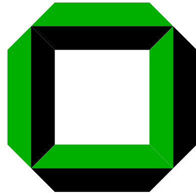

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Binary Device Driver Reuse

Bernhard Poess

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Joshua LeVasseur

22. März 2007

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 22. März 2007

Bernhard Poess

Abstract

This work presents a novel approach to driver reuse and isolation. We propose to deprivilege binary device drivers and to use the original kernel API to interface with the reused drivers.

We isolate device drivers by running them as regular applications in user-mode. The original environment is emulated to the reused device drivers. The drivers run as kernel-mode drivers in their native environment. We use lightweight virtualization to deprivilege device driver binaries and run them in user-mode. To demonstrate our approach we have developed a reference system that reuses Linux 2.6 device drivers on top of a microkernel-based component operating system. We successfully reuse network device drivers. Measurements show that our implementation achieves performance comparable to native Linux systems. However, the CPU utilization of our implementation is significantly lower than that of the Linux system. With roughly 23K source lines of code needed for our emulation environment, the engineering effort is very low.

Acknowledgments

First of all, I want to thank my advisors, Prof. Dr. Frank Bellosa and Joshua LeVasseur. Throughout this work and my studies they provided me with inspiration, motivation and exceptional expertise. Without their help, this work would not have been possible.

Thanks go to James McCuller for fulfilling all my hardware wishes and to Andrea Engelhardt for help with all the "small" issues that arose in everyday University life. In addition I want to thank all members of the System Architecture Group in Karlsruhe for providing a friendly and productive work environment.

Thanks go also to Dr. Uwe Dannowski and Jan Stoess for their patience in answering all my questions. Furthermore I want to thank Markus Osswald, Manuel Hammerich and Mark Johnson for proofreading my thesis.

A very special thank you to my parents, Barbara and Engelbert. I'm deeply grateful for all the selfless support and love they give me.

Contents

1 Introduction **13**
- 1.1 Motivation 13
- 1.2 State of the Art 14
- 1.3 Approach 14

2 Related Work **17**
- 2.1 Virtual Machines 17
  - 2.1.1 Simulation 18
  - 2.1.2 In-Place Virtualization 18
  - 2.1.3 Pure Virtualization 19
  - 2.1.4 Para-Virtualization 19
  - 2.1.5 Pre-Virtualization 20
- 2.2 Device Driver 20
  - 2.2.1 Device Communication 20
- 2.3 Driver Reuse 21
  - 2.3.1 Cohosting 22
  - 2.3.2 Virtual Machines 23
  - 2.3.3 Emulation 23
- 2.4 System Dependability 24
  - 2.4.1 Nooks 25
  - 2.4.2 Virtual Machines 25
  - 2.4.3 User-mode device drivers 25

3 Linux Driver Support **27**
- 3.1 Driver Architecture 27
  - 3.1.1 Kernel Extensions 27
  - 3.1.2 Device Driver 29
- 3.2 Kernel API 30
  - 3.2.1 Processes and Scheduling 30
  - 3.2.2 Synchronization 30
  - 3.2.3 Timing 31
  - 3.2.4 Deferred Work 31
  - 3.2.5 Memory 31
- 3.3 Device Driver Model 32
  - 3.3.1 Communication with hardware 33
  - 3.3.2 Interrupt Handling 33
- 3.4 Execution Environment 33

4 Design **35**
4.1 Requirements and Goals 35
4.2 Overview 35
4.2.1 Base Environment 36
4.2.2 Device Driver Environment 37
4.2.3 Applications 38
4.3 User-mode Device Drivers 38
4.4 Lightweight Virtualization 39
4.5 Linux Kernel Emulation 39
4.5.1 Emulation Obstacles 39
4.5.2 Code Reuse 41
4.5.3 Memory Management 41
4.5.4 Processes and Scheduling 42
4.5.5 Synchronization 43
4.5.6 Interrupts 44

5 Implementation **47**
5.1 Overview 47
5.2 Base Environment 48
5.2.1 Virtual Memory 48
5.2.2 I/O space partitioning 49
5.2.3 Dataspaces 50
5.3 Device Driver Environment 50
5.3.1 Emulation Layer 50
5.3.2 Driver Layer 53
5.3.3 Virtual machine Layer 54
5.4 Networking 55
5.4.1 Overview 56
5.4.2 Packet Representation 57

6 Evaluation **61**
6.1 Test Overview 61
6.1.1 Test Environments 61
6.1.2 CPU Related Measuring 61
6.2 Software Engineering 62
6.2.1 Engineering Effort 62
6.2.2 Memory Footprint 63
6.3 Transmission Test 63
6.3.1 Overview 64
6.3.2 Test Assembling 64
6.3.3 Throughput 66
6.3.4 CPU Utilization 66
6.3.5 Cache Usage 66

7 Conclusion and Future Work **71**

List of Figures

2.1 Virtualization Approaches 18
2.2 Simplified Interrupt Handling 21
2.3 Approaches to Driver Reuse 22
2.4 Approaches to Improve System Dependability 25

3.1 Linux Driver Architecture 28
3.2 Linux Thread States 30
3.3 Linux Memory Layout 32

4.1 Component Overview 36
4.2 Example system based on L4 37
4.3 User Mode Device Driver 38
4.4 Linux Semaphore Emulation 43
4.5 Linux Interrupt Priority Emulation 44
4.6 Linux Interrupt Handling Emulation 45

5.1 Implementation Overview 47
5.2 Dataspaces 49
5.3 Software Layers in a Device Driver Environment 50
5.4 Linux schedule() Emulation 51
5.5 Correlation between mem_map and page frames 53
5.6 Privileged Instructions and the Corresponding Emulation Functions 54
5.7 The Five Layers of the Internet Protocol Suite 55
5.8 Linux Interrupt Handling Emulation 56
5.9 Linux Packet Buffer Layout 57
5.10 Linux Packet Buffer Operations 58

6.1 Source Lines of Code 62
6.2 Symbols Imported by Linux Drivers 63
6.3 Memory Footprint of the Program Binaries 64
6.4 Streaming Send Test Receiver Loop 65
6.5 Streaming Send Test Linux Sender Loop 65
6.6 Streaming Send Test Packet Generator Loop 65
6.7 Streaming Send Test Network Throughput 66
6.8 Time for Packet Generator Operations 67
6.9 Streaming Send Test CPU Utilization 67
6.10 Streaming Send Test L2 Cache Misses 68
6.11 Streaming Send Test TLB Cache Misses 68

Chapter 1

Introduction

1.1 Motivation

Managing hardware devices is one of the core tasks of an operating system. Device management is usually done by device drivers. There exist a lot of devices and writing device drivers for all of them from scratch takes a lot of engineering effort. Especially new operating systems often lack support for a wide range of devices. Existing operating systems such as Microsoft Windows and Linux have build a big device driver base over years. It is therefore self-evident to *reuse* drivers from Windows or Linux in a new operating system. The engineering effort to reuse device drivers is only the engineering effort to build a reuse environment. It is thus significantly lower than the effort to build a complete driver base. In addition existing device driver's are debugged. They already deal with undocumented errors in certain device revisions. The bug rate of these device drivers is much lower than the bug rate of new drivers. Thus the bug rate of the reused drivers consists mainly of the bug rate of the reuse environment. The code size of the reuse environment is smaller than the code size of a complete driver base and therefore the bug rate of the reuse environment is smaller than that of a newly written driver base.

Device drivers are usually written by device manufacturers. Such device drivers are the intellectual property (IP) of the respective device manufacturers. Most manufacturers do not want to publish the source code of a driver due to IP concerns. A driver reuse solution must take that into account and support the reuse of *binary* device drivers.

The cost for system downtime (e.g. for large servers) is increasing [16]. System failures are a major source for system downtime. Device drivers account for a large number of system failures [11]. If device drivers execute in kernel mode, driver errors directly affect the whole system. Device driver *isolation* isolates a device driver from the kernel. If device drivers are isolated, errors in a driver are also isolated from the rest of the system. Therefore device driver isolation improves system dependability. All operating systems that provide a big driver base (Microsoft Windows, Linux) use non-isolated kernel-mode drivers. Full device driver isolation however requires device drivers to run in user-mode. A reuse solution that wants to isolate drivers must also deal with the *deprivileging* of device drivers from kernel- into user-mode.

Users of reused device drivers expect the drivers to perform as good as the drivers in the original system. A driver reuse environment must aim to achieve the same *performance* as the native environment.

A new software that wants to be widely adapted must be runnable on common platforms. A reuse environment must therefore be portable to e.g. Microsoft Windows or Linux. Its design has to be *hypervisor independent*.

1.2 State of the Art

There exist several approaches to driver reuse and/or isolation. *Cohosting* runs two operating systems parallel in kernel-mode with one of them only donating device drivers (donor OS) to the "real" system. Cohosted systems do not provide any isolation [2, 48]. *Virtual machine* based approaches add isolation to cohosting. The donor OS runs in a virtual machine. The host OS communicates with the donor OS in the virtual machine via special interfaces [21, 39]. While virtual machine based approaches add isolation, they hurt the performance of the system due to the overhead introduced by the virtual machines. *Emulation* based approaches take device drivers out of the donor OS and transplant them into the new OS. OSKit [18] and other approaches [4, 23] use a kernel-mode emulation environment and kernel-mode device drivers. Because they run in kernel-mode these environments and the drivers are not isolated. Nooks [49] uses kernel-mode drivers but provides a week form of isolation by putting the OS kernel and drivers into different address spaces. The address spaces only isolate kernel memory from driver memory, they fail to protect against non-memory related driver faults. A known technique to driver isolation are *user-mode* device drivers [17, 20, 27, 40, 46]. User-mode device drivers completely isolate the driver from the operating system. However all these approaches require to rewrite drivers from scratch. Writing all drivers from scratch imposes a high engineering effort. Van Maren et al. [44] and other projects [25, 36] use emulation to run kernel-mode device drivers in user-mode. They alter the source code of the drivers to deprivilege them and to adapt them to their emulation environment. None of these approaches delivered good performance results. In addition they are unable to use binary device drivers because of the source code alterations.

1.3 Approach

The lack of a solution that reuses and isolates binary device drivers with good performance motivated our approach. In this work we present a novel approach to binary device driver reuse and isolation. We suggest to use *emulation* and *user-mode* device drivers.

We reuse kernel-mode device drivers in user-mode (deprivileging). To make kernel-mode drivers runnable without altering their source code we use *lightweight* virtualization. Lightweight virtualization works on virtualized driver binaries. A virtualized driver binary must conform to a virtualization standard [3, 38, 47] that supports in-place virtualization. A virtualized driver binary therefore uses two APIs, the original OS kernel's API and a virtualization API. In-place virtualization is a fast virtualization technique that allows to perform virtualization in the same address space as the guest OS. Because device drivers use only very

few privileged instructions our virtualization layer is very lightweight.

We reuse device drivers via emulation. We emulate an OS kernel environment to the deprivileged driver. The driver uses the original operating systems's kernel API to communicate with the emulation environment.

We implemented our approach on top of the L4 microkernel [41] and reused Linux network device drivers. Linux allows to load device drivers at runtime via the module interface. We use that module interface to cut off the drivers from the Linux kernel. We construct our emulation environment using a bottom up approach starting with the functions that the driver modules import. Step by step we filled in missing functionality.

To evaluate our approach we use a UDP packet generator. A client runs our system and transmits packets from the generator via the network. On the server side an application measured the throughput of the received packets. In addition to the network throughput we also measured the CPU utilization during the send phase.

The structure of this thesis is as follows: In chapter 2 we discuss background and work related to our approach. In chapter 3 we analyze the Linux kernel and present the design of our work. Chapter 4 gives details on interesting implementation issues of our emulation environment. In chapter 5 we evaluate our prototype and present our test results. Finally chapter 6 summarizes important aspects of our approach and discusses if we met our goals.

Chapter 2

Related Work

In this chapter we first present background material about virtual machines and device drivers. Afterwards we describe past approaches to reuse drivers and improvement system dependability.

2.1 Virtual Machines

Virtual machines evolved from the idea of executing multiple operating systems on a single hardware platform with strong isolation between them. A virtual machine is a software abstraction of a hardware platform. The logic that implements virtual machines is called the virtual machine monitor (VMM). An operating system that runs inside a virtual machine is called a guest operating system (guest OS). The operating system that the virtual machine runs on is called the host OS.

An OS uses special instructions called privileged instructions to communicate with the system and to access operating system specific functionality of the processor. To help isolate the operating system kernel from the rest of the system modern processors implement different privilege levels. Depending on the privilege level in which code executes it has access to privileged instructions. Operating systems are designed with the prerequisite that they run in the highest privilege level of the system (privileged mode). To ensure isolation between virtual machines and protection of the host OS, the VMM can not allow a guest OS to execute in privileged mode. The execution of a guest OS in a less privileged mode (user mode) is called deprivileging of the guest OS. The process of changing a guest OS to make it runnable in user mode is called virtualization. The virtualization process depends on the underlying processor architecture. On some architectures only privileged instructions have to be virtualized. The instruction set of some architectures (e.g. the Intel IA32 instruction set [30]) however contains instructions that have a different behavior when executed in user mode instead of privileged mode. These instructions are called sensitive instructions. In contrast to privileged instructions which cause a synchronous interrupt when executed in user mode, sensitive instructions e.g. return different values. Due to that behavior sensitive instructions are harder to virtualize than privileged instructions because there is no processor mechanism that can be used to catch their execution.

In the following we will discuss approaches to virtualization with respect to Intel's IA32 architecture. The virtualization techniques apply to every other processor with an instruction set that contains privileged and sensitive instructions. Figure 2.1 presents an overview of current virtualization approaches. Pure virtualization operates on unmodified guest operating systems. Virtualization and emulation is performed at runtime. Para- and pre-virtualization virtualize a guest OS by modifying it. Only emulation has to be done during runtime. In-place virtualization operates in the same address space as the guest OS.

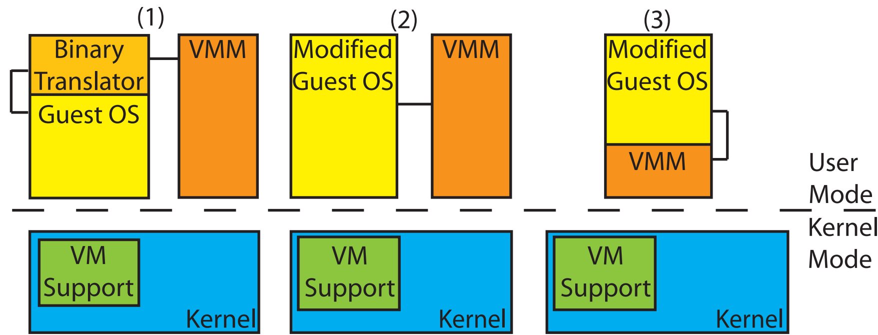

Figure 2.1: Virtualization Approaches
(1) Pure Virtualization (2) Para/Pre- Virtualization (3) In-Place Virtualization

2.1.1 Simulation

Simulators like QEMU [7] or bochs [35] emulate the complete instruction set of a CPU in software. Simulators are hardware-independent, they can be run on CPUs other than the one they emulate (e.g. QEMU emulates Intel's IA32 instruction set but also runs on the Power PC). While simulators are a hardware-independent approach, their major drawback is the performance overhead that is caused by the complete emulation. A simulator needs to fetch, decode and emulate every instruction in software. The simulation engineering effort depends on the size of the instruction set of the emulated CPU.

2.1.2 In-Place Virtualization

In-Place virtualization [38] moves the virtual machine monitor (VMM) into the same address space as the guest OS. Calls to emulation functions in the VMM do not need context switches anymore. Instead a local jump to the emulation code is sufficient. In-place virtualization reduces the emulation overhead significantly. The problem is that the VMM code has to be protected against the modifications by untrusted code. Guest OSs assume full access to their address space. To be able to use in-place virtualization the guest OS has to be aware of the VMM.

