Nooks for NT

by

Micah Z. Brodsky
&
Eric K. Kochhar

A senior thesis submitted in partial fulfillment of

the requirements of the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

March 2006

Presentation of work given on May 13, 2005

Thesis and presentation approved by _____
Professor Henry M. Levy

Date _____

University of Washington

Abstract

Nooks for NT

Micah Z. Brodsky
&
Eric K. Kochhar

Advisor:

Professor Henry M. Levy
Computer Science & Engineering

Today's computer users can expect their computers to fail frequently, forcing restarts, loss of data, and loss of time. On commodity systems such as Windows, 85% of failures are caused by bugs in device drivers [Swift05]. Device drivers are privileged software modules that control hardware devices, such as disks and audio cards, and are extremely complex and difficult to build. Unfortunately, that complex code is often written by inexperienced programmers at device companies, rather than by experienced kernel programmers in the company that wrote the operating system.

The goal of Nooks for NT is to demonstrate the feasibility of an isolation system to greatly reduce the number of Windows system failures by inserting a new protection layer between device drivers and the core of the operating system kernel. We create a new environment for driver execution, using memory isolation to ensure that driver bugs do not corrupt the rest of the system. Once Nooks detects a driver failure, it unloads the failed driver and reloads a working version of the driver without any user intervention. Nooks for NT is based upon the original version of Nooks developed for Linux by Mike Swift in his thesis work at the University of Washington. We reimplemented the architecture in Windows 2000, which has a much more sophisticated kernel environment and a much more complex driver and memory management model. This brought to light some important lessons about the interactions between reliability and complexity and the prospects for bringing backward-compatible reliability subsystems to the mass market.

In this thesis, we begin by explaining the concepts and architecture behind Nooks. We then discuss the differences between kernel development in Linux versus Windows, with a focus on driver related issues. Next, we cover our partial implementation of Nooks in Windows 2000, focusing on our design choices, the notable challenges we encountered, what we did and did not complete, possible future improvements, and a brief evaluation of our results. We conclude with some ideas for expanding the scope of Nooks and possible paths for future work.

Table of Contents

Acknowledgements iii

1. Introduction 1
1.1. The Problem 1
2. The Nooks Approach 2
2.1. Architecture 2
2.1.1. Protection Domains 2
2.1.2. Wrappers 3
2.1.3. Resource Tracking 3
2.1.4. Error Detection 3
2.1.5. Clean-Up and Recovery 4
2.2. Discussion 4
2.2.1. Why Best Effort Only 4
2.2.3. Why LPDs? 5
3. Operating System Environments Background 5
3.1. Linux 5
3.2. Windows 6
3.2.1. Programming Style 7
3.2.2. Windows Driver Model 7
4. Nooks for NT 8
4.1. Goals 8
4.2. The Design Under Windows 9
4.2.1. Binary Compatibility 9
4.2.1.1. NT Binary Ecology 9
4.2.1.2. The Nooks Loader 9
4.2.1.3. Macros and Object Tracking 10
4.2.1.4. Challenges 11
4.2.2. Libraries and Driver-Driver Interactions 11
4.2.2.1. NT IO Management 11
4.2.2.2. Boundary of Isolation 13
4.2.2.2.1. Introduction 13
4.2.2.2.2. The Kernel / Driver Boundary 13
4.2.2.2.3. The Driver / Driver Boundaries 14
4.2.2.2.3.1. Optimization 15
4.2.2.3. Reference Counting and Crash Recovery 16
4.2.3. Memory Management 16
4.2.3.1. NT Memory Management 16
4.2.3.2. Nooks for NT Memory Management Implementation 17
4.2.3.3. Handling User-Space Access 18
4.2.3.4. Memory Management: Picking The Right Approach 19
4.2.3.5. Out of Memory Corner Cases 21
4.2.4. Asynchrony 23
4.2.4.1. NT Asynchrony 23

4.2.4.2. Synchronization Interoperability & Asynchrony Safety 23
4.2.4.3. Asynchrony in the Driver Model 24
4.2.4.4. Leveraging Asynchrony to Reduce Domain Transitions 24
4.2.4.5. Kernel Preemptability and SMP Support 24
4.3. Evaluation and Limitations 25
5. Expanding the Scope: Flyweight Isolation 26
5.1. Introduction 26
5.2. Justification 26
5.3. Supporting Data 27
5.4. Conclusions 29
6. Future Work 29
6.1. Nooks Performance 29
6.1.1. Breakout Access 29
6.1.2. Memory Reservation Service 30
6.1.3. Video Drivers 30
6.2. Quality and Generality of Isolation 30
6.3. Integrity Beyond Domains 31
7. Conclusion 32

Bibliography 34

Acknowledgements

This project would never have been possible without the support, advice, and time of a large group of people. Professor Hank Levy taught the operating systems course that got us started in the first place and served as our faculty advisor throughout the project, keeping us honest about our progress and deadlines. We are deeply in debt to him for all his assistance and especially his patience with us completing this thesis!

Without Mike Swift there would be no Nooks. But, he provided more than just a research topic; his support, careful guidance, and genuine interest in our work were invaluable. Mike constantly challenged us when things were going well and provided inspiration during difficult stretches. Over the years we worked with him he became more than just an advisor, he became our friend.

We would also like to thank Gary Kimura, who lead the Operating Systems Capstone course, offering us his many years of experience and the Windows source code. Without this, Nooks for NT would have been impossible. Gary was always available to help us, and when he couldn't answer our questions, he found people who could. He also brought in several of his old colleagues from Microsoft to speak to us, which proved extremely helpful to our project. We would like to thank Landy Wang in particular, for his excellent suggestions and his additional assistance by email.

We would like to thank Jordan Hom, Jai Patel, and James Crompton for working with us during the early stages of the project and for their contributions to object tracking, recovery, wrapper generation, and debugging.

We also would like to thank our parents for the lifelong support and guidance they have provided us. Without them we would never have been as successful as we have, and none of this would have been possible.

Eric would also like to thank Melanie Madden for putting up with him working on this project on weekdays and every Saturday in addition to all of his undergraduate classes. She was always extremely supportive despite never getting to see him and helped motivate him to complete this thesis.

1. Introduction

1.1. The Problem

Today's computers fail far too often. These failures result in interruptions, loss of data, and loss of time. Computer crashes cost over 250 billion dollars in damages each year for small businesses alone, according to the London Business School's 2002 estimates [Butler02]. The high costs of crashes, and the resulting frustration of computer users, is one of the main shortcomings of today's computer technology.

These days, far more problems are attributable to software than hardware [Gray86] [Chou97]. A large portion of these software failures result from third-party extensions added to the operating system, such as device drivers. User applications running on a system are isolated from each others' faults, and the kernel is isolated from application failures, but in commodity operating systems, no such protection is afforded to kernel extensions. Device drivers, because they often require close coupling with the kernel and with low-level hardware resources, are typically implemented as kernel extensions rather than user-mode components. The resulting high privilege of device drivers means that a failure or a crash in a device driver typically brings down the whole system. Thus, the reliability of a system's device drivers has a critical impact on the reliability of the system as a whole.

Unfortunately, a powerful array of causes conspires against high standards for device driver quality. As low-level systems code, drivers present an extreme technical challenge to develop, especially for highly sophisticated kernels like NT. Besides simply supporting their hardware, drivers must deal with the complexity of the operating system environment and interface, with concurrency and asynchronous requests, with multiprocessor platforms, with power management, and with dynamic hardware insertion, configuration, and removal ("Plug and Play"). Furthermore, drivers are largely developed by third-party hardware manufacturers who often have little interest in software and who lack the kind of in-house expertise available to the system vendor. For many of these hardware companies, driver development is simply an afterthought. Such hardware manufacturers are often not held responsible for writing reliable code, because the system vendor typically gets blamed for any problems that appear. Finally, driver code has a much shorter lifecycle than kernel code, as hardware is replaced every few years, but kernels last for decades or more. The net result is that driver code tends to be of far lower quality than kernel code [Chou01], with a greatly disproportionate impact on system stability.

Indeed, in Windows XP, 85% of total system failures are caused by device drivers [Swift05]. As the core components of commodity operating systems have become more and more reliable over the years, the increasing significance of kernel extensions for system unreliability has come to the attention of Microsoft and the research community. Traditionally, the research community has explored safe kernel extension through radical new approaches to system design [Bershad95] [Engler95] [Seltzer96], without addressing the question of how to make kernel extensions safe in existing commodity systems. Vendors of commodity systems, for the most part, have been hesitant to implement such

radical, all-or-nothing solutions. Indeed, with such a broad market, not all Windows customers are likely to be interested in trading performance for reliability. Compatibility constraints would also demand that the system continue supporting existing, unsafe driver models. Furthermore, the Windows team's experience has shown that hardware companies are rarely willing to take extra steps to assure the reliability of their drivers, especially when it comes at the expense of their devices' performance [Wang04], and hence they would prefer to continue using their existing code bases and the old driver models. Radical solutions would thus be simultaneously expensive to implement and unlikely to gain any foothold. With more than 35,000 Windows drivers on the market [Swift05], there's little hope of fixing them all.

With no reasonable means to fix so many existing drivers and with driver bugs causing so many crashes, any attempt to improve system reliability must do so in spite of buggy drivers. Any attempt via improving driver reliability must be able to handle the numerous drivers on the market today and still protect the system from driver failures. In order to achieve these goals, such a reliability system must be transparent to existing drivers. In order to be practical, it must be also efficient, and for the most part, transparent to the rest of the kernel. Only then does it have a chance of truly impacting commodity operating systems.

2. The Nooks Approach

2.1. Architecture

Nooks is a driver-reliability subsystem created to improve system reliability while satisfying the goals listed above. It is invisible to drivers, largely invisible to the normal operation of the kernel, and does not impose an enormous performance cost, at the same time greatly increasing system reliability. Nooks improves driver-reliability by isolating drivers from the rest of the kernel. This allows each driver to be monitored carefully, tracking all its interactions with the rest of the system. If a driver fails, Nooks is able to tear down and remove the isolated driver and then restart a new instance so that the system is able to continue running without a reboot. On an unprotected system, such a failure would propagate into the kernel, causing a system crash. The Nooks architecture, designed to achieve all of these goals, is made up of five main components: protection domains, wrappers, resource tracking, error detection, and clean-up and recovery.

2.1.1. Protection Domains

A protection domain consists of a logical container for one or more drivers with a single shared fate. Hardware memory protection and well-defined cross-domain calling semantics define the boundaries of the domain, so that a fault in one driver may crash them all, but failures are contained within the domain. Thus, conceptually, a protection domain can be thought of as a barrier surrounding a group of drivers or a "driver sub-process". Although the drivers run under the same address space as the kernel, their access to the address space is limited by the protection domain's memory management code so that they can only write to addresses they specifically need access to. Placing such a barrier around drivers allows us to see all control and data transfers in and out of

the drivers. Combining protection domains with wrappers, we can track calls into the driver, the system resources it allocates, the system resources it has access to, etc., and any actions it may take that could propagate a fault into the kernel. Thus, protection domains are the most fundamental part of Nooks, upon which all the other abstractions and components are built.

