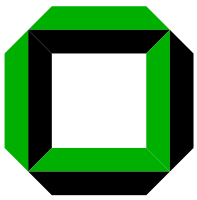

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Improving Operating System Decomposition by Microkernel Design

Sebastian Reichelt

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Jan Stoess

15. Dezember 2008

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

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

Karlsruhe, den 15. Dezember 2008

_____

Sebastian Reichelt

Abstract

There have been numerous attempts to decompose operating systems into multiple servers running on top of a microkernel. Decomposition offers a lot of advantages, such as better security and robustness, flexibility with respect to changing requirements, reusability of servers in different contexts, and maintainability due to clearly defined roles and interfaces. Yet, multi-server operating systems often turn out to be significantly more complex than their monolithic counterparts. This architectural overhead increases disproportionally to the number of components, thereby imposing a limit on the achievable granularity.

A major factor is the programming model which all microkernels explicitly or implicitly enforce on servers designed for them. A server always consists of both the code that performs its particular task within the operating system, and the glue code that maps the nature of the task onto the concepts of the underlying microkernel. If a multi-server OS turns out to be more complex than a monolithic OS performing the same tasks, then all of the additional complexity stems from the necessary glue code.

We postulate that fine-grained decomposition can be achieved by defining a suitable programming model for servers, designed in a way that minimizes the amount of glue code required. The key to this approach is our definition of servers as light-weight, universal components, which, instead of glue code, contain additional information describing their role and interaction. In this thesis, we build a prototype multi-server OS to evaluate in how far the model improves operating system decomposition.

The results are largely positive: The programming model has turned out to be suitable for various different types of system components. The components in our system are as fine-grained as possible from a technical point of view. Although a direct translation of existing interfaces to our programming model can be problematic, existing code can feasibly be reused when adapted to our own interfaces. However, the performance of our system is not satisfactory yet, possibly requiring modifications to the programming model.

Acknowledgments

I would like to thank all people who made this work possible with their knowledge, support, encouragement, and patience; especially my supervisor Jan Stoess, Matthias Nagel, and my parents.
Special thanks to James McCuller for quickly fulfilling all of my hardware and software requests.

Contents

Chapter 1

Introduction

The kernel of an operating system is its central piece of software, which is "mandatory and common to all other software" [32]. As such, its correct, robust, secure, and efficient implementation is critical to all other software. In a system designed for multiple programs or users operating independently from each other, this implies that it at least needs to manage and partition the available hardware resources.

The most straightforward way to restrict each program's allowed operations is to implement an abstraction layer on top of the hardware, such that the operations of this layer do not allow any uncontrolled interference between programs and/or users. Traditional operating system kernels include abstraction layers for most pieces of hardware, such as processes and threads for memory and CPU time, files for storage space, sockets for networking, etc. These kernels are called "monolithic;" their use is still very common.

The implementation of high-level abstractions generally makes modern monolithic kernels very large, which can render them inflexible (e.g. with respect to changing abstractions) and insecure (because a single error in the kernel can compromise the entire system) [32]. One alternative approach is that of a microkernel, which provides lower-level abstractions than traditional kernels and is therefore smaller and more flexible. In a microkernel-based system, higher-level abstractions can be implemented by "servers," which do not need to be part of the kernel as long as the security guarantees of an equivalent monolithic kernel still hold for the "multi-server" system.

In a monolithic kernel, a single error can compromise the security and robustness of the entire system. In a multi-server system, an error in a server is not a security issue, unless the server itself is critical for security. Moreover, it affects only those other servers that depend on the failing server's correct operation. Only errors in the underlying microkernel are necessarily critical for the security of the entire system. The possibility of such errors can be reduced over time because of a microkernel's small size and relative immutability [33], or the kernel's correctness can even be verified mathematically [15].

Like regular programs, servers are able to interact via mechanisms provided

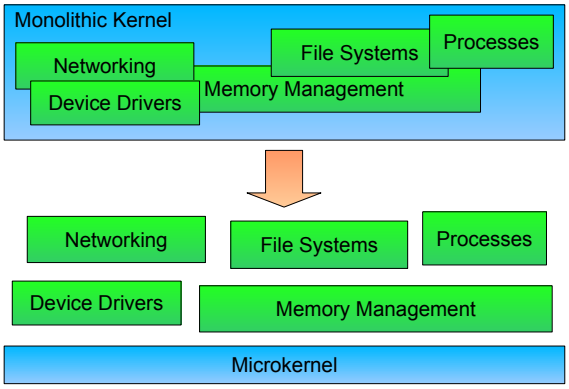

Figure 1.1: Monolithic vs. multi-server operating systems

by the kernel. In its most rigorous form, a microkernel implements little functionality beyond secure interaction. If all interaction of a server with its outside world is managed by a central entity, the server is effectively a reusable, separately maintainable component (see figure 1.1). Thus, a modular system structure is guaranteed – a property which is usually difficult to achieve.

Both the possible effects of errors and the potential for reuse in a changing environment depend on the size of individual servers, i.e. on the granularity of the system. However, a fine-granular system implies a high amount of interaction between servers, the performance cost of which is not zero. In the so-called "first-generation" microkernels, every interaction was rather expensive, enforcing a coarse-grained system structure and even compromises to the microkernel concept. "Second-generation" microkernels have been designed with the primary goal of reducing this overhead [31], so that fine-granular multi-server systems are feasible today, from this perspective.

Several microkernel-based operating systems have been developed from scratch, for example MINIX [24], QNX [29], Hurd [12], and K42 [5], with varying degrees of granularity and rigor. However, the idea of decomposing an existing monolithic system into multiple servers, as put forward by the SawMill [21] project, was not met with equal success so far. Since novel operating systems can gain acceptance only by supporting a wide range of existing hardware, software, and standards [37], reuse of existing pieces of software is a crucial requirement.

1.1 Problem Definition

Kernel programming tends to differ from application programming in many aspects; most prominently the widespread use of hardware-related low-level operations, but also the specifics of memory management, multi-processor support, stack space availability, module loading, etc. Since many of these aspects are consequences of the fact that the code is executed in the processor's kernel mode, server

programming on a microkernel does not necessarily inherit the differences; i.e., server programming can often be more similar to application programming than to kernel programming [4]. While this similarity is generally considered a positive facet of microkernels, it also poses two problems:

- Existing kernel code cannot be converted verbatim into server code. In some cases, the kernel-specific peculiarities are simply not necessary in user mode, but in other cases (e.g. multi-processor support), they need to be taken into account in a different manner.
- In a kernel, all code is able to access the entire kernel data directly, as well as (usually) all of the data in the address space of the program which executed a system call. For a server running in user mode, the latter is never true, and the former is true only for the code and data of a single server. In other words, even though the conversion of existing kernel code into a *single* server has been proven to be feasible (see section 2.5), decomposition into *multiple* servers is a separate issue (see section 2.3.2). Without such decomposition, however, much of the purported security, robustness, flexibility, and maintainability of microkernel-based systems is lost.

Since these two problems are inherent in the transition from a monolithic kernel to a multi-server system, solving them completely would be a utopia. Instead, the goal of this thesis is to explore ways to alleviate them, in light of the chance that the benefits of a multi-server system will outweigh the remaining difficulties.

1.2 Problem Analysis

The decomposition of existing kernel code into multiple servers can be regarded as a combination of two separate processes: First, the existing code is stripped of all immediate dependencies on the rest of the kernel, including hardware-specific low-level operations, special characteristics of the kernel, and also other modules which are intended to be decoupled into different servers. Second, the result is converted into an actual server according to the underlying microkernel. In other words, it is adapted to the microkernel's "programming model," which we define as everything that distinguishes servers developed for different microkernels. While the former process fully reflects the inherent problems in the transition from a monolithic to a multi-server system, the latter largely depends on the details of the programming model, and therefore on the specific microkernel in use.

These two processes are never separate from each other. In fact, their intermediate product – the kernel code free of all dependencies – is not even a tangible entity, since effectively all machine-readable code (as opposed to pseudocode) depends on a programming model of some sort. (For example, for a regular application, the programming model consists of the APIs and OS properties it depends on. The distinctive features of a specific monolithic kernel can also be regarded as a programming model.)

As the programming models enforced by existing microkernels are governed by the kernels' capabilities and specialties, they are usually not very generic, as evidenced by the observation that the transition from one microkernel to another essentially entails a rewrite of the entire operating system [1]. A generic programming model is desirable in order to shorten the path between the (abstract) dependency-free code and the actual servers. A shorter path implies less development effort for the decomposition of existing code, as well as a less complex result.

Therefore, we aim to define a microkernel API in such a way that the resulting programming model is as generic as possible. The code of each server developed according to this API should only solve the particular task for which the server is designed, and be largely free of microkernel-specific programming paradigms.

