THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

I/O Kit Drivers for L4

Geoffrey Lee

Submitted as a requirement for the degree
Bachelor of Engineering (Computer Engineering)

Submitted: November 3, 2005

Supervisor: Professor Gernot Heiser

Accessor: Dr. Sergio Ruocco

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Device drivers low-level software code that interact with computer hardware. Essentially, they function as software glue that binds the hardware and the rest of the operating system (OS) together.

The I/O Kit is an object-oriented device driver framework written to run on top of the Mach microkernel. It is the core driver framework for the Darwin system, a UNIX-like operating system which serves as a base for Mac OS X, a production-quality, general-purpose operating system that runs on Apple's line of personal computers. Drivers written for the I/O Kit are I/O Kit drivers, not Mach drivers, in the sense that in general they do not directly depend on functionality and data structures provided by Mach. The I/O Kit is feature-rich and provides a general framework that is common across device drivers.

The Mach microkernel is becoming a performance bottleneck inside the Darwin system, of which the I/O Kit is an integral component. Microkernel research has gone a long way since Mach was originally designed, and the Darwin system could stand to gain considerable performance improvements if it were ported to L4, a modern, state-of-the-art microkernel. On-going work is being made to port the BSD portion of Darwin to L4, but device drivers are still required in order to have a usable system. As part of an overall effort to have a working version of the Darwin system running on L4, I/O Kit and I/O Kit drivers must also be ported.

The purpose of this thesis is to investigate the possibility of running I/O-Kit-based drivers on top of an L4-based operating system. It aims to show that it is indeed feasible to migrate the I/O Kit away from a Mach-based environment to an L4-based environment with reasonable effort, and to migrate I/O Kit drivers away from a Mach-based environment to an L4-based environment with little or no effort other than a rebuild of the driver software. If successful, it could pave the way for the widespread adoption of an L4-based Mac OS X system with superior performance. A secondary aim would be identifying the places were the I/O Kit is dependent on Mach. These could be removed in favor of a generic interface so that any operating system which implements this interface can use the I/O Kit as its driver framework, along with I/O Kit drivers.

1.2 Outline

This thesis is comprises of this following chapters.

Chapter 2 This chapter gives a background into the various software components that this project involves, as well as the motivation and justification to port the I/O Kit and I/O Kit drivers from a Mach-based environment to an L4-based environment.

Chapter 3 This chapter builds on the introductory description on the I/O Kit. It describes the I/O Kit and I/O Kit drivers in detail.

Chapter 4 This chapter describes some related work and previous research that are of interest and revelant to this project.

Chapter 5 This chapter gives an overview of the porting process. Porting consideration such as the choice of the hardware architecture that the prototype is going to run on, and the exact L4-based operating system to use are discussed.

Chapter 6 This chapter describes the work that is done to port the I/O Kit framework and core related components from a Mach-based environment to an L4-based environment.

Chapter 7 This chapter describes the porting process of various device drivers from a Mach-based environment to an L4-based environment.

Chapter 8 This chapter presents describes and presents performance benchmarks that were done using drivers from an L4-based I/O Kit. The numbers obtained were compared to those obtained from a native Linux implementation and a native Mach-based implementation.

Chapter 9 This discusses some of the future work that should be done in light of the work achieved in this project. Both engineering work and further research opportunities are presented and discussed.

Chapter 10 This chapter concludes this report.

Chapter 2

Background

2.1 Introduction

This chapter aims to provide the requisite background knowledge behind the key components involved in this project.

This chapter will introduce the Darwin operating system and the L4 microkernel. A brief overview of the I/O Kit will be given, followed by the motivation to migrate away from the Mach microkernel to an L4-based system. A more complete description of the I/O Kit and I/O Kit device drivers will be given in the following chapter.

2.2 Darwin

2.2.1 Overview

The Darwin operating system [Appd] is a UNIX-like operating system. It is descended from the NeXTstep operating system [Lev]. With several exceptions, all components of the Darwin system are open source software, and can be downloaded and redistributed under an open source license [Appa]. At the time of writing of this report, Darwin 8.3 is the latest publically released version of Darwin.

Figure 2.1 shows the general layout of the Darwin system.

The Darwin system can be mainly divided into two separate portions: the kernel environment and the user environment.

2.2.2 User environment

The user environment contains standard UNIX programs and libraries which run with user-level privileges. These include UNIX command-line programs such as `ls` and `find`, and `libSystem`, which provides the standard C library and math library.

2.2.3 Kernel environment

The kernel environment is the privileged part of the Darwin system. This privileged component in the Darwin system is called XNU. The XNU component is actually a collection of several components, namely, the Mach microkernel, the BSD subsystem, the I/O Kit, libkern, and the Platform Expert. It is mostly the kernel component that this project is most concerned about. Hence, it makes sense to describe each of these in more detail. Figure 2.2 shows an approximate layered block diagram of the different components in the kernel environment. In practice, the boundaries are not so distinct.

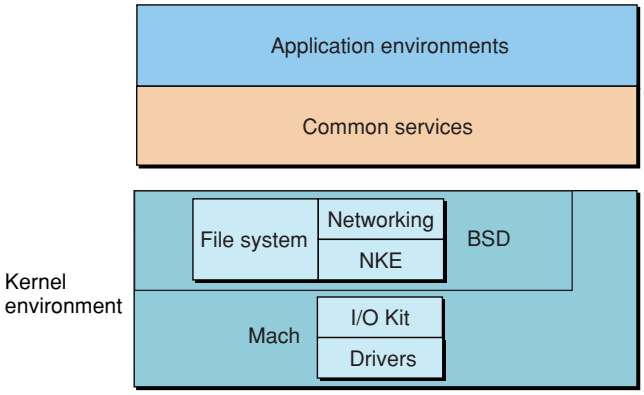

Figure 2.1: Darwin architecture. Reproduced from [Appd]

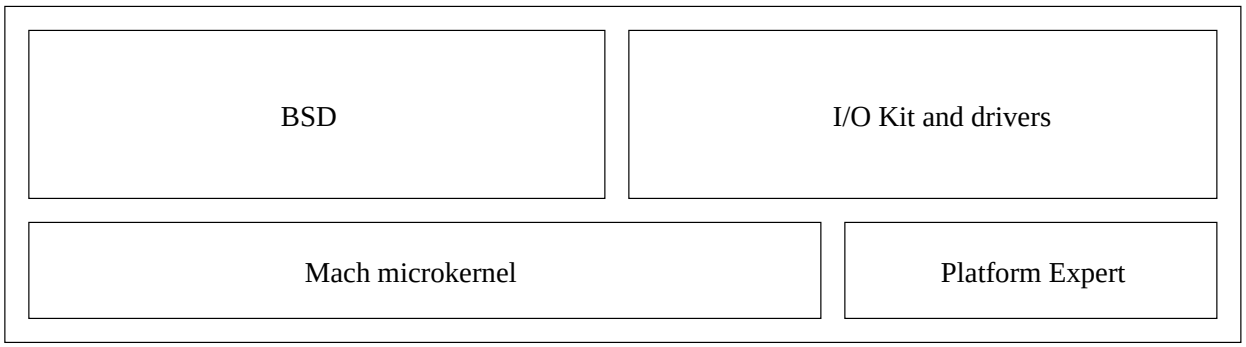

Figure 2.2: XNU environment block diagram.

Mach

The Mach microkernel [RJO+89] is a first generation microkernel initially developed at Carnegie Mellon University, and later at the University of Utah. The Mach microkernel has its roots in the BSD UNIX kernel [Lev].

The Mach microkernel provides threads, address spaces (called tasks), inter-process communication (IPC), and a virtual memory (VM) subsystem. Despite its apparent simplicity at a glance, it is actually a fairly complex piece of software. Apple's version of Mach, as found in Darwin version 8.2, contains over 180000 lines of source code, with 25657 lines of code and 53865 being architecture-specific code for the IA-32 and PowerPC architectures respectively.[1]

Platform Expert

The Platform Expert is responsible for implementing platform-specific routines. It exports its functionality via a portable C API in a platform-independent fashion. For example, routines used to reset a machine would be implemented inside the Platform Expert.

BSD

The BSD component implements the kernel portion of the BSD UNIX environment. It implements functionalities such as filesystems and networking. The BSD component is built on top of Mach, and makes use of certain features of Mach to implement the BSD UNIX subsystem. For example, it makes use of the VM subsystem and Mach tasks to implement UNIX processes.

I/O Kit

The I/O Kit is an object-oriented driver framework for the Darwin system. Though not explicitly mentioned, it appears that the I/O Kit's design was strongly influenced by, if not based on the driver framework found in NeXTstep, called Driver Kit. Extensive information on the Driver Kit can be found in the NeXT Developer Documentation [NeX]. Certain I/O Kit glue exist in driver stacks in order export a UNIX-like interface to the BSD layer. For example, the IOMediaBSDClient class exists to implement BSD-like block and byte I/O interfaces for storage devices.

Libkern

The `libkern` component implements the C++ runtime environment for the I/O Kit and I/O Kit device drivers. It does not contain all the features from a standard user-level C++ environment, since some features are deemed unsuitable for use in a multi-threaded kernel environment.

2.3 Mac OS X

The Mac OS X operating system builds on top of Darwin's infrastructure. The additions that Mac OS X provides are, unlike Darwin, mostly proprietary. Essentially, the Mac OS X operating system is a number of extra user-level applications added on top of the open-source Darwin system. Mac OS X is sold as a commercial software product, and is the de-facto operating system for Apple's line of personal computers and laptops. Figure 2.3 shows the architecture of Mac OS X.

At the time of writing, Mac OS X is available on Apple's PowerPC-based computers. However, Apple is currently making a transition from the PowerPC architecture to the IA-32 architecture in

[1] This was generated using David A. Wheeler's SLOCCount program [Whe].

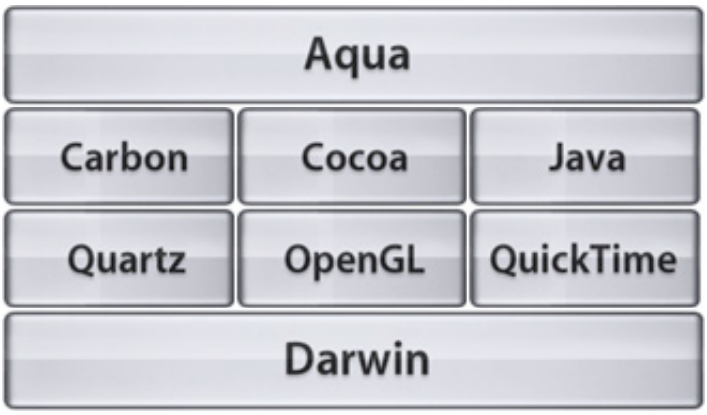

Figure 2.3: Mac OS X architecture. Reproduced from [Appe]

its line of personal computers and portables. A pre-release version of Mac OS X exists for Apple's IA-32-based transition machine platform, but it is available only to developers.

2.4 L4

L4 [Lie95b, HHL+97] is the generic name for a family of microkernels that implements a standard API, the latest of which is L4.X2. There currently exists an implementation of the L4.X2 API, known as L4Ka::Pistachio. This kernel is available for a variety of hardware architectures, including Alpha, AMD64, ARM, IA-32, IA-64, MIPS64, PowerPC (in both 32-bit or 64-bit mode) and and SPARCv9 (not yet released). It is mostly written in the C++ programming language, with certain frequently accessed code paths and architecture specific code implemented in assembly.

L4Ka::Pistachio is a high-performance, minimalistic microkernel. It is approximately 10000 lines of source code.[2] It provides address spaces and threads message-passing mechanisms. Communication between different threads is done either via shared memory, or via message-passing. The specification for it is described in the L4 X.2 Reference Manual [L4K01].

One of the goals of L4 is to separate *mechanism* and *policy*. Policy governs how a system is intended to behave, as opposed to mechanism, which provides the means to achieve an action inside the system. With policy removed from the microkernel itself, an operating system writer is free to implement whatever policy that is required for an *operating system personality*, that is, the policy that governs how the system is to behave, utilizing the mechanisms provided by the microkernel. It allows for much flexibility when implementing an OS personality on top of it.

More recently, National ICT Australia and the L4Ka team have announced the release of the L4-embedded API [Hei]. It is based on the X.2 API and aims to provide better support for embedded systems. The API is small, clean, and mostly compatible to X.2. There exists a conformant kernel, NICTA::Pistachio, which is a slightly modified version of L4Ka::Pistachio. While not available as a separate download at the time of writing, there already exists a version which very closely resembles NICTA::Pistachio which is bundled with and used by the Iguana system [NICb].

For the purposes of this report, L4 shall be synonymous with L4Ka::Pistachio, except for instances where L4 is regarded as the kernel for the Iguana system, in which case it means NICTA::Pistachio.

[2]The full source code is is around a factor of 10 bigger, however, a lot of it is architecture-dependent source code, and hence are never used when compiling a specific kernel for a specific platform built on a specific architecture.

2.5 Motivation

There is much to gain from having I/O-Kit-based drivers on an L4-based system. The following sections discuss some of the benefits.

2.5.1 Performance

The performance problems associated with Mach are well-known, with high IPC costs being one of them, as demonstrated by the Lites project [Hel94], and by Chen and Rashid [CB93]. This is an especially damaging flaw for a microkernel: IPC plays an important part in passing information between software components in a microkernel. In order to mitigate this, the BSD UNIX personality lives in the privileged address space of Mach itself. Thus, making use of Mach functionality within the BSD personality is normally just a simple function call, and it does not suffer from the high IPC costs of Mach that it otherwise would have in a traditional operating system personality implementation.

Even with the performance tweaks such as the one described above, it is still problematic. Wong [Won03] found that Mach IPC is used for communication between user processes in Darwin. Furthermore, it appears that the Aqua windowing system and related libraries found in Mac OS X, which is built on top of Darwin technologies, make extensive use of IPC.

Bearing in mind the high overhead of Mach IPC, using Mach IPC extensively is problematic, since will incur a significant overhead. There is significant interest to port Darwin to a more recent and high performance microkernel, such as L4. There has already been some limited success in porting the BSD subsystem over to L4, but it is not enough. In order for Darwin to be useful as a general purpose desktop or server operating system, various hardware devices such as disk controllers and network cards must be supported. Hence, bringing the I/O Kit and its device drivers to L4 is crucial to any eventual adoption of an L4-based Darwin system.

2.5.2 Robustness

Device drivers are known to be particularly prone to programming errors. According to private communication documented by Swift et al. [SABL04], misbehaving drivers cause 85% of Microsoft Windows XP crashes, while Linux drivers have seven times the bug rate compared to other kernel code [CYC+01].

In the current Darwin system, the BSD subsystem, the I/O Kit and all of its drivers reside in the privileged address space of Mach for performance reasons. This creates a very important trade-off in terms of robustness. Since the BSD subsystem and all of the device drivers live in the same protection domain as the Mach kernel, a bug in these components will likely cause a crash of the whole system.

In L4, almost all device drivers reside in user space, outside of the kernel's address space[3]. Thus, a misbehaving driver can at worst cause itself to crash, and not crash system as a whole[4].

If the I/O Kit were to be ported to L4, it would be possible for the I/O Kit software and its drivers to run completely in user space, that is, in an unprivileged and separate address space to the kernel and the rest of the operating system in general.

[3] There are efforts to remove the remaining drivers from NICTA::Pistachio.

[4] Strictly speaking, this is not entirely true, because a driver doing a faulty DMA operation may still cause the system as a whole to misbehave. Various methods have been proposed to solve this problem, such as using an input/output memory management unit (IO-MMU) on systems that have such hardware to further limit a faulty driver's impact. However, discussion on this is outside the scope of the current discussion.

2.5.3 Ease of porting

In general, I/O-Kit-based drivers do not directly depend on any functionality offered by the underlying kernel.[5] Instead, drivers depend on the the I/O Kit framework provide wrapper functions to shield operating-system-specific details away from the driver. An important implication of this is that once the I/O Kit component is extracted and ported, any drivers that were built for the I/O Kit will immediately and automatically be extracted out along with the I/O Kit too. This is noteworthy, because in general, a device driver built on top of a monolithic kernel will make use of OS-specific functionality directly.

There remains the questions of porting the I/O Kit itself. The I/O Kit is built directly on top of Mach, and makes use of Mach data structures and function calls. In the initial stages, it will need some sort of emulation layer to reproduce the required functionality that would normally be provided by Mach.

Fortunately, in general, there are clear and well-defined interfaces for accessing functionality from Mach. As well, in previous work, writing an emulation layer for Linux drivers, for instance, means that in effect, all Linux drivers must be ported. In I/O Kit's case, only the I/O Kit itself needs to be ported. The porting of drivers happen immediately and automatically on completion of the I/O Kit port.

