Linux Kernel Infrastructure for User-Level Device Drivers

Peter Chubb[*]
peterc@gelato.unsw.edu.au

January, 2004

Abstract

Linux 2.5.x has good support now for user-mode device drivers — XFree being the biggest and most obvious — but also there is support for user-mode input devices and for devices that hang off the parallel port.

The motivations for user-mode device drivers are many:

- Ease of development (all the normal user-space tools can be used to write and debug, not restricted to use of C only (could use Java, C++, Perl, etc), fewer reboot cycles needed, fewer restrictions on what one can do with respect to reentrancy and interrupts, etc., etc.)
- Ease of deployment (kernel $\leftrightarrow$ user interfaces change much more slowly than in-kernel interfaces; no licensing issues; no need for end-users to recompile to get module versions right, etc., etc.))
- Increased robustness (less likely that a buggy driver can cause a panic)
- Increased functionality. Some things are just plain easier to do in user space than in the kernel — e.g., networking.
- Increased simplicity (rather than have, say, a generic IDE controller that has to understand the quirks of many different kinds of controllers and drivers, you can afford to pick at run time the controller you really need)

There are however some drawbacks, the main ones being performance and security.

Three recent developments have made it possible to implement an infrastructure for user-level device drivers that perform almost as well (in some cases better than) in-kernel device drivers. These are

1. the new pthreads library (and corresponding kernel changes: futexes, faster clone and exit, etc);
2. fast system call support; and
3. IOMMUs.

Now that many high-end machines have an IOMMU, it becomes possible, at least in theory, to provide secure access to DMA to user processes.

Fast system calls allow the kernel $\leftrightarrow$ user crossing to be extremely cheap, making user-process interrupt handling feasible.

And fast context-switching and IPC for Posix threads, means that multi-threaded device drivers can have the kind of performance that until recently was only available in-kernel.

[*]This work was funded by HP, NICTA, the ARC, and the University of NSW through the Gelato programme (http://www.gelato.unsw.edu.au)

1 Introduction

Normal device drivers in Linux[1] run in the kernel's address space with kernel privilege. This is not the only place they can run —see figure 1.

	Kernel	A	
Address Space	Client		B
	Own	C	D
		Kernel	User
		Privilege	

Figure 1: Where a Device Driver can Live

Point A is the normal Linux device driver, linked with the kernel, running in the kernel address space with kernel privilege.

Device drivers can also be linked directly with the applications that use them (Point B) —the so-called 'in-process' device drivers proposed by Keedy (1979) —or run in a separate process, and be talked to by an IPC mechanism (for example, an X server, point D). They can also run with kernel privilege, but with only a subset of the kernel address space (Point C) (as in the Nooks system: Swift, Martin, Levy and Eggers, 2002).

2 Motivation

Traditionally, device drivers have been developed as part of the kernel source. As such, they *have* to be written in C, and they have to conform to the (rapidly changing) interfaces and conventions used by kernel code. Even though drivers can be written as modules (obviating the need to reboot to try out a new version of the driver[2]), in-kernel driver code has access to all of kernel memory, and runs with privileges that give it access to all instructions (not just unprivileged ones) and to all I/O space. As such, bugs in drivers can easily cause kernel lockups or panics. And various studies (e.g., Chou (2001) estimate that more than 85% of the bugs in an operating system are driver bugs.

Device drivers that run as user code, however, can use any language, can be developed using any IDE, and can use whatever internal threading, memory management, etc., techniques are most appropriate. When the infrastructure for supporting user-mode drivers is adequate, the processes implementing the driver can be killed and restarted almost with impunity as far as the rest of the operating system goes.

Drivers that run in the kernel have to be updated regularly to match in-kernel interface changes. Third party drivers are therefore usually shipped as source code (or with a compilable stub encapsulating the interface) that has to be compiled against the kernel the driver is to be installed into.

Drivers for uncommon devices (or devices that the mainline kernel developers do not use regularly) tend to lag behind. For example, in the 2.6.0-test10 kernel, there are 80 drivers known to be broken because they have not been updated to match the current APIs, and a number more that are still using APIs that have been deprecated.