Existing microkernels do not provide a completely satisfactory solution because their server programming models are a consequence of their APIs, not vice versa. The corresponding server code needs to solve two separate problems at once: the actual task of the server, and the concrete realization of that task in terms of the microkernel API. Fine-grained decomposition on such a microkernel is a tedious process because each individual component must be reshaped to match the server model.

Our solution is essentially the design of a new server model, without direct references to microkernel-specific concepts, but with a consideration for microkernel-specific requirements. The primary requirements we have identified – server interaction, object identification, and object lifetime management – directly lead to the abstractions we use in our server model, which are "servers," "services," "service calls," and "server references."

Our servers are light-weight passive objects that can be called in a very natural fashion (unlike the process model used in most microkernels). The programming model is rich enough so that most of the programming paradigms of modular monolithic systems can be realized, but close enough to the hardware to be usable as a microkernel API.

To evaluate our model, we have built a prototype kernel and multi-server OS. We can show that fine-grained decomposition is possible with relatively little additional effort. For conceptual reasons, a direct conversion of existing interfaces from monolithic code to our server model is not always possible. However, we are able to reuse large parts of existing code by adapting it to our own interfaces. The performance of our system still needs to be improved, but calculations indicate that the large overhead we experience is not a direct result of the server model.

In chapter 3, we will describe our analysis of the requirements on server models in general, and then develop an actual model that meets these requirements as generically as possible. In chapter 4, we will describe the concrete system we have built to evaluate the model. Finally, in chapter 5, we will analyze the model with respect to our goals.

Chapter 2

Background and Related Work

In this chapter, we will present the concepts behind microkernels and multi-server operating systems, as well as concrete examples that are relevant to our design.

2.1 Microkernels

Fundamentally, the concept of an operating system "kernel" is a consequence of processor design. In most common processor architectures, the processor knows (at least) two modes, one of which is the "kernel" or "supervisor" mode. Code that runs in kernel mode is "trusted" because it has access to the entire set of processor features. The kernel can restrict the privileges of "user" (i.e. non-kernel) code, but to do so, it needs to be fully privileged itself.

Operating system kernels must contain all of the code that is necessary so that different user "applications" can interfere with each other only via controlled mechanisms. Especially, this property can be ensured only if access to all hardware is managed and controlled by the kernel. Traditionally, this requirement has led to kernels that include device drivers and file systems as means of multiplexing the hardware, and that implement complex abstractions to manage user-level code.

Since all kernel code runs with full privileges, each driver and each feature of such a "monolithic" kernel can compromise the operation and security of the entire operating system. Moreover, different OS kernels typically provide different abstractions to user applications, and the abstractions used by a general-purpose kernel may not fit the needs of a special application.

The microkernel approach is an attempt to solve these problems by executing as much code as possible in user mode instead of kernel mode. For example, device drivers and file systems, but also other kernel components, do not necessarily require full privileges. Instead of directly managing and controlling all hardware, a microkernel effectively delegates this task to individual user-mode "servers."

2.1.1 Mach

Mach [4] is a popular early-generation microkernel developed at the Carnegie Mellon University in Pittsburgh. Originally, it was designed as a single operating system kernel compatible with BSD Unix, with additional support for multiple threads within an address space, as well as asynchronous, buffered IPC (inter-process communication) facilities. It can be described as a microkernel because the BSD compatibility layer was largely separate from the rest of the kernel, and was moved to user space in later versions of Mach [40].

Mach introduced the concept of implementing core system services, such as tasks, files, and networking, as servers in user space [40], similarly to daemons in a Unix system. Servers can also act as external pagers for memory objects [39], which allows for swapping and memory-mapped files to be implemented at user level. However, device drivers are integrated into the kernel, which distinguishes Mach from many later microkernels.

Communication between different servers in Mach is controlled by capabilities, called "ports." Since a server cannot interact with any other server without possessing a port to it, this model also provides the foundation for the confinement of servers, i.e. for the implementation of security policies. Our system follows a similar approach: "server references" are capabilities to a set of functions implemented by a server. A major difference is that Mach IPC is unidirectional, requiring a special reply port for the return message of a remote procedure call, whereas our communication model is based directly on function calls, and returning to the caller of a function is an integral part of the model.

There exists a research effort by the University of Utah to extend Mach into a similar direction, by introducing support for a "migrating" thread model, in which a thread that performs a remote procedure call transitions into the called server [20]. This work resulted in substantial performance improvements and a simpler system interface. However, since the migrating thread model was introduced into a system originally based on static threads, it also caused conflicts with existing user-level thread code. In our system, we employ a migrating thread model because our communication primitives are derived from function calls, which are normally not associated with thread switches. Special user-level thread code is not required because the kernel is able to provide concurrency to servers in a transparent fashion.

Variants of the Mach microkernel are in use today as the foundation for the GNU Hurd [12] and Apple Mac OS X [34] operating systems.

2.1.2 L4

L4 is a microkernel designed to be as small as possible, to achieve the desired performance and flexibility, and to be able to ensure correctness [33]. The first version was developed by Jochen Liedtke at GMD Germany in response to the slowness and inflexibility of Mach. Since then, several microkernel variants based on the L4 API have been created for different environments.

The main argument for making the microkernel as small as possible is the cache footprint of common microkernel operations. To further reduce this footprint compared to Mach, L4 uses synchronous, unbuffered IPC, as opposed to asynchronous, buffered IPC in Mach. This concept has influenced later microkernels such as the one used in MINIX 3 [24].

Since our function-call-based communication is inherently synchronous, we can hope to achieve a similarly small cache footprint. One important difference is that in our case, the call and return paths are entirely separate, which, in theory, can double the overall communication-related cache footprint. However, in terms of code, the return path is substantially simpler than the call path, and in terms of data, there is a large overlap with the call path.

As opposed to L4, our kernel is not designed to be as small as possible. We actually implemented parts of our API on top of L4 before deciding to write our own kernel, so we know that a larger kernel is not required in principle. In practice, an L4-based implementation of the API is slower than a direct implementation, as all communication must be managed by a central entity – which is not a problem if that entity is the kernel. Therefore, we possibly face a tradeoff between performance and kernel size. We declare that for most purposes, a small kernel is secondary to an efficient kernel.

We do share another, similar goal with L4: the desire for a policy-free kernel. Every policy that is fixed in the kernel limits the types of systems that can be built on top. The original L4 kernel made two compromises in this respect: It contained a fixed scheduler [42], and it did not support capability-based security. In our system, all servers, including schedulers, can run either in kernel mode or in user mode. Moreover, user-mode servers can access other servers via unforgeable server references only, which means that server references are capabilities.

One of the elementary features of L4 is the recursive construction of address spaces. Threads can send page mappings to other threads via IPC, and later revoke them asynchronously in a recursive fashion. Each thread has a pager that handles its page faults by mapping the corresponding page, or by asking another task to map the page.

Our system supports the same mapping functionality as L4, but instead of transferring mappings from one address space to another, servers attach "data spaces" (similar to those in SawMill [7]) to their own address spaces. Data spaces are implemented by supplying data to handle page faults, but the data originates from another data space instead of the address space of the pager. This concept also exists in the Grasshopper [13] operating system under the name of "containers."

2.1.3 Pebble

Pebble [11] is a component system designed for embedded systems. Pebble components are intended to be fine-grained and isolated; therefore cross-domain communication performance is critical. For this reason, communication between components is implemented using dynamically generated "portal" code. This code

switches the current protection domain and passes data between the domains, but does not change the current thread.

Our communication primitives are similar to those in Pebble in that conceptually, a cross-domain call does not involve a thread switch. Moreover, the primitives in both systems are designed so that the kernel can generate efficient communication code when servers are loaded. The details are somewhat different: For example, Pebble components can transfer memory mappings, whereas our servers can transfer server references and raw data only. While Pebble employs a single scheduler, schedulers can be stacked hierarchically in our system.

The main difference between our design and existing microkernel APIs is that we focus on the programming model implied by the API, not the features of the API itself. The capabilities of our system do not necessarily extend beyond what already exists. We do, however, believe that our system is the first where the server programming model has been designed from the ground up, resulting in very generic, light-weight, reusable servers.

2.2 Extensible Kernels

Technologically, extensible kernels are similar to microkernels in that system services are lifted from kernel mode to user mode, or are at least replaceable. The difference is mainly conceptual: While microkernels are designed to build an entire operating system out of servers, extensible kernels aim to give as much control as possible to individual applications.

2.2.1 Exokernels

The term "exokernel" was coined at M.I.T. in response to first-generation microkernels. The goal of the exokernel approach is to securely multiplex the underlying hardware in a way that allows for user-level "library operating systems" to manage it according to their own special requirements [17]. Instead of communicating with servers that implement high-level abstractions such as files and networking, a library OS accesses the hardware at the lowest level at which secure multiplexing is possible.

Compared to microkernels, exokernels follow the goals of flexibility and performance in a stricter sense, according to the statement that "mechanism is policy," i.e. that any kind of abstraction restricts implementation strategies for its users. For this purpose, however, exokernels must include drivers for all of the hardware that is to be multiplexed. In addition, complex software support is necessary for hardware that does not lend itself well to multiplexing. An example is a user-extensible network packet filter, which, for each packet, determines the correct library OS to forward it to.