2.1.2. Wrappers

Protection domains alone, however, are not particularly useful, because their strong isolation breaks compatibility with existing interfaces between kernel and driver. To allow drivers to seamlessly operate within protection domains, we provide interface "wrappers", adapter functions matching the original driver-kernel interfaces on either end but explicitly crossing the protection domain boundary in the middle. Besides serving as gateways, wrappers also provide crucial points for monitoring driver-kernel interactions, because they are the sole means of control transfer in and out of a domain.

Two conceptually similar classes of wrappers exist, one for kernel to driver calls and one for driver to kernel calls. The two classes require different mechanisms to attach to their respective interfaces, however. While installing kernel to driver wrappers is often a matter of making simple changes to the associated private kernel source code, modifying third party drivers to call out only through wrappers is somewhat more complicated. This is accomplished using a special binary loader which remaps statically imported kernel function calls to their respective wrapper routines. Using these statically interposed wrappers, the remaining cases, function pointers provided at run time, can be handled dynamically by substituting them with dynamically generated stubs.

2.1.3. Resource Tracking

Centrally tracking the driver's resource usage serves as an important infrastructure subsystem within the Nooks architecture. While a driver runs, it creates, acquires, and modifies system resources; resource tracking (or "object tracking") uses code invoked by the wrappers to maintain a comprehensive table of all such resources used by the driver and any interrelationships among them. Since errors in a driver can cause it to misuse or corrupt system resources, risking a crash, the object tracker is used to duplicate and synchronize mutable objects, such that the kernel never sees the driver's changes to an object until they've been verified to be safe. This same verification is also used to check invariants about driver's behavior, for example, preventing double-frees, so that a fault can be raised and the driver removed and restarted. Finally, when a driver fails, any resources it owned need to be released. The object tracker maintains all the information necessary to safely free shared resources one-by-one after a driver crash. It is important to note that the object tracker itself is a rather unintelligent service, however; it serves only as a warehouse of information about a protection domain. Higher-level services, like error detection and recovery, use the object tracker as a primitive tool.

2.1.4. Error Detection

Error detection is a crucial, high-level service provided by the system as a whole, based upon checks distributed throughout the wrappers. Using the tools provided by wrappers, resource tracking, and memory protection, error detection is responsible for catching any driver failures and triggering the recovery process before system corruption can occur (and, ideally, before the user is inconvenienced by a major service lapse). Error detection in Nooks is something of an approximate process, because of its best-effort design and because the interfaces isolated are rather complicated and ill-specified. There are roughly six classes of errors Nooks can detect: non-continuable processor exceptions (including memory protection violations), resource invariant violations, invalid parameters, corrupted objects, resource consumption limits, and broken service guarantees. The first two are relatively straightforward to implement comprehensively, but the others are more ill-defined and are implemented approximately or on an as-needed basis. Error detection will never be perfect, but given enough effort, it can be improved to cope with an arbitrarily wide space of errors and a broad variety of drivers.

2.1.5. Clean-Up and Recovery

Once a driver fault is detected, Nooks begins a rather elaborate procedure for safely unloading and reloading the driver. Conceptually, the key elements of this process are getting all threads of execution safely out of the driver, synchronizing, disentangling the driver from the rest of the system, freeing up the driver's resources, and triggering an appropriate unload/reload sequence. Much of this process revolves around repeatedly walking through the object tracker, releasing resources and disconnecting relationships, a process we casually refer to as "garbage collection". The details are complicated, nasty, and highly system-dependent, but for the most part, they follow the same basic recipe.

2.2. Discussion

2.2.1. Why Best Effort Only

Nooks was designed around two core principles: be resistant to faults, not impervious to them, and be robust against mistakes, not malicious abuse. Nooks is not intended to handle all imaginable failures; it is a best-effort-only system that tries to protect against the common case and thereby make a practical improvement in system reliability. Success is gauged not in terms of theoretical correctness-by-construction, but rather, as a measurable improvement in the end-user's experience. Handling all possible failure cases would be an irrelevant and impossible goal, and trying to do so would likely impose unreasonable performance degradation in a system designed for backward compatibility.

Nooks also does not attempt to protect against deliberate abuse. Protecting against abuse requires a much more complex and involved isolation system than one just protecting against bugs, and trying to protect against malicious kernel extensions at this level makes little sense. Few unprivileged users have any need to install kernel extensions. As the history of Windows applications shows, few application developers are interested in the extreme effort associated with building a kernel extension. Most legitimate kernel extensions are drivers, virus scanners, server modules, and other privileged or administrative services tightly coupled with the operating system. Furthermore, it is

virtually impossible to prevent such extensions from abusing the system once they are loaded; a network driver can always send spam and forward worms; a keyboard driver can always sniff passwords. Instead of imposing draconian restrictions on legitimate kernel modules, it makes much more sense to secure the mechanism used to load kernel modules and prevent abusive modules from being installed in the first place. Compared to restricting abuse once modules are loaded, this is an easy problem.

2.2.3. Why LPDs?

Besides the above two core principles around which Nooks was designed, another key design goal was to make the system adoptable and attractive for existing commodity operating systems. In particular, this means easy integration into existing OS code bases, support for existing drivers without modifications, and low runtime overhead. In order to meet these constraints, Nooks uses a novel approach based around lightweight protection domains. While many other research architectures for safe kernel extensibility have been explored, none of them meet all the constraints to the extent that LPDs can. The Software Fault Isolation (SWFI) memory isolation mechanism [Wahbe93] has been used successfully as the basis for safe kernel extensions [Seltzer96], but typical implementations have depended upon special compilers and were unable to isolate arbitrary existing binaries. Also, SWFI alone does not address the problems of safe, transparent integration and recovery, which requires either re-architecting the driver interfaces or implementing a LPD-like structure based on SWFI (which may well be possible). Type-safe languages similarly provide a memory isolation mechanism, as well as some level of interface safety, but they require completely re-architected interfaces and support for an entirely new programming model in the kernel [Bershad95]. User-mode drivers have also been used in some systems [Herder06] [Hunt97], but they are exceedingly difficult to retrofit into an existing operating system due to deadlock and performance considerations, to say nothing of backward compatibility [Microsoft05_2]. The most viable alternative to the LPD is perhaps the virtual machine. Whole-system VMs partitioning the user applications are not useful, because they inappropriately tie together the fates of drivers and applications, but per-driver VMs are a reasonable possibility [Erlingsson05] [LeVasseur04] [Fraser04]. However, VMs are rather costly structures to use, and they offer surprisingly few advantages as well as some new compatibility hazards, the drivers running in the environment of the wrong kernel. Unlike these alternatives, Nooks and its LPDs do not require any major shift in operating system or driver development, allow for easy integration with current platforms, are fairly lightweight, and properly handle driver recovery.

3. Operating System Environments Background

3.1. Linux

The Linux 2.4.18 kernel, under which Nooks was originally developed, is a far simpler and more simplistic system than NT. Little time went into original system engineering, since the kernel was developed from what was began as a toy system. Linux has since grown in large part by accretion as new demands and new markets appeared. The 2.4 kernel environment is relatively simple and unsophisticated compared to systems like

NT. Although SMP is supported, the kernel is non-preemptable, and many operations still hide behind a single master lock. Kernel code and data is not pageable, and most of the system memory space is simply direct-mapped.

Device driver interfaces are similarly simplistic and accreted. Each driver interface is designed as a domain-specific plug-in specification for the kernel. The driver interface protocols and object models are much less complicated than a powerful, centralized interface like NT's, though the individual, unique models are far more numerous. Support for power management and Plug and Play is limited and is largely implemented in user-mode in an ad-hoc manner. This simplicity may have a silver lining, however, in that it may make driver development easier and hence potentially result in more reliable drivers.[1]

Another distinct feature of Linux is its open-source development model. Unlike Windows, which operates on a binary-only economy, the Linux kernel lacks a standardized binary extension interface, and so the kernel and practically all its extensions are distributed in source form. Source code for the kernel and drivers is thus readily available for anyone to study from or modify, facilitating both learning by example and hack-like constructions. Leveraging this source-only environment, the kernel also relies heavily on compile-time configuration in lieu of a run-time configuration store like the Windows registry. An incredible variety of configuration and customization choices are available at compile time, often effected through preprocessor source code transformations, but comparatively few options can be set given a pre-existing binary, and almost none can be set at run-time.

From a research perspective, the net result is that Linux 2.4 is friendlier to budding research projects but less representative of a modern operating system. With its simplified, partitioned architecture, Linux 2.4 makes it easier to focus on the core concepts and feasibility assessment of a new research system, without succumbing to the greater challenges and overwhelming details necessary for a mass-market production implementation. However, the same simplicity also limits the efficiency, power, and growth potential of the operating system. Because of this, the Linux system architects have been pushing more and more in the direction of an NT-like design in recent years. The most recent version, 2.6, incorporates a number of elements characteristic of NT, such as kernel preemptability and a unified driver object model. While still comparatively simplistic, Linux remains a rapidly moving target.

3.2. Windows

Windows, based on the NT kernel, is a very feature rich and complicated operating system, providing a significantly more complex programming environment than Linux. Unlike Linux, which was a reimplementation of the UNIX tradition, NT was designed in the spirit of the Digital VMS operating system. NT was designed from the beginning with high goals, to be robust, secure, efficient, and concurrent, to be a server, a client, and much more. The combination of these goals led to a solid but complicated system, with

[1] For an interesting counterpoint, see [Semack04].

strong binary compatibility, kernel code paging and preemption, and multiprocessing and asynchrony all playing a prominent role in the operating system.

3.2.1. Programming Style

Windows, like its contemporaries, was built largely without object-oriented or type-safe programming languages. Nonetheless, it was built using a strong foundation of modularity and abstractions with well-defined binary interfaces. Besides being good software engineering practice, much of this work was done to support the proprietary binary ecology commercial operating systems are generally intended for. Being a proprietary operating system, driver writers would have limited knowledge about the internal implementations of the system, and given proprietary drivers, Microsoft would have little opportunity to manage or maintain the drivers' implementations. Therefore, Microsoft had to provide a coherent, well-specified interface that provides everything driver writers need.

This black-box approach has its good sides as well as its bad. Driver writers have limited knowledge of the system and are left at the mercy of the interface designers, which can make their job harder, but it also limits their ability to misuse implementation details of the operating system to build their drivers. This, in theory, prevents drivers from using operating system features they are not supposed to use and breaking when those behaviors change as the system evolves. Unfortunately, many driver developers still discover and exploit aspects of the operating system that they shouldn't, simultaneously making their drivers fragile and impeding the evolution of the system. Thus, it seems to be an open question whether full access to source or access only to interface specifications leads to better drivers in the antagonistic third-party driver development model.

Two additional key strategies Microsoft used to enable the system's binary-only ecology, as well as to improve the system's overall modularity, were the use of explicit binary modularity and the exposure of heavyweight infrastructure services such as the object manager, the security reference monitor, and the registry. Binary modularity, where the kernel itself and its suite of core drivers are divided up into multiple loadable modules, means that a single suite of binaries covers most every realizable configuration, and installing or upgrading drivers and services never requires recompilation. Similarly, the registry provides a centralized, structured store for configuration information, available even as the system is bootstrapping itself, which means recompilation is never necessary for reconfiguration either. This stands in stark contrast to the UNIX world, where recompilation is a regular event, and driver source is provided not because it can be but because it must be.

