User-Space Device Drivers in Linux: A First Look

Mats Liljegren

Senior Software Architect

Device drivers in Linux are traditionally run in kernel space, but can also be run in user space. This paper will take a look at running drivers in user space, trying to answer the questions in what degree the driver can run in user space and what can be gained from this?

In the '90s, user-space drivers in Linux were much about how to make graphics run faster [1] by avoiding calling the kernel. These drivers where commonly used by the X-windows server.

User-space driver has become ever more important, as a blog post by Tedd Hoff [2] illustrates. In his case the kernel is seen as the problem when trying to achieve high server connection capacity.

Network interface hardware companies like Intel, Texas Instruments and Freescale have picked up on this and are now providing software solutions for user-space drivers supporting their hardware.

Problems with kernel-space drivers

Device drivers normally run in kernel space, since handling interrupts and mapping hardware resources require privileges that only the kernel space is allowed to have. However, it is not without drawbacks.

System call overhead

Each call to the kernel must perform a switch from user mode to supervisor mode, and then back again. This takes time, which can become a performance bottleneck if the calls are frequent. Furthermore, the overhead is very much non-predictable, which has a negative performance impact on real-time applications.

Steep learning curve

The kernel-space API is different. For example, malloc() needs to be replaced by one of the several types of memory allocations that the kernel can offer, such as kmalloc(), vmalloc(), alloc_pages() or get_zeroed_page(). There is a lot to learn before becoming productive.

Interface stability

The kernel-space API is less stable than user-space APIs, making maintenance a challenge.

Harder to debug

Debugging is very different in kernel space. Some tools often used by user-space applications can be used for the kernel. However, they represent exceptions rather than rule, where LTTNG [3] is an example of the exception. To compensate for this, the kernel has a lot of debug, tracing and profiling code that can be enabled at compile time.

Bugs more fatal

A crashing or misbehaving kernel tends to have a more severe impact on the system than a crashing or misbehaving application, which can affect robustness as well as how easy it is to debug.

Restrictive language choice

The kernel space is a very different programming environment than user space. It is more restricted, for example only C language is supported. This rules out any script based prototyping.

User-space drivers

If there are so many problems with having device drivers in kernel space, is it time to have all drivers in user space instead? As always, everything has its drawbacks, user-space drivers are no exception.

Most of the issues with kernel-space drivers are solved by having the driver in user space, but the issue with interface stability is only true for very simple user-space drivers.

For more advanced user-space drivers, many of the interfaces available for kernel-space drivers need to be re-implemented for user-space drivers. This means that interface stability will still be an issue.

Challenges with user-space drivers
There is a fear in the Linux kernel community that user-space drivers are used as a tool to avoid the kernel's GPLv2 license. This would undermine the idea with free open source software ideas that GPLv2 has. However, this is outside the scope of this paper.

Apart from this there are technical challenges for user-space drivers.

Interrupt handling
Without question, interrupt handling is the biggest challenge for a user-space driver. The function handling an interrupt is called in privileged execution mode, often called supervisor mode. User-space drivers have no permission to execute in privileged execution mode, making it impossible for user-space drivers to implement an interrupt handler.

There are two ways to deal with this problem: Either you do not use interrupts, which means that you have to poll instead. Or have a small kernel-space driver handling only the interrupt. In the latter case you can inform the user-space driver of an interrupt either by a blocking call, which unblocks when an interrupt occurs, or using POSIX signal to preempt the user-space driver.

Polling is beneficial if interrupts are frequent, since there is considerable overhead associated with each interrupt, due to the switch from user mode to supervisor mode and back that it causes. Each poll attempt on the other hand is usually only a check for a value on a specific memory address.

When interrupts become scarcer, polling will instead do a lot of work just to determine that there was no work to do. This is bad for power saving.

To get power saving when using user-space drivers with polling, you can change the CPU clock frequency, or the number of CPUs used, depending on work load. Both alternatives will introduce ramp-up latency when there is a work load spike.