2.6 Justification

Currently Darwin is powered by the Mach microkernel. Core system components such as the BSD UNIX personality and device drivers reside in Mach's protection domain, creating a relatively inflexible system with none of the benefits that a microkernel is supposed to offer, in addition to the inherent performance penalty of having the Mach microkernel there. Initial research into the feasibility of a port of the BSD subsystem to the L4 microkernel has shown some positive results, but a port of the I/O Kit to the L4 microkernel will also be crucial step to an eventual adoption of the L4-Darwin system. In addition, device drivers source code in operating systems have been shown to account for an alarming percentage of operating systems code. In systems where drivers run in the same privileged protection domain as the kernel for performance reasons, like Darwin, whenever a device driver crashes, it is likely to cause a crash of the whole system, requiring a restarting of the whole operating system. In using a high-performance, minimalistic microkernel such as L4, it has the potential to provide isolated device drivers running in unprivileged mode, with minimal overhead. Finally, while portability is an accepted concept in user-level software, to this day it remains poor in device drivers. Because of the fact that the I/O Kit forces device driver programmers to write in an almost operating-system-independent fashion, it has the potential to pave the way for true driver portability across different operating systems running on a wide range of different hardware.

[5] A notable exeception to this is network drivers. Network drivers use BSD *mbufs* for buffer managements rather than I/O-Kit-abstracted buffers.

Chapter 3

The I/O Kit

This chapter aims to introduce the I/O Kit in more detail, building on top of the very brief overview of I/O Kit given in the previous chapter.

3.1 An Overview of the I/O Kit

The I/O Kit [App04] is an objected-oriented framework that aims to simplify device driver development, which appears to have been influenced, if not based on the Driver Kit found in the NeXTStep operating system.

The I/O Kit is implemented in a restricted subset of C++. It disallows certain features of C++, including:

- exceptions
- multiple inheritance
- templates
- runtime type information (RTTI)

These features were dropped because they were deemed unsuitable for use within a multithreaded kernel environment. It should be noted that while it is illegal to use the standard C++ RTTI system, the `libkern` library, which is a part of the XNU package, has its own RTTI system.

3.1.1 Driver layering

The I/O Kit adopts a modular, layered approach that captures the relationships between components in the system. They can be viewed as a chain of provider-client relationships. Figure 3.1 and Table 3.1 gives one example of this relationship.

3.1.2 Families and drivers

An I/O Kit family is a level of software abstractions that is common across a range of device device drivers of a particular type. It is implemented in I/O Kit as one or more C++ classes.

A driver becomes a member of a particular family through inheritance. It is in this way that a driver gains access to the data structures and the routines that are common to the family. For example, a disk controller that implements the Small Computer Systems Interface (SCSI) may require the SCSI

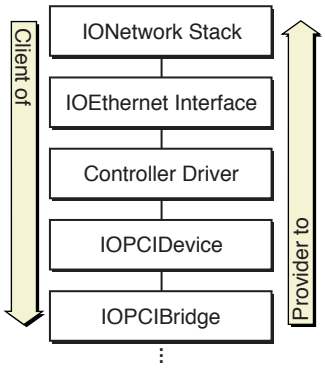

Figure 3.1: Driver objects as clients and providers. Reproduced from [App04]

IONetworkStack (interface managing object)	Connects I/O Kit objects to the BSD networking facilities.
IOEthernetInterface (nub)	Manages device-independent data transmission and reception.
Controller Driver (driver)	Operates the Ethernet controller through the IOPCIDevice object. This object inherits from a networking family class called IOEthernetController.
IOPCIdevice (nub)	Match point for the controller; provides basic PCI bus interaction in the controller.
IOPCIBridge (driver)	Manages the PCI bus. (Other objects provide services to the IOPCIBridge; the specific identities depend on the hardawre configuration.)

Table 3.1: Description of the overall layout in Figure 3.1. Reproduced from [App04]

bus to be scanned. This functionality would normally be implemented by the SCSI parallel family, unless the particular disk controller requires the bus to be scanned in some driver-specific way.

A device driver typically works with two device families. One is the family that the driver is a member of, such as the SCSI parallel family. The other is is a *nub* that is published by the family that the device is attached to.

3.1.3 Drivers and nubs

In the I/O Kit, there are two types of driver objects. One is a specific driver which drives a particular piece of hardware device. The other is a nub. A nub is an I/O Kit object that represents a communcation channel for a device or logical service and medates access to the device and service. For example, a nub could represent a disk, a disk partition, or a keyboard. Nubs are bridges between two drivers. A driver communicates with a nub as its client. A driver may publish a nub which finds a driver for which it is a provider. Nubs are also important for providing arbitration services, power management, and driver matching. In other words, nubs can be thought of as connectors between two different drivers.

As a driver detects a new device, a nub is created and it proceeds to attempt to match the appropriate driver. As an example, consider the case of a SCSI controller. The Peripherial Component Interconnect (PCI) bus driver detects the presence of a new device on the bus. It then publishes a nub for the new device. The nub identifies a suitable device driver for it. In this example, the appropriate SCSI controller driver is loaded into memory. The controller driver then proceeds to scan the SCSI bus for devices. Upon finding a device, it publishes as nub for the device. For example, if a SCSI disk is attached, the nub created for the device will go through the driver matching procedure, and will load the disk driver for it.

Figure 3.1 and Table 3.1 presents a more graphical explanation of how nubs and drivers are laid out in device layering.

3.1.4 The I/O Registry and the I/O Catalog

The *I/O Registry* is critical for providing the dynamic driver features of the I/O Kit. For example, it allows for a Firewire device to be plugged in to the computer, and be available immediately for use.

The I/O Registry tracks, dynamically, the client and provider relationships between devices continuously in time. Hence, if an external device were to be plugged into the system, for example, the I/O Registry would be updated to reflect this new system configuration.

The *I/O Catalog* is another dynamic database that works closely with the I/O Registry. Whenever a new device is discovered, the I/O Registry will request for a list of available matching drivers from the I/O Catalog.

The I/O Registry maintains a list of active objects running in the system, and the I/O Catalog maintains a list of available drivers installed on the system.

3.1.5 Work loops

A device driver must be able to act appropriately in response to *events*. This needs to be carefully done, since a driver's code could be run at any time in response to an event, whether it is due to a hardware interrupt, timeout events, or client I/O requests. This high level of concurrency requires proper protection of its data structures from concurrent access, as it may very well lead to data corruption.

A *work loop* is basically a gating mechanism that ensures single-threaded access to the data structures used by the hardware. It is in essence an exclusive (mutex) lock associated with a thread. It can be used as a gating mechanism that synchronizes the actions among different events. It is also used for

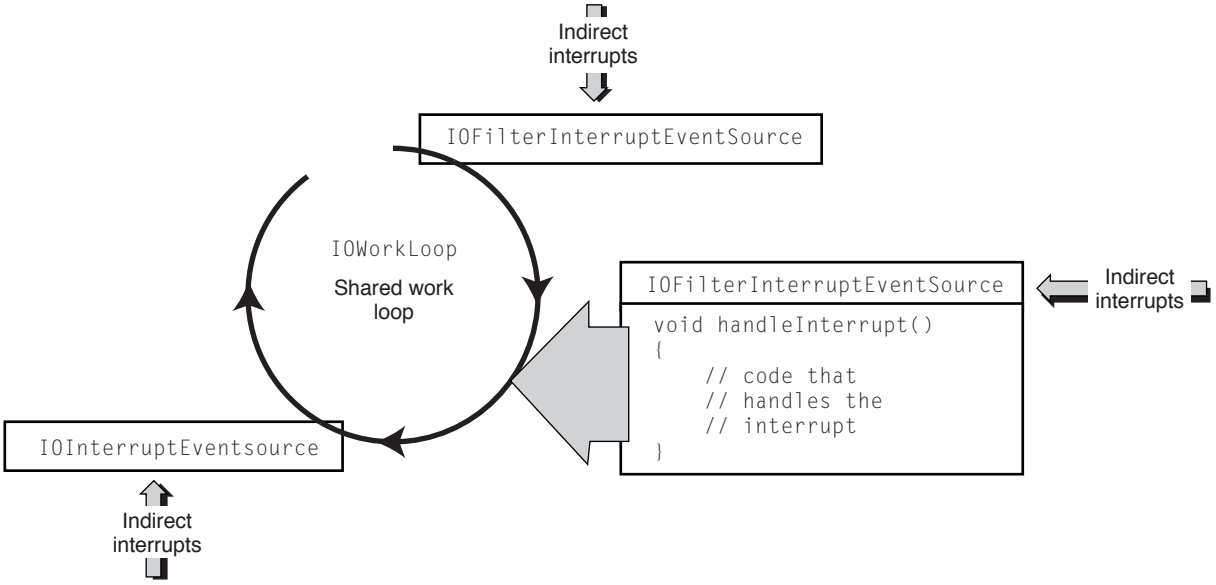

Figure 3.2: A work loop and its event sources. Reproduced from [App04]

interrupt handling to handle interrupts delivered by the interrupt controller. This mechanism serializes the interrupt handling, preventing multiple access to driver data by multiple interrupts. A work loop does not prevent concurrency: a driver can have multiple work loops, though extra care must be taken to ensure that no race conditions can occur. Figure 3.1.5 shows a work loop being utilized for interrupt handling.

An *event source* is an object that corresponds to a particular event that a driver is expected to handle. In order for a driver to handle a particular event, the event source must be registered with the corresponding work loop.

3.1.6 Direct memory access

Direct memory access (DMA) is a method that is present in certain bus controllers that involves directly transferring data between a device attached to a bus to system memory and vice-versa. Using DMA can provide significant performance gains, since it frees the processor from doing the data transfer.

Depending on the direction of the transfer, the source and destination can either be a DMA engine (or a specific device), or, it can be client memory represented by an IOMemoryDescriptor.

In the I/O Kit, each bus has its own specific DMA engine, each with its own properties. In particular, each DMA engine can have its own alignment requirement, endian format and size restrictions.

In addition to this, memory coming from the system may be partially or fully backed by the *Unified Buffer Cache* (UBC). The UBC is an optimization that combines the file-system cache and the virtual memory (VM) cache. The underlying structure of the UBC is the *Universal Page List* (UPL). The I/O Kit documentation terms an I/O request which are backed by the UPL a *conforming request*. In the case that it is not, it is called a *non-conforming request*. A UPL memory segment has certain characteristics, including:

- is at least page sized

- is page aligned
- has a maximum segment size of 128 kilobytes
- is already mapped into the kernel's address space

The I/O Kit uses the UPL because it is efficient for I/O transfers. It has the additional benefit of being easy to batch I/O requests. A UPL-backed IOMemoryDescriptor inherits the characteristics of a UPL segment as listed above. It is not explained anywhere why these rules are applied to conforming requests, though it appears to be a policy decision made by the designers of the I/O Kit.

The general policy regarding DMA controller drivers is that it must be prepared to handle a request of any alignment size, or other restrictions. If the request is a conforming request, it should never allocate memory to process the request as it could lead to a deadlock. If it is a non-conforming request, then the driver should try its best to process the request, including allocating resources if required, and fail if it cannot execute the I/O request.

3.1.7 Asynchronous events

In the life of a device driver, it can be mostly expected to handle mostly well-defined events. Well-defined events are those that a device driver is expected to handle. In general, these could include data transfer, or notification of a condition by the device via an interrupt. There are mainly two exceptions to this rule: power events and hot-plugging. They are unexpected events, because power state could be altered at any time that is out of the driver's control. Similarly, for hot-plugging, devices could be plugged or unplugged at any time, but is out of the device driver's control.

3.1.8 Power management

Darwin may run on hardware which may be power-constrained at times. An example of this would be a laptop while it is running on battery power. The I/O Kit contains facilities for power management.

The basic entity in power management is a device. For the purposes of power management, a device is defined to be a piece of hardware whose power consumption can be measured and controlled independently of system power. For hardware where that is not the case, they can be indirectly controlled through *p*ower domains. Figure 3.3 illustrates the concept of power domains.

There are at least two power states associated with a device, off and on. When a device is off, it has no power and does not operate. When a device is on, it has full power and it operates with full capabilities. A device may also have other intermediate states associated with it. For example, a device in a reduced power state may be able to function with reduced power. It can also be an intermediate state where it is unable to function, but maintains some state or configuration.

A *power domain* is a switchable source of power in the system. It provides power for devices that are considered members of the power domain.

Power domains are hierarchical. A power domain may contain another power domain. The power domain which contains all power domains is called the root power domain, which represents the main source of power of the system.

Power domains have power controllers and policy makers associated with them. The policy maker decides when to change the power state of a device, while the power controller executes the power change. Figure 3.4 illustates concept of power controllers and policy makers.

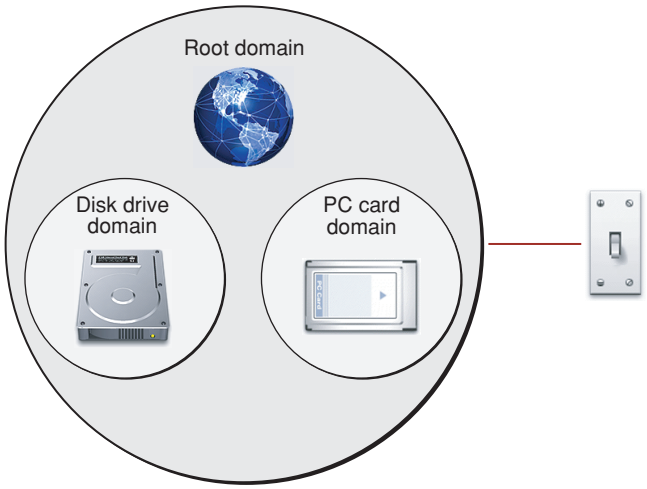

Figure 3.3: Power domains and devices. Reproduced from [App04]

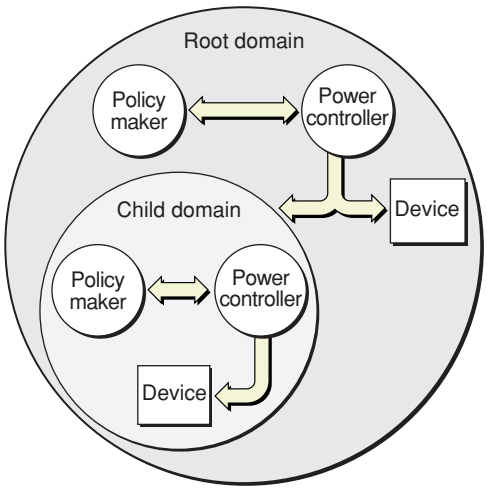

Figure 3.4: Power controllers and policy makers. Reproduced from [App04]

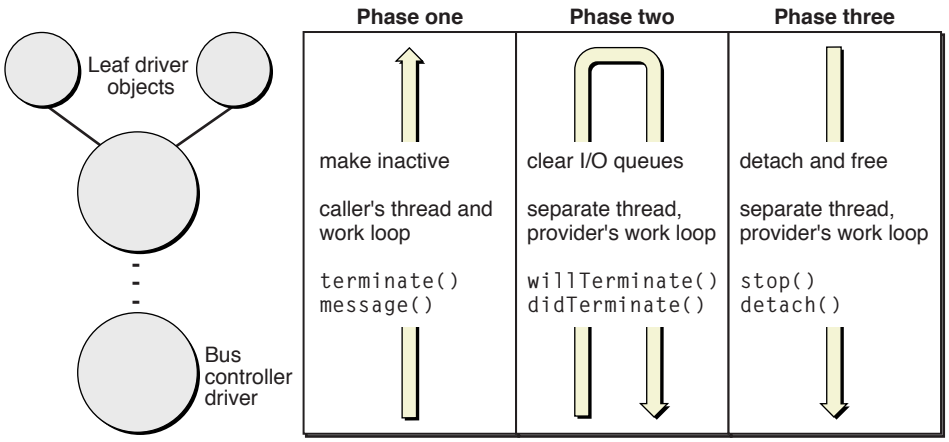

Figure 3.5: Phases of device removal. Reproduced from [App04]

3.1.9 Hot-plugging

Hot-plugging, also known as *hot-swapping*, refers to the ability of the operating system to initialize or safely detach a peripherial component, such as a Universial Serial Bus (USB) device, in reponse to a user inserting or removing the peripherial component while the operating system is actively running. Much like power change events, these are defined as unexpected, asynchronous events, since certain devices such as USB memory flash sticks could be plugged and unplugged at any time.

The I/O Kit contains facilities for hot-plugging. When a user plugs a in device, the driver matching mechanism kicks in and a suitable driver is found for it via the normal driver matching process.

When a device is removed however, the situation is different. The driver stack must be torn down rather than built up. In order to ensure that this is done in an orderly fashion, it requires a special API and procedure.

The I/O Kit performs this procedure in three phases. Figure 3.5 shows the three phases of driver shutdown. The first phase makes the driver objects in the stack inactive so they do not receive new I/O requests. The second one clears out pending and in-progress I/O requests from the driver queues. Finally, the I/O Kit invokes the appropriate driver life-cycle methods so that allocated resources are cleaned up.

