Nooks: An Architecture for Reliable Device Drivers *

Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195, USA

{mikesw, stevaroo, levy, eggers} @cs.washington.edu

1 Introduction

With the enormous growth in processor performance over the last decade, it is clear that reliability, rather than performance, is now the greatest challenge for computer systems research. This is particularly true in the context of Internet services that require 24x7 operation and home computers with no professional administration. While operating system products have matured and become more reliable, they are still the source of a significant number of failures. Furthermore, recent studies show that device drivers are frequently responsible for operating system failures. For example, a study at Stanford University found that Linux drivers have 3 to 7 times the bug frequency as the rest of the OS [4]. An analysis of product support calls for Windows 2000 showed that device drivers accounted for 27% of crashes, compared to 2% for the kernel itself [16].

The reasons for the high rate of driver failures are four-fold. First, drivers are typically written by device manufacturers rather than by operating system developers with extensive kernel programming experience. Second, drivers are frequently created by copying and editing code templates from existing drivers, often without complete understanding, leading to subtle bugs. Third, the kernel programming environment has many unenforced or poorly-documented conventions about synchronization and memory access, making kernel-mode programming and debugging challenging, at best. Finally, driver programming often requires understanding the operation of complex asynchronous devices, their control protocols, and their failure modes. As the number of new devices available increases to support new applications, such as cameras, digital video, etc., so does the number of drivers required and the number of (relatively unskilled) programmers responsible for creating them.

Device drivers can be viewed as a type of kernel extension, added after the fact. Commercial operating systems are typically extended by loading unsafe object code and linking it directly with the kernel. There have been many attempts to solve the general problem of safely extending the kernel [6, 2, 20], but they have demanded that programmers change the way in which they write code or the way in which operating systems are structured. Such approaches are unworkable for device drivers, which are the most common operating system extensions and represent a huge investment in development time; hence none of these approaches have been successful in a commercial system.

System availability depends not just on fault isolation, but also on quick recovery from faults. Therefore, the operating system must not just isolate faulty device drivers, but also allow them to quickly resume service, either after restarting or after recovering previous actions in progress. Recovery is also increasingly important due to the rising problem of hardware failures [16], which depends on isolating the effects of a fault and quickly recovering to a pre-fault state.

In the future, it is clear that improving operating system reliability depends on improving device driver reliability, because the kernel is no longer the primary source of bugs (or kernel-mode code!). In addition, as software matures and device integration levels shrink, hardware failures will become a greater problem. As a result, operating systems need to provide support for (1) tolerating and recovering from faulty drivers, and (2) tolerating and recovering from faulty hardware. The Nooks project is examining mechanisms and architectures to meet these goals.

*This work was supported in part by the National Science Foundation (grants ITR-0085670 and CCR-0121341).

2 Approaches

Many approaches have been proposed to safely execute user- or kernel-mode code, including device drivers. One difference between safely executing drivers and safely executing general kernel extensions is that one can assume that most device drivers are trustworthy: the problem is one of safety and not security, and absolute safety may not even be needed. Table 1 shows five key hardware and software techniques that can isolate driver code from the OS kernel. Each of these techniques has benefits and drawbacks, and may be appropriate in certain situations. Table 1 also shows the systems that used each technique.

Table 2 shows the relative advantages and disadvantages of each approach along the axes of software engineering, performance for large and small volumes of data, and ability to isolate memory corruption and deadlock errors.

In more detail:

1. Kernel wrapping surrounds all calls into and out of device drivers with special code, allowing resources to be tracked and pre- and post-conditions to be verified. Kernel wrapping can ensure that memory not owned by the driver is not freed, and that interrupts are enabled before blocking. However, kernel wrapping cannot prevent a driver from accidentally corrupting the operating system by writing through a stray pointer.

2. Virtual memory protection can be used to isolate data corruption problems, which are one of the most common driver faults, but can't catch deadlock errors, such as those caused by improper disabling of interrupts.

3. Lowering the privilege level of drivers (e.g., to supervisor or user level) prevents them from executing privileged instructions, accessing privileged address space, and corrupting the kernel. However, there is a large performance penalty, because calls into drivers require an additional trap and return to change privilege level.

