Encapsulated User-Level Device Drivers in the Mungi Operating System

Ben Leslie, Nicholas FitzRoy-Dale and Gernot Heiser

School of Computer Science and Engineering &
National ICT Australia
University of NSW, Sydney 2052, Australia
{benjl,nfd,gernot}@cse.unsw.edu.au

Abstract

The reliability of device drivers is of critical importance to the overall stability of computer systems. This paper presents the software architecture used for user-level device drivers in the Mungi operating system. We argue that this framework provides a safer environment in which to run device drivers, while making device driver implementation easier and more flexible, thus improving overall reliability of the system.

1. Motivations

This paper presents a software framework for running device drivers as user-level components, rather than inside the kernel. There are two main reasons this is desirable: isolation and flexibility.

Recent studies [CYC+01, SBL03] have shown that bugs in device drivers are the most common source of failure in operating systems. One approach that can be used to minimise the impact of these bugs is to isolate device drivers from the rest of the system. The first part of isolation is to define a set of *interfaces* through which drivers communicate with the rest of the system. This is not all that different from existing systems, which define an internal kernel API. However, such drivers will have full access to all the kernel's internal state, and therefore can (and often do) bypass interfaces.

To enforce encapsulation we run drivers without special privileges, i.e. in user mode rather than as part of the kernel. They can then be encapsulated in address spaces like any user processes, thereby protecting the rest of the system from misbehaving drivers. This forces drivers to use only the specified interfaces.

Our other motivation for the development of this framework is to provide a safe and more flexible environment in which device drivers can be developed and used. Developing code for an operating system kernel can be a time consuming and difficult process for a number of reasons. Firstly, tools for debugging kernel code are not as comprehensive as user-level debugging tools. Secondly, the kernel programming environment is usually restricted to a single language and limited runtime libraries. Finally the program-test-debug cycle can be quite long as restarting a device driver usually means rebooting the system.

2. Mungi device driver framework

The device driver framework [Les02] is designed for the Mungi operating system [HEV+98]. Mungi is based on the idea of a single address space [CLFL94] and supports a component model with hardware-enforced encapsulation for providing secure user-level extensions to the kernel [EH01]. Mungi components hide their instance data from external access (other than via declared method interfaces). Otherwise, the system places no restrictions on components and their implementation. In particular, any language can be used to implement a component.

Interfaces are specified in an *interface definition language*. The systems enforces encapsulation so only the interfaces explicitly exported by a component can be accessed.

There are many different ways of structuring device drivers in a system. Usually they are simply directly compiled into the OS kernel, or are modules that can be dynamically loaded into the kernel. In other systems device drivers are simply libraries that are compiled directly into the client program [vEBBV95, Dam98]. This approach is severely limited because it restricts the device to only being used by one program. It is also does not provide any form of protection as the library is given full access to the hardware.

In the Mungi driver framework each device driver is written as a *component class*, with a *component instance* being instantiated for each actual device present in the system.

2.1 Fine-grained hardware access

Device drivers obviously need to access hardware in order to correctly control a device. This usually means being able to read and write to hardware registers, receive interrupts and perform *direct memory access* (DMA). The Mungi operating system provides primitives that allow each of these resources to be handled in a fine-grained manner. This ensures that each device driver can only access the hardware resources that it actually requires.

In most modern hardware architectures device registers are mapped into physical memory. We use the *memory management unit* (MMU) to control a device driver's access to hardware registers. The granularity of protection in this case is limited to the page size, however in practise each device's hardware registers are mapped into different pages of physical memory, so this does not usually cause problems.[1]

Each device driver instance must be registered to receive only specific interrupts. During interrupt handling the Mungi kernel looks up the device driver it should deliver the interrupt to, and then invokes the driver's interrupt interface. This arrangement allows the operating system to regulate the rate at which interrupts are delivered to device drivers, which supports better resource management than traditional systems.

Although controlling access to device registers and interrupts is fairly straightforward, restricting DMA is difficult. This is a problem because if a device driver can perform unrestricted DMA it can circumvent normal memory protection and corrupt the operating system. However, many modern systems feature an I/O MMU, which maps (device-visible) PCI addresses to (physical) RAM addresses. We use this hardware feature for restricting device access to physical memory [LH03].

