Bachelorarbeit

Porting OpenBSD to Fiasco

Christian Ludwig

June 22, 2011

Technische Universität Berlin
Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Professur Security in Telecommunications

Betreuender Hochschullehrer: Prof. Dr. Jean-Pierre Seifert
Betreuender Mitarbeiter: Dipl.-Inf. Michael Peter

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Berlin, den 22. Juni 2011

Christian Ludwig

Acknowledgements

After having had my own style of going through university for a degree, with this thesis I am finally there. I want to thank everyone involved in making this thesis possible. First of all, there is Marlene, which had to suffer my way to study for a long time. Nonetheless, she always supported me and encouraged me to go on at all times. Michael Peter deserves great respect for his knowledge and wisdom on the internals of computers and CPUs in special. He never gave up explaining these things again and again to me. I also want to thank all companies, which believe in our solution and push it into the market. Last but not least, there are my parents. You know that I am going my way.

Zusammenfassung

Computersysteme findet man an jedem Punkt des modernen Lebens an. Sie werden für verschiedene Anwendungen genutzt, darunter E-Mail, Spiele und das Browsen im Web. Diese Systeme werden aber auch in wichtigen Bereichen eingesetzt, in denen Missbrauch zu schweren Schäden führen kann. Das Entwenden von Online-Banking Daten kann den Besitzer teuer zu stehen kommen. Was wir also brauchen sind vertrauenswürdige Geräte.

Angriffe auf Computersysteme werden meistens über die darauf laufenden Anwendungen durchgeführt. Aber erst Exploits, durch die man erhöhte Privilegien erlangt, weil sie zusätzlich das Betriebssystem angreifen, sind profitabel. Daher ist das Betriebssystem Teil der vertrauenswürdigen Kette.

Das eingesetzte Sicherheitsmodell vieler Betriebssysteme ist diesen Angriffen in den meisten Fällen schutzlos ausgeliefert. Es basiert auf Annahmen, die aus den Anfangstagen der Betriebssystem-Programmierung stammen und beinhaltet nicht die Strategie nur die geringsten Rechte zu nutzen. Zu dieser Zeit gab es andere Voraussetzungen.

Nun kann man die notwendigen Änderungen in bestehende Systeme integrieren, um die Vertrauenswürdigkeit des Betriebssystems zu erhöhen. Dieser Ansatz klärt jedoch nur die Frage nach der weiteren Einschränkung von Nutzerprogrammen. Er klärt nicht die Frage nach Einschränkungen, die im Kern selbst notwendig sind.

Daher ist eine neue Sicherheitsarchitektur notwendig, die nur kleine Schnittstellen bietet und nur kleine Komponenten nutzt. Mit dieser kann man auch weite Teile des Kerns selbst absichern, jedoch werden sich die Schnittstellen zu den Anwendungen ändern. Es müssen also alle bestehenden Anwendungen auf die neue Schnittstelle portiert werden, was unmöglich ist.

Um das Portieren der Anwendungen zu umgehen, kann man die bestehenden Anwendungs-Schnittstellen auf die neue Sicherheitsarchitektur bringen. Diese Maßnahme ist langwierig und fehleranfällig. Daher entscheidet man sich für einen zweiten Ansatz, das komplette Standardbetriebssystem in der neuen Sicherheitsarchitektur als Einheit zu kapseln. Aus diesem Standardbetriebssystem werden dann nur die wenigen hochkritischen Teile entfernt, die ihrerseits gekapselt ausgeführt werden müssen.

Die Kapselung kann auf modernen CPUs mit Hardwareunterstützung erfolgen. Hierbei ist der Portierungsaufwand gering. Enthält die CPU keine Unterstützung zur Virtualisierung, muss das Standardbetriebssystem angepasst und auf die Sicherheitsarchitektur portiert werden. Der Aufwand ist abhängig von der Unterstützung der Ziel-Architektur. Je mehr sich die Ziel-Architektur wie eine physische CPU verhält, um so geringer ist der Portierungsaufwand.

Diese Bachelor-Arbeit beschäftigt sich mit dem Design und der Implementierung des Umsetzens des Standardbetriebssystems OpenBSD auf den Fiasco Mikrokern als Sicherheitsarchitektur. Dabei ist zu beachten, dass die ausführende CPU Kapselung hardwareseitig nicht unterstützt. Fiasco ist ein moderner Mikrokern, der mit vCPU ein Feature hat, das ereignisgesteuert den ausführenden Thread unterbrechen kann und stattdessen eine Behandlungsroutine ausführt. Der unterbrochene Thread kann anschließend fortgeführt werden. Damit verhält sich die Schnittstelle ähnlich einer physischen CPU.

Zuvor gehen wir jedoch auf die Herkunft von Mikrokernen ein. Am Ende sehen wir noch eine Evaluation der erreichten Implementierung.

Contents

List of Figures

1 Introduction

After decades of rapid innovation, computers have become powerful commodities that open up a wide range of activities. Life without email, social networks or other information services is hardly imaginable. The adoption of computer systems is widening as new form factors, such as tablets and smartphones, become available.

But these systems are also used for applications, where misuse leads to severe damage. When an online banking account gets hijacked, the owner of that account risks loosing real money. Also, disclosing private information due to a virus infection may lead to a lack of confidence for the affected party. The user expects devices to handle data in a trustworthy manner, an expectation that often enough is not met nowadays.

Applications are inherently complex, so only little can be done at this stage. However, overall system security can be strengthened with a trustworthy operating system which enforces isolation. Although, most attackers go for vulnerabilities in applications, only deficiencies in the operating system allows adversaries to widen the scope of their attack. A compromised application becomes an entry vector to attack the whole system. In a way applications are used as trampoline to exploit the operating system. Fixing applications is not a long-term solution to that problem. Applications should never be able to compromise their operating systems. That makes the operating system a crucial component in the system architecture.

The reason for the deficiencies in security of many popular operating systems lies in their provenance. Many of them can be traced back to systems developed in the 1970ies. Back then, the attack profile was different. The operating system only had to ensure that users do not interfere with each other. However, today the classical notion of a user who only runs trustworthy software is gone. But the isolation mechanisms still remain and are not up to the new challenges. Running each application instance in the context of a dedicated user is not feasible.

1.1 Backward-compatible Security Enhancements

Changes to the security mechanisms are difficult, because they are linked to functionality in the system call interface. For example, file access permissions are tied to file system objects. Changes in the process identity would therefore render parts of the filesystem inaccessible.

One idea to fix the lack of trustworthiness is to retrofit the security architecture. The Linux security modules framework, for example, consults a kernel module whenever the operating system is about to grant access to an important internal kernel object, such as an inode or a task control block. The policy itself is derived from kernel policy modules. Security-Enhanced Linux (SELinux) ([LS01]), the first and most commonly

known module has been joined by AppArmor, Smack and Tomoyo Linux. These security enhancements to the Linux kernel implement a mandatory access control scheme, imposes rules on users and applications which cannot be easily circumvented. However, the approach remains backward-compatible with the large application base.

These approaches are able to restrict access for userland applications, but they do not give answers to access control within the kernel itself. Most operating systems allow loading additional drivers. When we have hardware devices with drivers provided by the vendor, how trustworthy are these? Do they crash the kernel and in turn stop the whole system from functioning? Additionally, SELinux introduces a huge system call interface and a complex configuration with a default set of 50000+ lines of policies.

1.2 Security Architecture

Monolithic kernels are so complex that it is difficult to assess their correctness. The problem was more manageable with small interfaces and components of low complexity. Although most of the components are still be needed. In that way, we can have encapsulated drivers and deploy a policy of least authority across the system.

For such a system, we will most likely need to change the exposed kernel interface for applications. That bears a new problem. Since most applications were developed against existing operating system interfaces, we will need to port all applications to the new kernel interface. That is infeasible.

Instead of rewriting all existing applications, we can provide a backward-compatible interface. SawMill ([GJP+00]) and the Hurd ([WB07]) provide a POSIX compatibility layer in the standard C library for applications. They also separate the kernel into distinct tasks, which leads to a higher robustness against faulty modules. The benefits of that architecture are implicitly used by all applications. For full backward-compatibility this approach is cumbersome and error prone.

1.3 Secure Operating System Reuse

In most cases it is sufficient to protect only a tiny set of applications. Think of digital signatures as an example. The user can continue to use existing applications to create his documents. That process is complex and the attacker might even be in control already. If we were not for further steps, the attacker can manipulate the document at will and influence the signing process. He may alter the target of a bank transaction request. Since the document carries the user's signature, he will be held accountable for its contents.

In a security architecture the document would be created as in the outlined scenario. The signing process, though, would be out of the attacker's reach. The user can validate the data in a tiny, isolated application. This application also signs the data, before it can be transmitted to the bank. That shows we can encapsulate the whole legacy operting system as a single entity in the security architecture. Only a small application set needs to be isolated.

There are different ways to encapsulate an operating system to fit into the security architecture. If there is hardware support for virtualization, an efficient encapsulation of operating systems is possible. The hardware provides a virtual interface, which behaves very similar to the original one. The porting efforts are minimal. The guest operating system proceeds as normal.

When the hardware does not provide virtualization support, we need to port the operating system on a standard CPU. We need to deprivilege the operating system kernel. It needs to be changed from a hardware interface to the interface of the secure architecture. The detailed porting efforts are dependent on the target interface. The more it behaves like a physical CPU, the easier it is to port the legacy operating system.

1.4 Outline

This thesis describes the rehosting efforts of the OpenBSD operating system on the Fiasco microkernel.

At first, we go through the history of operating system kernels and approach the microkernel design. We will focus on Fiasco, a member of the L4 family of microkernels. After that an introduction to virtualization is given with the aim to use Fiasco as a hypervisor.

Thereafter we will have a look into the design and implementation of the OpenBSD kernel and userland rehosted on the Fiasco microkernel. The focus is on memory management and the differences in the execution model. In addition to that, we examine the drivers needed for the effort. Afterwards the implemented solution is evaluated for code and performance.

At the end we will compare the solution in this thesis to other related work in the field, before we conclude with an outlook on possible enhancements to the system.

2 Background

In this chapter we are going to give an introduction to operating systems in general and see how they adopted monolithic kernels over time. We will contrast these to a microkernel approach, with a discussion of the L4 family of microkernels. Fiasco will serve as a representative of them and show its features. We will also discuss virtualization as another technology that promises to enforce stronger isolation.

