基于 USB 设备的用户态驱动框架①

叶小龙, 周学海, 陈 超

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(中国科学技术大学 苏州研究院, 苏州 215123)

摘 要: 传统的 linux 系统为了获得更好的性能, 将设备驱动运行在内核空间, 不可避免的降低了系统的可靠性和稳定性. 基于常用的USB设备提出了一种全新的驱动架构, 它将驱动以进程的形式运行在用户空间, 并且支持热插拔, 驱动管理等特性, 实验结果表明这种架构性能良好, 能够满足实际应用需求.

关键词: 驱动架构; 用户空间; USB 设备; 热插拔

User-Space Device Driver Framework Based on USB Device

YE Xiao-Long, ZHOU Xue-Hai, CHEN Chao

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
(Suzhou Institute for Advanced Study, USTC, Suzhou 215123, China)

Abstract: In order to achieve high performance, traditional linux operating system run device drivers in kernel space. Unfortunately, this architecture inevitably decreases reliability and stability of the system. Based on commonly used equipment USB devices, this paper presents a new driver architecture which run device drivers as unprivileged user-level code, encapsulated into a process, this architecture also support features such as hotplug, driver management and so on. Experiments on several usb devices show it works well in user space without significant performance degradation.

Key words: driver framework; user space; USB device; hotplug

随着信息产业的快速发展, linux 系统部署的范围越来越广泛, 尤其在航天, 军工, 医疗, 卫生, 通信, 控制等领域. 性能和可靠性是大多数系统的关键参数, 随着硬件技术的快速发展, 性能的问题不再是一个瓶颈, 然而系统的可靠性并没有随着硬件技术的发展而提高. 随着业务的发展, linux 系统对生命期的要求越来越高, 甚至是不间断的服务, 这使得系统的稳定性和可靠性成为重中之重.

在 Linux 操作系统中, 一方面基于性能和方便性等要求, 另一方面由于I/O指令是特权级指令, 因此设备驱动往往与操作系统内核运作在同一地址空间—进程的内核态地址空间, 具有内核的所有权限, 例如可以直接访问所有的物理内存, 任意调用内核其他模块等等[1,8,12]. 在 Linux 系统中, 驱动包括 Linux 自带的驱动以及一些硬件厂商自己开发的大量驱动. 从可靠性的角度看, 这些驱动都没有经过完整的可靠性验证, 特别是硬件厂商自己开发或移植的驱动, 而且完整的可靠性验证在实践中是非常困难的. 因此当这些驱动出现问题时, 比如发生非法访问内存等, 将可能导致整个 Linux 系统的崩溃. 研究表明, 85%的 windows xp 的崩溃是驱动导致的, linux 下驱动导致崩溃的可能性是其他内核代码的 3-7 倍[1,5].

基于目前广泛使用的 USB 设备, 本文提出了一种全新的用户空间驱动框架 UMDUSB, 它将 USB 设备驱动实现为用户态的进程, 它的优势如下:

① 提高了系统的稳定性, 一旦驱动出错, 也只是引起驱动进程的崩溃, 不会影响整个系统;

② 降低了驱动开发的门槛, 可以使用开发库来

① 基金项目:国家自然科学基金(60873221);江苏省产学研前瞻性联合研究项目(BY2009128)
收稿时间:2011-12-29;收到修改稿时间:2012-02-21

提高开发效率，并能使用 GDB 等常用工具进行调试;

③ 提供了用户空间的驱动管理功能，支持用户态热插拔.

1 UMDUSB体系结构概述

传统的 USB 设备驱动代码内嵌在内核中或者作为内核模块加载，使用/sbin/hotplug 实现热插拔，在 linux 设备模型中统一管理[9]. UMDUSB 则分为三个模块，用户态 USB 通信模块(libusb 库)，用户态 USB 设备热插拔监控模块(HOTPLUG_MONITOR)，用户态 USB 设备管理模块(UMD_MANAGER). 通信模块主要提供了用户态访问和控制 USB 设备的接口，它对 usbfs 提供的函数接口和数据结构进行了封装，在功能和原理上类似于 USB CORE，但是更加层次化和结构化，在用户态就能方便的和 USB 设备进行通信，提高驱动开发的效率. HOTPLUG_MONITOR 模块实时监控热插拔事件，发现设备插入或者移除时及时采取相应的措施处理，使得硬件能够正常工作，整个系统不受影响. UMD_MANAGER 模块管理 USB 设备与驱动，能够实时显示当前系统中设备插入或者移除的情况，查看设备的状态，加载或者卸载驱动，并实现了日志系统，能够记录热插拔事件的产生. 整体框架如图 1 所示.

