Prototyping based on UML Scenarios and High-level Petri Nets. Application and Theory of Petri Nets 2000 (Proc. of 21st Intl. Conf. on ATPN), Aarhus, Denmark, Springer. LNCS, 2000. 166～186
9 Rumbaugh J, Jackbson I, Booch G. The Unified Modeling Language Reference Manual. Addison-Wesley, MA, 1999
10 OMG. OMG Unified Modeling Language Specification. in Web Site, No. Version 1.3, Object Management Group, 1999. (URL http://www.omg.org)
11 da Silva P P, Paton N W. UMLi: The Unified Modeling Language for Interactive Applications. UML2000 - The Unified Modeling Language. Advancing the Standard. Third Intl. Conf. York, October 2000, Proceedings
12 Silva P P da, Paton N W. User Interface Modeling with UML. The 10th European-Japanese Conference on Information Modeling and Knowledge Representation, May 2000
13 Coutaz J. PAC, an Object Oriented Model for Dialog Desgin. In: H. I. Rullinger, R. Shacked, eds. Human-Computer Interaction-INTERACT'87, Elsevier Science Publishes, 1987
14 Griffiths T, McKirdy J, Forrester G, et al. Exploiting model-based techniques for user interfaces to database. In: Proc. of VDB-4 Italy, May 1998. 21～46

（上接第107页）

和使用，并从根本上实现了可扩展操作系统的高效性和柔性。

参考文献

1 Anderson T. The Case for Application-Specific Operating Systems. In: Third Workshop on Workstation Operating Systems, 1992. 92～94
2 Banerji A, Tracey J M, Cohn D L. Protected Shared Libraries -- A New Approach to Modularity and Sharing. In: Proc. of the 1997 USENIX Technical Conf. 1997
3 Bershad B N, et al. Extensibility, safety and performance in the SPIN operating system. In: Proc. of th Fifteenth ACM Symposium on Operating Systems Principles, Dec. 1995
4 riceno H M. UNIX Abstractions in the Exokernel Architecture. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1997
5 Candea G M, Jones M B. Vassal: Loadable Scheduler Support for MultiPolicy Scheduling. In: Proc. of the 2nd USENIX Windows NT Symposium, to appear, 1998
6 Hildebrand D. An architectural overview of QNX. In: Proc. of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures, April 1992
7 Deutsch P, Grant C A. A Flexible Measurement Tool For Software Systems. Information Processing 71
8 Engler D R, Kaashoek M F, O'Toole J. Jr. Exokernel: an operating system architecture for application-Specific resource management. In: Proc. of the Fifteenth ACM Symposium on Operating Systems Principles, Dec. 1995
9 Kaashoek M, Engler D, Ganger G, et al. Application performance and flexibility on exokernel systems. In: Proc. of the Sixteenth ACM Symposium on Operating Systems Principles, 1997. 52～65