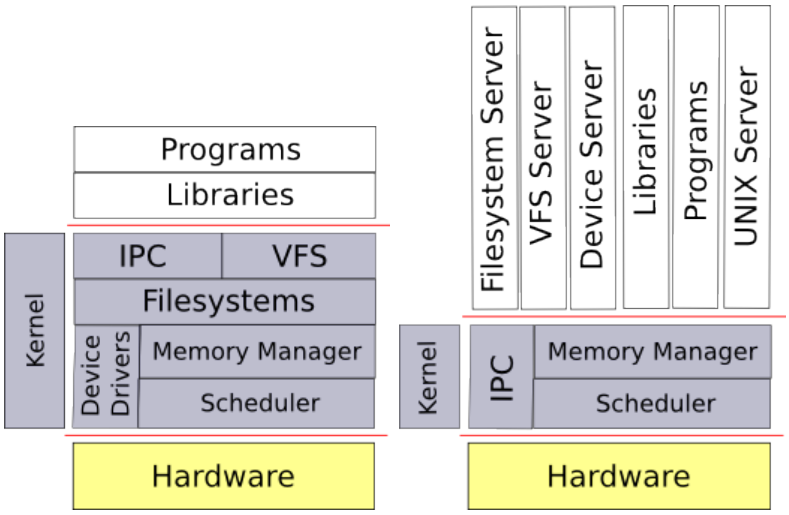
2.1 Operating Systems

Modern operating systems are capable of running multiple programs in a multi-tasking environment. The operating system needs to take care that all programs are protected from each other. All modern architectures support that idea and provide a separate privileged mode to the default runtime mode. An operating system runs in that privileged mode and can reconfigure system resources. There is also a hardware feature, called paging, which allows each program to have its own virtual address space. Configuring and switching between address spaces can only be performed in privileged mode. That gives us a basic protection mechanism. There is typically one program in control of the machine and its mode of operation, thus running in privileged mode. That program is called the operating system kernel. The kernel separates different programs with the use of address spaces from each other. However, programs in usermode can use kernel services for operations, which need to access shared resources. The kernel needs to provide an interface for these system calls. As a result all usermode programs need to trust the kernel.

2.2 Monolithic Kernels

Most popular operating systems today use a monolithic kernel. A schematic design can be seen in figure 2.1(a). All mechanisms needed to service user program requests are implemented in kernel space. In fact, beyond the needed services, most mechanisms implemented in the kernel are for the convenience of userspace applications.

Over time many features creeped into the kernel, like device drivers, protocol stacks or filesystems. With more and more features added, kernels got huge and complex. People tried to manage the complexity by grouping belonging parts together in subsystems. But the growing complexity puts the stability of the whole system at risk. Since there is no higher authority than the kernel in the system, nothing protects the kernel from itself. Especially since there is no separation in the kernel. Separation between subsystems is achieved only by convention. It is not enforced. Subsystems can interact with each other in subtle ways. Even if they normally do not, a misbehaving driver may write into



(a) Schematic representation of a monolithic kernel.

(b) Schematic representation of a microkernel.

Figure 2.1: Design of a monolithic kernel vs. a microkernel. Many subsystems of a monolithic kernel are implemented in userspace on a microkernel.

arbitrary parts of the kernel, thus changing vital data structures. If the kernel crashes, the whole system stalls. We can see that most of these features do not necessarily need to run in kernelmode, as they do not need the privileges to configure the system.

Access control policies for resources, for example, are based on access control lists (ACLs) in most cases. These ACL systems are highly inappropriate to implement a system where each entity has the least privileges they need to operate ([MYSI03]). On an ACL-based system revoking an entity's access from the system results in checking and setting all ACLs. With a capabilities system, that procedure would mean to only revoke a capability[1]. In addition, the implemented security model spans multiple subsystems. The huge size of the kernel makes it infeasible to change that model.

Due to their complexity, execution paths in monolithic kernels are rather long. That means it is fairly tough to predict temporal execution behavior. In long paths through the kernel, there are many objects involved. Therefore monolithic kernels have to take measures to ensure the integrity of their kernel objects by synchronizing access to them. One measure taken is to only handle one call into the kernel at a time. If a user-mode application issues a system call, the kernel is locked for other programs and thus non-preemptible. Synchronization within the kernel is achieved with locks for data structures. These ensure atomic execution of longer instruction paths. Locks in turn are implemented with atomic instructions. Some of these code paths even require to turn off IRQ delivery completely.

Operating systems implemented with monolithic kernels mostly have some serious limitations regarding their flexibility. They suffer from implicit policies embedded in

[1] Which really means to make any invocation bound to fail.

subsystems, e. g. a non-optimal memory allocation strategy or an unfitting filesystem prefetch algorithm for database applications. Database applications normally prefer raw disk access to implement their own filesystem strategy. Most systems also suffer scheduling problems under certain loads. The SPIN operating system tries to address that issue ([BCE+95]). It allows applications to bring their own kernel components. These components can register themselves to attach to events in the SPIN kernel. They can implement an application-specific policy on how resources are scheduled in a process group. The SPIN kernel checks the application-specific kernel components for sanity with static analysis.

The OpenBSD operating system for example implements a monolithic kernel. It also suffers from the problems mentioned above. In the next section we will have a look at an alternative approach to the one taken by OpenBSD. After that, we will have a look on how to combine these two different approaches.

2.3 Microkernels

We can see that monolithic kernels are too large to handle. To understand which parts are to be kept in a minimal kernel, we need to understand that all operating system kernels basically try to solve the following four different problems ([REH07]).

1. Abstraction
2. Resource sharing
3. Protection
4. Message passing

Abstraction denominates the way in which resources are presented to upper layers. So if we strive to offload subsystems from a kernel, we need to put them into userland. That raises the question which objects a kernel shall expose to the remaining system. We will see the answer below, when we have a closer look into the different microkernel generations.

There are inherent resources like CPU and RAM, which every program needs. All other resources can be assigned to programs explicitly, but they are all implemented using the inherent resources. The policy decision about which resource is granted to a program shall not be taken by the kernel. It can be implemented as a userspace program. Whereas the mechanism to enforce a policy must be provided by the kernel as the highest authority in the system. Userland applications are unable to enforce policies on other programs. But changing a userland program is easy, thus changing policy decisions at runtime is easy, too. There can even be different userland programs implementing different strategies for the same policy object.

A microkernel achieves better protection by offloading most of its subsystems as small userland programs, each in its own address space. In this way we can reduce the trusted computing base (TCB) for a program. The TCB counts all components, which a program relies on to perform its task. All programs rely on the kernel to operate

correctly. If the kernel is small, the TCB gets clearer and can be tailored. A program's TCB does not automatically contain all kernel subsystem anymore, instead it only contains specific offloaded userland programs.

Like the resource sharing mechanism, message passing in the kernel shall also only follow a userland policy. It should only implement the mechanism. If a program requests a memory page, for example, it should ask another program to provide such a page by sending a request message. The kernel should only pass the message to the other program. The kernel does not even need to be aware of the message type. It does not care, if the message is a request or response.

The issues described in the last section, show that monolithic operating systems may not be flexible enough for specific tasks and workloads. An idea on how to address these issues, especially the isolation of different subsystems and their ample privileges, was proposed by Hansen in [Han70] back in the 1970ies. The idea is to strip the kernel to its bare minimum of memory protection, message passing and CPU scheduling. The resulting class of operating system kernels is called microkernels. A schematic design of a microkernel can be found in figure 2.1(b). Subsystems of a monolithic kernel, which are not in need to secure the system are not implemented in a microkernel. Instead, they are implemented as isolated userspace applications with a well-defined API, which cannot be circumvented. That makes a microkernel significantly smaller. Code complexity also remains manageable. Therefore it is more likely to succeed in formally verifying a microkernel than a monolithic one.

Since kernel subsystems were then mostly implemented as operating system servers in userspace, servicing user requests always needs inter-process communication calls between the user program and the server, as well as a full context switch to the server task. The microkernel needs to copy the IPC payload between the two parties. Then the server performs its operations and returns the result again via IPC mechanisms and data copies to the calling program. After another context switch, the program is finally able to resume its operation. These additional transitions between different contexts lead to higher execution costs.

2.3.1 First Generation Microkernels

First generation microkernels, like the Mach microkernel, suffered from performance hits inherent in their design. Its original design was derived from the monolithic UNIX kernel. Slicing off subsystems from an existing kernel is not easy. So the resulting microkernel retained some functionality, which was problematic. For example, it still had a kernel swapping mechanism ([GD91]). IPC operations were implemented as asynchronous messages. With such a design, the kernel needs to allocate buffers in critical IPC paths. IPC messages could fail when the system was out of memory. It also had to check for a number of corner cases. There was a bad balance of functionality and performance for IPC operations. That has also led to a large cache footprint.

2.3.2 The L4 Microkernel Family

After the sobering experiences with first generation microkernels, it became common wisdom that it is not possible to implement microkernels in an efficient way. In fact, Mach did not perform well for IPC messages sent between processes. Jochen Liedtke showed an advanced view on microkernel design ([Lie93]). That principle-driven design exploits everything the underlying hardware could possibly achieve, on modern architectures. It assesses the theoretical minimum design and implements it.

All parts of the kernel should be designed to perform very fast IPC operations. Every heavy-used code path should be tuned. A few ideas involved rethinking the way data is copied between address spaces. For example, when exploiting the memory management unit (MMU), we can have 1-copy transfers even without shared memory. Aligning critical data structures to CPU caches is another way to improve the overall speed of the microkernel. Therefore, microkernels are not portable between CPU architectures when these techniques are used on a large scale. Newer versions of Mach also improved IPC performance using similar techniques ([GJR+92]).

Liedke proposed a minimal set of kernel objects for a microkernel, which are still sufficient to build a general system on top of it ([Lie95]). He implemented these in L3. The successor, called L4, is the basis for current third generation microkernels, which consist of the following basic abstractions.

Task A tasks is a separate address spaces, which can be backed with virtual memory. It is used for separation. Tasks cannot access each other's memory regions, unless specifically granted.

Threads Threads are units of execution on a CPU. They can be placed into a task and execute code mapped into a data space. There can be multiple threads per task.

IPC Interprocess communication (IPC) is used to send and receive data between tasks. The communication is synchronous in nature so that the kernel does not need to implement any message buffers. Message buffers in asynchronous IPC raise the question of memory allocations and queuing.

A major improvement in L4 was to unify memory management with IPC semantics. That gives us user-level memory management. A thread can send pages from its own task to another thread in a different task. That thread, in turn, can propagate the page further, so that we get recursively built address spaces. Commonly there is the σ_0 thread in its own task, which initially gets all physical memory. But the initial physical memory allocation can be rearranged to suit as needed.

Page faults are also consistently mapped to IPC operations. Each thread has a pager thread attached to it. The pager resolves page faults transparently. A page fault is mapped to an IPC message to the pager on behalf of the faulting thread by the kernel. The IPC message carries the address and access attempt.

All microkernels which implement that set of basic objects belong to the L4 family of microkernels. Modern L4 microkernels implement more objects. They also implement a capability system covering these objects. Fiasco.OC is one representative of such an L4 based microkernel.

2.3.3 The Fiasco.OC Microkernel

Fiasco.OC is based on the design and principles of the L4 microkernel family. So it features tasks, threads and IPC gates as basic building blocks. It implements user-level memory management, but also features rudimentary kernel memory management.

Beyond these, Fiasco was extended with a security system based on capabilities. That fits the need for a better implementation of the principle of least privileges (POLA) than using ACLs.

With vCPUs, we have a rather new feature in Fiasco. We will describe that feature in detail in the next section, as it is vital for our porting efforts.

There is support for hardware-assisted virtualization on modern x86 based computers available with KARMA. Fiasco implements a VMM and a hypervisor, making it a solid foundation to run virtual machines on top of it.

The Fiasco microkernel is also capable to use multiple CPUs in a symmetric multi-processing environment.

2.3.4 Virtual CPUs in Fiasco

In L4 a thread is designed to either execute or wait for an incoming IPC message. It cannot do both at the same time. Thus, asynchronous workloads have to employ multiple worker threads. While the main threads executes a program, the worker threads spin around an IPC gate in a tight loop and wait for events. If the workload requires to either receive a message or to execute the main thread, we need to synchronize the main thread with all other worker threads. On the one hand, this is cumbersome, since some workers may still wait for IPC while we want to synchronize. On the other hand, we generate a lot of extra threads in the system, all of which need to be scheduled correctly.