User/kernel interfaces tend to change much more slowly than in-kernel ones; thus a user-mode driver has much more chance of not needing to be changed when the kernel changes. Moreover, user mode drivers can be distributed under licences other than the GPL, which may make them attractive to some people.

User-mode drivers can be either closely or loosely coupled with the applications that use them. Two obvious examples are the X server (XFree86) which uses a socket to communicate with its clients and so has isolation from kernel and client address spaces and can be very complex; and the Myrinet drivers, which are usually linked into their clients to gain performance by eliminating context switch overhead on packet reception.

The Nooks work (Swift et al 2002) showed that by isolating drivers from the kernel address space, the most common programming errors could be made recoverable. In Nooks, drivers are insulated from the rest of the kernel by running them in a separate address space, and replacing the driver $\leftrightarrow$ kernel interface with a new one that used cross-domain procedure calls to replace any procedure calls in the ABI, and that created shadow copies in the protected address space of any shared variables.

This approach provides isolation, but also has problems: as the driver model changes, there is

[1] Linux is a registered trademark of Linus Torvalds

[2] except that many drivers, including the siimage.o module, currently cannot be unloaded

quite a lot of wrapper code that will have to be changed to accommodate the changed APIs. Also, the values of shared variables is frozen for the duration of a driver ABI call. The Nooks work was uniprocessor only; locking issues therefore have not yet been addressed.

Windriver allows development of user mode device drivers. It loads a proprietary device module `/dev/windrv6`; user code can interact with this device to setup and teardown DMA, catch interrupts, etc.

3 Existing Support

Linux has good support for user-mode drivers that do not need DMA or interrupt handling —see, e.g., Nakatani (2002) .

The `ioperm()` and `iopl()` system calls allow access to the first 65536 I/O ports; and one can map the appropriate parts of '`/proc/bus/pci/...`' to gain access to memory-mapped registers.

It is usually best to use MMIO if it is available, because on many 64-bit platforms there are more than 65536 ports (and on some architectures the ports are emulated by mapping memory anyway).

For particular devices —USB input devices, SCSI devices, devices that hang off the parallel port, and video drivers such as XFree86 —there is explicit kernel support. By opening a file in '`/dev`', a user-mode driver can talk through the USB hub, SCSI controller, AGP controller, etc., to the device. In addition, the `input` handler allows input events to be queued back into the kernel, to allow normal event handling to proceed.

'`libpci`' allows access to the PCI configuration space, so that a driver can determine what interrupt, IO ports and memory locations are being used; (and to determine whether the device is present or not).

There is an example driver to make the PC Speaker 'beep' at 440Hz at http://www.gelato.unsw.edu.au/patches/pcspeaker.c. It shows how to use ioperm() to gain access to various ports, and the use of inb and outb to adjust the values in device registers.

Even from user space, of course, it is possible to reprogram your machine so that it is unusable. Other recent changes —an improved scheduler, better and faster thread creation and synchronisation, a fully preemptive kernel, and faster system calls —mean that it is possible to write a driver that operates in user space that is almost as fast as an in-kernel driver.

4 Implementing the Missing Bits

The parts that are missing are:

1. the ability to claim a device from user space so that other drivers do not try to handle it;
2. The ability to deliver an interrupt from a device to user space,
3. the ability to set up and tear-down DMA between a device and some process's memory, and
4. the ability to loop a device driver's control and data interfaces into the appropriate part of the kernel (so that, for example, an IDE driver can appear as a standard block device).

The work at UNSW covers only PCI devices, as that is the only bus available on all of the architectures we have access to (IA64, X86, MIPS, PPC, alpha and arm), and does not yet cover point 4.

4.1 PCI interface

Each device should have only a single driver. Therefore one needs a way to associate a driver with a device, and to remove that association automatically when the driver dies. This has to be implemented in the kernel, as it is only the kernel that can be relied upon to clean up after a failed process. The simplest way to keep the association and to clean it up in Linux is to implement a new filesystem, using the PCI namespace. Open files are automatically closed when a process dies, so cleanup also happens automatically.