2.2 Replacing `ioctl`

Most hardware devices provide more functionality than just I/O. This functionality is generally hard to generalise across devices, which led UNIX-like systems to the unstructured `ioctl` system call. By basing the driver framework on an underlying component system we allow device drivers to export this functionality as well-defined methods. Although this doesn't solve the underlying problem of many devices presenting different interfaces, it does allow this functionality to exported in a controlled manner and for a client programmer to query a device and determine what functionality it provides. This is a clear advantage over the `ioctl` design.

2.3 Fine-grained device driver access

Our system not only support fine-grained access to the underlying hardware, as discussed in Section 2.1, but also supports fine-grained access to the drivers themselves. The Mungi system uses password capabilities [APW86] to control access to both memory and component method invocation. This allows us to not only control access to the device itself, but any particular method on the device. This, combined with the ability to define each of the control methods on a device, provides a more flexible system than the traditional approach of `ioctl` and mapping device nodes into the file system.

2.4 Naming of device drivers

One of the major advantages of a single address space is that it provides a uniform way of naming entities in the system: simply use their virtual memory address. Device drivers can therefore be addressed by the memory location of the device driver instance data. This compares favourably to the approach used

[1] The unfortunate exception to this is the port-space registers in IA-32 machines.

in UNIX, for example, where devices are identified by a pair, consisting of a major number identifying the driver, and a minor number identifying the specific device. These numbers are then introduced in an *ad-hoc* way into the file system. By contrast, Mungi provides a user-level naming service, which can map arbitrary strings to 64-bit addresses.

2.5 Remote devices

It is sometimes the case that access is desired to a device that is physically connected to another computer on a network. In existing systems this is often solved with different network servers and clients for different devices; i.e. in an entirely *ad-hoc* manner. The underlying Mungi system provides a transparent mechanism for accessing components on remote nodes. This means that no special mechanisms need to be created to access devices residing on remote nodes.

2.6 Dynamic loading of device drivers

The device driver framework provides an ideal way of handling hot-pluggable hardware such as *universal serial bus* (USB) devices. The underlying component system allows driver instances to be created on demand. This is something that is not possible when drivers are compiled into a kernel. It is also preferable to the loading modules into the kernel at runtime, because it does not require a special linker; the driver executes as a normal process.

A particularly attractive feature is the possibility of replacing devices that other components are already communicating with, e.g. changing a user's windowing session to use different devices. It is possible to change the input and output devices on the fly, which would allow a user's session to be transparently (for the the applications, not the user) migrated to a different node.

2.7 Register description language

A common source of bugs in device drivers is incorrect bit manipulation during device register access. To help alleviate this problem, a device driver register description language was created [Les02]. This simple domain-specific language specifies device registers and bit fields within them. The language is compiled to produce C register-manipulation functions and bit fields, or Python language bindings, as described below. This is similar to the Devil [MRC$^+$00] device language.

2.8 Support for other languages

One of the key advantages of using a component system for the device driver framework is that it opens up the possibility of implementing device drivers in languages other than C. As a proof of concept, component bindings for the Python programming language — a language far removed from C — were created. In combination with the hardware register functions described above, these bindings allowed the implementation of device drivers in Python.

3. Experience

To date, a number of drivers for a number of different devices have been written using this framework, including 100MB Ethernet cards, gigabit network cards, IDE, serial, keyboard, VGA and USB. The frameworks runs on a number of platforms including MIPS, Alpha, Itanium and x86. Most of the platform has also been ported to the Linux operating system which supports using our drivers in kernel or at user-level.

This section describes the design and implementation of an IDE disk driver.

3.1 Block device interface

The block device interface is exported by any storage device driver. The main characteristics of a storage device, compared to network or character devices, is that data must be explicitly addressed and requested.

IBlock interface, shown in Figure 1, provides `read` and `write` methods for requesting data to be transferred to/from the storage device. The `data` parameter contains a list of blocks on the device to be accessed, and a destination or source memory address.

Both the `read` and `write` methods are asynchronous, which means that the method only queues the request and returns to the client. When the transfer is finished the driver must notify the client of the completion. The `notify` parameter provides a capability to the component method which should be invoked on completion.

```
interface IBlock {
    int write (cap_t notify, block_seq data);
    int read (cap_t notify, block_seq data);
    int get_block_count ();
    int get_block_size ();
};
```

Figure 1. Block device interface definition.