The alternative is to provide an interrupt-driven hardware model to the user. Fiasco implements vCPUs as extended threads, which can be interrupted and later continue their operation ([LWP10]). vCPUs also provide an atomic address space switch, so that execution can continue in another task.

At every switch from/to kernel mode, the hardware saves all registers needed to restore the previous state. The interrupting routine now saves all other registers which it tampers with during its operation. Usually these are all registers in the machine. When returning to the interrupted operation, all tampered registers are restored. The vCPU feature in Fiasco implements a more general solution. The Fiasco kernel itself should not be aware of any state information beyond the one necessary to switch its L4 threads. On a vCPU event, the current set of CPU registers need to be provided to the registered event handler. The event handler's task has a vCPU saved state area which is also known to Fiasco. As the Fiasco kernel does not know which CPU registers will be clobbered by the event handler, it dumps all of them into the saved state area on an event. The current operation is now interrupted and the vCPU thread executes the event handler. Therefore it might need to migrate between L4 tasks. The event handler can now examine the reason for the interruption, exactly like interrupt routines do on real hardware. When resuming the interrupted operation the vCPU enabled thread migrates to the location and task provided in the saved state area.

vCPUs enable us to adapt legacy operating systems more easy. The operating system expects an interrupt-driven design, which vCPU enabled threads deliver. A physical CPU can be represented as a vCPU. On a multi-processor machine, there is more than one physical CPU executing code in parallel. We can establish multiple vCPU enabled threads to match multiple physical CPUs. We can even start more vCPU enabled threads than there are physical CPUs available in the system, although that would hit performance badly.

2.4 Virtualization

As we have seen in the previous sections, monolithic operating system kernels do not comply with the requirements. We realize that they have poor isolation and lack stability due to their complexity. But we also realize that applications rely on an operating system. So we face the problem that operating system kernels are incompatible to other operating system kernel interfaces, too. The exposed application binary interface (ABI) of different operating system kernels do not match. That situation cannot be handled by stacking operating systems on top of each other, because the expected instruction set architecture (ISA) does not match. So we cannot easily run an operating system on top of another. Operating systems assume to run on a hardware ISA.

We could cope with that situation with encapsulated monolithic operating systems on top of a small virtual machine monitor (VMM). The VMM is driving these virtual machines and is responsible for resource sharing. That would also open the door for new applications. We could place critical services outside of those unreliable virtualized operating systems ([CN01]). System security greatly benefits from that approach.

2.4.1 Virtualization Criteria

Popek and Goldberg have identified three criteria, which an ISA has to meet in order to build efficient virtual machines on it ([PG74]). Running an operating system as a virtual machine must have the same effect as running it on real hardware (*equivalence*). The virtualization should also be *efficient*, which means that most of the instructions should run on the current CPU and should not be emulated or simulated. This is only possible, if all sensitive operations are privileged. On the i386 architecture, however, this is not the case ([Law99]). The third criterion is *protection*. The VMM is the only instance of control for all resources in the system dedicated to each virtual machine. It guarantees that no virtual machine exceed their resource limits.

2.4.2 Encapsulation Techniques

Since on the i386 ISA not all sensitive operations are privileged, VMWare found a way to substitute the problematic instructions and translate these into uncritical operations ([DBR98]). That approach was very slow.

Processor manufacturers have recognized the need for efficient virtualization of the i386 ISA. They added a new operating mode, which compensates for the shortcomings of the i386 ISA. So register virtualization extensions came to high-end CPUs.

But it became clear soon, that virtualizing the MMU leads to a massive performance increase ([AA06]).

There are still many i386 CPUs out there, which lack these virtualization extensions. Especially in the low-budget segment, e. g. on smartphones. So we need to use a different approach to efficiently virtualize an operating system. As we have seen earlier, we could recompile problematic instructions. That approach is highly complex. It needs a deep understanding of how the ISA works in detail. It gets even more complex for multiprocessor environments.

Another approach is to have the virtualized operating system help the VMM on its intentions. With this para-virtualization measure, we patch the operating system to tell the VMM before it executes problematic instructions. We can go even further. We can alter large parts of the operating system's ISA interface to fit the VMM's API. In this rehosting effort, we essentially create a new architectural port of the operating system. This is still perfectly compliant to the Popek and Goldberg criteria.

When we speak of a virtual machine monitor, we want to have a small entity to drive the operation systems on top of it. Microkernels strive for compactness. So it is obvious to use a microkernel as a foundation for the virtualization approach. Modern L4 based systems have everything it takes already in place. Fiasco with its vCPU feature is the candidate to fulfill the rehosting efforts.

3 Design

This chapter discusses problems and possible solutions on all vital aspects of rehosting the OpenBSD operating system on the Fiasco microkernel. That basically involves looking into two things more closely: virtualizing the register set on the CPU and virtualizing the memory management. Therefore this chapter includes a discussion of the differences in memory management first. then we have a look at the general execution model with interrupt and exception handling. As a last step, we will discuss which device drivers are necessary to get a minimal working OpenBSD kernel service its applications.

The major goal is to be able to use OpenBSD as a virtual instance on Fiasco, without the need for virtualization extensions in the host CPU. That gives us the flexibility to use and extend the solution to run on low-end CPUs commonly found in mobile devices.

3.1 Memory Management

In this section we will have a look at the memory configuration of native OpenBSD on the i386 architecture. We compare it to the rehosted version, running on the microkernel.

3.1.1 Native OpenBSD Memory Configuration

OpenBSD memory management is separated into two distinct parts. The machine independent memory manager is called uvm(9) ([CP99]). This portion knows about high level information like shared memory regions, memory mapped files or code, data and stack segments. The uvm(9) subsystem assumes that memory is paged and that there is a memory mapping mechanism, which needs to be implemented in the machine dependent part of the kernel.

The machine dependent part of the memory manager is called pmap(9). It has to be implemented in the architecture code of the OpenBSD kernel. That layer basically maps virtual to physical addresses per process. It abstracts the different designs and

Figure 3.1: Memory layout of OpenBSD on the i386 architecture.

implementations of memory management units from the machine independent part. On i386, a pmap represents the complete 4 GB virtual address space. Every process in an OpenBSD system has its own pmap attached to it. As a special case even the kernel maintains its own pmap. On i386 the kernel pmap's page tables are merged with all process's pmaps to get a higher-half kernel as a result.

The current process's page tables are mapped recursively at a fixed address in kernel memory (for details see section 3.1.3). Pages above address $0xD0000000$ are kernel pages and are placed into the kernel pmap. Mappings below the *VM_MAXUSER_ADDRESS* address, are userland mappings. Figure 3.1 shows the memory layout of the OpenBSD address space on i386. We can see the pmap split at *KERNBASE*. Mappings between these two addresses are process-specific mappings and contain the kernel stack and the page tables. The uvm layer tracks all page directory pages used by the kernel pmap. When extending the kernel virtual address space, a page directory page is allocated and immediately copied into each process's pmap. In that way the kernel mappings stay the same between context switches.

3.1.2 Rehosted OpenBSD Memory Configuration

The Fiasco microkernel occupies the top-most gigabyte of virtual memory in in every task already. That leaves us with the lower 3 GB of virtual address space available in userland. As Fiasco is the only instance of control in the microkernel system, the rehosted OpenBSD kernel needs to be implemented as userspace application. That means, we are not able to access any memory location in the upper gigabyte region, the Fiasco space.

So for our rehosting efforts, we need to relink the kernel to an address below the Fiasco kernel memory space. As we want to retain the separation between kernel- and userland on the rehosted system as well, the OpenBSD kernel will service its applications as a separate L4 task, which effectively makes it an L4 server.

3.1.3 Page Tables on Native OpenBSD

To map virtual memory addresses to physical memory addresses on the i386 architecture, a two-staged set of page tables is used ([Cor10c]). Each page represents 4 kB of physical memory. OpenBSD does not use superpages on i386, which would allow larger page sizes. It uses a recursive approach to map page tables into the process specific address space (PDE area in figure 3.1). That address space needs to be accessible only from the kernel, so it registers as kernel address space. The maximum memory usable on the i386 architecture is limited by the word-width of 32 bits to $0xFFFFFFFF$ bytes. The pmap layer hides that complexity from the rest of the memory management code by providing each process with a pmap to maintain these mappings.

The whole set of page tables for an address space fits linearly into 4 MB of the virtual address space. The location in virtual memory is chosen to be aligned to a single first-stage page, a page directory page. All page table pages mapped in that page directory page contain the page table entries of the current active process. Entries in a page directory page and a page table page are compatible concerning their size and present

Figure 3.2: Exception handling on native OpenBSD for the i386 architecture.

bit. So we can have one entry in the page directory pointing to itself, acting as page directory page and as page table page at the same time. In this is the way that particular page directory page is referenced recursively as page table page, too.

When the kernel wants to manipulate page table entries of another process than the currently mapped one, a full context switch would be necessary. OpenBSD uses the *APDE* (figure 3.1) space as an alternative page table mapping area for that case. Page tables of another process can then be mapped there for manipulation.

3.1.4 Page Tables on the Rehosted OpenBSD Server

The rehosted OpenBSD server maps and unmaps pages into all of its applications' address spaces. We maintain a copy of all page tables for two reasons. First, since the page table handling is already in place, finding the place to map and unmap pages is trivial. It only needs to be done when manipulating page table entries. Second, with the available page tables we are always able to look up mappings later, if needed.

We cannot reload page tables on the MMU directly. On the other hand, we do not need to do so. Each userland application is placed in a separate task. The OpenBSD server maintains the page table mappings and informs Fiasco upon changes on them. Fiasco performs the context switch for us when we resume the vCPU operation and switch the address space.

3.2 Execution Model

We want to understand the execution model on i386 based hardware first and how OpenBSD implements it, before we are going into a detailed description on how to reproduce that behavior on a Fiasco microkernel system.

3.2.1 Native OpenBSD Execution Model

The i386 architecture knows four rings, while only two of them are used on OpenBSD. Ring 3 represents usermode and ring 0 is kernelmode. The OpenBSD kernel runs in kernelmode. When an exception occurs, the current control flow changes. The machine traps into the kernel and execution continues at the appropriate IDT vector to handle that fault. Figure 3.2 shows a schematic of that situation. When we need to transit rings, which means a switch from usermode to kernelmode needs to be performed, the

stack pointer also changes to a stack within the OpenBSD kernel area. The hardware saves the old stack segment and the old stack pointer on the current stack location. In every case, even without transiting rings, the hardware saves some more registers, needed to resume the interrupted operation. These are the flags, the code segment descriptor, the instruction pointer and an error number. On OpenBSD the remaining general purpose registers are now saved in software on the stack, too. That leaves us with a complete trapframe. Now OpenBSD assigns each of these traps a trap number to distinguish them in the further process. It calls a generic `trap()` function, passing it a pointer to the trapframe. With that information, the function is able to reconstruct the fault and take the appropriate measures.

There is a similar mechanism in OpenBSD for device interrupts. Everything is handled the same as for the exception handling. But as a last step, OpenBSD does not call the `trap()` function, though. It iterates along a chain of interrupt service routines, set up by device drivers during device enumeration at boot. These service routines first check, if the interrupt was sent by their device and handle them, if so.

3.2.2 Concurrency Control in OpenBSD

There are different measures to control concurrently running operations on OpenBSD. An interrupt may appear at any time, so we need to have these measures to save critical data structures from being modified by two parallel operations.

