DRIVER ARCHITECTURE IN A LINUX WORLD

John W. Linville, *Dux Linux*
LVL7 Systems, Inc.
13000 Weston Parkway, Suite 105, Cary, NC 27513, USA
linville@lvl7.com

Abstract

Vendors of modern networking hardware realize that the devices they provide require a great deal of software in order to function effectively. As a result, many hardware vendors provide large amounts of software for use with their devices. However, the prevalent use of the VxWorks®[1] operating system has allowed hardware vendors to become careless with the architecture of the driver software they provide. Often the driver software provided by hardware vendors is difficult to use in a protected environment like Linux®.[2] This paper begins by discussing strategies for making use of such drivers in a Linux® environment, and continues by discussing how hardware vendors might architect Linux®-friendly driver software. Finally, the paper discusses how vendors can provide first-class Linux® support for their hardware and why it is in their best interest to do so. This paper is a *must* read both for system integrators new to Linux® and for driver software architects unfamiliar with the Linux® environment.

1 Introduction

In the modern world of advanced networking, simple hardware just does not exist. Requirements for both speed and functionality have increased exponentially for many years. In many segments of the marketplace, designs relying on a system's CPU to forward frames, to learn addresses, or even to run simple protocols are at such a disadvantage that almost no one in those market segments produces such a system. Modern networking designs require hardware that is capable of off-loading basic networking functions from the CPU. Whether that hardware is a programmable co-processor or a configurable networking ASIC, complicated software is required to bond that hardware and the system's CPU into a cohesive networking system.

In order to encourage the use of this advanced but complicated hardware, networking hardware vendors provide driver software for use with their products. This software is generally not a formal device driver as defined by an operating system, but rather an informal collection of software designed to ease the use of the driven hardware. Nevertheless, this software is often incorporated directly into commercial end products. This situation has provided hardware vendors with the opportunity to differentiate their products by offering as much software support for exercising the features of their hardware as possible.

Today, most networking hardware vendors provide a large base of software to support their products. Such software typically contains hardware access routines, application support libraries, and even service tasks that perform maintenance services related to supported applications of the hardware. However, most hardware vendors are not in the business of selling software. As a result, hardware vendors tend to have limited software development resources. It is not surprising that these vendors tend to support only the most ubiquitous operating system platforms with their driver software.

For some time, VxWorks® has been the dominant operating system for embedded networking devices. Consequently, the primary focus for software support by networking hardware vendors has been VxWorks®. More recently, Linux® has become a strong player in the world of embedded systems and particularly in the embedded networking arena. At first glance, VxWorks® and Linux® may seem to have much in common with one another. For example, both support one or more of the industry standard Posix APIs. However, these two operating systems are distinctly different in many ways. Supporting one rarely implies support of the other.

[1] VxWorks is a registered trademark of Wind River Systems, Inc.
[2] Linux is a registered trademark of Linus Torvalds.

This situation is aggravated by the fact that many hardware vendors take advantage of the less restrictive "features" of the VxWorks® environment when implementing their informal driver software. The Linux® kernel enforces a strict driver model for hardware access and it segregates different software components into separate address spaces. In contrast, VxWorks® does not enforce any strict driver model or memory access protection.[3] The software provided by many hardware vendors makes no distinctions between hardware access routines, application support libraries, or system service tasks. The lack of such distinctions is incompatible with the protected environment provided by Linux®. As Figure 1 illustrates, the result is a software conglomerate that can be very difficult to integrate effectively into a Linux® system.

FIGURE 1: *Typical conglomerate driver*

2 The Naïve Approach

Linux® is a protected system. The Linux® kernel uses the memory management hardware available on modern CPUs not only to protect tasks from each other but also to protect the kernel from errant userland tasks. Furthermore, in order to ensure system integrity the kernel allows only "root"-owned processes to access physical address ranges. Even those processes are required to use specific programming interfaces in order to map physical address ranges into a process's virtual address space. Other types of hardware accesses are restricted entirely to code running in the kernel's context. This group includes interrupt handlers, DMA transfers, and any other hardware access that is not memory-mapped. These protections have great value, but they serve to make it very difficult to control hardware of any complexity entirely from userland.

As asserted earlier, modern networking hardware tends to be both highly functional and highly complex. Simple memory-mapped register access is generally insufficient for control of these devices. Other forms of hardware access generally require execution in kernel context, and vendor-provided conglomerate drivers tend to be poorly segregated between hardware access and higher-level functionality. The solution seems obvious: package the conglomerate driver in a way that allows it to execute within the context of the kernel.