The get_block_count and get_block_size methods allow other components to determine the number and size of blocks respectively.

The Mungi operating system provides a transparent persistence mechanism, which means that most components do not need to access the disk (or file system) directly. The IBlock interface provides a low level procedural interface which is designed to be used by other low-level components such as the physical file system.

3.2 Performance

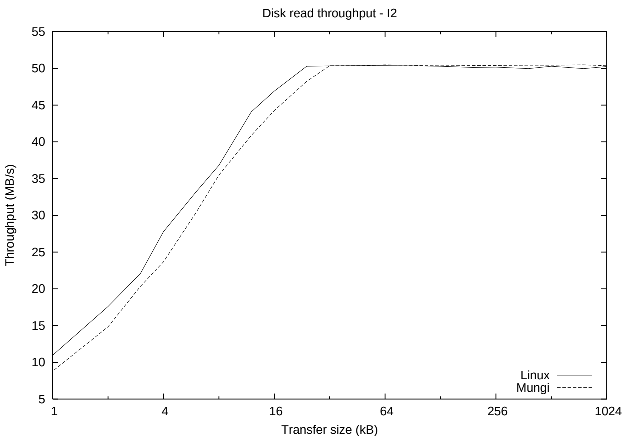

Figure 2. Disk read throughput

The IDE device driver runs successfully on a variety of different platforms. The results presented here show the performance on an HP rx2600 (Itanium 2) machine.

Figure 2 shows the throughput achieved for different request sizes. For large transfer size the throughput is quite similar for both systems, however as the transfer size decreases the context-switching overhead of the user-level drivers starts to have an effect, reducing

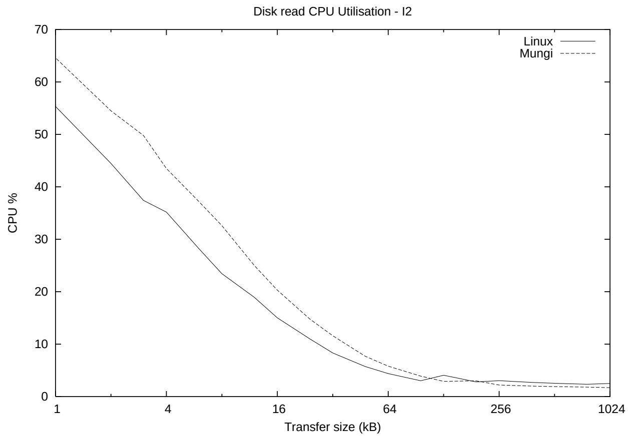

Figure 3. Disk read CPU load

throughput by up to 22 % (at extremely small transfer sizes).

Figure 3 shows CPU utilisation resulting from I/O processing (measured by instrumenting the idle loop). As is to be expected, Linux consumes less CPU for small transfers, a result of the context switching overhead of the user-level drivers. Mungi does however, perform, slightly better for larger transfer sizes which is probably the result of the cleaner drivers which might result in a smaller overall cache footprint. This indicates that a lean design might help to offset some of the inherent context-switching costs.

3.3 A device driver in Python

To demonstrate the potential of the system, an IDE driver was also implemented in the Python language, using the language bindings described in Section 2.8. While the implementation cannot get close to the efficiency of the corresponding C device driver (a topic for future work), it is identical in all other aspects: every function call on all exported MCS interfaces eventually results in the execution of Python code. Indeed, the resulting code is smaller and easier to comprehend than C, and thus has the potential to be a useful educational or prototyping tool.

4. Related Work

A number of previous systems have moved the network protocol stacks from kernel to user level while leaving the device driver in the kernel [TNML93, MB93, EM95]. Other approaches provided user code direct access to network interfaces in order to minimise latency for fine-grained communication in high-performance clusters [vEBBV95, Dam98].

Earlier work with real user-level drivers in Mach [GSR93] and Fluke [VM99] experienced significant performance problems, apparently resulting from the IPC costs in those kernels.

The *Palladium* approach of running Linux kernel extensions at an intermediate privilege level [CVP99] could, in principle, be used for device drivers without significant performance impact. While this approach could protect the kernel (to a degree) from buggy drivers, it would not protect applications, which still run at a lower privilege level. Pratt proposed an I/O device architecture that would allow the Nemesis system to run device drivers at user level [Pra97], but in the absence of devices conforming with this architecture, drivers are still in the kernel.