The OpenBSD kernel may execute privileged instructions, since it executes in kernelmode. In particular, it is able to turn interrupt delivery on and off for the whole system. This is necessary to guard critical code sections, which need to be executed without any interruptions. This is done with the `cli` and `sti` instructions.

If locking the whole kernel is not necessary, OpenBSD can globally lock subsystems at different levels by raising the bar for servicing interrupt requests. So the execution model of OpenBSD is based on different interrupt priority levels (IPLs). It may make use of prioritizing IRQ controllers. When an interrupt occurs which has a lower priority than the current executing kernel function, the interrupt routine is marked as pending. It is executed as soon as the IPL drops sufficiently low.

After servicing an interrupt or exception, OpenBSD/i386 restores all registers from the trapframe and returns to the interrupted context by issuing an `iret` instruction. Any kernel subsystem can express the wish to switch the current running process. Therefore OpenBSD implements a technique called an asynchronous system trap (AST), a feature originally found on VAX CPUs. That machanism finds the spot where the OpenBSD process is about to return to userland. There, it can execute delayed operations on that process. On the return path to userland we check the AST condition and enter the scheduler if it is met, thus giving up the CPU voluntarily. Nevertheless after scheduling back, we continue returning to userspace.

3.2.3 Rehosted OpenBSD Execution Model

With the recent advent of the vCPU feature in current Fiasco, we have an approach, which bears a strong similarity to physical CPUs. For newly created threads, we can

Figure 3.3: Exception handling on the rehosted OpenBSD server with vCPU.

register an upcall function as exception and page fault handler for our applications on the vCPU. We can also register as interrupt handler and receive asynchronous interrupt events. That upcall function is basically only an instruction pointer and a stack pointer in the OpenBSD server L4 task. A vCPU resume allows address space switches. With that design, we only have one thread to care for. No need for the synchronization hassle, as in the threaded approach.

When an interrupt or exception event in a userspace application thread occurs, the hardware jumps to the according IDT vector, eventually setting the stack pointer taken from the TSS. This cannot be prevented as this is the way i386 hardware works. We can see a picture of the situation in figure 3.3. The hardware saves the registers needed to resume the interrupted operation. Fiasco then saves all other general purpose registers. Then it looks up the location of the vCPU upcall function as the designated event handler. Since the OpenBSD server registers as event handler for all its applications, the Fiasco kernel prepares the vCPU saved state area – a shared page between the OpenBSD server and Fiasco. It puts all registers on that page, performs a context switch to the OpenBSD server and executes the vCPU upcall function. That function can examine the situation based on the register set from the vCPU saved state area.

Whenever we enter the vCPU upcall function, all event delivery is completely disabled. This contrasts the way in which the i386 architecture operates. We have to turn these flags on explicitly again, to be able to receive nested upcalls. Since we also have one entry point only, we need to figure out which event type caused the interruption first. After that we can go on to the respective OpenBSD routine.

We need to pay special attention to the interrupt delivery flag. On the i386 architecture we can voluntarily enable and disable interrupt events with special assembler instructions. These instructions need to be replaced with the respective vCPU library functions to turn the interrupt delivery flag on and off. Disabling interrupt events, results in just clearing the flag. When enabling interrupt delivery again, Fiasco signals pending interrupts in a separate flag. We need to process these, before continuing execution. The vCPU library functions already take care of that, though.

3.3 Device Driver Handling

There are two different kind of devices for each computer architecture, platform devices and peripheral devices.

3.3.1 Platform Devices

OpenBSD is a preemptive multitasking operating system. Preemption is enforced with a platform-specific timer clock, which interrupts the current process and checks, if it is allowed to continue or if another process is elected to run. That OpenBSD-specific scheduling policy can be reused. It was already tuned to fit the rest of the kernel subsystems and is proven and stable. The rehosted kernel will schedule all its processes on its own, instead of having the microkernel schedule them. On a hardware platform that timer interrupt is normally realized with a clock device external to the CPU. We have to use Fiasco's functionality to replace the platform devices now owned by Fiasco itself.

There is a real time clock server bundled with the L4 runtime environment. That can be used to put an additional thread in the kernel task to sleep. After waking up, the thread can raise a vCPU interrupt event. Putting that in a loop, we have a periodic timer. Unfortunately the resolution of the sleep operation is too low. The RTC server API provides a minimum resolution of one second. As an alternative approach, the looping thread can wait for an IPC message from a non-existent sender. The wait operation can be provided with a timeout scheduled by Fiasco. The timeout resolution is limited by Fiasco's scheduling granularity. It is within a nanoseconds frame. So we finally have a periodic timer with a fine-grained resolution.

3.3.2 Peripheral Devices

A monolithic operating system kernel also provides access to a variety of peripheral devices, and so does OpenBSD. Communication with devices can be achieved in many different ways with interrupts, port-based I/O and memory-mapped I/O. Fiasco.OC is able to route interrupts to a specific userland program. With that approach access to a specific device is possible. For that approach Fiasco exposes virtual IRQ gates. So we need to alter the IRQ controller code to create an interrupt gate in Fiasco for each interrupt in OpenBSD. From a security point of view, we would then need to secure the data transferred with an I/O virtualization technique. Otherwise the rehosted and unprivileged OpenBSD would be able to inject data anywhere in the physical memory of the machine, as DMA is not subject to MMU enforced isolation ([Meh05]).

There is a separate I/O virtualization server in the L4 runtime environment. It enumerates busses on the system and cleanly separates all devices. That server provides an API to access each of these devices and it behaves like a bus itself, so the appropriate bus drivers need to be changed to use that I/O server.

The other I/O operations are more involved and are subject to the problem of not being able to run privileged instructions or mapping specific physical memory pages in the OpenBSD kernel task.

As a rehosted kernel, we wish to have access to a number of peripheral devices. There should be a minimal number of devices necessary to have a running OpenBSD system, without networking. These devices need to be rehosted with the rest of the kernel. Unbundled device drivers may not be able to run correctly on the rehosted operating system. If there is a matching API from the L4 runtime environment available, a driver needs to be changed to use that to perform its duties. For input and output of a shell, a serial line driver can use the console API of the L4 runtime environment. Userland programs will be located in a ramdisk.

4 Implementation

In this chapter, we will go through the implementation details of rehosting the OpenBSD operating system on the Fiasco microkernel. At first, we will walk through the bootup sequence which needed a lot of attention. Then we will explain the event, memory and process subsystems. The last step is to have a look at the available drivers in our setup and how they behave differently from their native counterparts.

The implementation was based on Fiasco and the L4 runtime environment revision 29 from the official TUDOS repository.[1] We used the latest OpenBSD release available at that time. The work was performed on OpenBSD/i386 version 4.8. The implementation emulates a single processor machine. Fiasco was configured to implement the vCPU feature. Despite a configured L4 runtime environment it was important to have the RTC server for getting the system time, the I/O server and the vCPU library set up.

At first, we stripped down the default kernel configuration until we had a running OpenBSD/i386 kernel with a minimal feature set. This is useful because it allows to focus on the basic features to get started with. The minimal configuration set consists of the kernel debugger, the IP network stack, the FFS filesystem, a ramdisk device, a BIOS and CPU driver, and an ISA bus driver for IRQ handling. This new configuration for the i386 architecture was the basis for our porting efforts to the Fiasco microkernel. Later in the process, the driver for the mathematical coprocessor had to be ported, too. Additionally, a serial port driver had to be written from scratch.

Most implementation details can be found in L4Linux, too. It was a great source of inspiration for the current OpenBSD implementation. Especially the *l4lx* kernel library for L4Linux already encapsulates the L4-specific implementation of many generic tasks of an operating system and was used nearly unmodified. Namely the task and interrupt handling routines were extremely useful.

4.1 Loader

The OpenBSD development environment (toolchain) assumes that we are compiling the sources on an OpenBSD machine. More specifically, there is no official support from the upstream OpenBSD development team if we do not compile the kernel with a matching OpenBSD userland. Because userland and kernel are distributed as one, the kernel can rely on being built with the distributed toolchain. In that way OpenBSD developers do not need to preserve long-standing backwards compatibility. As opposed to the more common Linux environment, OpenBSD uses a BSD userland and a BSD toolchain. Since the L4 runtime environment and the Fiasco microkernel both are compiled and tested on systems with a GNU toolchain, it would be a hassle to get these running on

[1] `http://svn.tudos.org/repos/oc/tudos`

an OpenBSD stack. On the other hand, the rehosted kernel will need to link against the L4 runtime environment. A small loader, used as wrapper, was implemented. The loader solves the following problem.

After compilation, the OpenBSD/i386 kernel image is a static binary, containing all functionality to run on a PC. It does not depend on any external libraries. In our implementation, all function calls to the L4 runtime are stored as weak symbols in the final static ELF binary, in a special section. That means, it is possible to compile the OpenBSD kernel on an OpenBSD system. Then, the final kernel binary will be wrapped by the loader and compiled on a system using a GNU toolchain to link against the L4 runtime libraries. The loader loads the wrapped OpenBSD kernel image at the desired location in memory. Whenever the kernel executes an L4 function, the loader resolves the symbol on the fly and finally executes the corresponding L4 library call transparent to the OpenBSD kernel. It also stores the function pointer for subsequent calls. This works the same way as dynamic loaders do.

4.2 Early Boot Sequence

The early boot sequence is radically different from the original OpenBSD/i386. A detailed explanation of the OpenBSD bootup process can be found at ([DIN05]). When the rehosted OpenBSD kernel gains control from the loader at boot time, it is already running in paged mode. Moreover, it is running as a simple user process itself. To be able to exploit the vCPU capabilities of Fiasco, the program spawns a new thread with the vCPU feature enabled on it. For vCPU to function correctly, an upcall function and an initial stack was provided. That new vCPU thread will be servicing OpenBSD functionality to other tasks on the microkernel. For that matter, this thread is called the OpenBSD server.

The bootup thread serves as both, pager and exception handler for the vCPU thread. Should there be something going wrong with the OpenBSD server thread, the bootup thread still acts as a basic exception scheduler. This is a fail-safe solution for debugging purpose only. When exception handling is disabled on the vCPU thread, there may be exceptions raised nonetheless. This happens for example, if the stack for the vCPU upcall function points to an unmapped memory location. So we still get a decent error message in this case.

After having set up basic vCPU functionality on the OpenBSD server, it prepares the boot sequence of a genuine OpenBSD/i386 installation. The server first enumerates the CPU type, then it sets up the physical memory allocated from the L4 runtime environment. After that it eventually copies a ramdisk image into the space provided by the ramdisk driver, according to the kernel configuration. Now event delivery is enabled on the vCPU for the following call to `init386()`, which is normally the first C function executed on OpenBSD/i386 to "wire 386 chip for unix operation"[2]. At that point we have a booting OpenBSD instance. Finally, it jumps to the machine independent startup routine `main()`.

[2] See comment in sys/arch/i386/i386/locore.s

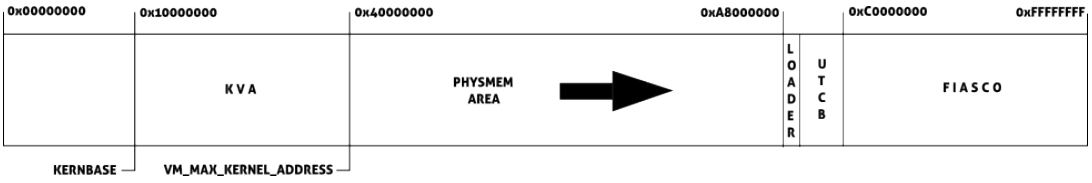


Figure 4.1: Virtual memory layout of the OpenBSD server on L4

4.3 Memory Handling

In this section we will have a look at the memory layout of the OpenBSD server and its programs. We will also see how problems with the recursive page table approach have been circumvented.