2.1.3 Pure Virtualization

Pure virtualization utilizes an unmodified operating system. If the processor architecture's instruction set contains only privileged instructions which cause a switch to privileged mode when executed in user mode the VMM may catch these switches, emulate the instruction and continues execution. On processor architecture's with sensitive instructions however this approach is impossible because sensitive instructions execute in user mode with a different result instead of causing a switch to privileged mode. VMware builds virtual machines that use binary translation [1] to implement pure virtualization on architectures that contain sensitive instructions. Binary translation parses the guest code at runtime, passes all non-sensitive instructions through to the processor and emulates the sensitive instructions. While this approach has less overhead than a simulator, there is still a significant overhead caused by the binary translation. First of all parsing the guest code introduces overhead. Second, the guest OS assumes full control over the address space. However the VMM needs to reserve a region of the address space of the guest for the binary translator. The code in the region must be sufficiently big to contain code that parses the guest code and switches to a different address space where the VMM emulates the instruction. Emulation of a sensitive instruction in this scenario can be sometimes done in-place (in the same address space) but may also lead to at least two context switches between the guest OS and the VMM. The code parsing and the context switches lead to performance degradation.

2.1.4 Para-Virtualization

Para-virtualization [52] like implemented by [24] or [6] does not require emulation of sensitive instructions. Instead a new software interface (paravirt interface) is defined that resembles the original hardware interface without privileged and sensitive instructions. Instead the paravirt interface defines in addition functions that directly call into the VMM. The simplest paravirt interface replaces privileged and sensitive instructions with similar code in the VMM. However the para-virtualization technique allows the introduction of new functionality that is not defined in the original hardware interface. The new functionality can significantly increase performance. The downside of para-virtualization is the high engineering effort to port an existing operating system to the new paravirt interface. Another problem of para-virtualization is the need to access the source code of the guest OS. If the source code of an operating system is not available and the manufacturer is not willing to make the necessary modifications the operating system can not be run in the para-virtualization environment. There are ongoing efforts to specify a virtualization standard [3,47]. Such a standard would allow interoperability between para-virtualization VMMs and lower the engineering effort for operating system manufacturers because they have to port their OS to one interface instead of many different ones. Para-virtualization has a much higher performance than pure-virtualization because it avoids the overhead of binary translation. In addition, the paravirt interface can reduce the context switches between the VMM and a guest OS.

2.1.5 Pre-Virtualization

Pre-virtualization [38] was developed to lower the engineering effort needed to para-virtualize an OS while still maintaining the performance of para-virtualization implementations. Pre-virtualization automates the virtualization process. Instead of finding and changing occurrences of sensitive instructions manually, a parser is injected into the compilation chain. The parser locates sensitive instructions and adds a pad of no-operation instructions after each sensitive instruction. In addition the parser adds annotations about the location of sensitive instructions to the final binary. A VMM uses the no-operation pad to rewrite the sensitive instruction with emulation code. One problem for pre-virtualization are memory sensitive operations. In contrast to regular sensitive instructions which can be easily extracted out of the source code, memory sensitive operations like writing the page-tables are hard to find. Because of these problems the guest OS still has to be modified manually. However the engineering effort to do so is several magnitudes lower than the effort for para-virtualization [38].

2.2 Device Driver

A device driver is software that manages a device. It exports an abstract interface to users of the device. The users do not need to know device internals, instead they use the abstract interface. It is the duty of the device driver to map calls to the abstract interface onto device operations.

Modern operating systems treat device drivers as black boxes. They allow users to load additional device drivers at runtime. Device drivers use a special interface to communicate with the operating system. This interface is defined in the device driver application programming interface (driver API). The driver API is either a formal specification or implicitly defined by a set of functions that a kernel exports for use by device drivers. Closed source operating systems such as Microsoft Windows use the formal specification approach. The advantage of a well-defined API is durability. The API usually does not change for years. Open source operating systems (e.g. Linux, *BSD) use the implicit approach. The disadvantage of the implicit approaches is that the driver API can change with every new version of the OS. Every time the API changes, all drivers have to be adapted to the new API.

In traditional operating systems, drivers execute in kernel mode. Considering that research shows that drivers account for a large fraction of operating system errors [11] kernel mode drivers represent a severe risk to system dependability.

2.2.1 Device Communication

Device drivers communicate with a device via I/O registers or device specific memory. I/O registers on the IA-32 architecture are accessed via the I/O space. The I/O space is partitioned into I/O ports. Each I/O port maps directly to one device register. The IA-32 architecture provides special instructions `IN` and `OUT` to read/write to /from I/O ports. Device specific memory can be mapped into an address space. Code accesses device specific memory with the same instructions as conventional memory.

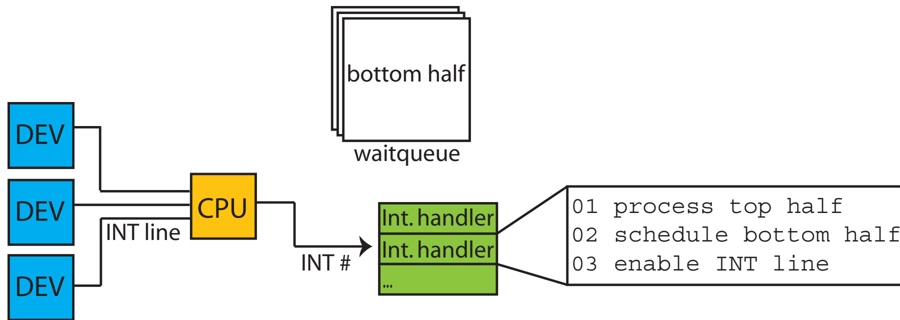

Figure 2.2: Simplified Interrupt Handling
The diagram shows a simplified view on interrupt handling from the perspective of a driver [31]. A device signals an interrupt via an interrupt line to the CPU. The interrupt number is defined by the interrupt line. The CPU looks up the interrupt handler in the interrupt lookup table using the interrupt number. Only the top half of an interrupt handler is registered with the CPU. The top half handler queues bottom halves during execution. The bottom halves are usually executed when no interrupt is pending.

Devices use asynchronous events (interrupts) to communicate with the system. Figure 2.2 shows a simplified diagram of the interrupt handling process. If e.g. a device is ready to perform a data transfer, it signals an interrupt to the processor. An interrupt is identified by its interrupt number. The processor then executes code that has been registered for that interrupt number. This code is called an interrupt handler. When an interrupt handler starts to execute, the interrupt line for its device is disabled. It is the duty of the interrupt handler to enable the interrupt line again when the system is in a state where it is ready to receive more interrupts from its device. Interrupts can happen very frequently, often multiple times each second. If an interrupt handler takes too long to execute, the system will lose interrupts. To circumvent the loss of interrupts, interrupt handlers are divided into a top- and a bottom half handler [15]. The top half contains only the code that is necessary to bring the system into a state where it can receive further interrupts. All other work that would be necessary is deferred to the bottom half handler. The bottom half handler is usually executed after all top half handlers have finished executing.

Not all devices use interrupts to signal events. For these devices a driver has to poll the device regularly to examine if action is needed.

2.3 Driver Reuse

Driver reuse refers to the process of taking drivers from one operating system (donor OS) for use in another OS (parasite OS). There are three different approaches to driver reuse. The first approach is called cohosting. The cohosting approach runs two operating systems in parallel in privileged mode. One is the donor OS and one is the parasite OS. The parasite OS uses the donor OS to manage devices. The second approach transplants the device drivers of the

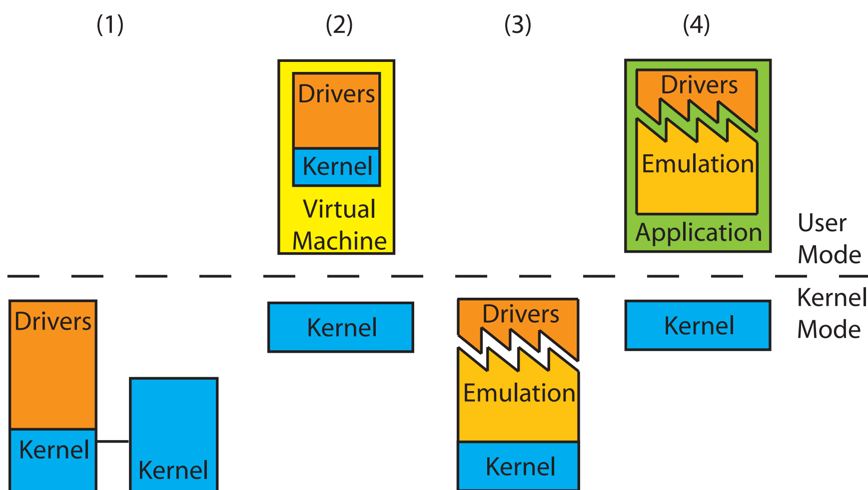

Figure 2.3: Approaches to Driver Reuse
(1) Co-hosting (2) Emulation in kernel-mode (3) Emulation in user-mode (4) Virtual machine based

donor OS into the parasite OS. The parasite OS emulates the donor OS to the drivers. The third approach runs the donor OS inside a virtual machine. The parasite OS communicates with the virtual machine to access a device through a loadable kernel module. In the following we will explain the three approaches in detail.

2.3.1 Cohosting

The cohosting approach [2,48] runs two operating systems parallel to each other in privileged mode. One of them acts as the donor operating system. It provides all the drivers. Cohosting separates the privileged mode into two worlds, the donor world and the parasite world. A switch between the two worlds (world-switch) is very similar to a context switch and equally costly. The parasite OS loads a kernel extension (bridge extension) into the donor OS to make the it aware of the parasite OS. In order to maintain full control over the donor OS the parasite OS has to have full control over interrupts. Some interrupts like e.g. timer interrupts are shared between the parasite and the donor OS. They are first handled in the parasite OS and then forwarded to the donor OS. All other device interrupts are directly handled by the donor OS [48]. The bridge extension in the donor OS ensures that a device driver in the donor OS transfers data to the parasite OS instead of to the donor OS if needed. If a program inside the parasite OS wants to send data to a device, the parasite OS first transfers the data to the bridge extensions which in turn pushes the data down to the donor OS device driver. World-switches between the worlds of both OSs are needed in device to system and system to device communication. These world-switches introduce additional overhead on a critical path. The donor OS and the parasite OS both execute in privileged mode. This co-location is very likely

Table 2.1: Approaches to driver reuse via virtual machines

Feature	**LeVasseur et al. [39]**	**Fraser et al. [21]**
bridge interface	customized	unified
# of virtual machines dedicated to driver OS	multiple	multiple
virtualization technique	para-virtualization	para-virtualization
modification of device drivers	unmodified	unmodified

to produce unpredictable side effects because at least the donor OS assumes full control over all system resources.

2.3.2 Virtual Machines

The need for better isolation between reused device drivers and the OS kernel inspired two approaches from LeVasseur et al. [39] and Fraser et al. [21] to achieve driver reuse through the use of virtual machines. Both approaches move a complete operating system kernel with additional drivers into a virtual machine. The operating system in the virtual machine is stripped down to the minimum functionality that is needed to run the OS kernel and drivers. Drivers inside the virtual machine use the hardware interface to access devices (pass-through access). Similar to cohosting a bridge extension is added to the donor OS kernel. The bridge extension is fully aware of the fact that it is running inside a virtualized environment. It uses special interfaces in the virtual machine monitor to transfer data to and from the virtualized OS. Table 2.1 outlines the differences between ref. [39] and ref. [21]. How the bridge interface is implemented depends on the trade-off between higher performance (customized interface) and lower engineering effort (unified interface). Because use of only one virtual machine would introduce a scalability problem on multiprocessor systems both approaches principally allow the use of multiple virtual machines. Both implementations show that the use of a dedicated virtualized driver OS implies a significantly higher CPU load. The high CPU load lowers performance.

2.3.3 Emulation

Various approaches [4,14,18,23,25,36,44] tried to achieve performance by transplanting drivers from the donor OS into the parasite OS. Instead of reusing the whole kernel and driver code base of the donor OS the parasite OS emulates the donor OS kernel personality to the drivers. An OS personality consists of a core kernel API, a driver API and an execution environment. Interfaces belong to the core kernel API if they are used by every kernel extension, not only device drivers. The driver API defines interfaces that are solely used by device driver kernel extensions. The driver API not only defines interfaces that device drivers use to call into the kernel, it also specifies a device driver model. A device driver model defines the behavior of device drivers and the kernel during driver to kernel and kernel to driver interaction. It also defines operations common to devices across device classes that the kernel is aware of. The execution

environment specifies the address space layout and the stack and thread layout.

Mach 4.0 [23], OSKit [18] and K42 [4] reuse Linux [42] device drivers. They leave the drivers in privilege mode and co-locate them with the new kernel. Between the new kernel and the Linux drivers an emulation layer emulates the Linux OS personality to the drivers. [23] and [4] are focused on the respective kernels Mach 4.0 [43] and K42 [33]. Both approaches are highly kernel dependent and not usable in different kernels. These dependencies make them unusable for future operating systems that follow a different design. OSKit [18] focused on research operating system. They designed a complete reuse system for OS components. These components are not only device drivers, but also e.g. file system drivers or a TCP/IP stack. They merged the interfaces of different kernels into unified interfaces. The emulation layer around the reused components maps calls to the original interface into calls to the unified interface. OSKit is designed to be hypervisor independent. Each component can work without the help of other components. The conversion to the unified interface introduces overhead e.g. additional memory copies if a reused component uses a buffer that is incompatible with the buffer of the unified interface.

Van Maren [44], Helmuth [25] and Lee [36] are approaches to user-mode driver reuse of kernel mode drivers. Van Maren attempts to port OSKit into user-mode on top of the Fluke [19, 32] microkernel. The design is driven by the architecture of the Fluke microkernel. Device drivers are implemented as user-mode servers. Interrupts are delivered to user-mode threads via IPC. There exist one thread for each interrupt vector. An interrupt line is disabled by the kernel when an interrupt is signaled on that line. The line is re-enabled from user-mode using a system call. The performance is poor compared to kernel-mode drivers [44]. Especially under high interrupt loads the performance of the fluke microkernel becomes a limiting factor. Helmuth and Lee both implement their approaches on top of the L4 [41] microkernel. Although Helmuth reuses Linux and Lee reuses I/O kit device drivers their design is very similar. Dedicated threads receive interrupts via IPC. An interrupt line is disabled by the kernel upon reception of a signal.The interrupt line is re-enabled by the kernel upon reception of a special IPC from user-mode. Helmuth's implementation covers only one sound card device driver. Therefore we can not make a statement about performance nor about completeness of the approach. Like Van Maren's the Lee's implementation suffers a performance degradation compared to native drivers. We suspect that the reason is the para-virtualized Linux kernel that they use to emulate a Linux user-mode environment.

2.4 System Dependability

The cost for system failures is constantly rising [49]. One step to reduce the likelihood of system failures and to improve system dependability is to protect the system against device drivers. Traditional operating systems execute drivers in the same privileged mode as the kernel. In this scheme, a bug in a driver is very likely to crash the whole system. Given the fact that drivers have a bug rate three to seven times higher [11] than kernel code the need to protect against drivers is evident. A first step to achieve protection is to isolate device drivers from the rest of the operating system [21].