3.2.2. Windows Driver Model

The main interface Microsoft exposes to driver writers is known as the Windows Driver Model (WDM) [Oney02]. WDM is the primary driver model for most classes of hardware. Based on the ancestral driver model of NT, it closely parallels the internal

structure of the IO manager. The core idea of WDM is an asynchronous messaging model based around "IO Request Packets", where requests generated on behalf of users are satisfied by propagating these packets through a stack of handler drivers. This model allows fast, efficient handling of both synchronous and asynchronous IO but also provides the basic foundation for an extensible binary driver interface. In particular, the layered structure encourages separation of concerns and supports extensibility via packet filters, both crucial for a viable binary ecology. The messaging model also serves as the basis for the higher level functionality provided by the WDM infrastructure, such as device discovery, configuration, and power management.

WDM is a complicated system, however, and the interactions between threading and asynchrony, plug and play and power management, serve to make driver development a rather tricky task. Threading and asynchrony create problems because drivers must internally queue and later process asynchronous events delivered concurrently in multiple threads, all the while being prepared for request cancellation and device state changes. This makes for a rather painful exercise in concurrent programming [Oney02]. Plug and Play and power management also create their own set of problems, because each are presented as two conceptually separate state machines, yet whether the device is configured and providing service to the system, Plug and Play concept, is inherently coupled to the physical power state of the device, a power management concept. The net result is that Plug and Play and power management, when implemented together, form a notoriously confusing hybrid API [Oshins04].

These services provided by WDM form an indispensable part of the system, helping to make Windows hardware support efficient and relatively seamless, but the APIs and programming environment it presents makes driver development unduly challenging and error prone. Microsoft is well aware of these issues and has put a lot of effort into mitigating them, in the form of better abstractions, a sophisticated test harness [DV05], static analysis [PREfast03] and model checking tools [SDV06], and most recently, an entirely new driver development interface known as Windows Driver Foundation (WDF) [Microsoft06]. WDF is essentially a wrapper around the core WDM interface, providing a much friendlier suite of abstractions and turnkey solutions for many of the tricky problems. Not only is WDF generally expected to significantly decrease the number of WDM-related bugs, the relative simplicity and clean, encapsulated design of its interface will likely make Nooks-style transparent isolation noticeably easier. Nonetheless, the Windows driver interface remains a complex beast and a significant source of driver bugs, above and beyond what one would expect with more simplistic driver interfaces such as Linux's. Unfortunately, to support the majority of the market and the features it demands, this is the kind of complexity Nooks must handle.

4. Nooks for NT

4.1. Goals

The original Nooks project proved very successful under the simpler environment furnished by Linux, but we wished to demonstrate that it would be effective not only there but on commodity systems in general. Windows, given its ubiquity, complexity,

and deep dissimilarity from Linux, was the obvious choice. The goal of the Nooks for NT project was thus to re-implement under Windows the architecture embodied in Nooks for Linux and to understand and document any new challenges and design questions that arose. A successful implementation under Windows would demonstrate soundly that the principles behind Nooks hold across operating systems. Building the system on Windows required significant design changes to the implementation of the Nooks subsystems, but the basic architecture held. Although we stopped short of a complete implementation, we were able to demonstrate most of the basic principles. We discuss our implementation and the lessons learned below.

4.2. The Design Under Windows

4.2.1. Binary Compatibility

4.2.1.1. NT Binary Ecology

Binary compatibility is the defining challenge for Nooks on NT. In Linux, the lax, source-based approach to configuration and module distribution makes it easy to implement quick-and-dirty approaches to compatibility. The Linux kernel is largely monolithic, using preprocessor source-level configuration and exposing few standard binary interfaces. Link-time binary interfaces, and sometimes even source-level interfaces, change from release to release. Given this unsteady foundation, few application and driver providers rely on binary distribution, and most are prepared to deal with periodic breakages as interfaces change. The original Nooks implementation took advantage of this flexible environment, altering macros, changing macros into functions, and occasionally modifying a few lines of driver source code.

NT, however, is based around a commercial binary-only ecology, where system interfaces are stable and compatibility is assured by Microsoft for large spans of time. Application and hardware vendors, each guarding their own trade secrets, provide a variety of binary-only modules that must ultimately cooperate to produce a working system. This is possible only because NT maintains persistent, well-defined binary interfaces, with long term bug-for-bug compatibility. Subsystems are then constructed out of layers of binary modules, loaded at boot or dynamically, where configuration and layering relationships are maintained in the registry. This yields predictable behavior and solid binary compatibility across versions and provides mechanisms for third parties to extend subsystems without access to the source code. Unfortunately, it also proves to be a major source of compatibility, interaction, and misconfiguration bugs. Third party kernel modules interacting with each other and with the kernel, often with incorrect implementations of incompletely documented interfaces, are a principal source of bugs. The rigidity imposed by strict binary compatibility also makes evolutionary progress difficult to realize. The problem faced by Nooks thus becomes one of treading lightly, preserving binary interfaces with high fidelity yet protecting the system from innumerable mis-implementations of those same interfaces by third party drivers.

4.2.1.2. The Nooks Loader

NT drivers are dynamically loaded and linked into the kernel by the kernel memory manager. Because driver source code is unavailable, we interpose our wrappers to track

allocations and maintain isolation by modifying the kernel's loader. All driver binaries list by name the routines they import from the kernel and from other libraries. At load time, the kernel loader traverses these import lists and places the routines' absolute address in the driver's "import address table" (IAT). Rather than introduce our interposition logic into the complicated core of the loader, we use the standard technique of replacing entries in the IAT with pointers to wrapper routines, after the kernel loader has finished its work. We use this mechanism to apply wrappers to all nontrivial kernel routines, allowing us to track control flow out of the driver.

Although this load-time interposition mechanism makes tracking control flow out of the driver straightforward, tracking control flow back in is more difficult. Kernel interfaces often employ callbacks or tables of function pointers. In many cases, we can make simple changes to all points in the operating system code which call into these entry points. In some cases, however, the function pointers are passed around arbitrarily, sometimes even to other third party drivers. It also may not be obvious statically whether a function pointer belongs to an isolated driver or to trusted code. Our solution to this problem was to dynamically generate wrapper stubs for every new function pointer and, using the existing wrappers, swap out the original pointers for the new stubs at runtime. In this way, we are able to interpose on all control flow in and out of the driver, allowing us to maintain kernel-driver isolation, track the resources exchanged across the boundary, and verify the correctness of the interactions.

4.2.1.3. Macros and Object Tracking

In NT, global data structures typically cannot be changed, which makes tracking shared resources more challenging (as well as impeding kernel evolution in general). Even data structures whose internals are intended to be private to the kernel often cannot be modified, because drivers do not always respect the official interface boundaries. Sometimes it is possible to make backward compatible changes to structures, such as by adding new fields at the end, but this too fails when drivers allocate the memory for structures on their own. This means that very rarely can we insert any new fields into existing shared structures in order to help us track their lifecycles, verify correctness, and clean up garbage. Instead, all tracking must be external to the objects, based on lookups in a large dictionary structure. An equivalent structure existed in the Linux implementation, but in NT it took on a greater role. Unlike under Linux, the NT object tracker was designed in a highly structured form with well defined interfaces. It was used to store a variety of information, including object type, lifecycle status, information for garbage collection, ownership, reference count, and any translations necessary across the boundary of isolation. This information was maintained continuously by the shell of wrappers surrounding the driver, tracking objects' lifecycle events as they flowed in and out.

As an alternative to functions, the use of macros in public interfaces is not inherently problematic for Nooks, but frequently their purpose is to manipulate shared objects and shared state. Because macro invocations are unobservable and cannot be wrapped, and unlike under Nooks for Linux, macros cannot be retrofitted into function calls, object

tracking on macro invocations is impossible. Instead, the effects of macros must be inferred after the fact from the changes they produce. In particular, many objects may come "out of nowhere", having been initialized by macros in driver-allocated storage and then provided to the kernel. These must be validated and tracked on demand. Also, unexpected state changes due to macro calls may occur on opaque objects already tracked. These changes must be detected, validated, and responded to the next time the object is passed back to the kernel.

4.2.1.4. Challenges

Although we successfully produced a promising start, binary compatibility for Nooks or a Nooks-like system on NT remains haunted by two challenges: a plethora of semantically rich interfaces that are difficult to wrap and numerous third party drivers that abuse those interfaces. The driver interfaces on NT are very carefully engineered, but not in the style of a typical kernel-user interface or an RPC interface. Indeed, they look much like high-performance internal interfaces rather than public interfaces.[2] Interfaces frequently employ semantically rich, complex shared data structures, often with embedded control information such as function pointers. These structures typically combine both public and private regions, with the private regions manipulated by macros. Some of these structures are even placed in driver-allocated memory, where their birth and death are unobservable. Although we've been successful at wrapping the interfaces we considered so far, such interface designs make the work far more difficult and the resulting system far more complicated than it needs to be.

Unfortunately, the complexity of these interfaces also makes them easy to confuse and very tempting to misuse. Driver authors frequently misunderstand the interfaces or leave significant bugs in their released implementations. Indeed, we observed special patches in the kernel to accommodate bugs in certain popular drivers. Our object tracking logic also discovered a notable (albeit benign) bug in the Windows 2000 PS2 keyboard/mouse port driver's initialization code (in handling of the notorious "pending bit" in a particular class of IO request packet). Even worse is the case of intentional abuse of the interfaces, where driver authors can't figure out how to achieve their goals legitimately or don't have time to implement a properly engineered solution and so they violate protocols and abstraction boundaries in order to implement the driver's core functionality. One popular example is system call hooking, which has so frustrated Microsoft that they are taking steps to disallow it completely in the Vista release [Microsoft05]. In wrapping many interface corner cases, we found ourselves asking the question not "is this allowed?", but "is this hack used?" Nooks must support a substantial fraction of what is actually done, not what is formally intended by the interface design. It remains an open question how flagrantly the majority of drivers violate the interfaces and how much of a barrier the diversity of violations would be to a large-scale Nooks implementation.

4.2.2. Libraries and Driver-Driver Interactions

4.2.2.1. NT IO Management

[2] We were told off the record that the interfaces in fact were *not* originally designed for the public, that Microsoft originally intended to write all drivers in house.

In NT, IO support is built up in a structured hierarchy of interacting drivers. The kernel's centralized IO manager interface is common to almost all drivers, responsible for service requests and communication, system state changes, device enumeration, and resource allocation. Even core bus subsystems like USB and PCI are implemented as ordinary drivers. All hardware devices and buses in the system belong to a single global hierarchy, with each parent responsible for enumerating and connecting its children (see figure 1 below).[3] Further, each node in the tree consists of a layered stack of cooperating driver instances (called "devices" in Microsoft literature). User requests propagate from top to bottom through the device stack. The bottommost layer in each stack is provided by the parent bus's driver, above which lies the driver specific to the device itself. Additionally, there may be one or more "filter drivers" in the stack, which modify or generalize the functionality of the device. Typical examples of filter drivers are antivirus and data encryption programs, which usually lay above the file system in order to intercept and manipulate file IO requests. Another important sort of filter driver is the "class driver", a generic filter provided by Microsoft which exposes standard interfaces to the rest of the system. For example, keyboard and mouse services are provided through the "keyboard class" and "mouse class" filter drivers, respectively.

