维普资讯 http://www.cqvip.com



# 基于用户态 JVM 的安全驱动模型的设计与实现*)

殷一鸣[1] 周玲玲[1] 应忍冬[1] 戈 弋[2]
(上海交通大学电子工程系 上海 200240)[1] (IBM 中国研究院 北京 100094)[2]

**摘 要** 设备驱动等内核扩展模块被认为是造成操作系统不稳定的主要原因,而目前对操作系统稳定性的研究也主要集中在研究如何将其与内核的主体部分分离。本文结合现有的研究成果,提出了利用用户态的 Java 虚拟机(JVM)将驱动程序与内核分离的架构。在这样的架构中,驱动程序运行在受限的 JVM 中,所有驱动程序的错误都将被捕获而不致影响到内核的其他部分。利用这样的架构,在 Linux 系统下实现了新的 USB 协议栈,并对其进行了性能测试。测试结果表明,与原内核驱动程序相比,新的驱动架构表现出更高的稳定性,而在性能方面,对于时间和数据量要求不高的设备,其性能与原驱动程序相当接近;而对于需要进行大量数据处理的设备,其性能有一定程度的下降。

**关键词** 操作系统,稳定性,驱动程序,错误隔离

## The Design and Implementation of a Safe Device Driver Model

YIN Yi-Ming[1] ZHOU Ling-Ling[1] YING Ren-Dong[1] GE Yi[2]
(Dept. EE, Shanghai Jiaotong Univ., Shanghai 200240)[1] (IBM China Research Laboratory, Beijing 100094)[2]

**Abstract** Kernel extensions, such as device drivers in operating systems, are proved to be much more unreliable than other parts of kernel. Recent researches on OS reliability are focusing on the isolation of extension modules from kernel. In this paper, a new architecture is proposed to isolate device drivers. Drivers are restricted in user-mode java virtual machine (JVM), which is an independent user-process. Bugs in drivers will be captured and well handled by JVM, and therefore, prevented from ruining the whole kernel. A new USB protocol stack in Linux is implemented based on this architecture. Performance test reveals that new architecture ensures a more reliable operating system with an acceptable performance overhead compared with traditional drivers.

**Keywords** Operating system, Reliability, Device driver, Isolation

## 1 引言

随着计算机普及率越来越高,个人电脑(PC)的使用者从原来从事计算机领域工作的人拓展到许多非专业领域的人群,其工作领域也从原来的专业开发应用拓展到日常生活,现在 PC 已越来越多地扮演了和电视机等家用电器一样的电子消费品的角色。

然而,PC 像电视机一样稳定吗?答案是否定的。和能够稳定运行 10 年,最后仅仅是由于器件老化而报废的电视机相比,PC 的稳定性相去甚远。其原因在于,PC 上运行着操作系统软件,而正是由于操作系统的不可靠性,造成了 PC 系统的不稳定[13]。

### 1.1 操作系统的不可靠性

一项对软件可靠性的研究表明[8],视软件模块规模的大小不同,每千行程序有 2 到 75 个不等的错误。如果把这个比例应用到超过 250 万行代码的 Linux 操作系统上,即使按最少-2 个错误每千行代码计,也至少有 5000 个错误。操作系统规模的不断扩大,使得其中包含的软件错误也随之增长。

目前操作系统中的错误更多地是源于书写不规范并且没有经过严格测试的设备驱动程序。操作系统包含了为数众多的设备驱动,用来支持品种繁多的各类外设。而这些驱动的质量参差不齐,并且有很多没有经过专业的测试。它们之中包含的错误要 3~7 倍于上面提到的比例[2]。这些设备驱动的存在,大大降低了系统的稳定性。

然而,以上大量存在于程序中的错误造成系统崩溃最终要归因于现在操作系统的单内核设计[5]。出于性能的考虑,目前主流操作系统都采用了单内核的设计。在这种设计中,内核的所有模块都是运行于同一个内存空间(内核空间)中,且处于特权模式,内核中的任何数据结构和内存地址对内核中的所有模块都是可读写的。这也就意味着有缺陷的内核模块可以肆意破坏整个内核空间,从而造成不可恢复的内核崩溃。

