International Journal of Information and Education Technology, Vol. 2, No. 5, October 2012

Device Drivers in User Space: A Case for Network Device Driver

Hemant Agrawal and Ravi Malhotra, *Member, IACSIT*

***Abstract*—Traditionally device drivers specially the network one's are implemented and used in Linux Kernel for various reasons. However in recent trend, many network stack vendors are moving towards the user space based drivers. Open Source–'GPL' is one of strong reason for such a move. In the absence of generic guidelines, there are various options to implement device drivers in user space. Each has their advantage and disadvantage. In this paper, we will cover multiple issues with user space device driver and will give more insight about the Network Device Driver implementation in User Space**

***Index Terms*—Network drivers, user space, zero copy.**

I. Introduction

However, in recent times, there has been a shift towards running data path applications in the user space context. Linux user space provides several advantages for applications with respect to a more robust and flexible process management, standardized system call interface, simpler resource management, availability of a large number of libraries for XML, regular expression parsing etc. It also makes applications easier to debug by providing memory isolation and independent restart. At the same time, while kernel space applications need to confirm to GPL guidelines, user space applications are not bound by such restrictions.

User space data path processing comes with its own overheads. Since the network drivers run in kernel context and use kernel space memory for packet storage, there is an overhead of copying the packet data from user-space to kernel space memory and vice-versa. Also, user/kernel-mode transitions usually impose a considerable performance overhead, thereby violates the low latency and high throughput requirements of data path applications.

In the rest of this paper, we shall explore an alternative approach to reduce these overheads for user space data path applications.

II. Mapping Memory to User-Space

As an alternative to the traditional I/O model, the Linux kernel provides a user-space application with means to directly map the memory available to kernel to a user space address range. In the context of device drivers, this can provide user space applications direct access to the device memory which includes register configuration and I/O descriptors. All accesses by the application to the assigned address range ends up directly accessing the device memory.

There are several Linux system calls which allow this kind of memory mapping, the simplest being the mmap() call. The mmap() call allows the user application to map a physical device address range one page at a time or a contiguous range of physical memory in multiples of page size.

Other Linux system calls for mapping memory include splice()/vmsplice() which allows an arbitrary kernel buffer to be read or written to from user space, while tee() allows a copy between 2 kernel space buffers without access from user space[1].

The task of mapping between the physical memories to the user space memory is typically done using Translation Look-aside Buffers or TLB. The number of TLB entries in a given processor is typically limited and as such they are used as a cache by Linux kernel. The size of the memory region mapped by each entry is typically restricted to the minimum page size supported by the processor, which is 4k bytes.

Linux maps the kernel memory using a small set of TLB entries which are fixed during initialization time. For user space applications however, the number of TLB entries are limited and each TLB miss can result in a performance hit. To avoid such penalties, Linux provides concept of a Huge-TLB, which allows user space applications to map pages larger than the default minimum page size of 4k bytes. This mapping can be used not only for application data but text segment as well.

Several efficient mechanisms have been developed in Linux to support zero copy mechanisms between user space and kernel space based on memory mapping and other techniques [2]-[4]. These can be used by the data path applications while continuing the leverage the existing kernel space network driver implementation. However they still consume the precious CPU cycles and per packet processing cost still remain moderately higher. Having a direct access to the hardware from the user space can eliminates the need for any mechanisms to transfer packets back and forth between user space and kernel space, and thus it can reduce the per packet processing cost to a minimum.

III. UIO Drivers

Linux provides a standard UIO framework [4] for developing user space based device drivers. The UIO framework defines a small kernel space component which performs 2 key tasks:

- Indicate device memory regions to user space.

Manuscript received January 6, 2012; revised June 18, 2012. This work was supported by the Freescale Semiconductor, India.

The authors are with the Freescale Semiconductor, Plot-18, Sector 16A, Noida, UP-201301, INDIA (e-mail: hemant@freescale.com, ravi.malhotra@freescale.com).

- Register for device interrupts and provide interrupt indication to user space.

The kernel space UIO component then exposes the device via a set of sysfs entries like /dev/uioXX. The user space component searches for these entries, reads the device address ranges and maps them to user space memory. The user space component can perform all device management tasks including I/O from the device. For interrupts however, it needs to perform a blocking read() on the device entry, which results in the kernel component putting the user space application to sleep and wakes it up once an interrupt is received.

IV. User Space Network Drivers

The memory required by a network device driver can be of three types

- Configuration space: this refers to the common configuration registers of the device.
- I/O descriptor space: this refers to the descriptors used by the device to access data from the device.
- I/O data space: this refers to the actual I/O data accessed from the device.