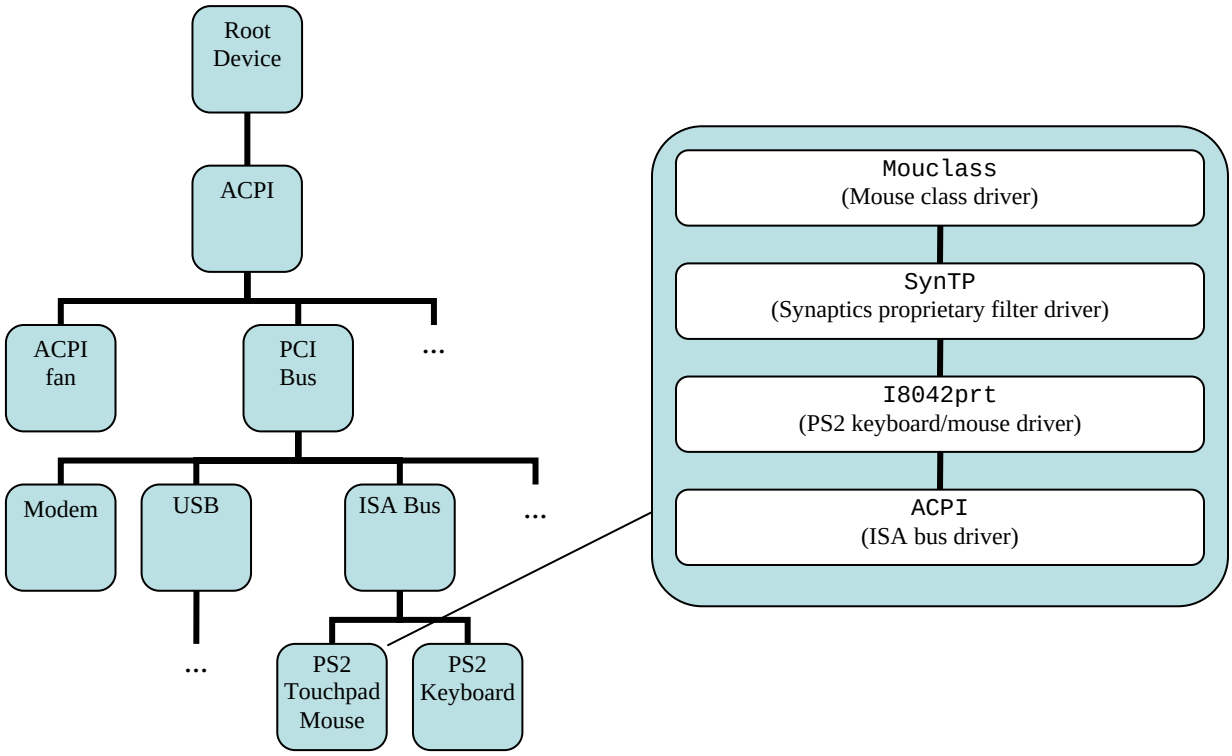

Figure 1 - Sample device tree, highlighting mouse device stack

This centralized IO manager interface is based around a mechanism known as the "IO request packet" (IRP). IRPs are an asynchronous communication mechanism, roughly analogous to a call stack divorced from any owning thread context. IRPs are used as a

[3] This hierarchy is rendered rather faithfully by Device Manager in the "View devices by connection" mode.

generic communication mechanism, employed for user IO requests, internal driver-to-driver requests, and system state notifications. It is IRPs that are passed down device stacks, created by the IO manager on behalf of users or by drivers in need of lower-level services. An IRP is sent to the top of the target device stack and, like the current execution context in a stack of function calls, propagated recursively downward until a driver instance completes it and returns it back upward. This mechanism is then used to implement a series of standard protocols for device attach and detach, reads and writes, IO control, and the like.

However, despite this centralized interface, a number of device classes use special, device-specific interfaces known as "miniports". Miniports are typically wrappers around the central IO manager interface, intended to provide either a simpler, more straightforward interface or cross-platform compatibility. By the same token, however, miniports constitute distinct, largely redundant interfaces that driver writers must learn and Nooks must support. While Microsoft has recognized this concern and is working to solve it in their new Windows Driver Foundation driver model [Microsoft06], miniports remain a thorn to deal with in any backwards compatible reliability system.

4.2.2.2. Boundary of Isolation

4.2.2.2.1. Introduction

The key defining feature of an isolation system is where the boundaries it creates are drawn. From this perspective, virtual machines, processes, and nooks form a continuum, differing primarily in the boundaries they provide. In most systems, two kinds of boundaries are present: the supervisor-child boundary and child-child boundaries. For transparent driver isolation, placement of the former, the system / driver boundary, is a nontrivial architectural question, though largely a tractable one. Since the system interfaces are reasonably well defined and complete source code for one side is available, the design can concentrate on the end goals of reliability, simplicity, and performance. Among Nooks-like architectures, the key challenge becomes where to put miniports, shared libraries, and system library services.

On the other hand, where to place the boundaries among the individual black-box drivers is inevitably an open question, one which may not always be answerable even at runtime. The simplest model for driver-driver isolation would be to place each driver module in its own protection domain, able to fail without disturbing other drivers. Unfortunately, this model is unrealistic under NT, where drivers interact directly and layer upon one another. As far as end-to-end functionality is concerned, this interaction necessarily ties the fates of different drivers together, regardless of any isolation policy. Further, it is commonplace for hardware vendors to provide multiple cooperating driver modules that share undocumented, proprietary communication channels, which cannot be marshaled across domains. Hence, it is necessary to support multiple drivers and multiple binary modules within a single protection domain, and somehow, to decide which modules to isolate together.

4.2.2.2.2. The Kernel / Driver Boundary

Although the overall notion of where the system / driver isolation boundary belongs is established by the Nooks architecture, the finer details must be resolved as implementation questions. In particular, miniports and shared libraries may be isolated along with their corresponding drivers, treating the entire monolith as a client of the central IO manager interface, or they may remain with the kernel, their module-specific interfaces split across the boundary. Such decisions are important, because the alternatives may be drastically different in terms of interface complexity and overhead.

One would prefer the isolation boundary to cross whichever interface is narrowest and simplest, to improve the odds of producing a robust, bulletproof implementation. For example, we were able to isolate our 3c905B-TX Ethernet card by wrapping about 60 functions from the NDIS network driver miniport interface. This was simpler than wrapping NDIS as a WDM driver, even though we already had a large set of WDM wrappers, because as a driver, NDIS is extremely complicated and broad, requiring on the order of 200 wrappers.

Alternatively, one would prefer that the boundary crosses the interface with the lowest rate of invocation, to reduce the overhead of cross-domain control transfer. This is particularly important for libraries that include simple, frequently used runtime services that, in principle, need not involve any kernel-driver communication. In some cases, it may even be worth factoring out and duplicating such code within the protection domain – we pursued this route for our heap allocator. Finally, it's desirable to minimize false dependencies and unnecessary shared fate, to make the granularity of failure and recovery as fine as possible. This becomes an issue particularly when libraries are used by multiple, independent client drivers yet maintain centralized shared state. In such cases, the library must be hacked to sever the shared state, must reside with the kernel, or must be isolated together with all dependent drivers in a single, unnaturally large domain.

4.2.2.2.3. The Driver / Driver Boundaries

Although the same issues of simplicity, efficiency, and fate sharing apply to the case of driver / driver boundaries as well, here they are largely overshadowed by the problem of compatibility and the tension between compatibility and adequate fault isolation. Since third-party driver modules can expose arbitrary undocumented, unmarshallable interfaces to each other, it is imperative to know which drivers communicate using such interfaces so that they can be isolated together. If this is not done correctly, serious failures can occur. If two cooperating drivers are isolated in separate domains, their communication is generally interrupted, and the drivers will likely stop functioning or crash. Even worse, if one driver is isolated and the other is not, the unprotected driver may crash and bring the system down. The trivial solution to this problem would be to isolate all drivers and put them in the same domain, but this leads to wildly shared fates, where a single driver failure causes a massive interruption in system functionality. This is further complicated by the separate but related problem that some drivers on the system may not be fully supported by Nooks; this means that neither they, nor any drivers they secretly cooperate with, can be isolated by Nooks. Thus, it becomes a crucial problem to be able to divide

the system's drivers into sets that do not separate any inseparable drivers but are otherwise reasonably fine-grained.

Unfortunately, for existing drivers, this problem cannot be solved by static inspection, and in the most general case, is extremely difficult even at runtime. The means of bootstrapping a communication relationship are varied, all of which are observable, but not all of which are easily interpreted. For example, a driver could send out an ioctl-type IRP with a proprietary function code; this would indicate an attempt to employ an unmarshallable interface. Unfortunately, it's not always easy to determine which driver is the target of such an IRP. Since IRPs are addressed to driver stacks, not drivers, any layer in the stack could be the destination[4]. Another conceivable problem case is the use of presumed "safe" shared stores for driver-driver communication. For example, two drivers could bootstrap their communication by passing pointers through the registry or the filesystem. Although we have never observed such a disgusting abuse of the interfaces, it remains a possibility. Thus, in general, this problem seems to be intractable.

In practice, however, there remain a few special cases we can handle easily. The most obvious solution is simply to have drivers ship with metadata listing their relationships. Although this is not a backward compatible solution, it is a very simple change for hardware vendors to add. Also, in some cases, cooperating drivers directly link against each other; this is trivial to detect at load time. Finally, it is straightforward to recognize the common special case of drivers that simply don't attempt to employ any proprietary interfaces. Although determining which drivers to isolate together remains a challenging issue for Nooks and an important question for future work, we think it unlikely to be an insurmountable obstacle. In practice, we expect that using conservatively large groups of drivers, and in the future, using driver-associated metadata, will prove to be sufficient.

4.2.2.2.3.1. Optimization

For driver-driver interactions, even though the problem of compatibility largely overshadows issues of efficiency and optimization, one can still say a few interesting words about the latter. One might expect that, with NT's heavy emphasis on driver-driver interaction, it would be necessary to optimize all cases. In practice, we find that domain-domain interactions are actually relatively rare. This is because, typically, a driver stack includes a few well-tested, trustworthy layers from Microsoft, and only one or two third party drivers, which are generally best kept in the same domain -- since they already share fate you typically lose nothing, yet you potentially gain better performance and compatibility. Hence, most interactions are either within the confines of a domain or between a domain and the kernel. For driver-driver interactions inside a domain, it's straightforward and worthwhile to streamline any Nooks overhead; typically, little

[4] This issue is somewhat mitigated by the fact that all driver instances in a stack have naturally shared fates; interrupting service on any one of them necessarily interrupts service for the whole stack. Thus, lumping together all untrusted drivers of a stack is not out of the question. There remains a problem, however, if some of the drivers in the stack are not supported by Nooks or if a single driver binary is instantiated in multiple separate stacks.

interference is needed. Thus, as with Nooks on Linux, the hot spot remains at the kernel / driver interface; anything to reduce the number of calls is a potential win.