### 1.2 将错误从内核中分离

从以上的分析可以看到,由于在操作系统内核中没有采取有效的隔离,含有错误的模块能够将错误传播到整个内核。如果将这些模块与内核空间分离,将会有效地减少内核的崩溃[4]。鉴于目前大量存在的驱动程序是造成内核崩溃的主要原因,研究如何将驱动程序产生的错误与内核隔离,将会提高操作系统稳定性。因此我们提出了这样的系统架构:将设备驱动程序从内核中分离出来,并利用类型安全的 Java 语言改写,使其运行在用户态的 Java 虚拟机(JVM)中,而 JVM 与内核中硬件抽象层进行交互,完成数据处理和设备管理。这样,

*)该项目得到了 IBM 中国研究院大学合作项目的资助。**殷一鸣** 硕士研究生,研究方向为嵌入式系统;**周玲玲** 副教授,主要研究方向为通信系统中的可用性和可靠性、软件可靠性以及嵌入式微处理器在应用领域的研究和开发;**应忍冬** 博士研究生,研究方向为嵌入式系统;**戈 弋** 博士。

维普资讯 http://www.cqvip.com

驱动程序的所有错误,都将被JVM捕获,并进行适当的处理,使其不影响内核的正常运行。即使最终产生了严重的错误导致JVM崩溃,但由于JVM仅仅是一个普通用户进程,拥有独立的虚拟地址空间,因此也不会影响到内核的安全运行。我们期望这样的架构可以有效地减少不安全代码对系统内核的影响,使得现有的操作系统更为稳定和可靠。

### 1.3 论文结构

本文第2节讨论相关的一些研究项目及成果,第3节给出系统架构,第4节描述整个系统的实现细节,第5节中给出测试结果。最后给出总结。

## 2 相关研究

目前,对操作系统稳定性的研究主要集中在对易出错的设备驱动程序的隔离上。隔离手段主要有两个方向:一是在现有的系统结构上进行修补,增加一些监控模块,使得一些错误能够被捕获和隔离,不影响内核的其他部分。这方面的研究有在内核中建立隔离监控层的Nooks[11,10]和基于虚拟机技术的$L^4$Linux[3],这些实现方式都对现有系统提供了很好的兼容性。另一个方向则完全摒弃现有的主流操作系统,研究下一代可靠操作系统的模型,比如MINIX3[6,12]抛弃单内核模式,使用一个作为多重服务器的微内核。微软提出的Singularity[7],同样使用微内核设计,它摒弃现有的主要开发语言C/C++,使用一种类型安全的语言Sing#重新书写全新的操作系统,并使用了软件隔离的思想。

### 2.1 对主流操作系统的改进

Nooks是在尽量不改变现有系统和现有驱动程序的基础上,在系统内核与设备驱动之间插入了一层透明的中间层(Isolation Manager),它解析设备驱动和内核之间所有的交互行为,从而提供有效的隔离和恢复手段。每个驱动程序实际上运行在一个受到保护的内核空间中,它对内核关键数据和内存的写操作是受限的。驱动程序使用的所有内核资源都被中间层记录下来,当驱动程序运行出错时,中间层可以有效地释放其使用的资源并自动重新启动驱动,恢复其原有的状态。而$L^4$Linux则将稍作修改的Linux内核运行在L4微内核之上,并且可以运行多个Linux的内核复本,这样就将驱动程序分离在多个复本中运行,一个驱动程序的错误只会影响到它所在的虚拟机,而不会影响其他的Linux内核进程。这样就起到了隔离的作用。但是它不具备Nooks的恢复功能。