A vendor-provided driver tends to have a top-level API that is intended for use by developers in creating a working system. Developers will expect to use this API when developing their networking applications. However, in this model the actual conglomerate driver will reside in a separate address space from the rest of the application. This makes traditional linking impossible. One solution is to provide a library for the application to link against that mimics the published API of the conglomerate driver. Underneath, this *doppleganger*[4] library will communicate to the kernel-resident code by established means, such as an `ioctl()` call to an actual Linux® device driver. Figure 2 illustrates the implementation of this approach.

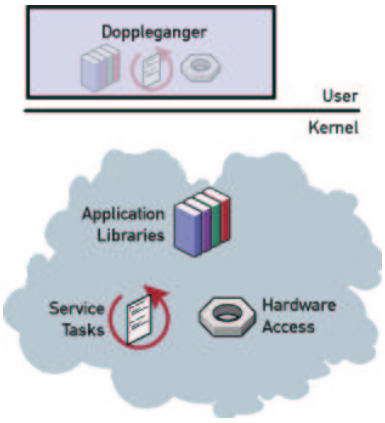

FIGURE 2: *The Naïve Approach*

In practice, this approach works reasonably well to provide access to a conglomerate driver's functionality from userland applications. However, the cost of switching between user and kernel contexts can accumulate to be quite large if driver APIs are called repeatedly or in rapid succession over a period of time. Also, many complex conglomerate drivers make use of application callbacks. Since there is no simple way to call userland functions from within the kernel's context, this method either precludes the use of callbacks or requires one to invent a means of simulating them appropriately for the given driver

[3] Wind River Systems, Inc. offers a product called VxWorks AE® which does offer a protection domain model. However, use of VxWorks AE® is not as prevalent as is the use of its predecessor. Therefore, all references to VxWorks® within this document should be interpreted to refer to the original VxWorks®.

[4] In German folklore, a doppleganger is the ghostly double of a living person—especially one that haunts its fleshy counterpart.

and application. Similarly, other means of application ⟷ driver communication that rely on shared memory may only be available with some difficulty, if at all.

Perhaps the biggest problem with this approach lies with the fact that standard Linux® kernels are not preemptible. While not generally a problem with properly written code, this can be quite a big problem with code that was not written with this environment in mind. Such software will generally not make any effort to bound its own execution time. Thus, improperly written code will very easily starve all of the other processes in a system. Since VxWorks® allows any or all tasks to be preemptible, vendor-provided conglomerate drivers generally fall into this category of improperly written code.

Note that non-preemptibility is a problem even if processes execute at a very low priority. Even though threads executing in the kernel's context receive no special consideration when it comes to schedul*ing*, once they are schedul*ed* they will continue to hold the CPU until they voluntarily relinquish it. Since it is presumable that every thread will be scheduled *eventually*, this is a potential problem with any thread executing within the kernel's context. There is a well-known kernel patch maintained by Robert Love that will tend to solve this particular problem.[1] However, not everyone will be using this patch at the present time or even in the near future.[5]

A less tangible problem with this approach relates to the amount of code running in the kernel's context. Vendor-provided conglomerate drivers tend to be rather large. These conglomerate drivers tend to provide a lot of functionality and are often targeted at multiple devices within a family of hardware. It is generally accepted that the number of bugs in a piece of software is proportional to the size of that software. So, the potential for bugs in driver code is fairly large. By placing code in the kernel's context, one makes it more difficult to debug that code and makes the consequences of bugs in that code greater than if the same code were running in a user's context. By itself, this should be enough to dissuade the experienced Linux® developer from pursuing this approach.

The Linux® kernel reserves most direct hardware access to code running in the kernel's context. Vendor-provided conglomerate drivers do nothing to segregate hardware access routines from library routines and service tasks. The simple solution seems to be to package the conglomerate driver in a way that lets it run in the kernel's context. Unfortunately, this approach is quite expensive in terms of context switches, it precludes certain common software practices, it can cause problems due to non-preemptibility in the kernel's context, and it puts extra code into the kernel's context where it is more difficult to debug and more dangerous in the event of failure. To the näive, this approach seems like a great idea. However, experience shows that it is rarely satisfactory. There must be a better way.

3 A Better Strategy

There are potentially many reasons why one may wish to preserve the code a vendor provides as a single conglomerate. This is particularly true if the conglomerate driver is an existing piece of developed and tested software. It has already been demonstrated that it is undesirable to run a conglomerate driver in the kernel's context. However, it is also possible to run most or all of a conglomerate driver in a user's context. This strategy can produce a workable system without major changes to an existing conglomerate of software.