4.2.2.3. Reference Counting and Crash Recovery

NT's uniform approach to kernel object management introduces a new challenge for the process of unloading a crashed driver. All standard "kernel objects," drivers included, are managed and reference counted by the centralized object manager service. Once a driver's hardware is removed or Nooks reports that the driver has failed, driver unload occurs automatically when the reference count reaches zero – but the system flatly refuses to unload any sooner (and with good reason). Unfortunately, referencing is a highly generalized mechanism; references can be held by user processes, by kernel objects and services, and even by other drivers. It's hard, perhaps impossible in some circumstances, to hunt down all reference holders and safely release their references. Instead, the solution is to convert the failed driver into a zombie, nonfunctional and invisible except for an object shell, its resources released. Then, a new copy of the driver can be loaded, and the driver's clients can reconnect.[5]

4.2.3. Memory Management

4.2.3.1. NT Memory Management

In this section, we discuss the NT memory manager, the impact of its design on the Nooks architecture, and our limited implementation of Nooks memory management. The NT memory manager provides the system with paged virtual memory with four gigabytes addressable on 32-bit x86 machines. Under the standard configuration, the low two gigabytes are reserved for user space and the upper two gigabytes are reserved for system space. Internally, the hardware's two-level lookup system is used to find the page table entry (PTE) for a specific virtual address, where each PTE stores permission and translation information. The user half of the address space is specific to the current running process, whereas most of the system half of the address space is global to all processes. All kernel code and loadable modules (including drivers) run in system space and are given mostly unchecked access to all addresses.

The most notable design element of the NT memory manager is the fact that it uses a virtually mapped kernel, with pageable code and data where possible. Up to half of system address space is dedicated to the large unified file cache, handled by the same paging mechanisms as kernel code and data. Unlike most UNIX variants (e.g. [Shah04]), physical memory is not mapped at all, except temporarily in small views for I/O requests. In general, the system handles physical memory relatively efficiently, and that is not normally a kernel bottleneck. On the other hand, kernel address space is very tight on 32-bit machines, given the large footprint of the file cache and other pageable structures.

[5] The situation would be somewhat simpler with shadow drivers [Swift04], however, where it's unlikely anyone but Nooks would ever have a reference to the real driver. Outside clients would probably only reference the proxy. Unfortunately, due to legal considerations, we could not explore shadow drivers under NT.

For Nooks, the most important aspects of this architecture are the synchronization and dependency constraints imposed by pageability, the precious nature of address space, the unusual means to access user memory, and the highly dynamic nature of the system address space. A few important questions also remain from the original Nooks work, relevant to NT and Linux alike, such as possible hardware optimizations and the challenge of reliably providing memory for object tracking and cloning even when free physical memory runs out. We discuss these in more detail in the following sections.

4.2.3.2. Nooks for NT Memory Management Implementation

The memory management architecture we implemented for Nooks was fairly simple and, due to time constraints, did not fill the requirements for a complete Nooks implementation. It allows us to specify special read-only or read-write permissions for each driver for every individual page of system space. There is no intelligent control over user address space – in principle, drivers are still allowed full access to user space addresses. Our current implementation is incomplete in particular because privileges are set statically when a driver is loaded into the system, and no changes are made as the system is running. Such privileges are assigned based on the major regions of the memory map; for example, drivers are never allowed to write to page tables or the file cache.

To specify permissions, each protection domain has a bit vector that represents its write privileges for system space. Each bit represents whether the isolated driver has read and write access or just read access for each PTE. When the system enters a protection domain, it switches over to the PTEs for that domain, but instead of just using a copy of the system's PTEs, it also checks the bit vector and masks off the write-enable bits as appropriate. While the driver is running, any attempt to write to address space protected by the bit vector results in an access fault. This access fault is then handled and treated as a fatal driver error, which the error detection subsystem catches, triggering driver recovery. It is important to note that, if memory protection is comprehensive, invalid memory accesses are never successful and hence no memory is corrupted before the driver can be unloaded.

We implement this architecture with a simple lazy management scheme. Rather than integrate tightly with the NT memory manager, we hook the TLB flush and page fault handlers, otherwise treating the NT memory manager as a black box. Each protection domain maintains a private copy of all the PTEs representing system address space, updated only when necessary. TLB flushes are used to trigger updates for PTEs that were previously valid, while page faults are used to trigger updates for PTEs that were previously invalid. In this scheme, new addresses are imported lazily, a potentially significant win since most drivers never touch more than a fraction of system address space. When a page fault occurs, the page fault handler checks to see if the page is valid in system space but not in the domain's private page tables, and if so, lazily imports the PTE.

Entire TLB flushes are also handled lazily. Individual TLB entry flushes do trigger synchronous updates across all domains, but when the whole TLB is explicitly flushed, Nooks instead sets a special dirty bit on each domain. Whenever a dirty domain is called or returned into, the entire page table is re-copied. Because copying page table entries for all of system address space, 2MB in all, is an expensive operation, laziness is a crucial optimization for the common case of multiple protection domains where only a few are active at a time. Indeed, although our performance is normally good, TLB flush intensive tasks can make the system noticeably jerky under this scheme. Fortunately, we do not need to consider ordinary address space context switches as TLB flushes, because all the pages in system space which drivers need access to are marked global and hence are not flushed.[6]

In all, the current implementation has three major limitations. The first is that there is no direct access to user space addresses; this is discussed more fully below. The second is that our address protection is entirely static, based on memory regions, not based on individual objects like the original Nooks system; the latter requires some additional infrastructure, including proper cloning of shared objects and a private, domain-local heap allocator, which we ran out of time to complete. Finally, our performance is somewhat less than optimal, because of our black-box treatment of the NT memory manager. Even within the black-box scheme, there was room for significant optimization which we did not exploit. These limitations were mainly a result of time restrictions, however, and do not reflect any fundamental incompatibility between the Nooks architecture and the NT memory manager.

4.2.3.3. Handling User-Space Access

Due to time limitations, we did not attempt to implement a complete and correct system for handling user-space access. Although limited support is straightforward, supporting complete access to user memory in Nooks is much more complicated under Windows than under Linux, and thus anyone building a Nooks-like system for Windows must carefully consider their mechanism for user-space accesses.

Under NT, access to user-space addresses through the Nooks architecture is not at simple as it might seem. Unlike Linux, NT does not provide special macros or functions to access user addresses but instead allows kernel code to execute arbitrary instructions manipulating user addresses, so long as it obeys the simple precautions of calling an access rights verification function and wrapping accesses in a try/catch block.

Because of time constraints, we did not implement proper access to user space addresses. This is not as much of a limitation as it might seem, however. NT provides drivers with three different mechanisms to exchange data with user processes. The simplest mechanism is buffered IO, where the kernel automatically double buffers user requests, and the driver never touches user memory. The next method is known as direct IO,

[6] On the other hand, the flushes associated with domain transitions impose a significant performance penalty of their own, since Nooks must switch address spaces and explicitly flush global pages on every entry and exit from a domain.

where the kernel provides the driver with a list of physical frame numbers to either pass on to hardware DMA or map into kernel address space. Finally, under special circumstances, drivers may use the so called "neither IO method" and directly manipulate user addresses as described above. This is relatively uncommon. Indeed, among the many sample and live drivers in the driver development kit (Server 2003 edition), virtually none use the neither IO method to communicate with user space. Unfortunately, this access method is also very difficult for Nooks to handle.

Under Linux, because all accesses to user addresses must go through special macros, it's straightforward to replace those macros with general purpose functions that automatically handle all the issues for user-space access. Under Windows, however, arbitrary driver code is more like a black box, and there are no such quick fixes. The driver's pagetables must provide access to user-space as appropriate, and they must be kept in synch with context switches as well as address space updates. Because the NT kernel is preemptable, this is difficult even on a uniprocessor, unless you are willing to maintain one set of pagetables for every protection domain / user process pair. The overhead of implementing this directly would seem to be prohibitive. Although we have some vague ideas (e.g. laziness), we do not know if there are any good solutions to this problem.

One additional concern is how to set the isolation policy between drivers and user mode processes. In most cases, the interfaces used between client processes and drivers will be standardized and wrapped, and so the memory access policy will be handled by wrappers. However, it's always possible that an undocumented channel exists between a driver module and a user process, for example, as with an antivirus filter driver and its user-mode counterpart. This is not such a catastrophic situation as a hidden channel between drivers, because a mistake cannot crash the system, but for proper compatibility, the system needs a policy that allows the driver appropriate, unchecked access to the specific processes with which it cooperates. Conceptually, this problem needs to be treated similarly to the more dangerous undocumented driver-driver interactions, where multiple drivers are placed in the same protection domain, but the mechanism required here is different.

4.2.3.4. Memory Management: Picking The Right Approach

Even though we did not implement all the memory management features needed for Nooks, the basic TLB-based mechanisms we developed were largely capable of supporting them – though not necessarily efficiently. Our design favored above all simplicity of implementation – a valuable goal, but one of many in practice. Besides simplicity, the principal concerns we encountered for a practical, efficient implementation are memory overhead for storing the page tables, CPU overhead for keeping the page tables up to date, page locality, and address space consumption.

The first main implementation consideration is memory overhead. The current system uses 2MB per domain to keep separate, pre-allocated copies of the kernel portion of the page tables. Though 2MB is not egregious, it can easily add up for a user wishing to preemptively isolate most of the drivers in their system. In fact, however, this 2MB is

mostly wasted, since large portions of address space may never be mapped by the system. Worse, the vast majority of address space is never even accessed by any one driver. One possible way to save most of this overhead is to use a more sophisticated memory management scheme that lazily allocates page table space when the driver first touches memory pages, although this may raise new concerns associated with inconveniently timed requests for more memory (a problem discussed more below). Alternatively, it is possible to share a single, read-only copy of system's page tables across all domains, creating private copies of page table pages only when a driver needs to be granted write access to some associated page.

The next potential concern is the overhead associated with keeping domain page tables synchronized with changes to the system's page tables. In our lazy, black box implementation, this becomes a serious issue because of the high cost of copying complete page tables after TLB flushes. It should be possible to greatly optimize this by invalidating en masse large chunks of the page tables rather than copying them, at the cost of some tricky implementation details. Alternatively, a scheme that integrated more directly with the NT memory manager and synchronously updated all domains' page tables together might be more efficient. This may not be as clear a winner as it was under Linux, however, because unlike Linux's rather static system address space, NT updates many thousands of system PTEs per second under normal load. A third, intriguing possibility is to segment system address space such that isolated drivers occupy their own designated region. Since no page table pages will ever map both kernel pages and driver pages, they can be maintained quite independently. This maximally leverages a single set of read-only system page tables shared across all drivers, saving both update overhead and memory.

Two final concerns are the locality of driver pages and the consumption of address space. A memory management scheme may keep a domain's pages in special, contiguous regions, or it may simply use the system's normal page allocator, leaving them scattered across the system address space. Contiguous regions offer the opportunity to use large pages to map the domain's memory, potentially saving on TLB misses inside the driver and during copy-in / copy-out. On the other hand, they consume more memory and more address space, due to internal fragmentation. System address space is a particular concern for NT on 32-bit machines, and with its added system memory consumption due to page tables and isolation overhead, Nooks does little to aid the situation. The contiguous region strategy suggests a possible countermeasure, however: rather than mapping all drivers in memory at once, swap them in and out of a single common region of address space[7]. Whether or not the savings justify the added complexity is debatable, but since Nooks causes isolated drivers to communicate through the kernel rather than directly among each other, swapping now becomes a viable option.