4. Software fault isolation (SFI) [23] provides many of the benefits of a privilege level change, but is difficult to implement when the range of addresses accessible are not contiguous. In contrast to lowering the privilege level, it is very cheap to call into and out of SFI code, but SFI code executes more slowly. In addition, SFI does not easily support recovery in the form of copy-on-write, which hardware memory protection does support.

5. Finally, safe languages such as Java [7] and Modula-3 [17] can prevent drivers from uncontrolled access to kernel memory. However, using a safe language requires rewriting drivers and may introduce significant overhead when safely copying data in and out of the driver. There is also not as of yet a good mechanism for accessing a device in a type-safe fashion.

As a result, there is no single approach that is applicable for all device drivers. High performance devices, such as network and disk interfaces, require minimum overhead for large quantities of data to match the speed of the device and must run in the kernel. Low performance devices, though, such as keyboards, mice, and serial devices, do not always require the performance benefit of running in the kernel with no protection. The reliability needs of computer installations also vary: a bank may be willing to suffer a significant performance decrease in order to improve reliability, while a game player would risk crashing periodically to improve the realism of the game. Thus, it is important for the operating system to support a variety of techniques, so that driver writers and system installers can choose the most appropriate for them.

Beyond fault isolation, fault recovery is also increasingly important. Some techniques, such as memory protection, lend themselves toward automatic fault recovery. For example, Discount Checking [13] and Lightweight Recoverable Virtual Memory [19] use copy-on-write to maintain a shadow copy of all memory accessed by an application, allowing automatic recovery when a fault is detected. Other techniques, like safe languages and SFI, provide little support for recovery.

3 Nooks Architecture

We propose that operating systems should support executing drivers in a fault-isolating and recoverable environment so that a faulty driver cannot prevent the rest of the OS from functioning. In addition, the operating system should offer multiple levels of isolation and performance to reflect different driver needs. Our goal is to provide these features with only an incremental change to the operating system and device driver architecture, to maximize compatibility with the existing code base.

Figure 1 shows our proposed architecture, called Nooks. A *nook* is a protected environment for driver execution. Not all devices execute within a nook, as illustrated by the SCSI device driver in the figure. In addition, multiple drivers may execute within the same nook for performance reasons, as illustrated by

Table 1. Kernel extension safety approaches

Name	*Description*	*Where Used*
Kernel Wrapping	Verify all parameters on calls between the kernel and device drivers	Microsoft Driver Verifier [15]
Hardware Memory Protection	Prevent device drivers from writing to kernel memory	Palladium [3], Shinagawa [21]
Privilege Level Change	Prevent device drivers from executing privileged instructions and/or emulate privileged instructions	L4 [12], Exokernel [6]
Software Fault Isolation	Inject code into device drivers to ensure that addresses and instructions are safe	Vino [20]
Safe Languages	Rely on the compiler/virtual machine to allow only safe (non-faulting) drivers to be loaded	SPIN [2]

Table 2. Comparison of driver safety approaches

	Kernel Wrapping	*Hardware Memory Protection*	*Privilege Level Change*	*Software Fault Isolation*	*Safe Languages*
Requires rewriting driver	No	No	No	Maybe	Yes
Easily supports recovery	No	Yes	Yes	No	No
High performance for small data volumes	Yes	No	No	Yes	Yes
High performance for large data volumes	Yes	Yes	Yes	No	No
Isolates memory corruption	No	Yes	Yes	Maybe	Yes
Prevents most deadlocks	Maybe	No	Yes	Yes	Yes