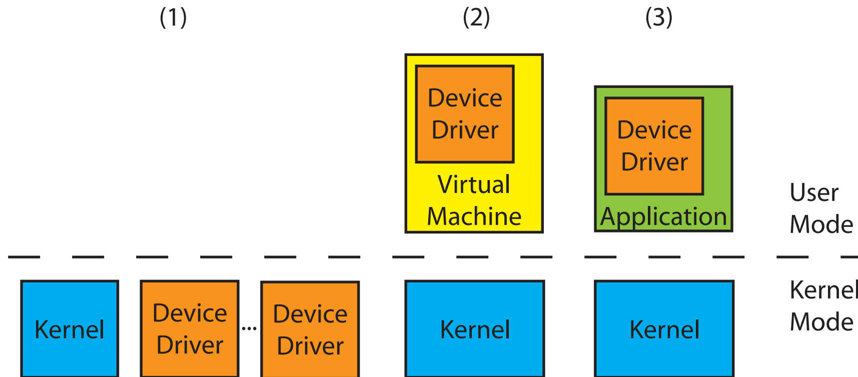

Figure 2.4: Approaches to Improve System Dependability
(1) show the Nooks [49] approach. Core kernel and drivers all execute in privileged mode. (2) shows virtual machine based approaches [21,39]. Device drivers are completely isolated from the system. (3) is the architecture as used by user-mode device drivers. Drivers are also completely isolated.

2.4.1 Nooks

The Nooks project [49] leaves drivers in kernel-mode but puts them in protection domains. The core kernel and all extensions execute in different protection domains. A protection domain in Nooks is simply an address space. Core kernel and extensions communicate via synchronous IPC. An interposition layer ensures data integrity across protection domains.

The Nooks approach does not provide full isolation of device drivers. Device drivers are still able to circumvent the scheduling mechanisms of the kernel. In addition drivers can still use all privileged instructions, which can lead to potential deadlocks of the OS.

2.4.2 Virtual Machines

A virtual machine (VM) emulates a hardware interface specifically it intercepts all attempts to access devices in the system. Thus Code inside a VM is completely isolated from the rest of the OS. [21,39] not only allow driver reuse but also improve system dependability through the use of virtual machines. These approaches need a high engineering effort and have a significant performance overhead.

2.4.3 User-mode device drivers

A common approach to protect the OS against device drivers is to move drivers from kernel into user-mode [17, 20, 22, 27, 40, 44, 46]. Drivers in user-mode are completely isolated from the OS kernel. In addition if the hardware supports it, devices can be protected against device drivers too. However all approaches (except for the virtual machine based approaches above) to user-mode device drivers so far required to rewrite device drivers from scratch which imposes a high engineering effort.

Chapter 3

Linux Driver Support

Design details of the emulation environment depend on the donor operating system. We reuse Linux 2.6 device drivers. In this chapter we describe interfaces of the Linux kernel from the perspective of device drivers. Device drivers only use a limited set of all kernel interfaces, thus the description is not complete. Further information about the Linux kernel and the Linux device drivers is available from [9, 13].

3.1 Driver Architecture

The Linux driver architecture is divided into several components (see Figure 3.1). At the bottom of the Linux device driver stack is the hardware abstraction layer (HAL). The HAL makes the basic hardware platform (processor, interrupt & memory controller) accessible via a unified interface. Bus drivers manage bus devices via the HAL. They offer interfaces to enumerate and access devices on the bus. Device drivers rely on bus drivers to manage devices. Buses provide unique identifiers (ID) for each connected device. A device driver registers itself with every bus that might have attached devices that the device driver wants to manage. Along with the registration the device driver provides the IDs of its manageable devices. The Linux kernel ensures that the drivers get attached to every matching device on a bus. Device drivers are either used by other modules or they directly expose themselves to user-space via Linux's user-space device interface. User-mode applications use system calls to utilize device drivers or modules in the kernel.

3.1.1 Kernel Extensions

Linux allows extension of the kernel at runtime. These loadable components are called modules. The Linux kernel exports a set of functions for use by modules. These symbols and the symbols that modules export are managed in a global flat symbol namespace. Modules are dynamic link libraries with additional symbol export information. When a module is loaded, Linux matches the undefined symbols of the module with the symbol namespace. If the namespace contains all needed symbols, the module is linked into the kernel's address space. From that point module and kernel code are not treated separately. Module code

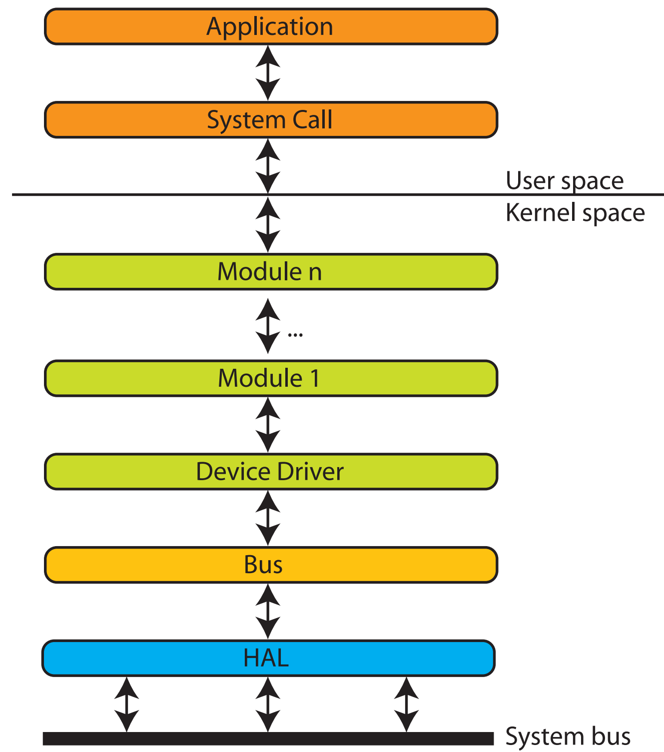

Figure 3.1: Linux Driver Architecture

Linux kernel extensions are called modules. Modules can be anything, from file system drivers over device drivers or bus drivers. Modules can be recursively stacked upon each other. Not every module exposes itself to user space. Only the topmost modules in the module stack communicate with user space via well defined interfaces.

has the same rights as static kernel code. Linux does not provide standardized mechanisms for stacking of modules. Stacking of modules is solely done via linking to symbols in the symbol namespace. There are no predefined data structures for data transfer between stacked modules.

User-space applications access modules via Linux's system call interface. Device drivers are usually represented by special file nodes to user-space. An open on the file node is mapped to an open call on the device and so on. However device file nodes support a special interface called ioctl. The ioctl interface allows a driver to provide functionality beyond the file interface. Each ioctl call contains at least a function number and a parameter. Those parameters are directly transferred to the driver. The function number corresponds to special functions in the driver like e.g. setting certain device parameter. The parameter can also be a pointer to user-space memory. Linux allows drivers to read and write arbitrary user-space memory. This allows a driver to exchange information with user-space applications.

3.1.2 Device Driver

Linux distinguishes between three main classes of devices: character- , block- and network devices. Character devices send and receive data as a stream of bytes, one at a time. Examples are the serial port or printers. A block device sends and receives data in blocks of bytes. A classic example for block devices are disks. The blocks have a fixed size. For performance reasons, the block size is always a power of 2. The smallest block size that Linux supports is 512 byte. Communication with character and block devices is straight forward. Upon reception of a system call, the driver issues a read or write request to the device. It then blocks and waits for an interrupt to arrive from that device. Other user-space requests are queued in the meantime. Upon completion of the I/O request, the device signals an interrupt. The driver receives the interrupt and completes the data transfer. Interrupts for character and block devices never occur unexpected, asynchronous to a user-request. They always occur synchronously after the device driver has issued an I/O request.

From a data-size perspective, network devices are block devices. They work with dynamic block sizes. However due to the nature of networks, they can receive data asynchronously and thus signal interrupts at any time. A network device driver has to be able to transfer data from the device without any user-space application waiting for it. This asynchronous data transfer is achieved through queuing. Linux maintains a packet queue for each device. Arriving packets are queued here for further processing by e.g. the next component in the network stack. User-space applications usually do not operate on network device file nodes. They use another special file type called sockets. Sockets are an abstract interface that allows buffered communication between two endpoints. E.g. a send operation on a socket in the network case usually corresponds to one or more send request issued by the driver to the device. Ref. [9] contains further information about the socket concept. The driver does not need to know about sockets. During e.g. a send operation, the message is processed by various layers in the network stack [50] until finally the bottom-most layer pushes the packets to the network driver.

3.2 Kernel API

We categorized all interfaces to the kernel API that are used by but are not specific to device drivers.

3.2.1 Processes and Scheduling

Device drivers use threads for example as servers that process requests to the driver. Requests are queued and the thread processes them one after another. The notions of tasks, processes and threads are not clearly defined in the Linux kernel. Every thread e.g. not only has a thread control block (called thread_info) but also a task control block (called task_struct) associated with it. Since the non existing naming scheme can lead to confusion we introduce terms that we will use throughout this thesis to describe these abstractions. We will call thread of execution a thread. There can be multiple executing threads in one address space. A process consists of an address space and all threads that execute inside that address space. Linux threads have three different states: running, waiting and blocked (see Figure 3.2). Running means that the thread currently executes on a CPU. A waiting thread would be able to execute but all CPU's are used by other threads. The thread waits to be scheduled. Blocked means that the thread is waiting for an external event. Such a thread can not be scheduled. Threads give their CPU away (yield) through a call to the scheduler.

Figure 3.2: Linux Thread States

3.2.2 Synchronization

The Linux kernel allows multiple parallel threads of execution inside the kernel. If these threads access shared data, they need synchronization mechanisms to protect the data. The simplest protection mechanisms that Linux supports are to switch off asynchronous events and spin locks. Switching off asynchronous events is achieved by disabling interrupt delivery. This is dangerous because if a bug occurs while interrupts are disabled the system may easily remain locked. Disabling interrupts is processor local. On multiprocessor systems other processors still execute regularly. That means that on multiprocessor systems a system global synchronization mechanism is needed. This global synchronization is achieved on Linux through the use of spin locks. Spin locks are only used on multiprocessor systems as they can easily lead to deadlocks on uniprocessor systems. Upon the basic synchronization mechanisms, Linux provides more convenient and less error prone solutions: Semaphores, Mutexes and Completions. Semaphores under Linux follow the classical semaphore scheme as described by Dijkstra. The `up` and `down` operations can be blocking or non-blocking. In the blocking case the down operation is guaranteed to succeed. If a semaphore that

a thread tries to acquire is locked, the thread is blocked and woken up by the up operation. It then checks again if the semaphore is locked and continues or blocks again accordingly. Mutual exclusions (mutex) under Linux are simply semaphores with a count of one. That means that at any given time, only one thread holds a semaphore lock. Completions are a lightweight mechanism that allows one thread to tell the other that a specific task is completed [13].

3.2.3 Timing

Timing refers to the concepts of delays and delayed work. The difference in the Linux kernel is that delays are blocking and delayed work is not. Delays are implemented as busy waiting loops. Thus delays should only be used for short time periods like a fraction of a second. Longer delays cause unnecessary processor overhead. However the advantage of delays is their accuracy, usually in the range of microseconds.

Delayed work is based on the timer interrupt. That means that every time the timer interrupt fires, the timer handler checks if the time for a specific delayed work item is used up. If so, it schedules execution of the delayed work. Device drivers use delays for example to make sure that a device has loaded a value into its registers. Delayed work is usually used for periodic tasks, e.g. to implement watchdogs [53].

3.2.4 Deferred Work

During execution a driver might want to defer work to a later point in time, e.g. during interrupt handling. Modern driver's interrupt handling routines are separated into two halves, a top and a bottom half [15]. The top half executes only the parts that are absolutely necessary to process the interrupt and defers the rest of the work to be done to a bottom half handler. The bottom halves are executed by the softirq thread. The softirq thread checks regularly if any deferred work is pending. In addition, a signal to the softirq thread immediately triggers deferred work execution if no interrupt handler is running.

3.2.5 Memory

Linux differentiates between three types of physical memory: DMA capable, normal and high memory. The types are directly coupled to Linux's address space layout (Figure 3.3. On modern systems, all physical memory is usable for DMA and thus the DMA and normal memory zone are equal. On older systems however, only a small area at the beginning of the physical address space is usable for DMA. Linux reserves the virtual memory between three and four gigabyte for the kernel (kernel region). Physical memory is mapped into the kernel region continuously starting at address zero. If the physical memory is bigger than the kernel region the upper, not directly accessible parts, are accessed via special mappings. DMA capable memory refers to all memory mapped into the kernel region that can be used for DMA access. High memory is physical memory that is not mapped into the kernel region and has to be accessed via specifically created mappings. Normal memory is all memory that is not DMA or high memory.

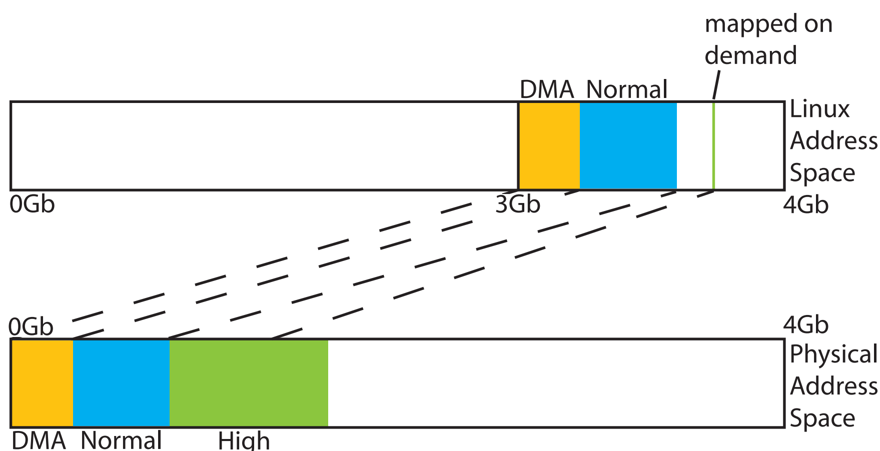

Figure 3.3: Linux Memory Layout
DMA Capable and a part of regular physical memory are mapped directly into the address space. High mem is mapped into the address space on demand.

There are three different aspects of memory management that drivers use: heap management, memory mappings and direct memory access (DMA). Heap management allocates arbitrary sized chunks of memory. Linux puts no size or alignment restrictions on memory allocated from the heap. However the kernel heap is restricted in its overall size. Memory mappings reserve whole pages for use by the driver. Therefore the supported granularity is the smallest page size that the hardware platform supports. There are special allocation functions that allow the reservation of continuous pages. Direct memory access is used to speed up data transfer between a device and the system. Instead of having the driver read all data out of device registers via I/O ports, the device accesses the physical memory directly via the bus. The driver translates the virtual address of the buffer that the device should use into a physical address and hands the physical over to the device during the I/O request.

3.3 Device Driver Model

A device driver model defines drivers from a kernel's point of view. It aggregates common mechanisms and abstractions for drivers into a unified model. The Linux driver model supports the following tasks [13]:

User-space Communication Linux provides a special virtual file system called sysfs. It is tightly coupled to the device model. It provides mechanisms for exposing device information to user-space and for user-space to driver communication.

Device Class Management Device classes abstract device specifics and aggregate functions that a group of devices supports into a device class. A driver

assigns a device to a class. The standard functions of that device can then be easily used by other kernel-code without specific knowledge of the device.

Power Management & System Shutdown To be able to shutdown devices or to switch the power state of a device, the Linux kernel maintains a tree like structure of device dependencies apart from hardware dependent mechanisms like e.g. ACPI [26]. The device model also provides a hardware independent, unified interface for power management and shutdown that a driver has to implement.

Hotplugging Hotplugging is a widely used concept in modern computer hardware. The hotplugging model has to deal with the addition and removal of devices during runtime. It uses the dependency tracking mechanism of the device model to notify components about the addition or removal of devices.

Object Lifecycle Management Hotplugging, power management and system shutdown need proper object lifecycle management. For example the kernel has to ensure that an object is not destroyed until the last entity that uses has shut down. The Linux device model provides automated mechanisms to control creation and destructions of dynamic objects that are used by several components.

