THE FLUKE DEVICE DRIVER FRAMEWORK

by

Kevin Thomas Van Maren

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1999

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Kevin Thomas Van Maren

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Frank J. Lepreau

John B. Carter

Wilson C. Hsieh

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Kevin Thomas Van Maren in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Frank J. Lepreau
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Providing efficient device driver support in the Fluke operating system presents novel challenges, which stem from two conflicting factors: (i) a design and maintenance requirement to reuse unmodified legacy device drivers, and (ii) the mismatch between the Fluke kernel's internal execution environment and the execution environment expected by these legacy device drivers. This thesis presents a solution to this conflict: a framework whose design is based on running device drivers as user-mode servers, which resolves the fundamental execution environment mismatch.

This approach introduces new problems and issues, of which the most important are synchronization, interrupt delivery, physical memory allocation, access to shared resources, and performance. We successfully addressed the functional issues, as demonstrated by the fact that the majority of device drivers execute successfully without change and are routinely used by Fluke developers. Based on our experience with the minority of drivers that did require changes, and our experience developing the framework, we propose guidelines for improving device drivers' portability across different execution environments.

Running device drivers in user mode raises serious performance issues but on the whole they were successfully mitigated. We compare the driver performance in Fluke with that in the original legacy systems, in terms of latency, bandwidth, and processor utilization. We find that reasonable performance (between 88–93% of the best-performing Unix systems in a realistic workload) and acceptable processor overhead (between 0–100%) are achievable. The limiting factor is the IPC performance of the underlying Fluke layers.

For my wife and kids

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

My thanks go to everyone who helped me get this far. In particular, I would like to thank Jay Lepreau, John Carter, and Wilson Hsieh for their support, for serving on my thesis committee, and most especially for their marathon efforts in helping me pull my thesis together at the end. My advisor Jay Lepreau deserves my thanks for hiring me as a research assistant and giving me the opportunity to prove myself. I would also like to thank Bob Kessler, who served on my committee when I began my thesis work.

Bryan Ford and Mike Hibler deserve special recognition for their work on Fluke, without which I would not have had the Fluke microkernel to use for my thesis project. My thanks extend to everyone else in the Flux group for their support and assistance over the last few years.

I thank my relatives, including parents, siblings, grandparents, and in-laws, for their support and encouragement, as well as for exerting pressure on me to complete my thesis and graduate. Finally, I would like to thank my wife, for her eternal patience and support while I worked on my thesis.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement numbers F30602–96–2–0269 and F30602–99–1–0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

CHAPTER 1

INTRODUCTION

This thesis describes and analyzes the device driver support in Fluke [20, 21], a research operating system (OS) developed at the University of Utah. We developed a framework that enables legacy device drivers to be used unmodified and still achieve reasonable performance. This work was a challenge due to the mismatch between Fluke's internal execution environment and the execution environment expected by the legacy device drivers. The rest of this chapter gives an introduction to device drivers, outlines the most important kernel execution environment issues that made this work challenging, and concludes with a summary of the overall thesis and the work's contributions.

1.1 Device Drivers

Device drivers are the parts of an operating system that communicate directly with the hardware and provide higher-level abstractions of the hardware for use by other software. Writing, debugging, and maintaining device drivers is a long and tedious process, especially given the enormous number of different hardware devices available.

Device drivers service requests from the operating system, usually driven by application requests. Drivers must also service requests from the hardware. By generating an interrupt, a device informs the driver it has data available or it has completed a task. Because requests are being sent to the driver from two "directions," device drivers are logically divided into a *top half*, which services operating system requests, and a *bottom half*, which services requests from the hardware, as shown in Figure 1.1. Since the top half executes in response to requests from applications, or processes, execution in the top half is also referred to as being

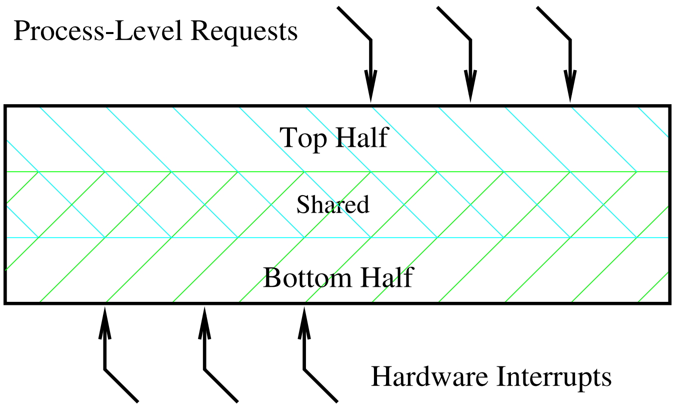

Figure 1.1. Top and bottom halves. The kernel processes requests from applications (top half), and requests from the hardware (bottom half). Synchronization between the top and bottom is necessary because of the state shared by the two halves.

at *process level*, while interrupt handling in the bottom half is referred to as being at *interrupt level*. As an example, the top half of a disk device driver adds read and write requests to a queue, while the bottom half removes the requests from the queue as the disk drive completes them.

Synchronization between the two halves of the driver can be accomplished in many different ways, depending on the operating system, but the device driver and the operating system must agree on the specific synchronization mechanisms. For example, some operating systems allow multiple top-half requests to be serviced in parallel with the bottom half. These systems use locking to coordinate the multiple outstanding requests. Other operating systems process only one request at a time and therefore do not require such locking, but still must coordinate the activity between the top and bottom halves. The particular synchronization mechanisms used are a crucial part of the kernel's execution environment.

Since device drivers are an integral part of the operating system, it is necessary for the device drivers to mesh with the rest of the operating system. This meshing is normally accomplished by writing device drivers that are tailored for the specific operating system environment. If device drivers are moved from one operating system to another, the kernel facilities may no longer match the device driver writer's expectations, which requires that the driver be modified for the new environment.

1.2 Kernel Execution Environments

An execution environment encompasses everything that is visible to the program. Execution environments include aspects such as the threading/concurrency model, the instruction set, the available libraries and routines and their semantics, and other available resources.

The threading model is defined by the semantics of synchronization, scheduling, and preemption. An important aspect of the threading model is determining when a thread can block, and what happens to its state when it does block. The state can remain on the stack, can be explicitly stored in a "continuation" [11], or may simply be discarded.

An important resource provided by an execution environment is memory: how it is allocated and managed, how it is mapped, and when it is paged. The available resources also encompass a notion of privilege, as the available resources generally depend on the privilege level of the process.

Whereas application execution environments are fairly standardized [26, 34, 64], the kernel's execution environment is determined by the kernel's implementation. Operating system kernels each have their own internal execution environment, which can vary from one operating system to another. Some of the ways in which they vary are discussed in the next few sections.

1.2.1 Blocking

A request blocks when it cannot be completely processed immediately, due to the need to wait for a resource or an external event, such as the completion of a disk read. Combined with the kinds of events that can cause blocking, the kernel's behavior upon blocking is a driving factor in the entire kernel's design and implementation.

Even though the specific events that cause blocking vary between operating systems, there are only a few models for how blocking is handled.

1.2.1.1 Blocking Model

In the traditional blocking model, every request has an associated kernel stack. When a request blocks, its state is preserved on that stack. When the request is allowed to continue, it can resume execution exactly where it left off, with all its state intact. This approach is used by Windows NT, VMS, and most Unix-like operating systems [42], as shown in Figure 1.2.

1.2.1.2 Nonblocking Model

With the "nonblocking" model, a request that blocks simply discards all of its kernel state. When the request restarts, none of the state associated with the request is still around, so the request must be able to continue without any state accumulated before it blocked. Since there is no state indicating progress, the system call is normally restarted from the beginning. Restarting the system call from the beginning allows the kernel to block in user mode just prior to the system call, instead of blocking in the kernel.

Not having to keep persistent state allows the kernel to use a single stack to process all of the requests, as the stack is only used while doing the actual processing of a request. The use of a single stack is shown in Figure 1.3. Nonblocking operating systems are generally written to avoid blocking, so they run system calls

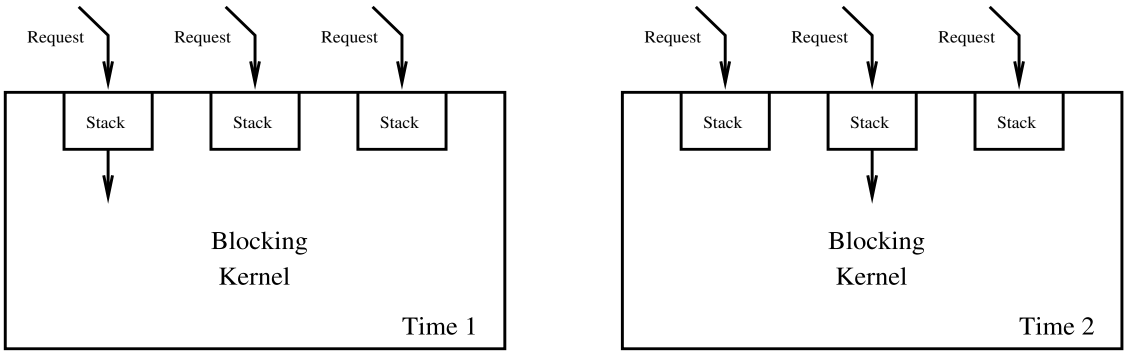

Figure 1.2. The blocking kernel configuration. In the left diagram at "Time 1," the left request is executing, while the other two requests that were previously executing are blocked. At "Time 2," as shown in the right diagram, when the left request blocks the kernel changes stacks to the center request and processes it. The blocked requests do not have to do anything special, as all of their state is retained on per-request stacks.

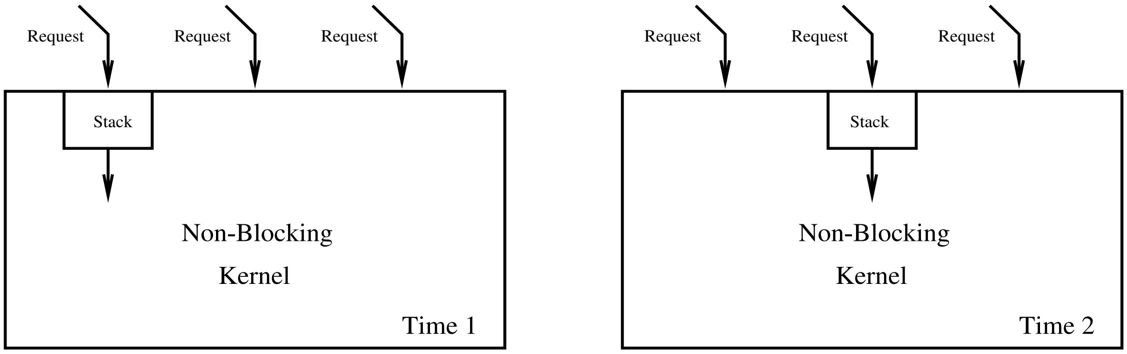

Figure 1.3. The nonblocking kernel configuration. The left request is executing, while the other two requests are blocked. When the left request blocks, all of its state is discarded. The center request gets the empty stack, and restarts its request from the beginning.

to completion. Operations that require blocking are generally handled in user mode when possible, or by having the kernel restart the system call from the beginning. The Cache Kernel [8] and the two exokernel implementations [15, 35] use only the nonblocking approach in their kernels.

1.2.1.3 Continuations

The continuation-based model falls between the blocking and nonblocking models. Before blocking, the kernel explicitly stores the request's state in a small continuation structure. When the request blocks, it loses its stack, as in the nonblocking model. Saving the state in a continuation structure allows the request to explicitly retain state across blocking. When the request is ready to resume, it receives a stack and the state is restored from the continuation structure. The use of continuations is shown in Figure 1.4. This approach was used by the V operating system [7] and in some parts of Draves' version of the Mach kernel [11]. Fluke also supports a continuation-based nonblocking model.

1.2.2 Synchronization

Within device drivers, synchronization between the top and bottom halves can be done using locks, by disabling interrupts, or through the use of atomic primitives. Traditional Unix kernels process system calls serially either to completion or until the system call blocks. A Unix kernel can simply disable interrupts in the top

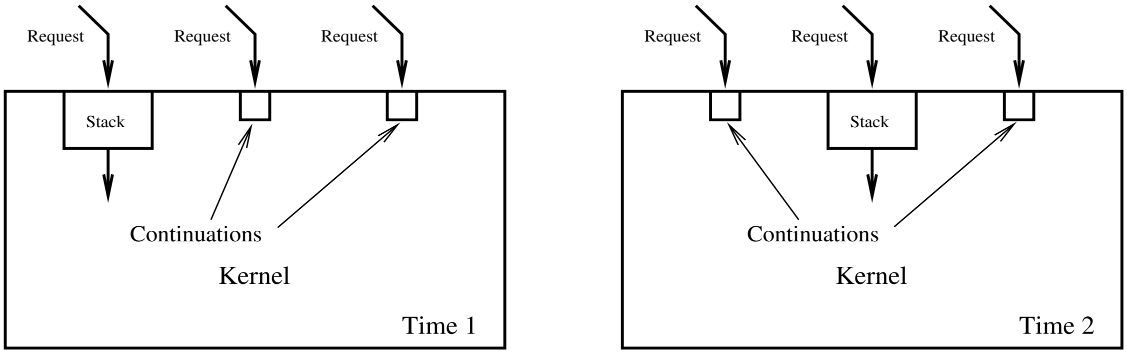

Figure 1.4. The continuation-based kernel configuration. The left request is executing, while the other two requests are blocked. When the left request blocks, its essential state is saved into a continuation structure, and another request is resumed by first loading the saved continuation information.

half to prevent the bottom half from running. By processing requests serially, the synchronization between process-level requests and the interrupt handler is fairly simple.

Operating systems that support multiprocessors can do so by either serializing system calls so that there is only one active at a time (Linux 2.0), or by multithreading the kernel (Solaris). Multithreaded kernels require sophisticated locking to coordinate between the multiple outstanding requests and the interrupt handler, as the interrupt handler may execute on a different processor than the activity that is attempting to synchronize with it.

1.2.3 Layering and Locking

The kernel's implementation can restrict the set of functions that may be called from a routine or impose a locking hierarchy that must be followed. Correctness requires that device drivers follow these rules. As an example, most operating systems do not allow the bottom half of the device driver to call blocking routines. This restriction is one reason it is important to know which routines block.

1.3 Making the Drivers Fit

The software development objective of this work was to take device driver components from the OSKit [18, 22], which provides device drivers that assume the blocking model environment, and run them in Fluke, whose kernel can use the nonblocking model. Although the blocking model mismatch is the most important concern, the mismatch between the execution environments is due to more than just blocking, as all of the previously mentioned areas come into play.

To achieve our goal of running OSKit device drivers in Fluke, it was necessary to provide an execution environment that resembled the monolithic OS kernel from which the device drivers originated. Since we found it impossible to run the device drivers in the Fluke kernel (due to the extreme mismatch of execution environments), we provide a framework for device drivers in Fluke by running them in user-mode processes. Running the device drivers in user mode makes them independent of the kernel's internal configuration and implementation, as they only need to deal with the exported kernel API.

1.4 Thesis Overview

1. As described in this chapter, we found that executing legacy device drivers in user mode is the only practical way to resolve the mismatch between their execution environment requirements and Fluke's internal environment. Chapter 2 gives more detail on the structure and internal execution environments of the software systems that provide the concrete context for this mismatch and its resolution: Unix, the OSKit, the Fluke kernel, and the Fluke runtime.

2. Running drivers in user mode raises numerous new issues and problems. We found four issues to be most important: synchronization, interrupt delivery, physical memory allocation, and access to shared resources. We discuss protection domain options for user-mode drivers and give background on these four issues in Chapter 3. The details of our solution for these issues are in Chapter 4.

3. A side effect of moving drivers to user mode is that doing so isolates them from the hardware, the kernel, and often, applications. This separation results in performance problems caused by several areas, including interprocess

communication (IPC), synchronization, and interrupt delivery. However, with effort we found that it is possible to achieve reasonable performance, which we quantify along multiple dimensions and driver placement option in Chapter 5. The limiting factor on performance is the IPC performance of the underlying kernel and, especially, the runtime. In Chapter 6 we discuss others' efforts to build user-level device drivers; notably, hardly any of the other efforts provides more than a superficial performance evaluation.

4. Our experience in developing this framework led to insights on some architectural advantages and disadvantages of Fluke and its runtime and allowed us to develop some modest guidelines for writing device drivers that are more portable across different execution environments, reported in Chapter 7.

Besides the above thesis contributions, the software artifacts that we developed in the course of this work contributed significant concrete benefits to the Fluke operating system and to the OSKit. Our software facilitated further research based on Fluke, including a thesis [9] and a paper [58]. The OSKit is a key part of several ongoing research and development efforts, such as MIT's ML/OS [57] and KaffeOS [2], and is widely used by others. The development work underlying this thesis provided the following benefits:

1. Device driver support in Fluke for dozens of Ethernet devices and disk controllers.

2. Device driver support for all of Fluke's kernel configurations (except the entirely non-preemptive configuration, which is of no practical use), with several options for device driver placement.

3. Support in Fluke for the OSKit's filesystem and network components, which require a subset of the driver execution environment.

4. Enhanced OSKit facilities in several areas, such as driver fixes and encapsulation of PCI configuration space.

CHAPTER 2

SOFTWARE SYSTEM CONTEXT

This chapter discusses relevant functional, structural, and execution model aspects of the software systems that define the context for our work. These software systems are Unix, which served as the original source of the OSKit device drivers, the OSKit, the Fluke kernel, and the three parts of the Fluke IPC runtime.

2.1 Unix

Our device drivers came from Linux [3, 54], a Unix-like operating system. This section describes the "traditional" Unix kernel execution model, used by the BSD Unix variants and Linux. Some newer commercial Unix variants, such as Solaris [28], have moved to a multithreaded kernel with different synchronization models.

The Unix kernel services synchronous requests (system calls) from user processes and asynchronous requests (interrupts) from devices [61]. When a Unix process does a system call, the processor transfers to a dedicated per-process stack in the Unix kernel. System call activity executes in supervisor mode at process level. Process-level activity is run to completion unless it blocks on a resource (such as a lock on a file descriptor), or on a hardware event (such as waiting for a disk transfer to complete). When the process-level request blocks, its state remains on the per-process stack inside the kernel, which waits until the request resumes. Only a single process-level request may be active inside the kernel at a time.

In addition to processing system calls, the Unix kernel also executes interrupt handlers in response to hardware interrupts. An interrupt handler executes in the bottom half of a device driver, which preempts the top half and runs without a true process context. Since it lacks a true process context, the bottom half is not allowed to block or access the user's address space and thus is processed atomically with respect to the process-level execution [42, Ch. 3]. The top half synchronizes with the bottom half by disabling interrupts, whereas the bottom half synchronizes with the top half by waking up blocked requests.

2.2 OSKit/Osenv

The OSKit [18, 22] is software from the University of Utah that provides single-threaded encapsulated components, such as networking code, filesystems, and device drivers. These components were taken from pre-existing monolithic Unix kernels (currently FreeBSD [25, 37], NetBSD [47], and Linux). The premise of much of the OSKit work is that many operating system components are large and complex, and it is too much work to continually reimplement the same functionality for each new research operating system. Therefore, the OSKit attempts to provide cleanly encapsulated code from existing operating systems which can be easily adapted to new operating systems. By reusing code from other operating systems, research efforts can be focused on the "interesting" parts of the OS, rather than being continually forced to reimplement all the necessary functionality.

To provide this encapsulation capability, the OSKit defines abstraction APIs and provides "glue" code to implement them. In particular, for each component the OSKit defines a set of "up side" Component Object Model (COM) [44] interfaces by which the client OS invokes OSKit services; the per-component OSKit glue code translates calls on the public OSKit "standard" interfaces such as `device_write` into calls to the imported code's idiosyncratic internal interfaces.

Most relevant to our problem of matching execution environments is the "down side" interface, through which every component invokes the services of its host operating system (e.g., memory allocation, interrupt management). For these services the OSKit defines a standard interface, that is called `osenv`. The "down side" glue code translates calls to low-level services made by the imported code into calls to the equivalent `osenv` public interfaces. Figure 2.1 is a diagram showing both sets of interfaces.

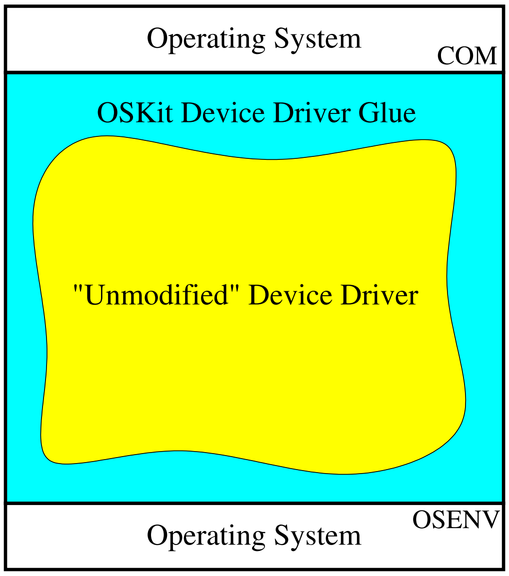

Figure 2.1. Diagram showing an OSKit device driver using an OS-provided `osenv` implementation and exporting a COM interface to the operating system.

Therefore, to import an OSKit component into an existing OS, the OS needs to provide an implementation of the `osenv` interface and know how to communicate with the OSKit's standard interface for that object's services.

`Osenv` defines both the functional interfaces and the required execution environment. The `osenv` API and execution environment specifications were driven both by the desire for simplicity and the blocking nature of the encapsulated code. Since all of the OSKit components come from a single-threaded Unix environment, `osenv` closely follows the Unix execution model. The `osenv` execution model has the following requirements: (1) only one process-level request may be in the code at any point in time; (2) requests must be processed until they complete or yield the processor; (3) interrupt handlers may not execute while interrupts are "disabled"; (4) interrupts must appear to be atomic to the top half; and (5) per-request stacks must be preserved across blocking operations, such as `osenv_sleep`.

`Osenv`'s functional interfaces, which deal with interrupts, memory, timers, and other hardware-specific interfaces, are discussed in detail in the Appendix. Complete `osenv` documentation is available as part of the OSKit distribution and can also be found on the Web [16].

2.3 Fluke Kernel

The Fluke kernel supports many different kernel configurations and execution models [20]. Since Fluke is a research operating system designed from scratch, its implementation was not constrained by the necessity of supporting any existing code or execution environments. In fact, one of the research areas explored by the Fluke kernel implementation was using a variety of unusual execution models for uniprocessor and multiprocessor machines. The Fluke kernel is fully multithreaded, and is able to process multiple requests concurrently, in contrast to the single-threaded Unix kernel.

The Fluke microkernel was designed with a fairly minimalist approach: the kernel provides only basic low-level objects, such as threads, mutexes, condition variables, interprocess communication (IPC), and memory mappings. Applications use these low-level primitives for multithreaded synchronization and communication. Any functionality not directly provided by the kernel, such as demand paging of memory and process management, must be provided by application-level servers. A subset of POSIX, including pthreads, is implemented using these low-level primitives. A detailed specification of the Fluke API can be found on the Flux Web pages [19].