As discussed earlier, the Linux® kernel restricts many kinds of hardware accesses to only code running in the kernel's context. Complex networking hardware will generally require more than simple memory-mapped register accesses in order to function properly. If the body of an existing conglomerate driver is to reside in a user's context, some means must be devised to account for these non-memory-mapped hardware accesses. The general approach to providing this type of access is to implement a small formal device driver that executes in the kernel's context. This small driver provides the required functionality (e.g. DMA transfer setup) through a small suite of custom `ioctl()` calls. As depicted in Figure 3, generally only a few small changes to the conglomerate driver are required to make this strategy work.

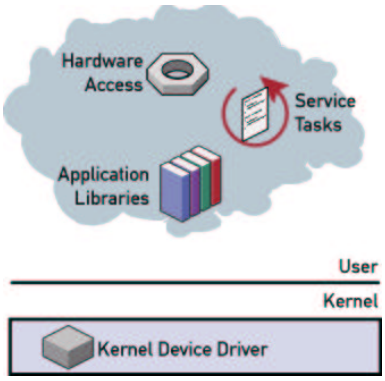

FIGURE 3: *A Better Strategy*

[5]This paper will avoid advocating the use of kernel patches that are currently outside the mainstream.

This strategy avoids many of the problems found with the previous approach. No special "shim" or *doppleganger* library is required to allow the application to access the conglomerate driver, and callbacks, two-way sharing of pointers, and other common software techniques are easily available to the body of the conglomerate driver's code. Thread preemption is not a problem since code running in a user's context is always preemptible. Debugging is possible using standard means,[6] and crashes do not automatically crash the whole system. In general, this is a much better strategy than the previously described approach.

However, this strategy is not a panacea. It may be very difficult to isolate those services that must run in the kernel's context from the rest of the conglomerate driver. Presuming that the conglomerate driver was not architected with this strategy in mind, it is likely that any *simple* partitioning will be less than optimal. It is very likely that some hardware accesses will occur more frequently than truly required and that others will be a small part of a larger atomic operation that has no other need to run in the kernel's context. Hence, this strategy is still prone to excessive amounts of expensive context switching or non-preemptible execution in the kernel's context. This situation is exacerbated if the networking hardware needs to be accessed through a Linux® net driver.[7] The Linux® networking stack is part of the kernel. As such it executes within the kernel's context. Consequently, Linux® net drivers execute within the kernel's context in order to provide access to networking hardware through standard Linux® networking APIs. If the current strategy is in use and the networking hardware needs an associated Linux® net driver, then either: a) the net driver needs to have a means of synchronizing hardware access between itself running in the kernel's context and the conglomerate driver running in a user's context; or, b) the net driver needs to be able to exchange incoming and outgoing frames with the conglomerate driver running in a user's context. Either of these possibilities is expensive with regard to context switches. In particular, option b) requires as many as three separate context switches for any given frame received at the CPU. Obviously, this is more context switching than is desirable.

This strategy also has an intangible problem. Most software architectural diagrams show the operating system and device drivers at the bottom and any applications at the top. With this strategy, the actual hardware driver lives somewhere in the middle of the diagram. So, this strategy is somewhat "upside-down" with regard to accepted norms. Of course, this is not a technical issue. However, the "upside-down" nature of this strategy can lead to much *confusion*. Confusion leads to *errors*, errors lead to *bugs*, and bugs lead to *failures*.

As we have already seen, running a conglomerate driver in the kernel's context has many problems. Still, it is very tempting to preserve existing code as a single entity. With slightly more effort, it is possible to run a mostly intact conglomerate driver in a user's context while still producing a workable system. However, this strategy is not without costs. Context switching is expensive, and this strategy requires more context switching than is desirable. Also, this strategy is "upside-down" when compared to how things are normally done. Confusion leads to failures and is best avoided if at all possible. While this strategy is better than the first approach, it seems clear that another solution must be found.

4 The Best Solution

The discussion so far has enumerated and demonstrated the problems with maintaining a conglomerate driver as a single entity in a Linux® environment. The cost of excessive context switching becomes significant, and the costs associated with debugging difficulties, system complexity, and design unconventionality are significant as well. While there may be defensible reasons for attempting to do so, using a conglomerate as-is can be quite expensive. It seems likely that the approaches described so far introduce as many problems as they solve.

So, what is the correct solution? All of the problems discussed so far stem from an architectural conflict: the Linux® kernel restricts most hardware accesses to code running in the kernel's context; and, conglomerate drivers do not adequately separate hardware access code from other conglomerate components. The deceptively simple answer is that one must remove this conflict between the Linux® architecture and the architecture of the conglomerate driver. This means dismantling the conglomerate driver and reconstructing it in a form compatible with the Linux® driver model.