If we allow ourselves to imagine some simple improvements to the x86 virtual memory architecture, the landscape changes dramatically. Adding a tagged TLB would virtually eliminate the bursts of TLB misses due to domain transitions and, if the Nooks for Linux experience is any guide, result in a massive speed-up. Moving to a 64-bit address space

[7] This is similar to the "session space" mechanism already used in NT for multiple graphical login sessions.

would render address space consumption a non-issue, although page table memory consumption could become significantly more troublesome. A dramatic solution to both TLB misses and page table memory consumption would be to separate permission checking from address translation at the architectural level. Address translation necessarily lies on the critical path for memory I/O; permission checking does not and could be performed lazily before instructions commit. Separating shared, bulky translation tables from private, compact permission tables should eliminate most of the memory overhead of protection domains and replace expensive, frequent TLB misses with cheap, infrequent PLB misses. Finally, taking this to the extreme, Mondrian Memory Protection [Witchel02] could even eliminate much of the data copying associated with cross-domain calls, making lightweight protection domains almost free.

Time constraints did not allow us to pursue any of the more sophisticated approaches to Nooks memory management, and our approach was clearly far from optimal. In order to determine which design would serve best, it would be necessary to gather data on the way drivers utilize memory and estimate the overhead of each scheme or to implement some of the alternative approaches and measure which had the best tradeoffs. Regardless, it's clear there is room for substantial improvement in software beyond our current scheme, and even more room with hardware changes as well. So, besides the problem of user space access (discussed above), it appears that the memory management requirements of Nooks do not present any serious obstacles for NT.

4.2.3.5. Out of Memory Corner Cases

Black box isolation techniques, which require cloning and tracking objects on demand, inherently demand memory allocations be made at arbitrary and potentially inconvenient times. In particular, allocations are required when new objects from the kernel are first passed to the driver or when stack-allocated or driver-initialized objects are passed to the kernel. Allocations are also needed on occasion for new thread stacks for the driver, new callback wrappers, and the like. These allocations can be rather difficult to fulfill at high priority level or when the system runs out of memory. In general, the system needs to be prepared to handle low memory situations gracefully. Unfortunately, even well-hardened code becomes vulnerable to memory shortages when wrapped by black box isolation. It is generally possible to restart the affected driver when memory runs out, but this potentially pushes the ill effects to user mode clients. This memory sensitivity is thus a new reliability hazard introduced by adding isolation.[8]

Although we suspect there is no practical solution to the entirety of this problem, we have devised (though not yet implemented) mechanisms we believe mitigate the problem to an acceptable level of risk. Allocation requests at high priority level, potentially higher than the system allocator can correctly handle, are the easiest to handle; we simply redirect them to a privately maintained heap. This then reduces the high priority problem to the

[8] Indeed, this was one of the principal concerns about Nooks voiced by members of the NT kernel team.

out of memory problem, with the caveat that paging activity is impossible.[9] We can address this out of memory problem with four different mechanisms: reserving a pool of memory in advance, speculatively pre-allocating memory, deferring requests, and gracefully downgrading protection.

The standard technique for dealing with unavoidable emergency memory demands is to maintain a pre-allocated reserve pool. A key difficulty with this method, however, is determining the right size for the pool; too large is wasteful, but too small risks running out. Fortunately, for Nooks, there appears to be straightforward solution to robustly scaling the size of the pool. Nooks unexpected allocations are not random; some of them correspond to tracking and cloning kernel resources being passed to the driver, and some of them correspond to driver resources being passed to the kernel. For the most part, there already exist readily exploitable failure paths associated with the former cases, and so only the latter cases present a problem. However, the corresponding driver resources must necessarily reside in memory already allocated to the driver, and so the amount of memory currently granted to the driver is directly proportional to an upper bound on how much Nooks could need unexpectedly in the worst case. Hence, by keeping in reserve a certain amount of memory for every page allocated to the driver, one should be able to satisfy unexpected allocations with reasonably high assurance.

For certain cases, it's also possible to avoid the problem of unexpected allocations by speculatively performing the allocations early, when the priority level is often low. For example, kernel objects placed in driver-allocated memory often have an initialization routine and various routines for handing off the object to the kernel. Because it's possible to deallocate such an object silently by re-using or freeing its memory, Nooks cannot begin tracking the object immediately when initialized. Instead, it must wait until the object is transferred to the kernel, which may occur at an inconveniently high priority level. However, one could speculatively allocate the necessary memory at initialization, leveraging the common case where such objects are used repeatedly. For drivers that do follow this common case, every such object now has two chances for a successful allocation, the first of which is usually at a low enough priority to allow paging activity. Only if we observe too many old, unused speculative allocations building up do we need to begin freeing the likely mis-speculations.

Another possible solution to handling out-of-memory when a new request is passed to the kernel is to leverage NT's ubiquitous asynchrony and defer processing the request until later. For example, when memory is scarce, objects such as work items, deferred procedure calls, and IRPs can be chained onto a private linked list for later processing, rather than immediately allocating memory and transferring them to the kernel. Some of these requests can be deferred indefinitely, in hopes of more memory being available. Others can be deferred to low priority level, allowing paging activity to resume. Either way, deferral gives the system more opportunities to find enough memory to allow Nooks to satisfy the request.

[9] A particularly troublesome case, allocation of stacks at high priority level, actually disappears entirely. Since thread context switching is disabled at high priority levels, we simply pre-allocate one stack per processor for use at high priority.

In cases where enough memory simply isn't available, a final possible choice is to gracefully downgrade protection. Certain allocations, such as object clones and private thread stacks, are necessary for proper protection but not for correctness. If memory is unavailable for a clone, it is often possible to share the original instead, at the expense of diminished integrity guarantees. If such steps are taken, it may no longer make sense to assume that the system is adequately protected from driver faults such that automatic recovery is safe. However, even when all else fails, this largely eliminates the problem of unexpected memory allocations introduced by Nooks.

4.2.4. Asynchrony

4.2.4.1. NT Asynchrony

NT is, by design, a highly asynchronous and concurrent system. Unlike Linux, multiprocessing support was included in the kernel from the very beginning. Also, the driver model and IO request processing are totally asynchronous, using the IRP mechanism. Asynchrony and concurrency significantly change the landscape and the set of tools available within the system. They are often challenging to work with, though except for one concern, by and large they do not introduce any fundamental changes to the Nooks approach. In some cases, native asynchrony even presents new opportunities for optimization.

4.2.4.2. Synchronization Interoperability & Asynchrony Safety

Although the Nooks code base is relatively simple and lightweight, it cuts across many layers of the system and levels of synchronization. Core services such as the object tracker must run at an extremely high synchronization level, since they may be invoked at any time by drivers running at arbitrary synchronization levels. In particular, NT assigns each processor a synchronization priority level known as the Interrupt Request Level (IRQL), a sort of fine-grained generalization of the interrupt enable flag. IRQLs are the primary global synchronization mechanism exposed to drivers, representing a series of concentric per-processor locks. Because drivers may demand Nooks services from a high IRQL, Nooks must synchronize against that level internally. This makes the affected Nooks code trickier to construct, but more importantly, the cross-cutting nature of Nooks means that kernel services which were never expected to be needed at a high priority level must somehow be shoehorned in.

One particularly troublesome example is the recovery and garbage collection code. Because recovery may be invoked from an extremely high synchronization level, and because the garbage collector must synchronize with the object tracker, many recovery and garbage collection routines must run at high IRQL. Unfortunately, few kernel routines for freeing resources were written with this requirement in mind. Also, many important library routines, such as those for manipulating Unicode names, are capable of running only at low priority level. These constraints have an important impact on how Nooks services are structured, demanding an intricate dance between high and low priority levels. Although we managed to work around the specific instances of this

problem we faced, there seems to be no guarantee that a subsystem as cross-cutting as Nooks will never encounter unsolvable cases.

4.2.4.3. Asynchrony in the Driver Model

The heavy use of asynchrony in the driver model, combining both threaded and event-driven programming, represents a significant source of complexity. Basic request processing is tricky, but the semantics of complications such as IO request cancellation, Plug & Play, and power management are nightmarishly intricate in the asynchronous framework [Oney02] [Maffeo04] [Oshins04]. The state machines for drivers and associated objects are thus incredibly difficult to implement correctly, representing a significant source of driver bugs. Although these bugs are generally straightforward to handle in Nooks, it is rather challenging to properly implement the wrappers and validation code to enforce correct driver behavior.

4.2.4.4. Leveraging Asynchrony to Reduce Domain Transitions

In one respect, however, heavy use of asynchrony in the standard NT driver model may prove to be an asset. As with most hardware-based isolation schemes, given the present state of commodity hardware, the single most expensive operation for Nooks is the domain transition. Hence, the principal goal in optimizing such an isolation system is to reduce the number of domain transitions. One effective way to do this is to batch requests, amortizing the cost of a single domain transition across multiple requests. This is only possible if the semantics of the requests allow them to be deferred, however. Fortunately, these are precisely the semantics of IRPs, the principal mechanism both for issuing requests to drivers and for driver-driver communication. Other common asynchronous mechanisms, such as "deferred procedure calls" and thread pool "work items", also may be deferred and batched with later requests. How much of an improvement such batching would provide in practice depends on what proportion of cross-domain calls are deferrable and also how often drivers synchronously wait on the results of such requests, but we suspect the improvement could be significant. This may be an interesting question for future implementation work.

4.2.4.5. Kernel Preemptability and SMP Support

Because NT was designed with fine-grained concurrency and SMP support from the beginning, we made an effort to determine just what would be necessary to make Nooks fully compatible with such an environment. In particular, Nooks must synchronize with the rest of the system and synchronize within itself; simply disabling interrupts is no longer adequate. As it turns out, however, for internal synchronization, what we ended up having to do amounted to almost the same thing. As discussed before, because Nooks must support the driver executing at its device's IRQL, it must synchronize against that IRQL. This leaves most device interrupts disabled, except for the most crucial ones such as the clock. Furthermore, because Nooks must synchronize across all processors, and because blocking synchronization is impossible at such a high IRQL, it must use spinlocks instead. Thus, we protect internal structures such as the object tracker by raising the IRQL extremely high and acquiring a domain-wide spinlock. Of course, we

try to hold such locks for as short a time as possible, but it's conceivable they could be become a bottleneck for multiprocessor scalability.

Another key concern on multiprocessor systems is the ability to bail out and restart a driver regardless of how many different CPUs it may be running on. In general, it is crucial to prevent old driver code from continuing to run after reload, because it could interfere with the restarted driver on the hardware. Furthermore, it is desirable to unwind as quickly as possible all threads that called into the driver, so that resources associated with the domain can be freed and lost user requests can be properly aborted. On a uniprocessor system, aborting and unwinding other threads involves a somewhat tricky and messy process, but it is guaranteed that all other threads are waiting. On a multiprocessor system, one must also be able to abort running threads, located on other processors. Fortunately, although we have not tested this, it does not seem too challenging, because all threads are probed periodically by timer interrupts, and a thread can easily be unwound during an interrupt handler epilogue. In NT, driver code almost never disables interrupts, relying on IRQLs to restrict them instead. In the case of a very stubborn driver thread that had interrupts illicitly disabled, one could un-map and overwrite all of its memory, which would boot it out in short order.