Taking the case of a typical Ethernet device, the above can refer to the common device configuration (including MAC configuration), buffer-descriptor rings, and packet data buffers.

In case of kernel space network drivers, all 3 regions are mapped to kernel space, and any access to these from the user-space is typically abstracted out via either ioctl() calls or read()/write() calls, from where a copy of the data is provided to the user space application.

User space network drivers on the other hand, map all 3 regions directly to user space memory. This allows the user space application to directly drive the buffer descriptor rings from user space. Data buffers can be managed and accessed directly by the application without overhead of a copy.

Taking the specific example of an implementation of a user space network driver for eTSEC Ethernet controller on a Freescale QorIQ P1020 platform, the configuration space is a single region of 4k size, which is page boundary aligned. This contains all the device specific registers including controller settings, MAC settings, interrupts etc. Besides this, the MDIO region also needs to be mapped to allow configuration of the Ethernet Phy devices. The eTSEC provides for up to 8 different individual buffer descriptor rings, each of which are mapped onto a separate memory region, to allow for simultaneous access by multiple applications. The data buffers referenced by the descriptor rings are allocated from a single contagious memory block, which is allocated and mapped to user space during initialization time.

V. Constraints of User Space Drivers

Direct access to network devices brings its own set of complications for user space applications, which were hidden by several layers of kernel stack and system calls.

- Sharing a single network device across multiple applications.
- Blocking access to network data.
- Lack of network stack services like TCP/IP.
- Memory management for packet buffers.
- Resource management across application restarts.
- Lack of a standardized driver interface for applications.