How does one begin? As was discussed earlier, the typical conglomerate driver contains not only basic hardware access routines, but also application support library routines and even services tasks. In many cases the hardware access routines will need

[6] Unlike code running in the kernel's context, code running in a user's context is easily debugged using software debuggers and other common debugging means.

[7] Use of the Sockets API to communicate over networking hardware will require the implementation of a Linux® net driver for that hardware.

to run within the kernel's context. However, the other components almost always need to run in a user's context. The task at hand is to separate the two application-oriented components from the system-oriented hardware access component. The result should look something like the depiction in Figure 4 below.

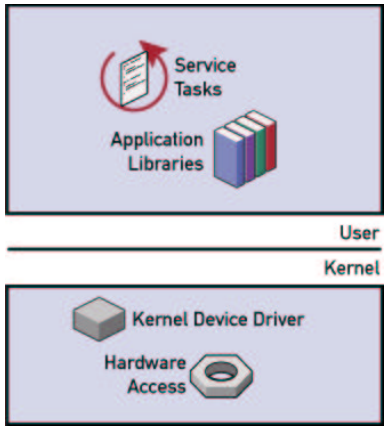

FIGURE 4: *The Best Solution*

One starts by sorting the pieces of the conglomerate driver into a hardware access group and an "other" group. Sometimes this will be a straightforward process, but often it will require quite a bit of code analysis. At this point it will likely be necessary to begin breaking existing files into hardware access and "other" versions of the original files. Also, many existing functions will need to be modified either to remove unnecessary hardware accesses or to formalize those accesses in a way that is supportable in the Linux® environment. The end result should be two distinct object libraries. The hardware access library should have no dependencies on the "other" library, while the "other" library should at most require access to the functions defined in the hardware access library. In particular, there should be absolutely no use of shared global variables between the two libraries and no calling of functions defined in the "other" library from functions defined in the hardware access library.

With the former conglomerate driver broken into these two distinct libraries, one has the basic ingredients necessary for implementing this solution. The hardware access library can be packaged to run in the Linux® kernel's context using a wrapper that implements one or more of the standard Linux® driver interfaces. The "other" library is used for linking against applications requiring access to the hardware controlled by the former conglomerate driver. The interface between the hardware access library and the "other" library is implemented either by adding Linux®-specific code to the "other" library or by creating a *doppleganger* library that provides the same symbol definitions as the hardware access library but uses Linux®-specific code to drive the hardware access code running in the kernel's context. The end result is a functional solution in nearly every case.

In most cases, however, this solution is not for the faint hearted. Once a conglomerate driver has existed for some time, the lack of architectural discipline amongst conglomerate driver developers tends to lead to a spaghetti-like mess of hardware accesses within service tasks, application callbacks from within interrupt handlers, and other questionable practices that can be difficult to translate to the Linux® environment. Sorting-out such a tangle of software can require a great deal of time, energy, and motivation. An intimate knowledge of both the conglomerate driver and the hardware it drives is required as well. Still, in the author's opinion the resulting "purified" driver is well worth the cost. That is especially true if quality, performance, or maintainability are desirable attributes for one's Linux®-based product.[8]

Unfortunately, this solution has a greater upfront cost than the approaches discussed so far. One must be willing to dig deeply into code that most developers consider a "black box." This can require a large investment of development time and resources. Properly separating a conglomerate driver into hardware access and "other" libraries requires not only an intimate knowledge of the conglomerate driver and the hardware it drives, but also enough Linux® knowledge and experience to accomplish the task. Covering both of those competencies can be difficult, and the investment of time and resources can be expensive. However, the payoff is in better performance, easier debugging, and greater stability. In the long run this is the cheapest of the alternatives discussed so far.

5 Driver Architecture 101

So far, the discussion has been about how to make use of an existing conglomerate driver in a Linux® environment. This discussion has been pertinent because the prevalent use of VxWorks® in the embedded networking arena has fostered the proliferation of poorly architected conglomerate drivers. But drivers would not require difficult adaptation for Linux®if they were developed properly in the first place! A set of driver libraries that are architected in a Linux®-friendly way should work equally well with VxWorks® or any other operating system.

[8] Hopefully this applies to everyone...

How does one structure a driver library so that it works equally well with both Linux® and operating systems like VxWorks®? A proper driver architecture should provide for a hardware access layer. This layer should encapsulate all of the atomic operations one would want to perform on the hardware. Once defined, this layer should not be violated for any reason whatsoever. Also, any dependencies on the hardware access must be "one-way"—calls from the hardware access layer to higher layers are specifically *not* allowed. In Linux®, the hardware access layer will be run within the kernel's context. In an unprotected operating system, this layer will be just another part of the driver library.