4.3.1 OpenBSD Server Memory Layout

During the early boot phase, the L4 runtime environment is used to ask our configured data space provider for a continuous chunk of memory, which will be used as guest physical memory by the OpenBSD server. Of course, the OpenBSD server only allocates virtual memory in its own address space, but we pretend that this area is usable as physical memory to the uvm(9) system (physmem area in figure 4.1). Therefore that memory can be provided to OpenBSD applications.

On the other hand, that memory will not be able to be used for DMA transfers or other memory mapped I/O. DMA needs direct access to the machine's physical memory. The L4 runtime environment also provides a solution here, but for the sake of simplicity there are no drivers available which need memory mapped I/O or DMA transfers at the moment.

The kernel virtual memory area used by native OpenBSD/i386 is located at the top-most $0,75\,\mathrm{GB}$ of virtual address space in every process, much like Fiasco. OpenBSD/i386 does not use any additional memory. Since the upper gigabyte is already occupied by Fiasco, we lay out the kernel virtual address space for the OpenBSD server to be at a very low address range (KVA in figure 4.1). We also tell uvm(9) about the size and location of that area within the kernel pmap. Other than that, the amount of allocated address space dedicated to kernel operation remains the same. Figure 4.1 shows the virtual memory layout of the OpenBSD server in detail. One can see, that the physmem area can grow from above the kernel virtual address space up to the loader. Therefore, we can see that OpenBSD on L4 can allocate as much as $1,6\,\mathrm{GB}$ of guest physical memory with the current server layout.

In a way, we can call the segmentation of virtual memory space between the OpenBSD server and its applications a $3\,\mathrm{GB}/3\,\mathrm{GB}$ split.

4.3.2 Page Tables

In the original OpenBSD/i386 memory layout, there is a kernel area which is local to each process (c. f. section 3.1.1). With such a fixed location for the page tables in kernel virtual address space, we would need to remap the page tables on every context switch. That would make the memory manager over-complicated and slow. We replaced the recursive mapping and scattered all page tables across the kernel virtual address space. Additionally, in our approach page tables themselves do not need a mapping. Page table entries always contain physical addresses and in our approach we can access these directly. So only every page directory page has a unique address in kernel virtual address space.

OpenBSD manages its own kernel virtual memory. None of these mappings are entered into any process's page tables. The OpenBSD server is treated like any other ordinary OpenBSD process, concerning memory mappings. There is no need to insert kernel mappings in process page tables. The only difference between them is that mappings for the kernel are done in the same L4 task.

The pmap layer only ever gets active when a page fault occurs or a mapping has to be established, removed or remapped. OpenBSD uses an eager mapping approach for kernel mappings. At that point it is also necessary to create the mapping on L4. Since we have told uvm(9) about both regions, the pmap layer is automatically always asked for kernel mappings in these areas only, although they reside in the same memory space. There is no need to handle kernel memory in a special way.

Page mappings for userspace applications are done in a lazy way. The pmap layer manipulates the page table entries as it would on real hardware, but no pages are mapped into the task. The mapping is not established on L4 until the vCPU runs into a page fault. At that time the process's page table is traversed looking for an already established mapping in its pmap. If it is found the page gets sent to the L4 task in which the process is running. Otherwise, the page fault handler is called like on real hardware, eventually establishing that mapping which we can evaluate afterwards. Page faults are a frequent vCPU event for a userland process. Therefore, a feature exists to send a page to the faulting process at the time when the vCPU upcall resumes. That saves an extra IPC call for the mapping. Unmapping virtual memory for any process undoes the mapping when clearing the page table entry for its pmap. At that point, the page in memory is eagerly unmapped for both, kernel and userland processes. That allows an easy remapping of a page table entry by eagerly unmapping the old page and map the new one in a lazy way.

When resolving page faults for processes, we do not need to emulate the access and dirty bits found in the i386 MMU. These bits are already taken care of when establishing a mapping in the pmap layer. They are set, regardless of the MMU. However, on i386 there is the %cr2 register, which indicates the last page fault address. We do get that address from the vCPU saved state area, but we need to rewrite the page fault handling code. That register is only accessible from kernelmode. Our solution attaches the vCPU page fault address to the current process. In this way we can always resolve the fault, even when we get interrupted.

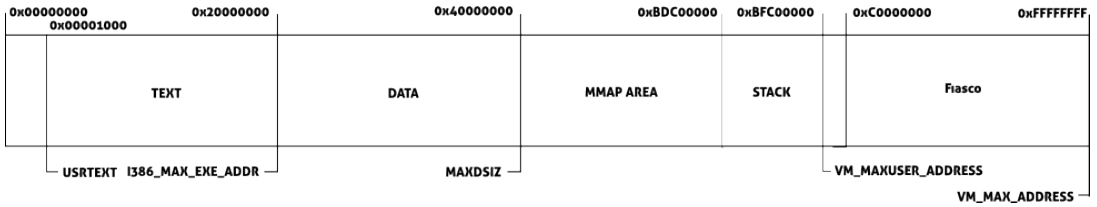


Figure 4.2: Memory layout for userspace applications on the rehosted OpenBSD system.

4.3.3 Userspace Memory Layout

Like many other operating systems on the i386 architecture, OpenBSD maps into the upper virtual memory region of all processes, too. In the original layout, the kernel starts at virtual address $0xD0000000$ (c. f. figure 3.1). In the rehosted environment, Fiasco occupies all memory starting from $0xC0000000$. That leaves ordinary OpenBSD programs with $0,25$ GB less virtual memory space. As we can see from figure 4.2 the upper memory area available in userspace is used for the program's stack. Current implementation on L4 lowers the stack area of an OpenBSD program, which always grows down from the *VM_MAXUSER_ADDRESS* memory location. Since Fiasco occupies the upper memory space already, we have relocated that address to start below the Fiasco kernel address space. The other address regions in userland remain unchanged. Even the stack size remains the same. With these changes no recompilation of userspace applications is needed. We are completely ABI compatible. However, these modifications reduce the virtual memory space for memory mappings of files or devices with the mmap(2) system call. This should not be a problem in practice and the out-of-memory case of an mmap(2) call shall be handled by programs in any case.

The i386 platform has a fundamental flaw in memory handling. Any mapping is executable That leads to all kinds of attacks, with buffer overflows being the most prominent. Figure 4.2 also shows a dedicated area for code in the program's virtual memory layout. The native OpenBSD implementation restricts execution of code to that specific area. OpenBSD enforces that restriction by altering the segment descriptor for the code segment and monitoring the upper boundary of executable code in the current memory layout. When a program tries to execute code outside that segment, an exception is raised and the process gets killed. So, while the OpenBSD memory allocator preserves the memory layout, we would need to propagate the %cs segment descriptor to Fiasco. At the moment only the %fs and %gs registers are propagated at vCPU resume. That change would have involved changing Fiasco. Therefore, the OpenBSD server does not have the write-or-execute feature found in the native implementation. From a security point of view it is no problem to limit any segment for a thread as it does not extend its access rights.

4.3.4 Accessing Userspace Memory

The OpenBSD server needs to access memory regions from userspace applications. On L4, the copy(9) functions cannot access userspace memory directly, unlike on real hard-

ware. Fortunately we have hints available where the userspace memory is located in the OpenBSD server memory space. We have direct access to the physmem area, where all physical memory for userspace processes is located. We also have access to the process's pmap, which holds the information about the mappings between the physmem area and the virtual memory in the process's L4 task. With that information, we can reassemble all parts necessary for a successful copy operation. We copy the desired data directly from its physical memory location into kernel virtual memory, by traversing the page table and copying the information from each physical page. Copying data from kernel virtual memory to userspace uses the same mechanism for target memory locations in the physmem area.

4.4 Process Handling

Processes are implemented as L4 tasks. The vCPU thread migrates between the OpenBSD server task and its running processes. A task has an own address space. In this respect from an OpenBSD point of view an L4 task is bound to a pmap, so we have a one on one mapping between these two. That pmap-centric approach makes it easy to map operations on pmaps to L4 task operations. There are basically only three important operations for a process during its lifetime.

1. Create
2. Switch
3. Destroy

4.4.1 Process Lifecycle

For each pmap, OpenBSD maintains an address space with an own set of page tables. That is exactly what a task represents on L4. The create operation is lazy evaluated. A new L4 task is created when resuming the vCPU upcall from a context switch to that newly created process for the first time. Additionally, on OpenBSD/i386 the process control block is located on the lower boundary of two consecutive pages in RAM. Above that block we save a pointer to the UTCB and one for the current vCPU state of the process. That simplifies a later extension for SMP support. The kernel stack is located on top of these pages and grows down.

When switching to a new process during a timer interrupt or voluntary preemption, OpenBSD/i386 saves all registers of the current running process and simply switches the kernel stack pointer to the new process. In that way, the stack frame of the new process can be rolled up again. After loading the register set of the new process, the switch is complete. Since there is no operation involved which uses privileged instructions, that behavior is also used on L4. The only thing we need to do is tell the vCPU to switch the task where the execution should resume. We also switch the vCPU stack pointer. It points to the kernel stack of the new process.

On process exit, OpenBSD destroys all its maintenance structures and releases its memory. When the pmap structure is freed, the L4 task is eagerly deleted, too. All

memory mappings in that task are deleted by Fiasco and the L4 task gets destroyed. OpenBSD removes all memory mappings from a dying process. Due to the fact that all memory mappings will also vanish on L4 when the task is destroyed, we do not need to invoke all the IPC calls to remove all mappings from the task in this case. We flag every pmap on destruction and skip all IPC calls to unmap the address space. However, we still perform the unmap operation on OpenBSD, as the native implementation does to free the page tables again.

The create and switch functionality are implemented using lazy evaluation. They are performed right before resuming to the interrupted operation. This is done, because we need to modify the vCPU saved state area. That area is not re-entrant safe. It must only ever be modified in one of the following two cases, each having interrupt event delivery disabled on the vCPU. One, we can modify this area right before resuming. And two, we can modify this area directly after entering the vCPU upcall function, where event delivery is automatically disabled. If we want to to modify the saved state area with interrupt delivery enabled, the current running operation might be interrupted, leading to a new state in that area. After resuming to our operation, we cannot tell if the saved state is valid or not anymore. This is true for all events delivered to the vCPU. But this is especially problematic for interrupt events, since we cannot influence them outright. In practice, chances for a page fault or an exception in the vCPU entry or exit phases are non-existent. That means, we consider the saved state for a vCPU bogus by the time interrupt events are enabled for it.

4.5 Asynchronous Events

The vCPU signals exceptions and interrupts as asynchronous events to the OpenBSD server. It also signals page faults, which we have covered in section 4.3. In this section we take a closer look at the event handlers in the vCPU upcall function.

In OpenBSD/i386 the entry vectors for asynchronous events are written in assembler. This is due to the age of the code and lack of manpower. The port to L4 provided here rewrites some parts of it in C. Especially the IPL handling code and traversing interrupt service routines.