3.2 An Overview of a Typical I/O Kit Driver

In previous sections, certain aspects of drivers that are related to the functioning of the I/O Kit were touched upon. However, no details have been said about what exactly an I/O-Kit-based driver is like. The following sections aims to retify this, by describing the anatomy of a typical I/O Kit driver in detail.

An I/O-Kit-based driver can be described succintly as a fancy kernel extension. In order words, there is nothing special about an I/O-Kit-based driver. It is essentially a kernel extension that depends on the I/O Kit, typically with operating-system-specific details hidden by the I/O Kit.

In brief, an I/O Kit driver is actually made up of two components, the code component and the property list component. In general, both are required in order for the correct functioning of an I/O Kit driver.

```
init()
attach()
probe()
detach()
free() /* if probe fails */
```

Figure 3.6: I/O Kit driver matching process. Reproduced from [App04]

3.3 The I/O Kit Driver Code

The code component of the driver is responsible for acting on events and servicing requests that are directly related to the device itself.

A basic device driver is quite simple. It essentially reduces to several member functions in the driver class, namely `init()`, `attach()`, `probe()`, `start()`, `stop()`, `detach()` and `free()`. During driver matching, the I/O Kit requests each driver in the pool of candidate drivers to probe the device to determine if they can drive it. The first function called is `init()`. The `init()` and `free()` member functions form a complimentary pair, as do `attach()` and `detach()`. `Init()` and `free()` are essentially `libkern`'s way of implementing constructor and destructor functions. The default action of `attach()` is to attach the driver to the nub through registration in the I/O Registry, while `detach()` detaches it from the nub. They may be overridden, but a driver rarely needs to do so.

After `init()` and `attach()` are called, `probe()` is called. This is always called if the driver's matching dictionary passively matches the nub in question. The `probe()` function returns a *probe score* which determines how well suited a device driver is for running the device. After all drivers have probed the device, the one with the highest probe score is attached and its `start()` function is called. The `start()` function initializes the device and prepares it for operation. The `stop()` function is used to stop the device.

3.4 The Driver Property List

Separate from the driver code, but still an integral part of the driver is the driver's property list. The property list is typically stored in a separate file serialized in the Extensible Markup Language (XML) format. This properties file is necessary, because it captures important information such as the matching parameters for the driver, the hardware that the device driver supports and the parameters should be used to drive the device, and other extensions that the driver in question depends on. For example, Figure 3.7 shows an exerpt of the XML property file from Apple's implementation of a PS/2 controller driver shows that it expects to be matched when the string "ps2controller" is seen.

The information stored in these XML files are generally not used directly. The `libkern` library contains functions that converts it from the XML representation into an internal representation before the data is accessed.

There exists a simpler format that is used for the bootstrapping of in-kernel modules. This format contains a subset of what would normally be found in its XML equivalent. It is stored as a string named *gIOKernelConfigTables*. Although this format is incompatible with the XML representation, both are converted into an internal representation before they are used by the I/O Kit. Hence, as far as the I/O Kit and drivers are concerned, they are presented with one uniform representation that can be accessed using C++ methods.

The separation of a driver into a code component and a data component is a little unusual, but it is good design. It cleanly separates data that describes the features of a particular piece of hardware

```
<key>IOKitPersonalities</key>
<dict>
        <key>ApplePS2Controller</key>
        <dict>
                <key>CFBundleIdentifier</key>
                <string>com.apple.driver.ApplePS2Controller</string>
                <key>IOClass</key>
                <string>ApplePS2Controller</string>
                <key>IONameMatch</key>
                <string>ps2controller</string>
                <key>IOProviderClass</key>
                <string>IOPlatformDevice</string>
        </dict>
</dict>
```

Figure 3.7: An exerpt of the properties file from the ApplePS2Controller driver. Reproduced from [Appc]

and the code that is used to drive the hardware, while preserving the ability to read and modify this information if necessary. It also allows for the ability to create a driver stack, because the matching information is stored in the property tables and the matching is done implicitly by the I/O Kit, not explicitly by the driver.

Chapter 4

Related Work

This section intends to cover related work on device driver frameworks and related software systems that have been ported to L4.

4.1 OSKit

OSKit [FBB+97] is a generic operating system framework for writing operating systems. It aims to allow operating system writers to concentrate on writing the operating system itself, and not have to worry about low-level systems code that is common across almost all operating systems. OSKit incorporates source code from various open source operating systems for some of its operating system components. Figure 4.1 illustates the architecture of OSKit.

In the context of device drivers, OSKit provides a simple generic device driver layer. The OSKit has both FreeBSD drivers and Linux drivers, but they can both be accessed using the generic driver glue layer present in the OSKit.

4.2 L4-Darwin

L4-Darwin [Won03] was the result of an attempt to port the Darwin system to a bare-bones L4 microkernel. The term L4-Darwin is a little misnamed, because strictly speaking, it does not involve a straight port of the Darwin system to L4. It actually involves replacing Mach, which is an integral part of the current Darwin architecture, with L4. Figure 4.2 shows a diagram of the design of the L4-Darwin prototype.

Initial efforts on replacing the Mach kernel with L4 has focused on only the BSD UNIX personality onto a 32-bit PowerPC-based machine. Wong [Won03] acknowledges that device drivers, and hence, the I/O Kit should be considered, but it was decided that it was outside the scope of the project at the time. Initial effects on the port as shown positive initial results. The system is able to boot and run a simple user program, but much needs to be done in order to make it into a usable system. In particular, it is obvious that the Mach emulation glue is a hack. For example, the almost non-existent virtual memory system in Wong's L4-Darwin prototype is particularly problematic.

Another attempt was made more recently to transform Darwin into L4-Darwin to a 64-bit PowerPC-based machine [Roo04]. However, the project was never completed.

There is currently a new effort to migrate the BSD portion away from Mach to L4, with the initial prototype running on IA-32-based machines. It is still a work in progress.

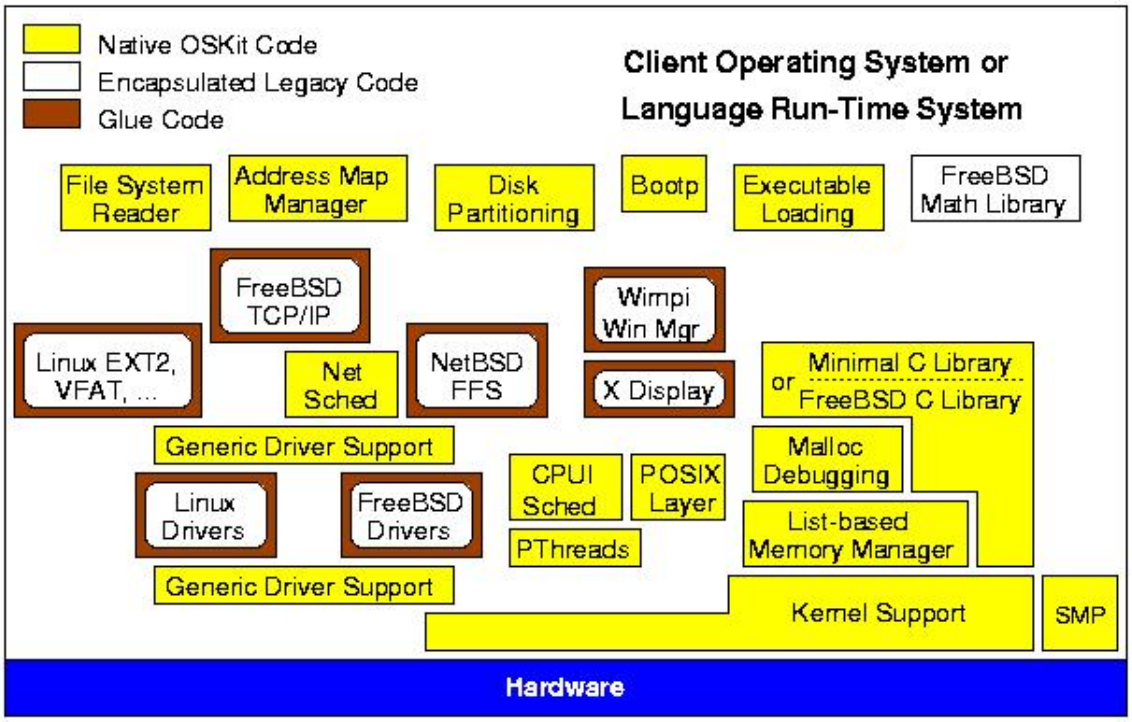

Figure 4.1: Architecture of OSKit. Reproduced from [Flu]

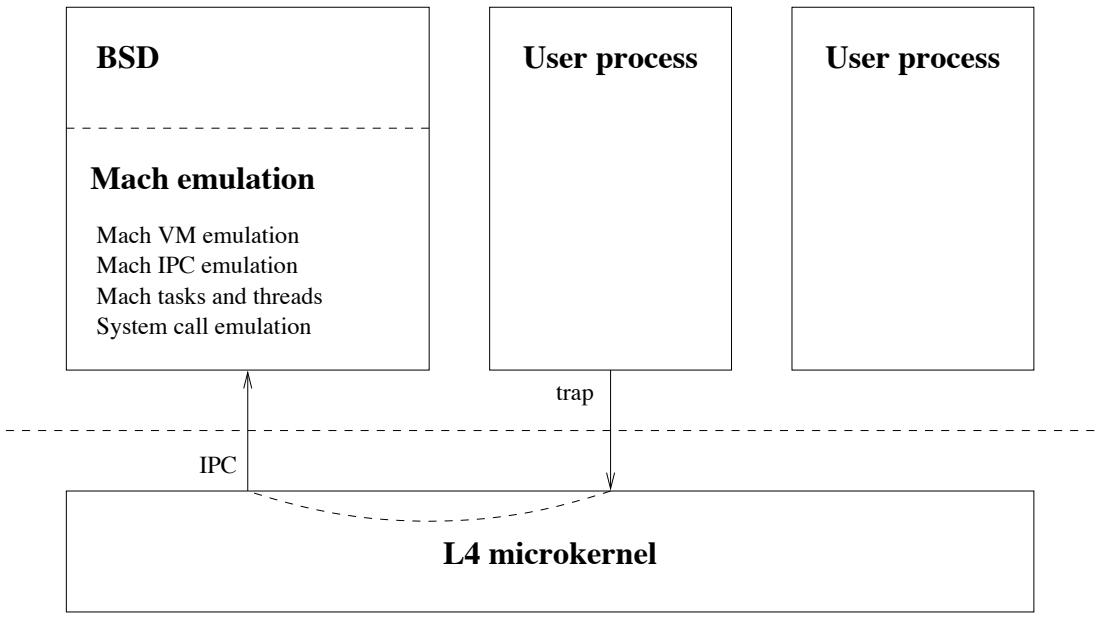

Figure 4.2: Early L4-Darwin prototype design. Reproduced from [Won03]

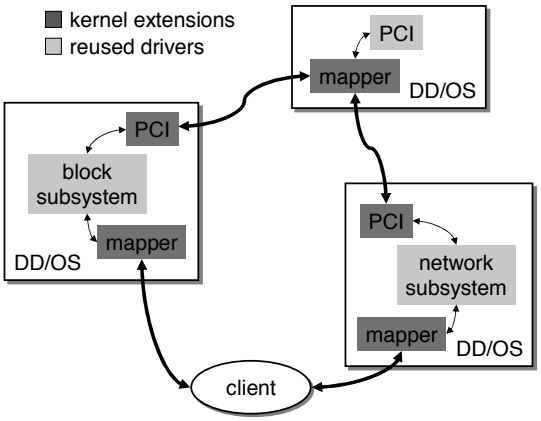

Figure 4.3: DD/OS architecture. Reproduced from [LUSG04]

4.3 Device Driver OS

The device driver operating system (DD/OS) as described by LeVasseur et al. [LUSG04] is aimed at achieving driver reuse and enhanced driver dependability via virtual machines with minimal overhead. Figure 4.3 shows a diagram which describes the DD/OS architecture.

Instead of porting a particular device driver from a donor OS to the recepient OS, the whole OS is ported to a hypervisor to run it its own protection domain. A client OS running on top of the hypervisor contacts the DD/OS via a well-defined interface.

To improve performance, the same DD/OS image can be run concurrently. This can reduce memory footprint. As well, as it is likely that the same housekeeping functions will be called in the same image, frequently run DD/OS code will likely already be in the processor cache. Performance can be improved even further by running multiple drivers in the same DD/OS protection domain, at the cost of dependability, because should a misbehaving driver cause a crash, it will cause a crash of the whole DD/OS that it was running in, including any drivers that may have been running in the protection domain of that DD/OS.

Benchmarks have shown that using L4 as a high performance hypervisor, good driver performance can be achieved with minimal resource and engineering overhead.

4.4 User-level Device Drivers

User-level device drivers are highly desirable, as they offer much flexibility and robustness that is simply not possible with in-kernel device drivers, which is the norm in many of today's general-purpose operating systems. Previous research into user-level device drivers have largely not made a lasting impact, due to the fact that they were often implemented on primitive microkernels with high overhead. As second high-performance microkernels have emerged, there is renewed interest in user-level device drivers.

Perhaps the most important in a real-world scenario is system stability. If a driver running at user-level causes a crash, it may be possible to restart it, instead of bringing the whole system down, as is normally the case with in-kernel drivers.

Another benefit of user-level drivers is language independence, as noted by Chubb [Chu04]. Tra-

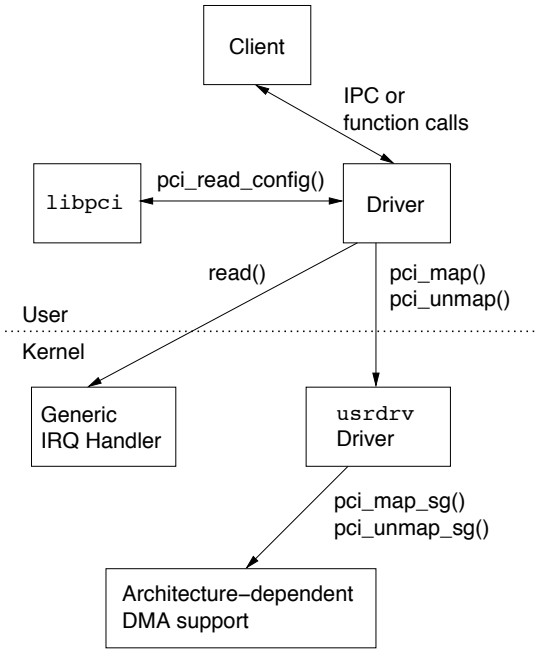

Figure 4.4: Block diagram of a Linux user-level device driver. Reproduced from [Chu04]

ditionally, in an in-kernel driver implementation, the driver is tied to the language that the operating system kernel was implemented in. In contrast, in user-level drivers, the driver may be implemented in another language. One example is FitzRoy-Dale's implementation of user-level drivers with the Python programming language on the Mungi operating system [FD03].

The idea of user-level device drivers is certainly not new. However, user-level device drivers remain the exception in mainstream systems. They are typically only used for peripherials where performance is less critical, or the work that must be performed by the driver is much larger than the overhead of an extra few context switches (the Linux X server is an example).

As fast system calls, good threading, and cheap context switches become available in modern kernels, the idea of user-level drivers is no longer limited to microkernels. It is now possible to write user-level drivers for a range of devices in traditional monolithic kernels as such Linux with negligible performance penalty, as noted by Chubb [Chu04]. Figure 4.4 shows one such implementation for a Linux-based system. It sho

There have been various attempts to design and evaluate user-level driver frameworks, either on top of a monolithic kernel such as Linux as described by Elphinstone and Götz [EG05] and Chubb [Chu04], or in an environment where the natural approach is to implement user-level device drivers as described by Leslie [Les02].

Performance-wise, these user-level driver frameworks have shown promising results, meaning that user-level device drivers are indeed feasible. Certain open problems still remain. In particular, these frameworks do not address the issue of code reuse. Porting an existing driver to run at user-level may require extensive emulation code to be written, and possibly the porting of other components which the target device driver depends on. Also, on monolithic systems, these frameworks do not address issues that are important in modern general-purpose device driver frameworks, such as power management, or hotplugging. Due to its monolithic design, it is likely to be quite tightly integrated with the rest of the kernel, and it is unclear what sort of engineering effort must be undertaken in order for user-level

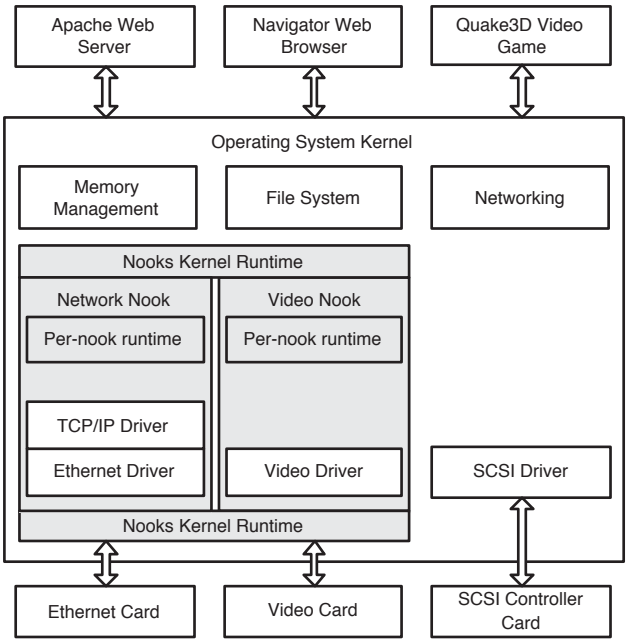

Figure 4.5: Nooks architecture. Reproduced from [SMLE02]

device drivers to become a general-purpose solution.