### 2.2 新的操作系统模型

MINIX3是一个使用微内核思想构建的操作系统。系统只提供了一个很小的用于处理中断信号、进程间通信和时钟中断等的微内核。设备驱动程序以用户态进程的形式存在,每个驱动拥有自己独立的进程地址空间,接受进程调度程序的调度。而进程间通过发送和接收固定长度的消息来通信,这样避免了进程共享内存造成的一些访问错误。同时由于驱动程序运行在用户空间,不具有特权级别,并且独立于其他用户进程,因此设备驱动的错误将不会影响到系统的其他部分。Singularity是微软研究院提出的一项研究项目,旨在提高操作系统的可靠性。该系统的可靠性是建立在一个新的类型安全的编程语言Sing#上的。这个系统使用单一的地址空间,但是提出了对象空间的概念,对每个对象实施监控。同时Sing#语言内嵌了进程间的消息传送机制,使得进程间的通信得到语言级的支持。

## 3 系统架构

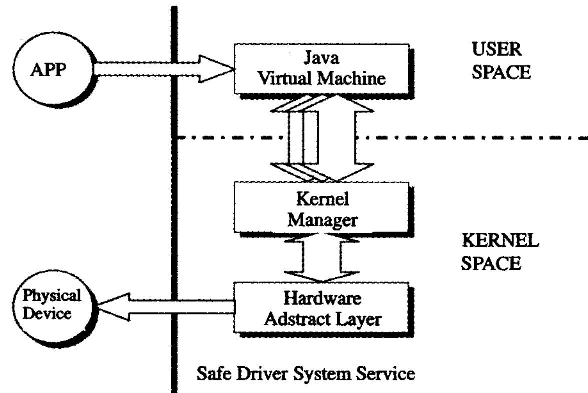


图1 安全驱动系统架构

我们的系统结合了第2节中提到的几项研究的成果。为了保持与现有系统兼容,同时也对下一代安全操作系统的模型做了一些探索性的实践,我们的系统设计采用了分层的结构(如图1)。它由三部分组成,自上而下分别是:运行在用户态的JVM,向应用程序提供设备驱动服务;运行在内核空间的内核管理模块,它是上层JVM和底层的硬件抽象层之间的接口;运行在内核空间的硬件抽象层,直接负责响应上层应用程序的请求以及硬件中断。

### 3.1 用户态JVM

设备驱动程序的主体部分运行在用户态的Java虚拟机中。在我们的系统中,JVM是起隔离作用的虚拟机。仿照上文提到的$L^4$Linux系统,我们也使用虚拟机将驱动程序与内核分离。JVM作为一个普通的用户进程,运行在受限的用户空间中,这样的模式运行起到了很好的错误隔离作用。将代码质量不高的设备驱动程序与内核关键数据分离,限制在Java虚拟机中。这样,即使错误代码造成虚拟机的崩溃,也不会影响到整个内核的运行。

更进一步,在JVM中运行的Java代码要比在$L^4$Linux虚拟机中运行的C代码安全得多。这里的安全性体现在类型安全和内存安全上。Java编译器对变量和对象类型进行强类型检查,以保证数据的操作和对象的引用是正确的;Java虚拟机在运行时管理并监控所有Java程序使用的内存,以保证对内存的引用没有越界且为有效地址。Java语言的安全性大大减少了使用传统C/C++语言编写的程序中存在的错误,随之带来的将是软件系统稳定性的提高。

不仅如此,Java是面向对象的程序设计语言,对设备驱动程序采用面向对象的思想定义结构,将会带来很好的扩展性和整合性。为此,我们使用Java语言开发了一部分底层驱动库,并定义了类结构,利用这些库,驱动程序的书写者们可以很方便地开发出上层设备驱动程序。而Java的通用性又使得开发的驱动程序可以运行在多个不同的平台之上。

### 3.2 内核管理模块

这个模块运行在内核空间中,它抽象出了上层用户态驱动的调用接口,将上层驱动的请求转换成与底层的硬件抽象层的交互。同时,这个模块也是一个监控模块,对于上层驱动的每一次调用,内核管理模块都需要严格检查其合法性。对于有可能产生错误的调用,内核管理模块都将返回一个错误信号给上层的驱动程序。这样就将上层的调用限制在可控的