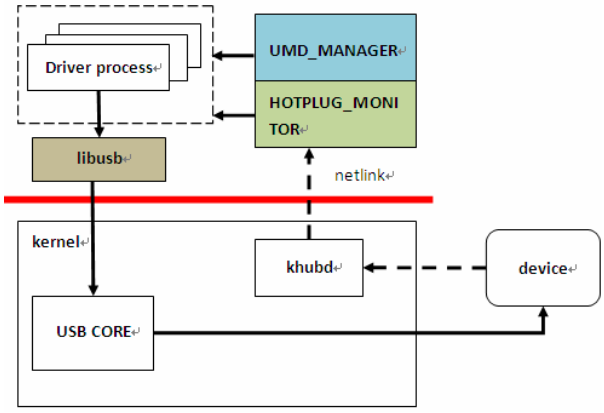

图 1　UMDUSB 系统总体框图

2 各模块的具体设计与实现

2.1 用户态通信模块

2.1.1 libusb 简介

我们使用 libusb 来实现用户空间对 USB 设备的控制访问. Libusb 是一个针对 USB 设备的用户空间通信接口库[6,14], linux 最初只能在内核空间访问 USB 设备，但后来产生了在用户空间访问 USB 设备的需求，因此 linux 开发人员在内核中添加了 usbdevfs 文件系统. USB file system 提供了一些在用户空间下操作 USB 设备的函数接口和数据结构，在开发用户态驱动时，可以直接利用这些函数接口来实现对 USB 设备的控制和数据传输. 但是这些接口比较复杂和底层，对开发人员的要求较高，使用过程中容易出错，不利于提高开发效率. 基于这方面的考虑, libusb 对 usb file system 提供的函数接口和数据结构进行了封装，可以有效减少程序中函数和数据结构使用不当造成的错误.

Libusb 对 USB 设备的访问提供了两种机制[6,14]，同步访问和异步访问. 同步访问方式编程模型简单，函数阻塞直到数据传输结束，效率不高. 异步访问采用 poll 机制实现了非阻塞的读写，提高了程序的灵活性和执行效率. 异步读写某种程序上是一种多线程的机制，大大提高了 USB 数据传输的吞吐量.

2.1.2 用户态驱动的编写

驱动开发的主要工作就设备的管理控制和数据的通信传输处理，使用 libusb 编写的用户态驱动可以方便的胜任这些工作. 利用 libusb 编写的用户态驱动的通用流程如下:

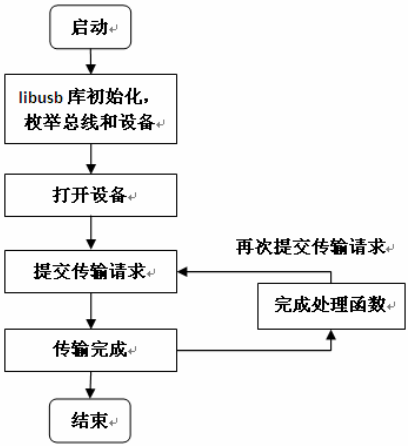

图 2　libusb 用户态驱动通用流程

驱动程序的核心在数据通信和处理，在 libusb 的框架中，数据传输完毕后会调用用户注册的完成处理函数，在完成处理函数里对数据进行处理，所以我们使用 libusb 开发一款 USB 设备的用户态驱动，首先需要明确设备的基本功能，数据格式的要求，然后编写完成处理函数对数据进行解析处理，必要时还需要与内核进行交互，比如 USB 鼠标，USB 键盘等交互设备

的驱动还需要和内核中的 input 子系统结合才能完成工作. 与内核进行交互的方法很多, 可以通过 ioctl 系统调用, netlink 机制等等.