2.3.1 Behavior Upon Blocking

The Fluke kernel's execution environment configurations do not meet the Unix driver's execution environment assumptions. In this section we discuss the major problems with using Unix components in both the nonblocking Fluke kernel and in the preemptive blocking Fluke kernel.

2.3.1.1 Nonblocking Configurations

In Fluke, much like in Unix, blocking can occur only at well-defined preemption points, such as when attempting to acquire a lock or encountering a page fault. Unlike in Unix, in some Fluke kernel configurations the stack is discarded upon blocking. The continuation information is saved by modifying the thread's system call state to represent any progress that has been made. The ability to represent the continuation in the system call state is possible because of a carefully selected kernel API [20], where the kernel exports the intermediate stages of long-running system calls.

When the thread is awakened, the system call is restarted. Where the thread restarts the system call depends on how much (if any) recordable progress was made before the thread blocked. By restarting the system call, the kernel, in effect, blocks the thread in user mode. Since the stack is discarded and Fluke does not require a separate continuation structure, this environment is referred to as the *nonblocking model* in Fluke.

The OSKit device drivers, which were written for Unix, cannot run without dedicated stacks, so we cannot use them in the nonblocking kernel configuration. One solution would be to heavily modify existing drivers, while another solution would be to write new device drivers. Neither of these solutions is practical.

2.3.1.2 Blocking Configurations

Alternatively, the Fluke kernel can be configured to run as a multithreaded, fully-preemptive kernel with per-thread stacks. In such an environment, system calls can be arbitrarily interrupted without blocking. System calls are not guaranteed to run to completion, as the kernel can arbitrarily preempt them. Providing preemption and multithreading violates the Unix execution environment's assumptions.

2.3.2 Strictly Layered Implementation

Fluke's highly layered kernel implementation poses another problem for device drivers. There are approximately 30 layers in the current Fluke implementation [17]. A layer may only call down to lower layers, and may not call up. Violating the layering model by having lower layers call higher layers may yield incorrect results or, more likely, lead to deadlock. Despite these sometimes awkward constraints, there are some good design reasons for highly-layered systems. Writing the kernel in a layered manner simplified the implementation and locking, which made it practical to support multiple kernel configurations. Since code can only call lower layers, call cycles have been eliminated, which in turn helps guarantee it is deadlock-free. Another advantage is that the layering made the code more assurable; for example it facilitated the verification of the Fluke IPC path, independent of the other components [60]. Assurance is a particularly relevant goal for Fluke because one of its configurations is the "Flask" high-security system [58], developed jointly with the NSA. There is precedent for such layering; OSF's MK++ kernel [39] was also highly layered, strictly due to its high-security goals.

Although the layering offers advantages from a software engineering viewpoint, it also violates the assumed execution environment of the Unix interrupt handlers. Interrupt handlers are run very low in the Fluke layering: just high enough to perform a context switch, but not high enough to allocate memory. Unfortunately, the Unix device drivers to be used in Fluke rely on allocating memory while handling an interrupt, which Fluke's strict layering prevents.

The layering also restricts access to synchronization mechanisms needed by the interrupt handlers. Since an interrupt handler is run on the current thread's stack, it cannot do a `mutex_lock`, as the thread owning the mutex may be the same thread whose context the interrupt thread is using. Interrupt handlers likewise cannot manipulate any Fluke objects, look up memory mappings, or do much of anything else. As a result of these restrictions, about all an interrupt handler *can* do is cancel or dispatch a thread.

These layering restrictions prevent us from running legacy drivers' interrupt handlers in Fluke interrupt handlers. In order to work around the layering restrictions, we have found it convenient to have the interrupt handler wake up a thread, which then executes the real interrupt handler.

2.4 Fluke IPC Runtime

The standard Fluke runtime is used by nearly all programs to perform IPC. The runtime consists of an IDL compiler (Flick), a library that provides a multithreaded object invocation layer atop the kernel and Flick-generated stubs (MOM), and a set of macros that adapt MOM and Flick to the single-threaded COM objects exported by the OSKit.

The runtime is needed for many reasons, many of which are mentioned in the sections below. One important reason is that directly invoking Fluke kernel IPC system calls is tedious and error-prone; the runtime provides a clean IDL-specified interface. However, as we will see in later chapters, in gaining this abstraction the programmer currently loses some of the flexibility of the kernel IPC mechanism, which leads to performance problems in the device driver framework.

2.4.1 MOM Runtime Support

Above the Fluke kernel is an important runtime layer implementing the "Mini Object Model," called MOM [41], used by Fluke operating system servers and the Fluke client-side libraries for IPC-based communication. Besides simple object invocation, MOM provides reference counting and a fully multithreaded dispatch server on top of IPC stubs generated by the Flick IDL compiler [14]. This relatively sophisticated runtime was designed to support multithreaded Fluke servers handling multiple concurrent requests from multiple clients.

2.4.2 Flick IDL Compiler

Flick, the Flexible IDL Compiler Kit [14], is a flexible Interface Definition Language compiler developed at the University of Utah. In Fluke, it is used to generate RPC stubs from the CORBA IDL [48] specification of the IPC interfaces. Although Flick does highly-optimized message marshaling, interactions between MOM and Fluke IPC introduce several additional data copies. Since MOM does not know what needs to be done with the data, it receives the entire message into a temporary buffer before handing the data off to Flick. Flick then copies the data into a buffer more suitable for the application. Work is in progress that will allow an outgoing data buffer to be sent directly by the Fluke scatter-gather IPC. However, incoming buffers are still copied out of the temporary storage provided by MOM. Even though this copy is not always necessary, it does allow the semantics to be preserved in the future if data is received directly into different buffers.

2.4.3 COM Wrappers

COM, or the Component Object Model [44], is intended to provide a way for independently developed components to be easily assembled in a program. Most OSKit components, including the device drivers, export a COM interface. The OSKit defines several COM interfaces, which provide a standard interface for each type of component, such as network device driver, regardless of where the component originated or how it is implemented. COM allows a Linux disk driver to have the same interface as a disk driver taken from FreeBSD, for example.

Since many of the components used in Fluke are based on OSKit COM objects, a set of macros has been developed that allow Flick-generated stubs to call COM object methods without the tedious duplication of server dispatch functions. Because of the single-threaded nature of the COM objects from the OSKit, these macros use either Fluke mutexes or special locking functions to serialize requests. Although this locking can introduce significant overhead, it is necessary to guarantee correct execution. By having the COM wrappers use the `osenv` process locking functions (Section 4.3), the serialization of requests is done automatically.

CHAPTER 3

DESIGN ISSUES

Designing a device driver framework is a complicated matter because of issues like synchronization, interrupt delivery, memory allocation, and shared resources. Fully specified device driver frameworks like UDI [56] contain hundreds of pages detailing the functional interfaces and their interactions.

Fortunately, our problem is constrained by the requirements of the encapsulated `osenv` device driver components in the OSKit. Although this chapter focuses on the design issues involved in implementing a user-mode device driver framework for the `osenv` execution environment, much of the following discussion is more broadly applicable. Because the Fluke device driver framework runs in user mode, we start this discussion with an overview of the device driver protection domain placement options. We then discuss the major issues of synchronization, interrupt delivery, memory allocation, and shared resources.

3.1 Protection Domains

In a traditional monolithic operating system such as Unix, all of the core OS functionality, including device drivers, is provided by a single program called the kernel, with applications running in separate address spaces. With these monolithic systems, device drivers and other components are added to the OS either by building a new kernel, or by dynamically linking them into the running kernel. Some microkernel operating systems have taken a different approach, and place device drivers and other OS components in their own separately protected address spaces. These operating systems add new functionality by simply running another server process, rather than modifying the kernel.

The design space for device drivers and other operating system components is much larger than the above two choices. The large size of the design space is based on the observation that hardware memory mappings can be set up in many different arrangements and on the further observation that memory mappings and being in *supervisor mode* are orthogonal issues. Privileged instructions are accessible only from supervisor mode, which is why the kernel runs in supervisor mode and applications run in user mode.

Figure 3.1 shows four possible arrangements, with arrows used to indicate the resulting communication paths between an application and a device driver. The model labeled "A" represents a traditional Unix-style arrangement, in which the device driver is in the kernel. Here the protection boundaries correspond directly to the privilege level.

In "B," the device driver is still in the kernel's address space, but it executes in user mode instead of supervisor mode. Here the user-level code is given the same page mappings as the kernel code. The device driver can directly access kernel data structures, but the device driver can no longer directly execute privileged instructions, since it is not in supervisor mode.

Model "C" represents an operating system which has the device driver as a separate user-mode server. This model is the traditional user-mode device driver arrangement, and is used by operating systems such as QNX [31, 52]. Although this configuration may sound radical, it is also used by Unix daemon processes, which extend the OS functionality. The device drivers are still protected from the applications, and they are in separate protection domains from the kernel.

In "D," the application contains the device driver, and can communicate with the device without operating system intervention or interprocess communication (IPC). Colocating the application with the device driver is a desired feature for very high-speed and low-latency network communications [6, 63].

A fifth example ("E," not shown because it is obvious) is the case in which there is no distinction made between user and kernel code. The kernel, application, and device driver are in a single address space and protection domain, so there are

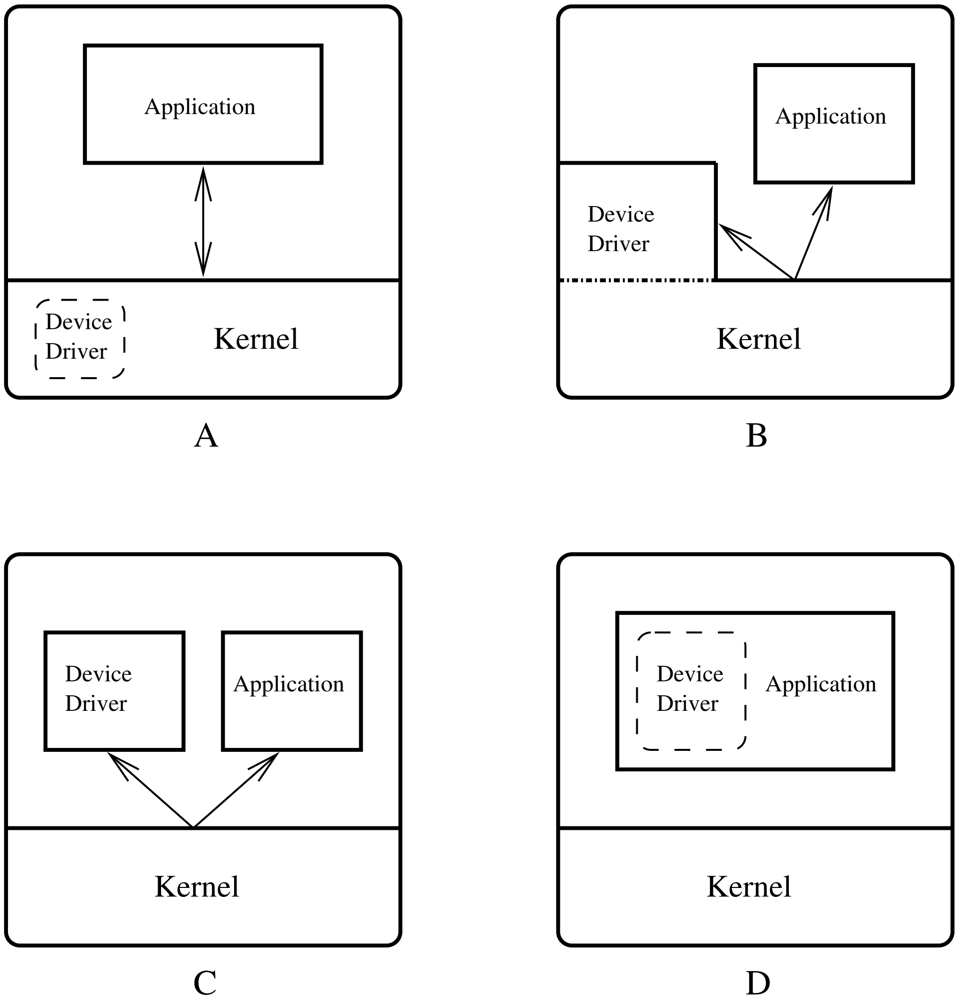

Figure 3.1. Device driver placement options. This figure shows device drivers in the kernel (A), in user mode with the kernel's address space (B), in a separate process (C), and colocated with an application (D).

no solid interior lines. Many embedded operating systems, as well as MS-DOS, take this approach. The OSKit allows applications to link themselves with OSKit libraries and device drivers to provide an operating system with this model.

Our Fluke device driver framework provides the ability to run device drivers as shown in Figure 3.1 "B," "C," and "D." As noted previously, arrangement "A" is not supported in Fluke. In the performance section of our thesis, we compare the performance of these arrangements in Fluke against the performance of Unix systems using model "A" and against the performance of the OSKit using a single protection domain ("E").

3.2 Synchronization

Proper synchronization between concurrent system calls and interrupt handlers is a critical aspect of a device driver framework. A device driver receives requests from two different directions: the top half receives requests coming from multiple application threads, often concurrently, while the bottom half executes in response to hardware interrupts. Access to shared data structures must be mediated in some manner to prevent concurrent accesses finding these shared resources in an inconsistent state.

The `osenv` model, as discussed in Section 2.2, imposes the following synchronization constraints: (1) only one top-half (process) thread may be in the code at any point in time; (2) the bottom-half (interrupt) threads may not execute while interrupts are "disabled"; and (3) interrupts must appear to be atomic to the top half. A process-level thread may choose to yield the CPU explicitly, thus logically exiting the driver and allowing another process-level thread to enter the driver. However, the thread that does the yield must then compete with other threads to reenter the driver when it wishes to continue. Although not strictly required by the `osenv` model, it is also necessary that an interrupt thread be able to preempt a process-level thread. This additional constraint is necessary because the legacy drivers we use often spin at process-level on a condition set by an interrupt handler (such as delaying for a short time by spinning on the "current time" variable).

These constraints are due to the nature of the current `osenv` device driver components. For example, interrupts must be processed atomically because the interrupt handler may modify state being accessed by the process-level thread. Process-level threads do not need to disable interrupts while reading the state if the interrupt handler ensures that the state is consistent upon exit. A common example of this would be an interrupt handler adding an element to a linked list used by the top half of the device driver.

Since these constraints are driven by the nature of the Unix device drivers used, they could be adjusted if we used different device drivers, e.g., multithreaded device drivers. If the device driver writer were willing to do more explicit synchronization, a barrier mechanism could be used to synchronize the top and bottom halves. Although explicit synchronization may be more efficient when operating in some environments, such as when it is very expensive to guarantee interrupt atomicity, the cost of this type of synchronization would most likely be greater for monolithic uniprocessor device drivers.

It is also possible to write a device driver that handles synchronization between multiple top-half threads internally, thus allowing concurrent access to the driver. Concurrent access is especially advantageous when running on a multiprocessor machine. However, synchronizing multiple process-level threads and interrupt processing would be more complicated than the simple "disable interrupts" model currently assumed by the `osenv` framework.

3.3 Interrupt Delivery

Most device drivers use interrupts to synchronize between events in the device driver and events in the hardware. Hardware often takes a long time to accomplish a task, such as reading a disk block, so it is a good idea for the OS to perform other tasks while the I/O hardware is busy. Additionally, some events, such as incoming network traffic, are asynchronous by nature. Polling for either type of event is inefficient in most situations. Since the device driver needs to know when its device requires servicing, the device generates an interrupt to signal the device driver to do more processing.

Interrupts are relatively expensive and are delivered in supervisor mode to a hardware interrupt handler. It is the responsibility of this handler to call the interrupt handler for any device drivers using the specified interrupt vector. In monolithic kernels, calling the interrupt handler is simply a function call, once the low-level handler sets up any required state. This design provides reasonably low interrupt processing overhead. With user-mode device drivers, however, the kernel must instead cause the interrupt to be seen by the device driver in user mode. Delivering interrupts to user mode is normally done by sending a software signal of some sort to the device driver process.

One of the biggest drawbacks to user-mode device drivers is the cost of signaling interrupts, due in part to the necessity of doing an extra context switch to the driver on each interrupt. However, people are investigating ways to make the delivery of interrupts to user mode more efficient [59]. Additionally, some custom-written user-mode device drivers use a split-driver model, where the interrupt handler is loaded into the kernel and executes in supervisor mode, and the process-level work is done in user mode. The Exokernel allows applications to download code into the kernel in a similar manner [15]. We do not have that option, since we reuse existing drivers.

3.3.1 Interrupt Mitigation

Most network interfaces generate an interrupt for every incoming packet. Since most operating systems process interrupts at high priority, it is possible to spend most or all of the processor's time handling interrupts during periods of heavy network traffic. This problem is well-known, and has historically been addressed by doing as little as possible in the interrupt handler [12]. Other solutions reduce the number of interrupts generated by "coalescing" the interrupts with special hardware. For example, ATM interface cards reassemble the ATM cells before delivering an interrupt. User-mode device drivers minimize the amount of work done at high priority in the kernel by having the kernel's interrupt handler simply signal a user-level thread.

The frequency at which a user-mode device driver processes interrupts is inversely proportional to overall system load, because the interrupt thread will be scheduled less frequently during periods of heavy load. This scheduling behavior results in low latency under light load, while maximizing throughput under heavy load. Processing interrupts in user mode enables the processor to run the tasks that are currently more important than the interrupt handler. Processing interrupts in user mode also allows interrupt mitigation on incoming Ethernet packets without any special hardware. When the system load is low, the interrupts get handled promptly. As the system load increases, more packets are handled per interrupt, which maximizes throughput under load, as fewer interrupts are handled per packet arrival. Interrupt-driven kernels, by contrast, process more interrupts per second under heavy load.

Interrupt mitigation techniques are possible because most Ethernet devices support a chain or ring of buffers into which they receive packets. The device will fill them without intervention until the buffers are full. By ensuring that there are enough buffers to receive packets under the anticipated worst scheduling behavior, no packets will be dropped. This design requires more buffer space, either on the device or wired by the driver. Under extremely heavy load, higher priority processes can cause packets to be dropped, which is the desired behavior as the software would drop the extra packets anyway.

The one case where interrupt mitigation would not work well is with "dumb" serial ports, such as the 16550 UART [43]. Dumb UARTs have very small buffers, 12–16 bytes, and at 115,200 bps, interrupts must be processed at least 900–1200 times per second to not drop characters. Older UARTs, such as the 8250, do not have a FIFO and only buffer a single character.

Despite the difficulties posed by dumb UARTs and similar devices, it is still desirable to run their drivers in user mode. An obvious solution, in this case, is to use an intelligent serial card that has a much larger buffer (on the order of kilobytes, instead of bytes). This avoids the problem by not using hardware that exhibits the problem. Alternatively, the user-mode interrupt handler for dumb UARTs could be run at a higher priority, and immediately dispatched from the kernel. To prevent possible live-lock and to guarantee fairness, it would be necessary for the user-mode interrupt handler to disable the device interrupt once its receive buffer became full. Upon hitting a low-water mark, the driver would then reenable the interrupt vector. Disabling interrupts when the software buffers become full was first proposed as a mechanism to eliminate receiver livelock in an interrupt-driven kernel [45, 46]. Emulating larger buffers, at the cost of additional processing time at high-priority, is similar to the VAX, which did pseudo-DMA by having a lightweight, high-priority interrupt handler deal with hardware that cannot do DMA [10].

A final solution would be for the kernel's interrupt handler to copy the data from the FIFO to a circular queue. Performing the critical processing in the kernel's interrupt handler avoids the problem of user-mode interrupt latency. Even though this arrangement requires running the interrupt handler in the kernel, the rest of the processing can be still done in user mode. However, the task of synchronizing the kernel's handler with the user-mode processing then becomes more complex.

Deferred processing of interrupts is advantageous, because the device interrupts should not have a priority inherently higher than other work being done on the system. Interrupts should only get handled when the device processing becomes the most important task. With some form of priority inheritance, which causes a thread doing work on another thread's behalf to receive the priority of the original thread, it is possible to prevent priority inversion as well. The device driver would assume the priority of the highest-priority client thread waiting on it. Minimizing priority inversion also requires early demultiplexing, which minimizes the work done before it is known for whom the work is being done.

It is possible to implement deferred interrupt processing in a monolithic kernel by reducing the priority of processing a hardware interrupt. Hardware interrupts are normally run at the highest priority, which can lead to receiver livelock [12]. Deferring of interrupts is actually already done to some extent; Windows NT has deferred procedure calls (DPCs) [62], and Unix has software interrupts [61]. Both of these mechanisms minimize the work done in the interrupt handler itself, but interrupt processing still executes at a higher priority than user-mode code.

3.4 Memory Allocation and Usage

Memory allocation is an important consideration for most applications. Device driver memory allocation is different in that drivers normally require access to the physical addresses of buffers, because devices use physical addresses when accessing memory directly through DMA.

Device drivers do not always use physical addresses, however, as some devices do not directly access memory, and some devices use virtual addresses to access memory, such as the VAX/780 and SBUS devices, which use a memory management unit (MMU) on the I/O bus [10, 40]. The use of an I/O MMU does not necessarily make the job any easier for user-mode device drivers, as memory still be made nonpageable. Additionally, the I/O MMU must be programmed with the correct mappings, which is not necessary for devices that access physical addresses.

Devices that do not access main memory directly are a good match for user-mode device drivers. Without a DMA engine writing to main memory, the kernel can be assured that the device driver will not overwrite critical parts of memory. However, more and more devices are using DMA to transfer data because of the lower CPU overhead required compared to programmed I/O.

3.4.1 Paging

In most monolithic kernels, the entire kernel and all its data are stored in nonpageable memory, because the kernel cannot handle an unexpected page fault. Even if pageable kernel memory is supported, it is still generally restricted to a few subsystems.

Pageable memory offers certain benefits to drivers, but at the price of some new problems. For device drivers, only the buffers that are being accessed by the device *must* be in nonpageable (wired) physical memory. With device drivers in user mode, it should be possible to page out the rest of the driver's address space under periods of high memory consumption. Obviously, one cannot page out the driver doing the paging. Although a page fault in a device driver will adversely impact performance, it may be desirable to page out device driver memory that is infrequently accessed to provide more system resources to other components and applications. The disk I/O necessary to handle a page fault generally takes about 10ms, which is comparable to the scheduling quantum on many systems. A user-mode device driver that can tolerate an occasional 10ms delay in processing should be able to tolerate infrequent page faults, as well as some delay in processor scheduling. Even though it is easy to make the scheduling quantum smaller or to improve the scheduling latency, it is not as easy to reduce the page-fault penalty. Unfortunately, it is not always possible for the device driver to tolerate such high latency. For example, when a network device driver is subject to periods of heavy paging, network packets would probably be dropped. Dropping packets under excessive load is actually desirable, as the packets would most likely have be dropped by the higher layers after additional processing [45].

With the existing `osenv` device drivers, there is no way to clearly distinguish memory that must be wired from normal memory allocations. The device drivers do not make the distinction when they allocate memory, since they were extracted from a monolithic, nonpageable environment. However, it is possible to distinguish wired from normal memory allocations for some data allocations. Thus the current `osenv` model allows paging of the initial `text`, `data`, and `bss` segments,[1] and some dynamically allocated memory.

It would be possible to page out even device buffers, provided the pager notified the device driver, and the device driver could deal with those notifications properly. However, modifying the device drivers in the `osenv` framework to support paging notifications is beyond the scope of this thesis.