Although exokernels provide functionality on which IPC primitives can be built, existing library operating systems are monolithic. Decomposition of oper-

ating systems on top of an exokernel has not been studied. Instead, the exokernel approach as a whole is more similar to virtualization (see section 2.5), since the interface offered by an exokernel resembles the hardware interface.

2.2.2 SPIN and VINO

The SPIN [9] and VINO [16] kernels more closely resemble monolithic kernels but are extensible by user-level code, under the assumption that a fixed kernel is inappropriate in many situations. SPIN loads extensions into the kernel, but uses a type-safe language with built-in threads to ensure that malfunctioning extensions cannot crash the rest of the system. VINO extensions are regular C++ code but secured by a trusted sandboxing compiler and a run-time transaction system in the kernel to recover from crashes.

From a design point of view, there is little difference between servers running at user level and kernel extensions. After all, even kernel extensions must be programmed according to some low-level kernel interface, which fundamentally corresponds to a microkernel API. To us, the question whether an extension or server is loaded at user level or at kernel level is secondary, as long as a system is properly decomposed.

However, the concepts of microkernels vs. extensible kernels do exhibit differences in terms of rigor with respect to decomposition: In a microkernel, all parts of a system are replaceable and logically independent from each other (although the granularity varies). In an extensible kernel, the typical application/kernel boundary remains, except that applications are able to replace or augment certain parts of the kernel. Thus, the modularization aspect is not part of the overall system design, in favor of application-specific code.

2.3 System Decomposition

Like every large program, an operating system is usually composed of vaguely independent parts. An OS-specific natural boundary is the user/kernel mode interface (although the terms “kernel” and “operating system” are sometimes used synonymously). Kernels are additionally divided into independently compiled modules. For example, in the Linux kernel, these modules can also be loaded dynamically at run time.

Since their interfaces are not formalized, such modules can usually not be reused across different operating systems. Moreover, data structures may be accessed from several modules, which implies that modules generally cannot be loaded into different protection domains. Therefore, systems which use such an approach are not fully decomposed from our point of view.

There exist several approaches towards decomposing systems in a stricter sense. Two of them are especially interesting for this thesis:

2.3.1 OSKit

OSKit [19] is an operating system framework created at the University of Utah, with the goal of easing the development of new operating systems. For this purpose, it contains reusable code covering many aspects of OS programming, concerning both hardware support and generic low-level OS functionality.

The majority of OSKit code consists of device drivers and other modules extracted from general-purpose operating systems such as FreeBSD and Linux. These modules are converted into a common format, to facilitate reuse in new operating systems. User-visible interfaces are largely based on COM [36], with only a few additional plain functions. The infrastructure required by the module implementations is provided by core OSKit libraries.

Our system shares some common goals with OSKit, particularly the intention of extracting and modifying components from existing operating systems for the purpose of easy reuse. Compared to servers in existing multi-server operating systems, OSKit components are much closer to the original kernel modules, which simplifies their integration and the exchange of future updates. In this respect, servers in our system are similar to OSKit components.

The most important difference is that OSKit components are not designed with isolation in mind. For example, they cannot automatically communicate via address space boundaries, and some aspects of the interfaces such as explicit reference counting assume a trust relationship between a component and its users. OSKit components assume that they are able to perform basic operations up to low-level I/O via the core OSKit functions. Concurrency and interrupts are handled in the same fashion as in the original monolithic systems from which the components were derived. For all of these reasons, OSKit components cannot be isolated from the rest of the system like servers in a multi-server OS.

2.3.2 SawMill

The SawMill [21] project at IBM Research developed and tested a methodology for the decomposition of an existing OS into a multi-server system on top of a microkernel. The project defined an architecture for a multi-server OS, and explored methods for reducing the number of cross-domain calls and the amount of data copied between protection domains.

The SawMill design decisions were evaluated by converting the Linux kernel into a server running on top of the L4 microkernel, and then partitioning the file system and IP network layers according to the SawMill principles.

Decomposition according to SawMill differs significantly from the form of decomposition presented in this thesis. In SawMill, decomposition was tackled from a full-system perspective, with a strong focus on L4-based multi-server OS development. Consequently, the main result was a multi-server architecture designed to support efficient partitioning, whereas the individual servers were secondary. In contrast, our approach focuses on extracting individual components from existing

operating systems, in such a way that they can work together even though they are protected from each other. The overall system architecture, though important, does not reflect on our individual servers.

This difference follows from the fact that our goals do not exactly match those of the SawMill project. The idea behind SawMill was to decompose an entire existing system while preserving the internal data structures and interfaces. Therefore, one of the major issues, especially in terms of security, was "control data" accessed by multiple servers. In our thesis, we aim for reuse of existing kernel components. Since we consequently adapt the components to our own interfaces, this particular problem does not arise. In fact, control data partitioned among multiple servers would significantly impact reusability of these servers.

As a result of the different goals, the SawMill approach mainly consists of *mechanisms* that enable decomposition, whereas our approach focuses on *abstractions* for decomposition.

Nevertheless, some insights from the SawMill projects are applicable to our system. For instance, when designing our interfaces, we make sure that it is possible for a client to obtain direct connections to servers handling its requests, instead of having to make calls through multiple layers of abstraction. Our system supports this paradigm especially well because servers can return secure references to other servers. For example, when a client requests a file via a virtual file system server, that server can pass the request on to the physical file server, and return the resulting server reference. As a result, the virtual file system server is no longer involved in actual file accesses.

2.4 Multi-Server Operating Systems

Several microkernel-based multi-server operating systems have been developed from scratch. An advantage to the decomposition of existing systems is that multi-server principles can be applied more rigorously.

2.4.1 Workplace OS

The IBM Workplace OS [26] was a general-purpose end-user operating system designed as a replacement for several other IBM systems. A multi-server structure was chosen in order to support multiple OS personalities, and as a means to dividing the system into separately marketable parts. However, despite substantial development costs, the system was never sold in significant numbers.

Workplace OS was based on a heavily modified Mach microkernel [18]. In addition to microkernel-based separation, it made use of fine-grained C++ objects within servers and within the kernel. This practice resulted in both complex and inefficient code; the available reports [18, 26] recommend against doing the same in future operating systems.

Since we aim for both fine-grained decomposition and the separation of personality-specific code from the rest of the system, our system is at risk of repeating the mistakes that led to the failure of Workplace OS, in theory. Therefore, we will analyze the differences between our approach and the approach taken by the Workplace project:

- We decompose systems into fine-grained servers with well-defined interfaces. Ideally, each of them provides a specific piece of functionality only, whereas all of the dependencies on other code are reflected in its interface. In particular, servers never depend on the existence of a common library.
- After developing servers in this manner, we can load them into different address spaces, into the same address space, or into the kernel. This way, we hope to avoid implicit performance limitations in the code.
- Our interfaces can be reworked and adapted over time. Since each server has relatively few dependencies, the resulting changes are always confined to a small subset of other servers. In the Workplace OS, the C++ library could not be modified easily due to its ubiquitous use.
- Likewise, a problem with Workplace OS was that specific requirements of all personalities, such as memory management and file system semantics, needed to be implemented in "personality-neutral" code. In contrast, our system does not depend on a fixed block of such personality-neutral code. Due to fine-grained decomposition, OS personality servers can be supplied with server implementations that fit their needs exactly, if required. For example, if a personality places special restrictions on the file system, it can depend on a specialized file system interface, which is then backed by a specialized file system implementation. (Obviously, personality-neutral code should be favored if possible.)
- Virtualization (see section 2.5) has become a widespread method of supporting multiple OS personalities on a single machine, which shows that multiple personalities are not a problem per se. The main difference is that the interfaces are closer to the actual hardware, so that the incompatibilities that were difficult to accommodate in Workplace OS are not visible. With fine-grained decomposition, we can freely choose the level of interfaces to support – including hardware interfaces, if necessary.

To conclude, we still believe that fine-grained OS decomposition and multiple concurrent personalities are feasible. The specific path taken by the Workplace project – the development of a complex support layer on top of Mach, instead of individual simple components – appears to have been responsible for the difficulties. Without the Workplace OS source code, we cannot prove this claim. However, reports strongly suggest that the Workplace OS code was overly complex because of the overall OS structure. In contrast, we design our server programming model specifically so that the code of each server can be written to fulfill one particular task, independently of the global operating system structure.

2.4.2 Hurd

The GNU Hurd [12] is a multi-server OS designed as a Unix kernel replacement. It is based on Mach, though several attempts have been made at porting the system to different microkernels for performance reasons [1].

The main reason for choosing a multi-server architecture was the desire to extend the file system hierarchy without special privileges. In a monolithic kernel with built-in file systems, an error in a file system can potentially crash the system or compromise security; therefore file system operations usually require special privileges. In contrast, if file systems are implemented as user-level servers on a microkernel, they can be isolated so that all file system operations can be made available to users without introducing any security issues.