A driver architected in this way should work reasonably well either in a protected operating system like Linux® or in an unprotected operating system like VxWorks®. A driver library architected in this fashion from the beginning should introduce neither extra runtime overhead nor extra complexity in either a protected or unprotected environment. Furthermore, this basic design framework is more in line with modern software design practices and should lead to fundamentally better software with fewer bugs and better stability. There really is no excuse for not following this basic design framework when architecting a new hardware driver.

As demonstrated, there are a number of ways to approach the problem of adapting an existing conglomerate driver to a Linux environment. However, this problem should not exist in the first place! A driver library architected in a Linux®-friendly way will work equally well in either a protected environment like Linux® or an unprotected environment like VxWorks®. And, due to the better design principles in use, a driver architected in such a way will tend to be better software. Clearly this is the way hardware driver libraries should be done.

6 Embracing Linux®

Not that long ago, Linux® was an operating system *only* for desktop and server computers. However, in the past few years Linux® has become quite a significant player in the world of embedded systems and particularly in the embedded networking arena. Some have suggested that Linux® will become *the* dominant operating system in the embedded systems market.[2] What is certain is that Linux® will continue to be a significant player in the embedded networking arena for the near future.

Given the growing popularity of Linux® in the embedded arena, it seems reasonable that a hardware vendor may want to make its hardware as attractive as possible to those using Linux®. How might a vendor do this? A good first step is to ensure that the vendor's driver architecture is Linux®-friendly. A vendor might do this either by purifying what exists now or by making sure that the driver is architected correctly in the first place. As well, the vendor should conduct the exercise of running the driver under Linux® in order to ensure that the driver actually works in the Linux® environment. These exercises will serve to guarantee that the Linux® market is available to the vendor's hardware.

Michael Tiemann,[9] CTO of Red Hat, Inc., has suggested that Linux® should be viewed as a platform rather than simply as an operating system.[3] What does this mean for a hardware vendor? For one thing, this means that the vendor should implement standard Linux® device drivers as appropriate for the capabilities of the vendor's devices. Moreover, a wise vendor will determine how to take advantage of existing Linux® functionality[10] as well as how to extend it in ways that are appropriate for the vendor's hardware. In this way, the vendor's hardware becomes accessible to any software already available for Linux® that is related to the functions provided by the vendor's hardware. It should be clear that this would benefit both the hardware vendor and the system integrator using Linux®.

That Linux® will be a significant embedded operating system for at least the near future seems certain. Linux® users are more likely to use hardware with good Linux® support. This is especially true since hardware that is well supported by Linux® is unlikely to require much custom software to make use of it. The benefits of embracing Linux® should be obvious.

7 Conclusion

For some time, VxWorks® has dominated the market for embedded networking operating systems. This market dominance has allowed networking hardware vendors to concentrate *only* on supporting VxWorks® with the software they provide to drive their hardware. Unfortunately, the unprotected nature of VxWorks® has fostered the development of driver software that is difficult to use with a protected operating system like Linux®. There are ways to use a more-or-less intact conglomerate driver with Linux®, but the results are generally sub-optimal.

Following simple rules, it is possible to re-architect an existing conglomerate driver in order to make

[9] Michael Tiemann was a co-founder of Cygnus, the first company with a business built around Open Source technologies.

[10] Linux® has extensive support for networking e.g. routing, bridging, VLANs, LAGs, etc.

it more compatible with Linux®-like operating systems. The results achieved by using a "purified" driver can be quite satisfactory. However, conducting such a purification requires a larger commitment of resources than the average organization is willing to make. As usual, much better results are achieved by doing the job right the first time.

A properly architected driver will work equally well with either protected or unprotected operating systems. There should be neither a runtime penalty nor a development penalty for doing things correctly in the first place. Such a driver will open a much wider market for the hardware it drives. Why would one build a driver any other way?

References

[1] Andrews, Jeremy, 2001, *Interview: Robert Love*, KernelTrap, http://kerneltrap.org/... .../node.php?id=1.

[2] Sprackland, Teri, 2002, *Embedded Market Accepts Linux*, Nikkei Electronics Asia, http://www.asiabiztech.com/nea/200205/... .../cmpu_183147.html.

[3] Tiemann, Michael, 2002, *How Linux Will Revolutionize the Embedded Market*, LinuxDevices.com, http://www.linuxdevices.com/articles/... .../AT7248149889.html.