3.5 Shared Resources

Accesses to resources shared by multiple device drivers must be arbitrated by the operating system. In a monolithic kernel, it is often the case that the operating system chooses not to restrict access to a shared resource after the OS has granted a driver permission to use the resource. For example, once a Linux device driver has allocated an ISA DMA channel, it may freely program the DMA controller, since the non-preemptive single-threaded nature of the kernel protects the controller from concurrent access.

Although this scheme works well in single-threaded monolithic kernels, it does not work with multiple address spaces, as the serializing properties only hold for a single address space. Since one of the goals of user-mode device drivers is to be able to run multiple single-threaded device drivers concurrently, it is necessary to deal

[1] Several of the SCSI drivers access SCSI scripts in their data segment, and do not function properly without modification unless their data segment is wired and they know the physical location of the scripts.

with shared resources in a safe manner, which requires providing synchronization at the interface used for access.

It is not always possible to provide safe resource sharing between device drivers in different address spaces without unduly restricting parallelism or making significant modifications to the device driver. For example, the ISA DMA controller cannot be used by device drivers in different address spaces, because the original Linux code accesses the DMA controller directly rather than through a serializable interface. Other resources, such as the PCI configuration space control registers and the interrupt controller can be safely shared as the drivers use an interface that can be safely serialized. Device drivers that share resources without external serialization must be run in the same address space to maintain the required sharing semantics.

3.6 Summary

This chapter described some of the issues involved in the design of a user-mode device driver framework and explained how using existing Unix device drivers restricts this design. In particular:

1. There are multiple address space options.
2. Proper synchronization requires that only a single active process thread and a single active interrupt thread may be inside the driver at a time, and the interrupt thread stops the process thread.
3. We dispatch interrupts to user threads for processing, rather than calling an interrupt handler function in the kernel.
4. We must have an interface that lets drivers allocate nonpageable physical memory.
5. We must serialize access to shared resources.

CHAPTER 4

IMPLEMENTATION

This chapter contains a detailed description of the implementation of the Fluke device driver framework, including the device server, controlling access to I/O devices, Ethernet support, and logging. However, the bulk of the chapter, Sections 4.3 through 4.8, contains details on the five most important issues covered at a high level in the previous chapter: synchronization, voluntary blocking, interrupt delivery, memory allocation, and access to shared resources. These implementation details will be useful for those readers facing a device driver implementation challenge similar to Fluke's, but can safely be skipped by most others. We now give an overview of the driver framework before delving into details.

4.1 Device Driver Framework Overview

An overview of the Fluke device driver execution environment is as follows. Device drivers are run in user-mode processes. Process-level threads handle requests from applications, and interrupt threads are dispatched by "signals" from the kernel. Activity between the threads is synchronized through the use of mutex variables.

To provide the required execution environment for the device drivers, it was necessary to provide wrappers both between the incoming requests and the drivers, and between device drivers and the underlying Fluke execution environment. The wrappers are illustrated in Figure 4.1, which shows the seven layers of the resulting system.

Using MOM, the device driver framework exports an IPC-based interface to users of the device drivers. Incoming requests (from the top of the figure) are coordinated with each other and with interrupt activity through the use of Fluke

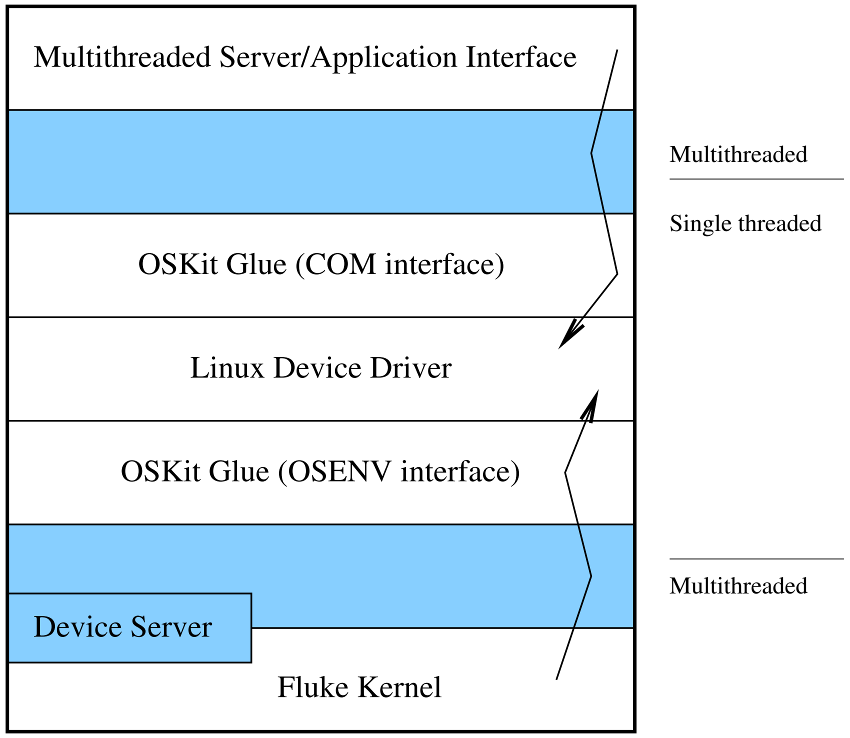

Figure 4.1. Seven layers in the Fluke device driver framework. The shaded areas represent the code written to allow the legacy device drivers to execute in Fluke. Application requests come in the top interface and are sent down to the device driver, while interrupts are delivered up into the device driver from the bottom. The middle layers are single-threaded while the outer layers are multithreaded. The device server is discussed in Section 4.2.

synchronization primitives which serialize the multithreaded MOM environment for the device drivers. In the driver framework, incoming requests are handed off to server threads, which run at process-level in the device driver. Interrupts are processed by special dedicated threads, one for each allocated interrupt vector; the kernel sends a signal to the appropriate thread when a hardware interrupt occurs.

It is possible for an application colocated with the device driver to use the underlying OSKit interfaces instead of or in addition to the IPC-based interfaces. Colocated applications can have either exclusive access to the device, or shared access with other applications accessing the device through IPC.

Even though the driver framework was primarily designed to support the `osenv` environment, most of the framework is also intended to be directly usable by native multithreaded Fluke device drivers. Native device drivers would differ most from the legacy device drivers in that we could tune their interfaces for IPC-based communication, rather than using the more general OSKit interfaces. Native drivers would also use a synchronization strategy designed for the multithreaded Fluke environment, and would not require external serialization of incoming requests.

We now discuss how the driver-specific issues were solved for the `osenv` execution environment, including synchronization, memory management, interrupt delivery, and other resources specific to device drivers. We begin this discussion with the device server.

4.2 Device Server

Although much of the support code necessary is in the driver application, additional external support is required to support user-mode device drivers. This support is provided by the *device server*, which executes in user mode in the kernel's address space as part of the *kserver*.

The *kserver* is very much like a normal Fluke application, except that it has the same memory mappings as the kernel, which corresponds to option "B" in Figure 3.1. Since the kserver is in the same address space as the kernel, memory allocation must be be coordinated with the kernel. We do that by providing a special system call to the kserver, which it uses to allocate memory.

All services that device drivers require that are not available to normal Fluke applications, such as allocating physical memory, receiving device interrupts, and allocating I/O space, are handled by the device server. The device server also mediates access to other shared resources, such as PCI configuration space and the interrupt controller. For example, the device server has access to special kernel hooks that allow it to register a thread to receive hardware interrupts and allocate physical memory for device drivers.

4.3 Synchronization

Before delving into the implementation, some background on the Fluke kernel interfaces is necessary. Fluke provides both POSIX-like [34] mutexes and condition variables [5]. Mutexes provide exclusive access to a region, while condition variables are used to wait until signaled by another thread. Condition variables are used with a mutex for synchronization. This mutex is automatically released while waiting on the condition variable, and is atomically reacquired by the thread when the condition variable is signaled. Fluke also provides a mechanism to send an asynchronous signal to another thread through the use of *thread interrupts*.

Synchronization in the Fluke device driver framework is an especially tricky aspect of the implementation. Synchronization between process-level (top half) activities and interrupt-level (bottom half) activities is done exclusively with Fluke mutex and condition variables; the device driver never disables actual physical interrupt delivery to the processor. Before entering the driver code, process-level threads acquire a mutex (`fdev_lock`). Similarly, interrupt threads acquire a different mutex (`intr_lock`) before executing. Interrupts can be disabled by the process-level thread simply by acquiring `intr_lock`. Figure 4.2 shows the process involved in acquiring and releasing the process-level lock. Figure 4.3 shows the steps required to enable and disable interrupts.

Since interrupt processing must be atomic with respect to process-level activity, the interrupt thread must also stop any process-level thread executing in the top-half code for the duration of the handler. To protect against the race condition in which the process-level thread changes while the interrupt thread is determining which thread to stop, the process-level thread may acquire or release the process-level lock (`fdev_lock`) only while holding the interrupt lock (`intr_lock`). This allows the interrupt thread to easily identify which thread (if any) it needs to stop before it can execute the device driver's interrupt handler.

This restriction on when the process lock can be acquired or released brings about additional complications. Threads may not block trying to acquire the process-level lock, as they would block holding the interrupt lock. If they blocked while holding the interrupt lock, the thread holding the process lock would not be able to release it, and interrupt threads would also be blocked, resulting in deadlock.

To avoid race conditions, if the process-level lock cannot be immediately acquired, the thread uses a condition variable (`process_cond`) in conjunction with

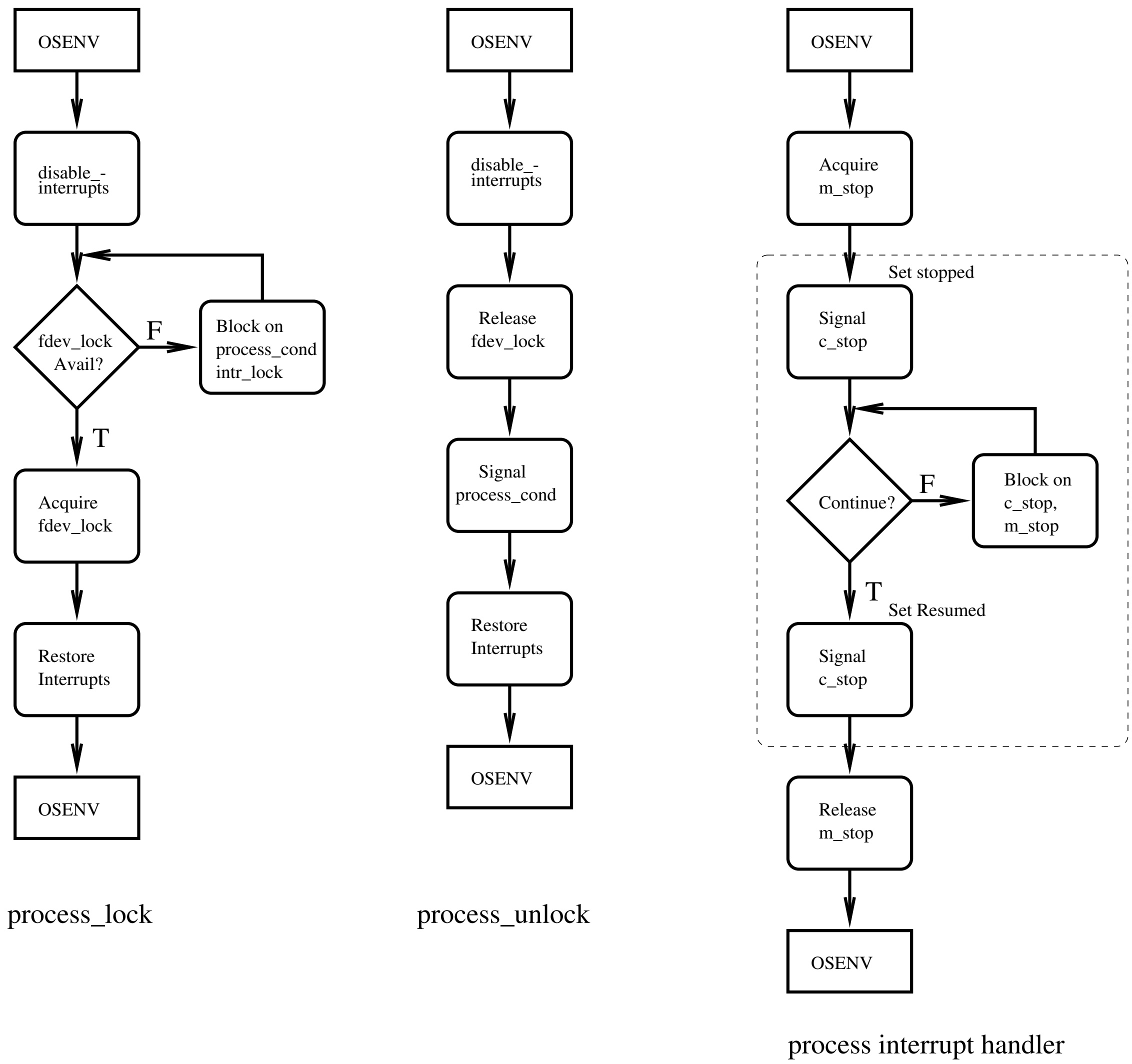

Figure 4.2. Simplified diagram showing `process_lock` and `process_unlock`. The signal handler, used to make interrupt handlers atomic with respect to the process-level thread, is also shown. The signal handler is closely related to `enable_interrupts` and `disable_interrupts`, which are shown in Figure 4.3. Restore Interrupts calls `enable_interrupts` only if `intr_lock` was not held when `disable_interrupts` was called. Process Thread returns `TRUE` if the current thread is a top-half thread.

the interrupt mutex. When the currently running thread is ready to release the process lock, it signals the condition variable to wakeup the blocked thread waiting for the process-level lock. The blocked thread is able to resume execution when it can reacquire the interrupt lock. It then acquires the process lock to become the new process-level thread.

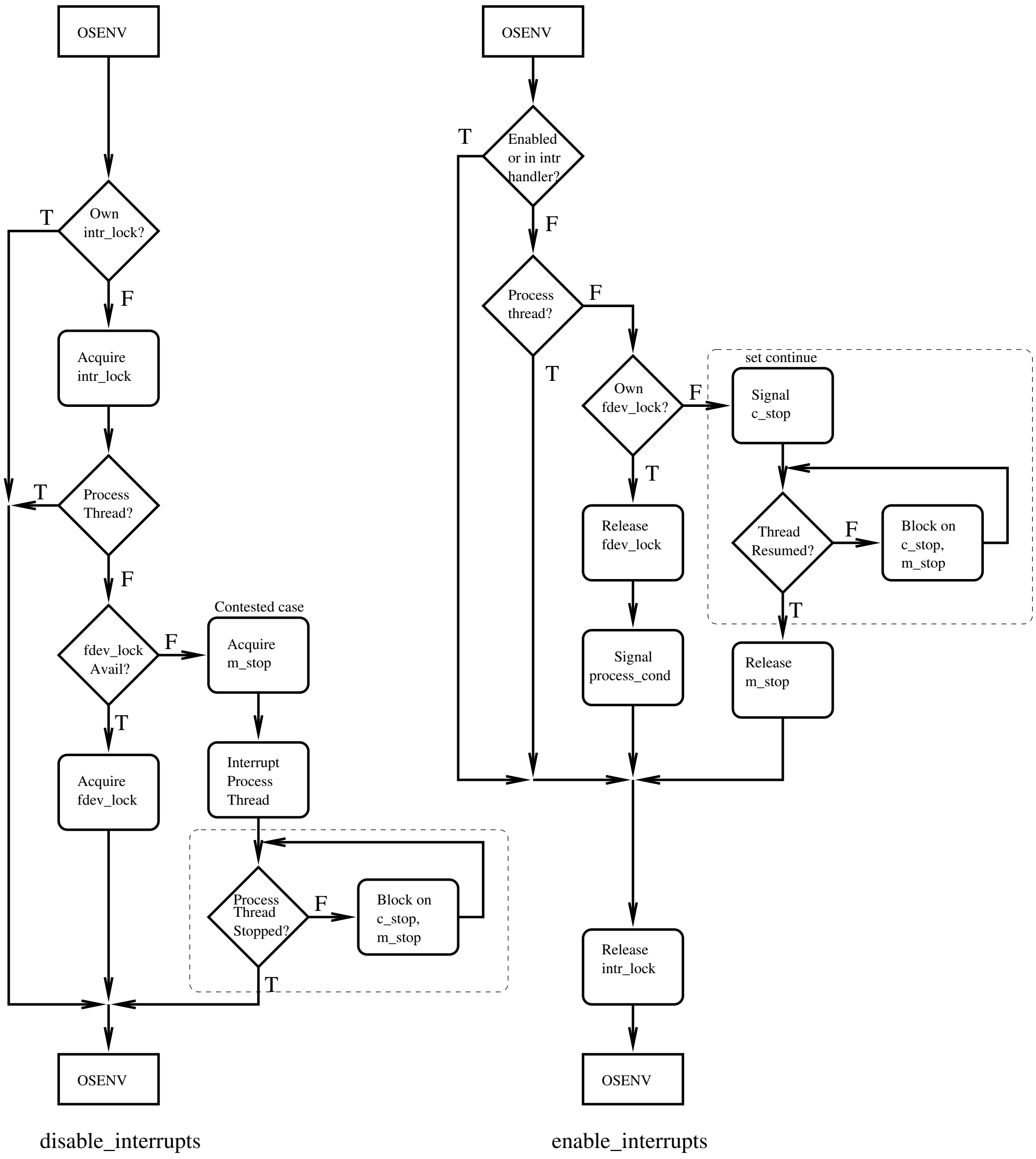

Figure 4.3. Simplified diagram showing `disable_interrupts` and `enable_interrupts`. Note that Interrupt Process Thread causes the current process thread to execute the *process interrupt handler* routine, shown in Figure 4.2.

Stopping the process-level thread while the interrupt handler is executing is done through a cooperative effort between the interrupt and process threads. The interrupt thread, after it has acquired the interrupt mutex, sends a `fluke_thread_-interrupt` to signal the process-level thread, then waits on a condition variable (`c_stop`, protected by the `m_stop` mutex), which the process thread signals when it has stopped. The process thread then blocks on the same condition variable, until the interrupt handler has finished and it signals the process thread. This process is also shown in Figures 4.2 and 4.3.

There are a couple things to note. The first is that if no process thread currently holds the process lock, the interrupt thread just acquires it itself and does not have to signal another thread. This is called the *uncontested case*. Second, it would not be necessary to wait for the process thread to stop (dashed box in Figure 4.3) if sending the signal synchronously stops a thread that has not disabled signals. It would be possible to optimize this case slightly by having only the process thread wait in its signal handler. This optimization was not done because interrupts are only rarely contested, and when this implementation was written, sending a signal was not guaranteed to synchronously stop the target thread. If the interrupt thread did not have to explicitly wait for the process thread to stop, it would eliminate a forced thread context switch to the process thread and back at the start of the interrupt handler. As a result of this work, the semantics for `fluke_thread_interrupt` have been modified so that the target thread will not execute any user-mode code between the time the thread interrupt system call returns and the target thread's signal handler is executed (provided that the target thread has thread interrupts enabled). This semantic change allows the code path inside the dashed boxes in Figures 4.2 and 4.3 to be eliminated.

It turned out that stopping a Fluke thread was a source of huge inefficiency. Fluke provides only `fluke_thread_get_state` to stop a thread; since capturing a thread's state requires a consistent view, the kernel stops the thread as a side effect. However, the kernel performed a large amount of work that was unnecessary for our purposes, with overhead that caused us to livelock under certain conditions while handling the interrupts from a 100Hz clock. When we changed the synchronization code to use the current `fluke_thread_interrupt` approach, we reduced overhead by one to two orders of magnitude. This example highlights some of the improvements possible with targeted kernel modifications.

4.4 Voluntary Blocking

Device drivers often give a command to a device and then wait for the request to complete. If the request takes a long time to process, it is advantageous to do other work during the delay. Generally, upon completion of the task, the device generates an interrupt to signal that the command has been processed. A good example of this is a disk read or write. It takes several milliseconds for the disk drive to move its head to the correct location and transfer the data. While this request is being processed, it is possible to do other useful work, including having the device driver process more requests.

The `osenv` API provides an `osenv_sleep`/`osenv_wakeup` interface which allows process level threads to sleep until awakened, usually by an interrupt thread. It is important to note that in `osenv` interrupt processing is not allowed to sleep or otherwise block.

When a process-level thread needs to wait on an event, it initializes a sleep record (`osenv_sleep_init`), and sleeps on it. An interrupt thread in the device driver is then responsible for doing a wakeup, which allows the thread to run again. By sleeping, the current thread allows a different process-level thread to enter the device driver and do work while the old thread is waiting for a wakeup.

Since a new process-level thread may enter the device driver when the currently executing thread sleeps, the implementation of sleep must be tied into the process-level synchronization code. They are connected by creating a fluke condition variable per sleep record, and using it in conjunction with `intr_lock` and `fdev_lock`. The sleep code ties these together by acquiring the interrupt lock and releasing the process lock, and then blocking on the sleep condition variable using the interrupt lock for synchronization. When the thread is awakened, it reacquires

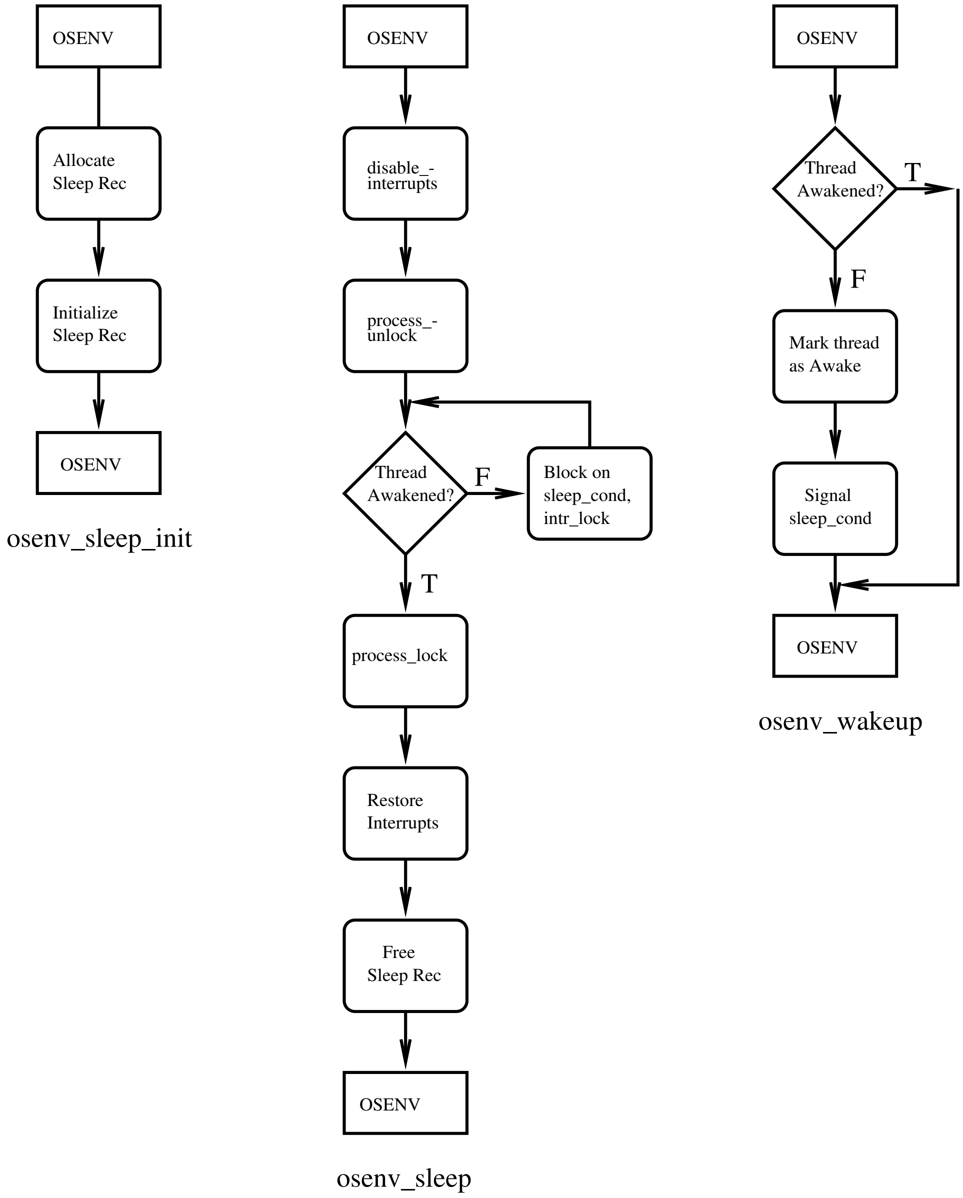

Figure 4.4. Simplified diagram showing the code involved in voluntary blocking: sleep and wakeup.

the process lock, and returns to where it was executing before it called sleep. The process involved in going to sleep and being awakened is shown in Figure 4.4.