Although the Hurd achieved this particular goal, current developers acknowledge that the system suffers from the inefficiency of the Mach kernel, and that Mach also has a number of other shortcomings that severely hinder development of the Hurd as planned [43]. Without going into detail, it is evident that the Hurd, too, emphasizes on the entire system structure instead of the tasks carried out by individual servers. Therefore, our remarks about Workplace OS should be equally valid for the Hurd.

2.4.3 K42

K42 [5] is a prototype operating system developed by IBM Research in collaboration with several universities. Its goals are diverse, ranging from high-level objectives such as customizability for future research, to the solution of concrete operating system problems such as multiprocessor scalability.

The design decisions of K42 closely follow its goals. For example, an object-oriented microkernel-based multi-server system design was chosen to improve customizability, since servers can be replaced more easily than parts of a monolithic kernel. For good performance especially on multiprocessor systems, K42 employs a new "clustered object" concept. Such objects are automatically distributed across servers based on dependencies between object accesses. The IPC system is closely tied to a C++-based interface system, following the goal of an object-oriented design; in particular, communication is based on a client/server model using C++ function calls.

K42 has a built-in process concept, which is tied to its memory-management model. Processes can be either K42 servers or programs designed for another operating system; in particular, K42 is designed for binary compatibility with Linux applications.

All of these decisions heavily influence the server programming model of K42. For example, servers have to be aware of the clustered object concept, which cannot be found in any traditional OS – whether monolithic or microkernel-based. More strikingly, the fact that both K42 servers and Linux programs are essentially handled in the same manner leads to the requirement that both types of processes are

compatible in certain fundamental ways. A concrete instance of such a compatibility problem has been observed in the Linux multi-threading implementation [27].

Thus, while K42 is a notable example of a large multi-server system with a wide range of features, its programming model is far from generic, but rather dependent on very specific design decisions, limiting reusability. To us, it shows that in order to achieve our goal of a server programming model that is as generic as possible, we need to focus on the programming model itself, instead of starting with a list of design criteria for a particular system.

As a consequence, the resulting multi-server system may not be particularly outstanding when measured according to the goals of K42. However, we hope to arrive at a very cleanly and fine-granularly decomposed operating system, which can greatly increase the chance that individual parts are reusable in different contexts.

2.4.4 MINIX 3

MINIX 3 [24], developed at Vrije Universiteit Amsterdam, is a POSIX-compatible multi-server system designed towards the goal of reliability. A microkernel-based design was chosen to prevent bugs in drivers and other traditional kernel components from crashing the system. In particular, failing servers can be restarted transparently to the rest of the system.

Essentially, the discussion about K42 applies to MINIX as well: For servers to be restarted transparently, their state (as well as the global system state associated with them) must survive such a restarting operation. Most types of system components are inherently stateful, and there is no canonical representation of their state (for example, consider open files in a file system implementation). Therefore, servers must be programmed specifically to support state restoration. In other words, the transparent restart feature becomes an integral part of the server programming model.

Decomposition in MINIX is less fine-granular than what we would like to achieve, due to the fact that MINIX servers are processes which need to communicate explicitly by sending and receiving messages. Current servers are not multiprocessor-aware because each of them executes a single event loop to process incoming requests. However, contrary to earlier versions, MINIX 3 supports user-space drivers, each in a separate address space, without any prohibitive performance overhead.

2.4.5 Singularity

Singularity [25] is a Microsoft Research operating system prototype aiming for dependability. It is essentially a multi-server system, except that servers are isolated using language-based (and thus, software-based) techniques instead of hardware mechanisms. The use of a safe language simplifies component development and

also improves the performance of cross-domain calls compared to hardware isolation.

Compared to our decomposition approach, language-based isolation is certainly more rigorous, and in some sense superior. In Singularity, protocols between communication partners are formalized in a machine-readable format, so that the implementation can be verified at compile time. However, using a fixed, specialized language has the strong drawback of not being able to reuse existing code in any straightforward way.

Even in Singularity, all system components are processes which communicate explicitly. When developing system components using a specialized language, the fact that every component is a process may not be directly visible. Still, processes are different from simple objects in several aspects such as threading and callbacks. This fact always needs to be considered, for example when designing communication protocols. It undoubtedly complicates Singularity's programming model, though we cannot assess the exact consequences due to the limited amount of information available.

2.5 Virtualization

We mention virtualization because in technical terms, there are many similarities between virtualized operating systems and multi-server systems. Most notably, guest operating systems run in isolated protection domains (in user mode or a dedicated hardware virtualization mode). Various projects, including several L4-based virtualization approaches, have shown that microkernels can be used as virtual-machine hypervisors [22, 30]. Similarly, some hypervisors feature "hypercalls" resembling IPC, blurring the distinction between microkernels and hypervisors.

Thus, virtualized operating systems can be regarded as large servers, even if they were originally not written that way. This is especially apparent in the case of paravirtualization [22]: When an existing kernel is ported to run on top of a microkernel, it is essentially adapted to the server programming model of the microkernel.

Virtualization fully meets one of our goals: The only way a guest operating system can interact with its environment is via a well-defined interface: the hardware interface, which is the only interface that all operating systems adhere to. At the same time, virtualization completely avoids decomposition. To bring both goals together, the hardware interface is not sufficient; therefore we need formalized software interfaces between our components.

2.6 Summary

Operating system decomposition has been the subject of many research projects. Roughly, past approaches fall into one of three categories:

1. **Source-code level decomposition.** Most major operating system kernels are developed similarly to large applications, i.e. partitioned into modules that interact via more or less well-defined interfaces. Often, such modules can be loaded at run time (e.g. in the Linux and Windows kernels).

   If the traditional static or dynamic linking method is used, dependencies between modules are implicit – any module can depend on functions or variables implemented in any other module. In an operating system kernel, this includes low-level functionality such as resource acquisition, paging, I/O, interrupt handling, timing, concurrency, synchronization, and debugging. In addition, any module may depend on higher-level kernel data structures such as files and processes.

   The OSKit project falls into this category as well, except that interaction is managed by COM interfaces. While the use of COM ensures that interfaces are well-defined and thus components are exchangeable, the major limitation of source-code level decomposition remains: All modules or components must be loaded in such a way that they can share code and data addresses (i.e., into the same address space, or into the kernel).

2. **Wrapping of individual modules.** Since every module or component has both a source-code and a binary interface defined by its external references, extracting and reusing individual modules/components from a larger operating system kernel is possible in most cases. Examples include the use of Linux device drivers in Hurd [2] and K42 [6] (at the source code level), the NDISWrapper Linux module to load binary Wireless LAN drivers written for the Windows kernel [3], and an L4-based project to reuse Linux binary drivers [38], among many others. Unless the wrapped module contains privileged processor instructions, the module and its wrapper can generally be executed at user level, in an isolated address space.

   Since this kind of decomposition involves developing a separate wrapper for every different module interface, it is suitable only in cases where either several modules implement and use the same interfaces, or the module in question implements complex functionality behind a simple interface. Otherwise, the benefit of module reuse is quickly outweighed by the effort required to implement the wrapper.

   Virtualization is the use of the same technique at a lower level. In that sense, the guest operating system takes the role of the module being wrapped, and its interface is defined by standardized hardware behavior. Although virtualization does not contribute to system decomposition, its success shows that this approach to reuse is well understood.

3. **Full decomposition.** Decomposing an entire kernel into isolated, self-contained components requires major source code modifications. In general, the interface of every module must be redesigned to replace all implicit assumptions about the rest of the system by explicit communication code. Contrary to an interface in a monolithic kernel, the communication code

must differentiate between data that is copied or mapped, and mere references to objects (which are then subject to further communication).

This type of decomposition was attempted for the Linux kernel in the L4-based SawMill project. The same interface design issues also apply to multi-server operating systems developed largely from scratch, such as MINIX, Hurd, and K42.

Major benefits of a fully decomposed system over wrapping of individual modules are better maintainability and a lower barrier to further modification and extension. Essentially, the source code modifications can be regarded as a port from the original system to a new multi-server OS. However, the modifications have to be repeated for every component. Moreover, since the details of communication and data transfer depend on the specific microkernel in use, and additional assumptions about the overall system structure are introduced into the source code, reusability is generally not improved.

Our approach to system decomposition can be described as a combination of all three variants. The goal is to arrive at a fully decomposed system with as few source code modifications as possible, and without the reusability problems caused by dependencies on the microkernel and the system structure.

For an existing operating system that is divided into modules, module wrapping requires the fewest source code modifications, if any. The reason why module wrapping does not lead to a fully decomposed system is that a separate wrapper is needed for every different type of module. We propose making minimal changes to the module interfaces such that the modules can be wrapped and combined with other modules automatically.