4.3. Evaluation and Limitations

Although a lot of work went into the project, there simply wasn't enough time to complete a full Nooks implementation. Instead, we followed a bottom-up approach, emphasizing the key infrastructure components and adding skeleton implementations of the higher layers sufficient to demonstrate a functional system. Basic infrastructure components we completed include the loader and associated interposition mechanisms, protection domains and cross-domain control transfer, object tracking, and garbage collection. On top of this we built a substantial library of wrappers and added object lifecycle logic for a number of core objects, including IRPs. These were sufficient to demonstrate complete crash recovery for the *i8042prt* driver (PS2 keyboard/mouse) and partial functionality for *null*, *beep*, and *mouhid* (USB mouse). We also made substantial progress towards supporting *el90xbc5* (3com EtherLink), notable because it was an NDIS miniport driver, not a native driver, and because we did not have access to driver source code.

The principal limitations of our implementation were significant but mostly due to time constraints. Most notably, our memory management component provided very little isolation, and we allowed drivers to work directly with kernel objects, rather than cloning them. We also did not support multiple driver modules per protection domain, in spite of the strong argument we developed for its necessity. Our selection of wrappers and fully tracked objects were limited as well, though this is a task of arbitrary size; each additional driver demands a few, and each additional interface demands a significant number. We also cut corners on performance and did not make any rigorous measurements. Finally, had we had more time and been able to address the licensing issues, we would have also liked to try implementing shadow drivers [Swift04], a mechanism for protecting client applications from the disruption associated with the loss of driver state during a restart.

Our goal originally was to complete most of the above and conduct rigorous performance and reliability measurements of the resulting system, so that we could make concrete claims about the commercial viability of Nooks beyond the UNIX world. Unfortunately, our time proved too limited. In particular, we were unable to complete proper memory isolation and object marshalling, the two most critical components for meaningful performance measurements. Hence, we did not attempt any formal measurements, although our informal tests seemed reasonable and the system was usable, despite the lack of tuning. While we did not reach our original goal, we did manage to demonstrate most of the basic building blocks, and we believe our experience suggests that the remainder would not be too difficult to implement.

5. Expanding the Scope: Flyweight Isolation

5.1. Introduction

In addition to our work applying the fundamental components of Nooks to NT, we also examined ways to modify the Nooks architecture to be more appealing to commodity system vendors and consumers. Unfortunately, our Windows implementation was not complete enough to support meaningful modifications and comparison measurements, but we were able to perform preliminary experiments on Linux. We noted that, while the performance of Nooks on Linux is generally very good, its overhead is not entirely negligible. For some groups, particularly the benchmark-sensitive commodity OS vendors, it may yet be too large. Given the incredibly high success rate of Nooks recovery, we decided to investigate whether weaker forms of isolation could provide reasonable reliability improvements with significantly reduced performance costs. Our initial results proved quite encouraging. The crucial questions, then, are how much isolation is truly necessary to provide solid reliability improvements, and how much performance can reduced isolation buy?

We propose that operating systems can tolerate driver faults without memory isolation, in spite of driver code written in type-unsafe languages. By executing drivers only on copies of kernel data structures, and by segregating driver memory from kernel memory, drivers are unlikely to corrupt the kernel when they fail. Existing Nooks recovery techniques can then recover the failed driver and allow the system resume execution.

5.2. Justification

Fundamentally, tolerating driver failures requires that the OS first detect driver failures and then recover from the failure. Failures can be detected in many ways, including hardware memory protection, explicit software checks on memory accesses, or higher level checks interposed at the kernel-driver interface. Recovery requires that the system, as a whole, be moved to a clean, un-corrupted state. This may be done by rolling backwards and undoing any corruption caused by a failed driver, or by rolling forwards and repairing any corruption.

Memory isolation, as provided by Nooks and other systems, provides error detection by trapping illegal memory accesses, and aids recovery by limiting the scope of corruption to data within the driver's protection domain. Hence, failure recovery consists of discarding the contents of the driver's domain and then recreating its contents by restarting the driver. However, memory isolation alone is not sufficient to ensure that the corruption is limited to the driver's domain. Data passed out of the driver could still cause corruption in the OS or applications, because driver interfaces do not strictly check for all possible bad outputs.

While simplifying recovery, full memory isolation comes at a cost, which is a result of the domain transition required on every call form the kernel to a driver. In Nooks for Linux, given the current state of commodity hardware support, nearly half of the overhead is due to memory isolation. The total overhead is negligible for most drivers (0-15%), but in particularly demanding drivers this cost may be too high. Full memory isolation is also complex and resource intensive, because it requires maintaining either a whole virtual machine running a separate kernel [Erlingsson05] [Fraser04] [LeVasseur04] or a copy of the kernel page tables. In both cases, providing memory isolation requires substantial code and many megabytes of kernel memory. Finally, full memory isolation may pose compatibility problems for devices or components that use non-standard, memory-based interfaces, because they may no longer have access to user memory and shared objects or, in the case of virtual machines, may access the wrong object, i.e. in the guest instead of the host system.

Even without full memory isolation, recovery is possible for the wide variety of common, non-corrupting driver failures [Microsoft03] [Maffeo04] or if the scope of corruption is limited to the driver's private data structures. External manifestations such as bad parameter usage, violations of protocols, unresponsiveness, corrupted private heap headers, and invalid memory accesses can be detected at the interface between the driver and the kernel. The existing recovery mechanisms from Nooks are still able to unload, reload, and restart a failed driver. For errors that are detected before the kernel or application is corrupted, the recovery ability is equivalent with and without memory isolation.

In general, then, if corruption is confined to a driver's private data, the OS can recover by unloading and reloading the driver. As a result, it is possible to greatly improve system reliability by executing drivers on private copies of kernel data, but without the expense of full memory protection. We call this limited isolation flyweight isolation.

5.3. Supporting Data

For flyweight isolation, we rely on separating in space the driver's working data from kernel data, giving the driver separate thread stacks, a separate heap, and private copies of shared kernel data structures. Whenever data is passed between kernel and driver, we use wrappers and the cross-domain calling mechanism to carefully validate all parameters, translating and synchronizing between kernel and driver copies of shared objects as necessary. The removal of memory isolation may seem rather dangerous, but

this approach makes for a reasonable strategy because most memory errors are not in fact random "wild writes" [Sullivan91], and most instances of wild corruption turn out to be harmless [Messer01]. Instead, most significant corruption occurs within data structures already being manipulated or in nearby blocks of memory, the manifestations of common bugs like off-by-one errors, buffer overrun/underrun, dangling references, and race conditions. By separating driver data from kernel data, we protect against the first-order effects of these errors, confining the corruption to within the driver. As long as the faults are then detected before multiple external manifestations begin to appear, the driver can be swept away and the system can continue running without any adverse consequences.

In a first attempt to compare how flyweight isolation behaves compared to Nooks we performed two experiments. First, we conducted synthetic fault injection experiments demonstrating that flyweight isolation detects and recovers from most of the failures detected by full memory isolation. Second, we compare the performance of flyweight isolation to Nooks to demonstrate the performance benefits of flyweight isolation over full isolation. Both of these tests were done under Linux.

In the first tests, we performed a set of random fault injection experiments on the pcnet32 100Mb Ethernet driver. We chose this driver because it is emulated by VMware, allowing us to perform these tests within a virtual machine. We loaded the driver into memory, injected five realistic bugs into the program text, and then tested the system to see if it failed. We performed these tests on three platforms: *Native Linux*, a system without any isolation or recover, *Nooks*, a system with the Nooks driver-fault isolation subsystem, and *flyweight Nooks*, a system with Nooks object tracking and recovery mechanisms but no memory isolation. We performed 900 hundred fault injections across 9 different types of synthetic bugs injected into the driver.

	Native Linux	**Nooks**	**Flyweight Nooks**
Crashes	163	2	3

Table 1 – Fault injection outcomes.

Table 1 shows the number of system crashes experienced on the three platforms. The flyweight Nooks system experienced only one additional crash beyond Nooks and prevented 160 crashes as compared to native Linux. These results demonstrate that full memory isolation is not required for tolerating at least a large number of driver failures. Rather, fault detection and recovery are the critical ingredients.

To evaluate the performance benefits of flyweight Nooks, we ran a driver on the same three platforms and compiled the Linux kernel source tree. These tests were done on a 1.7 GHz Pentium 4 with 900 MB of RAM. Because the pcnet32 driver uses little CPU, we instead chose the VFAT file system driver (although, unfortunately, we were unable to perform new reliability tests on it). We note that compared to most drivers, VFAT is a worst case because of the amount of sharing with the operating system, and hence the amount of data copying. In Table 2 we show the number of seconds spent in the kernel on our three test platforms (the time spent in user mode was unchanged). The results demonstrate the savings of removing lightweight kernel protection domains with their

inherent page table changes and subsequent TLB misses. The cost of the remaining flyweight Nooks isolation mechanisms is not negligible, however, because it still requires many additional data copies and lookups in the object tracker.

	Native Linux	**Nooks**	**Flyweight Nooks**
Time in Kernel	38 s.	105 s.	67 s.

Table 2 – Time spent in kernel mode when compiling the Linux kernel.

Overall, these experiments demonstrate first, that flyweight isolation can provide a tangible benefit to system reliability by recovering from non-corrupting failures. Second, they demonstrate that there can be a sizeable performance benefit to removing hardware-based memory isolation, because it avoids the single largest overhead source in isolation systems.

5.4. Conclusions

Although the concept of flyweight isolation shows promise, as yet it raises more questions than it answers. Most fundamentally, how often do wild writes occur across drivers in general, and how many distinct external manifestations do typical faults cause? Is it reasonably safe to conclude the first external manifestation observed is in fact the first one experienced? These are basic questions about the behavior of commodity code reacting to type-unsafe corruption, for which only rough hints exist in the literature.

Another important question flyweight isolation raises is the issue of policy. Most operating systems today treat failure handling as a matter of pre-ordained mechanism. NT, for example, immediately halts the system and writes a crash dump when it detects a null pointer dereference. Linux, on the other hand, responds to the exact same fault by terminating the active process and printing a message to the console. What accounts for the difference? Not any documented difference in robustness to corruption, as far as we are aware. Rather, we contend that this illustrates that failure handling behavior is an issue of policy, not mechanism, and that different user communities have different preferences. Nooks adds the option of driver isolation, but it comes at a performance cost, which may be unacceptable for some users. High-end, video-intensive applications may be a perfect example, very sensitive to driver performance but also plagued incessantly by driver bugs. Flyweight isolation adds a new point in the reliability spectrum, allowing users to reap most of the reliability advantages of Nooks with significantly less cost. Of course, as with any policy, someone must be prepared to make the choice.