As mentioned in Section 4.3, the interrupt lock must be held when trying to acquire or release the process lock. Instead of blindly acquiring the interrupt lock before acquiring the process-level lock, the code checks to see if it already has it. After the thread acquires the process lock, it only releases the interrupt lock if it did not have it before it tried to acquire the process lock. The same thing occurs when the process lock is released. This behavior allows the process lock to be acquired and released with "interrupts disabled" in the sleep code. The interrupt lock is released when the thread blocks on the sleep condition variable, and reacquired when awakened, which allows sleep and wakeup to operate properly with the process and interrupt locking.

As an optimization, since sleep is called quite frequently and the cost of creating a Fluke condition variable is reasonably expensive, the condition variables associated with sleep records are cached in a LIFO linked-list. Even though it may be desirable to limit the number of condition variables that are cached, the current implementation caches every condition variable it ever allocates. This means that the number of condition variables allocated is equal to the maximum number of simultaneously sleeping threads, which is no greater than the maximum number of concurrent requests in the device driver.[1]

4.5 Interrupt Delivery

It is necessary to deliver hardware interrupts to the device driver. Since the device drivers are executing in user mode, we accomplish this by using a dedicated thread that is created for each interrupt vector a device driver registers to receive. This thread does an IPC to the device server to register or unregister for the interrupt vector. After it has registered for the interrupt vector, it blocks processing interrupts until it is time to unregister. The device server, in turn, executes a special privileged system call to notify the kernel to deliver the specified hardware interrupts to the thread that registered with the device server. Using a device server allows the *policy* of determining which threads can receive hardware interrupts to be implemented outside of the microkernel, although the kernel provides the *mechanism* for delivering the hardware interrupt. Hardware interrupts are delivered

[1]We use a condition variable per thread because it is expensive to wake up all the threads when one needs to wake up. Using a condition variable per sleep event allows us to only wake up the correct thread. In turn, this fine-grained control allows us to avoid the "thundering herd" phenomenon where all the threads are awakened but all but one must block again.

to the user-mode thread by the kernel's interrupt handler sending a `fluke_thread_interrupt` to the interrupt handler thread. This process is analogous to a Unix kernel sending a `SIGINT` to a user process.

There are a few motivating reasons for using a dedicated thread per interrupt vector. The mechanism used to deliver interrupts to user mode, via `fluke_thread_interrupt`, simply provides an indication that someone sent the thread an interrupt, but with no associated data. A mechanism for indicating the interrupt vector could have been added, but the current approach is simpler. The current approach would also allow us to perform some optimizations that would eliminate the IPC to the device server when disabling and enabling specific interrupt vectors.

Currently, attempts to disable or enable individual interrupt vectors are handled by performing an IPC to the device server. Since each interrupt vector is sent to a different interrupt thread, it is possible to add an additional mutex for each interrupt vector, and have the interrupt handler acquire that mutex in addition to the existing interrupt mutex. With some additional complexity, this would then allow individual interrupts to be disabled on a per-driver basis without performing an IPC. However, the most common reason for disabling a single interrupt is to allow different interrupts to occur while the interrupt handler is running. Since the process of delivering an interrupt to the interrupt handler disables the interrupt, and we do not allow interrupts to be enabled inside the interrupt handler, there is usually no need to manipulate individual interrupt enables except during interrupt probing during driver initialization.

4.5.1 Concurrent Interrupts

To remain responsive to higher-priority interrupts, most operating systems allow higher-priority interrupts to occur during long-running interrupt handlers. We do not currently support this in our Fluke user-mode device drivers. It is not currently possible to reenable other interrupts inside the interrupt handler, as that would create the situation in which one interrupt thread needed to stop another interrupt thread. Instead, we can run different device drivers in separate address spaces. The kernel itself only keeps interrupts disabled for very brief periods.

While a thread is processing a thread interrupt, further thread interrupts are disabled for that thread. It would be possible to handle other interrupts by saving the interrupt state and explicitly reenabling interrupts in the thread interrupt handler, to handle other interrupts. Since the kernel does not reenable that hardware interrupt until all the handlers have finished execution, there was no reason to add the complexity of recursive interrupt locks.

Given our use of multiple interrupts threads and a single interrupt lock per address space, concurrent interrupts are not feasible, which prevents a high-priority interrupt from preempting a lower-priority interrupt in the same driver process. If the interrupt thread released the interrupt lock to reenable interrupts, a process-level thread could acquire the interrupt lock and execute in parallel with it, which would produce incorrect results, as discussed in Section 3.2. To avoid these problems, any attempts to reenable interrupts from inside an interrupt handler are ignored.

In practice, the main reason for enabling interrupts in an interrupt handler is to avoid losing clock interrupts while processing long interrupt handlers, or to allow higher-priority interrupts to be processed quickly. It usually does not matter if a device driver occasionally loses a clock interrupt, and the Fluke kernel itself will not drop clock interrupts. Also, device drivers may be run in separate address spaces and their interrupt handlers can be run at a higher priority to reduce the impact of using only a single interrupt lock. Additional steps can be taken to ensure that the interrupt handlers do very little by deferring as much processing as possible until later in another thread, which also reduces the amount of time the interrupt lock is held. This strategy is currently used by the network drivers, where the interrupt handler queues up the packets at interrupt time without doing any processing of the data.

If we used a single interrupt thread per driver process, it would be possible to block `fluke_thread_interrupt`s to disable hardware interrupt processing, instead of using the interrupt mutex. Using a variant of the current process-level synchronization code, where the interrupt thread waits until the process-level thread has stopped, process-level threads could disable thread interrupts to prevent the interrupt thread from running. Reenabling thread interrupts inside the interrupt thread would allow for reentrant interrupt handlers. However, given the cost of saving the interrupt state, it would only make sense to reenable interrupts if interrupt handlers execute for a long time and need to be preempted by another interrupt thread for that device driver.

With the dedicated interrupt threads we use, it is also possible to to use thread priorities to reflect device priorities. However, since the interrupt thread's execution cannot be preempted by another interrupt thread in the same driver, priority inversion could still occur. It is interesting that the approach that uses a single interrupt thread allows "higher priority" interrupts to preempt the executing interrupt handler, but it does not allow assigning different priorities to the different interrupts.

4.5.2 Autoprobing Lessons

Many Linux device drivers auto-probe to determine which interrupt vector a device is using. They do this by registering for all of the interrupt vectors of interest, causing the device to generate an interrupt, and unregistering for the interrupts. Then, by looking at what interrupts occurred, they can normally determine which interrupt vector the device uses.

The initial Fluke implementation of the interrupt support code did not function properly in this environment. When registering for an interrupt, the process-level thread simply created a new thread (whose job it was to register for the interrupt) and returned. When it was time to unregister, it signaled the interrupt thread to unregister, and returned immediately. The newly created thread was not given the opportunity to register for the interrupt before the interrupt was generated, and the interrupt handler was not given a chance to run before the driver checked to see if an interrupt had occurred. This behavior resulted in auto-probing the interrupt vector to fail, as "no interrupts" had occurred.

To provide the synchronous registration and unregistration required by the device drivers, we use a mutex and a condition variable per interrupt handler. This allows the process thread to wait until the interrupt thread has accomplished the registration or unregistration activity. This design guarantees that the interrupt thread will be registered before the driver generates the interrupt, and that the thread will be given the opportunity to process a pending interrupt before the call to unregister returns.

4.5.3 Shared Interrupts

Some hardware allows multiple devices to use the same interrupt vector. This is normally accomplished through the use of level-triggered interrupts, where a device asserts an interrupt until the event that caused the interrupt has been dealt with. This behavior contrasts with the use of edge-triggered interrupts, where the interrupt signal is asserted only briefly before being unasserted automatically.

It is important to note that shared interrupts, like any other shared resource, can cause problems when sharing between buggy or malicious device drivers. If a device driver is sharing an interrupt line with another device driver, it can refuse to acknowledge the interrupt, which prevents the other device drivers sharing that interrupt vector from receiving further interrupts.

Because level-triggered interrupts cannot be reenabled until the cause of the interrupt has been dealt with, software cannot force the safe sharing of these interrupts. Edge-triggered interrupts could be reenabled immediately in the kernel, but the only solution when dealing with level-triggered interrupts would be for the administrator to ensure that device drivers that do not have the necessary mutual trust use different interrupt vectors.

4.5.4 Software Interrupts

Linux and other Unix operating systems use a notion of software interrupts. Software interrupts are used to move processing outside the hardware interrupt handler so that the processing may be done at a lower priority. Software interrupts are still processed at a higher priority than the process-level activity, although hardware interrupts may interrupt software interrupt processing. The OSKit does not export a notion of software interrupts, which are simply processed in the hardware interrupt handler after reenabling interrupts. Since Fluke does not actually reenable interrupts, software interrupts are processed as part of the hardware interrupt handler without making a distinction.

To minimize the impact of running software interrupts with hardware interrupts disabled in the driver, most of the processing normally done by software interrupts, such as TCP/IP protocol processing, is done in the process-level threads. With networking, we defer the processing by having the interrupt thread place the packet on a queue, which is emptied by the process-level threads. The process-level threads may either process the data, or perform an IPC to send the packet to the waiting application. We were able to defer processing without driver modifications by writing a Fluke-specific callback function which we passed to the device driver.

4.6 Memory Allocation

Memory allocation in the driver framework is different from other Fluke applications in two ways. First, the device driver framework tracks two independent pools of memory: a pool of nonwired virtual memory and a pool of wired physical memory. The memory type must be explicitly requested. Second, nonwired memory uses the same allocator as normal Fluke applications, except that we use the interrupt lock instead of using a separate mutex to protect the memory pool's data structures.

Memory allocations must use the interrupt lock to avoid deadlock, which otherwise could be caused if an interrupt handler attempts to acquire the memory lock after it has stopped a process level thread that holds the memory lock. This locking results in memory allocation being done with "interrupts disabled," which provides the proper synchronization for the `osenv` environment. Memory allocations are protected in the BSD kernel in a similar manner, where `malloc` executes at a very high priority to protect it against recursive calls from a network driver's interrupt handler [42].

A second memory pool, unique to the Fluke device drivers, manages physical memory. Access to this pool is synchronized the same as for the first pool. Physical memory regions are allocated through an IPC to the device server (Section 4.2), which returns a memory object that can be mapped into the driver's address space.[2] Instead of being mapping physical memory arbitrarily, the driver framework takes advantage of the fact that only a few pages used for the kernel's entry points are normally mapped below 128MB, which is where the program text is loaded in the address space. The physical memory pool allocator maps memory at the virtual address corresponding to the physical location. Thus the driver framework only allocates physical pages below 128MB.

Even though it is not necessary to map memory with identical virtual and physical addresses, it does have advantages. Many of the Linux device drivers in the OSKit assume that a virtual address corresponds to the same physical address. By guaranteeing that assumption, it is possible to run more unmodified device drivers in Fluke. This convention also assures that virtually contiguous pages are also physically contiguous, which simplifies tracking of the memory. Also, given the virtual address, it is trivial to determine if a page is wired and if so, its physical address.

There are some limitations of this approach. First, it limits the device drivers to 128MB of physical memory. Also, there is no provision in Fluke to request that an existing virtual page be wired. This limitation is not usually a problem, as wired memory is allocated when it is not clear what type to allocate, it does prevent one class of drivers from working without modifications: SCSI device drivers that assume the SCSI scripts loaded with their data segment are in wired memory. Although some of them try to convert the virtual address to a physical address, there is no underlying wired physical memory as the initial `text`, `data`, and `bss` segments are all allocated out of virtual memory when the parent created the process. To get around this problem, those problem device drivers are run with all their code and data wired, which is done by running them in the root server, or *kserver*.

[2]Information on Fluke regions and mappings may be found in the Fluke API reference [19].

4.6.1 Blocking and Interrupt Time Behavior

Memory allocation is one of the actions during which Unix-like operating systems frequently block. Since processing cannot block during interrupt handling, interrupt handlers must deal with the possibility that memory is unavailable.

One of the reasons that running Unix device drivers on Fluke is difficult is that Fluke threads block and are preempted at different times than the Unix code expects. Even though Fluke threads may "block" at inopportune times for the device driver, we ensure that other requests in the same driver address space do not *see* them block. The synchronization provided by the mutex variables prevents other driver threads from running unless the device driver blocks explicitly. If the underlying Fluke thread blocks or is preempted, the driver behaves as though it is simply a slow operation, not a blocking one, even if other threads (outside of that `osenv` environment) execute.

Memory allocations are defined by `osenv` to be blocking, unless the "do-not-block" flag is explicitly passed as a parameter. Our implementation for Fluke does not take advantage of the ability to block because not all of the `osenv` glue code in the OSKit has been written to properly deal with memory allocations blocking, even though the underlying driver code can deal with it. Only the "sleep" call has been properly implemented with respect to blocking, since many of the other interfaces "allowed" to block have problems in the current implementation of the OSKit wrappers. In the event that we run out of memory, we return NULL, instead of blocking until memory becomes available. Since virtual memory is assumed to be plentiful, that should never occur—pageable memory will get paged as necessary. The problem is that much of the code is not prepared to receive NULL unless the do-not-block flag is specified, which will cause problems if we do run out of memory.

In the current Fluke implementation, nonblocking memory allocation is handled in the same manner as normal blocking allocation: if there is no memory immediately available, either more wired memory is requested via an IPC to the device server or more virtual memory is allocated, depending on the type of memory requested.

Since IPCs are somewhat expensive, multiple pages are allocated at a time, even if only a few bytes are required to satisfy the request. An IPC is safe to perform at interrupt time because it returns from the server without being arbitrarily blocked, and it does not cause any locks to be released. Once physical memory is allocated, it is never released until the device driver exits. Virtual memory is also not released, but may be paged out by the system as necessary.

4.6.2 Future Work

If we implemented blocking memory allocation, it would still be desirable to have the interrupt handler do an IPC if necessary to allocate memory. In fact, the do-not-block behavior would be the same: return NULL if the IPC failed to return more memory. The change would be for requests that could block: they would be delayed arbitrarily until more memory was available. This change would allow the driver framework to operate more gracefully under situations where the underlying operating system ran out of available memory.

Since the drivers execute in a pageable user mode environment, memory allocations are unlikely to fail, so the advantage of supporting blocking is rather small. The biggest constraint would be if all 128MB of physically mappable memory was allocated. However, since only a fraction of memory allocations require physical memory, that is not likely to occur in the immediate future. If limiting the amount of mappable physical memory to 128MB becomes a problem, it would be necessary to increase the limit or to implement blocking memory allocations. One disadvantage of the current framework is that physical memory not currently being used by one device driver process cannot be used by another device driver process. However less memory is being wired by the drivers than would be wired in a comparable monolithic kernel, so this would imply that the drivers in a monolithic kernel would also require more than 128MB of memory to execute.

4.6.3 Lessons

In our early experimentation, we did not do an IPC at interrupt time, instead expecting that allocations done at the process level would make memory available.

It was thought that an IPC at interrupt time would be slow and could delay processing. However, when receiving network packets, all incoming packets were getting dropped. This behavior was due to an interesting interaction where the interrupt code was never able to allocate memory because no process-level thread was ever doing an IPC to request memory. This phenomenon was caused by the interrupt-driven nature of the device driver. Since all incoming packets were dropped, there was no need for the application to send a reply at process level, so no memory was ever being allocated.

4.7 Controlling Access to I/O Devices

4.7.1 Restricting Access to I/O Space

Although some devices support memory-mapped I/O, hardware device registers are normally mapped into I/O space on the 80x86. I/O space is different from memory mapped I/O in that it requires special instructions to access it, and I/O addresses are not translated by the MMU.

On the 80x86, it is possible to grant access to I/O space in one of two ways: either allow a process access to all of I/O space, or selectively grant access to only certain addresses in I/O space through the use of a per-process I/O bitmap. The kernel can also virtualize accesses to I/O space by trapping accesses to I/O space. Although trapping access does not grant access to I/O space, the kernel can then determine whether to allow the access to proceed, essentially "virtualizing" the registers. Trapping accesses is not normally done due to the high cost of taking the trap, and the necessity for the kernel to know as much about the device as the application or device driver. However, OS/2 virtualized some device registers for compatibility with DOS applications. Since many DOS applications accessed the DMA registers directly, virtualizing I/O space access allowed OS/2 to run them in pageable memory, and still allowed DOS applications to perform ISA DMA. Virtualizing the registers also allowed multiple DOS applications running under OS/2 to access the real-time clock concurrently, as well as enabling OS/2 to support most DOS-based games, which programmed the ISA DMA controller directly to produce sound. Since the cost of taking the trap is rather high, and one motivation for moving the device drivers out of the kernel is so that the kernel does not need to know about the hardware, Fluke does not currently attempt to virtualize any hardware.

Unfortunately, the x86 processor does not make a distinction between read and write access to I/O space. Although the granularity of memory is a page (4KB), the granularity of I/O space on the x86 is a byte. A byte is not sufficient granularity for all resources, as several devices share I/O space at a finer granularity. Examples of this are the PS/2 keyboard and mouse, which are controlled by the same registers, and the IDE and floppy controllers, which use different bits in the same status register.

On Intel's Pentium Processor, an unrestricted I/O space access takes four cycles, while restricting the access through the use of a bitmap requires 21 cycles, plus any cache misses on the bitmap. The extra cost involved in updating the bitmap on a context switch may be considerable, depending on the strategy used. As for the space, with a 16-bit I/O space and one bit per byte of address, the per-process bitmap is 8KB.

Fluke provides the ability to use both the direct-access and the bitmap-controlled I/O space access mechanisms. By default, device driver processes are given unrestricted access to I/O space. However, it is possible to configure the kernel and the device server to use the bitmap to prevent unauthorized access. The ability to enforce access controls proved to be invaluable when debugging the device driver framework code, as well as in exposing many bugs in the original Linux device drivers. By far the most common problem encountered was simply the absence of a call to `check_region` in the original Linux driver to see if the region was already used before accessing it. However, in several cases the drivers requested the wrong amount of I/O space or accessed more than they requested.

The bitmap mechanism was implemented to aid in debugging and to show that access controls could be easily implemented. Since the primary goal was to make the bitmap mechanism functional, little effort was made to optimize that code path.

When performing a context switch, the bitmap is updated to contain the values for the new process. The use of a bitmap adds 8KB of additional state that needs to be copied on a context switch, although it is possible to compress the state. Since most drivers only allocate a few small densely-packed regions, updating just those regions would be significantly more efficient. It would also be possible to store multiple bitmaps in the TSS (the data structure that contains the bitmap) and just update the bitmap pointer, or to use a different TSS for each process. Both of these optimizations would eliminate the copies.

All of the code outside the kernel runs under the assumption that the kernel disallows access to I/O space unless it has been granted by the device server, even though it may not be enforced. When a device driver asks if a region is free, the driver framework attempts to allocate the region. If the device server denies the request, then the driver is told that the region is in use, and it cannot probe or access that region. Otherwise, the driver is allowed to access that region.

Since it is likely that a device driver will attempt to probe several regions that belong to a device driver that has not probed yet, at the end of the driver's initialization phase, any regions that were not subsequently allocated by the device driver are returned to the device manager. Note that freeing I/O space is not necessary for PCI device drivers, as the configuration space can be inspected to determine exactly where the devices of interest are without probing random locations.

4.7.2 Memory-Mapped I/O

Even though most PC devices map their registers into I/O space, some devices map their registers into memory addresses instead. Some device drivers also access configuration information stored in the BIOS. Gaining access to specific physical addresses is accomplished by calling `mmap` on `/dev/kmem` (a file exported by the kserver that extends over the entire kernel's virtual address space). Physical addresses below 128MB are mapped at a virtual address equal to their physical address, just as with wired memory allocations.

This approach works well for mapping the BIOS and locations below 128MB, but it does not work well for PCI devices utilizing memory-mapped I/O, because they are mapped into high physical addresses that are beyond the end of physical memory. Fortunately, the encapsulated device drivers are already able to deal with these regions being mapped at a different virtual address, so it is possible to map those buffers into arbitrary virtual address the same as other `mmap`ed buffers.

The ability to map high addresses has not been implemented and is left as future work. Although there are no significant technical reasons it could not be done, the extra complexity was not justified given that none of the device drivers we have tried to use depend on it because most PCI devices that provide a memory-mapped interface also provide the same registers in I/O space. Some drivers (such as the NCR/Symbios Logic SCSI driver) attempt to use memory-mapped I/O and display a warning message if mapping the device memory fails, but fall back to using the I/O-space registers.

Other reasons for not implementing memory-mapped I/O support are due to the fact that currently the kernel's virtual address space only maps physical memory. It would have been necessary to add code to the device server to map physical address regions that the kernel did not map. Due to some peculiarities with the Fluke and *kserver* memory management schemes, memory-mapped I/O support would have taken several days to implement. Because of the current scheme of mapping physical memory at the equivalent virtual address, virtual-to-physical address translations are not tracked by the driver framework. It will be necessary to track these mappings for device registers above 128MB.

4.8 Shared Resources

4.8.1 ISA DMA

ISA devices that use DMA generally do so through the use of a DMA controller on the motherboard. The ISA DMA controller can be programmed to transfer memory in the lower 16MB of physical memory, and is extensively used by many "high-performance" ISA devices, such as SCSI controllers, sound cards, and some network cards. As the ISA bus offers a relatively low transfer rate, few new devices, other than some sound cards, now use ISA DMA. In fact, most devices have now migrated to the PCI bus or other interfaces such as USB.