This involves converting the modules to components with well-defined interfaces (akin to OSKit), under the additional constraint that components may reside in different address spaces and are generally untrusted. Moreover, the interfaces must be defined in a way that enables the system to "wrap" the components automatically, i.e. to manage communication and data transfer between components, as well as other inter-component aspects such as security and component lifetime. In effect, this means that we design a microkernel API, except that we start with an intended server programming model and derive all API elements from this model.

Chapter 3

Design

Every microkernel implicitly defines a programming model for server code that is influenced by the features of the kernel. When existing operating system code is converted into servers, it must be adapted to this programming model, resulting in additional development effort and code complexity – which is acceptable for a single server but places a limit on fine-grained decomposition into multiple servers. To alleviate this problem, we define a generic server programming model that is not based on the features of any particular microkernel but on constructs used in monolithic kernels. After that, we build a kernel according to the server model.

3.1 Analysis

A central concept of our thesis is the "programming model" of operating-system code. By "programming model," we essentially mean all aspects of the code that are unrelated to the problem the code is intended to solve. More specifically, all machine-readable code is embedded in an environment consisting at least of the programming language and libraries in use. In a monolithic kernel, the specific kernel conventions and features are part of the programming model, and in a multi-server system, the model is governed mainly by the features of the microkernel in use.

As a result, the programming models of monolithic kernel code and multi-server code differ substantially, leading to the aforementioned problems concerning the fine-grained decomposition of existing code. We aim to bridge this gap by defining a new server programming model. Consequently, the first step has been to analyze the minimum requirements on such a server model.

The main requirements we have identified are server interaction, object identification, and object lifetime management. The abstractions of our server model – "servers," "services," "service calls," and "server references," as described in section 3.2 – are directly derived from these requirements. In the following sections, we will describe the requirements, and how they relate to monolithic kernel code.

3.1.1 Server Interaction

Interaction between servers is perhaps the single most important aspect of a server programming model, both because it is a frequent operation (especially in fine-granular systems) and because existing microkernels exhibit a lot of diversity in this respect (see chapter 2). In general, servers need to be able to

1. cause the controlled execution of code within another server,
2. transfer a potentially large amount of data between each other,
3. retain state between interactions (consider, for example, a server querying another server for information, and resuming its current operation when that information arrives), and
4. accept interaction initiated by other servers in the case described in point 3.

Traditionally, the answer to points 1 and 2 has been that servers pass messages to each other, in a format defined by the microkernel. Such messages, however, are neither part of the programming model of any monolithic kernel, nor defined in a universally accepted fashion. Point 3 is usually achieved by making the operation block until the result arrives, so that all information stored on the stack is still available afterwards. Since this solution matches the way function calls and local variables are handled in all modern programming languages, it is very natural and thus a good candidate for our programming model as well. On the other hand, if point 3 is solved in this way, existing microkernels require the explicit use of threads for point 4, contrary to monolithic kernels.

3.1.2 Object Identification

A less obvious commonly used mechanism is the identification of objects (where the term "object" is meant as a placeholder for anything that can be identified, such as a data structure). To clarify this, we first consider monolithic kernels, to observe that the most common concept for object identification is that of a pointer.

In fact, pointers serve a twofold role. The main reason for their existence is to be dereferenced, in order to examine or modify the data they point to. However, according to the principle of modularity, such use of a pointer should be restricted to a single module (and this makes the module a natural candidate for conversion into an individual server). At other places, pointers are not dereferenced but used to identify the object they point to. Consider the following (hypothetical and abbreviated) example:

```
File *openFile(const char *name);
void readFile(File *file, ...);

Process *createProcess(File *file)
{
    readFile(file, ...);
    ...
}
```

```
void runProgram(const char *name)
{
    File *file = openFile(name);
    Process *process = createProcess(file);
    ...
}
```

The `file` pointer is never dereferenced in the `createProcess` and `runProgram` functions, but passed as an opaque argument from `runProgram` to `createProcess` and finally `readFile`. Such use of pointers is common in regular applications as well as kernel code, since in a single address space, the address of an object uniquely identifies it – except that the programmer manually needs to ensure that no pointer references an object that no longer exists.

If the hypothetical operating system of the example above is decomposed such that `runProgram`, `createProcess`, and `readFile` end up in different servers, pointers can no longer be used for identification. The use of pointers would force the servers to be in the same address space, which is contrary to the goal of independence. The programming model must therefore include a different mechanism for the identification of objects across multiple servers. Still, for every object, one server must be able to inspect and modify its state directly, i.e. use pointers according to their main role. This makes every object a part of a specific server.

It should be noted that although existing microkernels do not directly define such a mechanism, there are many ways to uniquely identify objects without any support from the kernel, as long as servers can be identified uniquely. The specific mechanism in use effectively becomes part of the programming model.

3.1.3 Object Lifetime

An aspect strongly related to the identification of objects is the management of their lifetime. In a monolithic kernel, global policies exist for the construction and destruction of objects. These policies can generally not be enforced by individual independent servers in a multi-server system.

A common policy is that objects are guaranteed to exist as long as there is a reference to them (such as a pointer in a monolithic kernel). In other cases, objects are destroyed explicitly, and references (if any) are purged in the process. A programming model should accommodate both cases.

Again, mechanisms for object lifetime management are traditionally not part of the microkernel itself.

3.1.4 Other

Other aspects that concern multiple servers and hence require consideration in the server model include parallel code execution (e.g. on multi-processor systems), error handling, global security policies, and compatibility with legacy code. We

will not discuss them at this point because they are less relevant in the introduction of our server model – although they are equally important for the model as a whole.

3.2 Concepts

The main idea of our design is that we derive the concepts we use directly from the requirements discussed above. We do not make any references to hardware concepts such as address spaces, or to traditional operating system concepts such as processes. Instead, servers, their interaction, the identification of objects, etc. are "first-class" elements of the server model.

In other words, our server model is mainly a formalization of natural concepts in a multi-server system. However, this formalization leads to very specific code and data, which we are able to serialize as concrete server files, so that a collection of such files can be used as a multi-server system.

Specifically, our server model, called "Binary Service Specification" or "BSS," employs the following basic abstractions:

- **Servers** are the components of a system.
- Each server implements a **service** consisting of one or more **service functions**.
- Servers use **server references** to identify other servers and perform **service calls** to their functions.

We will explain the development of our server model by means of an example of a common type of component within every operating system, namely a file system implementation. (By "file system," we mean the implementation of a specific data layout on a storage medium, as opposed to the file system hierarchy of an operating system.) File systems are a good choice of example because they place a lot of demands on the server model, such as the ability to map parts of files to other servers, including automatically when a part is accessed, or the ability to handle several requests in parallel (especially in case one request requires data to be loaded from a disk while another can be handled directly from a cache in memory).

Our most fundamental abstraction is a "server," which is our term for a self-contained entity consisting of code and data. The "server" abstraction relates to the concepts of objects, modules, or components in a traditional application or OS. In this sense, a server most closely resembles a component, as:

- a server communicates with other code via well-defined interfaces only (unlike an object), and
- a server can be instantiated multiple times (unlike a module).

In our example, the file system classifies as a server (see figure 3.1). Communication with applications (or with an intermediate "virtual file system" layer) is bound to well-defined interfaces in every operating system, as is communication with the storage medium driver. The server may also have to interact with

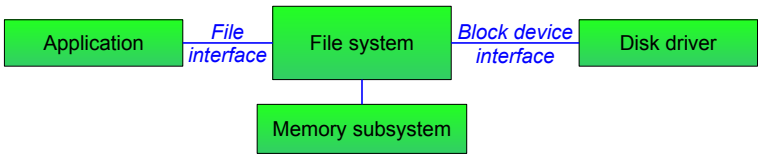

Figure 3.1: File system as a server with interfaces to other servers

other parts of the system, such as the memory subsystem. For this to happen, those other parts must also be realized as servers. The same file system server can be instantiated several times, for example for each partition of a disk.

In section 2.6, we presented an approach to system decomposition that consists of "wrapping" individual modules. We stated that this approach is applicable to virtually every module in every OS, but that individual modules must be wrapped manually in that case. We declared that our approach employs the same technique, except that BSS servers are wrapped and combined with each other automatically.

Thus, the entire interface of a BSS server must be defined in a machine-readable fashion. More precisely, whenever an operation performed by a server involves interaction with another server, the microkernel (or, more generally, the BSS implementation) must be able to:

- intercept this operation (which is the essence of "wrapping"),
- interpret as much of the semantics of the operation as necessary,
- determine the target server, and
- invoke the target server in a way that is consistent with the semantics of the operation.

For example, the common practice of accessing the same data from multiple modules is problematic, as these operations cannot be intercepted (aside from the fact that this practice breaks encapsulation). Instead, each BSS server possesses local data that no other server can access directly. Other servers need to explicitly invoke the server in possession of the data.

In a file system, this data would, for example, include file metadata such as names and inode numbers. In a monolithic kernel, such data may be available for access from any module, whereas in our model, all accesses across servers must be explicit operations (see figure 3.2).