维普资讯 http://www.cqvip.com

范围内，可以抑制错误代码对内核关键数据的影响。从这个角度，内核监控模块可以看作是用户态 JVM 在内核中的延伸，其监控功能同样起到了隔离的作用。

由于内核模块处于内核空间中，其自身的安全性对整个系统的稳定性有重要的作用，因此内核模块必须经过严格的测试，以保证其与内核的其他模块代码具有同等的可信度。

### 3.3 硬件抽象层

这一层位于整个系统的最底层，接受上层的请求，直接和硬件交互。在现代操作系统中，与硬件直接相关的操作大都需要在特权模式下进行，这是用户态进程无法完成的。因此，我们在内核中设计了硬件抽象层，用来将对硬件的最基本操作转化为可供上层用户态驱动调用的系统调用。同时，出于性能和实时性的要求，诸如 DMA 传输、简单的中断数据收发等任务也在这里完成。由于硬件抽象层处于内核之中，并且与内核的其他部分没有任何的隔离措施，因此这一层应当尽量简单，以最少的代码实现最基本的功能。这样既减少了代码出错的机率，又给上层以更大的灵活性。在 PC 架构中，这一层一般使用经过严格测试并且稳定运行的总线驱动程序。

## 4 系统实现

目前，USB 设备层出不穷，与之对应的驱动也为数众多，而驱动的稳定性和安全性不一。鉴于此，我们基于以上的系统架构，在 Linux 系统下实现了 USB 驱动协议栈。以此替换原 Linux 内核中的 USB 协议栈，可以在此协议栈上进行新的 USB 设备驱动的开发，或者改写现有的驱动。而所有在此协议栈上开发的 USB 设备驱动都将利用类型安全的 Java 语言书写。作为测试，我们移植了 USB 摄像头及 USB 键盘和鼠标驱动。测试结果表明，利用此协议栈书写的 USB 驱动对可能造成内核崩溃的错误有较好的隔离作用。而在性能方面，对于需要进行大量数据处理的设备，与原驱动程序相比其性能有一定程度(约 50%)的下降；而对于时间和数据量要求不高的设备，其性能与原驱动程序相当接近。

### 4.1 USB 协议栈

在第 3 节给出的分层结构中，我们将原 Linux 内核中的 USB 协议栈的不同部分分别移植到我们系统中，同时提供了一个通用 Java 包为 USB 设备驱动程序的书写提供灵活性。原 Linux 内核中的 USB 协议栈可分为三层[9]：HCD、USB-core 及 USB Device Driver，如图 2 所示。

其中 HCD 层是 USB 主机控制器的驱动程序，直接与硬件交互[1]；USB-core 层是协议栈的核心部分，它提供一套完整的 API 给 USB 设备驱动程序调用，并将这些调用传递给 HCD，从而完成与 USB 设备的通信；最上层的 USB 设备驱动是由各个不同的 USB 设备开发商提供的设备驱动，由于其多样性和质量的差别，这部分最有可能导致内核不稳定甚至崩溃。

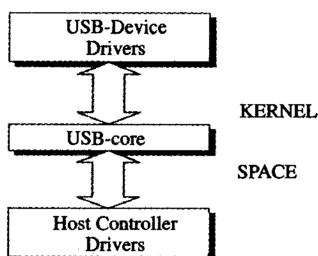


图 2 原内核 USB 协议栈

将原协议栈结构移植到我们的安全驱动架构中，我们得到了如图 3 所示的协议栈。新的结构将原有的三层进行了必要的隔离，以提高其稳定性。我们在 USB-core 与 HCD 之间加入了一层内核管理层，同时将 USB-core 与设备驱动层搬移到用户空间，与内核空间分离。对于易出错的设备驱动程序，我们采用类型安全的 Java 语言进行改写，并且在用户态的 Java 虚拟机中运行。这样可以降低各层错误的相互影响，提高整个系统的稳定性。