4.5.1 Exceptions and Interrupts

To understand interrupt and exception handling on the vCPU, we need to understand how the vCPU details map to the expectations of the OpenBSD system. When entering any of these events from usermode, OpenBSD expects a trapframe on top of the current process's kernel stack. Since we get the information for the trapframe from the vCPU saved state area, containing a complete set of registers of the interrupted task, we can simply copy that register set to the top of the current process's kernel stack, where it is supposed to be located. But as the vCPU stack pointer acts as the initial kernel stack pointer already, we need to lower the first usable stack address below that trapframe. When we interrupt the OpenBSD server itself with a nested event, the stack pointer is not touched at all. It just progresses on, as no ring transition was performed. In that case, we construct the trapframe in a local variable in the upcall function. This mimics

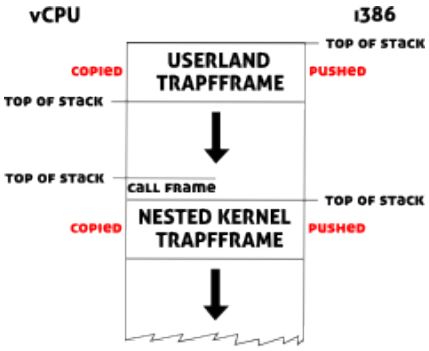


Figure 4.3: Kernel stack layout with trapframe location on vCPU and i386.

the i386 behavior best. Figure 4.3 illustrates the differences in how the trapframe is built between the original i386 architecture and our vCPU implementation. The bottom line is that we need more stack space for the vCPU implementation, since it has a new call frame on each vCPU upcall invocation. That includes a new set of local variables, which were not needed on bare metal. In the end we will run out of kernel stack space sooner with the vCPU implementation. So we need to make sure to not be wasteful with stack space, specifically in the upcall function.

The trapframe contains all necessary information to service the event. One of the interesting points is the distinction between kernel and user mode. On i386 hardware, the code segment register is only used as a selector for the global descriptor table. The lowest two bits in that register indicate if an interrupt originated from userland. When copying all registers from the vCPU saved state area, we set these two bits according to the vCPU state. OpenBSD uses that information to see where an event originated from. We also set the interrupt flag in the flags register according to the saved vCPU state to get as much information out of the vCPU state as possible as long as it is valid. We need that information to restore a valid CPU state in the return path. Now that we have extracted all information from the saved state area, we can enable interrupt delivery, if it was enabled in the interrupted context. We need to do that distinction here, since we also service page faults with this very same routine. These may occur regardless of the interrupt flag.

To service exceptions, a mapping table is now consulted, making generic OpenBSD trap numbers out of the vCPU event numbers. After that the generic `trap()` function is called to service the exception. That function gets a pointer to the current trapframe and is thus able to recover the fault situation on its own.

To service interrupts, the interrupt number was provided as a label attribute along with the register set in the saved state area before. We check the IPL and traverse the list of interrupt service routines accordingly. The IPL is also set according to the current IRQ. If the IPL is too high to service the current IRQ, we flag it for later execution. When the IPL is lowered, all outstanding service routines are executed.

When returning to the interrupted context, we need to disable interrupt event delivery first. That is to prevent clobbering the vCPU saved state area. Then we copy all the information from the trapframe back to the vCPU saved state area. After that we can resume and eventually change the address space.

We also need to provide a callback routine to handle pending interrupts for the vCPU library functions. In that case, before going through the chain of interrupt service routines, the callback routine needs to adjust the current trapframe, to make the code segment selector look like the interrupt service routines were called from kernel mode. Normally we would need to generate a completely new trapframe. But this way is shorter and quicker. After servicing the pending interrupts, it must adjust the code segment selector back, of course. That implementation is possible, because OpenBSD always winds up its call frames again. There is no fast path or shortcut to return to usermode.

4.5.2 Timer Interrupt

On L4, we regularly need to generate an event to trigger the vCPU upcall to service the clock interrupt. Therefore, a separate thread in the OpenBSD server task is created. It also registers an IRQ gate. Then it waits for a non-existent IPC message with a timeout of $1/hz$ seconds. After the timeout arrives, it triggers the IRQ gate so that it invokes an IPC message to the OpenBSD server thread and make it believe an interrupt event occurred. The hz(9) variable is globally defined in OpenBSD and declares the frequency for the timer interrupt. By default, the clock interrupt triggers at a rate of 100 Hz.

The service routine for the timer interrupt is registered during system startup. Therefore the vCPU upcall function runs the interrupt service routine on a regular basis. That service routine needs to call hardclock(9), which figures out if the scheduler needs to run, since OpenBSD schedules its own processes independent from the microkernel. The service routine also updates pending timers. OpenBSD also knows a second routine, called statclock(9). That function is used to update the statistical counters for the current running process. These counters are supposed to be updated at a different rate than the timer. This is meant to be a security measure, because it makes it harder for processes to trick the scheduler [3]. For simplicity, we update the statistics counters with the same interrupt service routine as we run the scheduler for the moment.

When registering an interrupt service routine for any other interrupt request, an IRQ capability is acquired and the L4 I/O server is asked to deliver IRQ events to the vCPU thread. In this way, hardware interrupts get delivered to the OpenBSD server.

4.5.3 General Timers

An operating system needs to have a notion of timers to wait for certain events. If, for example, a program executes the nanosleep(2) system call, the kernel puts it into a wait queue and runs the scheduler to execute another process. When is the process supposed to wake up again?

[3] See the hz(9) manual page for details.

In OpenBSD, there is a generic timeout(9) interface. It wraps around timecounter device drivers ([Kam02]). The L4 port of OpenBSD implements its own timecounter device. That device uses a continuous time value from L4's kernel information page. That value has a microseconds granularity and perfectly fits into the timecounter framework, since a timecounter device only needs to provide the timer granularity, the value at which the time value wraps, and of course the time value itself. As an alternative source, the TSC is used, if available. That further improves granularity. The paravirtual timecounter device is used as the default time source in the rehosted system.

All timecounters in the system are queried periodically during hardclock(9). Then, all timeouts are updated. If the timeout for a process expires, a soft interrupt runs, which flags the process as runnable again and puts it back on the run queue.

4.5.4 System Calls

A process may never execute privileged instructions. But it may execute a special software interrupt to pass control to the kernel. That system call interface uses interrupt $0x80$ on OpenBSD. On real hardware, an interrupt is triggered which gets handled by its interrupt handler. The Fiasco kernel has not registered that interrupt gate, so it does not know how to handle it. On the other hand, it does not really have to care either. The vCPU exception handler is on duty here. When an OpenBSD process issues a system call, the Fiasco kernel notices an unknown interrupt and passes a general protection fault to the vCPU upcall function. The upcall routine then generates a trapframe as for any other exception, too. We get the software interrupt number delivered in the error code along with the register set by Fiasco. In the end we simply call the higher-level `syscall()` function, instead of `trap()` (see section 4.5.1). The higher-level function gets that trapframe as a pointer reference. After that we resume execution to the userland process again, but we modify the instruction pointer to point after the faulting instruction, which led us to the invocation of a system call in the first place. That saves us from issuing the same system call over and over again.

4.6 Device Drivers

As mentioned earlier, the implementation of OpenBSD on L4 uses a minimal kernel configuration. Therefore there is only a limited amount of drivers available. The OpenBSD code-base is rather old. This is especially true when it comes to the i386 port, which was used as a foundation for our efforts. Some drivers are completely scattered through the architectural code, like the cpu(4) driver. The most tedious part however is to find all places where privileged instructions are called and replace them with an equivalent operation on the L4 runtime environment. However, there is no driver for external peripherals yet, so we did not need to create separate threads and emulate concurrent behavior. The only device we needed to do that for was the system clock driver clock(4) and our own rehosted serial driver. All other drivers were reduced to stubs with basic functionality. For example, the bios(4) device is clearly not needed at all, nevertheless the i386 architecture code is tightly coupled with it. When asked for BIOS information that driver now provides static answers only.

4.6.1 Serial Driver

For basic operation, an input and output device needs to be available. On i386 hardware that can be accomplished by BIOS operations. Apparently input and output are managed through different devices, so there are more drivers needed for the implementation. A serial device manages input and output in one. It is accessed as a single entity.

When implementing a driver for OpenBSD, [Kun03] is a good reference. Although Kunz was implementing a block device for NetBSD, many of his statements also hold true for OpenBSD. The serial line driver l4ser(4) implements a full tty(4) device. The output is merely piped through to the virtual console interface of the L4 runtime environment. Input is handled by hooking the timer interrupt. The driver establishes a new interrupt service routine and polls the virtual console interface of the L4 runtime environment for new keystrokes in a non-blocking way at each timer interrupt invocation.

4.6.2 Ramdisk

The ramdisk driver needs some attention as it is vital for the implementation. Without it we would not have a working userland. Depending on the kernel configuration, a ramdisk may be compiled during the OpenBSD build process. In OpenBSD terms that ramdisk is called miniroot. The miniroot is a file which encloses a complete filesystem and on it some userspace applications, including everything up to a shell. On the i386 architecture the ramdisk driver leaves some space in the resulting kernel binary. The miniroot filesystem is then patched into the ramdisk driver section. Unfortunately, the patching only takes place, if we build a full-blown release for distribution on CD-ROM.

One aim of the project is that userland applications shall run unmodified. Thus we separately build an unmodified miniroot and keep it as a populated filesystem image. Generating a ramdisk is easy, we need to compile the whole userland and install it in a fresh directory. That directory can be a basis for the ramdisk image. A complete OpenBSD userland installation takes up about 512MB as miniroot filesystem image.

When booting Fiasco the miniroot file needs to be placed along the rehosted OpenBSD kernel executable. All initial binaries are provided as modules to the bootloader grub. During the early boot sequence the OpenBSD server allocates a data space and populates it with the ramdisk. It then patches some pointers in the ramdisk driver to point to the miniroot location. Data space mappings pointing to initial data, loaded by grub, can only be mapped read-only into the address space. Other programs may potentially use that initial data, too. OpenBSD always assumes a writable boot medium. A copy-on-write semantic on the ramdisk data space solves that and the L4 runtime environment has that feature built-in.

OpenBSD is now able to boot without the need for hard-disk drivers. The ramdisk size may exceed all possible sizes for the ramdisk on a physically running OpenBSD instance, since it does not reserve that space in kernel virtual memory anymore. On the other hand, the ramdisk occupies precious virtual memory in the server's space. That memory cannot be allocated as guest physical memory. There is a trade-off between ramdisk size and RAM size. If one of these is too big, the OpenBSD server first allocates

the physical RAM and then fails to allocate a data space big enough to hold the ramdisk. Only access to a harddisk can solve that dilemma.

5 Evaluation

In this chapter, we evaluate the code needed for implementation the rehosted OpenBSD operating system. We also take a look at the performance of the solution.

5.1 Code Evaluation

For the code evaluation, we count the total source lines of code (SLOC) for the current minimal implementation of the rehosted OpenBSD kernel. SLOCCount version 2.26 by David Wheeler provides these numbers[1]. That tool measures physical lines of code, ignoring comments and whitespace lines. Since the current implementation is integrated in the i386 architecture, we only need to analyze the contents of the directories in the source tree containing that machine dependent code. We compare a copy of vanilla OpenBSD 4.8 with the rehosted equivalent. Table 5.1 shows the effective lines, divided in assembler and C code.