4.5 Nooks

The Nooks project [SMLE02] aims to improve operating system reliability based on the observation that device drivers are the most common cause of operating system failures, despite years of extensive research in extensible operating system technology. Figure 4.5 shows an architectual diagram of the Nooks system.

The Nooks reliability system aims to enhance OS reliability by isolating the OS from device driver failures. The Nooks project achieves this by isolating drivers inside *lightweight protection domains* inside the kernel's address space, so that hardware and software mechanisms can prevent drivers from corrupting the kernel. This is done by using hardware and software mechanisms to provide read-only access to certain portions of the kernel's address space. The fact that these protection domains are are globally visible in the kernel's address space has the advantage that data-sharing is easy: no marshalling is required. Swift acknowledges that this is not unlike the management of address space in single address space operating systems [SBL03]. More recently, the Nooks architecture was extended to include the notion of *shadow drivers* [SBL03]. A shadow driver is layer that stands in the middle of applications and the driver. This layer allows the operating system to conceal driver failures and be able to transparently restore a malfunctioning drivers into a functioning state.

The Nooks implementation suffers from poor performance. In particular, contrary to what Swift et al. claims [SABL04], the Nooks system does not achieve what is advertised with minimal overhead. Subsequent benchmarking done by Tsui [Tsu04] showed that in some cases, the Nooks kernel had twice the CPU utilization than the vanilla Linux kernel. Clearly, this sort of overhead is unacceptable for

everyday use. In addition to this, the whole Nooks system is very large. It contains approximately 23000 lines of code [SABL04]. The is not clear how this aims to solve the problem of reliability, as device drivers will then itself depend on the Nooks system, which, being a complex piece of software, would itself prone to errors. An application-level filesystem benchmark performed by Swift et al. [SBL03], specifically, the time it took to untar and compile the Linux kernel on a local VFAT filesystem, showed that with Nooks enabled, the compilation ran about 25native case. From the graph presented, having Nooks enabled increased the time spent in kernel mode by approximately a factor of 5. The authors note that this is due to both Nooks and the high number of translation lookaside buffer (TLB) misses associated with switching across domains. The authors speculated that Nooks could be optimized but does not make a convincing case of it. In particular, they note that one of the ways to reduce costs associated with the TLB is to selectively flush the TLB on protection domain switch, which actually is not possible with the architecture they benchmarked with (Pentium 4) because it has an untagged TLB.[1]

Nooks incurs performance penalties that are commonly associated with poorly-designed user-level drivers, without the benefits.

4.6 Mungi Device Model

The Mungi operating system [HERV93] is a distributed, single address space operating system written by the DiSy group at the University of New South Wales, Australia. It currently runs on top of the L4Ka::Pistachio kernel, but it has used previous versions of L4 as a base in the past. It is interesting to investigate its device model in detail, because, it allows for the isolation of different device drivers, and provides robustness by requiring device drivers to run at user-level.

Leslie's [Les02] proposed solution for a Mungi device driver framework is an interesting one, since it contains some similar properties to this project. In particular, device drivers are run at user-level with mechanisms to to run them in isolated environments. The former property is important because Leslie notes that extra support from the operating system is required in order to run device drivers at user-level.

4.6.1 The Mungi *PagePin* system call

In order to support DMA-capable devices at user-level on machines which uses virtual addressing, it is important for the OS to, given a protection domain and a virtual address, be able to translate it into a list of physical address. This is needed, because while the OS and user-level programs in general deal with virtual addresses, devices deal with physical addresses. Apart from this, it is important that the buffer which is used for a DMA transfer remains valid for the duration of the transfer.

The *PagePin* system call was devised to solve these problems in one step. Given a virtual address, it pins the page into memory, and returns the physical address that the virtual address in question is mapped to. The current implementation is only able to return a single physical address and hence is unable to translate virtual memory regions bigger than a page size in one iteration. The *PageUnpin* system call may be used to unpin pages after a DMA transfer is complete.

[1] Technically this is not true, since a segmented model could be used to achieve this. However, Swift et al. dismissed the use of such tweaks such as those proposed by Liedtke [Lie95a], because they claim in order to preserve compatibility, a segmented model could not be used.

4.7 WinDriver

Windriver and Windriver USB [Tec] are two commercial products that allows for development of user-mode PCI drivers and USB drivers respectively. WinDriver and WinDriver USB retain some of the advantages of pure user-level drivers, however, WinDriver contains provisions for allowing performance-critical portions to run in kernel mode[2]. The product page of WinDriver claim that WinDriver products can achieve performance comparable to a fully in-kernel implementation, but they do not substantiate their claims with actual benchmark results. Also, it is unclear whether such a driver is completely running in user-mode (not counting the extra operating support that is required to export necessary interfaces in order to write PCI drivers), or whether the in-kernel/user-level hybrid approach is used.

4.8 Mach User-level Drivers

Work had been done previously at Carnegie Mellon University on user-level device drivers for Mach 3.0. Interestingly, one of the motivations was performance due to excessive copying between the kernel and user space. This problem can ve solved by mapping the device directly into the application's address space, obviating the need to copy to and from buffers. Other reasons for user-level device drivers given were location transparency, preemptability, and easier code-sharing between related drivers. The authors reported thoughput improvements for Ethernet and SCSI devices, but no CPU information on CPU utilization was given (the CPU utilization might have a rough indication of the performance overhead induced by IPC).

4.9 Windows NT User-level Drivers

An implementation of user-level drivers for Windows NT was done by Hunt [Hun97]. The motivation was simplicity and ease of development. Under this model, the user-level drivers are supported by an in-kernel proxy that redirects I/O request packets (IRPs) from the proxy to the user-level driver. It suffers from poor performance, and does not support access to hardware.

4.10 Other User-level Drivers

The user-level drivers work listed above are by no means the only work done previously. Apart from the work listed above, there exist other work done previously, either using purely software approach, such as the Fluke user-level driver model [VM99], or using a combination of hardware and software support such as that proposed by Pratt [Pra97] or Schaelicke [Sch01].

[2]This feature is not supported for WinDriver USB.

Chapter 5

An Overview of the Design and Implementation

5.1 Introduction

This chapter gives an overview of what needs to be considered in migrating the I/O Kit and its drivers away from the Mach microkernel to L4.

5.2 Hardware

The Darwin operating system, and hence, the I/O Kit, currently supports two architectures, the PowerPC architecture running in both 32-bit mode or 64-bit mode, and the IA-32 architecture.

Apple is currently in a state of transition to migrate away from the PowerPC architecture to the IA-32 architecture. As one of the goals of this project is to demonstrate a Darwin on L4 prototype with device drivers, in the long run, as the IA-32 architecture becomes increasingly important for Apple, it is anticipated that having the initial prototype running on the IA-32 architecture will likely provide the largest impact.

5.3 L4 Considerations

There are mainly two ways to port the I/O Kit to an L4-based system. The first method involves a straightforward port to an existing L4-based system. The second involves a method of virtualization called *pre-virtualization*.

5.3.1 Evaluation of Iguana

Introduction

Iguana [NICb] is a base for the provision of operating system services on top of the L4 microkernel. It is primarily designed for use in embedded systems. Presently, Iguana is implemented on top of a slightly modified L4Ka::Pistachio microkernel. This kernel has an API that is mostly conformant with the NICTA::Pistachio specification [NICa]. Iguana is relatively portable, and has been ported to many hardware architectures including Alpha, MIPS64, IA-32, and ARM.

Iguana contains design decisions that are adically different from regular general purpose operating systems. In particular, it implements a *single address space* with protection for objects living inside

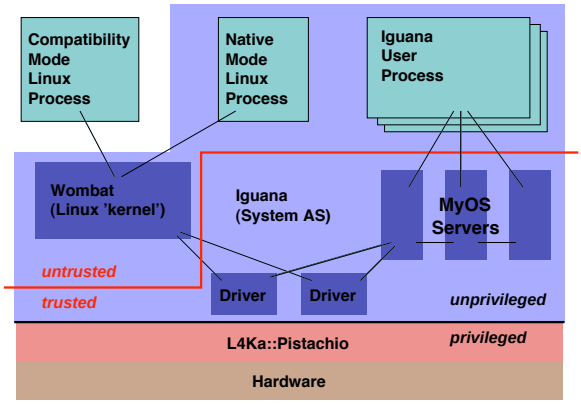

Figure 5.1: Iguana block diagram. Reproduced from [Hei04]

this address space enforced by *capabilities*.

Iguana contains a feature called *external address space* (EAS) that provides legacy support for components that require for *multiple address spaces* (MAS). One such example is Wombat, which is a port of the Linux kernel onto Iguana. It allows unmodified Linux applications to be run inside the Wombat environment.

Iguana driver model

The work of porting on porting I/O-Kit-based drivers to L4 will be closely tied in to the device driver support that Iguana has.

Drivers in Iguana are implemented on top of the `libdriver` library. A device driver registers itself with Iguana, the driver class that it belongs to, and a list of functions that should be called in order to perform an operation on the hardware device that the device driver controls.

Iguana currently does not have a proper device driver model. In particular, the functionality that `libdriver` provides is very simplistic, and is not even fully implemented. As such, it makes this driver model unsuitable for general-purpose use.

The time and effort that is required to implement a proper device driver model for Iguana and to write drivers for it is likely to be better spent on porting an existing, proven solution. Since Iguana does not currently have a proper device driver model, it will be a good base to showcase a port of the I/O Kit successfully running nearly unmodified I/O Kit drivers.

The case for using Iguana

Currently, there does not exist many operating systems or frameworks for operating systems which operate on top of the L4 microkernel. Hence, it was not a difficult decision to use Iguana. As a bonus, Iguana does contain a lot of libraries which may turn out to be potentially useful in the porting process. In addition to this, Iguana's already usable Linux environment may prove to be very useful, as it allows the use of conventional, well-established benchmarks that are written for UNIX-like operating systems.

5.3.2 Evaluation of L4-Darwin

The primary goal of this project is to provide device driver support for the L4-Darwin prototype mentioned in Section 4.2.

The Darwin software that was used as a base for the L4-Darwin prototype is now severely outdated. There is, however, currently an effort to bring this up to date with the current version of Darwin, but is unlikely to be ready within the timeframe of this project.

The port is incomplete at this moment. Much of the Mach functionality which L4-Darwin tries to emulate are either not implemented or have simply been replaced with simplistic code that does not work for the general case. There is likely to be a huge flux of changes as this port is brought up to date and its missing features implemented. L4-Darwin is likely to remain buggy and unstable for the near future.

For the above reasons, it appears to be undesirable, at least for the time being, to use L4-Darwin as a base for a port of the I/O Kit.

5.3.3 Evaluation of pre-virtualization

Pre-virtualization [LUC+] is a method of virtualization that involves virtualizing at compile-time and boot-time. Simply speaking, it involves replacing *sensitive instructions*, that is, machine instructions which depend on the privilege level of the processor, with an emulation version implemented in software.

Since the process of pre-virtualization is done automatically by the assembler and there already exists a set of compiler toolchain which is able to generate pre-virtualized code for the IA-32 architecture using L4 as the hypervisor, pre-virtualization should be the easiest and quickest path to show demonstrable results with running I/O-Kit-based drivers on top of L4.

The motivation and goals of running I/O-Kit-based drivers, and for that matter, L4-Darwin as a whole, can be implemented on top of a virtual machine, but that does not come close to a primary goal of this project. In particular, virtualization does not have the potential to give the performance gain that may be possible from migrating away from Mach, nor does it offer the possibility of separating out the the I/O Kit software so that it can be used as a generic device driver framework for different operating systems.

A pre-virtualized XNU kernel will likely be worthy of investigation, and performance benchmarks may be useful for comparison purposes. It is clear, however, that pre-virtualization would only satisfy a subset of this project's motivation and goals, and hence, should not be considered as a primary direction that this project should head towards.

5.3.4 Porting Iguana

Using Iguana as the base for an L4-based I/O Kit prototype appeared to be a suitable approach from preliminary evaluation.

At the time of the initial evaluation, Apple's PowerPC-based computers were the hardware of choice, because it was not until later that Apple announced its intention to switch to Intel-based IA-32 processors.

As Iguana did not support the PowerPC architecture at the time, work was undertaken to port Iguana to Apple's PowerPC-based computers.

Currently, Iguana is able to run on Apple's 64-bit PowerPC computers. The system is able to run to the point of being able to run the example programs that are included with the Iguana distribution.

Although it will not be used for the initial prototype, efforts undertaken to port Iguana to the PowerPC architecture will likely become useful in the future. Firstly, the PowerPC architecture is known to be used in the embedded world. It is likely that Iguana will benefit from support another

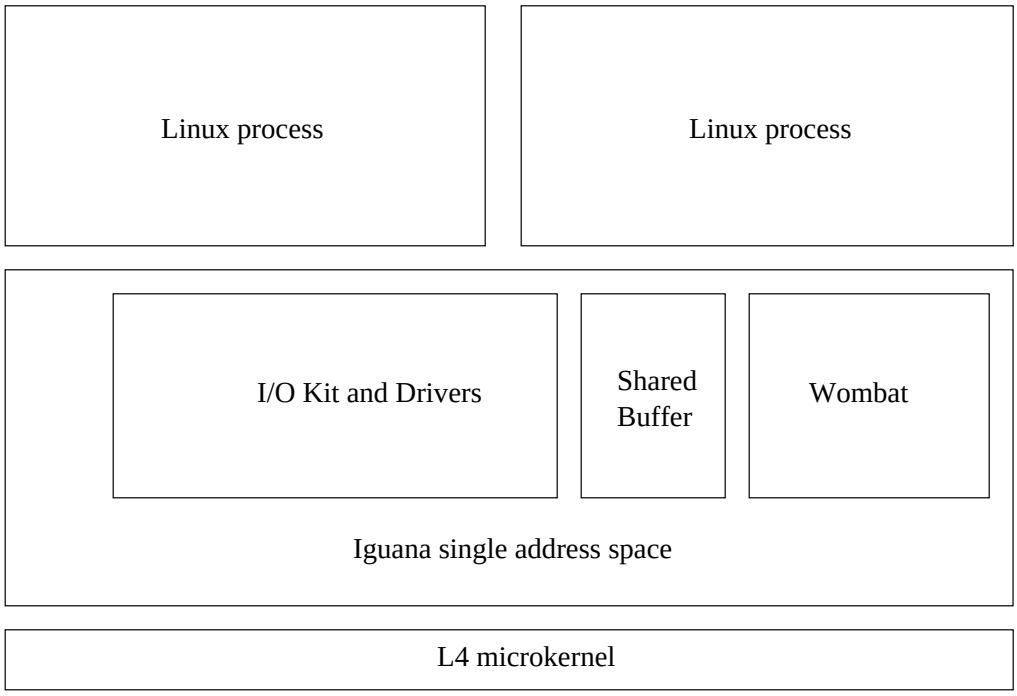

Figure 5.2: A block diagram of the prototype system.

processor that is known to be used in the embedded market. In addition to this, Apple is not planning on phasing out the use of PowerPC processors until the end of 2007, and it is likely that Mac OS X need to be supported on Apple's PowerPC-based computers for some time after that. Also, like what was done with the IA-32 architecture previously, the PowerPC may be used in the future as a reference architecture to ensure that the code base remains relatively portable. Hence, it will become important that there is a PowerPC-based version of Iguana in the near future.

5.4 I/O Kit and Driver Porting

As a first step, it is important to show a working I/O Kit framework along with selected drivers running on an L4-based system. This can be done porting the I/O Kit along with the selected drivers as application libraries, which applications can link directly. Figure 5.2 shows a block diagram of the prototype system.

The I/O Kit and its drivers reside in one protection domain, while other Iguana components, such as Wombat, reside in their own respective protection domains. The shared buffer is a shared memory region that allows data to be transferred between the I/O Kit protection domain and other Iguana components. As the only client of I/O Kit drivers for the prototype is Wombat, it is simple and convenient to have a shared, static memory region for data transfer.

The addresses in the shared region have a one-to-one correspondence between virtual and physical addresses because it greatly simplifies device driver I/O in the prototype. Iguana deals with virtual addresses, while devices deal with physical addresses. Iguana currently does not have an interface that allows the conversion of a block of virtual memory into a list of physical addresses that can be handed to a device. Having a one-to-one mapping between physical and virtual addresses effectively solves this problem, since the addresses do not need to be converted in the first place. The address of the buffer

can be handed directly to the device driver for I/O.