A new system call, `usr_pci_open(int bus, int slot, int fn, const char *name)` returns a file descriptor. Internally, it calls `pci_enable_device()` and `pci_set_master()` to set up the PCI device after doing the standard filesystem boilerplate to set up a vnode and a `struct file`.

Subsequent attempts to open the same PCI device will fail with `-EBUSY`.

When the file descriptor is finally closed, the PCI device can be released, and any DMA mappings removed. All files are closed when a process dies, so if there is a bug in the driver that causes it to crash, the system recovers ready for the driver to be restarted.

4.2 DMA handling

On low-end systems, it's common for the PCI bus to be connected directly to the memory bus, so setting up a DMA transfer means merely pinning the appropriate bit of memory (so that the VM system can neither swap it out nor relocate it) and then converting virtual addresses to physical addresses.

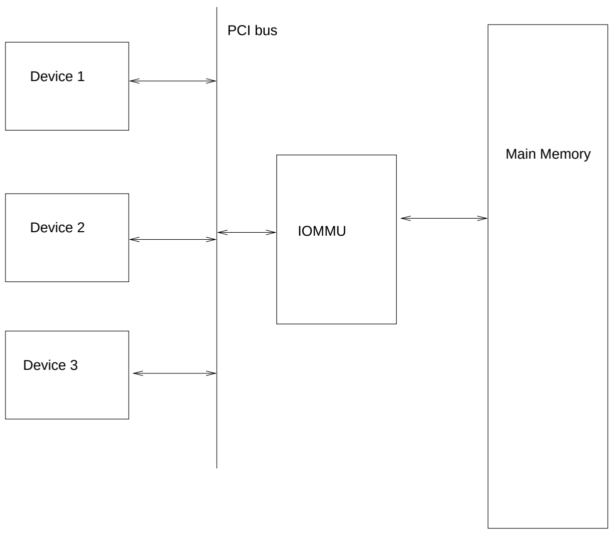

Figure 2: The IO MMU

Many modern architectures have an IO memory management unit (see figure 2), to convert from physical to I/O bus addresses —in much the same way that the MMU converts virtual to physical addresses. The MMU also protects one virtual address space from another. Unfortunately, the IO MMU cannot totally virtualise the PCI bus addresses. It can be set up to prevent DMA via bus addresses to arbitrary locations, but all mappings are seen by all PCI devices.

On such systems, after the memory has been pinned, the IOMMU has to be set up to translate from bus to physical addresses; and then after the DMA is complete, the translation can be removed from the IOMMU.

There are, in general, two kinds of DMA, and this has to be reflected in the kernel interface:

1. Bi-directional DMA, for holding scatter-gather lists, etc., for communication with the device. Both the CPU and the device read and write to a shared memory area. Typically such memory is uncached, and on some architectures it has to be allocated from particular physical areas. This kind of mapping is called *PCI-consistent*; there is an internal kernel ABI function to allocate and deallocate appropriate memory.

2. Streaming DMA, where, once the device has either read or written the area, it has no further immediate use for it.

I implemented a new system call[3], `usr_pci_map()`, that does one of three things:

1. Allocates an area of memory suitable for a PCI-consistent mapping, and maps it into the current process's address space; or

2. Converts a region of the current process's virtual address space into a scatterlist in terms of virtual addresses (one entry per page), pins the memory, and converts the scatterlist into a list of addresses suitable for DMA (by calling `pci_map_sg()`, which sets up the IOMMU if appropriate), or

3. Undoes the mapping in point 2.

The file descriptor returned from `usr_pci_open()` is an argument to `usr_pci_map()`. Mappings are tracked as part of the private data for that open file descriptor, so that they can be undone if the device is closed (or the driver dies).

Underlying `usr_pci_map()` are the kernel routines `pci_map_sg()` and

[3]Although multiplexing system calls are in general deprecated in Linux, they are extremely useful while developing, because it is not necessary to change every architecture-dependent '`entry.S`' when adding new functionality

`pci_unmap_sg()`, and the kernel routine `pci_alloc_consistent()`.