2.2 用户态热插拔监控模块

监控模块主要完成热插拔事件的实时监控, 在用户态捕获相应的事件, 解析出相应的信息, 如果发现是设备插入则加载相应的驱动, 如果是设备移除则卸载相应的驱动.

热插拔(hot-plugging)事件是指在系统插入或者移除设备是导致系统配置发生变化, 我们需要在用户空间截获这些事件进行相应处理. 热插拔机制需要硬件和操作系统的支持[9].

2.2.1 硬件支持

热插拔事件的产生通常由总线驱动级别上的逻辑所控制[4], 主机集线器 HUB 监视着每个端口的信号电压, 当有新设备接入时就能察觉. 集线器端口的两根信号线中每一根都有 15K 欧的下拉电阻, 而每一个设备在 D+上都有一个 1.5K 欧的上拉电阻, 当用 USB 将 PC 和设备接通后, 设备的上拉电阻使得信号线电位升高, 因此被主机集线器检测到, 之后由集线器 HUB 的驱动进行响应.

2.2.2 内核支持

在 linux 内核初始化的时候[11], HUB 驱动创建了内核线程 khubd, 该线程用来管理监视 HUB 的状态, 一旦发生热插拔就会唤醒这个线程, 进而添加或者移除设备.

Khubd 线程检测到热插拔事件后, 通过创建或者删除 kobject 对象, 最终调用 kobject_uevent 函数将发送 netlink 消息传递给用户空间, 这些工作不需要驱动处理, 在 linux 的统一设备模型的子系统层, 已经做好了这部分代码的处理, 包括在设备对应的 kobject 创建和移除时都会发送相应的 add 和 remove 消息, 前提是需要在内核配置中添加对 hotplug 和 netlink 的支持. Netlink 作为一种内核和用户空间的通信方式, 不仅仅用在 hotplug 机制中, 同样还应用于其他很多网络相关的内核子系统中[13].

2.2.3 用户空间处理热插拔事件

用户态热插拔监控框架主要是一个基于 netlink 的消息传递处理框架. Netlink 是 linux 中一种在内核空间与用户空间进行双向数据传输的通信方式, linux2.4 版本之后的内核中, 几乎所有的中断过程和用户态进程通信都是使用 netlink 实现的[13]. 与传统的 socket 通信不同, netlink 使用进程 id 作为通信的地址, 如果通信的对象是内核, 则 PID 设置为 0.

用户态热插拔监控框架中有两个 netlink 对象, 分别位于内核和热插拔监控线程当中, 其中内核中的 netlink 对象由 kobject_uevent 负责, 当硬件发生热插拔会导致设备对应的 kobject 对象的创建和删除, 每一次 kobject 对象的改变都会使得内核广播一个 netlink 消息, 消息内容中包括了热插拔事件的类型, 设备的类别, 设备的路径等等信息. 关键代码如下:

```
/* 创建一个 netlink socket, NETLINK_KOBJECT_UEVENT 表示内核事件向用户态通知*/
uevent_sock=netlink_kernel_create(net, NETLINK_KOBJECT_UEVENT, 1, NULL, NULL, THIS_ MODULE);
……
/* 广播事件消息 */
retval = netlink_broadcast_filtered(uevent_sock, skb, 0, 1, GFP_KERNEL,kobj_bcast_filter, kobj);
```

Netlink 接收对象由用户态监控线程所创建, 它负责接收内核广播的热插拔事件, 解析相应信息并做处理. 关键代码如下:

```
/*创建用户态 netlink socket, SOCK_DGRAM 表示数据报格式, NETLINK_KOBJECT_UEVENT 表示内核事件的协议类型, 与内核态创建 socket 时的协议相对应*/
s=socket (PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);

/*设置接收报文的长度*/
setsockopt(s, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz));
/* 将 sock 与本进程地址(进程 ID)绑定 */
 bind(s, (struct sockaddr *) &addr, sizeof(addr));
/*接收内核广播的消息*/
buflen = recv(ctx_bus->monitor_fd, buf, UEVENT_MSG_LEN, 0);
…//解析消息处理
```

Netlink 相应于传统 socket 的优势在于, 它不存在客户端和服务器端的概念, 通信的双方可以被动的接收信息, 也可以主动的发起通信. 因此框架在运行时, 监控线程既可以接收到内核广播的热插拔事件消息,

也可以主动的发送消息通知内核处理.