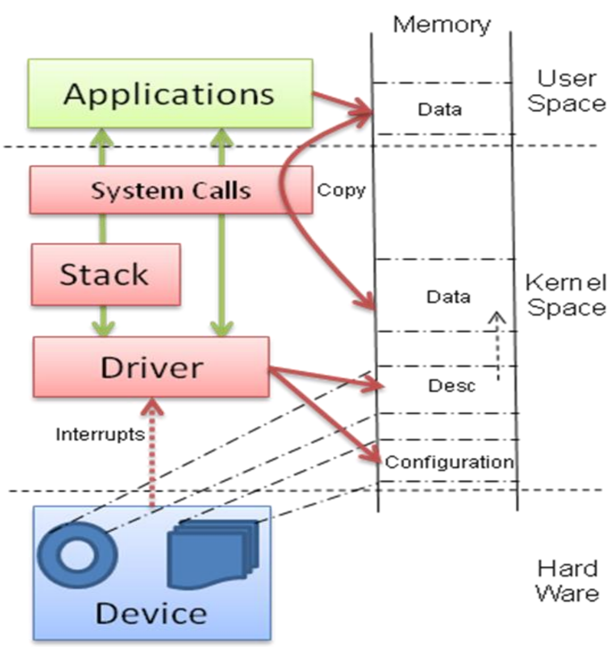

Fig. 1. Kernel space network driver

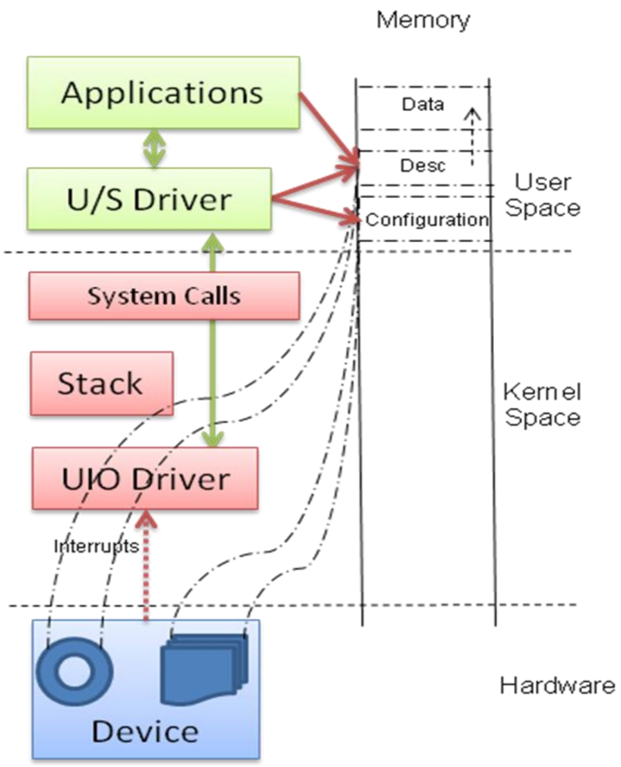

Fig. 2. User space network driver

A. *Sharing Devices*

Unlike the Linux socket layer which allows multiple applications to open sockets – TCP, UDP or raw IP, the user space network drivers allow only a single application to access the data from an interface. However, most network interfaces nowadays provide multiple buffer descriptor rings in both receive and transmit direction. Further, these interfaces also provide some kind of hardware classification mechanism to divert incoming traffic to these multiple rings. Such a mechanism can be used to map individual buffer descriptor rings to different applications. This again limits the number of applications on a single interface to the number of rings supported by the hardware device. An alternate to this is to develop a dispatcher framework over the user space driver, which will deal with multiple applications.

B. *Blocking Access to Data*

Unlike traditional socket based access which allows user space applications to block until data was available on the socket, or to do a select()/poll() to wait on multiple inputs, the user space application has to constantly poll the buffer descriptor ring for an indication for incoming data. This can be addressed by the use of a blocking read() call on the UIO device entry, which would allow the user space application to block on receive interrupts from the Ethernet device. This also provides the application with the freedom of when it wants to be notified of interrupts – i.e. instead of being interrupted for each packet, it can choose to implement a polling mechanism to consume a certain number of buffer descriptor entries before returning to other processing tasks. When all buffer descriptor entries are consumed, the application can again perform a read() to block until further data arrives.

C. *Lack of Network Stack Services*

The Linux network stack and socket interface also abstract basic networking services from applications like route lookup, ARP etc. In the absence of such services, the application has to either runs its own equivalent of a network stack or maintain a local copy of the routing and neighbor databases in the kernel.

D. *Memory Management for Buffers*

The user space application also needs to deal with the buffers provided to the network device for storing & retrieving data. Besides allocation and freeing of these buffers, it also needs to perform the translation of the user space virtual address to the physical address before providing them to the device. Doing this translation for each buffer at runtime can be very costly. Also, since the number of TLBs in the processor may be limited, performance may be hit. The alternative is to use Huge-TLB to allocate a single large chunk of memory, and carve out the data buffers out of this memory chunk.

E. *Application Restart*

The application is responsible for allocating and managing device resources and current state of the device. In case the application crashes or is restarted without being given control to perform cleanup, the device may be left in an inconsistent state. One way to resolve this could be to use the kernel space UIO component to keep track of application process state and on restart, to reset the device and reset any memory mappings created by the application.

F. *Standardized User Interface*

The current generation of user space network drivers provide a set of low level API which are often very specific to the device implementation, rather than confirm to standard system call API like open()/close(), read()/write() or send()/receive(). This implies that the application needs to be ported to use each specific network device.

VI. Conclusion

While the UIO framework provides user space applications with the freedom of having direct access to network devices, it brings its own share of limitations in terms of sharing across applications, resource and memory management. The current generation of user space network drivers works well in a constrained use case environment of a single application tightly coupled to a network device. However, further work on such drivers must take into account addressing some of these limitations.

References

[1] M. Welsh *et al*., “Memory Management For User-Level Network Interfaces,” *IEEE Micro*, pp. 77-82, Mar.-Apr. 1998.
[2] D. Stancevic, “Zero Copy I: User-Mode Perspective,” *Linux Journal*, pp. 105, Jan. 2003.
[3] N. M. Thadani *et al.*, “ An Efficient Zero-Copy I/O Framework for UNIX,” *Sun Microsystem Inc*, May 1995.
[4] H. Koch, The Userspace I/O HOWTO, [Online]. Available: http://www.kernel.org/doc/htmldocs/uio-howto.html

Hemant Agrawal is a software architect with Freescale Semiconductor, Noida, India. He has over 14 years of industry experience in design and development of networking & telecommunication systems and applications. His Primary focus & expertise is in Networking acceleration software, IPSEC, Voice Over IP, SIP, H.323, SS7, ISDN Q.931, MGCP, MEGACO protocols. He was part of IETF ́s SIP-H.323 interworking standard development team. Hemant holds a B.Tech. in Electrical Engineering from Institute of Technology, Banaras Hindu University (IT-BHU), Varanasi, INDIA.

Ravi Malhotra is a software architect with Freescale Semiconductor, Noida, India. He has over 12 years of industry experience in design and development of networking & embedded systems and applications. His Primary focus & expertise is in Networking acceleration software, IPSEC, Routing protocols and QoS. Ravi holds a B.Tech. in Electrical Engineering from Institute of Technology, Banaras Hindu University (IT-BHU), Varanasi, INDIA.