We had originally intended to wrap access to the DMA controller much like accesses to other resources, such as the interrupt controller. However, even though the Linux device drivers consistently used kernel routines to allocate a DMA channel, programming the controller is usually done directly by the device drivers, without even the use of standardized macros. Rather than modify the device drivers to do the "right" thing, we decided to leave them alone, in the hope that future versions of the device drivers were more well-behaved than the current drivers. Supporting ISA DMA is even harder than supporting other shared resources because the Linux device drivers often access the DMA control registers *before* they allocated the DMA channel. This practice was too pervasive to make fixing the device drivers attractive. Despite these many problems posed by ISA bus devices, we still managed to support them.

To support ISA DMA, we took several steps. First, we restricted device drivers that wish to use ISA DMA by requiring them to all execute in a single address space. This restriction allows accesses to the DMA controller to be serialized properly by the existing `osenv` serialization properties. Second, as part of initializing a device driver, we attempt to allocate the I/O space corresponding to the ISA DMA controller before we run the driver's initialization code. If the device driver does not ultimately request a DMA channel, the DMA controller's I/O space is released when the other unused I/O space is released, and may be allocated by the next device driver.

If I/O space accesses are enforced, and the DMA registers cannot be allocated, then accesses to the DMA controller will cause the device driver to take an exception and probably die as a result of the signal. This exception could occur if the device server decided not to grant access to the DMA controller, e.g., another another driver had already allocated it. This behavior is exactly the same behavior that a device driver will experience when accessing any unallocated I/O space address or an unmapped memory location.

Although it is possible to modify the device drivers to use a standard interface to access the DMA controller, the cost of doing an IPC every time it is necessary to access the DMA controller would be quite high. This expense means that we would be likely to retain the restriction on placing all the ISA DMA device drivers in a single address space for performance reasons, even though we could check the values used.

4.8.2 PCI Configuration Space

The PCI bus [50] has the traditional notions of memory and I/O space, but it also has a third “address space,” called PCI configuration space. Each device has its own unique set of configuration space registers, which are used for configuration and initialization of the PCI bus devices.

Accesses to PCI configuration space are mediated by the device server. Fortunately, all the encapsulated PCI device drivers use standard routines to access PCI configuration space. It was fairly easy to add PCI configuration space access to the `osenv` API, and replace the original routines in Linux with ones that used the `osenv` versions.

Even though the current device server does not enforce any access controls, it would be simple to regulate accesses to configuration space, as the device server sees all the requests. The device server can easily deny access to a device by returning “-1” for all the register contents. This value tells the device driver that no device is present at that location, with no harmful effects. Access controls are left as future work, although the basic hooks are fully implemented.

Additional future work could be done to use the information in PCI configuration space to further limit the activities of device drivers. If it is known that a device driver is a networking device driver for a certain vendor’s card, it is trivial to allow access only to the configuration space for those devices. It is also easily possible to restrict memory and I/O space accesses to the driver handling the corresponding device.

Unlike accesses to the ISA DMA controller, PCI configuration accesses are not performance-critical because device drivers generally only need to access configuration space during initialization. Because configuration space cannot contain registers that are used during normal operation of the device, and device drivers cache any values they need from the configuration space, there are no IPCs to read configuration space during normal execution.

4.8.3 Timers and Clocks

Many of the device drivers need a notion of time. The `osenv` interface to obtain the current time simply calls a standard library routine (`gettimeofday`), which does an IPC to its time server (normally part of the kserver) to find the time current time.

The drivers may also register a periodic handler, or a one-shot handler to be executed some number of clock ticks in the future. Timers are all implemented on top of a simple periodic timer, driven by hardware clock interrupts, using the default implementation of the timer support code in the OSKit. In the Linux drivers, a 100Hz clock is used to increment the 'jiffies' variable, which is used by device drivers to explicitly busy-wait and as an indication of the current time.

Clock interrupts are handled differently by the Fluke kernel than other device interrupts. Timer interrupts need to be handled by the kernel, for scheduling and other internal tasks. Additionally, since the timer is an edge-triggered interrupt shared by all the device drivers in the system, the interrupt is reenabled immediately by the kernel, to avoid a buggy or malicious driver from "stopping time" for the entire system. Thus, although a device driver might occasionally miss a clock interrupt under heavy system load, the kernel will not miss any clock interrupts.

Clock ticks are received by registering for the clock interrupt. If the kernel went to a 1kHz timer, a few lines of code in the kernel would allow a 100Hz timer by "dropping" extra interrupts before delivering them to the device driver.

4.9 Ethernet Support

Networking support is a little unusual in that work may be initiated by a remote node. Because of this, it is possible for another node to cause a computer to perform excess work. Under severe cases, unsolicited network communication can cause livelock [45], where the computer spends all its time starting work that it throws away before completing the overall task. To handle this problem, we wrote the network support code to minimize the amount of work that is done at interrupt level.

In Fluke, an application that wishes to receive packets does a blocking read on the network file interface. At interrupt time, incoming packets are placed on a FIFO queue. If the queue is empty with a waiting reader, the reader is awakened, which then processes the packet at process level. If there is no thread waiting to read the packet, it remains on the queue. When the queue becomes full, further packets are simply dropped in the interrupt handler. Not performing the IPC reply at interrupt time defers all significant processing until later. This design allows us to keep the interrupt handing time as short as possible, and is analogous to scheduling software interrupts in Unix.

On transmit, packets are sent to the device driver to be processed. The desired behavior when the device driver's transmit queue is full can be selected at open time. The device driver can either block the request until it is able to add it to the queue, or return an error to the application and have it retry at a later point.

4.10 Logging

Device drivers generate informative messages that are generally displayed on the system console. We provide a crude logging interface by doing a `write` on `stdout` for driver messages. The current console interface is blocking, with the kernel using a serial console for I/O. Although it would be possible to have a separate thread handle output asynchronously, the low-level kernel output routine would still busy-wait on the serial port, outputting one character at a time. A desired kernel enhancement would be to have interrupt-driven console I/O, which would allow output to be done asynchronously, and dramatically reduce the processor overhead required to generate output. An improved console driver would benefit all applications doing console I/O. Fortunately, doing a `printf` from a device driver after initialization is relatively rare, which is why many Unix kernels also implement kernel `printf`s using polled I/O.

CHAPTER 5

PERFORMANCE ANALYSIS

This thesis involved engineering a device driver framework to support the use of legacy device drivers from a traditional monolithic kernel in the context of our research microkernel operating system, Fluke. Although functionality was the primary concern, we also wanted to quantify the device driver performance, which we did by measuring latency, bandwidth, and processor utilization, and comparing our measurements with those from other systems as appropriate.

5.1 Methodology

We ran several tests to measure different aspects of device driver performance in Fluke. Each test was run 35 times consecutively, with the time recorded after each run. The first run, used to warm the caches, was thrown out and the remaining 34 times were averaged. In all cases, the standard deviation was less than one percent.

Each test program was run under Linux 2.0.29, FreeBSD 2.2.6 (with the 980513-stable CAM patches), the OSKit, and Fluke. Additionally, Fluke tests were run both with the application in the same address space as the device driver, and with the device driver and the application in separate address spaces. For the OSKit tests, the application was linked with the device driver and the low-level initialization code, so the test program *was* the operating system. All of the operating systems except the OSKit ran the tests in user mode. Linux is included because it uses the same device drivers as Fluke, and FreeBSD was included to provide a broader basis for comparison.

All tests were run on a pair of machines with Intel DK440LX motherboards, Pentium II/300 processors, 128MB ECC SDRAM, and Intel EtherExpress Pro/100+ cards, connected via a crossover Ethernet cable. The disk tests were done using an Adaptec 2940UW SCSI controller and a Seagate ST34501W hard drive. The integrated Adaptec 7895 SCSI controller was used for a 1GB Jaz disk containing FreeBSD 2.2.6, while Linux 2.0.29 was installed on a 1GB Quantum Fireball IDE hard drive. Neither the Jaz drive nor the IDE hard drive were accessed during these tests. The integrated i82557 Ethernet controller was used for normal network connectivity under FreeBSD and Linux. Since Linux 2.0.29 did not contain a driver for the EtherExpress Pro/100+ network card, we used the same Linux driver that we added to the OSKit (Donald Becker's eepro100.c: v0.38 2/17/98).

Accurate results were obtained through the use of the performance monitoring counters found on Intel's Pentium II processors. Two counters were used: the Time-Stamp Counter (TSC), which the CPU increments once every processor cycle, and the CPU_CLK_UNHALTED counter, which the CPU increments once for every cycle the processor is not in the HALT state. Of the operating systems used for these tests, Fluke, FreeBSD, and Linux all halt the processor when there is nothing to do, and they all provide the ability to read these counters. The OSKit, by contrast, never halts the processor, as it busy-waits when sleeping, which is why we report zero idle time for the OSKit tests. The use of the unhalted counter, along with the cycle counter, enabled a very precise and accurate measurement of elapsed time and CPU utilization.

5.2 Tests

We ran multiple tests to measure performance across a wide range of uses and to measure specific features of the device driver performance. We performed disk reads and writes, sent and received network packets, used the disk and the network simultaneously, and measured the interrupt processing overhead.

5.2.1 Disk Read Test

The goal of the disk read test was to determine how much of the disk bandwidth could be utilized by the device drivers, and how much computation was required to utilize that bandwidth. This test involved reading of various-sized sequential blocks from the beginning of the hard drive, for a total of 8MB per run. Each run consisted of a seek to sector zero, followed by sequential reads. We used the raw disk device (`/dev/rsd1`) under FreeBSD, `/dev/sda` under Linux, and `/dev/disk0` for the "separate" Fluke tests. For the OSKit and the "colocated" Fluke tests we used the OSKit's block device interface directly. We also read from a "null" device (`/dev/zero`) in Fluke, to measure the communication overhead. This Fluke test is labeled "IPC-only," and reflects the raw bandwidth achievable through IPC using the MOM runtime with Flick-generated IPC stubs.

Since Linux does not have a raw block interface and it performs read-ahead buffering, `hdparm`[1] was used to disable the read-ahead buffering on the hard drive before running the tests. There is no way to bypass the buffer cache in Linux. To work around this problem with Linux, we had to remove the seek to the beginning of the disk at the start of each run to avoid re-reading data from the buffer cache. Although the tests stayed well within the first zone on the drive, the Linux results do not include the seek to sector zero, which artificially reduces the time per run by a few milliseconds, or less than 1% of the test execution time.

5.2.1.1 Results

Read bandwidth is shown in Figure 5.1, where larger values indicate better disk bandwidth. An item to notice is the Linux performance at 512 bytes. Even with disk read-ahead disabled, Linux still reads data 1KB at a time, which enabled it to do relatively well on the 512-byte reads. Fluke, even in the colocated case, does worse than the monolithic systems below 8KB transfer sizes, since it is more expensive in Fluke to block and wake up threads than in the other operating systems. However, even the separate Fluke test is able to achieve the maximum disk bandwidth with 16KB transfers. The "IPC-only" transfers (upper-left corner) have been cropped to show more detail for the other cases. The IPC-only case peaks at 45MB/s for 4KB transfers, where it remains for the larger transfer sizes. Even though IPC offers enough bandwidth for disk reads, the additional overhead required to process IPC

[1] `hdparm` is a Linux utility that allows the user to get and set various hard disk parameters, including the read-ahead buffer size.

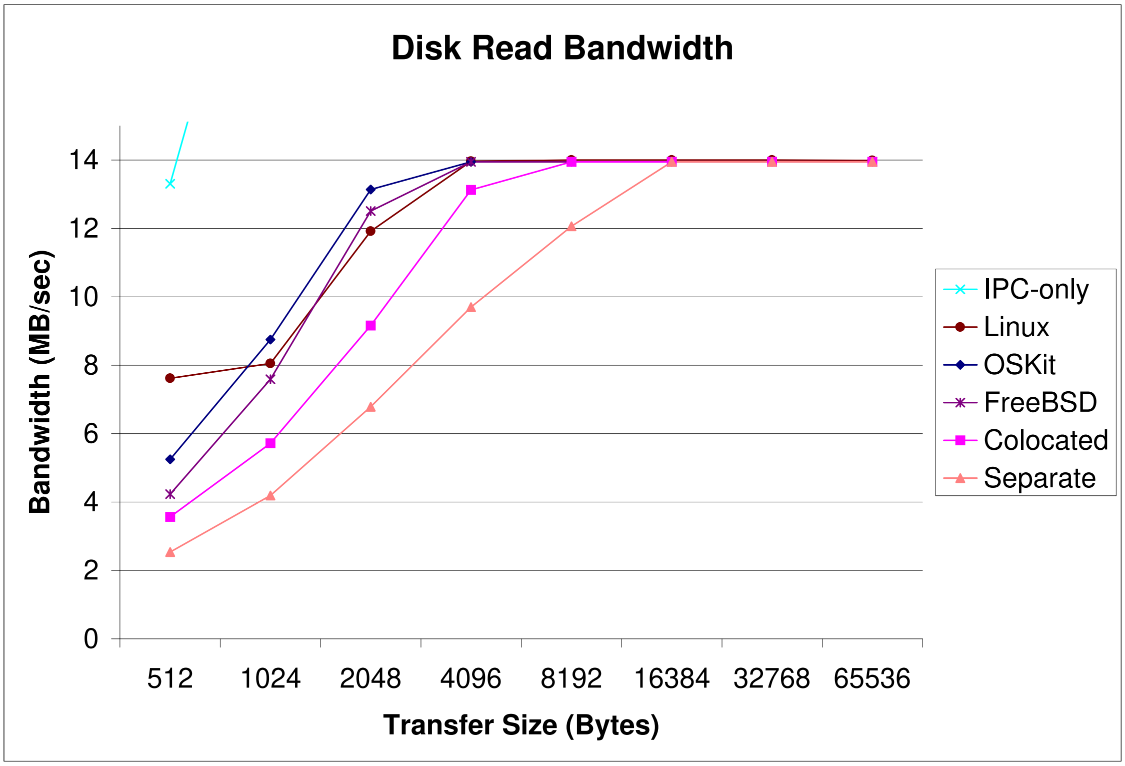

Figure 5.1. Read bandwidth of the hard drive. Higher is better. The Fluke configurations achieve the available disk bandwidth at larger transfer sizes than the Unix systems.

severely impacts the achievable transfer rate when communicating with the driver through IPC.

With a 4KB transfer size, a common filesystem block size, the colocated Fluke driver comes within 94% of the disk bandwidth, while the separate driver achieves 70%. At 8KB transfer sizes, the default filesystem block size under FreeBSD, the colocated driver has maxed out the drive, while the separate driver is achieving over 86% of the maximum transfer rate.

The bandwidth results for larger block sizes are encouraging, but they do not tell the whole story, as Fluke requires extra computation. Figure 5.2 is a plot of the processor cycles spent busy during the read tests. Larger values indicate that less processor time is left over for use by other applications. On this test, the traditional Unix systems do the best, followed by Fluke device drivers colocated with the test application, followed by Fluke with everything separate. At small transfer sizes,

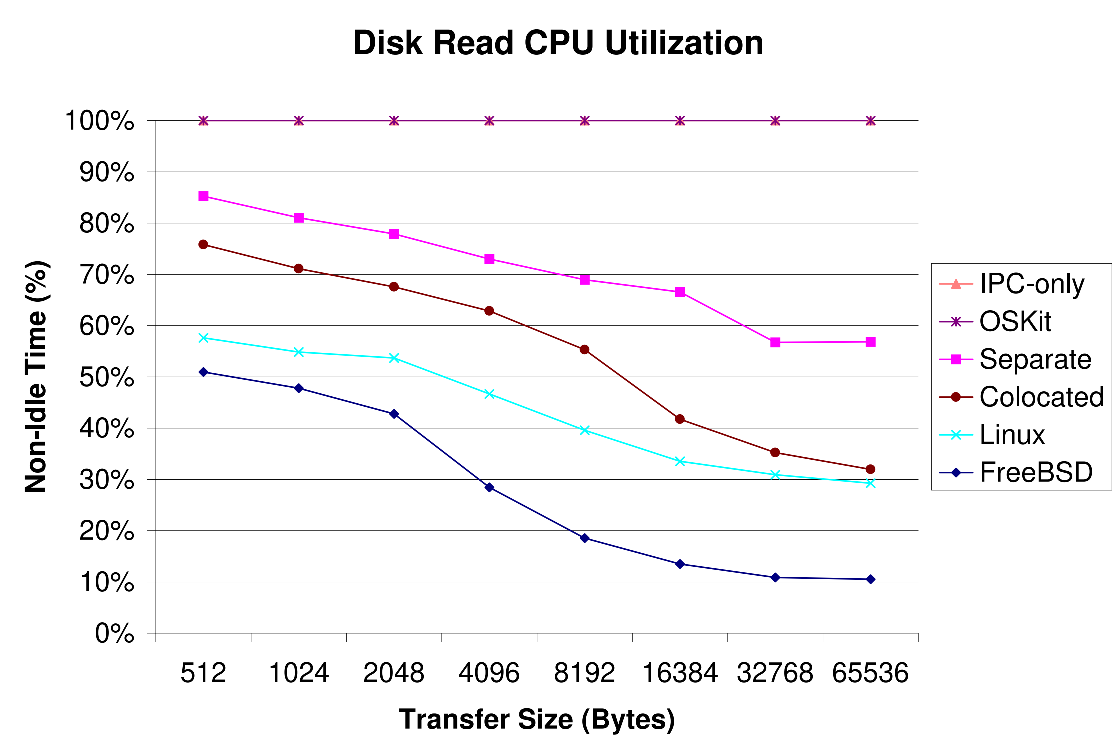

Figure 5.2. Busy time while reading the hard drive. Lower is better. Note that both the OSKit and the IPC-only tests have no idle time, so their lines are at the top of the graph. The colocated Fluke configuration gets closer to the Linux CPU utilization at larger transfer rates, while the separate Fluke configuration remains much higher.

colocated Fluke uses roughly 30% more CPU cycles than the Linux systems, which decreases to under 10% more CPU cycles for large transfer sizes. In the separate driver case, Fluke goes from under 50% to nearly 95% higher processor utilization due to the IPC costs.

The curve of the lines corresponds to the measured disk bandwidth. When the disk transfer rate becomes the bottleneck, the busy time drops significantly. Knees in the curves can be seen in all the tests, in particular at 2KB for FreeBSD and Linux, 8KB for the colocated Fluke driver case, and 16KB for the separate driver case. The colocated case performance gets very close to Linux at large transfer sizes, where the Fluke overhead is a smaller part of the execution time. Although the colocated case does better for larger transfer sizes, the copies along the IPC path make a bigger difference, as can be seen in Figure 5.2.

Note that the utilization for both the OSKit and the IPC-only test are at 100%. The IPC-only test never waits for hardware, and hence is never idle. The OSKit does not have any idle time because the read requests busy-wait for the request to complete. At first it may appear unfair to characterize the OSKit as having zero idle time, but with blocking requests and only a single thread, the time spent spinning cannot be used anyway, so it does not make sense to measure the spin time.

5.2.2 Echo Test

Since latency is often a critical component of networking performance, we measured the round-trip latency of network communications over a 100Mb Ethernet network. We measured the latency by determining the time it takes to send a UDP datagram back and forth across the network: Machine A sends a UDP packet to Machine B, which then sends the same data back to Machine A in a new UDP packet. Each run timed 100 of these ping-pongs, with the same configuration on each end. Under FreeBSD and Linux, the native UDP protocol stack was used, while a simple UDP implementation in the application was used for the OSKit and Fluke tests.

To isolate the Fluke IPC costs from the rest of the device driver overhead in the echo test, we also ran a test in which a single process sends the datagrams via IPC to a loopback server on the same machine. The loopback server simply buffers the data when written, and returns the same data when read. A single application did a write/read combination to the loopback server twice per "packet," which is the same number of IPCs done in the separate test case.

5.2.2.1 Results

The per-packet ping-pong times are shown in Figure 5.3, and the corresponding CPU utilization is shown in Figure 5.4. The OSKit achieves the lowest latency, but worse CPU utilization. Following the OSKit are the colocated Fluke configuration and the monolithic Unix operating systems, with the separate Fluke configuration achieving the worst latency results. With the exception of the separate Fluke

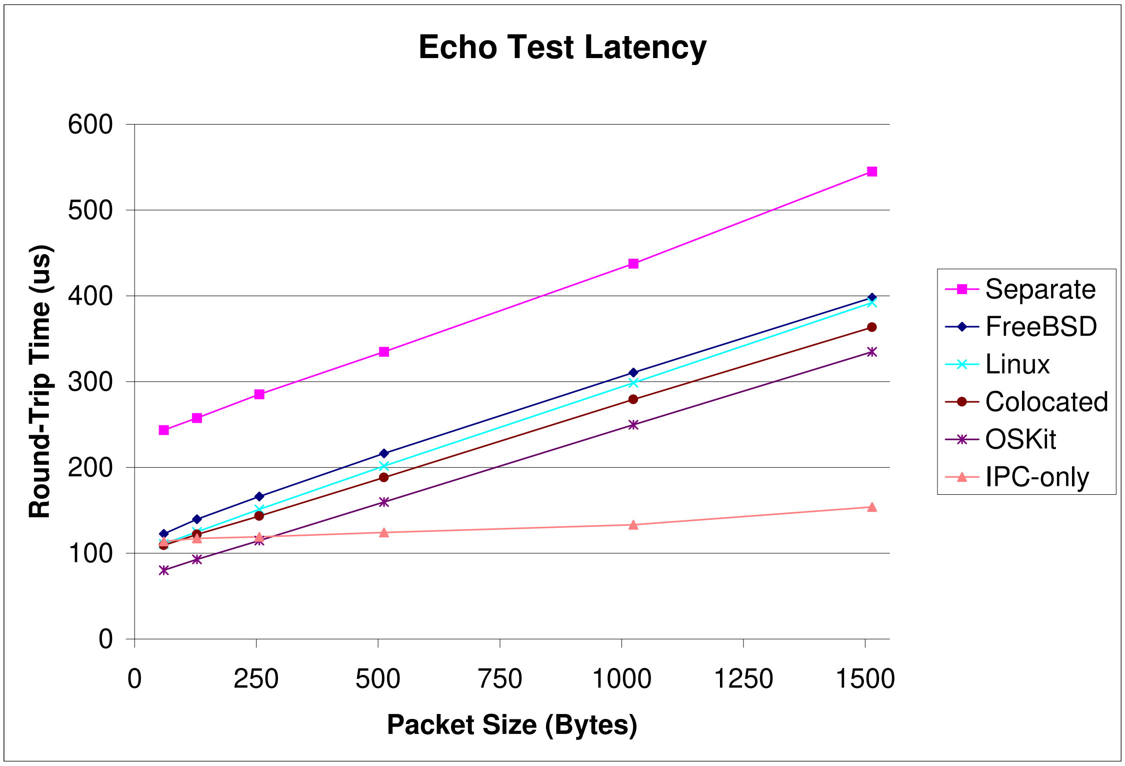

Figure 5.3. UDP round-trip time. Lower is better. Colocated Fluke does slightly better than the Unix systems, while the separate Fluke case has a much higher latency.

configuration, all of the operating systems perform approximately the same.

Fluke gets very good latency when running colocated with the device driver. The latency difference between the OSKit and the colocated echo test program is in the 20 to 30μs range, which is the additional overhead involved in moving the device driver to the Fluke user-mode environment, including interrupt delivery and thread synchronization.