Regular function calls are interceptable using various different methods. However, plain function calls do not exhibit the other necessary properties: They neither identify a target server, nor do they carry enough semantic information to invoke the potential target server if that server is not in the same address space.

Consequently, we augment the function calls with the necessary information. Since we want to employ this principle as the basis of potentially frequent communication between servers, we make sure that we can interpret all of the information at the time when we instantiate a server, instead of processing it during each call. This leads to the other three basic abstractions:

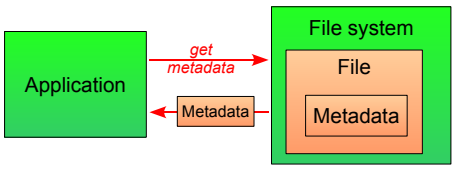

Figure 3.2: Data access across servers

A "service" is a collection of functions with well-defined semantics (an "interface" in component terminology). Services are declared in a machine-readable format that contains enough information to intercept function calls and invoke a target server in a different address space. Such a call is called a "service call." A dedicated parameter of each function specifies the target server; its argument is called a "server reference."

To relate these concepts to our example: Prior to implementing the file system server, we need to define a service (interface) for file systems in general. It consists of functions, one of which might be related to the query of file metadata, as outlined above. The description of the service must contain enough information to transfer the resulting metadata (e.g. a file name string) to the calling server, in light of the fact that the caller cannot access any of the file system server's data directly.

Finally, we adopt the common notion of "threads," as an abstraction of processors in coherent-memory multiprocessor systems. In microkernels such as Mach (in its original form), L4, K42, MINIX, and many more, the thread model diverges from virtual processors. For example, threads are usually bound to a single address space, and often used as a communication endpoint [4, 28]. However, in BSS, communication between servers follows the semantics of regular function calls as closely as possible, and function calls normally do not involve any thread switches. Accordingly, in the BSS thread model, threads are not bound to a single server, but enter and leave servers via service calls.

This difference has a strong influence on the design of a file system server. Rather than explicitly starting multiple threads to handle file operations in parallel, the file system passively awaits calls from other servers, which are automatically handled concurrently because they originate in different threads (see figure 3.3).

We will now describe the abstractions in more detail:

3.3 Servers

A server is primarily a reusable block of raw machine code. Reusability in this case means that the code makes no hidden assumptions about the environment it runs in – which simply cannot be true for pure machine code. To achieve this, the code is supplemented with additional information that enables the kernel to

- invoke the server correctly, and

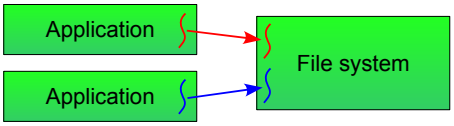

Figure 3.3: Semantics of threads and service calls

- perform appropriate actions whenever the server needs to interact with other servers, or with the kernel itself.

In other words, all assumptions are formalized in a format that the kernel can interpret in order to actually fulfill the assumptions. The exact data we need is very BSS-specific, but in contrast, the code is roughly equivalent to the implementation of a class in an object-oriented programming language, albeit with special restrictions according to the requirements of full decomposition.

To formalize invocation of the server, a server data structure contains:

- An indicator for the calling convention (stack and register layout) the server expects for incoming and outgoing calls. This parameter can be chosen by the user based on the capabilities of the compiler and the kernel.
- A description of the implemented service. The description includes signatures (parameter and return value descriptions) of all functions defined by the service, and also an ID number that (in most cases) uniquely identifies the service.
  The kernel needs this information in order to transfer data correctly between this server and other servers in the context of service calls. *For example, in the case of a file system metadata query mentioned above, the query involves a transfer of data from the callee to the caller.* Although the data transfer semantics are fully defined by the server model (see section 3.3.2), not every call actually involves such a transfer. The signature of a function indicates how the server expects the kernel to behave when calling the function. The caller of a function specifies a signature as well, which must match.
- For each implemented service function, an offset into the code block corresponding to the first instruction of that function.
- A value that indicates the amount of private data required by the server. The kernel must allocate a block of data of this size, and pass the address to every service function it invokes. To the server programmer, the address appears as the first argument of the function, or as the `this` pointer in C++.
  The private data of a file system would, for example, include global information such as superblock data, and also temporary state such as a list of open files.
  (If the memory requirements of a server are dynamic, the server can use memory management facilities implemented at a higher level (see section 3.5.3). However, in most cases, dynamic needs naturally correspond to the creation of new servers (see also 3.3.4)).

- Offsets of optional constructor and destructor functions, which are called when the server is instantiated or destroyed, respectively.
- The amount of stack space required. In traditional module-based kernels, programmers implicitly assume that they are able to allocate a certain amount of space on the stack, depending on constraints defined by the kernel. Since BSS servers are self-contained, this implicit assumption must be made explicit.
- Optional relocation data, so that the server can be loaded at different virtual addresses.
- Thread-related attributes of the server (see section 3.4.1). The attributes define whether the server is programmed in a thread-safe fashion, and which synchronization mechanisms are used. In addition, the server can allow or prohibit callbacks, i.e. whether an outgoing call of the server may again result in an incoming call from the same thread. Moreover, the server must specify the maximum required stack size (for a single invocation).

The manner in which a server interacts with the kernel and with other servers is the main difference between a server and a regular class. In normal code, external symbols refer to functions defined in other modules, and pointers refer to other objects at run time. In our case, the external symbols are replaced by service and system calls, and opaque values ("server references") are used as a substitute for pointers.

Thus, to formalize interaction with the kernel and other servers, the contents of the server data structure, as outlined above, need to be extended so that all calls are interceptable by and meaningful to the kernel. The additional contents are:

- A list of "required services." Server code assumes that the fixed server references 0, 1, 2, etc. refer to servers implementing these services (see section 3.3.3). Required services roughly correspond to constructor parameters of a class.
  A file system generally requires a "block device" service, which denotes the physical medium or partition the file system resides on. In addition to that, the list of required services specifies all resources the implementation needs, such as anonymous memory.
- A list of the code offsets of all service calls. This list also includes the corresponding service IDs and function signatures. To intercept service calls, the system needs to modify the code at these addresses to call system-defined stub functions. Such modification corresponds to the resolution of external references in regular code.
- A list of calls to functions which implement BSS-specific functionality such as synchronization, error handling, and debugging. The functions take the role of system calls in a usual system. (In the file format, this list is actually integrated into the service call list.)

3.3.1 Services

A service is an interface implemented or required by a server. Independently of BSS, an interface in a component architecture always consists of two separate aspects: the formal, machine-readable interface declaration, and an informal description of the intent of the interface within the context of the problem domain.

A BSS service encompasses both aspects; however, at the binary level, an ID number is the only indicator of the intent. It serves several purposes:

- It identifies the service within the context of server code. For example, a server can query another server for a specific service, using the ID as an argument.
- It is used in the list of required services of a server, each of which must be resolved when the server is loaded.
- For each service call, the service ID is stored in the server file, in addition to the signature of the function. Provided that service IDs are unique, this information frees the kernel from having to perform conformance checks on every invocation.
- If a service function contains server reference parameters or return values, the corresponding service IDs are considered part of the signature.

For each of these purposes, service IDs should be unique in a single system, but uniqueness is not a strict requirement. In other words, service IDs are merely hints; the user is responsible for making sure that any two communicating servers use the same IDs for the same services.

Therefore, service IDs are not a fundamental aspect of BSS in the sense that the kernel does not necessarily associate any specific information with a particular service ID. Nevertheless, their inclusion in BSS is justified by the following points:

- They enable the kernel to move conformance checks from invocation time to load time if they are unique.
- There is no other way to define a list of required services of a server, as the signatures of functions alone do not portray the meaning of a service. Although the required service IDs are opaque to the kernel, the loader of a server can use them to determine which server references to provide.
- Some services, for example those related to memory management, server loading, threads, hardware access, etc., are "predefined" as an appendix to the server model (see section 3.5). These services are intended to be part of the specification and therefore invariable, although updated versions can always be defined with different IDs. The IDs of predefined services may be built into the kernel.

Apart from its ID, a service consists of a list of functions, which are identified by their index at the binary level. Functions can have one or more parameters and return values, which the kernel must know about in order to properly transfer data between servers. For this purpose, the relevant information about parameters and

return values is summarized into a single value called the "signature" of the service function.

Not all usual types of function parameters make sense in the context of server communication. Parameters and return values of regular functions can be classified roughly into three categories:

1. Value parameters and results. These have scalar types of fixed size, such as integers and bit fields.
2. Opaque pointer parameters and return values. Many functions in modular code take pointer arguments and return pointers without ever dereferencing them. The pointers are only used as references to objects, not as actual memory addresses; they might be stored locally and/or passed to other functions.
3. Pointers which are dereferenced in the function, i.e. which point to data that is read or modified by the function on behalf of the caller. The caller and callee must agree on the data layout in memory.