3.3.1 Communication with hardware

Device drivers communicate with devices via I/O ports or device specific memory. Drivers access I/O ports via the hardware interface provided by the processor. Device memory must be mapped into a driver's address space. Linux provides functions that deal with access management and mapping of I/O ports and device memory.

3.3.2 Interrupt Handling

The Linux kernel manages the interrupt controller. Drivers register interrupt handler with the interrupt number of their device and hand over a function pointer. The function pointer points to the driver's top half interrupt handler. It is possible that multiple drivers register for one interrupt number (shared interrupts). Linux maintains an array with all registered interrupt handlers. If an interrupt is received, it calls all handlers that are registered for that interrupt number. It is the duty of the handler to check if the interrupt was sent by its device. The interrupt handler signals via a return value if it handled the interrupt or not.

3.4 Execution Environment

Linux maintains a kernel space stack for every thread in the system (kernel threads). The stacks have a fixed size and are aligned to their size. The thread control block (TCB) is located at the bottom of a stack. Thus accessing the TCB of a thread is done by masking out bits of the current stack address. The stacks are not protected against overflows. Linux also provides a process control

block (PCB) for every process in the system. Every TCB contains a pointer to the PCB of the process that the thread belongs too.

Chapter 4

Design

4.1 Requirements and Goals

First, we define the requirements and goals of our design. Most notably our design should provide:

Isolation We want to improve system dependability by isolating the device drivers we use. We want to do so without hurting compatibility with existing driver APIs. Improved system dependability with equally good performance will give as an advantage over traditional operating systems.

Performance Our approach is specifically targeted towards production based systems. On such systems we have to compete with established operating systems such as Linux or Microsoft Windows. Thus our approach needs to perform equally good concerning e.g. network throughput.

Low Engineering Effort Previous approaches to driver reuse had performance problems [21, 39] or required driver developers to completely rewrite their drivers [17, 20, 22, 27, 40, 44, 46]. Completely rewriting a driver requires a huge engineering effort for a large driver base. We put almost no engineering effort on the driver writer's and thus on the device manufacturer's side.

Hypervisor Independence To be widely adapted a novel approach must be usable on different operating systems not only an esoteric research system. Our design should be completely independent of the underlying system. We will show that Linux and Microsoft Windows would both be feasible for approach.

4.2 Overview

We reuse kernel-mode device drivers in user-mode. In a first step the device drivers are deprivileged using a novel lightweight in-place virtualization solution. To run the drivers we emulate the personality of the donor OS to the device drivers. The reused drivers run together with the virtual machine monitor (VMM) and the donor OS emulation in a protection domain called a device

driver environment (DD/Env). Figure 4.1 presents an overview of our reuse environment. Device driver environments (DD/Env) run on top of a base environment. The base environment has no knowledge about devices in the system. It partitions the I/O space to provide access management for device registers and device specific memory. A DD/Env contains the reused driver and emulation code. It runs as a regular user-mode application inside a protection domain. It exports the interfaces of the driver to clients. Clients can be applications or other DD/Envs. Applications use the abstract service interfaces of the DD/Env to communicate with devices.

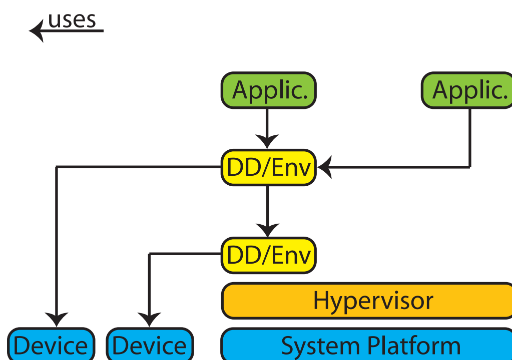

Figure 4.1: Component Overview
Device driver environments (DD/Envs) contain the reused driver. They offer services to applications and other DD/Envs. DD/Envs have exclusive access to devices.

4.2.1 Base Environment

The base environment is the only hypervisor dependent layer in our design. It converts operations such as mapping a physical page into a driver's address space into one or multiple calls to the underlying hypervisor. A base environment provides the following services: memory management, I/O space partitioning, CPU management and PCI management. Depending on the underlying kernel, most of this functionality may be implemented in kernel-mode. For example [12] describes kernel extensions for Linux to support user-mode device drivers. The base environment manages memory, CPU and the I/O space through protection domains. Each protection domain has one address space, one I/O space and one or more threads. Address spaces are used to manage physical to virtual memory mappings. The base environment provides mechanisms to map physical pages to address spaces in user-mode. The I/O space of a protection domain maps a subset of the global system I/O space for use by code inside the protection domain. Rights to access an I/O port are local to a protection domain. Access rights can be either shared or exclusive to the protection domain. A global management instance ensures proper management and setup of access rights. Threads are used to manage the CPU resource. We require our base environment to provide kernel-mode threads because we do not want a thread to block all

other threads in its protection domain when blocked in the kernel. In addition to protection domains, the base environment also manages the PCI subsystem. The PCI subsystem is used to find devices and to communicate with devices on the PCI bus.

Base Environment on L4

We use the L4 microkernel [41] in our approach. L4 runs as a privileged hypervisor in kernel-mode. It provides address spaces and threads. All other tasks are performed in user-mode (Figure 4.2). We provide memory management

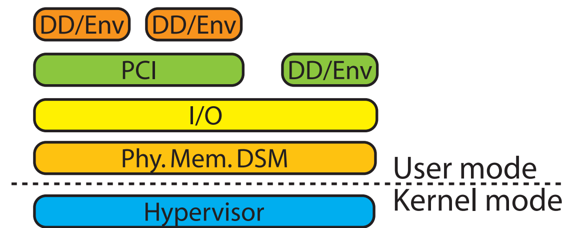

Figure 4.2: Example system based on L4
The L4 microkernel runs as a privileged hypervisor. User-mode servers build the base environment.

and paging through dataspaces.

A dataspace is a container that abstracts memory objects such as files, physical memory, shared memory regions [5]. A dataspace can contain any memory that is mappable or can be made mappable. If a thread wants to access data in a dataspace, the dataspace is attached, e.g. mapped into it. We fill address spaces by attaching dataspaces to them. For example to run an application, we attach dataspaces that contain the application code and data, heap and stack memory. Dataspaces are implemented by dataspace managers. A dataspace manager is simply a task that implements the dataspace protocol. Dataspace managers map regions of the dataspace either direct (the client ask for the region) or indirect (the client causes a page fault in a region that is attached to a dataspace manager). The dataspace protocol allows clients to pin a page and ask for its physical address. These mechanisms allow clients to use DMA

Partitioning of the global system I/O space is done by the I/O server. An application (e.g. a device driver) asks the I/O server to setup permissions to access an I/O port. Depending on whether the application is allowed to access the I/O port and the type (shared or exclusive) and state (reserved or free) of the I/O port, the I/O server grants access to the I/O port to the application.

The PCI server provides operations such as finding a specific device on the bus or communicate with devices via the bus.

4.2.2 Device Driver Environment

Device driver environments (DD/Env) run in user-mode inside protection domains. They consist of the virtualized device driver, a lightweight virtual machine monitor and the OS personality emulation. They offer the device driver

interface as services to applications. Since DD/Envs are regular applications, they can make use of the services of other DD/Envs if necessary. This allows stacking of DD/Envs.

4.2.3 Applications

Besides DD/Env other applications execute in the system. They use services of the DD/Envs or embed DD/Envs to communicate with hardware devices.

4.3 User-mode Device Drivers

We achieve isolation through the use of user-mode device drivers. We deprivilege kernel-mode device drivers and reuse them in user-mode. Our emulation environment maps calls to the original kernel-mode API into calls to our user-mode environment. That means that our user-mode environment has to be flexible enough to support any possible kernel-mode API. We achieve this flexibility by providing only a minimal set of mechanisms for memory and device management in the environment. All other functions and policies are implemented in the emulation environment.

There are two possible designs for user-mode device drivers (Figure 4.3) [17]. In the first design the driver is implemented in its own protection domain. It acts as a server that reacts to requests from (client) applications. The system is fully isolated from the driver with that design. However if multiple drivers and applications are stacked, the multiple protection domain crossings increase CPU load and decrease performance. Thus we also allow to embed drivers into applications. Embedding drivers means that on a driver failure we have to restart the whole application, not only the driver. These restarts are only possible if the application's interface is well enough defined to store the application's state between a restart.

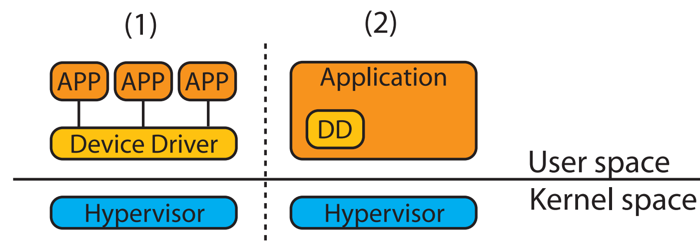

Figure 4.3: User Mode Device Driver
(1) Device driver acts as a server for applications (2) Device driver embedded into an application

Our framework does not enforce the usage of a specific design approach. We explicitly allow both scenarios. Different users have different needs. We believe that the trade-off between isolation and performance should be decided by the user and not by us.

4.4 Lightweight Virtualization

To run kernel mode drivers in user-mode we deprivilege them. The deprivileging is done in two step. The first step is applied by the driver manufacturer. Automated preprocessing converts the device driver to confirm to a virtualization standard. In our work we used pre-virtualization [38], however our design works with every other standard such as VMI [3] or paravirt-ops [47] too. Virtualization preprocessing produces a binary that is dynamically linkable to our in-place virtual machine monitor (VMM). Because drivers only use a few privileged instructions the VMM is very thin and lightweight. Our VMM only emulates asynchronous events. Pre-virtualization adds a byte pad of no-operations instructions after every privileged instructions. Additionally pre-virtualization adds information about the location of the privileged instruction to the binary. During the loading process of the binary our VMM locates the privileged instructions and replaces the byte pad with emulation code. Because we only emulate asynchronous events, the VMM has a very small size. Thus privileged instructions are emulated with the lowest possible overhead.

Emulation of the interrupt delivery flag A disabled interrupt flag means the currently running code will not be preempted under normal circumstances. If it is interrupted (e.g. on the x86 architecture via a non-maskable interrupt) it is the duty of the interrupt handler to ensure data integrity. We do not stop execution of all driver threads if the emulated interrupt flag is disabled. Instead we treat the code that is executed between the disable and enable interrupts instructions as a critical section. To protect the critical section, we use a global lock (interrupt lock). Every function that tries to read or write the interrupt flag has to acquire the interrupt lock beforehand. If a thread wants to save the interrupt flag, we acquire the lock, save the flag and release the lock. However if a thread disables interrupts, we only acquire the lock. We release it when interrupts are enabled again.

4.5 Linux Kernel Emulation

In the following section we describe the design of our Linux kernel emulation. Our emulation completely replaces the Linux kernel's memory management and scheduling functions. The replacement of Linux's memory management allows us to define our own address space layout.

4.5.1 Emulation Obstacles

Emulation of the Linux kernel to drivers imposes several problems, most of them are inherited from the fact that the Linux kernel uses an ad-hoc API approach. The Linux APIs are not documented and not formally specified. Linux simply exports a set of functions and variables that a driver can import. In the following we will give a detailed view on the problems and restrictions that the Linux API has with respect to emulation.

Emulation of the Linux kernel APIs is difficult because of several reasons: Missing modularization, encapsulation and documentation. Modularization is often broken in favor of gaining performance e.g. through the use of inlined

functions or direct pointer access. Export of global variables and compiled in constants break encapsulation. Missing documentation about assumptions and side effects complicates the emulation task even more. These missing features have two reasons, the lack of a formalized API specification and the Linux approach to open source development.

Binary Modules

We use binary Linux device driver modules. That means that we can't make any changes to a driver's source code. We are bound to conform to Linux's kernel API. For example unlike past approaches [25, 36, 44] we have to emulate every variable that a driver might import.

Full exposure of structures

Linux fully exposes data structures to device drivers. Drivers can access every field in the structure. These accesses are a problem for emulation. The emulation layer has to ensure that at any time the fields contain correct information. Given the fact that most structures are undocumented this can be a error prone task. Most drivers do not use the kernel related fields. However we have to protect our device driver environment against false accesses e.g. by catching access to pointers that we do not emulate. Another problem that arises from these accesses is that some structures force us to exactly emulate internal kernel functions. For example the `page` struct assumes a specific address space layout. This is explicitly bad for us because to speed up emulation we want to avoid duplicating the Linux virtual memory management.

Function inlining

Linux defines parts of its APIs as inline functions. The functions are automatically added to the driver code during compilation. Most times, an interface is partially defined via inline functions and partially in code modules. This mixture of inlined and non-inlined function causes difficulties for emulation. We have to completely emulate the behavior of the non-inlined function with all undocumented assumptions of the inlined functions. E.g. if an inlined function assumes that a specific lock is held when it is called, we have to ensure that the lock is held under every circumstance that may lead to a call to that function. These assumptions prevents us from using optimized functions that would have the same effect to the driver. An example are semaphores. The semaphore struct is fully exposed to drivers. All down operations are inlined for performance reasons. If a inlined down operation fails it calls an in-kernel function that deals with the failure. If the down and up operations would not be aligned, there would be no need to expose the semaphore struct to drivers. Instead the driver would use an abstract semaphore interface that would allow the interface implementation to implement optimized semaphore code.

Size and alignment restrictions

Another issue for emulation are undocumented size and alignment restrictions. Especially when it comes to size and alignment restrictions, the ad-hoc API approach of Linux shows its weakness. Some data structures in the Linux kernel

have size or alignment restrictions that are not even documented in the source code. An example are kernel thread stacks. Their size is fixed, usually to 8Kb. However some inline function and macros, e.g. the macro that returns a pointer to the TCB at the bottom of the stack, assume that the bottom address of a stack is also aligned to the size of the stack. Such assumptions are hard to find and hard to debug on errors.

Compiled in constants

Linux provides kernel extensions with compiled in constants. They are usually defined via the C preprocessor and pulled into the driver binary during compilation. Compiled in constants represent compile time policies. An example is the length of the periodic timer interrupt. Emulation code might want to change some of the policies for optimization reasons. However with compiled in constants this is not possible without changing the Linux source code.

Global Variables

Besides regular functions, Linux also exports global variables for use by kernel extensions. Often kernel code makes assumptions about the values of these variables or side effects of changes of the variables. These assumptions and side effects are undocumented and complicate the emulation of global variables. For example although it is not necessary for our timekeeping architecture we have to increment a special variable (`jiffies`) with each timer tick because device drivers use it to determine packet timeouts.

4.5.2 Code Reuse

Wherever possible we try to reuse original Linux kernel code if it meets the following criterion:

1. The code does not pull in memory management or scheduling related code
2. The code is runnable in user-mode

The first criterion ensures that we are able to use our own memory management code, which is trimmed for our user-mode interface and thus faster than the Linux code which is optimized for bare hardware. The first criterion also prevents unmodified reuse of source code that makes any assumptions about the address space layout. The second criterion prevents the reuse of code that contains sensitive instructions. Such code would either throw an exception when run in user-mode or even worse produce unpredictable side effects. We want to avoid both and thus we disallow the use of sensitive instructions.

4.5.3 Memory Management

Linux's memory management is written to run on bare hardware. For example it contains a lot of code that deals with the management of page frames and virtual memory. We do not need such code in our emulation environment. Thus we decided to not reuse Linux's memory management code at all. Instead we emulate Linux's memory allocation behavior.

To emulate Linux's memory management, our allocator has to fulfill the following requirements and restrictions:

- Allocate physical and I/O memory on a per page basis. If more than one page is requested, the pages have to be contiguous.
- Allocate conventional and physical memory in arbitrary sized chunks. If physical memory is allocated and a chunk spans over more than one page, the pages have to be contiguous (I). The caller of the allocation function may specify a special flag indicating that allocated physical memory has to be suitable for direct memory access (DMA).