4.2.1 The IOMMU

Some machines (including the ZX2000 series from HP) interpose a translation look-aside buffer between the PCI bus and main memory, allowing even thirty-two bit cards to do single-cycle DMA to anywhere in the sixty-four bit memory address space.

As currently implemented, these devices allow themselves to be bypassed if the card about to do the DMA can address the memory it is DMAing to. For fully secure user-space drivers, one would want this capability to be turned off, and also to be able to associate a range of PCI bus addresses with a particular card, and disallow access by that card to other addresses.

4.3 Interrupt Handling

There are essentially two ways that interrupts can be passed to user level.

They can be mapped onto signals, and sent asynchronously. This is a good intuitive match for what an interrupt *is*, but has other problems:

1. One is fairly restricted in what one can do in a signal handler, so a driver will usually have to take extra context switches to respond to an interrupt (into and out of the signal handler, and then perhaps the interrupt handler thread wakes up)
2. Signals can be slow to deliver on large systems, as they require the process table to be searched to find the appropriate target process. It would be possible to short circuit this to some extent.
3. One needs an extra mechanism for registering interest in an interrupt, and for tearing down the registration when the driver dies.

For these reasons I decided to map interrupts onto file descriptors. '`/proc`' already has a directory for each interrupt (containing a file that can be written to to adjust interrupt routing to processors); I added a new file to each such directory. Suitably privileged processes can open and read these files.

The files have open-once semantics; subsequent attempts to open them will return -1 with `EBUSY`.

When an interrupt occurs, the in-kernel interrupt handler disables just that interrupt in the interrupt controller, and then does an `up()` operation on a semaphore.

When a process reads from the file, it enables the interrupt, then calls `down()` on a semaphore, which will block until an interrupt arrives.

The actual data transferred is immaterial, and in fact none ever is transferred; the `read()` operation is used merely as a synchronisation mechanism.

Obviously, one cannot share interrupts between devices if there is a user process involved. The in-kernel driver merely passes the interrupt onto the user-mode process; as it knows nothing about the underlying hardware, it cannot tell if the interrupt is *really* for this driver or not. As such it always reports the interrupt as 'handled'.

5 Driver Structure

The user-mode drivers developed at UNSW are thus structured as a preamble, an interrupt thread, and a control thread (see figure 3).

The preamble:

1. Uses '`libpci.a`' to find the device or devices it is meant to drive,
2. Calls `usr_pci_open()` to claim the device, and
3. Spawns the interrupt thread, then
4. Goes into a loop collecting client requests.

The interrupt thread:

1. Opens '`/proc/irq/`**irq**`/irq`'
2. Loops calling `read()` on the resulting file descriptor and then calling the driver proper to handle the interrupt.

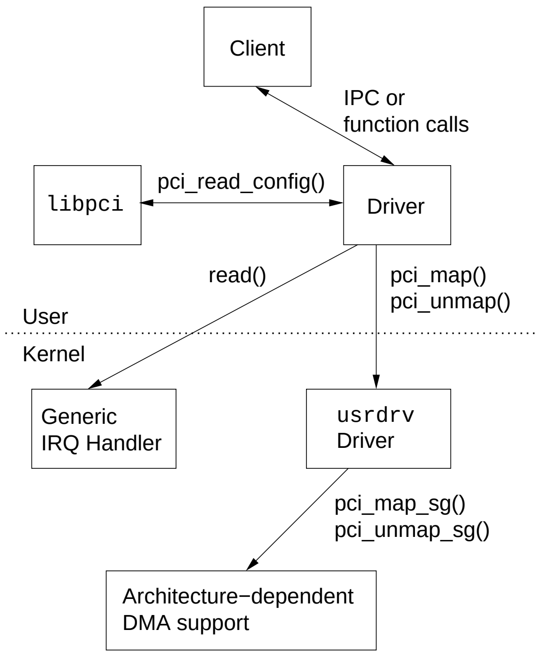

Figure 3: Architecture of a User-Mode Device Driver

3. The driver handles the interrupt, calls out to the control thread(s) to say that work is completed or that there has been an error, queues any more work to the device, and then repeats from step 2.