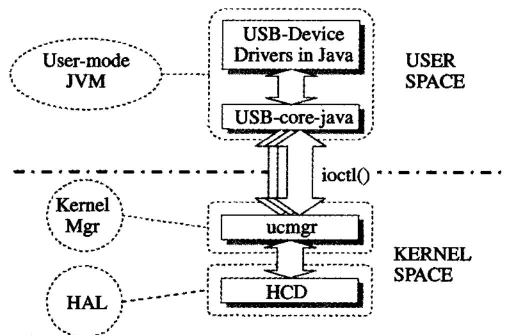


图 3 新 USB 协议栈

### 4.2 设备抽象层

我们将 HCD 驱动层作为系统中的最底层–设备抽象层，HCD 使用的是原 Linux 内核中的 UHCI 驱动模块，未做任何改动，因为我们相信这部分驱动代码是经过严格测试的可信任代码。HCD 用来驱动 PC 机上的 UHCI USB 主机控制器，实现了 USB 1.1 协议。

上层与硬件的交互都是通过 uhci_submit_urb 和 uhci_unlink_urb 接口来实现的。这样的接口抽象可以给上层提供更大的灵活性同时简化内核管理层对传入数据的检查。

### 4.3 内核管理层

内核管理层的实现沿用了许多原 USB 协议栈中的底层函数，其作用是管理 USB 协议栈核心数据结构，处理上层用户态 USB 设备驱动和下层 HCD 的交互。其中用户态通过系统调用向内核管理层发出请求，管理层以阻塞模式处理请求，即管理将请求进一步封装交由 HCD 处理，并同时阻塞用户进程；当 HCD 处理结束发出中断请求时，管理层作初步的中断处理，然后唤醒用户进程，将中断数据交由上层进一步处理。

内核管理层被实现成一个字符型的内核驱动模块，用户态程序通过 ioctl()系统调用与内核管理层交互，对于每一个传入参数，内核管理层都作必要的检查。出于安全性和效率的考虑，上层 USB 设备驱动所申请的 USB 协议栈核心数据结构(如 USB 设备描述符、数据传输请求块等)仍在内核空间，被内核管理层记录在线性链表中，上层只保留一个操作句柄。与硬件的数据通信也处于内核空间中，由内核管理层统一管理。当上层需要数据时，再通过 ioctl()向内核管理层申请数据。

### 4.4 用户态 JVM

用户态的 JVM 是运行主体驱动程序的部分，我们将原协议栈中 USB-core 层的核心代码利用 Java 语言进行改写，并封装成 Java API 类库，提供给 USB 设备驱动程序调用。用户态 JVM 使用 JNI 接口与内核驱动模块进行交互，将 JNI 提供的接口封装成一个 HostIntf 类，提供给驱动程序一些与底层相关的操作函数。我们定义一个面向对象的类结构树，可

维普资讯 http://www.cqvip.com

以看作上层 USB 设备驱动程序的结构框架，实际编写的驱动程序可以通过继承和实现接口的方式简化编程。

USB-core-java

我们在用户态 JVM 中编写了两个核心函数包(package)：usb_first.linux 和 usb_first.core，来取代原 Linux 协议栈中的 USB-core API 函数。并定义了一些 JVM 中使用的数据结构。其中核心类为 usb_first.core.UsbCore，其为 USB 系统的管理类，管理所有的设备驱动程序，并管理 USB 总线上的拓扑结构。大部分 USB Specification 中定义的数值常量也在这里定义。

原协议栈实现了 USB 的全部四种传输方式，在 USB-core-java 中我们实现了 USB 的中断和等时传输而没有实现块传输方式。在原协议栈中，利用这些传输方式的设备驱动程序的编写者们需要维护一个用于 USB 数据封装的 struct urb 结构体，如果在驱动卸载或设备断开时没有适当地销毁这个内核数据结构，那么就有可能造成内核的崩溃，在我们的实现中，这个数据结构不是由 USB 设备驱动的编写者来维护的，而是提供了一些接口函数。由 UsbCore 来管理此类 urb 结构可以有效地避免驱动的编写者在断开函数中遗漏的回收工作，提高系统的稳定性。

USB 设备驱动

我们用 Java 语言重写了 USB 摄像头及 USB 键盘和鼠标驱动，还改写了原来用 C 书写的摄像头图像显示软件，全部使用 Java 语言实现，以此作为应用程序，与提供驱动服务的 JVM 进行通信。