The adjustments needed to make a minimal OpenBSD configuration work on L4 have been realized in about 6000 additional lines of code altogether. The setup and initialization code takes up about 1000 lines of the overall volume. These numbers are equivalent in size to the adjustments necessary to port the generic OpenBSD port for ARM to the Beagle board. That adjustment was acheved in about 5100 source lines of code in OpenBSD 4.8.

5.2 Performance Evaluation

To evaluate the performance of the rehosted OpenBSD operating system, we compared it to a native running OpenBSD/i386 installation. All measurements were done on an AMD Athlon XP X2 5200 processor with 512 KB of L2 cache. The machine has 4 GB of RAM, which was completely available to the native system. The rehosted operating system was configured to use 768 MB RAM. Since the rehosted kernel only features support for a single processor, the native installation is also only running as a

[1] SLOCCount can be obtained from http://www.dwheeler.com/sloccount/

	sys/arch/i386 (vanilla)	sys/arch/i386 (L4)
Assembler	4457 LOC	4606 LOC (+3,3 %)
Ansi C	40743 LOC	46661 LOC (+14,5 %)

Table 5.1: Comparison of amount of code between the native and rehosted OpenBSD kernel.

System call	Time (native)	Time (rehosted)	Penalty
Simple syscall	0.1597 μs	1.1476 μs	+618, 6 %
Simple read	0.3843 μs	1.9137 μs	+398, 0 %
Simple write	0.3703 μs	1.8091 μs	+388, 5 %
Simple stat	1.4628 μs	3.0574 μs	+109, 0 %
Simple fstat	0.3351 μs	1.7051 μs	+408, 4 %
Simple open/close	1.9905 μs	5.1110 μs	+156, 8 %

Table 5.2: System call performance compared between native and rehosted OpenBSD.

uniprocessor system. We used lmbench version 3alpha4[2] and needed to adjust minor bits to make it run on OpenBSD ([MS96]). lmbench needs the `/proc` filesystem, which is not configured on a stock OpenBSD installation. There is some Linux-support for `/proc` on OpenBSD, which is fortunately enough to run lmbench without heavy modifications. Otherwise porting efforts would have been more complicated. lmbench was configured to use 64 MB RAM at maximum for its tests.

Both versions of OpenBSD, the native and the rehosted one, have option `DIAGNOSTIC` enabled. That means there is a lot of extra debugging and sanity checks compiled into the kernel. Since the original GENERIC kernel shipped with OpenBSD has that option enabled, We left it in for the performance measurements, too. We could most certainly gain some extra speed, if we left out some of the debugging options. The aim is to have a kernel configuration, which matches that found on production systems as close as possible. Therefore the debugging and extra sanity checks were left enabled. In addition, the rehosted kernel is configured with the option `SMALL_KERNEL`. That option is normally used on ramdisk kernels for the OpenBSD installation CD set. It disables a lot of features, like CPU quirk handling and the hardware sensors framework.

5.2.1 System Call Performance

With system call performance measurements, we get the duration of a system call operation. The baseline is a simple (or null) system call. The simple system call test measures only the entry and exit overhead for calling operating system services. Other basic tests include a simple read of one byte from /dev/zero and a simple write of one byte to /dev/null. More sophisticated filesystem tests exploit a long-running kernel chain. They eventually include waiting for disk operations. The results for the system call benchmarks can be found in table 5.2.

We can see that the overhead to call operating system services is about six times slower on the rehosted kernel. Other basic tests show a similar degradation in speed. But the more complex a system call operation gets, the less impact results from the entry and exit overhead.

Native OpenBSD on the i386 architecture has a distinct software interrupt gate for system calls. The vCPU upcall function is called on every event, including system calls.

[2] lmbench can be found at: http://www.bitmover.com/lmbench/

Fork type	Time (native)	Time (rehosted)	Penalty
fork+exit	213 μs	830 μs	+289, 7 %
fork+execve	631 μs	1906 μs	+202, 0 %
fork+/bin/sh -c	1327 μs	3943 μs	+197, 1 %

Table 5.3: Process fork and exit performance compared between native and rehosted OpenBSD.

We need to distinguish the reason for the upcall first, before we can service the system call request in the rehosted environment.

5.2.2 Process Creation and Destruction Performance

Performance numbers from the fork benchmark can be found in table 5.3. There are three different types of tests performed here. The *fork+exit* test simply forks a new process and exits it immediately. The *fork+execve* test forks a new child process and then executes a new program in-place. The old address space is simply re-used for the text, data and stack of the new program. Finally, the *fork+/bin/sh -c* test forks a new process and starts a new shell in it, which is instructed to execute another program. That leaves us with one more process in the system than for the other two tests.

We can see that a simple fork operation on a rehosted OpenBSD kernel is about 3 times slower than on hardware. As the syscalls itself only account for a fraction of the delay, the remainder is process creation and memory mapping IPC calls to the L4 runtime environment. Although we implement lazy memory mapping, we can see that if we unmap large chunks of process memory immediately, the penalty grows very high. Address spaces are unmapped completely before the process finally exits on OpenBSD. In that case we flag an exiting process's pmap on the rehosted system. Page table updates are not forwarded to L4 anymore. That saves a large number of IPC calls when unmapping all memory regions. Instead of cleaning up the L4 task's memory space, the address space goes away with the deletion of the task itself. However, for the *fork+execve* test, we are unable to perform that optimization. The penalty can grow very high in that case. In the test scenario the working set of the process executing execve is not very large, so the operation is quick here.

OpenBSD zeroes every page before use. The native kernel uses some performance speedups found on the testing CPU, namely SSE2, to zero the contents of a fresh page. The option `SMALL_KERNEL` normally disables these speedups. Nevertheless, on the rehosted kernel these speedups are available and used during the tests.

5.2.3 Real World Performance

The above performance measurements are special cases, showing the net effect of the virtualization layer. In a real-world scenario, programs not only ask for kernel services, but also consume their computing time in userspace. We choose to compile the lmbench suite as a test case. The results can be seen in table 5.4.

Compile run	Time (native)	Time (rehosted)	Penalty
lmbench (system)	1.67 s	1,19 s	$-28,7\,\%$
lmbench (user)	30,34 s	33,68 s	$+11,0\,\%$
lmbench (real)	32,83 s	36,09 s	$+9,9\,\%$

Table 5.4: Real world performance compared between native and rehosted OpenBSD, compiling lmbench.

We can see that user times are not significantly higher than without a virtualization layer. The system time is even a quarter below the native kernel. Due to the low number of drivers, the rehosted kernel is much smaller and lightweight. In addition to that, we are running off a ramdisk, in contrast to the native kernel. Therefore we expect hard drive access times to be lower. A realistic system time benchmark will be possible in the future with a virtual hard drive.

Altogether, the mix concludes in a speed penalty of only ten percent. That also includes all other running threads scheduled on on the microkernel while the test ran. Since we are running on a minimal configuration, the impact is not vast though. If we account for the neglected hard-drive performance, we take another look at the system call benchmarks. We are interested in read and write performance, so we can assume a realistic system time penalty of 400 %. We would then have a real world penalty of about 26 % in total. All that without special hardware support for the virtualization layer.

6 Related Work

There is a vast number of possibilities to run virtual instances of operating systems on a computer. Some require modifications to the virtual kernel or special hardware support, others do not. In this chapter, we will look at other research in this field, but we want to focus on rehosted operating systems as this is the subject matter of this thesis.

A large number of research projects have shifted their focus to Linux as target operating system for virtual machines. This is understandable as most of today's so-called legacy applications for a virtual machine are available on Linux. Only very few of the other projects focus on a BSD platform. So it comes to no surprise that most of the rehosting projects mentioned here have a Linux port only. But this thesis also showed that rehosting multiple operating systems on a microkernel also exposes minor glitches in it, which would have been remained hidden otherwise.

6.1 L4Linux

One approach is to simulate a monolithic kernel by implementing all functionality of all subsystems in a single server, just like the rehosted OpenBSD implementation does. With that design, a single task does all the high-level operating system work. Programs which interface with that operating system server have an easy job finding the responsible task to service their requests. There is no tricky request routing needed. That approach is especially handy to rehost an existing monolithic operating system as we do not need to split its subsystems into different tasks. It still maintains the problem of unpredictable interaction of different subsystems. But on the plus-side we can see, that if a bug hits in such a rehosted operating system and the server dies, it does not affect the kernel, hence the rest of the system remains stable.

L4Linux implements a rehosted Linux single server on top of the Fiasco microkernel ([HHL+97]). The server uses the L4 runtime environment. All Linux programs run as independent L4 tasks. They do not know anything about their different runtime environment and think they are running native on Linux. All communication between Linux programs and the Linux server is accomplished using the microkernel's interprocess communication facility. That can be backed with capabilities which only allow Linux programs to communicate with the Linux single server.

The server uses a set of helper threads to synchronize userspace applications with the kernel. Since Fiasco only provides synchronous IPC for synchronization, resembling the asynchronous nature of an operating system kernel is hard to get right. There is a helper thread in each task for every running process to synchronize to kernel events. There are also helper threads for each IRQ in the kernel task, each with a different scheduling priority on Fiasco, to get interrupt priorities. That behavior makes an inherent assumption about the Fiasco scheduler, which it should not.

Linux processes were originally scheduled on the microkernel. But since the Linux scheduler implements the policy, or scheduling algorithm, that turned out to be not a good idea and was abandoned later. The Linux scheduler is well optimized for most workloads.

Newer versions of L4Linux also use vCPU threads to implement asynchronous behavior in the kernel, although there is no paper describing that system yet. That makes the L4Linux and our OpenBSD implementation be on par concerning the platform abstraction.

6.2 MkLinux

Linux was also rehosted to the Mach3 microkernel in the MkLinux project ([dPSR96]). That approach is very similar to the L4Linux approach and uses equivalent mechanisms found on Mach and its runtime environment.

The Linux kernel was implemented as a single server with multiple threads. Since a kernel is an event-driven program, MkLinux featured multiple threads. Each thread was waiting for an explicit type of event, like system calls or interrupt requests. All threads share a global lock, If a thread runs the risk of going to sleep, it must release the lock. That makes operations in the Linux server not easily preemptible. The locking semantics are even more complicated, since the microkernel schedules all user programs, too. But Mach doe not know of any policy decisions. It cannot tell when a thread may run and when it does not. So the global lock also influences which user task may run.

As the Linux server runs as a separate task, accessing user memory is quite an issue here, too. Userspace memory areas are mapped into the server's address space for access and inspection. MkLinux did not have the total physical memory in its address space, nor did it keep its page tables. So accessing userspace memory becomes less reliable, since MkLinux is bound to an IPC call, which might eventually fail.

The Linux server could also share its address space with the Mach kernel as a collocated instance. But that would completely antagonize the need for a separate Linux server in the first place, of course.

6.3 User Mode Linux

There were also successful efforts to rehost Linux on a monolithic kernel, namely itself ([Dik00]). That version of Linux is called User-Mode Linux, as it runs the kernel as a simple userspace program. The rehosted kernel uses standard facilities of the underlying Linux kernel. Processes in the virtual machine are mapped to processes on the real machine. Whenever a virtual process issues a system call, it needs to be redirected to the rehosted kernel. This is done using the ptrace facility of the underlying kernel with a dedicated tracing thread. The same mechanism is used to send signals to processes. Traps and interrupts are implemented using the signaling facility of the underlying kernel. All of these facilities were not designed to serve these purposes in the first place. In that respect, the interface between the host and the virtualized kernel is suboptimal and calls for slow performance.