2.3 用户态 USB 设备驱动管理模块

用户态 USB 设备热插拔管理模块借鉴参考了 linux 内核中的统一设备模型, 引入了总线、设备、驱动的概念.

在 linux2.6 的设备模型中, 最重要的就是总线、设备、驱动这三个实体, 总线将设备和驱动绑定, 在系统中每注册一个设备的时候, 会寻找与之相匹配的驱动, 相反在系统中每注册一个驱动的时候, 寻找与之匹配的设备, 匹配由总线来完成[4].

一个现实的 linux 设备和驱动通常需要挂接在实际的总线上, 常见的物理总线有 PCI, I2C, SPI 等等, 但有些设备相对独立, 并不依附于任何物理总线, 基于这一背景, linux 提出了一种虚拟总线, 称为 platform 总线. 我们借鉴了这一概念, 设计了用户态热插拔总线 hotplug-bus, 它上面挂接了两条链表, 设备链表和驱动链表. 所有的热插拔发现的设备都通过 hotplug-bus 相连, 每一个设备都用 hotplug_device 结构的一个实例来表示, 而这里的驱动则是相应的用户态驱动进程, 每一个驱动用 user_mode_driver 结构的一个实例来表示. 在热插拔监控进程最开始初始化的时候, 它根据事先制定的规则, 从系统配置脚本中读取当前系统中所有的用户态驱动的信息, 并将它们链入总线中.

用户态热插拔总线, 热插拔设备, 用户态驱动的关系结构图如下:

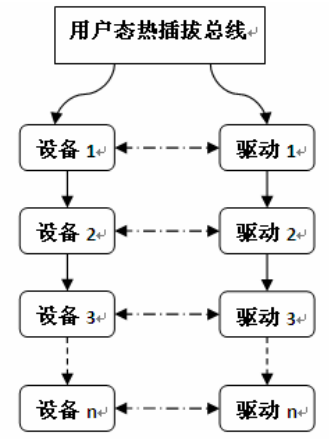

图 3　用户态设备驱动框架图

3　测试结果与评价

在上述框架的基础上, 我们实现了 linux 下常用的 USB 鼠标和键盘驱动, 并对其进行了相关测试, 以验证我们的设计模型. 测试平台为:

Cpu: intel CORE i3, 2.3GHZ

Cache size: 4094KB

Ram: DDR2, 2GB

OS: ubuntu10.04 + linux 2.6.35-x86_64

测试主要分为功能性和性能两方面进行, 功能性测试指用户态驱动是否工作正常, 框架是否支持驱动管理功能, 是否支持热插拔, 是否具有容错功能. 测试结果如表 1 所示.

表 1　功能性测试结果

测试用例	测试结果
用户态驱动功能测试	鼠标键盘正常响应工作
用户态驱动管理功能测试	可实时查询系统当前用了哪些用户态驱动; 可实时加载或者卸载驱动
热插拔支持测试	鼠标键盘插入能够自动识别, 并加载驱动 反复进行插入拔出操作, 驱动工作正常
容错测试	驱动发生异常(如非法访问内存, 数组越界等)不会导致系统崩溃

驱动由原来的内核态改为用户态后对系统的性能有多大程度的影响是最为关键的问题, 因此我们测量了用户态 usb 驱动的响应时间, 并于原来的内核态驱动进行对比, 以验证用户态框架的可用性.

测量用户态 usb 驱动程序所花费的时间, 也就是每一次 USB 数据传输所花费的时间, 整个过程分为两个时间段, USB 数据的传输时间 T1 和 USB 数据的处理时间 T2, T2 也就是完成处理函数执行的时间, 这个时间根据实际设备应用的不同而不同, 我们主要关注 USB 数据传输的时间 T1, 测试方法为在驱动提出传输请求时记录下系统时间, 接收到数据后再次读取系统时间, 两者相减得到函数的实际运行时间. 我们分别在内核的 usb 驱动程序和我们的用户态驱动进行测试, 测试时间数据如下:

由图 4 可以看出, usb 用户态驱动在数据传输方面的速度甚至是优于内核态驱动的, 在实际使用中, 由于在完成处理函数中需要进入内核与其他内核模块进行交互, 会损失一些性能, 这一块可以根据应用需求

进行优化, 使得整体的性能损失在可接受的范围之内.