## 5 系统性能测试

我们对系统的中断响应速率、实际的数据流量以及摄像头采集图像的显示帧率进行了测试，并且针对摄像头驱动，与原来利用 C 书写的运行在内核空间的设备驱动进行了对比。

### 5.1 测试环境及方法

| | |
|---|---|
| 硬件平台 | 普通 PC |
| | Celeron-M 600MHz |
| | 512MB RAM |
| 操作系统 | Redhat Linux，内核版本 2.4.20 |
| | Java 虚拟机 |
| | JDK1.5.08 |
| 测试设备 | 罗技公司生产的 USB 键盘及鼠标 |
| | 使用 Vimicro 芯片的 USB 摄像头 |

我们的测试均采用 USB1.1 传输标准，分别测试了 USB 键盘鼠标以及 USB 摄像头在系统中表现的性能。对于中断频发型设备(USB 键盘鼠标)，我们分别记录了内核所接收的中断数以及 JVM 中的设备驱动程序处理的中断数，以此作为比较，考察系统的中断响应性能；对于大量数据传输的设备(USB 摄像头)，我们记录了内核所接收的数据总量、JVM 中设备驱动程序接收的数据总量以及驱动程序实际处理的数据量，以此考察系统的数据吞吐和处理能力。对于摄像头驱动程序，我们还测试了原内核中 C 代码书写的驱动程序的相应数据，与我们的系统进行了对比。

### 5.2 interrupt 传输性能测试

这部分测试使用了支持 interrupt 型传输的 USB 键盘和鼠标，主要测试系统对 interrupt 型传输的响应速率。由于单次中断传输的数据量非常小，因此这部分不进行数据流量的测试。

表 1 USB 键盘及鼠标测试结果

| 测试[1] | 频率 | 总计 | 响应 | 丢失[2] | 响应率 |
|---|---|---|---|---|---|
| 键盘 1 | 30.24 | 3024.4 | 3000 | 23.4 | 99.24% |
| 键盘 2 | 6.82 | 682.2 | 679 | 2.2 | 99.68% |
| 鼠标 1 | 119.66 | 11966.2 | 11949.4 | 15.8 | 99.87% |
| 鼠标 2 | 7.98 | 797.8 | 796.4 | 0.4 | 99.95% |

1)这里的键盘 1 为最大限度地敲击键盘，键盘 2 为正常录入速度；鼠标 1 为鼠标移动情况，鼠标 2 为鼠标点击情况

2)由于在每次测试结束时调用 unlink 操作会固定丢失一次中断请求，所以实际丢失的次数中减掉了一次

我们测试了在约 100s 的时间内系统对常规输入设备的中断响应性能，其测试结果如表 1 所示。可以看出，系统对输入设备的中断(6～120 次/s)，响应率是很高的，可以达到 99%以上。因此，我们认为系统能够满足输入设备的中断响应要求。

### 5.3 isochronous 传输性能测试

这部分对摄像头驱动实时采集数据的性能进行了测试。由于摄像头数据流量大，且中断产生频繁，因此这部分的测试可以比较好地反映出整个系统的性能。为了与原有驱动进行对比，我们同时也测试了原内核中摄像头驱动的性能指标。

表 2 数据吞吐量性能测试

| 测试项 | 平均值(单位为字节) |
|---|---|
| 内核接收数 | 10,074,502 |
| Java 驱动程序接收数 | 9,501,411 |
| 原内核驱动接收数 | 9,844,709 |
| 实际丢失数[1] | 565,480 |
| 数据接收率 | 94.31% |

1)这里有一小部分的字节是在 unlink 操作中丢失的。

对数据吞吐量的测试如表 2 所示。我们测试了约 100s 内整个系统的数据流量，并且与原内核驱动进行了对比。表 2 的最后一列记录了在近似相同的时间内(100s)原内核驱动所接收到的字节数。从这组数据可以看出，我们的系统与原系统相比，数据吞吐量方面的性能损失不大，而对于内核的数据接收能够达到 94%的接收率。而由于 Linux2.4 内核本身的中断性能不够好，所以对于这样的表现应该是可以满足系统的要求的。