Although this preliminary design does not provide the security and robustness guarantees that this project is ultimately aiming for, in that all drivers and the I/O Kit framework still all reside within one single protection domain, doing a simple port is the quickest way to show something that is demonstrable. In addition to this, Iguana does not yet have an implementation of a software component framework. In light of this, work done in implementing protection boundaries between driver components will likely become obsolete in the future in favor of the generalized component framework that is to be offered by a future version of Iguana.

Chapter 6

Porting of the I/O Kit Framework to Iguana

6.1 Introduction

This chapter describes the process of porting the various components from the XNU kernel to an Iguana-based environment. In addition, it documents the design and implementation of the XNU emulation layer that was devised to provide the missing functionality required that would normally be provided by the XNU kernel.

6.2 Analysis of the I/O Kit

The I/O Kit depends on several features inside the XNU kernel. In order for the I/O Kit to run, those functionalities must either be disabled, if that can be done, otherwise, those functionalities must be present. If it uses a large portion of the code in a particular component of the XNU kernel, it should be ported along with the I/O Kit. Alternatively, if it only uses very specific parts, the effort to rewrite those parts will likely be much smaller than taking the effort to pull in the complete component from the XNU kernel.

The following sections aims to detail what is required and what was done to resolve the dependencies of I/O Kit.

6.2.1 XNU-psecific functions and data structures

The exported I/O Kit API appears to be relatively clean of references to Mach data structures and functionality. However, within I/O Kit itself, it does not make any attempt to hide the fact that it was written on top of Mach. Mach functions and data structures are used directly within I/O Kit.

The I/O Kit was compiled into a standalone image. This is done so that the required data structures can be determined at compile-time, and the required symbols that it references within XNU can be determined at link-time. A list of the symbols that the I/O Kit depends on can be found in Appendix A.

6.2.2 Loadable kernel modules

The I/O Kit contains a large amount of device drivers. It is not reasonable for them to be statically linked to the I/O Kit framework itself, because this makes the resulting executable file unnecessarily large. Also, missing loader functionality means that extra third-party extensions and drivers cannot be loaded on the fly during run-time.

The I/O Kit makes extensive use of *loadable kernel modules* (LKMs). LKMs are machine executable object files which can be dynamically loaded and unloaded by the operating system at run-time.

Dynamically loading device drivers on the fly from an object file is not the only way to initialize device drivers in the I/O Kit. The I/O Kit can also load *in-kernel modules*. In-kernel modules are those that are loaded and linked into the same executable image at compile-time. For all intents and purposes, they are probed, matched and initialized like an LKM driver in the I/O Kit.

LKM is an extremely useful feature. However, LKMs do not add anything new to the discussion regarding driver performance for an L4-based I/O Kit implementation. In addition, drivers could be still be dynamically probed and loaded by using in-kernel modules. Hence, it was determined that an LKM implementation was outside the scope of this project.

6.3 I/O Kit Port Details

6.3.1 Introduction

This section aims to go into the technical details of the actual Iguana-based I/O Kit prototype. It aims to document the modifications made to both Iguana and related I/O Kit dependencies in the prototype system.

6.3.2 General Issues

Some general issues were encountered in porting the I/O Kit and related components to an Iguana-based environment. In particular, the C library offered by Iguana was a little problematic.

The C library included with Iguana does not have support for C++. In particular, C++ explicitly requires that any C-compatible functions to be marked as such.

In addition, Iguana's C library does not provide thread-safe access to its functions. That is to say, in general, the safety of the C library's internal data structures could not be guaranteed if two or more threads were running in a protection domain at the same time. This is certainly the case for the I/O Kit. Hence, explicitly locking was required whenever making calls that are directly or indirectly dependent on C library functions which are not thread-safe. A future version of Iguana libraries is slated to provide thread-safe libraries, which will eliminate the need for this temporary ad-hoc solution.

6.3.3 Porting Process

It is useful to know how the I/O Kit and its dependent components are being ported to the I/O Kit. The I/O Kit, along with other components from the XNU kernel that are ported in whole are to be ported as Iguana application libraries, which Iguana applications can link against. For those that are emulated in the Mach emulation layer, they are included in the `xnu_glue` Iguana application library. In order for an Iguana application to make use of the I/O Kit framework, it must link in the I/O Kit library, along with the libraries which it depends on.

6.3.4 Libkern

The `libkern` library implements the C++ runtime system for the I/O Kit and I/O-Kit-based drivers.

The `libkern` component is relatively easily to port, as by its very nature, it was designed to be a standalone C++ runtime system, and thus does not depend on other operating-system-specific functionality.

The changes that were required to be made for the `libkern` library to run on top of Iguana are documented below.

ABI changes

The current version of the GNU C++ compiler produces code that is binary-incompatible with code produced with the 2.95 version. In order to provide backward compatibility, Apple ships with a C++ compiler that includes a flag that, when set, will cause the compiler to emit code that more closely resembles the 2.95 ABI. This is quite problematic, as `libkern` depends on this in order to function properly. A more detailed explanation of this may be found in the *build_ptrmemfunc1()* function in `gcc/cp/typeck.c` [Appb].[1]

For the purposes of showing a working I/O Kit demonstration, it is not necessary to provide backward compatibility. In addition, modifying the compiler to suit `libkern` was deemed too difficult to be worth the effort when `libkern`, which is a much simpler piece of software, can be modified instead. It is anticipated that if, in the future, this work is to be integrated into a future release of Darwin, then it would be possible to compile this work with Apple's modified C++ compiler, obviating the need for this modification.

Libkern initialization changes

As part of the C++ initialization process, the `libkern` initialization sequence parses the in-memory executable image to locate the C++ constructors and runs them. The code made assumptions concerning the format of the executable image, in particular, it expected the image to be in Mach-O format, which is not the case for the prototype. This was modified so that the constructors were run correctly in the prototype.

6.3.5 The xnu_glue library

`Xnu_glue` is the library which emulates much of the XNU-related functionality that is required for the I/O Kit and I/O-Kit-based drivers to function. Figure 6.1 shows a block diagram of the `xnu_glue` library. The I/O Kit framework, I/O Kit drivers, `libkern`, and `xnu_glue` are all linked together in a single application to run in a single protection domain. The `xnu_glue` library can be thought of as a combination functions which the I/O Kit and drivers can call directly, and a number of threads for those functionalities which requires a context of execution in order for them to be implemented. Currently, when the `xnu_glue` library is initialized, it starts three threads: *l4intr*, *xnu_sched*, and *threadcallserv*. They implement interrupt handling, synchronization, and thread callouts respectively.

The Platform Expert and BSD emulation code are perhaps not interesting, because they are just library functions that does data copying, or functions to initialize the necessary data structures which the I/O Kit expects to be in a well-defined state at when the I/O Kit is initialized. The Mach emulation however, should be explained in more detail, because they actually implement actual operating system functionality that would otherwise be provided by Mach. The following sections detail what exactly in Mach were required.

Virtual memory

The I/O Kit uses Mach primitives to allocate and deallocate kernel virtual memory. Depending on the type of allocation, they were either emulated using a generic implementation of `malloc()` and `free()` for small allocations, or, for larger allocations, they were allocated using Iguana *memsections*. For the purposes of Mach virtual memory emulation, memsections are blocks of memory that are page sized or multiples of that, and are aligned on page boundary. Each memory address valid within a memsection has the same protection. Memsections are also used for DMA operations, because it is

[1] There exists another ABI incompatibility, but it only pertains to the PowerPC-based platforms.

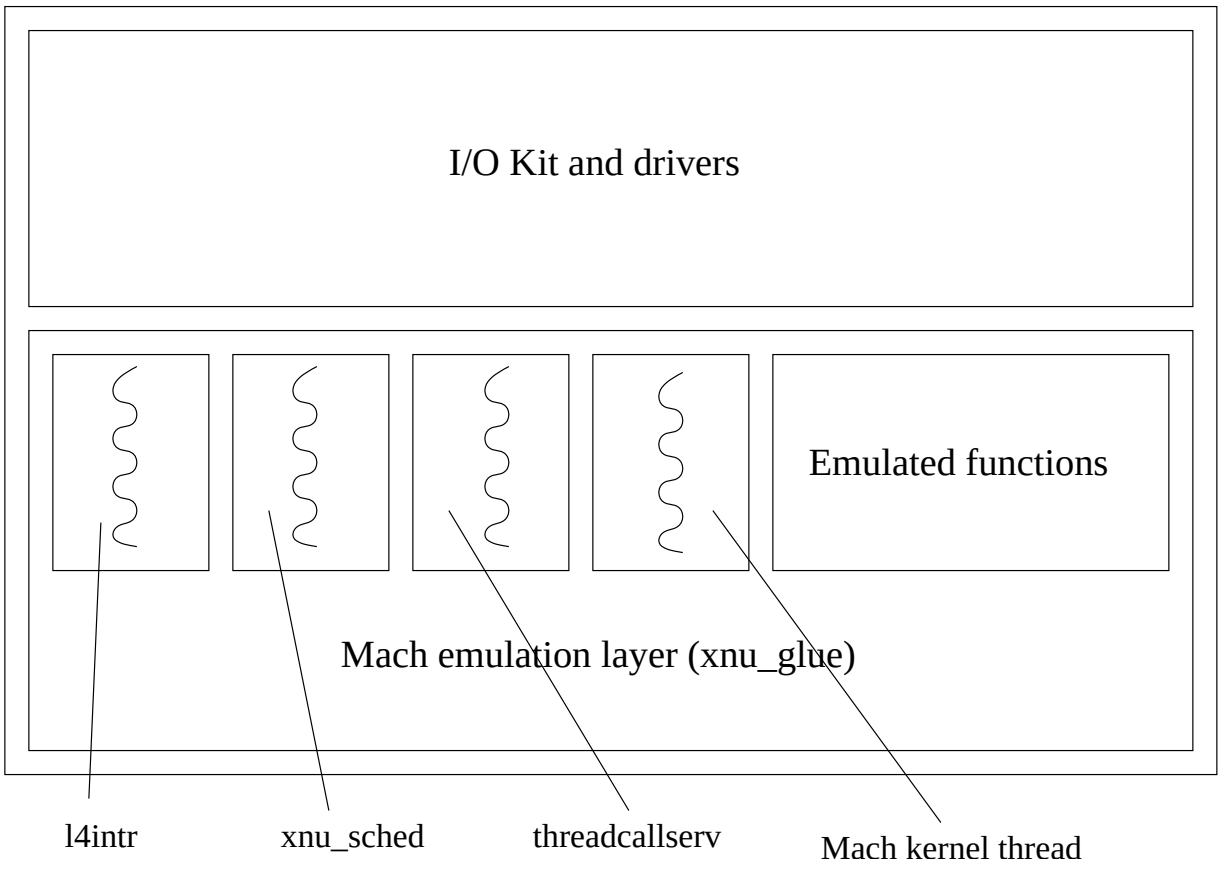

Figure 6.1: Xnu_glue block diagram.

possible to allocate memory that has a one-to-one mapping between physical and virtual addresses. This is required until Iguana implements functions similar to the *PagePin* and *PageUnpin* system calls in Mungi, which was discussed in Section 4.6.

Interrupt Handling

L4 models interrupts as threads. Each interrupt has a unique global thread identifier. In order to receive an interrupt, the handler must register the interrupt with the kernel. Afterwards, notification of interrupts are delivered to the registered thread as an IPC call. The thread must then process the interrupt and acknowledge the interrupt. The interrupt remains disabled until an acknowledgement reply message is sent back to the interrupt virtual thread. Interrupt registration is a privileged operation: it can only be done by Iguana. However, Iguana exports an interface which allows arbitary threads to register itself as an interrupt handler for a particular interrupt with L4.

In the original Darwin system, the I/O Kit will first register a function which is to be called whenever an interrupt occurs due to a device signaling a condition. The Mach kernel will then, on receiving an interrupt from a device, call this function which the I/O Kit has registered with Mach.

Interrupt handling is done by the l4intr thread in `xnu_glue`. When it initializes, it registers all available system interrupts with Iguana. It then waits in an infinite loop for interrupt notifications. Instead of registering the interrupt handler with the Mach kernel, however, the I/O Kit instead registers it with `xnu_glue`. The l4intr thread will then arrange for this interrupt handler to be run on receiving an interrupt notification from the kernel's virtual interrupt threads. This job is done by the platform-specific interrupt controller I/O Kit driver.

Locking and synchronization

Locking and synchronization primitives are required in a multi-threaded environment like the I/O Kit. The I/O Kit contains its own interface for locking. These are in turn implemented on top of the locking primitives that Mach provides. There are mainly three types of locking and synchronization primitives used by the I/O Kit and its drivers. They are spinlocks, mutexes, and semaphores. In terms of scheduling, spinlocks differ from the latter two in an important aspect: a thread that acquires a spinlock will never sleep due to the spinlock, while for the latter two, it is possible that the caller may be put to sleep until the mutex or the semaphore becomes available again. Spinlocks simply require an atomic test and set function be implemented, while for the latter two, more work as required, because if the mutex or semaphore in question is not available, the requesting thread may have to be put to sleep and another thread needs to be chosen to run. In other words, a simplistic user-level scheduler is required in order to implement the semantics associated with mutexes and semaphores.

The sleeping and waking up of the emulated Mach kernel threads is done by the xnu_sched thread. The xnu_sched thread runs in an continuous loop waiting for requests to put threads to sleep, or wake them up. Sleeping and waking up threads is implemented in the following fashion. Each mutex or semaphore operation that may potentially sleep is translated into an L4 IPC call operation. Such a call operation involves sending a request to the server, and then sleeping until a reply is received from the server. The call operation resembles typical remote procedure calls (RPC) where the client sends a message to the other party, and waits for a response from the other party in response to the initial message. This is useful, because it allows a thread to be implicitly blocked by not immediately replying to the call operation.

Mach kernel threads

Mach kernel threads are emulated using native Iguana threads, which are in turn implemented using native L4 threads. There exists functions that emulate the Mach equivalent, which internally calls the appropriate functions to spawn a new thread with the appropriate starting instruction address and the appropriate arguments.

Thread calls

Thread calls are functions which a client may register with the operating system, requesting that a certain function with certain user-supplied arguments be executed at some later point in time. The I/O Kit, and certain drivers, use this feature to implement delayed function calls.

Thread calls can be created, destroyed, registered and cancelled using Mach API function calls. The thread callout thread, threadcallserv, actually executes the callouts. The thread callout server is woken up at periodic intervals to determine whether there are any outstanding callouts that need to executed. If it is determined that there are, the thread callout server will execute the registered function with the specified arguments in the thread server's thread context. This process is iterated until there are no more outstanding thread callouts that need to be executed. The thread call server will then proceed to block until it is woken up again.

Evaluation of the xnu_glue implementation

The size XNU emulation library is nearly 20000 lines of code, however, the coe that was actually written is closer to 3000.[2] The size of the code required to reimplement the required XNU functionality appears to be quite reasonable, considering that a single implementation of the `xnu_glue` library is able to support the I/O Kit and its drivers.

6.3.6 I/O Kit Modifications

It was found that the I/O Kit could remain largely unmodified. The parts which did require modifications are documented below.

IOMemoryDescriptor modifications

The IOMemoryDescriptor class is essentially an interface to the Mach virtual memory system, catered for device drivers. The XNU kernel and its applications operate on virtual addresses, however, devices on the system expect to be presented with physical addresses. The role of the IOMemoryDescriptor is to efficiently convert, when presented with an address space and a virtual address, into a scatter gather list of physical addresses for device I/O.

I/O Catalog modifications

The I/O Kit, in particular, the I/O Kit's I/O Catalog, expects loadable kernel modules to be a supported feature in the underlying operating system. This is currently not true for the prototype system. Support for this feature was hence removed from the I/O Catalog.

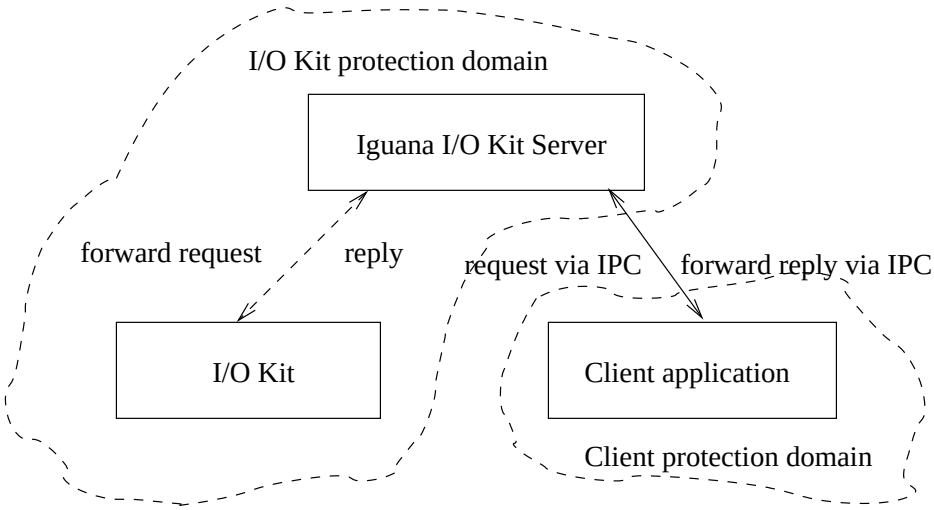

Figure 6.2: Iguana I/O Kit server operation.