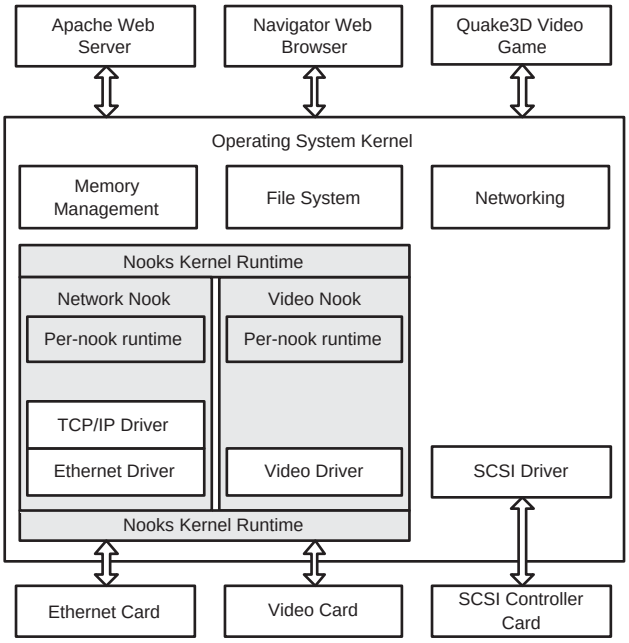

Figure 1. NOOKS **architecture diagram**

the combination of the Ethernet and TCP/IP drivers within the *network* nook. Nooks interpose between devices and device drivers by forwarding interrupts and, depending on the level of safety required, emulating access to memory-mapped device registers. Nooks also wrap calls from the operating system kernel into device drivers and from device drivers into the kernel, allowing the operating system to track resource usage and verify data that is passed into and out of the kernel. Rather than fully isolate device drivers in a separate address space, all drivers execute in the kernel address space, but within different protection domains. Thus, a device driver may use pointers supplied by the kernel without copying the data or translating addresses. However, the NOOKS architecture prevents device drivers from writing to memory outside their protection domain, limiting the damage of an errant memory access. Initially we use virtual memory protection and lowered privilege levels for isolating and recovering faulty code, but we plan to experiment with Software Fault Isolation (SFI) as well.

The NOOKS architecture minimizes the number of

crossings between the kernel protection domain and device drivers by separating kernel resources into those that must be shared from those that are only shared incidentally. For example, the processor must be shared among all drivers, so kernel intervention is required for scheduling functions. However, other operating system resources, such as wait queues and memory heaps, are only shared for convenience. NOOKS takes advantage of these two classes of resources to improve performance by duplicating the resources that are only incidentally shared. Drivers may then directly access the resource without crossing to the kernel's protection domain. In addition, some of the work that must be performed in the kernel need not be performed synchronously, such as delivering network packets. These operations can be deferred until the driver has completed execution, and then performed in a single batch.

Device drivers may require small changes to execute within the NOOKS architecture. In particular, operating system support routines that make kernel data structures directly available to device drivers (such as by returning a pointer to an kernel data structure which the driver then updates) cannot be supported. Instead, drivers must call wrapper routines that update kernel data structures on their behalf. The NOOKS architecture will initially support two levels of recovery: full restart, which unloads and restarts drivers, and rollback, which uses recoverable virtual memory to maintain a shadow copy of driver state, allowing it to be recovered after a fault. Many device requests are by their nature idempotent, such as sending or receiving a network packet or writing disk block, so in some cases a failed operation can be retried safely.

4 Implementation

We implemented a prototype of the NOOKS architecture in the Linux kernel. The prototype runs on Linux version 2.4.10, and we have experimented with isolating network interface device drivers, including the 3Com 3c905 fast ethernet adapter and the Intel Pro/1000 gigabit ethernet adapter. The prototype follows the architecture shown in Figure 1, and wraps all calls into and out of device drivers to prevent write-sharing of data and to verify parameters. We plan on using hardware memory protection to isolate drivers, and we therefore maintain a copy of the kernel pagetable for drivers that only grants read access to kernel text and data.

While our implementation does not execute drivers in a separate protection domain, it does emulate the cost of switching protection domains. First, to emulate the cost of changing page tables, we flush the TLB on all calls into or out of a driver. Second, to emulate the cost of lowering the privilege level of a driver, we execute a software trap and return both when entering and leaving a driver as well as when the driver executes a privileged operation, such as disabling interrupts.

Our prototype NOOKS implementation wraps 147 calls made by device drivers into the Linux kernel, and 103 calls from the kernel into device drivers through ten different interfaces. To ensure that drivers do not call directly into kernel functions, we modified the *insmod* program, which is responsible for binding symbols in dynamically loaded kernel code, to bind symbols imported by these drivers to the NOOKS wrappers rather than to the kernel functions.