DMA
Many drivers use hardware dedicated to copying memory areas managed by the CPU to or from memory areas managed by hardware devices. Such dedicated hardware is called direct memory access, or DMA. DMA relieves the CPU of such memory copying.

There are some restrictions on the memory area used for DMA. These restrictions are unique for each DMA device. Common restrictions are that only a certain physical memory range can be used, and that the physical memory range must be consecutive.

Allocating memory that can be used for DMA transfers is non-trivial for user-space drivers. However, since DMA memory can be reused, you only need to allocate a pool of memory to be used for DMA transfers at start-up. This means that the kernel could help with providing such memory when the user-space driver starts, but after that no further kernel interactions would be needed.

Device interdependencies
Devices are often structured in a hierarchy. For example the clock might be propagated in a tree-like fashion using different dividers for different devices and offer the possibility to power off the clock signal to save power.

There can be devices acting as a bridge, for example a PCI host bridge. In this case you need to setup the bridge in order to have access to any device connected on the other side of the bridge.

In kernel space there are frameworks helping a device driver programmer to solve these problems, but those frameworks are not available in user space.

Since it is usually only the startup and shutdown phases that affect other devices, the device interdependencies can be solved by a kernel-space driver, while the user-space driver can handle the actual operation of the device.

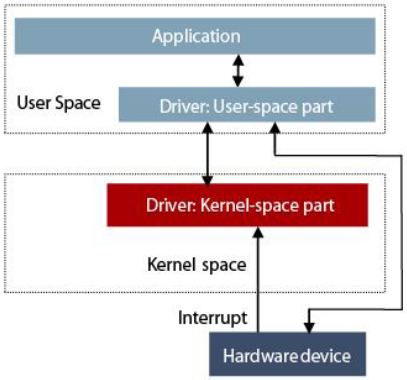
Kernel services
Network device drivers normally interfaces the kernel network stack, just like block device drivers normally interfaces the kernel file system framework.

User-space drivers have no direct access to such kernel services, and must re-implement them.

Client interface
The kernel has mechanisms for handling multiple clients accessing the same resource, and for blocking threads waiting for events or data from the device. These mechanisms are available using standard interfaces like file descriptors, sockets, or pipes.

To avoid using the kernel, the user-space driver needs to invent its own interface.

Implementing user-space drivers



The picture above shows how a user-space driver might be designed. The application interfaces the user-space part of the driver. The user-space part handles the hardware, but uses its kernel-space part for startup, shutdown, and receiving interrupts.

There are several frameworks and software solutions available to help designing a user-space driver.

UIO

There is a framework in the kernel called UIO [5] [4] which facilitate writing a kernel-space part of the user-space driver. UIO has mechanisms for providing memory mapped I/O accessible for the user-space part of the driver.

The allocated memory regions are presented using a device file, typically called /dev/uioX, where X is a sequence number for the device. The user-space part will then open the file and perform mmap() on it. After that, the user-space part has direct access to its device.

By reading from the same file being opened for mmap(), the user-space part will block until an interrupt occurs. The content read will be the number of interrupts that has occurred. You can use select() on the opened file to wait for other events as well.

For user-space network drivers there are specialized solutions specific for certain hardware.

DPDK

Data Plane Development Kit, DPDK [6] , is a solution from Intel for user-space network drivers using Intel (x86) hardware. DPDK defines an execution environment which contains user-space network drivers. This execution environment defines a thread for each CPU, called lcore in DPDK. For maximum throughput you should not have any other thread running on that CPU.

While this package of libraries focuses on forwarding applications, you can implement server applications as well. For server DPDK applications you need to implement your own network stack and accept a DPDK specific interface for accessing the network.

Much effort has been put in memory handling, since this is often critical for reaching the best possible performance. There are special allocation and deallocation functions that try to minimize TLB [10] misses, use the most local memory for NUMA [11] systems and ensure even spread on multi-channel memory architectures [12]

USDPAA

User-space Data Plane Acceleration Architecture, USDPAA [7] , is a solution from Freescale for the same use case as DPDK but designed for their QorIQ architecture (PowerPC and ARM. The big difference is that QorIQ uses hardware for allocating, de-allocating and queuing network packet buffers. This makes memory management easier for the application.