6.3.7 The Iguana I/O Kit server

Since the I/O Kit and related components have been ported as application libraries, they cannot be used in a standalone manner. An application, which makes use of these libraries, is required to listen and reply to requests on the I/O Kit library's behalf. This is done by creating an extra OS server application in Iguana. This server listens for requests and passes them to the I/O Kit. After the request is processed, it passes the results back to the client application. Figure 6.2 is a block diagram that shows how these components interact with each other. This server runs in the same protection domain as the I/O Kit and its drivers, because the drivers and the I/O Kit are linked into this server application.

[2]Both of these numbers were generated with the help of David A. Wheeler's SLOCCount program [Whe].

Chapter 7

I/O Kit Drivers on Iguana

7.1 Introduction

This chapter aims to describe the steps involved in porting a typical device I/O Kit device dirver onto the Iguana-based I/O kit prototype driver framework. Following this, issues with drivers that were selected to be ported to run on an Iguana-based I/O Kit environment are discussed.

7.2 An Overview of the Porting Process

Given the Iguana-based I/O Kit framework as described in Chapter 6, the next logical step to take is to understand what must be done in order port some given I/O Kit driver to the Iguana-based I/O Kit framework. Similar to what was done with the Iguana-based I/O Kit and its dependencies, drivers are ported as Iguana application libraries. The I/O Kit server needs to explicitly link against these drivers to make use of them.

There are mainly two things that need to be done in porting an I/O Kit driver. These two are porting the property list file[1], and porting the driver source code.

7.2.1 The Property List File

As the current prototype has no mechanism to unserialize the XML data and the property list file residing on the filesystem to the internal representation used by the I/O Kit, it is necessary to convert the XML data into more rudimentary representation as described in Section 3.4 of Chapter 3. This step is required as this data is read into the I/O Registry, so that based on this information, it can run the driver matching process when required.

7.2.2 Source Code Porting

Due to the fact that that driver source code is typically mostly only directly dependent on the interface exported by the I/O Kit, porting a driver is mostly a relatively simple process. Usually, it involves a recompile in the prototype environment with very minor modifications due to limitations in the prototype.

[1]This is only required because support for loadable kernel modules is non-existent in the current prototype. Future versions which implement this feature should obviate the need to do this.

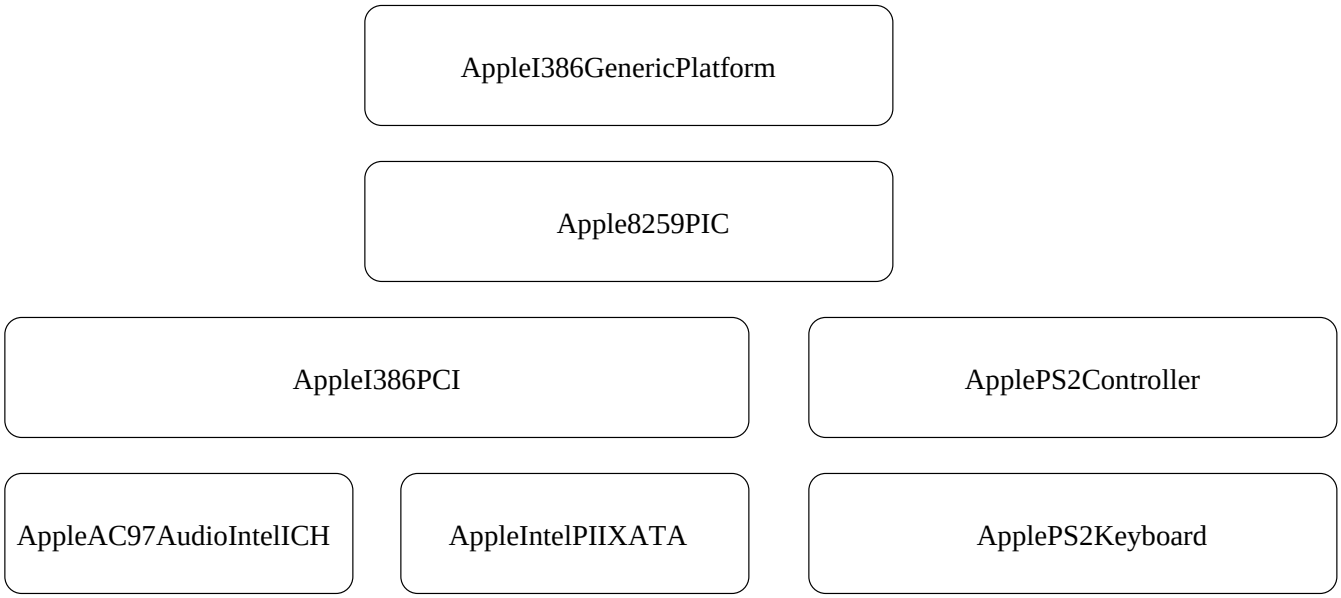

Figure 7.1: Selected I/O Kit driver hierarchy.

7.3 The I/O Kit Driver Hierarchy

Directly following the question of how a driver is ported, the next logical question to ask is what must be ported. After deciding on this, the next step is to undertand exactly what nubs and drivers the selected drivers depend on, so that they can be pulled into the prototype too.

Figure 7.1 shows the list of selected drivers and their location within the driver hierarchy. The specific drivers selected for peripherial devices are AppleAC97AudioIntelICH, AppleIntelPIIXATA, and ApplePS2Keyboard.

7.4 The Platform Expert Device Driver

Each platform must have a platform driver. The platform driver, in essense, is the top-level root driver of which all other drivers and nubs are children of (in the sense of a device layout hierarchy on a system, not in terms of the C++ language).

In the case of generic IA-32-based personal computers, are two types of drivers which could be used as the top-level platform driver. One is the AppleACPIPlatformExpert driver which utilizes the Advanced Configuration and Power Specification (ACPI) [CCC$^+$] found in newer personal computers, or it can use the AppleI386GenericPlatform driver which is intended to work on older personal computers too. The AppleACPIPlatformExpert driver is a rare example of a proprietary Darwin component. Given this, there was no choice other than to use the AppleI386GenericPlatform driver. A bonus reason for choosing to use the AppleI386GenericPlatform driver over the AppleACPIPlatformExpert driver is that the ACPI platform includes many features that are not required for the initial prototype. Even if the AppleACPIPlatformExpert driver was open source, using it will likely complicate porting efforts.

This driver is obviously required, since it is at the top level of the device driver tree. Porting this driver was a relatively simple and painless process, which simply involved a recompile of the driver in the Iguana-based I/O Kit prototype environment.

7.5 The Interrrupt Controller Device Driver

The interrupt controller is modeled in the I/O Kit as a separate driver. On generic personal computers, the Intel i8259 programmable interrupt controller (PIC) chip is responsible for processing interrupts.

The AppleAPIC driver is actually a combination of two different drivers: the AppleIntel8259PIC driver, and the AppleAPIC driver. Only one can be used at any one time. The AppleI386GenericPlatform only expects the former to be matched and used.

Strictly speaking, the low-level interrupt-handling should be done by the L4 kernel. This merely exists as a compatibility layer so that minimal modifications has to be done to the I/O Kit driver architecture.

7.6 The PS/2 Keyboard Device Driver

PS/2 is a common interface on personal computers that is used to connect keyboard and mouse devices. The PS/2 keyboard device is an ideal driver for the prototype because it is one of the simplist I/O Kit drivers. Hence, it is useful for verifying the basic functionalties of the I/O Kit.

The PS/2 keyboard driver for the I/O Kit runs the keyboard device in interrupt-driven mode. The keyboard is a very simple device. Ignoring the setup and initialization, essentially the keyboard only ever needs to handle 2 types of events: key-press and key-release. Both of these events are signaled to the driver through a device interrupt. When a key is pressed, the driver is signalled and the driver then proceeds to read in a *scancode*. The scancode indicates to the keyboard driver what key was pressed. When the key is released, another interrupt is sent to the device driver. This has the special scancode of the scancode of the key that was pressed, with 0x80 added to the key's scancode.

The keyboard is currently able to read in key-presses and key-releases via interrupt-driven I/O. Usually, this information is sent to the BSD layer for further interpretation. As this is non-existent in the current prototype, the driver was modified to print out the scancode received in response to a device interrupt.

7.7 The PCI Device Driver

While there is only one PCI driver, it can be said that there is actually two parts to the PCI driver. One is the PCI driver itself, and the other is the PCI device nub, which allows devices to be attached to the PCI bus.

7.7.1 The I386 PCI driver

The PCI bus provides three separate 32-bit address spaces in which device registers may be mapped [SA99]. The address spaces are:

- Configuration I/O address space
- I/O address space
- Memory address space

How these address spaces are mapped depends on which platform is used. On IA-32 platforms, the I/O address space is directly mapped into the I/O address provided by the architecture, the memory address space is mapped one-to-one with the physical address space, and the configuration address space is accessed using a special set of registers.

The PCI bus can be operated in two different modes. They are configuration mode 1 and configuration mode 2.[2] The PCI driver detects whether PCI configuration mode 1 or mode 2 is requested, and uses the appropriate mode when accessing the PCI bus. There is evidence to suggest that this information is detected in the bootloader and then stored as a boot argument for the kernel, which in turn gets stored into the property tables. The PCI driver retrieves this information from the property table when it initializes itself. In the prototype, the driver was modified so that this information is hardcoded, though there is no reason why this is so other than for the sake of convenience.

This driver is required in order to support peripherial devices that reside on the PCI bus. Porting this driver to the Iguana-based I/O Kit prototype was a relatively simple process. Other than making the modification mentioned above, nothing was done other than to recompile the driver against the Iguana-based I/O Kit environment.

7.8 The Intel ICH5 ATA Device Driver

The Intel I/O Controller Hub 5 (ICH5) [Int] contains an ATA controller that can be used to drive ATA-compliant devices such as ATA disk drives. This chip is widely used on many Intel Pentium 4 machines.

7.8.1 The AppleIntelPIIXATA Driver

The AppleIntelPIIXATA is the driver that is used to drive the ATA controller that is present on the ICH5. It is capable of performing DMA operations. This driver was chosen because disk controllers are essential on almost all personal computers and portables. A running disk controller driver in an Iguana-based I/O Kit environment will likely provide some indication on the performance of devices that would typically exist on a user's system.

The AppleIntelPIIXATA driver was able to run unmodified, even with DMA enabled[3]. The driver is able to handle both read and write requests to the disk using DMA.

7.9 The Intel ICH5 AC97 Controller Driver

The AC97 is a specification for implementing low-cost audio functionality in hardware. Hardware which comply with the AC97 specification share common properties, but they still require a specific driver tailored specifically for each individual device in order to function properly. IA-32-based computers with AC97 compliant audio hardware are quite common. This driver was chosen because it is quite representative of what would be found on personal computers and portables.

In the I/O Kit, all AC97 audio drivers are in one single source driver package called AppleAC97Audio. It includes a driver, among others, for the AC97 audio controller found on the ICH5 hub. The specific driver in question is named AppleAC97AudioIntelICH.

Currently, it appears clear from the device driver output that the device is able to initialize correctly. However, no tests have been done to determine whether the device driver is indeed able to function properly. Devising a test program to do this may be a little difficult, because the audio drivers tie in closely with the user-level audio framework of Mac OS X. However, there is not much documentation as to exactly what happens to a piece of sound data after it is passed into the audio framework, nor does there appear to be any documentation on how the audio framework communicates with the audio

[2]Note that configuration mode 2 is obsolete. Since PCI 2.2, configuration mode 1 is the only configuration method available.

[3]One line was changed, but it is believed to be a bug in the driver.

drivers in order to produce sound at the output. In addition to this, even though having sound drivers would certainly be a big step towards having device drivers in an Iguana-based I/O Kit, it does not reveal much as far as the prototype's performance is concerned. There are no benchmarks that exist which can be used to test the performance of sound drivers and in turn estimate the performance of the Iguana-based I/O Kit prototype.

Chapter 8

Evaluation

This chapter describes the tests performed in order to evaluate driver performance on the I/O Kit prototype. It discusses the results that arise from these tests and states the outstanding problems and potential solutions that currently affect performance in the prototype.

8.1 LMbench

LMbench [McVa] is a well-established benchmark for UNIX-based operating systems written by McVoy and Staelin [MS96]. The LMbench benchmark suite comes with many tests that test system performance. It is those that are directly related to raw disk performance that of interest. In particular, the lmdd benchmark is a disk benchmark that can be used for testing disk performance.

8.1.1 The lmdd benchmark

The lmdd benchmark [McVb] is a variant of the original UNIX `dd` command. It copies a specified input to a specified output with possible conversions. It is particularly useful for timing I/O because it has the ability to print out timing statistics after completing. One common usage is to measure disk performance by reading or writing to a disk from the filesystem. In order to test raw disk performance, care must be taken to ensure that the file is not already in the operating system's buffer cache if it is being read, because it could artificially inflate transfer rate significantly. For similar reasons, if the file is being written to, then care must be taken to ensure that the file is flushed back to the disk after completing.

8.1.2 The memsize program

The `memsize` program is a utility for determining the amount of memory should be used for benchmarking purposes. In particular, for the lmdd benchmark, it determines the amount of data that should be moved.

8.2 Evaluation Method

8.2.1 Hardware

For hardware, a Dell Optiplex G270 Pentium 4 at 2.8 GHz fitted with 512 megabytes of memory was used. The disk is a 40 gigabyte Seagate Barracuda 7200RPM disk. The disk is plugged into the onboard ICH5 ATA controller. Both the disk and the controller are capable of operating at ultra ATA

mode 5 (ATA100). The choice to use the Dell machine over Apple's IA-32-based transition platform was because there was not enough time to get a functional DMA driver running on the transition machine. A polling I/O (PIO) driver could not be used, because the hard disks on the transition machines were connected via Serial ATA (SATA), which Apple's generic PIO driver does not support.

8.2.2 Operating system and drivers

The version Wombat used for testing corresponds to the 2.6.10 version of the Linux kernel. For native Linux, Knoppix version 3.9, which is a version of Linux that is able to boot off a CD-ROM device, was used. Knoppix contained a 2.6.11 Linux kernel. Darwin 8.2 was used for benchmarking. In all cases, HFS+ filesystem, which is the native filesystem for Darwin and Mac OS X, was used. In native Linux, Linux's native piix ATA driver was used, while for Darwin and Wombat, the AppleIntelPIIXATA I/O Kit driver was used. Benchmarks using Wombat with the Linux piix driver is not possible as Wombat does not currently have support for native Linux drivers. Finally, Linux was set not to overcommit memory. This is done so that Linux does not try to hand out more memory than it claims to be able to offer, which may affect the number returned by the `memsize` program.

The amount of memory usable by Wombat is fixed when the protection domain for it is created by Iguana's initialization program. In the current prototype, this is defined to be 32 megabytes. As a measure to ensure fairness, the Knoppix was booted with the `mem=` argument which constrains the amount of RAM to emulate a similar memory environment to that of Wombat, such that the `memsize` utility from LMbench gives similar results. On Wombat, this yielded 14 megabytes, while for Knoppix, `mem=54M` was used, which yielded 14 megabytes. The XNU kernel contains a similar boot argument called `maxmem=` which provides similar functionality. The `memsize` program does not appear to behave deterministically when used in conjunction with the `maxmem=` kernel argument. It was determined experimentally that using a size of 48 megabytes will give roughly the same number (16 megabytes).

For this test, Wombat was modified in three ways. Firstly, the *memcpy()* function in Wombat is unoptimized for the IA-32 architecture. This was replaced with a slightly more optimized version, since the I/O Kit glue driver spends a significant amount of time copying to and from the shared data region. Also, the stock implementation of Wombat's interrupt loop[1] was modified to stress the processor less, because the stock implementation appeared to be woken up too frequently and performance suffered as a result. Finally, the Wombat kernel build configuration file was modified to include built-in support for the HFS+ filesystem.

The fact that different Linux versions were used for the evaluation deserves a short discussion. Using different Linux versions represents a testing a slightly different code base, which may affect the results measured. However, the transition from 2.6.10 to 2.6.11 is a revision point release. The benchmarks use fairly well-tested kernel subsystems which did not substantially change during the transition. Hence, it is anticipated that the difference in the data gathered due to the fact that two different Linux versions are used should be quite negligible.