The Linux operating system and the Intel IA-32 architecture proved to be difficult choices for implementing NOOKS. First, Linux allows drivers to modify many kernel data structures. For example, drivers typically update a kernel queue when freeing network packets. In addition, many kernel functions are implemented as inline function calls. We converted several functions to true procedure calls, which requires that drivers be recompiled to execute within our prototype. The Intel IA-32 architecture proved difficult because of its hardware TLB miss handler. The architecture enforces a page table layout that does not support multiple protections for the same page at the same privilege level. As a result, we instead duplicated the kernel page table and copied all updates from the kernel page table.

The NOOKS wrapper code copies all parameters passed from drivers into the kernel, and also verifies that pointers reference data structures allocated to the driver making the call. In addition, the wrapper layer maintains a mapping between kernel data structures and copies of the data structure used by the driver, and ensures that the data structures are synchronized by copying changes during calls into the driver or kernel.

We emulate different levels of protection, ranging from just wrapping calls to executing drivers at a lower privilege level by controlling the cost of protection domain switches and privileged operations. We are therefore able to determine the impact of different protection techniques on specific device drivers and workloads.

5 Performance

To test the performance of our prototype, we used the Netperf benchmarking tool [9] to measure TCP bandwidth and UDP request/response performance. The bandwidth tests stream 32KB messages between two machines, and the UDP tests send a 100 byte request and receive a 200 byte response. Our experimental

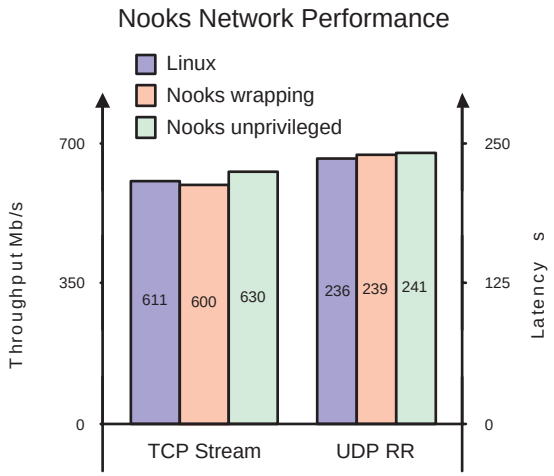

Figure 2. NOOKS **Netperf performance compared against unomdified Linux**

platform was a pair of PCs with 1.7 GHz Pentium 4 processors, 1 GB of memory, and gigabit Ethernet adapters, one of which ran drivers isolated with NOOKS. We tested Linux with the driver loaded as a module, and NOOKS with just wrapping and with full protection, which entails both flushing the TLB and executing a software trap on all calls into and out of the driver and on privileged instructions. In all cases we used the default values for the driver's parameters.

The results are shown in Figure 2, and demonstrate that isolating a driver using NOOKS has a negligible impact on network performance. Despite the additional overheads from flushing the TLB and executing a trap, the bandwidth of TCP streams actually improved, from 611 Mb/s to 630 Mb/s. We believe the performance increase is due to the increased number of packets received during each driver interrupt, which increased from 7.1 to 8.6 packets per interrupt. The UDP request/response test meeasures measures the round-trip time across the network and through the network stacks on the two machines. The latency increased slightly, from 236 microseconds to 241 microseconds, demonstrating the low overhead introduced by NOOKS.

We measured the impact of isolating the network driver on interrupt handling by measuring the number of cycles spent handling interrupts from the network interface. On Linux, the Intel PRO/1000 driver executes in 20,800 cycles on average. With just wrapping, handling an interrupt takes nearly twice as long, 37,200 cycles. Raising the privilege level and changing page tables raises that cost to 47,800 cycles, about 15 microseconds longer than plain Linux. We also measured the load incurred by processing the TCP tream. On unmodified Linux, processing TCP required 17.6% of the CPU, while running the driver isolated with both wrapping and a lowered privilege level required 20.7% of the CPU, an 18% increase.

Overall, these experiments demonstrate that on modern processors the cost of isolating device drivers is low. Furthermore, other architectures with faster operating system operations, such as the Alpha [11] or Itanium [5], could further reduce these overheads.