表 3 解码及显示性能比较测试

| | 接收帧数 | 解码帧数 | 显示帧数 | 所用时间 | 显示帧率 |
|---|---|---|---|---|---|
| JVM | 2017 | 1121 | 1108 | 100.30 | 11.04 |
| 原内核 | 2025 | 2024 | 1998 | 100.02 | 19.98 |

对 JVM 驱动的解码和显示性能的测试结果如表 3 所示，表中我们列出了在相同的时间内原内核驱动程序的性能。可以看出，我们系统的瓶颈在于解码性能比较低，与原内核驱动相比，降低了约 50%。这样的性能降低表明系统的数据处理能力不强，这是由 Java 虚拟机本身的运行效率不如 CPU 本地机器码造成的。

**结束语** 本文提出了新的设备驱动程序架构，目的在于将设备驱动程序可能产生的错误从操作系统内核中分离出来，从而提高整个操作系统的稳定性。我们结合了第 2 节中介绍的几项研究成果，设计了如图 1 所示的系统。

维普资讯 http://www.cqvip.com

我们利用用户态的JVM对驱动程序进行隔离，这样充分保护了整个系统内核的安全。严格定义了Java虚拟机和外部的接口，并且利用用户态进程的虚拟机将驱动程序对外界的影响降到最小。出于性能和现有操作系统的中断处理机制，我们将与硬件直接相关的部分留在了内核中，并在其上增加了一个内核管理模块，用于和上层JVM进行通信。这样的结构可以监控每个进入内核的请求，严格控制了上层驱动对内核的操作行为，起到了隔离的作用。

将原Linux内核中的USB协议栈移植到了图1所示系统中。系统性能测试表明，我们的系统可以满足一般USB设备的中断响应要求，数据吞吐量可以满足摄像头设备的要求。但与原本地代码驱动相比，数据处理能力下降了约50%。相信这些性能的降低随着计算机硬件性能的不断提高可以得到改善，而新的系统架构将会保证操作系统在一个更为稳定的状态运行。

## 参考文献


1 毛德操，胡希明. LINUX内核源代码情景分析. 杭州：浙江大学出版社，2001
2 Chou Andy, Yang Junfeng, Chelf B, et al. An empirical study of operating systems errors. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems principles, Oct. 2001
3 Härtig H, Hohmuth M, Liedtke J, et al. The performance of microkernel based systems. In: Proceedings of 16th ACM Symposium Operating System Principles, 1997. 66～77
4 Herder J N, Bos H, Gras B, et al. Construction of a highly dependable operating system. In: Dependable Computing Conference, 2006. EDCC '06. Sixth European, Oct, 2006. 3～12
5 Herder J N, Bos H, Gras B, et al. Tanenbaum. The architecture of a reliable operating system. In: 12th ASCI conference, Jun, 2006
6 Herder J N, Bos H, Gras B, et al. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. ACM SIGOPS Operating Systems Review, Jul. 2006(3)
7 Hunt G, Larus J R, Abadi M, et al. An overview of the singularity project: [Technical report]. Microsoft Research, Oct 2005
8 Ostrand T J, Weyuker E J. The distribution of faults in a large industrial software system. ACM SIGSOFT Software Engineering Notes, Jul. 2002(4)
9 Rubini A, Corbet J. Linux Device Drivers. 2nd edition. O'Reilly, Jun 2001
10 Swift M M, Bershad B N, Levy H M. Improving the reliability of commodity operating systems. ACM Transactions on Computer Systems (TOCS), Feb. 2005(1)
11 Swift M M, Martin S, Levy H M, et al. Nooks: an architecture for reliable device drivers. In: EW10: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC, New York, NY, USA: ACM Press, 2002. 102～107
12 Tanenbaum A S, Woodhull A S. Operating Systems Design and Implementation. 3rd edition. Prentice Hall, Jan. 2006
13 Tanenbaum A S, Herder J N, Bos H. Can we make operating systems reliable and secure? IEEE Computer, May 2006, 39(5): 44～51