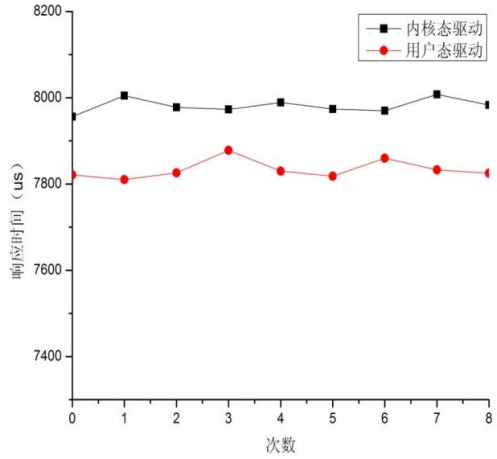

图 4 响应时间对比图

4 结论与展望

用户态驱动作为一种提高操作系统可靠性的有效方法, 越来越受到研究人员的重视[1,8,12]. 本文基于常用的 USB 设备, 提出了一直全新的驱动框架 UMD-USB, 能够将 USB 设备驱动的大部分代码以进程的形式运行在用户态, 实现了驱动与内核的隔离, 以有效提高系统的可靠性. 同时, UMD-USB 还提供了对驱动管理和热插拔等特性的支持, 实验表明, 大部分的 USB 设备驱动能用该架构改写到用户空间, 并且性能满足实际应用的需求.

该架构的不足之处在于用户态驱动的模块化程度不够, 与上层应用结合太紧密, 并且上层应用独占设备, 不支持对设备的并发访问, 下一步的工作重点是尽量降低应用与驱动的耦合性, 并且通过信号量, 锁等同步机制实现设备的并发访问.

参考文献

1 Swift M, Bershad B, Levy H. Recovering device drivers, Apr. 6 2006, uS Patent App. 11/398,799.

2 Yamauchi H, Wolczko M. Writing solaris device drivers in java, in PLOS'06: Proc. of the 3rd Workshop on Programming Languages and Operating Systems. New York, NY, USA: ACM, 2006. 3.

3 Hunt G, Larus J, Abadi M, Aiken M, Barham P, Fahndrich M, Hawblitzel C, Hodson O, Levi S, Murphy N, et al. An overview of the Singularity project. Microsoft research msr-tr-2005-135, Microsoft Corporation, Redmond, Washington, 2005.

4 魏永明,耿岳. Linux 设备驱动程序.第 3 版.北京:中国电力出版社,2010.

5 Swift M, Bershad B, Levy H. Improving the reliability of commodity operating systems. ACM Trans. on Computer Systems (TOCS), 2005,23(1):110.

6 Libusb. http://libusb.sourceforge.net/.

7 Härtig H, Hohmuth M, Liedtke J, Schönberg S, Wolter J, The performance of microkernel-based systems. Proc. of the 16th ACM Symposium on Operating System Principles. 5–8.

8 Shen YT, Elphinstone K, Heiser G. User-Level Device Drivers: Achieved Performance.

9 http://kernel.org/doc/ols/2001/hotplug.pdf

10 Herder J, Bos H, Gras B, Homburg P, Tanenbaum A. MINIX 3:A highly reliable, self-repairing operating system. ACM SIGOPS Operating Systems Review, 2006,40(3):89.

11 Linux kernel 2.6.35: http://www.kernel.org/pub/linux/kernel/v2.6/

12 Ganapathy V, Renzelmann M, Balakrishnan A, Swift M, Jha S. The design and implementation of microdrivers. Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2008.168–178.

13 董昱,马鑫.基于 netlink 机制内核空间与用户空间通信的分析.测控技术,2007,26(9):57–58,60.

14 徐家.Linux 下 USB 视频设备用户空间驱动研究与开发. 2010,28–29.