6 Related Work

Many projects have tackled the difficulty of writing device drivers. The Stanford study using the MC tool [4] and Microsoft's SLAM project [1] mechanically found many bugs in device drivers, but did not automatically fix those bugs. Microsoft's Driver Verifier [15] wraps operating system calls, but is meant as a debugging tool only, and does not prevent memory corruption. The Devil Project [14] aims to simplify the process of writing device drivers by providing a domain-specific language for specifying the interface between the device and the processor, which could be used as part of NOOKS to better isolate driver's hardware access. The WinDriver architecture [10] and Hunt's user-mode drivers [8] both allow device drivers to be run in user-mode, but support a different API from the kernel and don't provide the performance option of executing in the kernel but with memory protection. Van Maren [22] built a user-mode device interface for the Fluke microkernel, but it was not applicable to conventional operating systems. Finally, the Uniform Driver Interface project (UDI) [18] provides a driver interface to the kernel that prevents deadlock and allows for memory isolation, but again requires that drivers be completely rewritten.

7 Conclusion

Reliable services depend on a reliable operating system. While the core operating system kernel code has become reliable, device drivers have not kept pace. Device drivers are by their very nature difficult to write and even worse, are not written by experts at kernel programming. Therefore, operating systems must assist device driver writers by reducing the penalty of a faulty driver. The NOOKS architecture accomplishes this goal by isolating drivers with a variety of techniques, including kernel wrapping, virtual memory protection, privilege level lowering, and software fault isolation, which are suited to many different environments

and driver types. NOOKS does not require rewriting drivers, and can support recovery from both software and transient hardware errors using copy-on-write virtual memory techniques to maintain a shadow copy of uncorrupted memory. Executing drivers within a nook isolate the two most common driver bugs, memory corruption and deadlock, leading to more reliable systems. Finally, the NOOKS architecture can achieve high performance for large volumes of data using virtual memory remapping, while also maintaining performance for low-bandwidth devices with software fault isolation.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th POPL*, Portland, OR, Jan. 2002.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th SOSP*, pages 267–284, Copper Mountain, Colorado, Dec. 1995.

[3] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. 17th SOSP*, pages 140–153, Kiawah Island Resort, South Carolina, Dec. 1999.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empiracle study of operating system errors. In *Proc. 18th SOSP*, Lake Louise, Alberta, Oct. 2001.

[5] I. Corporation. *The IA-64 Architecture Software Developer 's Manual*. Intel Corporation, Jan. 2000.

[6] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th SOSP*, pages 251–266, Copper Mountain Resort, Colorado, Dec. 1995.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[8] G. Hunt. Creating user-mode device drivers with a proxy. In *Proc. 1997 USENIX Windows NT Workshop*, Seattle, WA, Aug. 1997.

[9] R. Jones. Netperf: A network performance, version 2.1, 1995. Available at `http://www.netperf.org`.

[10] Jungo. Windriver cross platform device driver development environment. Technical report, Jungo Corporation, Feb. 2002. `http://www.jungo.com/windriver.html`.

[11] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[12] J. Liedtke. On μ-kernel construction. In *Proc. 15th SOSP*, pages 237–250, Copper Mountain Resort, Colorado, Dec. 1995.

[13] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, Nov. 1998.

[14] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. 4th OSDI*, pages 17–30, San Diego, CA, Oct. 2000.

[15] Microsoft. Windows XP device driver development kit. Technical report, Microsoft Corporation, Oct. 2001.

[16] B. Murphy. Fault tolerance in this high availbility world. Talk given at Stanford University and University of California at Berkeley. Available at `http://research.microsoft.com/users/bmurphy/FaultTolerance.htm`, Oct. 2000.

[17] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[18] Project-UDI. Introduction to UDI version 1.0. Technical report, Project UDI, Aug. 1999. Available at `http://www.project-udi.org/Docs/pdf/UDI_tech_white_paper.pdf`.

[19] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *Proc. 14th SOSP*, pages 146–160, Asheville, North Carolina, Dec. 1993.

[20] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd OSDI*, pages 213–227, Seattle, Washington, Oct. 1996.

[21] T. Shinagawa, K. Kono, and T. Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Dept. of Information Science, University of Tokyo, May 2000.

[22] K. T. Van Maren. The fluke device driver framework. Master's thesis, Department of Computer Science, University of Utah, Dec. 1999.

[23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th SOSP*, pages 203–216, Asheville, North Carolina, Dec. 1993.