Work at the University of Washington [SBL03] encapsulates device drivers by introducing protection domains, called nooks, within the kernel's address space. This has the advantage of potentially fewer changes required to existing drivers. In spite of nooks being somewhat half-way between normal in-kernel and user-level drivers, the cost of nooks is significant — CPU load for sending network packets is almost double that of native Linux. This is almost certainly due to the more than doubling the interrupt latency [SMLE02]. The inherent overhead of nooks is essentially the same as that of user-level drivers, and it is therefore not clear what their advantage is over what we consider the cleaner approach of running drivers as proper user-level processes. In particular this approach does not protect against errors involving DMA.

The Devil [MRC+00] project takes another approach to improving reliability by reducing complexity. To reduce driver complexity they define a *domain specific language* which can specify the layout of a device's registers, and any constraints relating to the access of the device registers. By specifying constraints earlier the Devil compiler ensures that the programmer does not later violate these constraints. This is definitely a valuable tool, and provided the basic idea behind the register description language described in section 2.7.

5. Conclusion

This paper has presented the design and implementation of the Mungi device driver framework. In particular, the paper has presented the advantages of using a component model to implement device drivers at user-level. The main advantages are as follows.

Clearly-defined interfaces: A component model both provides clearly-defined interfaces for device classes, and ensures that those interfaces are adhered to, through the use of OS-level protection.

Fine-grained device access: Device drivers are treated exactly the same way as any other Mungi component, with the result that fine-grained access control can be achieved through the use of capabilities.

Support for other languages: With an appropriate component-system interface, it is possible to write device drivers in any programming language.

Protecting the OS: The operating system is completely protected from rogue drivers, because they execute outside the operating system and may only modify the state of the OS through well-defined interfaces.

6 Future Work

Future work on this project will involve comprehensive analysis of the performance results, in the disk driver described in Section 3 and also network drivers and network protocol stacks.

We will also be further investigating the use of high-level languages for device driver development to see if improvements can make such as approach feasible.

References

[APW86] Mark Anderson, Ronald Pose, and Chris S. Wallace. A password-capability system. *The Comp. J.*, 29:1–8, 1986.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *Trans. Comp. Syst.*, 12:271–307, 1994.

[CVP99] Tzi-cher Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. 17th SOSP*, pages 140–153, Kiawah Island, SC, USA, Dec 1999.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proc. 18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[Dam98] Stefanos N. Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks*. Phd thesis, Princeton University, 1998.

[EH01] Antony Edwards and Gernot Heiser. Components + Security = OS Extensibility. In *Proc. 6th ACSAC*, pages 27–34, Gold Coast, Australia, Jan 2001. IEEE CS Press.

[EM95] Aled Edwards and Steve Muir. Experiences implementing a high performance TCP in user-space. In *Proc. SIGCOMM*, 1995.

[GSR93] David B. Golub, Guy G. Sotomayor, Jr, and Freeman L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proc. USENIX Mach III Symp.*, pages 153–171, 1993.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw.: Pract. & Exp.*, 28(9):901–928, Jul 1998.

[Les02] Ben Leslie. Mungi device drivers. BE thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Nov 2002. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[LH03] Ben Leslie and Gernot Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2003.

[MB93] Chris Maeda and Brian N. Bershad. Protocol-service decomposition for high-performance networking. In *Proc. 14th SOSP*, pages 244–255, Asheville, NC, USA, Dec 1993.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proc. 4th OSDI*, 2000.

[Pra97] Ian A. Pratt. *The User-Safe Device I/O Architecture*. PhD thesis, King's College, University of Cambridge, Aug 1997.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. 19th SOSP*, The Sagamore, Bolton Landing (Lake George), New York, USA, Oct 2003.

[SMLE02] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *Proc. 10th SIGOPS European WS.*, pages 101–107, St Emilion, France, Sep 2002.

[TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Networking*, 1:554–565, 1993.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. 15th SOSP*, pages 40–53, Copper Mountain, CO, USA, Dec 1995.

[VM99] Kevin Thomas Van Maren. The Fluke device driver framework. Msc thesis, University of Utah, Dec 1999.