We rely on the base environment to allocate memory on a per page basis. The base environment serves physical, conventional and I/O memory pages. If a driver requests a whole page, we map it into the driver's address space using the base environment. Conventional memory allocation has no special restrictions. We use a standard memory allocator to serve conventional memory. The conventional memory allocator retrieves conventional memory from the base environment on a per page basis and serves it in arbitrary chunks. Conventional memory is not suitable for DMA and might get swapped out by the base environment to backing store. Because it might get swapped, access to conventional memory is potentially slow which makes it unusable in fast paths (e.g. a driver's interrupt routine). In contrast to conventional memory, physical memory is guaranteed to remain mapped until it is released. The drawback to conventional memory is that physical memory is a limited resource. Allocation of physical memory in arbitrary sized chunks, is also more restricted than allocation of conventional memory. Our physical memory allocation consists of two independent parts: Allocation of pre-defined memory blocks and allocation of memory blocks with no prior knowledge of their size. To allocate memory blocks of pre-defined size we use a slab-like [8] allocator. If we have no prior knowledge of the memory block to be allocated we use a simple algorithm: If we have enough free space on a page for the block, we allocate it there. If the block would span over multiple pages, we request enough pages that meet restriction (I) and allocate the block on the new pages. If there is free space on the new pages left, we add it to our free space.

Direct memory access (DMA) is highly hardware dependent. E.g. on a system with an ISA bus, only the lower 16MB of the physical memory are suitable for DMA. We rely on the base environment to page-wise allocate DMA-able memory. If the requester requests DMA-able memory, our memory allocator takes memory out of the DMA pool instead of the regular physical memory pool. The DMA pool is backed by the base environment. Linux allows driver's to build their own DMA pools. These DMA pools are not allocated using the regular kernel allocator. Instead they use memory mappings to acquire physical pages suitable for DMA. DMA pools are not resizable. They serve chunks of DMA-able memory in a fixed size. We reused Linux's management code for DMA and replaced the code to establish memory mappings with calls to the base environment.

4.5.4 Processes and Scheduling

We mirror Linux threads using threads of the base environment. Each Linux thread corresponds to a thread in the base environment. We replace Linux's

scheduling with the scheduling of the base environment. Linux compiles information about the stack layout into drivers. To reuse drivers we have to ensure that the stack layout of our threads is the same as the stack layout of a Linux thread. Specifically that means that the stack size and alignment has to be the same. We emulate the thread states using the base environment. During thread execution, we ensure that the thread state information in Linux's thread control block (TCB) is always correct from Linux's perspective. Initially a thread is in the running state. If a thread asks the scheduler to put it into the waiting state, it wants to yield the CPU. We emulate yielding the CPU by indicating to the base environments scheduler that we voluntarily give up the rest of our time slice. We do not have to change the thread state here because if the thread is allowed to execute again, it will be in the running state. If a thread requests to be blocked, we use a base environment call to the scheduler to block it. Similarly if another thread unblocks it, we call to the base environment's scheduler to unblock it.

4.5.5 Synchronization

We emulate disabling of interrupts, spin-locks, semaphores and completions. Disabling interrupts is handled by the virtual machine monitor (VMM). Spin-locks use the following algorithm:

1. Disable interrupt delivery
2. If (multiprocessor system) acquire lock

Following Linux's spin-lock semantics, on uniprocessor systems, only interrupt delivery is switched off. The real lock is only acquired on multiprocessor systems. Acquiring the lock is not necessary on uniprocessor systems because disabling interrupt delivery stops execution preemption.

There are three different types of semaphores, blocking, interruptible and non-blocking semaphores (See Figure 4.4). Semaphores consist of a count that

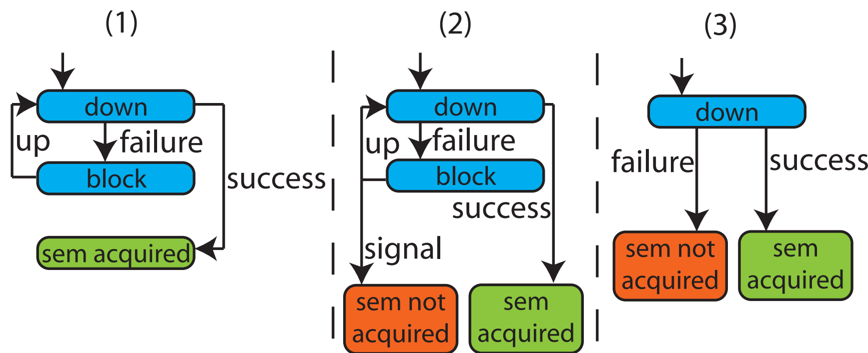

Figure 4.4: Linux Semaphore Emulation
(1) Blocking semaphore (2) Interruptible semaphore (3) Non-blocking semaphore

is atomically decremented if a thread tries to acquire the semaphore. The count

is atomically incremented when the semaphore is released. If the count is zero, the semaphore is locked. The three versions (blocking, interruptible and non-blocking) only differ in the case where the semaphore is locked. The blocking version blocks the caller until the semaphore is released by another thread. There is no other way of waking up the blocked caller. The interruptible version blocks the caller until another thread releases the semaphore or the blocked caller receives a signal. The non-blocking version does not block the caller that tries to acquire the semaphore but returns an error code that signals success or failure of the acquire operation.

Completions are similar to blocking semaphores with an initial count of zero. When a caller enters the completion we block him until another thread releases the completion.

4.5.6 Interrupts

The base environment delivers interrupts to user-mode. We use a dedicated thread for each interrupt source (interrupt thread). Each interrupt thread has a handler assigned to it. Upon reception of an interrupt, the interrupt executes the handler. When the handler is done executing, the interrupt thread blocks again and waits for the next interrupt. The interrupt threads have the highest priority in our emulation environment (Figure 4.5). We use strict priority scheduling. That means that at any time the threads with the highest priority always run. The second highest priority in our emulation environment has the softirq thread. It executes any pending deferred work. The lowest priority have the worker threads. They execute all regular code. Drivers register interrupt

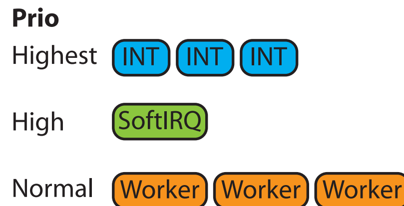

Figure 4.5: Linux Interrupt Priority Emulation
We use strict priority scheduling. Each interrupt thread is dedicated to one interrupt source. The softirq thread regularly checks for scheduled softirqs and executes them if necessary. The softirq thread runs if no interrupt thread is active. The worker threads are allowed to execute when no interrupt or softirq thread is runnable.

handlers with the kernel. Figure 4.6 outlines the execution flow during an interrupt.

An interrupt handler is characterized by an interrupt number, a function and flags. The interrupt number corresponds to the devices interrupt line, that the device driver wants to receive interrupts from. The function has to be executed by the interrupt handling code upon reception of interrupt. The flags specify whether the interrupt handler is capable of sharing an interrupt with another interrupt handler or if it expects interrupt delivery to be disabled upon

execution. To emulate Linux's interrupt handling, we maintain a global array per protection domain. Each entry in the array corresponds to an interrupt line number. We maintain a list of interrupt handlers per entry. Because interrupt arrays are local to each protection domain, we do no allow shared interrupts across protection domains. When an interrupt is received we call each interrupt handler from the list in the corresponding array entry until one of them signals that the interrupt was handled.

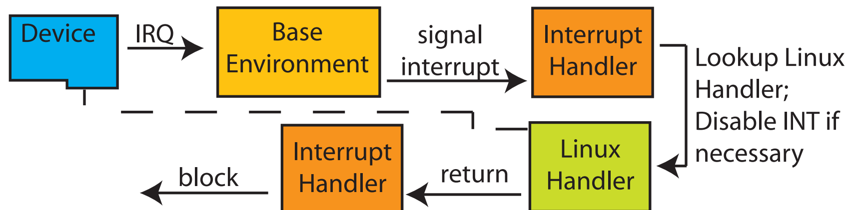

Figure 4.6: Linux Interrupt Handling Emulation

Emulation of the timer interrupt Instead of using the programmable interrupt controller to receive timer interrupts, we rely on the delay mechanism of the base environment. We dedicate a thread with interrupt priority to the timer interrupt. The thread periodically executes Linux's timer interrupt handler. The delay period equals the timer tick period that Linux would use. The timer interrupt handler performs the following tasks:

1. Update timekeeping
   (a) Update timer tick count
   (b) Update the time elapsed since system startup
   (c) Update time and date
2. Update timers

Linux abstracts timers from the underlying hardware, which allows us to reuse the timer code unmodified. However parts of the timekeeping architecture are highly hardware dependent, so we completely emulate these parts. The timekeeping architecture uses special timing objects. There is only one timing object instantiated per system during runtime. A timing object has to provide the following functions:

- Mark offset
- Get offset

Mark offset is invoked by the timer interrupt handler, it records the exact time of the last tick. Get offset, e.g. invoked by drivers, returns the time elapsed since the last tick. For our emulation purposes we construct and instantiate our own timing object. The mark offset function returns an increasing time value with microsecond granularity. We use the base environment to get that value. The get offset function of our timing object returns the number of microseconds

elapsed since the last call to mark offset. The rest of the timekeeping code, that update the different timekeeping variables is hardware independent so we reuse it unmodified.

Chapter 5

Implementation

As a proof of concept we implemented our approach on the x86 architecture with L4 as a hypervisor and Linux 2.6 as the donor OS.

5.1 Overview

We implemented a simplified version of our design for Linux network drivers (Figure 5.1). L4 runs as a privileged hypervisor in kernel mode. On top of L4 two dataspace managers manage physical memory and grub modules. We moved the I/O and PCI management into the same address space as the network device driver environment (DD/Env). The inclusion of the I/O and PCI management does not enhance performance because they are only used during initialization of a driver to setup proper access to resources. Especially they are not used on data transfer paths.

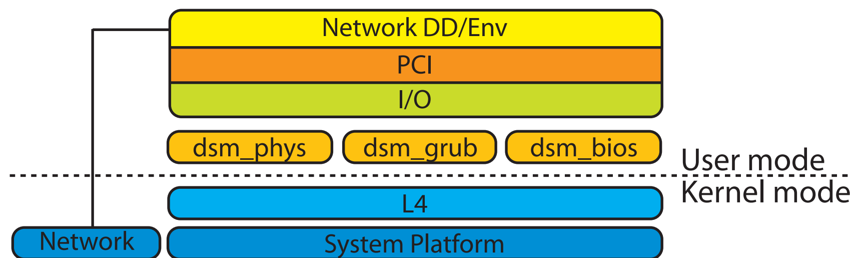

Figure 5.1: Implementation Overview
L4 runs as a privileged hypervisor in kernel mode. Three dataspace managers (DSM) provide access to physical memory, bios memory and to grub modules. I/O and PCI management are implemented in the same protection domain as the network device driver environment.

5.2 Base Environment

Our base environment consists of the L4 microkernel [41] as a privileged hypervisor in kernel-mode and several user-mode servers. L4 provides two abstractions, address spaces and threads, and two mechanisms, mappings and inter process communication (IPC). L4 mappings allow to map one or more pages that are mapped in the source address space into a region in the destination address space. Via these mappings the address spaces are constructed hierarchical starting with a source address space that contains all physical memory mappings. In addition to page mappings L4 also supports I/O page mappings. An I/O page describes a continuous region in the I/O space of the system. Like memory pages, I/O pages are mapped from a source space in which they are already mapped into a destination space. I/O spaces are also constructed hierarchical starting with one source space that contains all I/O space mappings after the start of the system.

If a thread accesses a memory page or an I/O page that is not mapped into its space the kernel generates a page fault or I/O page fault IPC. The fault IPC is send to the pager of the faulting thread. The pager of a thread is a thread in a different space that is specified during thread creation. After receiving the memory or I/O fault IPC, the pager is allowed to map memory or I/O pages anywhere in the faulting thread's space.

Software generated interrupts (exceptions) and hardware interrupts are also send by the kernel via special IPCs to dedicated threads. Each hardware interrupt number is represented by a special thread. An IPC to the thread enables the interrupt line that belongs to the interrupt number. In addition a thread registers with these threads to receive hardware interrupt requests (IRQ) from a device connected to the interrupt line. After reception of such an IRQ message, the interrupt line is disabled. It has to be enabled by the thread who received the IRQ message.

5.2.1 Virtual Memory

Our virtual memory framework is based on the dataspace concept [5]. Our dataspace implementation provides two entities: dataspace managers and region mappers. Dataspace managers implement data spaces. They make dataspaces accessible via the dataspace manager (DSM) interface 5.2. Our generic DSM interface consists of five operations open, close, pin, unpin and map. Open checks if the caller has the right to access the dataspace and returns a handle to the dataspace. The handles are local to each DSM. The map call requests pages of the dataspace that has been previously opened to be mapped into the callers address space. The pin operation pins a specific page to the callers address space and returns the physical address of the page. Accordingly by calling unpin the caller signals that a specific page is not needed to be pinned anymore. The close operation makes the handle invalid and unpins and unmaps all pages that have been previously mapped. Region mappers rely on the L4 pager concept. A pager is a thread that receives page faults caused by other threads in the system. The page faults are delivered by L4 via IPC. The pager then handles the page fault e.g. it responds via a mapping or it destroys the thread. Region mappers implement the following calls: pagefault, addRegion, removeRegion, virtToPhys and physToVirt. The pagefault operation follow L4's

fault protocol [34]. A thread uses addRegion to add a region of a dataspace to its address space. Addregion does not map any pages. Pages are mapped into the regions explicitly via the map operation or implicitly by the pager upon a page fault in the region. RemoveRegion removes a region from the caller's address space and unmaps all pages in the region. VirtToPhys and physToVirt translate virtual addresses into physical ones and reverse. A common operation

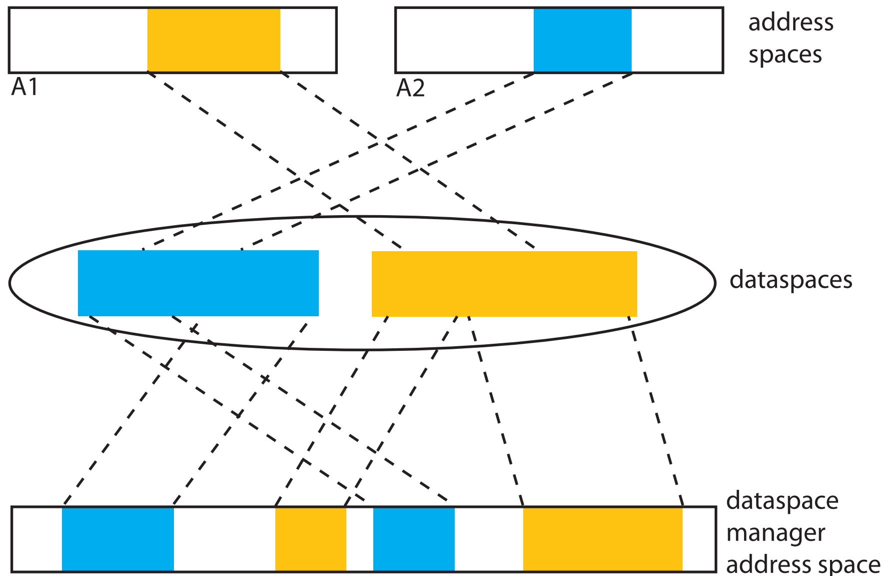

Figure 5.2: Dataspaces

for a driver is to share a buffer with an application and a device. With our dataspace implementation, the following operations are necessary to setup the shared buffer and use it. First the buffer has to be allocated. To do that, the driver opens a handle with the dataspace manager for physical memory. Then it calls addRegion with that handle. To share the buffer the driver maps it into the applications address space. To be able to do DMA operations on the buffer the driver pins the buffer and receives the physical addresses of the pages. In a last step, the driver programs the device to do DMA using the received physical addresses.