---

（上接第267页）

试覆盖准则。

• 假设活动图中的每个循环都执行一次，没有讨论嵌套循环等更为复杂的结构。

### 4.4 关于系统的形式方法

在文[7]中提出的系统化方法，是对自适应细菌Agent的一种改进：1)在处理SFJ时不需要建立一个堆栈，而是通过为每个结点定义一个迁移"候选集"和"迁移输出集"，再对候选集的每个迁移分配操作优先级来实现。2)不需要分裂NFJ和BNFJ成SFJ，而是重新定义了算法自动完成测试场景的生成。3)能够自动处理活动图中的混合分叉-汇合。4)在工具的实现上，定义了一个描述UML活动图的XML Schema，从UML建模工具(或其它插件)生成的XMI文件中提取表示UML活动图的XML文档。工具读入XML文件，然后直接生成测试场景，整个生成过程能够自动完成。但是这种方法也不能适用从更一般的UML活动图产生测试场景\用例，也同样没有包括UML的全部建模元素，如中断、拴(Pin)等，尚需进行进一步的研究。几种方法的算法效率和适用范围见表1。

表1 算法效率和适用范围

| 方法 | 算法效率 | 适用范围 |
|---|---|---|
| 反蚂蚁 Agent | 慢 | 较小 |
| 自适应细菌 Agent | 较慢 | 较大 |
| 灰盒方法 | 较快 | 小 |
| 系统形式方法 | 快 | 大 |

**进一步的工作** 本文介绍了从UML活动图中生成测试场景的方法，总结了几种从UML活动图生成测试场景\用例的方法及其使用的算法，即反蚂蚁Agent方法、灰盒方法、自适应细菌Agent方法和系统的形式化方法。对这些方法进行了分析与比较，指出一些不足之处。

由于UML活动图语义的不精确性，各种活动状态之间的存在多种组合，且没有严格的规则，基于UML活动图的测试需要进一步的研究：

• 要对UML活动图各种活动状态之间的多种组合进行进一步的总结分类，如上述方法中对分叉-汇合类型的划分也不是充分的，需要进一步研究。

• 要考虑到UML活动图中的其他元素，特别是UML 2.0对活动图新增的元素，如流终结状态、发送和接受信号、信息栓(Pin)、中央缓存结点[8]等。

• 要考虑到UML活动图的其他逻辑结构，如异常处理机制、可中断区域、扩展区域和多维分区[8]等。

• 针对UML活动图的新的元素和新的结构，需要扩展测试覆盖准则，以提高测试的充分性。

• 现有的UML活动图的测试覆盖准则尚需进行评估。

## 参考文献


1 Li H, Lam C P. Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams. In: Proc. TESTCOM 2005, LNCS 3502, Montreal, 2005
2 Wang L, Yuan J, Yu X, et al. Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In: Proc. APSEC' 04, 2004
3 Xu Dong, Li H, Lam C P. Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams. In: Proc. of 12th APSEC 2005, IEEE Computer Society Press, Taipei, 2005
4 Bai X, Lam C P, Li H. An Approach to generate the Thin-threads from the UML Diagrams. In: Proc. COMPSAC 2004, Hong Kong, 2004
5 张榍，刘超，孙昌爱. 基于UML活动图模型的测试用例生成技术研究. 北京航空航天大学学报，2001，27(4)
6 袁洁松，王林章，李宣东，等. UMLTGF：一个基于灰盒方法从UML活动图生成测试用例的工具. 计算机研究与发展，2006(1)
7 Xu Dong, Lam C P, Li H Z. A Systematic Approach to Automatically Generate Test Scenarios from UML Activity Diagrams. In: The Proceeding of the Third IASTED International Conference on Advances in Computer Science and Technology, April 2-4, 2007, Phuket, Thailand, 2007
8 Object Management Group. Unified Modeling Language: Superstructure. version 2.0 July 2005. Available at www.omg.org