The restrictions we place on the types of parameters and return values constitute one of the most visible aspects of the BSS server model. Since every server both implements and uses many service functions, the versatility of these functions has a major influence on the complexity of server code. Therefore, we explicitly support all of the three categories above:

1. Value parameters and results are unproblematic across server boundaries, since the kernel simply needs to read and write the values according to the signature of the function and the specified calling convention. To simplify the definition of a signature, we restrict ourselves to two different sizes. Furthermore, on 32-bit architectures, the two sizes are actually equal. On architectures with more address bits, we want to be able to handle values with the same high number of bits (e.g. offsets and sizes), but do not necessarily want to force the larger address size on all data types, possibly compromising efficiency.
2. In BSS, the use of opaque pointers is not possible across server boundaries. However, due to the goal of minimum possible granularity, we can assume that the referenced objects are always servers. Since references to servers are managed by the kernel, we can allow reference parameters and return values, as a replacement for such use of pointers.
3. Pointers to arbitrary data structures cannot be supported, as servers can access only their own data, but not the data of their callers (which may reside in a different address space). Still, for the data structures that are flat, i.e. do not themselves contain any additional pointers, the kernel can copy data between servers to give the appearance of regular pointer parameters. Such parameters can be regarded as an extension of value parameters and return values to support arbitrary data sizes. The kernel needs to know the copying direction(s), which are part of the signature, and the data size, which is specified at run time as another argument (for flexibility).

In a file system, all types of parameters are common. For example, a function that "opens" a file given a file name will take the file name as a pointer argument with inbound direction. Its return value would normally be a pointer to a file data structure, or an opaque file handle. According to the reasoning above, in BSS, the server model must allow the realization of files as individual servers; then, the "open" function can return a reference to such a server.

3.3.2 Service Calls

A service call is the invocation of a service function. Service calls are the only means of communication between servers. From the perspective of the calling server, a service call involves setting up the arguments according to the calling convention, particularly including the reference to the target server, and executing a `call` processor instruction to a specific address. Since no meaningful target address is known at compile time, the location of the call instruction is recorded in the server file, so the address can be set up by the kernel at load time.

Like regular function calls, and unlike IPC in other microkernels, execution in the calling servers does not continue until the service function returns. There are no asynchronous or non-blocking service calls. From the perspective of non-multi-server code, service calls are virtually equivalent to regular function calls in this respect. Since regular function are not associated with thread switches but regarded as operations within a single thread, our notion of a "thread" covers the entire chain of service calls from their originating server to the server whose code is executed on the processor. Analogously to a function call stack in a regular program, every BSS thread possesses a service call stack.

In terms of microkernel design, such synchronous and blocking behavior normally constitutes a security problem in light of the possibility that the callee might never return. In BSS, a server waiting for a service call to return is entirely unaffected by this fact under one condition: It must be "thread-safe" in the sense that other servers can call functions of this server in the mean time. For this reason, and also as a simplification over traditional server design, BSS is designed such that every server can be programmed in a thread-safe fashion, integrating concurrency into the core server model (see section 3.4).

In practice, the situation just described is not always a security issue even in traditional multi-server systems, since the calling server may not be able to continue working properly if the call does not return. A file system, for instance, is crucially dependent on the underlying disk driver; a failure of this driver inevitably results in the inability to access files.

In contrast, a more representative case is a virtual file system server, which forwards file requests to one of several actual file systems (see figure 3.4). If one of the individual file systems fails, requests to the other file systems should not be affected. Since the virtual file system server does not need to maintain any state across different threads, it is indifferent to threads that fail to return from outgoing

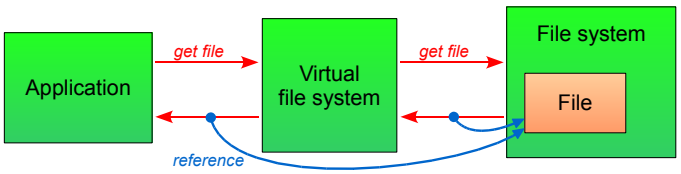

Figure 3.4: Virtual file system as proxy

service calls – as long as the kernel does not limit the number of threads that can enter a server.

3.3.3 Server References

As we already mentioned, server references act as opaque handles used in servers to point to other servers. They are used to define the target servers of service calls (akin to object references in object-oriented programming languages) and can be passed to other servers via service call arguments and return values. Since the server model is designed so that individual servers are as independent of their environment as possible, it is not guaranteed that server references valid in one server are valid in another. When they are used in service calls, the kernel possesses enough information to convert the arguments and return values accordingly.

For this purpose, the kernel needs to associate some state with the references of each server, which leads to the question about the lifetime of references. The following cases are common:

- A server calls another server and obtains a reference to a third server as a result of the call. It uses the reference (e.g. calls the associated server) within the same function, then returns from the function without "remembering" the reference as part of its own state. *For example, a server might open a file to read its data. After the operation is finished, the file reference is no longer needed.*
- A server calls another server, obtaining a server reference. However, it does not actually use the reference, but returns it as a result of the service function. In other words, the server acts as a proxy, merely passing the reference on. *A typical example is the "open" function of a virtual file system, as discussed above (see figure 3.4).* The case also occurs frequently in connection with local servers (see section 3.3.4).
- A server obtains a server reference and stores it in its own private data structure. The reference is used and possibly returned from different service functions; its lifetime exceeds the duration of a single service function. *In a file system, the root directory may be such a case.*

Clearly, we need to distinguish only two different cases: Either a reference is stored in the server's private data structure, then its lifetime is usually that of the

server; or it is used in a single service function, then its lifetime ends when the function returns (even if it is actually the return value of the function).

To accommodate these two cases, we define two different types of server references, called "permanent" and "temporary." All references obtained from service calls are temporary. This means their scope is the service function where the reference is obtained; when this function returns, the reference is no longer valid. For the most part, temporary references free server programmers from having to consider the lifetime of references. For references that are stored in the server data structure, there exists a system call converting temporary references to permanent references, whose lifetime is that of the server. In addition, all references can be "released" when they are no longer needed, to avoid resource exhaustion.

Since server references as an aspect of the server model require the kernel to maintain lists of the referenced servers, every kernel implementing the model inherently knows for each server whether it is still referenced from at least one other server. Therefore, the kernel is able to destroy servers automatically when they are no longer referenced (although circular references can prevent servers from being destroyed this way).

Such automatic destruction is of great value: Consider again the file system server returning references to file servers from its "open" function. Rather than implementing a corresponding "close" function, the file system can rely on the automatic destruction of these individual file servers when they are no longer needed.

3.3.4 Local Servers

If a server wishes to return a server reference from a function, it can obtain that reference from a call to another server, or by loading a server file (which, ultimately, is a service call as well; see section 3.5.1). In either case, it can talk to the referenced server only via service calls. Very often, this is too restrictive, since both objects are essentially part of a larger data structure.

The case of files in a file system is especially apparent: The file system data on a disk is a single data structure with a global consistency requirement. Individual files cannot be treated entirely separately from the rest of the file system, for example because they can grow and shrink and thus occupy variable space on the disk (among a lot of other reasons). In other words, the files are an inextricable part of the surrounding file system data structure.

Such a situation occurs in virtually every sufficiently complex server. Consequently, it qualifies for inclusion into the server model, in the form of "local servers." A local server takes on two different roles: When a server creates a local server, the local server appears as a regular object, which can be accessed directly using a pointer. However, the local server also implements a BSS service, and it is possible to obtain a server reference to it which is indistinguishable from a server reference to a non-local server.

For a server developer designing a set of services for a particular field, the existence of local servers ensures that every set of services can really be implemented.

For instance, if a file system service has an "open" function returning a file server reference, that service could never be implemented without local servers because encapsulating all of the file logic within a separate server would be impossible.

3.4 Threads

In section 3.3.2, we stated that from a server programmer perspective, service calls are similar to regular function calls. Since function calls do not involve thread switches, we define our notion of a thread orthogonally to service calls. Thus, a threads does not belong to a single server; any thread can potentially enter any other server, provided that the concrete system setup permits it.

Such thread behavior does not imply that a server has to be aware of the different threads that enter the server via service calls. With the exception of callbacks, all threads entering a server are equivalent from that server's point of view. In fact, there is no need to include a facility to name threads in the server model. Servers are completely passive, ready to be called from any thread.

The kernel then needs to ensure that all threads really are equivalent from the point of view of every server, which has nontrivial implications for scheduling (see section 3.4.2). Threads do not, however, need to be equivalent at a global level. For example, they may be scheduled entirely differently, as long as they are running in different servers.