5.2.2 I/O space partitioning

We construct address spaces via virtual memory mappings. Accordingly we construct the I/O space of a space as a subset of the global system I/O space. We extended the functionality of our region mappers to deal with I/O pages. Threads can register and release access to regions of global I/O space with the region mapper. If an I/O fault occurs inside such a region, the region mapper maps the I/O page into the faulting thread's I/O space. Access to the I/O space is always exclusive in our environment.

5.2.3 Dataspaces

Our base environment manages four dataspaces, physical memory, device memory, grub module memory and bios memory. Physical and device memory are both managed by a single dataspace manager (physmem DSM). The physmem DSM uses a buddy allocator to efficiently keep track of free physical memory. The grub module dataspace contains all modules that were provided by the Grub [10] bootloader. These modules are all executable binaries in the ELF [51] format. The bios dataspace contains all pages that are related to the system's BIOS e.g. BIOS tables.

5.3 Device Driver Environment

A device driver environment consists of three software layers the emulation layer, the virtual machine layer and the driver layer (Figure 5.3). All layers are in the same address space. The emulation layer emulates the Linux kernel to drivers. The virtual machine layer implements an in-place virtual machine monitor based on pre-virtualization. The driver layer loads and executes the reused Linux driver module.

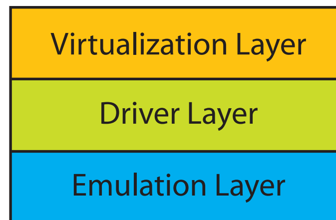

Figure 5.3: Software Layers in a Device Driver Environment

5.3.1 Emulation Layer

The emulation layer emulates all parts of the Linux kernel API, that are needed to run the driver.

Processes and Scheduling

We support only a very small subset of Linux's process model. We do not emulate the creation of additional processes because of the difficulties with address space creation and scheduling that would evolve. All device drivers we investigated do not use these functions. As stated earlier, Linux mixes up the notions process, task and thread. A thread has an associated `thread_info` and `task_struct` block. Linux uses the `thread_info` block to store architecture dependent thread data and the `task_struct` block to store architecture independent thread data. In the following we will call the combination of both structures the thread control block (TCB).

Threads We mirror Linux threads using L4 threads. Linux threads use stacks with a fixed size and a start address that is aligned to the size. The TCB is located at the bottom of a thread's stack. To access the TCB, Linux masks out bits of the stack address. The code for TCB access is inlined into the driver module. This inlining forces us to use the same stack layout as Linux.

Thread state transition To transition to a different thread state, a thread has to perform two operations. First it sets the thread state variable in the TCB to the desired thread state (RUNNING / INTERRUPTIBLE / UNINTERRUPTIBLE). In the second step it calls the `schedule()` function. The schedule function then determines the desired thread state from the TCB and puts the thread into that state. The states have the following meanings in the schedule function: RUNNING means that the thread freely gives up its remaining time slice (yield). INTERRUPTIBLE means that the thread wants to block until another thread wakes it up or it receives a signal. UNINTERRUPTIBLE means that the thread blocks until another thread wakes it up. In our emulation implementation we do not emulate signals because they are not used by our device drivers. Thus the INTERRUPTIBLE and UNINTERRUPTIBLE states are treated the same. Figure 5.4 shows the pseudo code of our `schedule` implementation. If the thread's state is set to RUNNING, we call `L4_Yield` which

```
switch(THREAD_STATE) {
  case RUNNING:
    L4_Yield()
  case INTERRUPTIBLE:
    WAIT_FOR_IPC
  case UNINTERRUPTIBLE:
    WAIT_FOR_IPC
}
```

Figure 5.4: Linux schedule() Emulation

voluntarily releases the rest of the thread's time slice. If the state is INTERRUPTIBLE or UNINTERRUPTIBLE, we block the thread by waiting for a wakeup IPC message. To wake up a thread, we send an IPC message to that thread.

Wait queues Whenever a thread needs to be blocked and queued in a data structure, Linux uses wait queues. Wait queues consist of a FIFO list of the waiting threads and of a spin-lock that protects the list. If a thread wants to be enqueued, it first acquires the spin-lock, adds itself to the list, releases the spin-lock and then blocks. When a thread wakes up the threads in the list, it grabs the spin-lock, removes and wakes up one or more threads from the list and releases the spin-lock after it is finished.

Synchronization

Linux synchronization mechanisms that are used by device drivers are spin-locks, semaphores, completions. Linux spin-locks are completely inlined into the code

via the C preprocessor. The inlining forces us to implement spin-locks in exactly the same way as Linux does. Spin-locks on uniprocessor systems don't acquire a lock, they simply switch off interrupts. On multiprocessor systems however they have a lock attached. A lock value of 1 means that the lock is acquired and a value of 0 that the lock is free to take. There are two versions of the acquire routine. The first version only tries to grab the lock one time and returns an error value indicating success or failure. The second version tries to grab the lock in a busy waiting loop. It only returns if it succeeds.

Semaphores are partially implemented inlined and and partially inside the Linux kernel. We have to emulate the kernel side. The inlined part of a semaphore first tries to acquire the semaphore. If it fails, it calls one of the kernel functions. The inlined code uses a special calling convention so the first thing that the fail-over code does is to convert the registers into the regular C calling convention. After that depending on the type of the acquire call, the caller is blocked (if the call was a blocking or interruptible call), or we return an error value that indicates the failure of the acquire operation.

Completions implement the scheme were one thread wants to wait for another thread to reach a certain point of execution. A completion is similar to a semaphore with an initial count of 0. If a thread calls the completion's wait routine it is blocked and queued in the completions wait queue. Another thread then wakes up dequeues the threads in the completions waiting queue, indicating that it reached the desired execution point.

Deferred Work

Linux supports three different types of deferred work, softirqs, tasklets and work queues. Softirqs are statically allocated and defined during compile time. They have assigned priorities and are limited to 32. We manage softirqs in a static array. They are executed by a dedicated thread (softirq thread). The softirq thread has a higher priority than regular code but a lower priority than interrupt handling code. To raise a softirq, we set the according flag in the softirq status word and send a message to the softirq thread. The softirq thread then checks the whole array for pending softirqs and executes them. In addition the softirq thread checks every second if any softirqs are pending.

There are two different types of tasklets, high priority and regular tasklets. They are implemented on top of softirqs. Because we emulate softirqs, we can reuse the complete tasklet code.

In contrast to softirqs, code deferred to a work queue runs at the same priority as regular code. For each instantiated work queue, we create a dedicated thread. If no work is pending, the thread blocks without a timeout and waits for new work to get enqueued. To enqueue work, a thread enqueues the work item atomically into the work queue and wakes up the work queue thread. The work queue thread then executed any items that are marked for immediate execution. If pending work queue item has an associated delay, the work queue thread sleeps for the delayed time and checks again when it wakes up.

Page Frames

Every physical page frame in the Linux kernel has an associated page descriptor. These descriptors are stored in an array starting at mem_map. The function

alloc_pages allocates one or more physically contiguous pages and returns a pointer to the descriptor of the page with the lowest address. Page descriptors are stored in the mem_map array in the same order as they lie in the physical address space (Figure 5.5). The page descriptor for the page frame with address

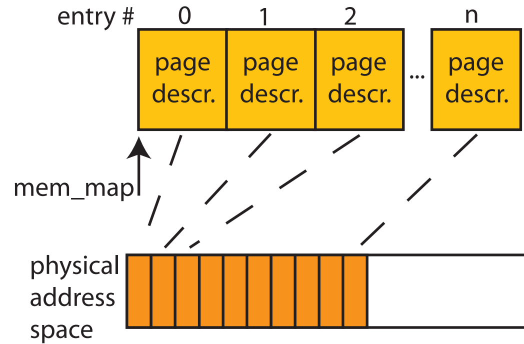

Figure 5.5: Correlation between mem_map and page frames

0x0 is in the first entry, the one with address 0x1000 in the second entry and so forth. The page frame number (pfn) for a given page descriptor is calculated via the formula pfn = (page_descriptor - mem_map). The mem_map array only allows to calculate the page descriptor from a given physical address. To get the page descriptor for a virtual address, the virtual address first has to be translated into a physical address.

5.3.2 Driver Layer

The driver layer's purpose is to load and run Linux device driver modules. A module is a binary file that conforms to the ELF [51] file format. It is relocatable and dynamic linkable. In addition to the standard ELF sections, a module contains two sections called `__ksymtab` and `__ksymtab_gpl`. These sections contain size and value of symbols that the module exports for use by other modules. To emulate Linux's symbol namespace we maintain a hash table with all symbols that have been exported. The loading of a module works as follows:

1. Copy the binary to memory
2. Find and resolve all undefined symbols
3. Copy and link all needed sections
4. Apply relocations
5. Export all symbols from the special export sections
6. Release the previously allocated memory
7. Initialize the module

If not all symbols can be resolved, the loading process is stopped. We maintain the symbol namespace per address space. Thus it is not possible to use exported

symbols from other address spaces. That means that modules that depend upon each other (e.g. the USB module stack) have to be loaded in the same address space. However such modules usually heavily interact with each other and thus putting them in different address spaces would cause severe performance overhead. Linux defines a special function that all modules have to provide called `init_module`. This function is called to initialized the module.

5.3.3 Virtual machine Layer

We use pre-virtualization [38] to virtualize the drivers. Our virtual machine monitor (VMM) only virtualizes interrupt delivery. The VMM adds another step to the module loading process. After the undefined symbols in the module are resolved and all sections have been relocated, it rewrites the pads that where generated in the pre-virtualization step. To help locating the pads, the binary has an additional section with the addresses of all pads. For each pad, the VMM first scans the code to determine the instruction. After the instruction has been identified, the VMM rewrites the binary with emulation code. The emulation code calls functions of the VMM instead of the privileged instructions. Figure 5.6 lists the privileged instructions that device drivers use and the corresponding emulation function. We pass through the IN and OUT instructions. We only

`IN/OUT`	none
`CLI`	`VMM_DisableInterrupts`
`STI`	`VMM_EnableInterrupts`
`PUSHF`	`VMM_Pushf`
`POPF`	`VMM_Popf`

Figure 5.6: Privileged Instructions and the Corresponding Emulation Functions

have to emulate the delivery of interrupts locally in the driver's address space. We use a mutual exclusion semaphore (mutex) for emulation. If a driver wants to disable interrupts we try to acquire the mutex. If the mutex is already taken, the thread is blocked and waits for a wakeup message. When the thread who has locked the mutex releases it, the blocked threads are woken up and can try to acquire the mutex again. However we do not expect that case to happen often because usually we have one device driver per address space which drives one device. More instances of a driver can be running in different address spaces, driving different devices but that does not affect local interrupt delivery.

The pushf instruction pushes the content of the EFLAGS register on the stack. The EFLAGS register contains a bit indicating whether interrupts are enabled or disabled. We set that bit according to the current state in the VMM.

The counterpart to the pushf instruction is the popf instruction. It pops the top word of the stack into the EFLAGS register. If the bit in the stack word indicates that interrupts should be enabled, we enable interrupts in the VMM again like a real CPU would.

The mutex has to be acquired by all instructions who read or write the interrupt flag. If e.g. the pushf emulation does not acquire the mutex there exists a race condition. The usual sequence to disable and enable interrupts is:

1. `PUSHF`

2. `CLI`
3. `POPF`

The pushf instruction stores the interrupt flag state on the stack, cli disables interrupts and popf restores the state that was stored on the stack. The following instruction sequence between two threads (A and B) leads to a race condition if the mutex is not also acquired by the pushf emulation: (A) disables interrupts via cli, (B) stores the state (interrupts disabled) via pushf, (A) enables interrupts, (B) disables interrupts, (B) executes popf which does not enable interrupts because the wrong state was stored before. To avoid that race condition we acquire the mutex if a thread executes pushf and when interrupts are disabled.

5.4 Networking

Our networking implementation is based on the Internet protocol suite [37]. The Internet protocol suite defines five layers (see Figure 5.7).

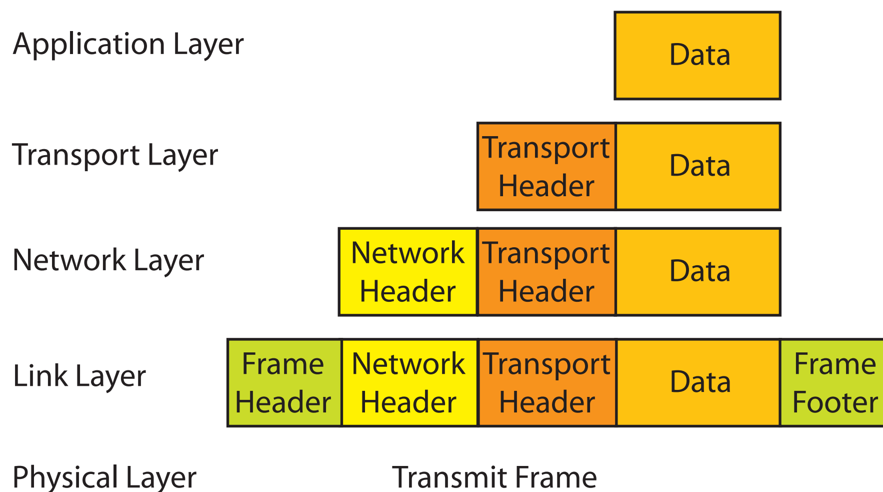

Figure 5.7: The Five Layers of the Internet Protocol Suite

Physical Layer The physical layer is the bottom most layer in the protocol suite. It encodes and transmits data over a physical network. It sends and receives data from devices that are physically connected with the host.

Data Link Layer The physical layer does not alter packets, it only transmits them. During transmission, the duty of the data link layer is to convert packets into a format usable by the physical layer. Usually it just adds a header to the packet. If a packet is received, the data link layer simply pushes it up to the next layer. The data link layer expects packets to have the correct size for use in

the physical network. Especially it does not split up packets with a size bigger than the maximum transfer unit of the network device.

Network Layer The purpose of the network layer is to transfer and receive packets within a logical network. A logical network spans one or more physical networks. It converts packets that are pushed down by the upper layers into one or more packets usable in the link layer (fragmentation). If packets are pushed up by the link layer during a receive operation, the network layer may hold packets that are fragmented until it is able to reassemble them.

Transport Layer The transport layer adds quality of service (QoS) features to the data transfer. Which features are supported depends on the actual protocol that is used. QoS features are for example congestion control, data transfer control, ordered data transfer, ...

Application Layer The application layer implements a session based management layer on connections. Optionally it includes a presentation layer that e.g. converts data into a specific format.

In our implementation we focused on the data link and physical layer. The reused driver together with the network device builds the physical layer. For the data link layer we implemented the Ethernet protocol [28].