TransportNetLib

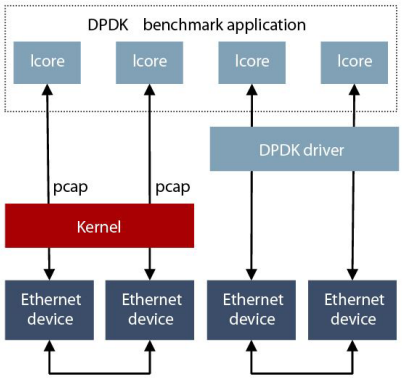
TransportNetLib [8] is a solution from Texas Instruments. It is similar to USDPAA but for the Keystone architecture (ARM).

Open DataPlane

Open DataPlane, ODP [9] , is a solution initiated by Linaro to do the same as DPDK, USDPAA and TransportNetLib, but with vendor generic interfaces.

Trying out DPDK

To get the feeling for the potential performance gain from having a user mode network device driver, a DPDK benchmark application was designed and executed.



ENEA

The design of the application can be seen in the picture above. It executes as four instances each running on its own CPU, or lcore, as DPDK calls them.

Each instance is dedicated to its own Ethernet device sending and receiving network packets. The packets sent has a magic word used for validating the packets and a timestamp used for measuring transport latency.

The instances are then paired using loopback cables. To be able to compare user-space driver with kernel-space driver, one pair accesses the hardware directly using the driver available in DPDK, and the other pair uses the pcap [13] interface. All four Ethernet devices are on the same PCI network card.

There is a fifth lcore (not shown in the picture above) which periodically collects statistics and displays it to the screen.

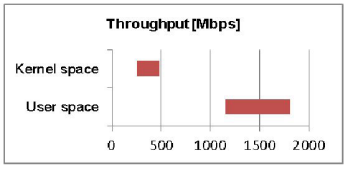
The hardware used was as follows:

- Supermicro A1SAi-2750F mother board using Intel Atom C2750 CPU. This CPU has 8 cores with no hyperthreading.
- 16GB of memory.
- Intel Ethernet server adapter i350-T4, 1000 Mbps.

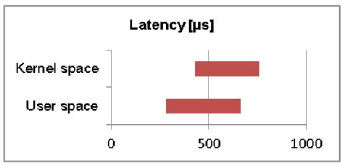
The table below shows the throughput and latency for user-space driver compared to kernel-space driver.

	User space	Kernel space
Throughput, Mbps	1152 – 1808	259 – 487
Latency, microseconds	281 – 666	431 – 762

A graph showing the throughput:



A graph showing the latency:



The theoretical throughput maximum is the sum of the send and receives speed for the network interface. In this case this is 1000 Mbps in each direction, giving a theoretical maximum of 2000 Mbps. The throughput includes packet headers and padding.

User-space driver achieved a throughput boost of about four times over kernel-space driver.

Latency was calculated by comparing the timestamp value found in the network packet with the current clock when packet was received. The latency for user-space driver was slightly less than for kernel-space driver.

Four threads, each continuously running netperf TCP streaming test against loop-back interface, were used as a stress while running the DPDK benchmark application. This had no noticeable impact on the measurements.

Conclusion

Implementing a user-space driver requires some work and knowledge. The major challenges are interrupts versus polling, power management and designing interface towards driver clients.

Support for user-space network drivers is a lot more developed than for other kinds of user-space drivers, especially for doing data plane forwarding type of applications.

A user-space driver can do everything a kernel-space driver can, except for implementing an interrupt handler.

Comparing a user-space network driver with a kernel-space network driver showed about four times better throughput for the user space driver. Latency did not show a significant difference.

The real-time characteristics should be good for user-space drivers since they do not invoke the kernel. This was not verified in this paper, though.

ENEA