An interesting observation in Figure 5.3 is that the loopback Ethernet device is actually slower (for minimum-size packets) than the colocated driver communicating over the 100Mb network. Even though IPC has more bandwidth available than a 100Mb Ethernet, there is more per-packet control overhead involved in processing the IPC-based requests than in transmitting and receiving a packet. The relatively poor performance is caused by inefficiencies in the IPC runtime layers, which copy the data and also create and destroy a connection for every transfer.

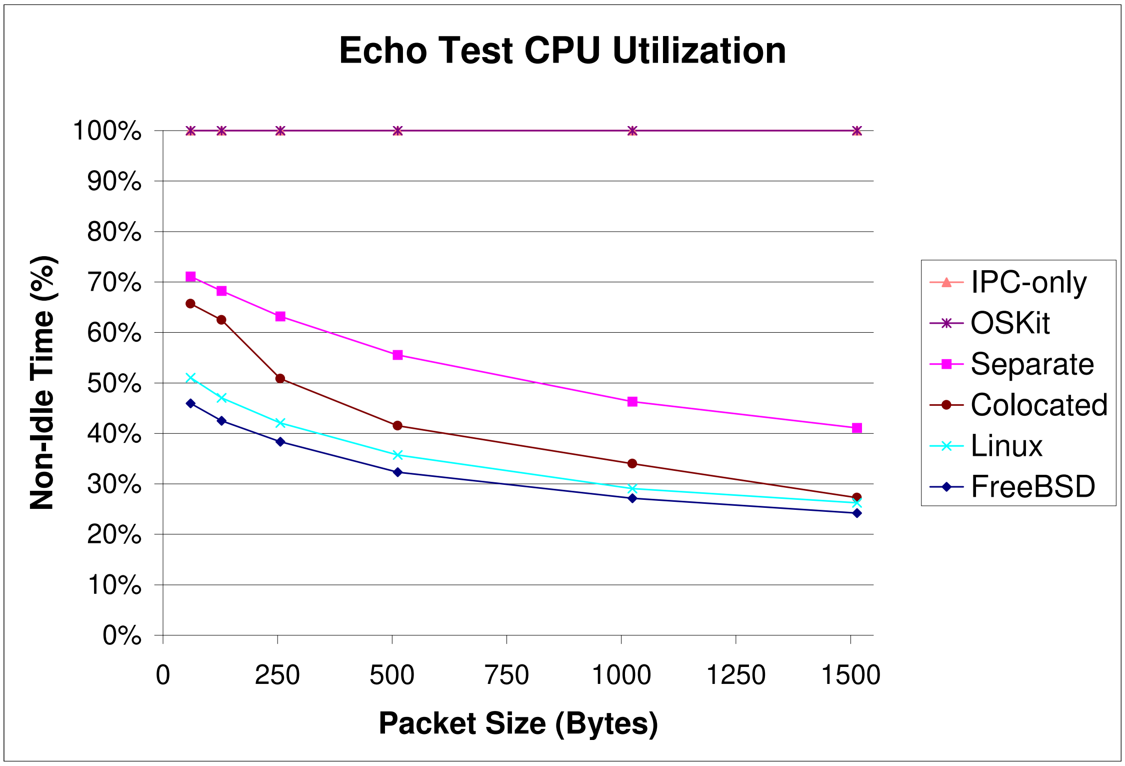

Figure 5.4. Busy time while sending and receiving UDP packets as fast as possible. Lower is better. The OSKit and IPC-only tests are pegged at 100%. The colocated Fluke case tracks Linux at all but the smallest packet sizes, while the separate configuration remains higher throughout.

The separate case, where the network driver is accessed through IPC, is much slower than any of the other cases. It has twice the latency for small packets. The high latency is due to the control overhead of Fluke IPC, both in the Flick/MOM runtime and in the kernel IPC path. The separate round-trip time is nearly identical to the colocated time plus the IPC-only time. The separate case consumes considerably more CPU time than the other cases, as shown in Figure 5.4. The CPU consumption is due to all the additional computation required for the IPC-based communication. Not only is a larger fraction of the time spent busy, but the test runs for significantly longer, as shown by the higher latencies in Figure 5.3. The drop in the colocated busy time at 256 bytes is due to scheduling interactions with the interrupt thread, which runs sooner with small packets. High-performance local IPC is crucial to achieving good performance for user-level device drivers.

5.2.3 Combination Test

Real applications and systems stress more than one device at a time. To measure the performance with multiple device drivers, we combined the disk and network benchmarks to create another synthetic benchmark, in which we sent the contents of the disk out the network interface.

Using the same block interface that we used for the disk read test and the same network code that we used for the echo test, a single thread read the first 10MB of the disk, in 80KB blocks, and sent out the data as 1322-byte packets, with a 1280-byte UDP datagram payload. This test amounted to 128 disk reads and 8192 packets per test run. These sizes were chosen to allow overlap of the network and disk activity (64 packets per disk block, which is less than the number of packets that can be buffered by the device drivers), and to provide an integral number of equally-sized packets per disk transfer.

Fluke results were obtained for drivers in the same address space as the application ("colocated"), for the Ethernet driver with the application but separate from the SCSI driver ("etherapp"), for the SCSI driver with the Ethernet driver but separate from the application ("scsiether"), and for each driver in an address space separate from the application ("separate").

5.2.3.1 Results

Figure 5.5 shows the bandwidth measured for each of the tested configurations, and Figure 5.6 shows the corresponding CPU utilization. Depending on configuration, Fluke achieves between 60–95% of the bandwidth of the best performing systems, the OSKit and FreeBSD. For the three noncolocated Fluke configurations, which must perform local IPCs, Figure 5.5 shows that the bandwidth drops more significantly if the *network* device driver is accessed through IPC ("etherapp" to "separate"), than if the *disk* driver is accessed through IPC ("colocated" to "etherapp"). In general, any configuration that does not perform an IPC to the network comes close to achieving the maximum bandwidth, although performing an IPC impacts the achievable throughput.

It is likewise notable that in the colocated case, Fluke edges out Linux in

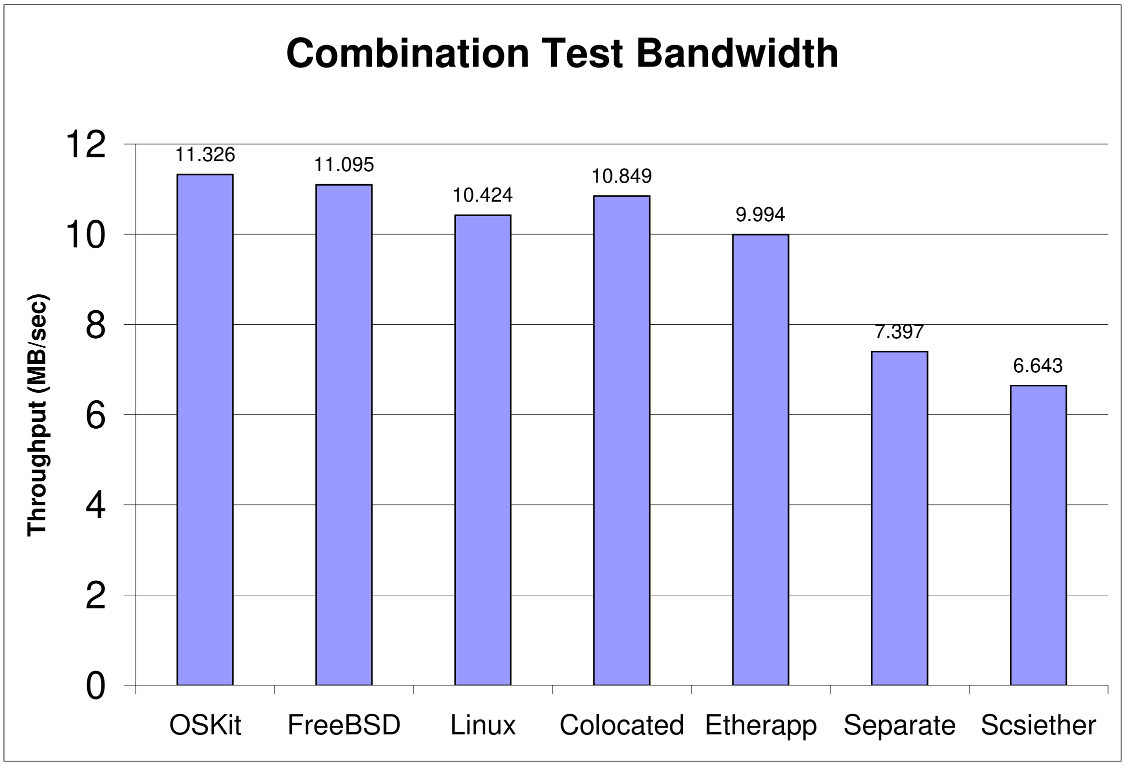

Figure 5.5. Bandwidth reading the hard drive and sending the data out the network in UDP packets. Higher is better. Only the two Fluke configurations that access the network device driver through IPC (separate and scsiether) do not achieve most of the possible bandwidth.

throughput, albeit at some cost in CPU utilization. Linux performed worse than FreeBSD for unknown reasons, most likely due to problems with the Linux UDP protocol layer. Additional speculation about the Linux performance is contained in Section 5.2.4.1. The "separate" case also outperformed the "scsiether" case, which is due to the increased contention between the device drivers when they are in the same address space.

The CPU utilization graph, Figure 5.6, does not contain any surprising results. FreeBSD leads the pack followed by Linux and then the colocated Fluke case. As more IPCs are performed under Fluke, the amount of time spent processing increases. The results follow from the previous tests: although the latency increased dramatically for the separate UDP-echo test, as shown in Figure 5.3, the separate disk read test was capable of maxing the drive at 16KB transfers, as shown in

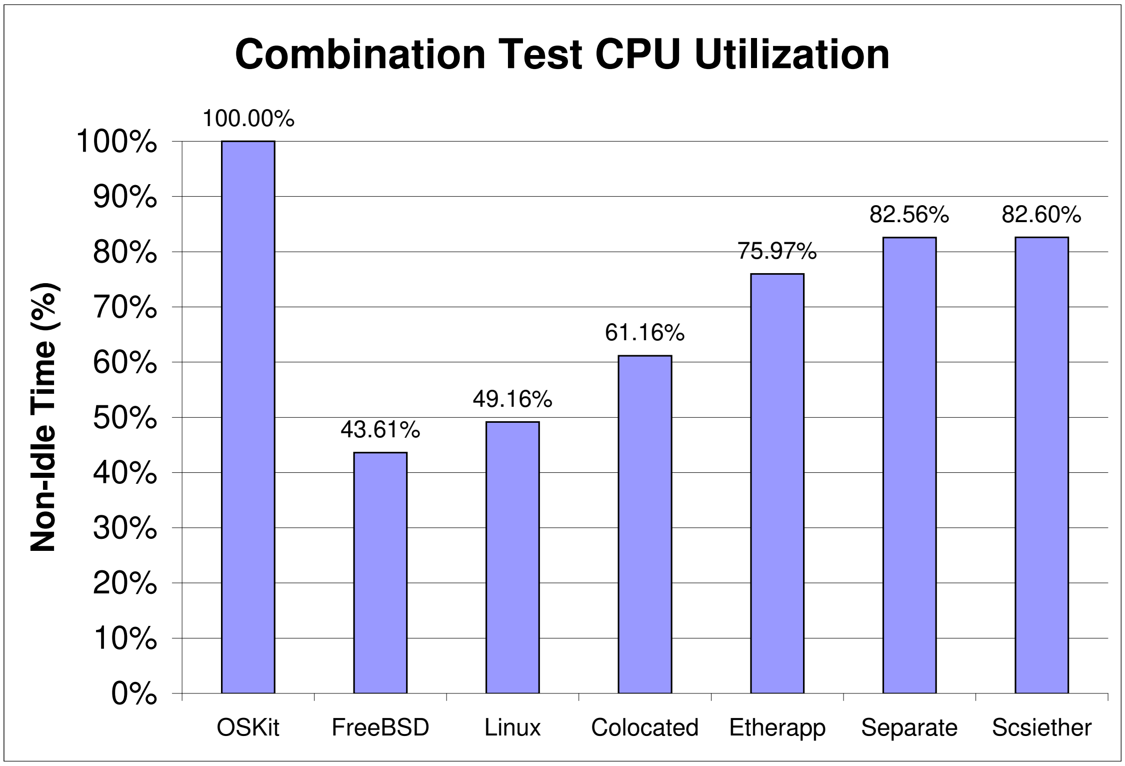

Figure 5.6. Busy time while running the combination test. Lower is better. The two Unix systems have the lowest processor utilization, followed by the colocated Fluke case, with the three Fluke configurations using IPC bringing up the rear.

Figure 5.1. The IPC overhead can more easily be absorbed by the large transfer sizes performed to the disk device driver, while the number of IPCs done to the network device driver causes significant bandwidth degradation. This overhead is partly due to the packet-oriented nature of the Ethernet IPC messages, where each local IPC transfers exactly one Ethernet packet, which are much smaller than the disk blocks. The network performance could be improved by sending multiple packets per IPC, or by increasing the maximum packet size (which is not practical for Ethernet).

5.2.4 Disk Trace Test

Even though the combination test evaluates both device drivers under load, with serialized sequential reads, it does not reflect typical usage, where disk access patterns are pseudo-random with multiple outstanding requests. With a disk seek between requests, the driver overheads become a much smaller part of the execution time. To measure the performance under a more realistic load, we developed a test that issued requests that are more reflective of a real workload.

Since Fluke does not have a fast multithreaded filesystem, using filesystem traces or benchmarks like the Andrew benchmark [32] to determine the performance of the device drivers on Fluke would produce results that unfairly penalize the Fluke device drivers. Instead, the approach we took was to use disk traces, which recorded the read and write activity to the disk after requests have been processed by a filesystem. This approach gives a level playing field for the device drivers across the multiple platforms, as variance due to the quality of the filesystem implementation is eliminated.

The trace used [30, 55] was collected by HP Labs over the course of one week in 1992 on a computer with a pair of 300MB HP335H hard drives with 256-byte sectors. It consists of 44,519 I/O operations. Since our disks have 512-byte sectors, we divided the sector numbers by two when replaying the trace. Changing the sector numbers did not distort the trace, since all accesses were to multiples of 512-bytes starting on even sector numbers.

The trace contained the request enqueue time, the completion time, the start sector number, and the length. We generated a list of request and completion events ordered by time, thus preserving the dependency information from the original trace. If the current event is a start event, we send the request. If the current event is a completion event, we wait until that request has completed before issuing the next request. This procedure allows the driver to process requests in a different order from the original trace, while still preserving the ordering and parallelism present in the original request stream. By removing the delays between the requests we compress one week of disk activity (approximately 18.6 minutes of I/O on the original machine) into about 4.5 minutes on a faster computer with newer hard drives, while still retaining the parallelism from the original trace. The trace contains as many as 102 outstanding I/O operations, although most of the trace consists of serialized requests.

We replayed the trace by sending requests embedded in UDP datagrams from a FreeBSD machine to another machine that processed the requests and sent replies. Replaying the trace on another machine allowed us to isolate the device driver activity from the processing required to replay the disk trace and provided an easy way to test the network and disk drivers being used together in a more realistic manner. To avoid needing to retransmit requests, which would skew the results, we increased the FreeBSD socket buffer size and removed the limit on the number of packets that Fluke and the OSKit could buffer. We also found it necessary to increase the number of hardware receive descriptors in the Fluke driver to a reasonable number to prevent packet loss due to interrupt latency.

Linux does not have a raw disk interface, which makes it impossible to bypass the buffer cache. With 128MB of memory available, most of the reads and writes for the 300MB drives are absorbed by the buffer cache, which would obviously skew the results. Since it is important to have a comparison with the same device drivers used in Fluke, we modified the Linux kernel in two carefully controlled ways to allow a comparison. First, to eliminate socket buffer overflows, we removed the check on the socket buffer size, as the maximum buffer size is not configurable under Linux. Second, we forced I/O requests to the block disk interface to behave as though they were to a *raw* disk device. We accomplished this by marking the buffer cache entry as invalid at the start of every read request, and by making writes be handled synchronously. Even though accesses to the disk device still go through the buffer cache, this was the only obvious solution that did not require enormous changes to the Linux kernel.

For FreeBSD and Linux, we forked off 100 server processes that serviced requests, while under Fluke we used 100 threads. Since the OSKit did not support multiple threads, it serviced requests sequentially using a single thread. We enhanced the simple UDP layer used for the previous tests to support IP fragmentation and reassembly, which we used for the Fluke and OSKit tests. When the UDP layer had reassembled a UDP datagram, it handed it off to a server thread, which handled the disk I/O and sent the reply.

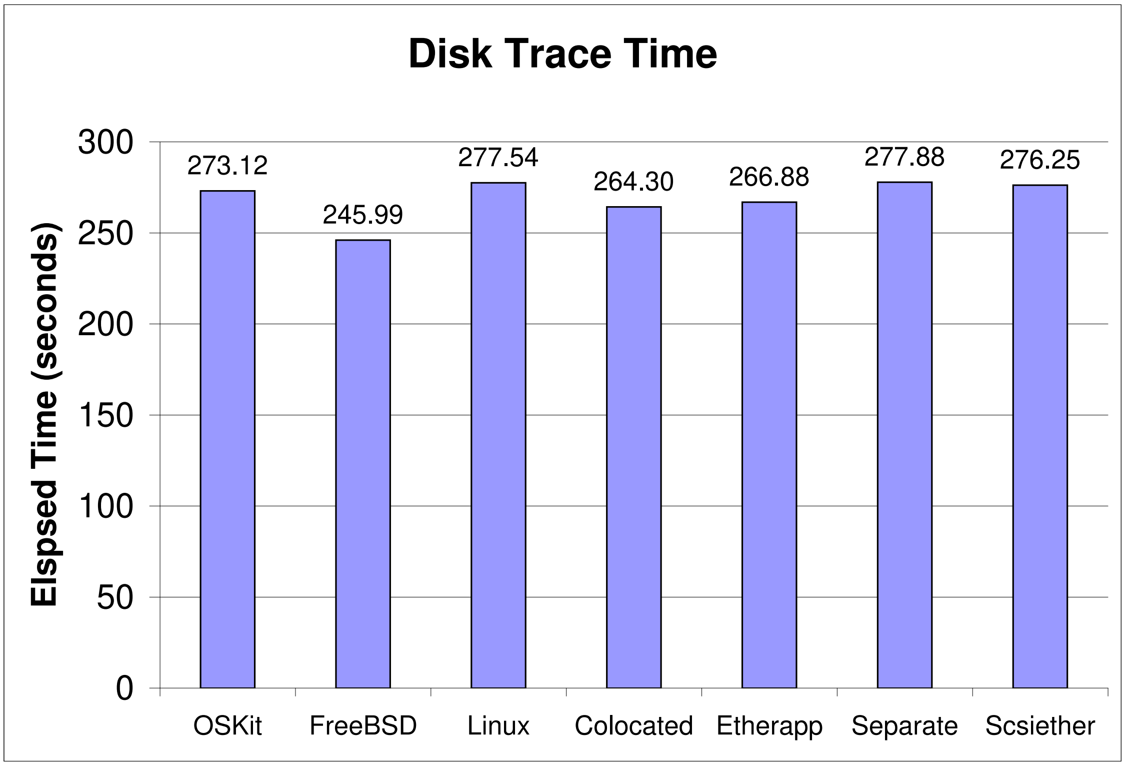

Figure 5.7. Elapsed time to replay the disk trace. Lower is better. Fluke, Linux, and the OSKit are within 5%, while FreeBSD is 11% faster than the slowest system.

5.2.4.1 Results

The results of this test are shown in Figure 5.7. Fluke is only 7–13% (depending on configuration) slower than the fastest system, FreeBSD. Also, as shown in Figure 5.8, Fluke's CPU utilization ranges from slightly less to twice as much as FreeBSD, depending on the configuration. Although this overall result is pleasing, the fact that FreeBSD is using a different base device driver confounds the analysis, as the earlier synthetic benchmarks demonstrated that FreeBSD has faster device drivers.

The CPU utilization for the IPC-based Fluke tests is much higher than for FreeBSD and the colocated case. However, compared to Linux, a monolithic OS using the same basic device driver, we find Fluke doing much better. Fluke, despite the overheads imposed by user-mode device drivers and IPC, outperformed Linux by 63–280%. Unfortunately, we suspect this result is due to other problems in Linux, not its device driver framework. Although we have not yet determined the

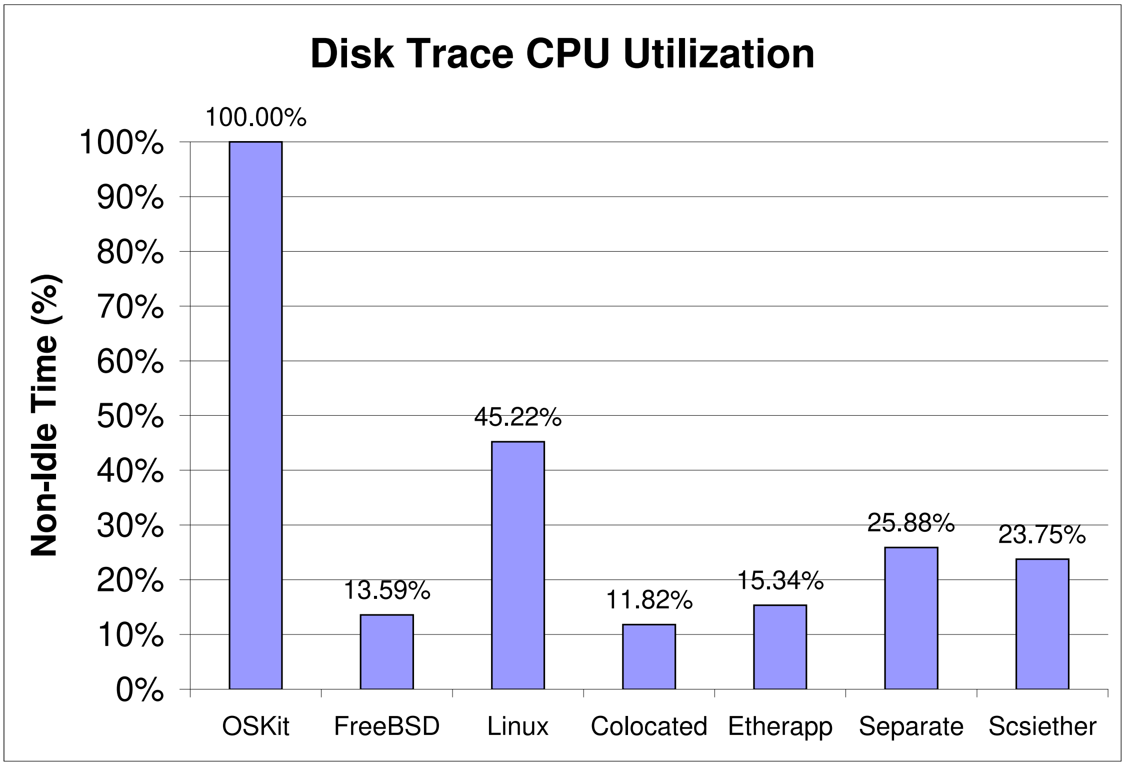

Figure 5.8. Busy time while replaying the disk trace. Lower is better. The separate and scsiether Fluke configurations have significantly higher CPU utilization than the FreeBSD, colocated, and etherapp configurations. Linux has an inexplicably high CPU utilization.

exact cause, we suspect it has something to do with one or more of the following: the "thundering herd" phenomenon (all blocked processes are awakened on every incoming message), some unknown problem in the Linux UDP protocol stack, an unknown problem with Linux disk writes (the microbenchmark only tested reads), or some problem with the buffer cache code. Given that Linux does well on the microbenchmarks, we believe it is most likely due to the thundering herd phenomenon or a buffer cache interaction problem.