5.4.1 Overview

Figure 5.8 presents an overview of our networking implementation. We use a

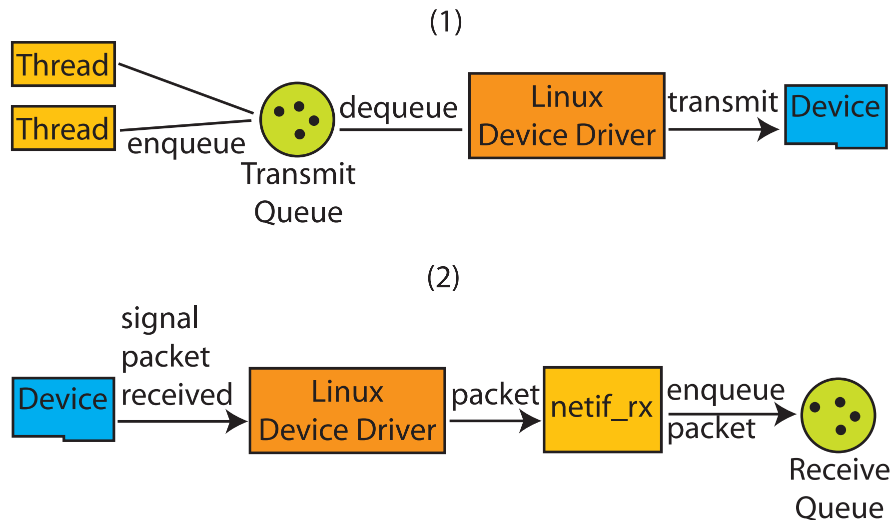

Figure 5.8: Linux Interrupt Handling Emulation
(1) Packet Transmission (2) Packet Reception

per device transmit and receive packet queue. In the transmit case, one or more

threads enqueue packets on the transmit queue. To transmit the packet, they call a function in the device driver that starts the transmission in the device. In the receive case, the device triggers an IRQ upon packet reception. The interrupt handler of the device driver calls a predefined receive function to deal with the new packet. The receive function queues the received packet in the receive queue.

5.4.2 Packet Representation

Linux network device drivers use the sk_buff data structure to represent network packets. Thus we have to use these packets in our network stack too to avoid additional data copies caused by packet conversions. Figure 5.9 shows the layout of a regular, not fragmented sk_buff. Data in an sk_buff is described by

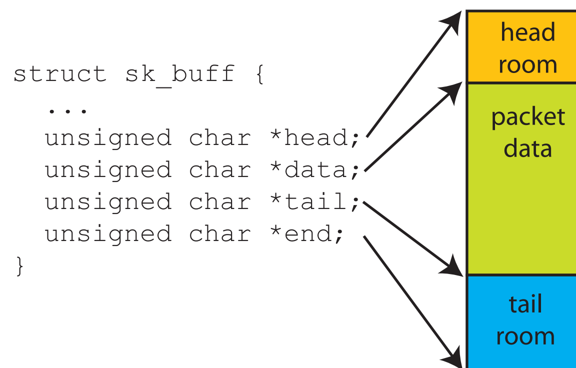

Figure 5.9: Linux Packet Buffer Layout

four pointers a head, data, tail and end pointer. The head pointer always points to the start of the data area. The data pointer points to the start of the packet data. The room between the head pointer and the data pointer is called the head room. It is reserved space for the various layer headers. The tail pointer determines the start of the tail room, the area that is available for packet data. The end pointer always points to the end of the data area. Initially the head, data and tail pointer all point to the beginning of the data area and the end pointer points to the end. Figure 5.10 shows operations on the packet buffer. Through all operations, the head and end pointer remain constant. The reserve operation operates on empty buffers. It reserves space for layer headers at the start of the buffer. It increases the headroom and decreases the available tail room. The put operations reserves space for packet data in the buffer. It leaves the data pointer but decreases the tail pointer. The space between data and tail pointer is now known to contain packet data. A push operation reserves space in the head room e.g. for the header of a layer. It increases the data pointer and leaves the tail pointer untouched. The size of a sk_buff is annotated in the len parameter. It is updated through the various operations. For non-fragmented buffers it always equals data - tail.

We use UDP in the transport layer, IPv4 in the network layer and Ethernet in the data link and physical layer. First `alloc_skb` is called to allocate and create

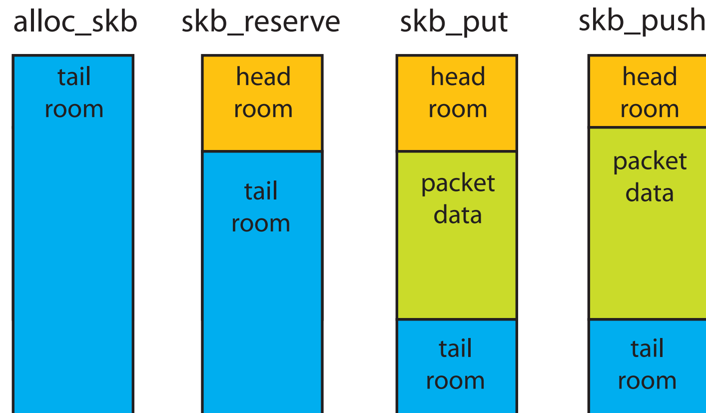

Figure 5.10: Linux Packet Buffer Operations

a new sk_buff. Next we reserve space for the UDP, IP and Ethernet header via `skb_reserve`. Now that we reserved the header space we can reserve space for packet data in the buffer. Packet data is reserved via `skb_put` which returns a pointer to the data area. We use that pointer to copy data into the buffer. Next we create an UDP header. We reserve space for the header via `skb_push` and write the necessary data. The next step is executed in the network layer. The network layer has to check if the size of the buffer exceeds the maximum transfer unit of the network device. If it does, the network layer has to split up the buffer into two buffers. We assume for simplicity reasons that the buffer meets the size requirements. We reserve space for the IP header via `skb_push` and fill in the header. The data link layer now adds the Ethernet header and calls queues the buffer into the transmit queue. If the thread that operates the transmit queue is blocked, the link layer wakes it up. The transmit thread then dequeues the packet and calls the `hard_start_xmit` function of the device driver. The `hard_start_xmit` function programs the device to transmit the buffer via the physical network.

Additional Variables In addition to the pointers that define the data of a packet, the sk_buff structure contains more variables. Table 5.1 outlines the variables that are important for our implementation. See [9] for a complete reference. The len field is automatically updated during data operations on the buffer. The other fields have to be set explicitly by the respective layers.

Fragmented Packets Fragmented packets in this paragraph means packets fragmented in the physical address space, not packet fragmentation as it is performed by the network layer. Modern network devices support a feature called scatter-gather I/O. Scatter-gather I/O describes the ability to gather multiple chunks of data that are scattered over the physical address space via DMA into one data chunk in the network device's memory. The data is then transmitted like every regular packet. Linux supports fragmented packets via the sk_buff data structure. A fragmented packet describes only the first part of the packet data via the operations described above. In addition it contains

Table 5.1: Important Variables of the sk_buff struct

h	Points to the header of the transport layer
nh	Points to the header of the network layer
mac	Points to the header of the data link layer
len	length of data in the packet
data_len	If non-zero data_len indicates that this packet is fragmented
ip_summed	Indicates what type of checksumming has to be done for the packet (software / hardware / none)

Table 5.2: Network packet fragment descriptor

page	Pointer to the page descriptor of the page frame that contains the fragment
page_offset	Offset to this fragment's data in the page frame
size	Size of the fragment's data

an array of fragment descriptors that describe the position of each fragment. Table 5.2 shows the fields of a fragment descriptor. The page descriptor is used to calculate the physical or virtual address of the page page frame. The position of the fragment's data is thus calculated with pos = get_address(page descriptor) + page_offset. If a packet is fragmented, the data_len field in the packet's buffer descriptor is non-zero. The data_len field contains the overall length of the packet data described by the packet's fragment descriptors.

Chapter 6

Evaluation

6.1 Test Overview

We compared our driver reuse environment against a Linux 2.6 system. The tests were performed on a Pentium 4 machine with 256 Mb RAM. The Intel Pentium 4 CPU we used has a clock speed of 1.5GHz, a 12Kb I-Cache, 8Kb D-Cache and a 256Kb unified L2 cache.

6.1.1 Test Environments

Linux For our Linux test systems we used Knoppix [45]. Knoppix is a Linux distribution that boots from a CD. Thus the tests were easily reproducible. We stopped all network unrelated services on the system to minimize the overhead caused by programs not related to our test.

Driver Reuse Our driver reuse environment used a current version of L4Ka:: Pistachio. L4Ka::Pistachio is the latest L4 microkernel that is developed at the University of Karlsruhe. On top of L4 runs our base environment and driver reuse environment. The driver environment reused drivers from Linux 2.6.9.

6.1.2 CPU Related Measuring

The Pentium 4 can be configured to count CPU related data using the time stamp counter (TSC). Measurable data are the number of clock cycles, the number of active clock cycles, the number of data- and instruction translation look-aside buffer (TLB) misses and the number of L2 cache misses. All numbers are set to 0 at system startup and updated with every clock cycle. To measure a specific value, we read the value at the start of a benchmark and read the value at the end of a benchmark. We subtracted the start from the end value to calculate the usage during the benchmark.

We calculated the CPU utilization by dividing the number of active clock cycles that elapsed during the benchmark through the elapsed number of overall clock cycles.

6.2 Software Engineering

6.2.1 Engineering Effort

We used the number of source lines of code as a metric for engineering effort. Figure 6.1 shows the overall, core related, network related and PCI related lines of code in Linux 2.6.19 and our driver reuse environment.

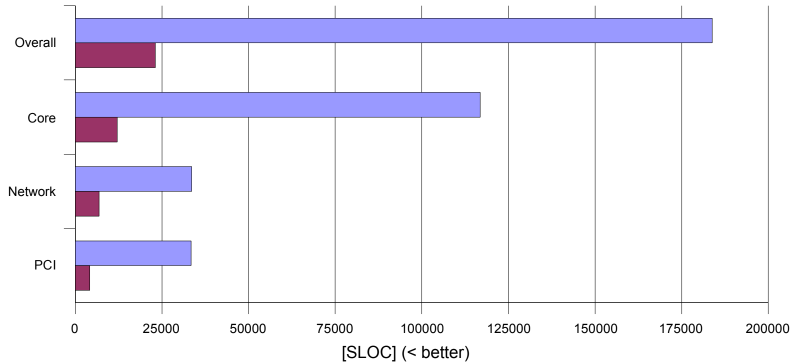

Figure 6.1: Source Lines of Code
Generated using David A. Wheelers 'SLOCCount'

We calculated the overall source lines of code from three subsets. The core subset contained all code for initialization and management tasks not related to a specific subsystem (e.g. memory management, scheduling). The network subset consisted of all code related to the data link layer such as device management and the Ethernet implementation. All code that manages the PCI subsystem belonged to the PCI subset.

The whole Linux kernel consists of roughly 5 million lines of code. For our calculation we only accounted the parts that are directly related to the three subsets. For example we left out all device driver and file system code. The overall code size of our reuse environment is 23420 source lines of code (SLOC). Compared to Linux's 183837 SLOC our code size is 8 times smaller which has two reasons. First we did not reuse Linux's initialization, memory management and scheduling code. All these tasks are easier to implement on top of a microkernel than on bare hardware and thus need less code. Second we only emulated Linux's network subsystem. The engineering effort to emulate a certain Linux subsystem is smaller than the overall Linux code size related to that subsystem. The core and PCI related code in our reuse environment account for 70% of the overall code size. These parts are independent of the network subsystem and can be reused for implementations of different subsystems. With only 23K SLOC for our reference implementation the engineering effort to implement our approach in a production environment is very low.

Completeness We measured the completeness of our implementation using the symbols that drivers import from the kernel (see Figure 6.2). To calculate the number of all imported symbols of a driver class we summed up all undefined

symbols in the class's modules. We identified the not implemented symbols by matching the symbols that our emulation environment exports with the symbols imported by the modules. The unmatched symbols were counted as undefined. Although we only implemented the functions needed by the tulip and

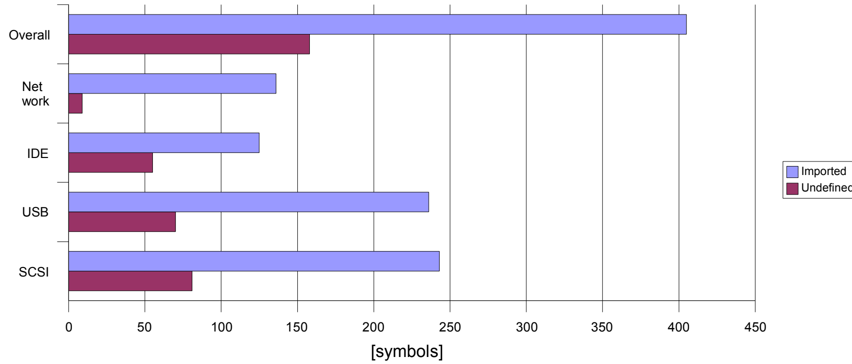

Figure 6.2: Symbols Imported by Linux Drivers
The overall size is the size of the superset of the symbols of all four driver classes. Some symbols are used by more than one class and thus the overall size is smaller than the sum of the size of all classes.

e1000 network driver, our implementation covers the network driver class almost completely. Because there are only few undefined symbols, the engineering effort to add new network drivers is close to zero. In addition we already emulate 60% of all symbols (247 out of 405) needed by all USB, IDE and SCSI drivers. Thus the engineering effort to implement the emulation operations for more driver classes will be lower than the effort needed for this first implementation.

Correctness Our design puts device drivers into multiple applications (decomposition). This decomposition eases debugging because bugs and errors are easier to match to a specific driver. In addition the small size of the decomposed software modules lowers the complexity of each module and thus lowers its bug rate.

6.2.2 Memory Footprint

We calculated the memory footprint of the Linux kernel and our driver reuse environment by adding up the memory size of all loadable sections in each binary (Figure 6.3). Due to its monolithic design the size of the Linux kernel is bigger than the size of our reuse environment. Our environment is ten times smaller than the Linux kernel. The small size significantly reduces the cache miss rate of our environment.

6.3 Transmission Test

In this section we present data gathered during a streaming send test. We measured the network throughput as well as the CPU utilization and CPU

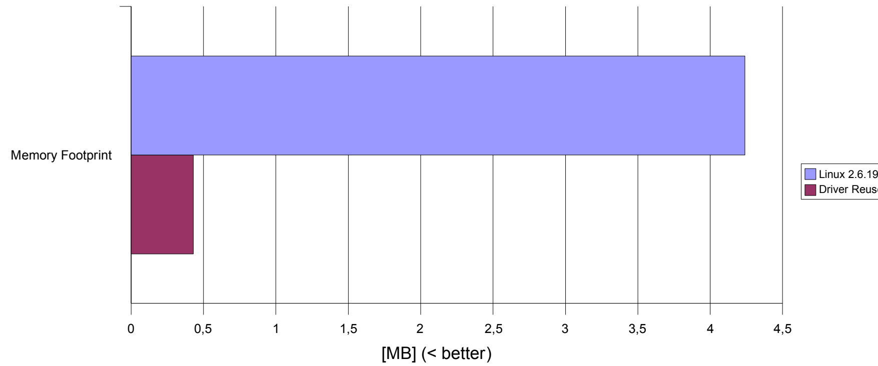

Figure 6.3: Memory Footprint of the Program Binaries

cache related data.

6.3.1 Overview

For our streaming send test we connected two systems (sender and receiver) directly through a Gigabit Ethernet network. In a contiguous loop, the sender sent packets over the network, and the receiver counted the amount of the received data and calculated the throughput. On the sender side we calculated the CPU utilization and cache miss rates. We used a Linux 2.6.19 system on the receiver side throughout the test. On the sender side we used the same Linux 2.6.19 system in a first test and our reuse environment in a second test. Afterwards we compared our results against the Linux environment.

6.3.2 Test Assembling

On both, the sender and the receiver side we used a network card based on Intel's e1000 chipset. Our sender sends UDP packets over the network. The packets have the maximum size that fits into one Ethernet frame to avoid side effects due to IP fragmentation.

On the receive side we used a 256Kb socket buffer to avoid packet loss. The receiver waited in the receiver loop (see Figure 6.4) until a predefined number of packets was received. The throughput was calculated by dividing the amount of data received through the time that was needed to receive the packets.

The Linux sender (see Figure 6.5) sent the packets through a socket. Each packet was allocated on the heap. Afterwards the sender wrote a signature into the packet and sent it through the socket. After the send operation completed, the packet was freed.