The user-mode Linux kernel uses a special device driver devoted to use for block devices. The driver is implemented as part of the host system to connect arbitrary files – e. g. a flat file or a device node – as storage backend for the virtual system. This hack, which does not fit to the rest of the idea, was introduced to circumvent double buffering files. Although, it provides a clean interface between the host and the guest.

6.4 Xen

Xen ([BDF+03]) differs from the previous approaches as it was never meant to be a fully-fledged kernel. It was designed and built as a virtual machine monitor from the ground. Although it borrows some patterns from modern microkernels, like having policy decisions made in userland and only having the hypervisor provide the mechanism and enforce them. For that reason there is a dedicated virtual machine for policy configurations.

Furthermore, Xen deeply exploits the x86 architecture. The VMM runs with ring 0 privileges. Virtual machines run at ring 1 and therefore can maintain their own page tables and segment registers. As opposed to our solution presented in this thesis, they maintain the actual page tables, which also get activated on the CPU at a point. That needs strict security checks in the hypervisor. Xen's philosophy is to secure the whole system with proper access control within the hypervisor. This is a contrast to the concept of a microkernel and is thus not possible on Fiasco.

The Xen hypervisor is located at the top of every address space. The virtual operating systems are located below. When an architecture only supports two privilege levels, the guest operating system needs to be placed into a separate address space to protect it from its programs. In that case the guest cannot use global pages for their mappings either. In that case, Xen reflects something like the solution provided in this paper. But on x86 with its 4 rings, Xen can perform nearly at native speed.

Xen's event mechanism also resembles more closely to what a guest operating system expects from a real CPU. The rehosted kernel may be interrupted at any point through asynchronous events. On the other hand it may only issue synchronous calls to the hypervisor. That reduces the porting efforts. The design implemented in our approach is based on the same idea.

6.5 Conclusion

In all of the examples above, userland applications always run unmodified with respect to the binary loaded by the kernel. In some cases the binary may have been altered at load time to rearrange some operations for the rehosted kernel. These are mostly performance optimizations. The solution provided in this thesis gets by without any binary patching or alien threads in the application task. In any case, in all of the systems mentioned above userland programs do not see any difference in their runtime behavior, besides speed. For dynamically linked programs even the speed factor can be mitigated.

7 Conclusion

In this thesis we have seen the design and implementation of the OpenBSD operating system on an L4 family microkernel, namely Fiasco. We went through the different kinds of virtualization techniques and figured that rehosting an operating system is a special case of paravirtualization. So we created a new architecture for OpenBSD that is not based on a hardware design, but the operating system interface of Fiasco and its runtime environment. That operating system interface features vCPUs, which provide us with a decent abstraction of real CPUs. The memory abstraction was achieved with a separate kernel task on Fiasco, isolating it from the rest of the OpenBSD system. That has left us with relatively light efforts to rehost OpenBSD onto that abstraction layer. We can even run userland applications for OpenBSD/i386 unmodified on it.

We can separate the rehosted OpenBSD server from the rest of the system by using the capabilities system in Fiasco.

A complete L4 runtime system, containing the leightweight OpenBSD server and a simple stripped down userland fit on an 8 MB CD-ROM image. It boots with only 42 MB RAM altogether. So it is fairly useful for embedded systems. Some features are still missing, though. We will discuss possible future directions of this project in the following sections.

7.1 Future Work

Now that we have a minimal OpenBSD server on L4, where can we go from here? As a start we could compare the performance of Linux on L4 to the OpenBSD equivalent. With that information, we can gain insight on which operating system design decisions are better suited to rehost operating systems. That can be extended to all subsystems of both operating systems. That also means that the minimal OpenBSD server needs to be extended to support networking or a framebuffer based console. We can also prepare the OpenBSD server to be runnable on different hardware architectures. As we have seen on Linux, a lot of code of the rehosted kernel can be shared among the different architectures.

7.1.1 General Performance Improvements

We have taken the i386 architecture port as a basis for our rehosting efforts, as our target hardware architecture was i386, too. Even with the naive implementation of issuing one IPC operation per page table manipulation, we have an overall speed degradation of only about 20 % for real world applications. We could further improve the performance by looking at the costs of each IPC operation and try to change the new L4 port in a way that we minimize IPC calls during kernel operations. We might be able to free

several pages with one call, so we could collect IPC operations of the same kind and issue them as a batch job. That would require changes to the Fiasco interface and to the servers used by the rehosted OpenBSD kernel.

7.1.2 Multiprocessor Support

With the use of more vCPU enabled threads in the kernel task, we can also fairly easily extend the current implementation to support multi-processor setups. All ground-work is already in the system, as OpenBSD itself is capable to handle SMP hardware on i386. We would need a vCPU thread per CPU. These CPUs do not necessarily need to map to physical CPUs in the system. Then we need to change the inter-processor communication (IPI) to synchronize the kernel vCPU threads.

7.1.3 Network Performance Improvements

Another possibility would be to separate performance-critical parts from the OpenBSD server. In contrast to Linux, OpenBSD still has the big kernel lock. That means, the kernel is not preemptive. This especially hits multiprocessor setups. Whenever one processor is in kernel mode and holds the kernel lock, the others need to wait when operating on another part secured by the big kernel lock. That can be a problem when handling high-speed network traffic with a firewall rule-set. In that scenario, one idea is to factor out the lower network layer into a separate server. That small, multi-threaded server can directly interact with the network card and forward packets of the same network flow. It only passes unknown packets for new flows to the pf firewall layer of the rehosted OpenBSD operating system, which can then take the decision on what to do with it. This is much like getting network flow information with a protocol like IP Flow [Cla08] on a monitoring station. The difference is that it features a back-channel, which instructs the packet forwarding server how to proceed with new flows, thus increasing throughput and minimize latency on the network when using that solution on a router. The flow control mechanism might look similar to OpenFlow [MAB+08].

All network management and configuration can remain on OpenBSD, so administrators do not need to be retrained. They retain a familiar interface and can continue using familiar tools.

7.1.4 Enhanced Separation

For high security requirements, there is the possibility to have a small server running separated on the microkernel to provide a distinct service. Access control to that service can be controlled in a fine-grained way with capabilities. In this way we can provide a secured service outside of any virtual machine. If that service should be needed in one of the virtual machines, that machine needs the appropriate capabilities and a special driver to access that server. There is no way for the virtual machine to access or manipulate internal data structures of the separated server. The service can only be accessed via its exposed API functions. With that approach we could implement a store for private keys in a cryptography compartment, which can be used by rehosted

operating system instances. The capabilities to update the keys may not be available to the rehosted operating systems. With that approach, we can build a separate domain for cryptographic operations, where legacy operating systems cannot tamper with the important data.

Glossary

API Abbreviation for *Application Programming Interface*. It specifies the way in which a task sequence has to be performed. In programming, an API is often realized as function calls. It hides away the implementation details and is thus a form of abstraction. Applications should not be able to work around an API, changing the internal state of the task sequence by the use of e. g. global variables.

BSD Berkeley Software Distribution. A derived work from the original UNIX operating system, licensed under the BSD license.

context switch Switching between contexts means that the MMU is reloaded with a different set of mappings for virtual to physical addresses. The current address space switches to a new context. At that time, the TLB has to be flushed on the i386 architecture. Otherwise the TLB will still answer chached lookups from the old context.

GNU GNU's Not Unix is a recursive acronym, describing a project with efforts to implement a UNIX-like operating system and the according userland tools.

ISA Abbreviation for *Instruction Set Architecture*. An ISA defines in which way a CPU must be programmed. It contains the exposed instructions for an application programmer. The closest programming language to the ISA is an assembler. The i386 instruction set exposed by its ISA can be found in the instruction set reference from [Cor10a] and [Cor10b].

MMU Abbreviation for *Memory Management Unit*. In computer architectures, an MMU is usually a hardware device, which translates physical addresses in RAM to virtual addresses and vice versa. In most cases the translation can be performed in one direction only.

TCB Abbreviation for *Trusted Computing Base*. The TCB for a program contains all components which that program needs to rely on for proper execution. That includes the program binary and all used libraries. It also includes the operating system kernel. The kernel is part of every program's TCB, because every program uses kernel services.

TLB Abbreviation for *Translation Lookaside Buffer*. A TLB is a hardware device, which caches a pair of data, usually mappings. By default, these mappings need to be queried from a slower medium. The cache can answer requests quicker. For MMUs, the TLB caches all mappings of physical to virtual memory. If a mapping is needed more than once in a short period of time, the TLB can answer the mapping MMU's request quicker than fetching the results from RAM every time.

toolchain The term describes all necessary software utilities needed for software development. This includes the compiler, debugger, and all supporting software for that process, like the build system tool *make*. GNU features a toolchain that is not compatible to the one in BSD.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural*, pages 2–13. ACM Press, 2006.

[BCE+95] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. Spin – an extensible microkernel for application-specific operating system services. *SIGOPS Oper. Syst. Rev.*, 29:74–77, January 1995.

[BDF+03] 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 nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[Cla08] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[Cor10a] Intel Corporation. *Intel 64 and IA-32Architectures Software Developer's Manual, Volume 2A*. Intel Corporation, June 2010.

[Cor10b] Intel Corporation. *Intel 64 and IA-32Architectures Software Developer's Manual, Volume 2B*. Intel Corporation, June 2010.

[Cor10c] Intel Corporation. *Intel 64 and IA-32Architectures Software Developer's Manual, Volume 3A*. Intel Corporation, June 2010.

[CP99] Charles D. Cranor and Gurudatta M. Parulkar. The uvm virtual memory system. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 9–9, Berkeley, CA, USA, 1999. USENIX Association.

[DBR98] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent 6397242 B1*, October 1998.

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[DIN05] Catherine Dodge, Cynthia Irvine, and Thuy Nguyen. A study of initialization in linux and openbsd. *SIGOPS Oper. Syst. Rev.*, 39(2):79–93, 2005.

[dPSR96] Francoir Barbou des Places, N. Stephen, and F. D. Reynolds. Linux on the osf mach3 microkernel. In *Conference on Freely Distributable Software*, 1996.

[GD91] David B. Golub and Richard P. Draves. Moving the default memory manager out of the mach kernel. In *Proceedings of the Usenix Mach Symposium*, pages 177–188, 1991.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, EW 9, pages 109–114, New York, NY, USA, 2000. ACM.

[GJR+92] David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13:238–241, April 1970.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 31:66–77, October 1997.

[Kam02] Poul-Henning Kamp. Timecounters: Efficient and precise timekeeping in smp kernels., 2002.

[Kun03] Jochen Kunz. Writing drivers for netbsd, August 2003.

[Law99] Kevin Lawton. Running multiple operating systems concurrently on an ia32 pc using virtualization techniques. 1999.

[Lie93] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[Lie95] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[LS01] Peter A. Loscocco and Stephen D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. Virtual processors as kernel interface. In *Twelfth Real-Time Linux Workshop 2010*, 2010.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[Meh05] Frank Mehnert. *Kapselung von Standard-Betriebssystemen.* Dissertation, Technical University of Dresden, 2005.

[MS96] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[MYSI03] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability myths demolished. Technical report, 2003.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[REH07] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 4:1–4:6, Berkeley, CA, USA, 2007. USENIX Association.

[WB07] Neal H. Walfield and Marcus Brinkmann. A critique of the gnu hurd multiserver operating system. *SIGOPS Operating System Review*, 41(4):30–39, July 2007.