The control thread queues work to the driver then sleeps on a semaphore. When the driver, running in the interrupt thread, determines that a request is complete, it signals the semaphore so that the control thread can continue. (The semaphore is implemented as a pthreads mutex).

The driver has to do three system calls per I/O: one to wait for the interrupt, one to set up DMA and one to tear it down afterwards.

This could be reduced to two calls, by combining the DMA setup and teardown into a single system call.

The driver relies on system calls and threading, so the fast system call support available in IA64 Linux, and the NPTL are very important to get good performance. Each physical I/O involves at least four system calls, plus whatever is necessary for client communication: a `read()` on the interrupt FD, calls to set up and tear down DMA, and a `futex()` operation to wake the client.

6 Results

Device drivers were coded up by Leslie (2003) for a CMD680 IDE disc controller, and by another PhD student here for a DP83820 Gigabit ethernet controller

6.1 IDE driver

The disc driver was linked into a program that read 64 Megabytes of data from a Maxtor 80G disc into a buffer, using varying read sizes. Control measurements were made using Linux's in-kernel driver, and a program that read 64M of data from the same on-disc location using the raw device interface and the same read sizes.

At the same time as the tests, a low-priority process attempted to increment a 64-bit counter as fast as possible. The number of increments was calibrated to processor time on an otherwise idle system; reading the counter before and after a test thus gives an indication of how much processor time is available to processes other than the test process.

The initial results were disappointing; the user-mode drivers spent far too much time in the kernel. This was tracked down to `kmalloc()`; so the `usr_pci_map()` function was changed to maintain a small cache of free mapping structures instead of calling `kmalloc()` and `kfree()` each time. This resulted in the performance graphs in figure 4.

The two drivers compared are the new CMD680 driver running in user space using both the NPTL and the old LinuxThreads, and Linux's in-kernel SIS680 driver. As can be seen, there is very little to choose between them when the requested transfer size is above 16k. The new Posix threads implementation is slightly faster than the old one.

The graphs show average of ten runs; the standard deviations were calculated, but are negligible.

Each transfer request takes five system calls to do, in the current design. The client queues work to the driver, which then sets up DMA for the transfer (system call one), starts the transfer, then returns to the client, which then sleeps on a semaphore (system call two). The interrupt thread has been sleeping in `read()`, when the controller finishes its DMA, it cause an interrupt, which