8.2.3 Benchmarks

The prototype was tested by reading a fixed amount of data (14 megabytes) into a buffer from the the raw block device. Different block sizes were used in order to determine the effect of the block size on driver throughput. No write tests were performed, as the Darwin operating system lived on the disk that was being tested. A write to the raw device would mean that the filesystem would be destroyed. Sufficient time and resources were not available to install a separate disk on the machine for testing purposes.

[1] The interrupt loop in Wombat is a thread of execution which does interrupt handling.

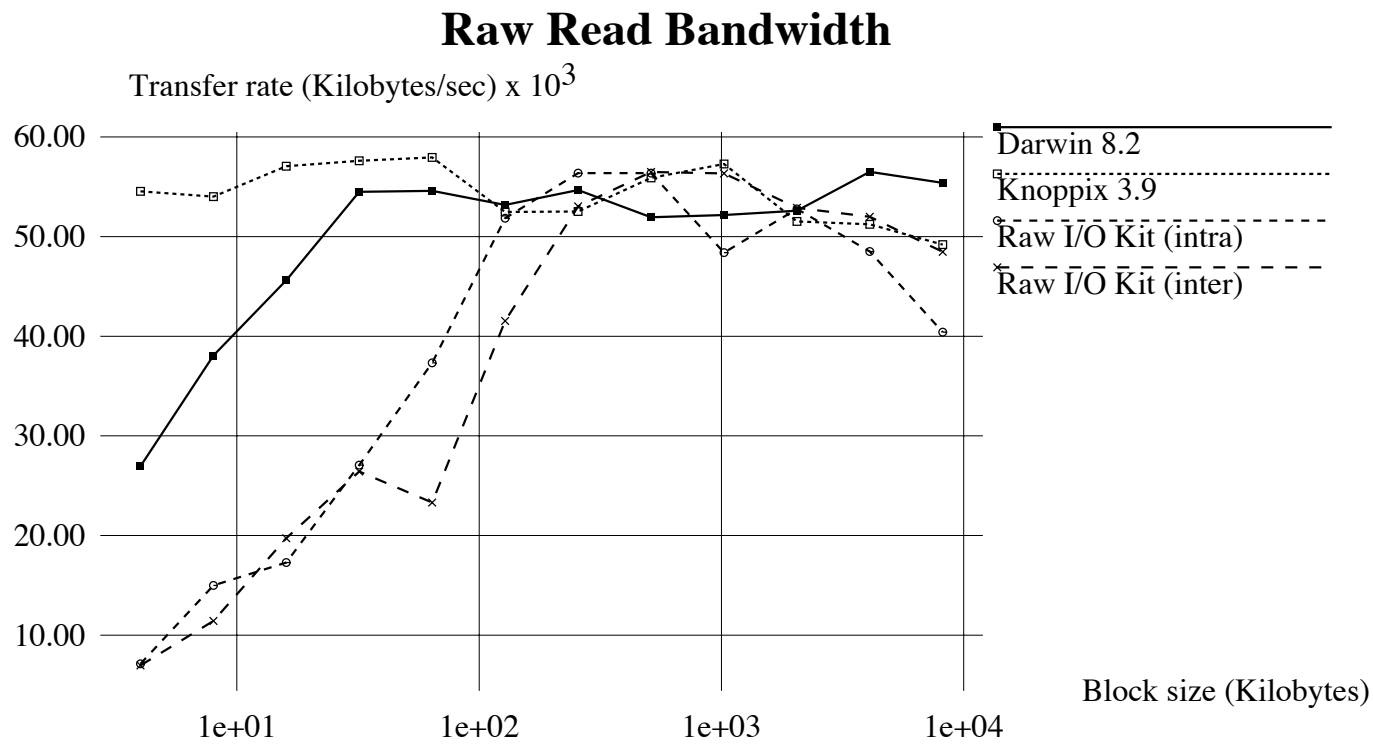

Figure 8.1: LMbench lmdd read benchmark with raw I/O Kit read benchmark.

Two different variations were used to measure disk throughput. First, test code was inserted into the I/O Kit Iguana server to directly read in data from the disk using the BSD I/O interface functions provided by the I/O Kit. This tests the performance of the I/O Kit prototype.

The same test was applied but with the client running in a separate protection domain to the I/O Kit server. This was done as an attempt to instrument the effects of context-switching on the performance of the disk driver.

The device was also read under Linux and Darwin using the `lmdd` command so that these results can be used as a reference for comparing performance results. The exact commands that were used to run the benchmarks can be found in Appendix B.

8.3 Results

Figure 8.1 shows the results obtained from the I/O Kit prototype. It shows that for small block sizes, it suffers from poor throughput compared to the native Mach-based implementation. Performance is also poor compared to Linux. However, at block sizes of 128 kilobytes or above, it shows comparable performance compared to the reference data.

It is speculated that one of the major contributing factors to the poor performance when using small block sizes may be attributed to the rudimentary Mach emulation layer in the prototype. This can be seen from the very similar curves for both the intra-protection-domain case, which should be fast since the I/O Kit is called into directly as a normal function call, and the inter-protection-domain case, which should be slower since it has the overhead of context-switching.

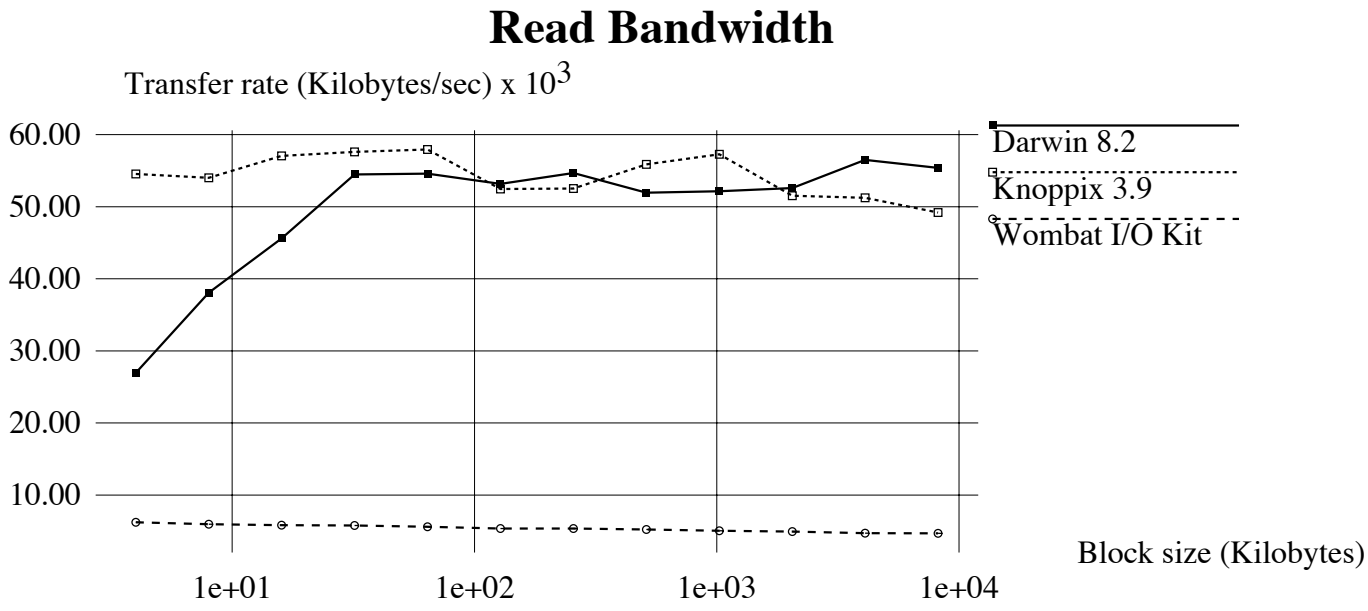

Figure 8.2: LMbench lmdd read benchmark.

8.4 I/O Kit on Wombat

An effort was made to determine how the prototype might perform when plugged into a production UNIX-like operating system. The Wombat environment provided a good test bed for this. The same benchmark that was run on the native Linux and Darwin systems were re-run under the Wombat environment. As a side-effect, having an operating system that supported HFS+ means that writes, not just reads, could be tested too, since writes could be tested indirectly by going through the filesystem. The results were plotted along with the numbers obtained from native Linux and native Darwin.

From the read and write tests from LMbench shown in Figure 8.2 and Figure 8.3 respectively, it shows that the prototype in combination with Wombat exhibits poor performance. In addition to this, the downward trend in throughput with an increasing block size is problematic, since in theory, the reduced number of context-switches that are required to be made using larger block sizes to transfer the same amount of data should imply that performance should increase with the block size, not decrease.

In order to determine whether the downward trend was due to a problem in the block size or that lmdd programs were run in succession, the benchmark run was modified on Wombat to run only one test per boot cycle. The numbers were gathered and replotted along with the previous results gathered for native Linux and Darwin.

As can be seen from Figure 8.4 and Figure 8.5, the results show that it has been somewhat improved, but its performance under Wombat is still very poor. There are too many different factors that may affect the results of the benchmark that no real conclusion can be drawn from the graphs, except that in conjunction with the results obtained from the raw benchmark, the performance gap is likely due to interfacing problems with Wombat, or with Wombat itself.

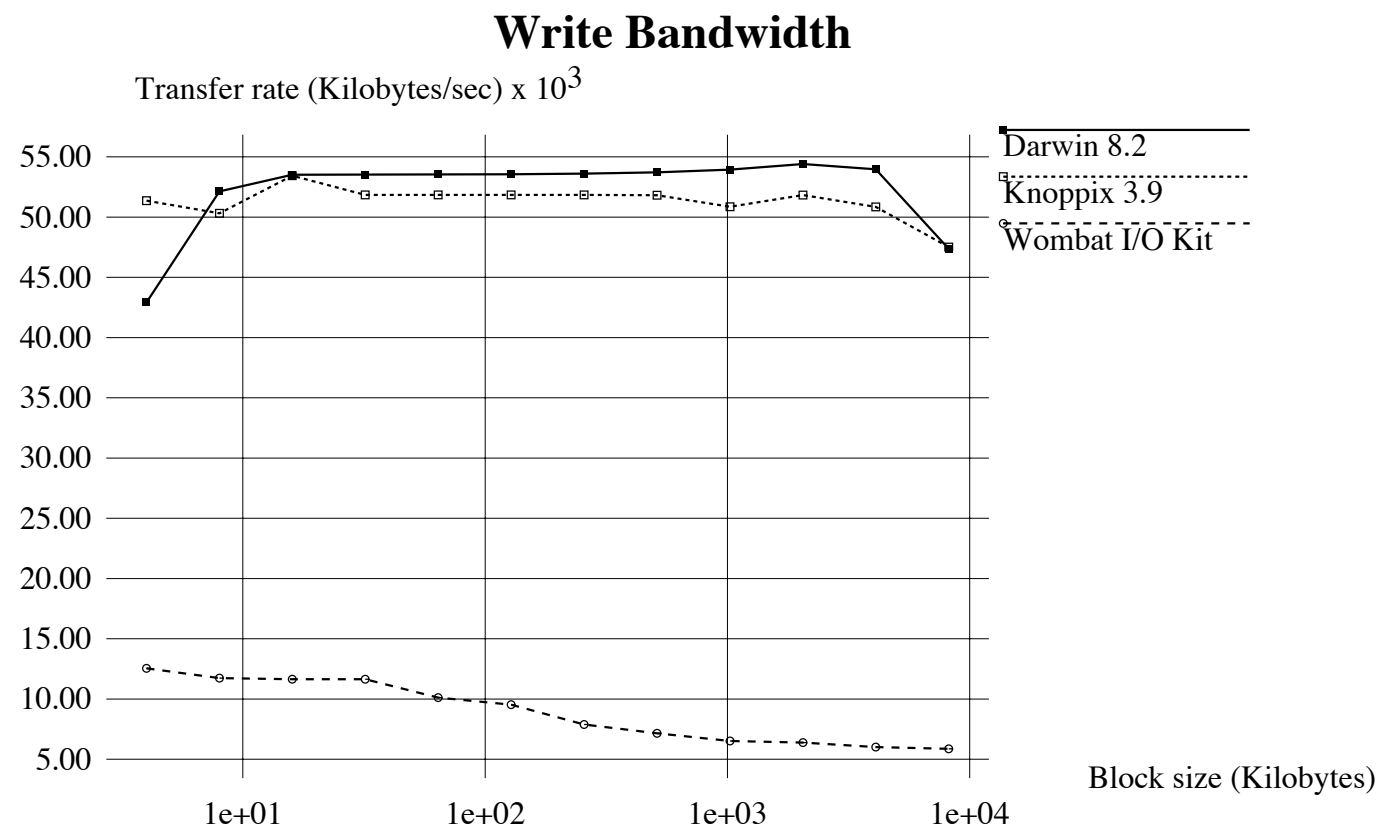

Figure 8.3: LMbench lmdd write benchmark.

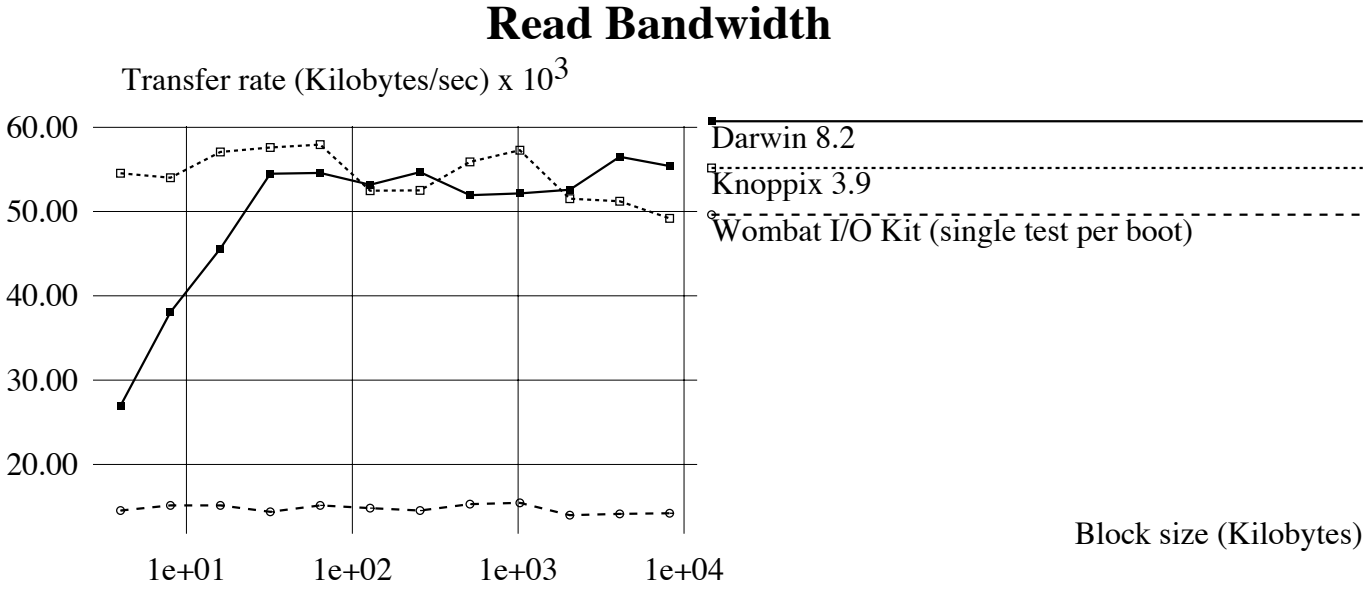

Figure 8.4: Modified LMbench lmdd read benchmark.