5.2.5 Clock Interrupt Test

Since delivering interrupts to user mode and synchronizing between process-level and interrupt-level activity can be expensive, we quantified that overhead. Measuring the processor overhead associated with contested and uncontested interrupts gives us a basis for determining how well the implementation scales to high interrupt rates. We measured the processor overhead by measuring the idle cycles on an idle machine processing clock interrupts in the device drivers.

Unlike the other tests, this test was only run on Fluke; the test involved running multiple copies of a simple device driver, with one driver per address space. This test allowed us to measure the overhead at higher interrupt rates and with multiple device driver processes. For the duration of the test, each device driver did nothing except increment a counter every 10ms in the clock interrupt handler. Each run lasted one second, covering 100 interrupts per device driver. The processor utilization was measured with one to six different device drivers running, both with a process-level thread needing to be stopped by the interrupt handler ("contested"), and with no process-level thread running ("uncontested"). To measure just the interrupt-handling overhead, the contested case was artificially created by having a thread intentionally block while holding the process-level lock.

5.2.5.1 Results

Based on a linear extrapolation of the overhead in processing the clock interrupts, shown in Figure 5.9, a 300MHz Pentium II would become saturated at around 20,000 interrupts per second with contention, or at 50,000 to 65,000 without contention. Interrupt handling takes approximately 0.15% of the CPU per 100Hz for the uncontested case (about 15μs/interrupt), and about 0.5% per 100Hz for the contested case (about 50μs/interrupt). However, those numbers do not take into consideration two counteracting issues. First, a real device driver will do some processing in the interrupt handler, unlike the drivers used in this benchmark. Second, the overhead for handling contested interrupts can be significantly reduced by implementing the synchronization optimizations discussed in Section 4.3, which would decrease the contested case overhead to be near that of the uncontested case.

It is further interesting to note that the maximum number of contested interrupts that get processed is related to the scheduling frequency, f, and the number of interrupt handlers in the system, p. This result follows from the observation that a process-level thread will not get preempted more often than f times per second, so a corresponding interrupt thread will not have to stop a process-level thread

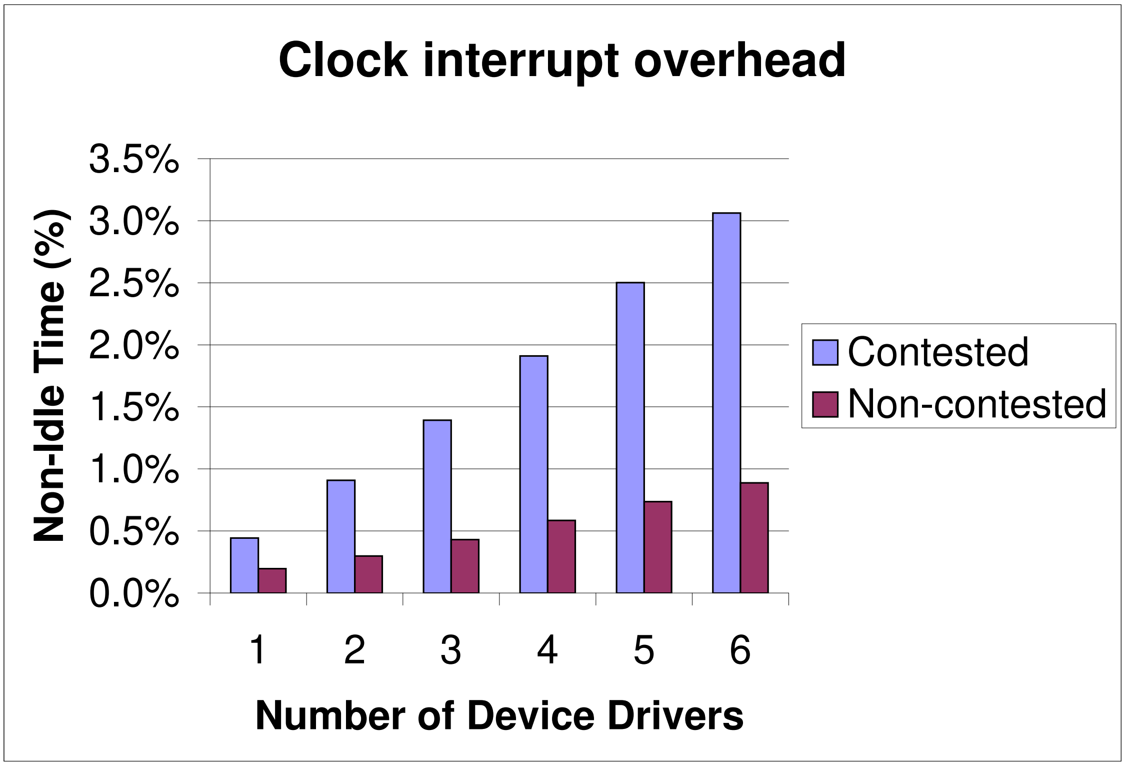

Figure 5.9. CPU time spent processing clock interrupts. Lower is better. Processor utilization is three times as high for the contested case, and scales linearly with interrupt handling frequency.

more often than the scheduling frequency. It is conceivable that every interrupt handler would get to run on a clock interrupt, and each interrupt handler would need to stop a process-level thread holding the lock, so the worst-case upper bounds for contested interrupt processing would be approximately $p * f$ times per second. Interrupts can be handled more frequently, due to a high scheduling rate, but that would only occur if there is idle time in the system, which would indicate that the process-level threads are blocked and not holding locks, as they normally block without holding the process-level lock.

5.2.5.2 Interrupt Latency

The interrupt latency is related to the scheduling algorithm and the preemption frequency. Although it would be possible to always preempt the process-level thread on a hardware interrupt, the extra synchronization required to stop the process-level thread could be prohibitive. With the current implementation, it is possible for an interrupt thread to be delayed for up to M preemption cycles, where M is the number of CPU-bound processes in front of the thread in the run queue. Under heavy load, interrupts will not be handled as promptly. Fortunately, the ability to defer interrupt processing allows high-priority threads to run before low-priority interrupt handlers.

Under moderately light load, as with the colocated echo test program (Figure 5.3), interrupt latency is very good. The extra cost of scheduling a user-mode thread and the associated synchronization cause the colocated case to run more slowly than in the OSKit, but it still outperforms FreeBSD and Linux, due to lower overhead in the networking stack and the lack of a kernel/user copy.

5.3 Summary

The results have demonstrated that the performance of the prototype device driver framework is acceptable. User-mode device drivers on Fluke, when they are colocated with the application, offer the performance of in-kernel device drivers on Unix. The colocated benchmarks show the Fluke driver framework performing at least as well as Linux, the source of the device drivers. Compared to Linux, colocated Fluke gets a 5% higher transaction rate on the disk trace test, 4% greater bandwidth on the combination test, and 1.5–7% lower latency on the UDP-based echo test.

However, the high cost of IPC in Fluke reduces the device driver performance and increases the processor utilization when the drivers are accessed through IPC. IPC is necessary when one must isolate the device driver from an application, but IPC does add significant latency. When accessing the driver through IPC, the echo test has 40–120% higher latency than Linux and the combination test has 29% lower bandwidth. The disk trace achieves the same throughput as Linux because the disk seeks dominate the execution time. Reducing the number of IPCs, such as by transferring more data per IPC, would reduce the cost of the IPC to the device driver.

CHAPTER 6

RELATED WORK

In this chapter we discuss some related efforts to build user-level device drivers, as well as a few even more closely related efforts that use legacy drivers with research kernels. Naturally, almost all of the efforts were based on microkernels, where it is more natural to locate components outside the kernel, and we begin with those.

6.1 Mach

In Mach [1], device drivers are normally placed in the kernel. However, there have been a few efforts to move the device drivers out of the kernel and into user-mode servers. Some of the earliest work on user-mode device drivers was done by Forin and others using the Mach microkernel [24]. Their major focus was not on the user-mode aspect but rather on dividing the functionality of the device drivers into device-specific routines (bottom layer) and more general routines that can be used by several device drivers. A small amount of device-dependent code was used for each different device, while all devices of the same type (e.g., SCSI or terminal) used the same higher-level code. They were able to reuse the kernel-mode drivers in user mode by providing "scaffolding" and utilizing one thread per device driver. This approach is similar to the approach we took, although we used multiple threads because of runtime constraints. Even though the focus appears to have been on sharing code between drivers, they did claim to have doubled the networking throughput of 10Mb Ethernet (from 120KB/sec to 230KB/sec) and increased the disk throughput (from 700KB/sec to 850KB/sec). They did not discuss the impact their drivers had on CPU utilization.

Golub and IBM extended this work by adding a so-called Hardware Resource Manager (HRM) that arbitrated access to devices [27]. Multiple user-mode device drivers used the same hardware by attaching and unattaching the drivers as necessary to share the resource, such as the (floppy controller based) tape backup driver sharing the controller with the floppy disk driver. The HRM is similar in function to the Fluke *device server*, although the Fluke device server does not handle revocation or sharing of resources between drivers.

With the HRM design, it would have been possible to link the device drivers in with the programs that wanted to access the device, and to use the HRM to arbitrate access. However, because Golub et al. wanted the device drivers to be in separate protection domains from the applications, and because the arbitration cost was relatively high, they never utilized the arbitration capability to colocate device drivers with applications.

The Golub/IBM paper reported no performance results, at least partly because the system was not yet fully implemented. However, anecdotal reports [38] indicate that performance was dismal; perhaps half the speed of in-kernel device drivers.

6.2 The Raven Kernel

The Raven kernel [53] ran user-mode device drivers in a multiprocessor kernel. Its solution to the preemption problem was (at least partly) solved by *not* arbitrarily scheduling user-mode tasks. When the clock interrupt occurred, a task that held a mutex was not scheduled; rather, the Raven kernel deferred scheduling a mutex-holding task until its mutex lock count was zero. Preemption was deferred to prevent a thread from being blocked on a mutex owned by a nonrunning thread.

In Fluke, the problem of a thread blocking on a lock owned by another thread will be solved in a general way when Fluke's CPU inheritance scheduling framework [23] is completely implemented (by others). With CPU-inheritance scheduling, when a thread requests a mutex being held by another thread, the thread that owns the mutex is scheduled instead of the current process. CPU inheritance allows good performance without arbitrarily blocking other processes. Unlike the Raven kernel, Fluke does not provide a way for a thread to avoid being preempted.

6.3 QNX

The QNX [31] microkernel-based operating system allows a privileged user process to register for a hardware interrupt vector. Drivers, as user-mode processes, can be dynamically added and removed. The interrupt handler can directly access anything it desires in the application that contains it.

Since the drivers are designed for QNX's message passing mechanism, data can be directly written to the device using the IPC with the scatter-gather mechanisms. The drivers combine the IPC copy with the single copy to the device. In Fluke, we cannot do that due to the use of encapsulated device drivers and limitations in the runtime support code.

QNX has a significant speed advantage over Dell's SVR4 [31]. However, much of that advantage may have been due to the smaller size of QNX, which made it easier to optimize. It is unclear how much advantage QNX would have over a more well-tuned monolithic kernel, such as FreeBSD.

6.4 V

The V distributed system [7] is very similar to most other microkernels in that device drivers are still in the kernel, but filesystems and protocol stacks are not. V runs with a single kernel stack per processor, very much like the nonblocking Fluke kernel. It also placed much of the operating system in user-mode servers, including filesystems and the networking protocols. However, since it could not guarantee that a user-mode device driver would not corrupt the kernel with an incorrect DMA transfer, V kept the device drivers in the kernel, accessed through IPC. The rationale seemed to be that since the device driver had to be trusted, it was better to keep it in the kernel where trust is explicit.

6.5 U-Net

In U-Net [63] on SunOS, user-mode network device drivers were linked directly into the TCP/IP stack when possible. Using ATM interfaces with an i960 co-processor, they were able to export the device interface directly to the application. However, with Ethernet devices, they added special system calls and packet filters to demultiplex packets to the applications, as demultiplexing could not be done safely in user mode. Fluke allows the device driver and networking stack to be linked with a single application, since hardware constraints prevent the Ethernet devices from being shared.

6.6 Exokernel

Other research microkernels besides Fluke are using some OSKit device drivers. One such kernel, the MIT Exokernel [15], resembles Fluke in that it presents a kernel execution environment that is a mismatch for the OSKit drivers. The Exokernel uses the nonblocking driver/kernel model, while the OSKit drivers derive from a process-model kernel and may block. Unlike our development of the Fluke device driver framework, the Exokernel developers did not develop general support for such legacy drivers. Consequently, the Exokernel can use only the network drivers, which do not block, and not the disk drivers, which do.

In Fluke's nonblocking kernel configuration, kernel threads can still block and be forced to restart. However, the Exokernel is carefully written so that the network drivers do not block, which is possible because routines such as memory allocation are nonblocking. Fluke does not have this feature and can block on kernel memory allocation.

6.7 L^4Linux

L^4Linux [29] is the Linux kernel, including device drivers, running in user-mode on top of the L4 microkernel. Synchronization is achieved by single-threading the Linux server and disabling processor interrupts. A dedicated thread handles hardware interrupts, blocking until L4 indicates to it that an interrupt has occurred. Another dedicated thread processes software interrupts. This implementation is much different from Fluke's, which does all software interrupt processing in the interrupt handler thread, as the OSKit abstraction does not export software interrupts to the OS. However, in Fluke, the software interrupt handlers do not have much to do, as protocol processing and other work is done in separate threads or outside the device driver.

In L^4Linux, interrupt threads run at a higher priority than kernel threads to avoid concurrent execution on a uniprocessor. L^4Linux actually uses four different priorities to schedule the two interrupt priorities, the kernel, and Linux user-mode threads. In Fluke it is not practical to rely on scheduler priorities, because it is possible for an interrupt handler to block. A general problem with scheduler priority schemes is that such schemes alone will not support multiprocessors, as additional synchronization must be added to prevent multiple threads from being executed in parallel.

6.8 Windows NT Driver Proxies

Writing device drivers for Windows NT is difficult because of both the execution environment and the programming environment [33, 62]. To simplify the task of writing a device driver, Hunt developed a kernel proxy [33] that forwarded requests to a user-mode device driver. Only the kernel proxy needs to be fully reentrant and interface with the kernel environment. The device drivers themselves may be written using any available tools or libraries. By moving the device driver to user mode, Hunt was able to avoid the restrictive kernel programming environment. His work focused on filesystem device drivers, and did not support hardware device drivers. He was able to rapidly develop new filesystems, including a network filesystem based on FTP.

Along similar lines, KRF*Tech* provides driver stubs that enable device drivers for Windows 95/98/NT and Linux to execute in user mode [36]. User-mode drivers are accomplished by providing a proxy inside the kernel that relays messages to and from each driver. This is conceptually quite similar to Hunt's work with kernel proxies, but unlike Hunt's NT proxy for filesystems, KRF*Tech* has added features to support hardware device drivers in addition to filesystem drivers.

6.9 Nemesis

Nemesis [6] is a microkernel-based operating system designed to support multimedia applications. It is a vertically-structured operating system: the applications do as much processing as possible in their own address spaces, which allows Nemesis to more accurately track resource usage by processes. Accurate accounting in turn allows Nemesis to provide consistent Quality of Service (QoS).

Nemesis uses a combination of Linux device drivers ported to Nemesis and new device drivers. Even though the device drivers normally reside in the kernel, the Nemesis developers would like to run the device drivers in the applications safely. To that end, they have come up with the idea of user-safe devices [51]. User-safe devices are ones that can safely multiplex their resources among multiple applications. Since user-safe devices do not generally exist, they are emulated by the kernel device driver providing a user-safe interface to applications. This approach is similar to U-Net's "emulated" user-mode network driver.

CHAPTER 7

CONCLUSIONS AND LESSONS LEARNED

As a result of our work, we have learned much about the design and implementation of device driver frameworks. This chapter summarizes our major conclusions, lists advantages and disadvantages of user-mode device drivers in general, lists some lessons learned about the good and bad aspects of the base Fluke system, and outlines some problems with encapsulated legacy device drivers. It then gives a set of simple guidelines for writing flexible device drivers, and concludes by suggesting some future work.

7.1 Conclusions

1. Four issues dominate the design space when providing legacy device drivers to foreign execution environments such as the Fluke kernel. They are *synchronization, interrupt delivery, memory allocation,* and *access to shared resources.* Each of these issues presents subtle design and implementation choices as well as performance challenges. We found that the choices and performance challenges can be largely resolved through careful experimentation and analysis.

2. Executing legacy device drivers in *user mode* is an important technique for isolating them from the idiosyncrasies of the kernel execution environment. Decoupling the notions of "kernel/user address space" from "supervisor/user privilege" adds flexibility in supporting legacy device drivers, but is not essential.

3. *Acceptable performance* of such drivers is achievable. "Acceptable" means between 88–93% of the best-performing Unix system in a realistic disk workload and between 60–95% of the best-performing Unix systems in synthetic scenar-

ios typical of real use. The lower range in performance occurs when using local IPC to communicate between the driver(s) and applications. When drivers are colocated with applications, they typically match Unix performance. *Processor utilization* by user-level drivers is as much as twice as high when drivers are not colocated, but the drivers are still usable.

4. *Local IPC costs*, both kernel and runtime, are the main culprits in limiting performance and increasing processor utilization. Kernel IPC on Fluke is faster than IPC on most traditional operating systems like Unix and faster than IPC on "first generation" microkernels such as Mach [21]. However, Fluke was written in C to be portable, so does not contain the IPC optimizations of recent "second generation" microkernels such as L4 [29]. When the kernel costs are added to the large slowdowns incurred by the current runtime layers, IPC adds considerable overhead for high-performance device drivers.

7.2 Benefits and Downsides of User-Mode Device Drivers

The only way a single device driver framework can support all of the internal Fluke kernel configurations is by running the device drivers in user mode. Running device drivers in user mode offers numerous advantages and some disadvantages, which we list here.

7.2.1 Benefits

Kernel implementation: As we have seen, the kernel can be implemented using unusual programming models, without requiring all the OS components to be written for that model.

Simplified kernel: Placing functionality in servers allows the microkernel to be smaller and potentially more optimized and more predictable in execution time than a monolithic kernel. Components are also simpler and have well-defined entry points. Modular, smaller pieces are easier to understand and debug.

Debugging: "Normal" user-mode applications are certainly easier to debug than the OS kernel. User-mode debuggers and tools can be used, rather than the usually limited kernel debugging tools. However, device drivers are not "normal" applications and others' experience has shown that debugging OS servers running on a microkernel can be more difficult than debugging a monolithic kernel, because control and state are spread over many protection domains. Our limited experience with Fluke leaves us neutral on this issue. However, it is certainly true that the amount of damage a programming error can inflict is reduced, since the kernel and other servers are in different protection domains.

Scheduler control: Server threads are scheduled along with those of other applications. As a result, the scheduler has control over more of the processor time. which increases scheduling determinism and flexibility.

Parallelism: Even if each component is single-threaded, multiple components can execute in parallel, unlike in a monolithic single-threaded system. Increasing parallelism is particularly advantageous when migrating to a multiprocessor. User-mode device drivers can take advantage of increased parallelism without the complexity and implementation burden of a fully multithreaded monolithic kernel.

Trust/Security: User-mode servers are subject to the same security policies and mechanisms used to control other applications, since the servers are not part of the kernel. Trust domains may be smaller than with monolithic kernels, and may be reinforced through the use of hardware protection domains. Assurability of the system is increased since the smaller components can be independently analyzed and verified. However, since hardware transfers data using physical addresses, device drivers that perform DMA (direct memory access) must still be trusted not to corrupt the system.

Memory use: Normally the kernel's code and data are stored in "wired" memory. Since user-mode components are not in the kernel, unused regions can be paged out the same as normal applications.

7.2.2 Downsides

Performance: Performance has historically been the biggest problem with user-mode device drivers. Although previous user-mode device driver work has been weak on the performance analysis, there are anecdotal reports of a 2X slowdown caused by user-mode device drivers [38]. Performance problems have been caused by cross-domain copies (IPC), indirect access to the hardware, and interrupt delivery to user mode. One of our contributions is a user-mode device driver framework that alleviates many of the performance problems normally associated with user-mode device drivers.

Complexity: Although we find the problem hinted at only once in print [4], a disadvantage of component-based systems rarely recognized or acknowledged is the additional complexity involved in a large microkernel-based operating system. We base this observation on our experience with Fluke and that of many members of our research group with Mach, the Hurd, and Fluke. Even though individual components may be simpler, the aggregate complexity when all the components of a large OS are assembled is typically significantly greater than the equivalent monolithic system. This complexity leads to increased implementation and debugging difficulty.

7.3 Advantages and Disadvantages of the Fluke Model

With the exception of the performance implications, Fluke supports device drivers quite well. Among the positive, though not novel, features of Fluke is the ability for processes to easily mount devices into the filesystem transparently. We use the same interface for network devices as for block devices, which also makes it easy to get a handle on a device driver in another process. Having network devices in the filesystem contrasts with Unix, where there is no `/dev/eth0` device, although there is a `/dev/sd0` device for direct access to the disk drive.

Placing device drivers in user mode makes them independent of the Fluke kernel implementation, except for a small amount of support code necessary to dispatch interrupts. Fluke is defined by the exported kernel API, not the programming model. Therefore, a completely new implementation of the kernel could use the device driver framework immediately.

Our heavy use of Fluke primitives and our high number of context switches make us highly dependent on the performance of those items, although we do not characterize this impact. This dependency is especially true for the colocated case, where we do not have IPC overhead dominating the performance of the other primitives.

Despite this dependence on Fluke primitives, the performance-related problems are mainly related to IPC. The kernel's IPC is simply too slow. Even a null kernel IPC takes far too long, due to all the code executed for control flow. While at larger transfer sizes the copy time dominates the CPU overhead, eliminating data copies (such as through the use of a shared buffer) would not be much of an improvement for small transfers, as the vast majority of small-message time is spent in control code and context switching, not in data copies.

The Fluke IPC runtime, comprised of Flick and MOM, also adds considerable overhead, both in control logic and in unnecessary copies. Although MOM and Flick provide a nice runtime abstraction, it is not at the appropriate level for copy-free transfers. The upshot is that for small transfers (such as 60-byte Ethernet packets), around 60% of the IPC transfer time is spent in the runtime, rather than in the kernel.

Although the runtime negatively impacts performance, the ability to add IDL interfaces, such as multiple-packet reads and writes, with only minor modifications and the addition of a new IDL interface definition, is quite advantageous. This flexibility will allow the expensive IPC path to be amortized by sending multiple packets per transfer.

Besides being slow, we also have a problem with the nature of Fluke IPC, which transfers words instead of bytes. Word-based IPC requires extra copies to deal with such things as adding a 14-byte Ethernet header. Even if Flick and MOM did not have to do any copies, the runtime would still need to copy data so that the buffers would all be a multiple of the word length. For example, data fragments are copied on the client side to ensure they are contiguous during the IPC transfer, mainly because of the word-based IPC requirements of Fluke. However, the support for persistent connections, while not currently used, would reduce the kernel overhead by allowing multiple transfers to occur per connection.