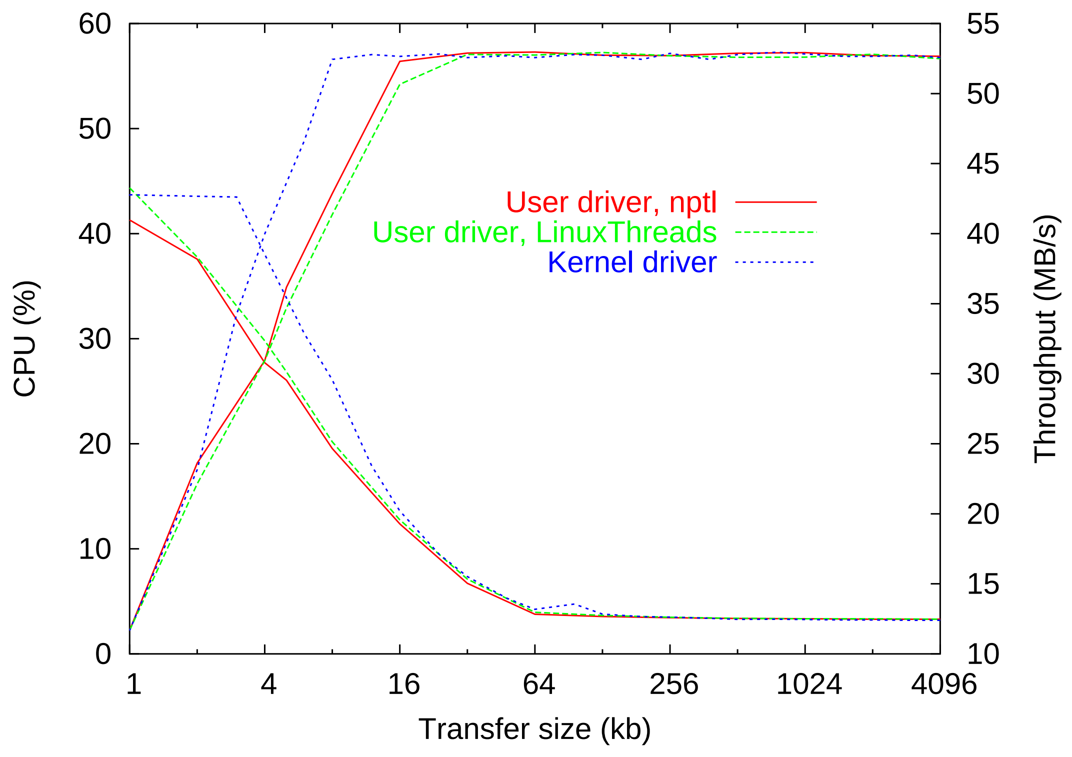

Figure 4: Throughput and CPU usage for the user-mode IDE driver on Pentium-4

wakes the interrupt thread (half of system call three). The interrupt thread then tears down the DMA (system call four), and starts any queued and waiting activity, then signals the semaphore (system call five) and goes back to read the interrupt FD again (the other half of system call three).

When the transfer is above 128k, the IDE controller can no longer do a single DMA operation, so has to generate multiple transfers

The time spent in this driver is divided as shown in figure 5

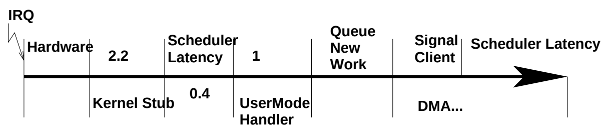

Figure 5: Timeline (in μseconds)

6.2 Reliability and Failure Modes

In general the user-mode drivers are very reliable. Bugs in the drivers that would cause the kernel to crash (for example, a null pointer reference inside an interrupt handler) cause the driver to crash, but the kernel continues. The driver can then be fixed and restarted.

7 Future Work

The main focusses of our work now lie in:

1. Reducing the need for context switches and system calls by merging system calls, and by trying new driver structures.

2. Adding mechanisms for looping back the 'top end' of the user mode drivers into the kernel, so that, for example, the disc driver in user space can talk to a file system running in the kernel. We're looking at Jeremy Elson's work on FUSD, but as it supports only character devices, we're also writing our own.

3. Improving robustness and reliability of the user-mode drivers, by experimenting with the IOMMU on the ZX1 chipset of our Itanium-2 machines.

4. Measuring the reliability enhancements, by using artificial fault injection to see what problems that cause the kernel to crash are recoverable in user space.

5. User-mode filesystems.

In addition there are some housekeeping tasks to do before this infrastructure is ready for inclusion in a 2.7 kernel:

1. Replace the ad-hoc memory cache with a proper slab allocator.

2. Clean up the system call interface

8 Where d'ya Get It?

Patches against the 2.6 kernel are sent to the Linux kernel mailing list, and are on http://www.gelato.unsw.edu.au/patches

Sample drivers will be made available from the same website.

References

[1] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating systems errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001. http://citeseer.nj.nec.com/article/chou01empirical.html.

[2] Jeremy Elson. A framework for user level file systems.

http://www.circlemud.org/~jelson/software/fusd, October 2003.

[3] Windriver.

www.jungo.com/windriver.html, 2003.

[4] J. L. Keedy. A comparison of two process structuring models. MONADS Report 4, Dept. Computer Science, Monash University, 1979.

[5] Ben Leslie and Gernot Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, Operating Systems and Distributed Systems Group, School of Computer Science and Engineering, The University of NSW, March 2003.

CSE techreports website ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW-0303.pdf.

[6] Bryce Nakatani. ELJOnline: User mode drivers.

http://www.linuxdevices.com/articles/AT5731658926.html, 2002.

[7] Michael Swift, Steven Martin, Henry M. Leyand, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept 2002.