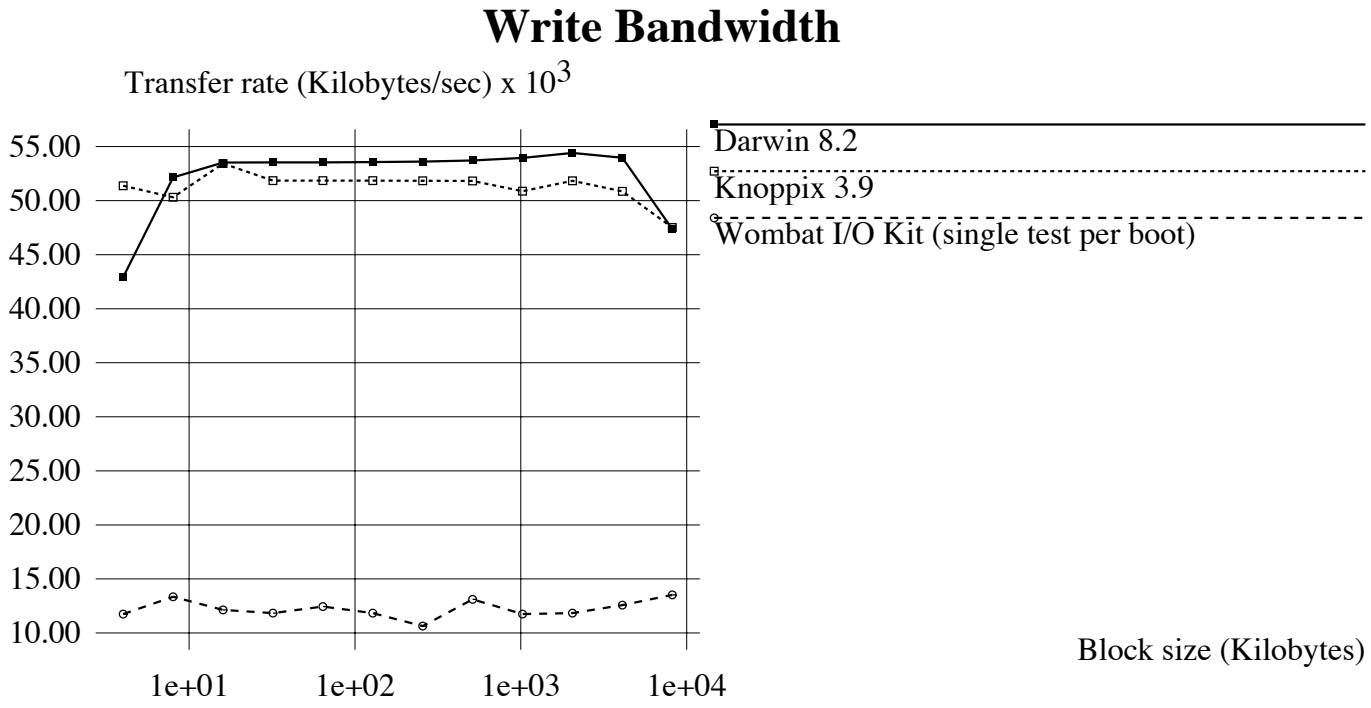

Figure 8.5: Modified LMbench lmdd write benchmark.

Chapter 9

Future Work

9.1 Introduction

Much has been achieved in this project. However, there are still certain areas where there are research opportunities and there are still questions which remain unanswered. This section intends to detail what might be desirable to investigate into in the near future.

9.2 Total OS Independence

Porting the I/O Kit and drivers involved identifying places where it was dependent on the OS in some way. One of the futher work on the I/O Kit would be to not only emulate Mach, but to completely remove the Mach references within the I/O Kit source code. Rather than simply making the I/O Kit source code L4-dependent instead of Mach-dependent, it is better to devise a generic, pluggable OS glue layer that allows the I/O Kit to be plugged into an arbitary operating system.

9.3 Isolated Drivers

As part of porting the I/O Kit to an L4-based system, device drivers now do not run with full privileges associated with the kernel's protection domain. Instead, they now run at user-level. It is much harder for a misbehaving driver to cause an unrecoverable operating system error.

For the sake of simplicity, in the current prototype, all drivers reside within the I/O Kit's protection domain. This is problematic, since a single device driver failure has the potential to corrupt anything within its protection domain, in this case, it may corrupt other drivers or even the I/O Kit itself. It would be useful to investigate into the possibility of isolating each device driver in its own protection domain. This has great potential in further reducing the amount of damage that a given driver may cause if it misbehaves.

9.4 Size Considerations

The I/O Kit was originally designed to run on computers where the code size was not so important. At the same time, the I/O Kit contains many useful features such as hot-plugging and power management, which may appeal to embedded devices that are resource-constrained. Future research could make the I/O Kit more accessible to embedded devices as far as resource constraints are concerned.

9.5 Performance Issues

The Iguana-based I/O Kit prototype is unoptimized. Throughput was shown to be poor when smaller block sizes were used for disk transfers. It is possible that with an optimized implementation, the overhead could further be reduced such that comparable performance to the Mach-based implementation can be achieved, even for smaller block sizes. Also, additional work is required to benchmark other classes of drivers, for example, network drivers, to show that good performance shown by the disk driver is not the exceptional case.

Chapter 10

Conclusion

The I/O Kit is the device driver framework for the Darwin operating system. The present Darwin is built on top of the Mach microkernel, which has design deficiencies related to IPC and has long been known to exhibit poor performance. L4 is a high performance, minimalistic microkernel that solves many of performance problems of Mach, and hence, there has been interest in migrating Darwin away from Mach to L4. An initial port of the BSD UNIX subsystem has shown promising results, but in order for an L4-Darwin port to gain widespread adoption, device drivers, and hence, the I/O Kit, must be ported over to L4 as well.

Device drivers to this day exhibit poor code reuse. A device driver targeted for one specific operating system will generally not run on another operating system, due to operating system specific dependencies. The I/O Kit abstracts operating system dependencies away from the device driver programmer, and has the potential to pave the way for cross operating system driver compatibility. A port of the I/O Kit should therefore be attempted to investigate the possibility of this.

Finally, the I/O Kit could potentially gain from having support for mechanisms to isolate device driver faults if it were ran on a high-performance microkernel, because reasons for putting device drivers in the same protection domain as the kernel, which are tightly related to performance, no longer apply.

A suitable L4-based system, called Iguana, was identified to be suitable base to port the I/O Kit onto. The I/O Kit, and the C++ runtime environment which it depends on have been ported from being XNU kernel components to Iguana-based application-level libraries. In addition, selected drivers were ported to the Iguana-based I/O Kit prototype.

Disk benchmarks indicate that performance is comparable to the native implementations if the block size used is sufficiently large. For smaller block sizes, performance is poor, most likely due to the high costs associated with context switching. However, it is anticipated that with careful optimizations, context switching costs in the current prototype could be much reduced, and as a result the Iguana-based I/O Kit prototype could achieve comparable performance even for smaller block sizes. Work remains to be done to show that the Iguana-based I/O Kit prototype shows good performance in the general case, and not only for disk performance.

Appendix A

Mach Symbol Listing

This chapter gives a listing of the symbols that need to be present in order to compile and link the I/O Kit into a standalone executable image. These symbols were identified by the linker. A small number of functions have already been implemented, and thus are missing from this listing.

DoPreemptCall
_IOBSDNameMatching
_IODefaultCacheBits
_IOOFPathMatching
_LockTimeOut
_MutexSpin
__doprnt
_absolutetime_to_nanoseconds
_assert_wait
_assert_wait_timeout
_clock_absolutetime_interval_to_deadline
_clock_get_uptime
_clock_interval_to_absolutetime_interval
_clock_interval_to_deadline
_clock_wakeup_calendar
_cnputc
_conslog_putc
_copypv
_current_act
_dbugprintf
_debug_mode
_device_pager_deallocate
_device_pager_populate_object
_device_pager_setup
_flush_dcache64
_iokit_alloc_object_port
_iokit_destroy_object_port
_iokit_lookup_connect_ref_current_task
_iokit_make_send_right
_iokit_release_port
_iokit_retain_port
_iokit_switch_object_port
_kernel_map
_kernel_pmap
_kernel_task
_kernel_thread
_kernel_upl_abort
_kmod_create_fake
_kmod_create_internal
_kmod_load_extension
_kmod_lookupbyname_locked
_kmod_retain
_kmod_start_or_stop
_lock_alloc
_lock_done
_lock_free
_lock_write
_mach_msg_send_from_kernel
_master_device_port
_ml_static_mfree
_ml_static_ptovirt
_mutex_alloc
_mutex_free
_mutex_lock_acquire
_mutex_lock_wait
_mutex_preblock_wait
_mutex_unlock_wakeup
_pmap_find_phys
_printf
_rootdevice
_simple_lock
_simple_lock_init
_simple_unlock
_snprintf
_sprintf
_sscanf
_static_memory_end
_sync_internal
_thread_call_allocate
_thread_call_cancel
_thread_call_enter1
_thread_call_enter1_delayed
_thread_call_enter_delayed
_thread_call_free
_thread_cancel_timer
_thread_set_timer_deadline
_thread_sleep_mutex
_thread_sleep_usimple_lock
_upl_get_internal_pagelist_offset
_upl_offset_to_pagelist
_vm_map
_vm_map_copyin_common
_vm_map_copyout
_vm_map_deallocate
_vm_map_get_upl
_vm_map_reference
_vm_region_64

Appendix B

Benchmark Script

The benchmark script used for all benchmark tests is essentially the same. The only difference is the different device naming scheme for the device used in the read tests. For Darwin, the device is /dev/rdisk0s1, for Linux, it is /dev/hda1, finally, for Wombat with I/O Kit, the name is /dev/iokitdisk0. In Darwin, the /dev/rdisk0s1 is actually a character device. This is unavoidable as the root filesystem is mounted on that particular device and Darwin by administrative policy disallows read access to it while the filesystem is mounted on a block device. The character device must be used in order to circumvent this. It should be noted that it appears the I/O Kit makes no distinction of whether it is reading from a block device or a character device, in that they are read or written to in exactly the same manner. Hence, using the character device should not affect the results obtained. Linux makes no such distinction, nor does Wombat with I/O Kit.

```
#!/bin/sh

echo 2 > /proc/sys/vm/overcommit_memory

mount /mnt

# copied from lmbench
MB=‘memsize 1024‘
HALF="512 1k 2k 4k 8k 16k 32k 64k 128k 256k 512k 1m"
ALL="$HALF 2m"
AVAILKB=‘expr $MB \* 1024‘
i=4
while [ $i -le $MB ]
do
ALL="$ALL ${i}m"
h=‘expr $i / 2‘
HALF="$HALF ${m}m"
i=‘expr $i \* 2‘
done
```

```
echo "Running LMbench"

# Make sure that these are there first
mkdir -p /mnt/tmp
touch /mnt/tmp/fileXXX
FILE=/mnt/tmp/fileXXX

BLOCKSIZ="4k 8k 16k 32k 64k 128k 256k 512k 1m 2m 4m 8m"
BLOCKSIZREV="8m 4m 2m 1m 512k 256k 128k 64k 32k 16k 8k 4k"

# try write bandwidth.

# starts off as umounted.
umount /mnt

# try with different block size, starting from 4K
echo "WRITE BANDWIDTH, SEQUENTIAL BLOCKSIZE"

for i in $BLOCKSIZ; do
mount /mnt
lmdd label="File $FILE write bandwidth, bs=$i: " \
of=$FILE move=${MB}m bs=$i fsync=1 print=3
rm $FILE
umount /mnt
# can be used to verify things if you find things
# a bit dodgy.
#mount /mnt
#lmdd label="File $FILE write bandwidth, bs=$i, no fsync: " \
# of=$FILE move=${MB}m bs=$i fsync=0 print=3
#rm $FILE
#umount /mnt
done
echo ""

# try with different block size, reversed.
# This is commented out, but it can be used to verify things
# if you find the results a bit dodgy.

#echo "WRITE BANDWIDTH, REVERSED BLOCK SIZE"

#for i in $BLOCKSIZREV; do
# mount /mnt
# lmdd label="File $FILE write bandwidth, bs=$i: " \
# of=$FILE move=${MB}m bs=$i fsync=1 print=3
# rm $FILE
# umount /mnt
# mount /mnt
# lmdd label="File $FILE write bandwidth, bs=$i, no fsync: " \
```

```
# of=$FILE move=${MB}m bs=$i fsync=0 print=3
# rm $FILE
# umount /mnt
#done
#echo ""

# try to read it back, using various blocksizes.  Each time,
# we must unmount in order to get rid of the adverse effect
# of the cache on the benchmark.

OLDFILE=$FILE
FILE=/dev/hda1

echo "READ BANDWIDTH, SEQUENTIAL BLOCKSIZE"

for i in $BLOCKSIZ; do
mount /mnt
lmdd label="File $FILE read bandwidth, bs=$i: " if=$FILE \
move=${MB}m bs=$i fsync=1 print=3
umount /mnt
done
echo ""

# This can be used to verify dodginess, perhaps.
#echo "READ BANDWIDTH, REVERSE BLOCKSIZE"

#for i in $BLOCKSIZREV; do
# mount /mnt
# lmdd label="File $FILE read bandwidth, bs=$i: " if=$FILE \
# move=${MB}m bs=$i fsync=1 print=3
# umount /mnt
#done
#echo ""

# Reset the file.
FILE=$OLDFILE

# mount this back, as other tests may follow, and expect
# the filesystem to be mounted.

mount /mnt

# This tests the buffer cache mainly and is useful only
# for comparison purposes to make sure that you aren't
# doing anything wrong.

#echo "read bandwidth"
#for i in $ALL; do bw_file_rd $i io_only $FILE; done
```

```
#echo ""

#echo "read open2close bandwidth"
#for i in $ALL; do bw_file_rd $i open2close $FILE; done
#echo ""

echo "Unmounting."
umount /mnt

echo "All done."
```

Bibliography

[Appa] Apple Computer, Inc. Apple Darwin Releases. `http://www.opensource.apple.com/darwinsource/`.

[Appb] Apple Computer, Inc. Apple-modified GCC source code exerpt — gcc/cp/typeck.c. `http://www.opensource.apple.com/darwinsource/10.4.2/gcc_os_35-3506/gcc/cp/typeck.c`.

[Appc] Apple Computer, Inc. Apple PS/2 controller driver. `http://www.opensource.apple.com`.

[Appd] Apple Computer, Inc. Darwin project FAQ. `http://developer.apple.com/darwin/projects/projects/darwin/faq.html`.

[Appe] Apple Computer, Inc. Mac OS X System Architecture. `http://developer.apple.com/macosx/architecture/index.html`.

[App04] Apple Computer, Inc. *Introduction to I/O Kit Fundamentals*, 2004.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993.

[CCC+] Hewett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd, and Toshiba Corporation. *Advanced Configuration and Power Interface Specification.*

[Chu04] Peter Chubb. Linux kernel infrastructure for user-level device drivers. In *Linux.conf.au*, Adelaide, Australia, January 2004.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on OS Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[EG05] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Proceedings of the 9th Asia-Pacific Computer Systems Architecture Conference, Beijing, China*, September 2005.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Leprau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 38–51, St Malo, France, October 1997.

[FD03] Nicholas FitzRoy-Dale. Python on Mungi. BSc(Hons) thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2003.

[Flu] Flux Research Group. The OSKit Project. http://www.cs.utah.edu/flux/oskit/.

[Hei] Gernot Heiser. ANNOUNCE: NICTA L4-embedded API. https://lists.ira.uni-karlsruhe.de/pipermail/l4ka/2005-October/001393.html.

[Hei04] Gernot Heiser. Iguana: A protection and resource manager for embedded systems. http://www.disy.cse.unsw.edu.au/Software/Iguana/iguanax4.pdf, 2004.

[Hel94] Johannes Helander. Unix under Mach: The Lites server, 1994.

[HERV93] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. Technical Report UNSW-CSE-TR-9314, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1993.

[HHL+97] 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 OS Principles*, pages 66–77, St. Malo, France, October 1997.

[Hun97] Galen C. Hunt. Creating user-mode device drivers with a proxy. In *Proceedings of the 1st USENIX Windows NT Workshop*, 1997.

[Int] Intel Corporation. *Intel 82801EB I/O Controller Hub 5 (ICH5) / Intel 82801ER I/O Controller Hub 5 R (ICH5R) Datasheet*.

[L4K01] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual Version X.2*. University of Karlsruhe, October 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[Les02] Ben Leslie. Mungi device drivers. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2002. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[Lev] Steve Levenez. Unix history. http://www.levenez.com/unix/.

[Lie95a] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report 933, GMD SET-RS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany, November 1995.

[Lie95b] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[LUC+] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Submitted to Eurosys 2006, available on request.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, December 2004.

[McVa] Larry McVoy. LMbench. http://www.bitmover.com/lmbench/.

[McVb] Larry McVoy. lmdd manual page. Available from LMbench source distribution. http://www.bitmover.com/lmbench/lmbench2.tar.gz.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, USA, January 1996.

[NeX] NeXTstep 3.3 — developer documentation. `http://www.channelu.com/NeXT/NeXTStep/3.3/nd/`.

[NICa] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual*. `http://www.l4hq.org/docs/manuals/refman-n1.pdf`.

[NICb] Iguana. `http://www.disy.cse.unsw.edu.au/Software/Iguana`.

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

[RJO+89] R.F. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. *Spring COMPCON*, pages 176–8, 1989.

[Roo04] Joshua Root. L4-Darwin on 64-bit PowerPC. Undergraduate thesis interim report, available on request, 2004.

[SA99] Tom Shanley and Don Anderson. *PCI System Architecture*. Mindshare, Inc., 1999.

[SABL04] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on OS Principles*, Bolton Landing (Lake George), New York, USA, October 2003.

[Sch01] Lambert Schaelicke. *Architectural Support for User-Level I/O*. PhD thesis, University of Utah, 2001.

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

[Tec] Jungo Software Technologies. WinDriver - USB/PCI Device Driver Development Tools. `http://www.jungo.com/windriver.html`.

[Tsu04] Mark Tsui. Nooks Benchmarking. Summer Report, available on request, 2004.

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

[Whe] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[Won03] Ka-shu Wong. MacOS X on L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 2003.