In summary, improving the performance of the IPC path would be a tremendous performance boost, provided both the kernel and the runtime were improved. For small transfer sizes, most of the slowdown versus Unix is mainly due to the slow IPC execution path compared to the `copyin` routine in a Unix kernel, and not to extra copies performed along the Fluke IPC path. This slowdown is exacerbated because the runtime establishes and destroys a "connection" on every data transfer, even though the Fluke kernel supports persistent connections. For larger transfer sizes, the control overhead is amortized, and the dominating factor becomes the data copies.

7.4 Encapsulation

Encapsulating legacy device drivers certainly allowed us to provide more device drivers in Fluke more quickly than would have been possible otherwise. However, there were also a few downsides to using encapsulated drivers.

The lack of an asynchronous interface for OSKit block devices requires that servers be multithreaded. Each outstanding request requires a separate device driver thread to provide a context for blocking. The right way to solve this problem is to add an asynchronous device interface to the OSKit by writing more glue code and exporting an interface to the low-level disk "strategy" routine.

Atomic processing of interrupts is required by most drivers, and so we must provide that guarantee for all of them. Providing this guarantee introduced additional complexity and overhead into the design that would not have been necessary without that constraint.

The OSKit block device glue code, written when the device drivers were encapsulated, performs copies and breaks requests into small chunks for sequential processing. This design was due to the fact that buffers passed to the glue code are not guaranteed to be in wired physical memory. This problem can be fixed through modifications to the glue code and changes in how buffers are passed into the device drivers.

Throughly encapsulating the drivers for the OSKit was difficult. In a few cases we had to modify the device drivers, while in other cases (such as ISA DMA), we simply ignored the required abstractions due to the pervasive changes that would have otherwise been required. Such a cavalier approach often does little practical harm; in the ISA DMA case, the only effect of ignoring that particular encapsulation constraint is that all drivers that use ISA DMA must be linked into the same program. However, for the typical interfaces that did not require pervasive changes, the advantages of providing a single simple interface, `osenv`, outweighed the work required to encapsulate them.

7.5 Device Driver Guidelines

If all device drivers followed a common set of rules, it would be much simpler to support not only user-mode device drivers in Fluke, but the `osenv` device driver framework in other operating systems as well. Below are some of the observations that we have made in creating the `osenv` framework and in adapting that framework to Fluke. We make no attempt to provide a complete device driver specification, such as UDI [56], but rather merely provide guidelines for writing flexible device drivers. We focus on single-threaded device drivers, as those are the ones that we have adapted so far. Multithreaded device drivers would more closely match the Fluke environment, and it is expected that many of these issues are already dealt with by multithreaded device drivers.

The rules we propose are the following:

1. Do not directly manipulate the interrupt enable bit—use a macro defined in a single header file. Over time, the base Linux device drivers have improved considerably in this regard. Efforts to port the drivers to different architectures eliminated most of the embedded assembly used to manipulate the interrupt flags that plagued earlier versions of Linux. It would not have been feasible to run the device drivers in Fluke without using external synchronization routines, which requires replacing the interrupt management routines.

2. Do not assume that the driver's virtual memory address maps directly to the device's physical memory address. The bus may be able to do translations, as

on the Alpha's PCI/ISA busses. Since the drivers assumed there was no conversion process, and that `malloc`ed memory was nonpageable, our framework had to meet that requirement, which limited flexibility.

3. If there are any timing-critical sections of code, indicate the constraint, preferably with a function call, instead of just disabling interrupts. Disabling interrupts actually works fairly well for synchronization—as long as there is only one active process thread. However, it is inadequate for timing problems, since disabling interrupts may not normally disable all of the interrupts; e.g., the clock may be left running. Also, disabling interrupts does not disable any of them in our user-mode driver framework, nor does it prevent preemption. Although this driver framework does not currently support drivers with timing-critical sections, supporting such drivers is tractable only if it is clear which code sections have this requirement.

4. Use a standard macro or function call for all accesses to shared resources, such as the timer or a DMA controller. Do not assume that all other activity in the system is suspended while the driver is executing, even if it is single-threaded. This requirement is essential to running device drivers in separate address spaces. Although most shared resources were sufficiently abstracted, ISA DMA was not.

5. Use a function to enter a critical section, instead of assuming that interrupts are atomic. If fine-grained locking is too slow, then the driver should put the whole interrupt handler inside a critical section, but should not assume that interrupts are necessarily handled atomically. If the driver is more explicit about the synchronization, additional optimizations (and simplifications) can significantly reduce the synchronization overhead.

6. Use a different memory allocator for memory that needs to be physically addressed than for memory only accessed through virtual addresses. The use of separate allocators (or allocation flags) would increase the amount of the

driver memory that is pageable by the system. The current drivers do not make the distinction clear, which requires that more nonpageable memory be allocated than is necessary.

These guidelines not only help run device drivers in different execution environments, but they also simplify the task of incrementally multithreading a monolithic kernel.

7.6 Future Work

The best way to quickly improve Fluke so that it better supports legacy device drivers is to change the IPC granularity from words to bytes. If a data transfer must do a copy, scatter-gather should be done there at a byte granularity. This is true even though word-aligned, word-based transfers provide greater efficiency when byte-based scatter-gather is not required.

Improving the performance of Fluke IPC and the associated runtime support code would also provide a large performance boost. Improving Flick and MOM to use Fluke IPC scatter-gather when possible, instead of copying, is an important part of that. Implementing a multiple-packet IPC interface would improve network performance, and implementing packet filters would increase the functionality of the system. Optimizing the synchronization code is another obvious source of potential performance enhancements.

Fluke and its supporting runtime should also improve their handling of memory and buffers, perhaps through the implementation of a zero-copy framework similar to *I/O-Lite* [49] or *fbufs* [13]. It is currently very difficult to know what type of memory to allocate in one component (such as the networking stack), since that depends on whether it is colocated with a device that uses DMA or whether it is communicating with the device over IPC. The same problem arises in MOM/Flick, where the MOM runtime receives data into a buffer and Flick unmarshals the data into an allocated buffer. The device driver glue code then has to copy the data into a wired buffer, just in case the driver needs to perform DMA or hold on to the buffer. Passing around immutable fixed buffers would eliminate the copies and the need to marshal the data for an IPC.

Relatively straightforward improvements include supporting mapping of high physical addresses for memory-mapped I/O and providing more robust support for shared level-triggered interrupts. Implementing more checks and utilizing additional information in those checks (such as using information from the PCI configuration registers), would also be straightforward. Adding additional checks and access controls for the highly-secure version of Fluke, called Flask [58], would further increase the security of the device driver framework.

Nemesis's postulated "user-safe devices" [51] would allow multiple processes to share a device without the expensive communication costs that we currently pay. Because our device driver framework already supports drivers colocated with applications, we are prepared to support such user-safe devices.

Finally, desirable future work includes writing a customized multithreaded device driver for Fluke that uses hand-rolled stubs and does no unnecessary copies. This driver would allow us to take full advantage of the Fluke features without the overhead of using encapsulated device drivers. Additionally, it will allow precise measurement of the performance impact of the runtime, which currently is confounded with the base Fluke IPC mechanism.

APPENDIX

OSENV API

This Appendix gives an overview of the functional interfaces provided by the operating system as part of `osenv`. The `osenv` device tree functionality is not included, along with the other OS entrypoints into the device drivers. Instead, this section focuses on the OS-provided interface layer used by the device drivers. For more information on the `osenv` API, please consult the OSKit user's manual [16].

A.1 Interrupts

Interrupts are used by hardware to inform the device driver that it needs servicing. `Osenv` routines that process a single interrupt vector contain *irq* in the name, while `osenv` routines that manipulate interrupts in general contain *intr*.

A.1.1 osenv_irq_alloc

`int` **osenv_irq_alloc**(`int` *irq*, `void` *(*handler)(void *)*, `void *`*data*, `int` *flags*);

Allocate an interrupt vector. The specified handler is called with `data` as its parameter when the hardware interrupt occurs. The `flags` parameter currently only specifies if the device driver wishes to allow the interrupt to be shared with another device (by passing `OSENV_IRQ_SHAREABLE`). An error code is returned if the interrupt cannot be allocated.

A.1.2 osenv_irq_free

`void` **osenv_irq_free**(`int` *irq*, `void` *(*handler)(void *)*, `void *`*data*);

Free an interrupt vector. This function is used to unregister the interrupt handler previously registered with `osenv_irq_alloc`.

A.1.3 osenv_irq_disable

`void` **osenv_irq_disable**(`int` *irq*);

Disable only the specified interrupt. It is acceptable to merely prevent this interrupt from making it into the driver set. This routine corresponds to disabling the interrupt source at the interrupt controller.

A.1.4 osenv_irq_enable

`void` **osenv_irq_enable**(`int` *irq*);

Enable the specified interrupt. This routine corresponds to enabling the interrupt source at the interrupt controller.

A.1.5 osenv_irq_pending

`int` **osenv_irq_pending**(`int` *irq*);

Return the interrupt vector status. The result is only meaningful if the specified interrupt is currently disabled.

A.1.6 osenv_intr_disable

`void` **osenv_intr_disable**(`void`);

Disable interrupts. Prevent all interrupt sources from making it into the driver set. This routine corresponds to disabling processor interrupts.

A.1.7 osenv_intr_enable

`void` **osenv_intr_enable**(`void`);

Enable interrupts. Allow any pending or incoming interrupts, not individually disabled, to enter the driver set. This routine corresponds to enabling processor interrupts.

A.1.8 osenv_intr_enabled

`int` **osenv_intr_enabled**(`void`);

Return an indication of the interrupt-enabled status. Zero indicates that interrupts are currently disabled.

A.2 Sleep/Wakeup

Sleep and wakeup events are based on sleep records. The contents of these are system-dependent, but the `osenv` API requires two words of storage for each sleep record. Since there is no destroy method, every sleep record created should be slept on so that the operating system may garbage collect the sleep records.

A.2.1 osenv_sleep_init

`void` **osenv_sleep_init**(`osenv_sleeprec_t *`*sr*);

Create and initialize a sleep record. This routine is called by the device driver in preparation for blocking on an event.

A.2.2 osenv_sleep

`void` **osenv_sleep**(`osenv_sleeprec_t *`*sr*);

Block on a sleep record. If a wakeup has already occurred, return immediately. Spurious wakeup are allowed.

A.2.3 osenv_wakeup

`void` **osenv_wakeup**(`osenv_sleeprec_t *`*sr*);

Signal a sleep record. Wake up the thread blocked on the sleep record indicated.

A.3 Memory

This sections provides routines to allocate and free virtual and physical memory, and to convert between virtual and physical addresses. Several of the routines take an `osenv_memflags_t` parameter, which is an `unsigned int`. Valid memflags are: `OSENV_AUTO_SIZE`, `OSENV_PHYS_WIRED`, `OSENV_VIRT_EQ_PHYS`, `OSENV_PHYS_CONTIG`, `OSENV_NONBLOCKING`, and `OSENV_ISADMA_MEM`.

A.3.1 osenv_mem_alloc

`void *`**osenv_mem_alloc**(`oskit_size_t` *size*, `osenv_memflags_t` *flags*, `unsigned` *align*);

Allocate memory with the specified flags and alignment. This routine should not return `NULL` unless `OSENV_NONBLOCKING` is set and no memory of the requested type is currently available.

A.3.2 osenv_mem_free

`void` **osenv_mem_free**(`void *`*block*, `osenv_memflags_t` *flags*, `oskit_size_t` *size*);

Release a previously-allocated block of memory. The `flags` parameter must be the same as it was during the allocation, and `size` must be valid unless the `OSENV_AUTO_SIZE` flag is set.

A.3.3 osenv_mem_map_phys

`int` **osenv_mem_map_phys**(`oskit_addr_t` *pa*, `oskit_size_t` *size*, `void **`*addr*, `int` *flags*);

Map the specified physical memory and returns the virtual address where it is located. Return non-zero to indicate failure. The driver may request that the memory be treated as write-through or uncacheable by the processor.

A.3.4 osenv_mem_get_phys

`oskit_addr_t` **osenv_mem_get_phys**(`oskit_addr_t` *va*);

Return the physical address for a valid virtual address. The returned address is only guaranteed to be valid for memory requested with `OSENV_PHYS_WIRED` specified, although the operating system may honor requests for other virtual addresses.

A.3.5 osenv_mem_get_virt

`oskit_addr_t` **osenv_mem_get_virt**(`oskit_addr_t` *pa*);

Return the virtual address corresponding to the given physical address.

A.3.6 osenv_mem_phys_max

`oskit_addr_t` **osenv_mem_phys_max**(`void`);

Return the amount of physical memory. This routine may be used by device drivers to determine whether bounce-buffers are required for devices that cannot address the entire address range of the processor.

A.4 I/O Space

Routines to allocate and free I/O space. The drivers may directly access I/O space that they have allocated. On systems with memory-mapped I/O, `osenv_mem_map_phys` is used to gain access to the device instead of these routines.

A.4.1 osenv_io_avail

`oskit_bool_t` **osenv_io_avail**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

Check I/O space usage. Return `TRUE` if the specified I/O range is not in use and may be allocated by the device driver.

A.4.2 osenv_io_alloc

`oskit_error_t` **osenv_io_alloc**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

Allocate the I/O-space range. If any part of it is already in use, or permission is denied, an error is returned.

A.4.3 osenv_io_free

`void` **osenv_io_free**(`oskit_addr_t` *port*, `oskit_size_t` *size*);

Free the I/O range specified, marking it as available for use. A driver may only free I/O space it has allocated.

A.5 Clock

Functions to read and write the realtime clock

```
struct oskit_timespec;
```

A.5.1 oskit_rtc_get

`oskit_error_t` **oskit_rtc_get**(`struct oskit_timespec` **time*);

Read the real-time clock. This routine returns the current time.

A.5.2 oskit_rtc_set

`void` **oskit_rtc_set**(`struct oskit_timespec` **time*);

Set the current time in the RTC. The operating system is not required to honor these requests from the device driver.

A.6 Timers

Device drivers use timers to receive periodic notification, to implement timeouts, or to delay for a period of time. The `osenv` API provides low-level timer functionality, upon which the drivers may implement the desired behavior. For example, one-shot timers are not provided, as they are expected to be implemented on top of the periodic timer support provided by `osenv`.

A.6.1 osenv_timer_init

`void` **osenv_timer_init**(`void`);

Initialize the `osenv` timer support routines. This routine must be called before the device driver attempts to use any of the other timer routines.

A.6.2 osenv_timer_spin

`void` **osenv_timer_spin**(`long` *nanosec*);

Spin without blocking. Wait for the specified amount of time to elapse, without blocking or reenabling interrupts. This should only be used infrequently for small periods of time.

A.6.3 osenv_timer_register

`void` **osenv_timer_register**(`void` *(*func)(void)*, `int` *freq*);

Register a specified function to be called at `freq` hertz.

A.6.4 osenv_timer_unregister

`void` **osenv_timer_unregister**(`void` *(*func)(void)*, `int` *freq*);

Remove the timer callback function from the call-back list. The frequency is specified in case the same function has been registered more than once.

A.7 Bus-specific Interfaces

The `osenv` API contains additional bus-specific interfaces to support features present on various expansion busses. The `osenv` API currently contains bus-specific interfaces for the PCI and ISA expansion busses.

A.7.1 PCI Bus

The PCI bus-specific routines provide an interface to read and write PCI configuration space.

A.7.1.1 osenv_pci_config_init

`int` **osenv_pci_config_init**(`void`);

Initialize the PCI support code. Return non-zero if there is no PCI bus present or another problem occurs.

A.7.1.2 osenv_pci_config_read

`int` **osenv_pci_config_read**(`char` *bus*, `char` *device*, `char` *function*, `char` *port*, `unsigned *`*data*);

Read a word from PCI configuration space.

A.7.1.3 osenv_pci_config_write

`int` **osenv_pci_config_write**(`char` *bus*, `char` *device*, `char` *function*, `char` *port*, `unsigned` *data*);

Write a word to PCI configuration space.

A.7.2 ISA Bus

The ISA bus-specific routines are used to allocate and free ISA DMA channels.

A.7.2.1 osenv_isadma_alloc

`oskit_error_t` **osenv_isadma_alloc**(`int` *channel*);

Allocate the specified ISA DMA channel. The driver may program the ISA DMA controller once it has allocated a DMA channel.

A.7.2.2 osenv_isadma_free

`void` **osenv_isadma_free**(`int` *channel*);

Release the previously allocated ISA DMA channel specified.

A.8 Logging

`Osenv` components may generate informative messages. The OS may redirect these messages to wherever is appropriate. The default priority is `OSENV_LOG_INFO`.

The log priorities, in descending order, are: `OSENV_LOG_EMERG`, `OSENV_LOG_ALERT`, `OSENV_LOG_CRIT`, `OSENV_LOG_ERR`, `OSENV_LOG_WARNING`, `OSENV_LOG_NOTICE`, `OSENV_LOG_INFO`, and `OSENV_LOG_DEBUG`. `OSENV_LOG_DEBUG` messages are not normally output by default.

A.8.1 osenv_vlog

`void` **osenv_vlog**(`int` *priority*, `const char *`*fmt*, `void *`*vl*);

Log driver output. Sends a message from the driver, usually to the console.

A.8.2 osenv_log

`void` **osenv_log**(`int` *priority*, `const char *`*fmt*, *...*);

Log driver output. Calls `osenv_vlog`.

A.9 Panic

A.9.1 osenv_vpanic

`void` **osenv_vpanic**(`const char *`*fmt*, `void *`*vl*);

The driver calls this function if there is an error in the driver that prevents further execution. The driver set that called this function should be terminated immediately.

A.9.2 osenv_panic

`void` **osenv_panic**(`const char *`*fmt*, *...*);

Calls `osenv_vpanic`.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 1986.

[2] Godmar V. Back and Wilson C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, March 1999. IEEE Computer Society.

[3] Michael Beck, Harold Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, second edition, 1998.

[4] B.N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E.G. Sirer. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 62–65, Orcas Island, WA, May 1995. IEEE Computer Society.

[5] Andrew D. Birrell. An introduction to programming with threads. Technical Report SRC-35, DEC Systems Research Center, January 1989.

[6] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *22nd IEEE Conference on Local Computer Networks*, pages 179–188, 1997.

[7] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[8] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, Monterey, CA, November 1994. USENIX Association.

[9] Ajaya Chitturi. Implementing mandatory network security in a policy-flexible system. Master's thesis, University of Utah, 1998.

[10] Digital. *VAX Hardware Handbook*, 1982.

[11] Richard P. Draves. *Control Transfer in Operating System Kernels*. PhD thesis, Carnegie Mellon University, May 1994.

[12] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network

subsystem architecture for server systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, October 1996. USENIX Association.

[13] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville, NC, December 1993.

[14] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, June 1997.

[15] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995.

[16] University of Utah Flux Research Group. The OSKit. http://www.cs.utah.-edu/flux/oskit/.

[17] Bryan Ford. Portable Fluke microkernel prototype design and implementation. September 1996. Unpublished report.

[18] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[19] Bryan Ford and Mike Hibler. Fluke: Flexible μ-kernel environment application programming interface reference. http://www.cs.utah.edu/flux/fluke/.

[20] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, February 1999. USENIX Association.

[21] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, October 1996. USENIX Association.

[22] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable components for OS implementation. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, Cape Cod, MA, May 1997. IEEE Computer Society.

[23] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages

91–105, Seattle, WA, October 1996. USENIX Association.

[24] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, pages 163–176, Monterey, CA, November 1991.

[25] FreeBSD, Inc. FreeBSD home page. http://www.freebsd.org/.

[26] Boll O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilley and Associates, Inc., 1995.

[27] David Golub, Guy Sotomayor Jr., and Freeman Rawson III. An architecture for device drivers executing as user-level tasks. In *Proceedings of the Third USENIX Mach Symposium*, pages 153–171, Sante Fe, NM, April 1993.

[28] John R. Graham. *Solaris 2.X : Internals and Architecture*. McGraw-Hill, 1995.

[29] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[30] Hewlett-Packard Company. *HP Laboratories Disk I/O Traces*. http://-www.hpl.hp.com/research/itc/csl/ssp/traces/ajw.sample2.txt.Z.

[31] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.

[32] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[33] Galen C. Hunt. Creating user-mode device drivers with a proxy. In *Proceedings of the USENIX Windows NT Workshop*, pages 55–59, Seattle, WA, August 1997.

[34] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1996. Std 1003.1, 1996 Edition.

[35] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, St. Malo, France, October 1997.

[36] KRF*Tech*. KRF*Tech* device driver development toolkits. http://www.-krftech.com/.

[37] Greg Lehey. *The Complete FreeBSD*. Walnut Creek CDROM Books, 1996.

[38] Jay Lepreau. Personal communication, February 1999.

[39] Keith Loepere et al. Mk++ kernel executive summary. Technical report, Open Software Foundation, November 1995.

[40] James D. Lyle. *SBus Information, Applications, and Experience*. Springer-Verlag, 1992.

[41] Roland McGrath. MOM, the Mini Object Model: Specification (Draft). July 1998. Unpublished report. Available at http://www.cs.utah.edu/flux/docs/-mom.ps.gz.

[42] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[43] Hans-Peter Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, third edition, 1997.

[44] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, 1995.

[45] Jeffery C. Mogul and K.K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of the Winter 1996 USENIX Conference*, pages 99–112, San Diego, CA, January 1996.

[46] Jeffery C. Mogul and K.K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[47] NetBSD Foundation, Inc. NetBSD home page. http://www.netbsd.org/.

[48] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[49] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 15–28, New Orleans, LA, February 1999. USENIX Association.

[50] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, 1995.

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

[52] QNX Software Systems, Ltd. *QNX Operating System System Architecture*, 1996.

[53] D. Stuart Ritchie and Gerald Neufeld. User level IPC and device management in the Raven kernel. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 111–125, San Diego, CA, September 1993.

[54] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc., 1998.

[55] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*, pages 405–420, San Diego, CA, January 1993. HP Laboratories Technical Report HPL-92-152, December 1992.

[56] The Santa Cruz Operation, Inc. *Uniform Driver Interface (UDI) Specification*, 1998. http://www.sco.com/UDI/.

[57] Olin Shivers, James W. Clark, and Roland McGrath. Automatic heap transactions and fine-grain interrupts. In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP)*, Paris, France, September 1999.

[58] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, Washington, DC, August 1999.

[59] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, CA, October 1994. ACM SIGPLAN.

[60] Patrick A. Tullmann, Jeff Turner, John D. McCorquodale, Jay Lepreau, Ajay Chitturi, and Godmar Back. Formal methods: A practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, Cape Cod, Massachusetts, May 1997. IEEE Computer Society.

[61] Uresh Vahalia. *UNIX Internals: The New Frontier*. Prentice Hall, 1996.

[62] Peter G. Viscarola and W. Anthony Mason. *Windows NT Device Driver Development*. Macmillan Technical Publishing, 1999.

[63] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain, CO, December 1995.

[64] X/Open Company Ltd., Berkshire, UK. *CAE Specification: System Interfaces and Headers, Issue 4, Version 2*, September 1994.