For the send test with our reuse environment we used a packet generator that was embedded into the driver environment (see Figure 6.6). After allocating the packet the generator added a signature and the UDP, IP and Ethernet headers. The generated packet was sent using the device driver's transmit function.

```
start timing;
WHILE (num_packets < max_packets) {
      receive next packet;
      num_packets++;
}
stop timing;
calculate throughput;
```

Figure 6.4: Streaming Send Test Receiver Loop

```
start performance counter;
WHILE (num_packets < max_packets) {
  allocate memory for packet;
  write signature into packet data;
  send packet through socket;
  free packet memory;
}
end performance counter;
```

Figure 6.5: Streaming Send Test Linux Sender Loop

```
start performance counter;
WHILE (num_packets < max_packets) {
  allocate memory for packet;
  write signature into packet data;
  add UDP, IP and Ethernet header;
  call driver's transmit function;
  free packet memory;
}
end performance counter;
```

Figure 6.6: Streaming Send Test Packet Generator Loop

6.3.3 Throughput

Figure 6.7 shows the network throughput of Linux and of our reuse implementation. Linux achieves 50.2Mb/s, our reuse environment 46.8Mb/s. The gap

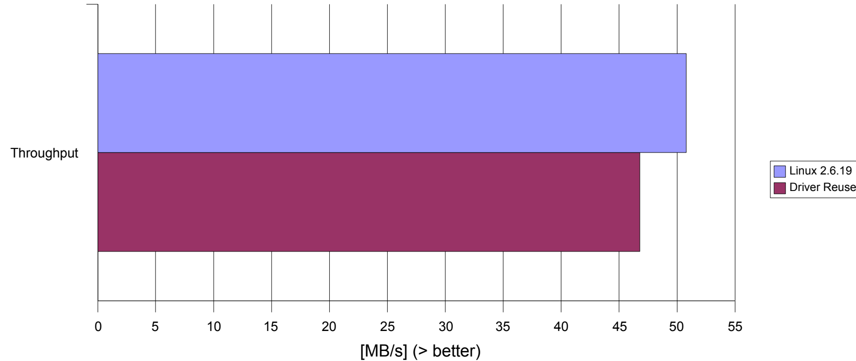

Figure 6.7: Streaming Send Test Network Throughput

comes from different queuing strategies. Linux adds an extra queue before the device driver's packet queue. The driver's packet queue is small and fills up fast. The extra queue is used to queue packets that can not be added to the driver's packet queue. Thus a thread may continue queuing packets even if the driver's queue is full.

Due to time constraints we were unable to add an extra queue in our implementation. If the driver's queue is full, we yield the CPU and wait for the queue to be available again. During this waiting process, we block and do not generate any new packets. Figure 6.8 shows the time needed for each operation in the packet generator loop when generating 100000 packets. We were spending a large fraction of the overall time blocked, waiting for the driver's queue to empty. An extra queue together with a special queuing thread would allow us to queue packets and continue execution. If we make sure that the extra queue is never empty, the driver's queue will be continuously filled by the queuing thread and thus the throughput would be higher.

6.3.4 CPU Utilization

An important indicator for the possibility of future optimizations of an implementation is the CPU utilization. Figure 6.9 shows the CPU utilization of Linux and of our reuse implementation during the send test. The CPU utilization of the Linux implementation (77.2%) is significantly higher than ours (25.1%). One reason are the different queuing strategies. Another reason however is the high L2 cache miss rate of Linux (see Figure 6.10).

6.3.5 Cache Usage

In this section, we discuss the L2, Instruction-Tlb and Data-Tlb miss rates of Linux's and our implementation.

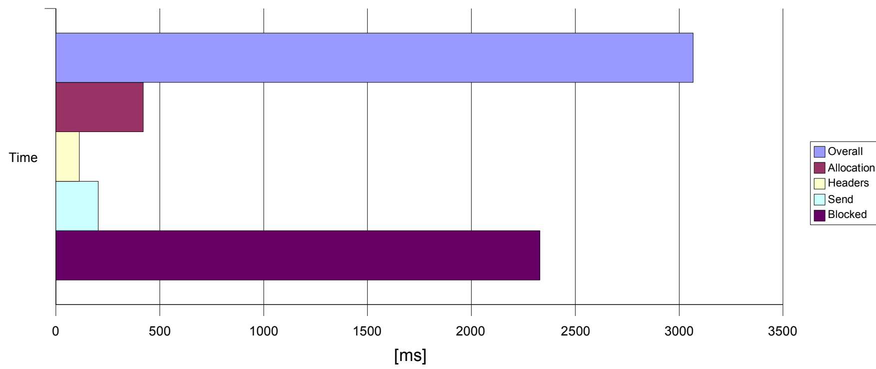

Figure 6.8: Time for Packet Generator Operations
Overall: Time needed for 100000 loops. Allocation: Time needed for packet allocation. Headers: Time needed to generate and write the UDP, IP and Ethernet header. Send: Time spent in device driver's transmit function. Blocked: Time spent waiting for driver's packet queue to be empty.

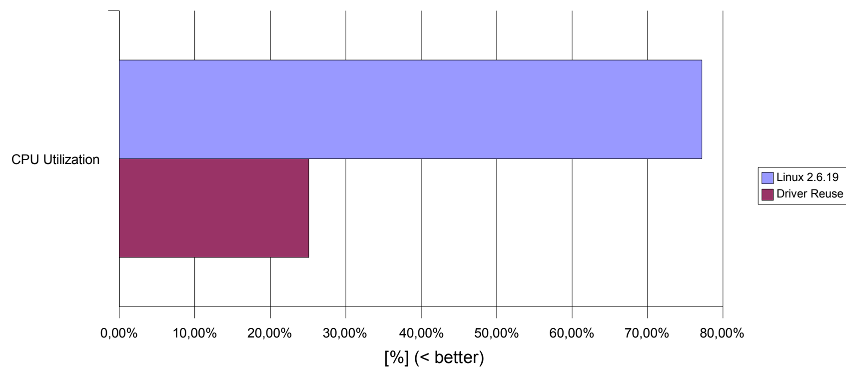

Figure 6.9: Streaming Send Test CPU Utilization

Figure 6.10 shows the number of L2 cache misses during the sender loop (100000 packets). Linux's L2 cache miss rate is eight times higher (2996151)

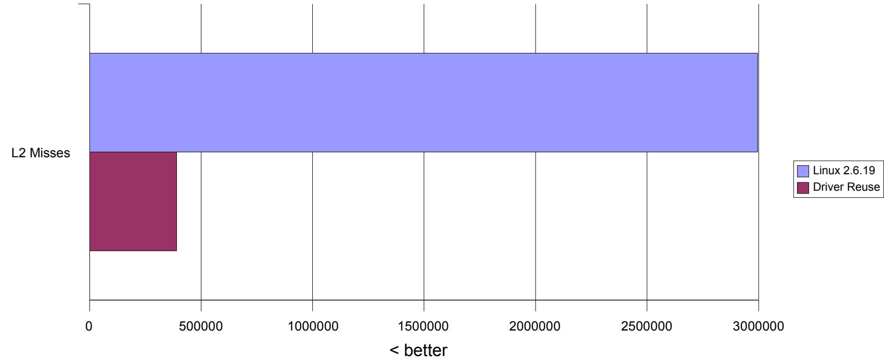

Figure 6.10: Streaming Send Test L2 Cache Misses

than the L2 cache miss rate of our implementation (390229). We had no time to investigate the runtime memory footprint of Linux but we suppose that it is higher than the one of our system because Linux is a full featured operating system and our implementation is only a test environment for research purposes. However, the test shows that L2 cache misses can have a significant performance impact.

Translation Lookaside Buffer In addition to the L2 cache miss rate we measured the miss rates of the instruction- and the data translation lookaside buffer (TLB) (see Figure 6.11). Our implementation has a higher miss rate on

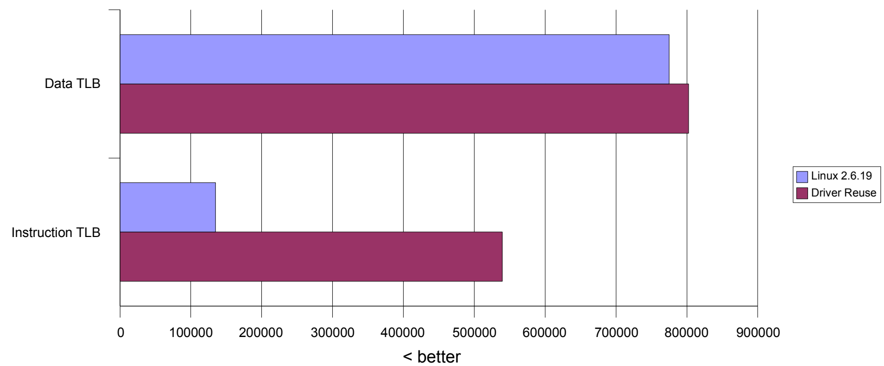

Figure 6.11: Streaming Send Test TLB Cache Misses

both caches. The higher rates are due to our design. We use user-mode device drivers. Every interrupt and exception may potentially lead to an address-space switch and thus to a TLB flush. However the low CPU utilization of our implementation shows that the higher TLB miss rate has only little or no

performance impact. We do not expect the performance impact to be higher on different CPUs because of the Pentium 4's Netburst [29] architecture that is known to have a high penalty for TLB flushes.

Chapter 7

Conclusion and Future Work

The goal of this thesis was to show that it is possible to reuse and isolate binary kernel-mode device drivers and that the reused drivers are as performant as in their native environment. The second goal was to reduce the engineering effort needed to reuse the device drivers and to show that the engineering effort to implement the emulation environment for the drivers is small. Furthermore we requested the design proposed in this thesis to be hypervisor independent.

No previous approach to device driver reuse and isolation managed to meet these requirements. Some approaches provide unmodified driver reuse but not isolation. Others provide isolation and unmodified driver reuse but fail to deliver good performance. Finally there are approaches that isolate device drivers but require high engineering effort.

This thesis introduces a new approach to device driver reuse that performs well and has a low engineering effort. Drivers are isolated by running them inside lightweight virtual machines in user-mode. Reuse of device drivers is done by emulating a kernel environment to the drivers. The proposed design uses a base environment to achieve hypervisor independence. Device driver environments depend on the interface of the base environment instead of a specific hypervisor interface.

The evaluation of our reference implementation proves that our unoptimized systems has a performance comparable to that of a Linux system. The conducted benchmarks show that due to the low CPU utilization of our system there is enough room for future optimization. We verified that the engineering effort to develop our approach or to add additional drivers is low by comparing the amount of source lines of code and the unimplemented symbols of our implementation.

We consider our work as a starting point for new operating systems and as a motivation for existing operating systems to move more device drivers into user-mode. We believe that we created an environment that is easily extensible to incorporate additional device driver classes. Although we have shown that our approach works well with a UDP implementation, it is important to prove that we can also achieve high throughput with a full featured TCP/IP stack which is known to have higher interrupt rates. In this thesis we focused on a

specific donor operating system. Although we see no further issues, future work should show that our approach is feasible for other donor operating systems as well.

Bibliography

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS*, San Jose, USA, October 2006. ACM.

[2] Dan Aloni. Cooperative linux. In *Linux Symposium*, Ottawa, Canada, July 2004.

[3] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. VMI: An interface for paravirtualization. In *Proc. of the Linux Symposium*, pages 363–378, Ottawa, Canada, July 2006.

[4] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, and M. Ostrowski. Utilizing linux kernel components in K42. Technical report, IBM Watson Research, August 2002.

[5] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 164–177, Bolton Landing, NY, October 19–22 2003.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, Anaheim, USA, April 2005.

[8] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*, pages 87–98, 1994. Available from `citeseer.ist.psu.edu/bonwick94slab.html`.

[9] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel 3rd Edition*. O'Reilly, 2005.

[10] Pdraig Brady. Details of Grub on the PC, March 2007. Available from `http://www.pixelbeat.org/docs/disk/`.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

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

[13] Jonathan Corbet, Allesandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers 3rd Edition*. O'Reilly, 2005.

[14] Microsoft Corp. Architecture of the [Windows] User-Mode Driver Framework. Technical report, Microsoft Corp., 2007. Available at `http://www.microsoft.com/whdc/DevTools/ddk/default.mspx`.

[15] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the Second Symbosium on Operating Systems Design and Implementation*, Seattle, USA, October 1996.

[16] Ltd. Eagle Rock Aliance. Cost of Downtime Study. Available from `http://www.contingencyplanningresearch.com/cod.htm`.

[17] 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 7–9 2004.

[18] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint-Malot, France, October 1997.

[19] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and Execution Models in the Fluke Kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, USA, February 1999.

[20] A. Forin, D. Golub, and B. Bershad. An i/o system for mach 3.0. In *Proc. of the Second USENIX Mach Symposium*, Monterey, CA, November 1991.

[21] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, October 2004.

[22] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding Denmark, September17–20 2000.

[23] S. Goel and D. Duchamp. Linux device driver emulation in mach. In *USENIX Annual Technical Conference*, San Diego, USA, January 1996.

[24] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 66–77, Saint Malo, France, October 5–8 1997.

[25] Christian Helmuth. Generische Portierung von Linux-Geratetreibern auf die DROPS-Architektur. Master's thesis, TU Dresden, July 2001.

[26] Hewlett-Packard Corporation and Intel Corporation and Microsoft Corporation and Phoenix Technologies Ltd. and Toshiba Corporation. *Advanced Configuration and Power Interface Specification Revision 3.0b*, October 10 2006.

[27] H. Hartig, J. Loser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An i/o architecture for microkernel-based operating systems. Technical Report TUD-FI03-08-Juli-2003, TU Dresden, July 2003.

[28] *IEEE 802.3-2005: IEEE Standard for Information technology Telecommunications and information exchange between systems Local and metropolitan area networks Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.*

[29] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, March 2006.

[30] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, March 2006.

[31] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide Part 1*, March 2006.

[32] Linus Peter Kamb. Extending Fluke IPC for Transparent Remote Communication. Master's thesis, University of Utah, December 1998.

[33] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a real operating system. In *Proceedings of Eurosys'2006*, Bruessels, Belgium, April 2006.

[34] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2 Rev 6).* Universität Karlsruhe, 2006. Available from http://l4ka.org/.

[35] Kevin Lawton. The bochs IA-32 emulator project. Available from `http://bochs.sourceforge.net/`.

[36] Geoffrey Lee. I/O kit drivers for L4. Undergraduate Thesis, University of New South Wales, November 2005.

[37] B. Leiner, R. Cole, J. Postel, and D. Mills. The DARPA Internet protocol suite. *IEEE Communications Magazine*, 23(3):29–34, March 1985.

[38] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-Virtualization: Slashing the Cost of Virtualization. Technical Report 2005-30, University of Karlsruhe, Germany and IBM T. J. Watson Research Center, New York and National ICT Australia and University of New South Wales, Australia, November 2005.

[39] 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 Systems Design and Implementation*, San Fransisco, CA, December 6–8 2004.

[40] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a microkernel based OS. *acmosr*, 25(2), April 1991.

[41] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, December 3–6 1995.

[42] Linux. `http://www.kernel.org`.

[43] Mach. Available from `http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html`.

[44] K. V. Maren. The Fluke device driver framework. Master's thesis, University of Utah, December 1999.

[45] Kyle Rankin. *Knoppix Pocket Reference*. O'Reilly Media, 2005.

[46] D. S. Ritchie and G.W. Neufeld. User level ipc and device management in the raven kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, San Diego, USA, September 1993.

[47] Rusty Russel. Paravirt-ops, 2007. Available from http://ozlabs.org/ rusty/-paravirt/.

[48] J. Sugerman, G. Venkitchalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USEXNIX Annual Technical Conference*, Boston, MA, June 2001.

[49] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, Bolton Landing, USA, October 2003.

[50] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[51] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification V1.2*, May 1995.

[52] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, USA, June 2002.

[53] Albert S. Woodhull and Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 2006.