6. Future Work

6.1. Nooks Performance

6.1.1. Breakout Access

One intriguing possibility for improving Nooks performance despite the lack of hardware support for low-cost domain transitions is to leverage the lightweight nature of the

protection domains to allow partial domain exits. Consistent with Nooks' best effort design, entry and exit from a lightweight protection domain are voluntary. The procedure does not depend on special secure traps or call gates but rather on a particular sequence of privileged operations extremely unlikely to be executed by accident. In this vein, it's also reasonable to consider partial transitions, which when invoked allow partial write access to kernel memory. By exploiting obscure aspects of the x86 architecture, it is possible to implement such mechanisms, unlikely to be triggered by accident, but much faster than completely switching page tables and flushing the TLB. Possible mechanisms include segment limits – reloading the segment registers could allow access to the top of memory – and toggling the processor's write-protection-enable bit. This could then be exploited to provide accelerated implementations of critical-path system services or cross-thread synchronization inside the protection domain, without the overhead of full domain transitions. Whether this might be worthwhile would depend on the particular mechanism chosen, the particular hot spots of interest, and how sensitive their implementations would be to the integrity of the execution context. If hardware manufacturers continue to disregard efficient support for lightweight protection, this may be an interesting question for implementers of Nooks-like services to explore.

6.1.2. Memory Reservation Service

An interesting possibility for reducing the performance impact of memory reserved for unexpected allocations is to allow reserved pages to be used for non-dirty pageable memory while they're not needed. This could be encapsulated as a "memory reservation service," which would allow clients to reserve quantities of memory and then guarantee its availability at any time, any priority level, regardless of free memory levels. When the number of free or non-dirty pages in the system exceeded the sum of all reservations, the memory would be utilized as normal. Only when the number of claimable pages dropped to the number reserved would the pages be explicitly evicted, allocated, and locked down. While we don't necessarily expect a large amount of memory to be required to render negligible the risk from unexpected allocations in Nooks, this service could be useful generally, as well as helping to make the cost of Nooks more palatable to the most demanding of users.

6.1.3. Video Drivers

Although not fundamentally different from other devices, isolating video drivers represents an important unexplored direction, since video drivers are notoriously complex, buggy, and extremely performance sensitive. Early attempts at isolating video drivers using Nooks for Linux were abandoned, because Linux's split video driver architecture, involving both kernel modules and a user mode X11 library, makes adapting the Nooks mechanisms unnecessarily complicated. Fortunately, NT video drivers appear to be amenable to the ordinary Nooks isolation mechanisms, but whether the performance will be adequate or whether it might require special considerations such as flyweight isolation remains unanswered.

6.2. Quality and Generality of Isolation

An important question left unanswered by our work under NT is, how often do real-world, third party drivers violate the basic interface specifications, working only by virtue of system implementation details? In principle, Nooks can provide virtualization for most any interaction, but if every driver used its own, distinct hacks, the complexity of supporting most drivers could quickly become overwhelming. Until Nooks became widely deployed and driver writers had to test against it, this issue might represent a serious practical obstacle.

A related question is, how much does a cleaner, more naturally isolatable interface help Nooks? Typical driver interfaces make heavy use of shared data structures and cross-domain pointers, client-allocated memory, mutator macros, and other unpleasant complications for isolation. Cleaner interfaces designed around opaque handles and abstract data types, such as the new WDF interface, could potentially eliminate many kinds of hacks and driver misbehaviors, simplify the complexity of Nooks, and, at the same time, reduce its overhead. How large these possible benefits prove to be is a question of significant practical importance.

Another possible direction for the future might be to solicit assistance from drivers themselves to facilitate isolation, error detection, and recovery. If drivers explicitly identified what other drivers and user processes they shared fate with and communicated with over nonstandard interfaces, it would eliminate the problem of deciding which modules to isolate together. Alternatively, if drivers employed self-describing RPC-like interface styles, automatically isolating drivers in spite of proprietary, one-of-a-kind interfaces might be possible. Drivers might even provide lightweight integrity or functionality probes, signaling Nooks to restart them more promptly, minimizing downtime and the internal spread of corruption.

A final practical question not yet addressed is how to decide which drivers to isolate at all. Ideally, one might imagine isolating all drivers, but this is neither necessary nor practical. Anecdotal evidence suggests some drivers are significantly more reliable than others; isolating them incurs a small performance hit in exchange for no real benefit. Furthermore, there will always be some number of drivers incompatible with Nooks, and attempting to isolate these drivers will cause them to fail. In some cases Nooks will be able to detect these failures, but it other cases they may be too subtle. What other techniques are available to help determine which drivers to isolate, besides trial and error? Perhaps online repositories could provide compatibility information, or collaborative aggregation of fault data could help determine which drivers are most bug-prone?

6.3. Integrity Beyond Domains

How do Nooks and other isolation and recovery systems behave in the presence of genuine hardware problems, kernel bugs, or other unconstrained sources of corruption? Do they provide any benefit, helping to keep corruption confined? Or, do they make the situation worse, by recovering and continuing to run when the system should be shut

down, masking the true problem while allowing damage to propagate? Is it even possible to recognize the difference between isolated and global corruption?

One interesting solution to this problem might be to attempt kernel data structure integrity checking. Besides being useful for offline crash dump analysis, integrity checking could be a valuable online tool, serving a complementary role to Nooks. Integrity checking could help distinguish local, isolated corruption from global problems, informing Nooks whether to reboot the system or to continue recovery with confidence. Integrity checking could also complement flyweight isolation, allowing the system to perform only minimal isolation and decide after the fact whether outside corruption had occurred. Integrity checking might even be a helpful debugging and analysis tool, shedding light on when and where corruption first appears.

7. Conclusion

Operating system reliability remains a serious problem. Over the past several years, much work has gone into improving the reliability of Windows and its device driver ecosystem. In spite of this, device drivers remain the number one cause of crashes and a principal source of other, less well quantified failures including hangs and loss of functionality. In this thesis, we investigated the viability under Windows of the Nooks approach to withstanding driver failures. As with Nooks on Linux, we implemented a best effort system for handling the most common driver failure scenarios, with the potential to make a drastic improvement in the overall health of the system.

Nooks for NT had essentially the same subsystem components as Nooks for Linux, with most of the implementation differences stemming from the vastly more complex operating system environment and driver ecosystem in Windows. While substantial work remains to demonstrate a commercially viable implementation of Nooks on NT, we believe our work has shown that the basic Nooks architecture is equally applicable to NT as it was to Linux. We successfully demonstrated the core functionality for wrappers and protection domains, resource tracking, garbage collection, and recovery. Challenges remain, but it appears that most of the basic problems can be overcome. Binary compatibility, though tricky, appears to be solvable. Driver-driver communication introduces new issues, but these issues can for the most part be handled. Isolation-induced out of memory cases cannot be solved perfectly, but with sufficient attention they need not cause unnecessary disruptions.

We also highlighted some of the key uncertainties and difficulties that remain. Synchronizing with the system in spite of the cross-cutting nature of Nooks, though tamed so far, remains a potential snag. Providing drivers direct access to user space addresses is an outstanding problem. Inferring proprietary communication relationships, though easily solvable with metadata, remains tricky for existing driver binaries. Finally, drivers that misuse and abuse the interfaces could furnish a long string of headaches. We are optimistic at the prospects for handling these challenges, however. At worst, it would mean only a subset of legacy drivers would be supported under Nooks, and driver writers

would need to make some changes going forward. We suspect the reality will be substantially better.

While a large amount of work remains to produce a commercially viable version of Nooks for Windows, the benefits are likely to be substantial. Even as crashes become less frequent with ongoing reliability improvements elsewhere, other Nooks-amenable problems such as driver hangs, power management and shutdown disruption, and loss of device functionality remain as serious as ever. We believe the argument for lightweight driver isolation and recovery is a persuasive one. Now, Nooks removes backward compatibility as an excuse.

Bibliography

[Bershad95] 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 *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995, pages 267–284.

[Butler02] Butler Group. Organizations lose five weeks in a year. *OpinionWire*. 04 April 2002.

[Chou01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Oct. 2001.

[Chou97] T. C. Chou. Beyond fault tolerance. In A. Somani and N. Vaidya, Understanding Fault Tolerance and Reliability. *IEEE Computer*, 30(4) pages 45-50, 1997.

[DV05] How to Use Driver Verifier to Troubleshoot Windows Drivers. Q244617, Microsoft Corp., 2005. Available at `http://support.microsoft.com/?kbid=244617`.

[Engler95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[Erlingsson05] U. Erlingsson, T. Wobber, P. Barham, and T. Roeder. VEXE'DD: Virtual EXtension Environments for Device Drivers. Available at `http://research.microsoft.com/research/sv/vexedd`.

[Fraser04] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On- Demand IT Infrastructure*, Oct. 2004.

[Gray86] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[Herder06] J. Herder, H. Bos, and A. Tannenbaum. A Lightweight Method for Building Reliable Operating Systems. Vrije Universiteit Technical Report IR-CS-018, January 2006.

[Hunt97] G. Hunt. Creating user-mode device drivers with a proxy. In *Proc. 1997 USENIX Windows NT Workshop,* Seattle, WA, Aug. 1997.

[LeVasseur04] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. *In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[Maffeo04] G. Maffeo and N. Ganapathy. Driver Hangs – Detection and Prevention. Windows Hardware Engineering Conference 2004. Slides available at `http://download.microsoft.com/download/1/8/f/18f8cee2-0b64-41f2-893d-a6f2295b40c8/DW04011_WINHEC2004.ppt`.

[Messer01] A. Messer, P. Bernadat, G. Fu, D. Chen**,** Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic. Susceptibility of Modern Systems and Software to Soft Errors. HPL-2001-43. Computer Systems and Technology Laboratory, HP Laboratories Palo Alto, 2001.

[Microsoft03] Microsoft Corp. Common Driver Reliability Issues. 2003. Available at `http://www.microsoft.com/whdc/driver/security/drvqa.mspx`.

[Microsoft06] Microsoft Corp. Windows Driver Foundation. Available at `http://www.microsoft.com/whdc/driver/wdf/default.mspx`.

[Microsoft05] Microsoft Corp. Kernel Enhancements for Microsoft Windows Vista and Windows Server Longhorn. 2005. Available at `http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kernel-en.doc`.

[Microsoft05_2] Microsoft Corp. Introduction to the WDF User-Mode Driver Framework. 2005. Available at `http://www.microsoft.com/whdc/driver/wdf/umdf_intro.mspx`.

[Oney02] W. Oney. *Programming the Microsoft Windows Driver Model, Second Edition*. Microsoft Press, 2002.

[Oshins04] J. Oshins and D. Holan. WDF - Overview of PnP and Power Management Model. Windows Hardware Engineering Conference 2004. Slides available at `http://download.microsoft.com/download/1/8/f/18f8cee2-0b64-41f2-893d-a6f2295b40c8/DW04036_WINHEC2004.ppt`.

[PREfast03] PREfast for Drivers. Microsoft Corp. Available at `http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx`.

[SDV06] Static Driver Verifier. Microsoft Corp. Available at `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`.

[Seltzer96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Oct. 1996, pages 213--227.

[Semack04] M. Semack. Linux's (Lack Of) Driver Architecture. 2004. Available at `http://www.semack.net/Articles/LinuxsDriverArchitecture.html`.

[Shah04] A. Shah. High Memory In The Linux Kernel. 2004. Available at `http://kerneltrap.org/node/2450`.

[Sullivan91] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In Proceedings of the 1991 Symposium on Fault Tolerant Computing (FTCS), pages 2-9. IEEE, June 1991.

[Swift04] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[Swift05] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.

[Wabhe93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Dec. 1993, pages 203–216.

[Wang04] Landy Wang, Distinguished Engineer, core operating systems division, Microsoft Corp. Private communication.

[Witchel02] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, Oct. 2002.