3.4.1 Synchronization

If multiple threads can enter the same server simultaneously, race conditions need to be prevented. At first sight, such synchronization requirements seem to place an additional burden on server programmers, compared to traditional microkernel designs. However, with appropriate support from the server model, synchronization is not a problem:

- In contrast to monolithic operating systems, properly decomposed multi-server systems do not have any global synchronization requirements. Secure decomposition implies that every function of every server can be called at any time without causing damage to the server. If the multi-server system is designed correctly, then race conditions can occur only in individual servers, corrupting their private data structures. Thus, synchronization can always be handled locally.
- Not all servers need to be thread-safe. Since the kernel manages service calls, it can serialize calls at the request of the callee. To simplify server development, such a feature is included in the server model. Without kernel support, the same effect can be achieved on the server side by placing appropriate synchronization mechanisms around all service functions.
- The ability to intercept service calls places the kernel in a particularly convenient position to synchronize threads: It can ensure that only a single thread

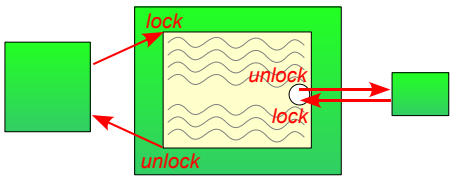

Figure 3.5: Automatic thread synchronization based on service calls

can run in a server at a given time, while permitting another thread to enter the server whenever the server executes a service call. In other words, the server is locked whenever it is executing code, but unlocked at each service call; all blocks of code between service calls are critical sections (see figure 3.5). This way, synchronization happens automatically without intervention from the server programmer. The programmer does, however, need to ensure that the server is in a consistent state at each invocation of a service call, so that other threads entering the server do not cause crashes or read invalid data. Moreover, the state after each service call is not necessarily the same as before the call, as another thread may have manipulated it. In practice, these two conditions rarely cause any difficulties; very often, all of the state manipulation within a function happens at the beginning or at the end, without any intermediate service calls.

Not all cases are covered by the kernel-managed synchronization models described above. If a server performs long parallelizable calculations (unlikely in an operating system, but there is no reason why application-level code cannot be realized as BSS servers), then all automatic synchronization would result in a complete serialization of the calculations. Hence, such a server needs finer control over critical sections. Since the implementation of critical sections without any kernel intervention is problematic, “lock” and “unlock” system calls are introduced in BSS. These system calls do not necessarily need to enter the kernel at all, but the kernel is free to supply an implementation that matches the kernel’s SMP and scheduling behavior.

3.4.2 Scheduling

The thread concept used in BSS has several different implications for thread scheduling. First of all, unlike IPC in other microkernel-based systems, service calls are orthogonal to scheduling in the sense that a service call or return operation does not imply a scheduling decision. In the context of fine-grained system decomposition, this aspect is an important efficiency criterion. No decision is required or even appropriate because the target server always acts on behalf of the caller, carrying out part of the caller’s operation. *If, for example, scheduling priorities exist in a system, and a high-priority thread calls a file system server to perform an operation on a file, it would be unnatural if the thread were descheduled because the file*

server has a lower priority of some sort.

The fact that threads represent operations spanning multiple servers and can be scheduled as a whole simplifies scheduling, but also places restrictions on scheduling operations because of the requirement that all threads entering a server must be equivalent from the server's point of view. In particular, the thread cannot be preempted because it has a low priority; doing that would leave the target server in a locked state and prevent other threads from entering it, enabling the caller to perform a denial-of-service attack.

The solution is not entirely obvious, but can be derived from the way this situation is handled in a monolithic kernel: At user level, threads have different priorities, but once a thread enters the kernel, it usually cannot be preempted (except at certain safe places). This is not a problem if the length of all operations is bounded, which can be ensured indirectly by limiting the resources of the application.

In BSS, certain service calls, such as file operations, are directly equivalent to system calls in a monolithic kernel, and thus can be handled in the same manner in terms of scheduling. The general criterion that can be applied is that all outgoing service calls of a server correspond to system calls in a monolithic kernel, except if the target server was loaded (directly or indirectly) by the caller itself. Accordingly, in BSS, each server can potentially schedule every thread it creates, as long as the thread is running within one of the servers under its own control.

To handle the case when the thread leaves its originating server (corresponding to a system call in a monolithic kernel), the concept of a thread hierarchy is introduced: When a server creates a thread, this thread is not scheduled per se. Instead, scheduling always happens within the context of a lower-level thread (the lowest-level threads being the actual CPUs of the system). When the higher-level thread makes a call that prevents the thread from being preempted, the scheduler of the lower-level thread can still preempt both threads at the same time.

Although the concept is more complex than scheduling in most other systems, it is important to note that this complexity does not reflect on the individual servers. A server file is essentially a passive block of code that may be loaded into memory and executed at will; in this sense, servers can always be preempted. The scheduling issue arises only if a single kernel is responsible for loading all servers and managing their interaction, and only if the scheduling behavior must fulfill some system-wide security policy.

That is also the reason why scheduling is not covered in detail in this thesis. In a particular microkernel, scheduling is an important design factor, determining properties such as real-time support. However, according to our modularity criterion, each individual server should be entirely indifferent to scheduling. Thus, scheduling is deliberately *not* a core part of the server model. Still, we need to make sure that the fact that threads are not bound to a single server does not implicitly cause any problems with respect to scheduling.

One consequence of the omission of scheduling from the server model is that developers of multi-server systems cannot rely on pre-existing scheduling facilities,

as present in most microkernels. One cannot, for example, program some servers, let them start threads with certain priorities, and then examine the run-time performance of the resulting system. Instead, programmers are forced to separate the development of individual servers from the design of the overall system structure, and scheduling belongs to the latter. The question whether this restriction has a positive or negative impact on system development is debatable.

3.4.3 Error Handling

Error handling is a thread-related issue because an error is always associated with a specific operation, represented by a service call, and thus local to a thread. Error handling is part of the server model because a single error concerns several servers, in particular the caller and callee of a service function.

Most of the time, when a server receives an error as a result of a service call, its only sensible reaction is to report the error to its own caller. In high-level programming languages, exceptions exist for this purpose: When a function throws an exception, the exception is propagated along the entire call stack, until it arrives at a place where the error can be handled in a reasonable manner.

In kernel code, exceptions are typically avoided because of their performance impact, which occurs even if no exception is actually thrown. The overhead is caused by the requirement to manually release all of the resources obtained by the affected functions. Without extensive help from the compiler, the only way to ensure correct resource deallocation is typically to catch and re-throw exceptions in every function.

In BSS, the situation is different because for each service function, the kernel already maintains a list of resources that need to be released when the function returns (in the form of server references). Thus, if exceptions are part of the server model, they free most servers from having to handle errors at all, while neither the kernel nor the servers need to maintain any additional state. In fact, due to the elimination of error checks, the use of exceptions can have a positive impact on performance.

Consequently, the server model contains three system calls related to error handling: To “catch” (i.e. handle) errors, a server can tell the kernel about the beginning and end of a “try” block. If an error occurs within this block, the kernel resumes execution in the server in the appropriate “catch” block. The third system call “throws” an error, i.e. aborts the current service function immediately, as well as all of the service functions in the service call stack where no “try” block exists.

3.5 Predefined Services

All BSS servers specify their requirements in the form of “required services” which must be resolved to server references at load time. This raises the question of who resolves the references of the first servers loaded at boot time, i.e. the servers

implementing basic system facilities. The requirements of these servers correspond to the raw hardware resources of the system, such as physical memory, processor time, I/O capabilities, and so on.

Most microkernels have APIs covering all hardware resources. The most basic servers then manage and abstract those resources, providing higher-level interfaces to the rest of the system. In BSS, we can take a different route, simplifying the implementation of both the microkernel and the basic servers: For servers, all references are opaque, they merely represent "something that can be used in service calls." The actual interpretation of the references is entirely kernel-specific. Thus, the kernel itself can hand out references to internal "servers," which act as regular servers in all aspects, except that they are implemented inside the kernel. This also eliminates the need to define a special security mechanism for hardware resources: Servers only can use these resources directly if they are in possession of an appropriate reference.

Such kernel servers are sufficient to replace the entire microkernel API. Furthermore, the services they implement can often be specified in a way that is general enough even for higher-level purposes. For example, physical memory can be represented in exactly the same way as regular anonymous memory, provided the kernel contains basic memory management functionality (which is required anyway, for kernel memory). The generality of low-level services has the added benefit that the basic system servers are also very general and reusable.

Predefined services take the form of an appendix to the server model. They represent an attempt to standardize basic facilities, but their use is entirely optional. Here, we will explain a selection of significant predefined services:

3.5.1 Server Loading

The most basic capability of any kernel implementing BSS is the ability to load server files. Given a reference to a block of data (see section 3.5.3), the "server loader" service can be used to load the data as a server and obtain a server reference.

In the process, the calling server needs to resolve the required services of the loaded server. This is done by specifying a "query server" which must return a server reference for each individual required service. The calling server can implement the query server as a local server of its own, but usually a more generic, globally defined query server suffices.

3.5.2 Threads

There are two predefined services for thread management. For regular servers, a simple "thread factory" service can be used to formalize the creation of separate threads of execution. Threads are created by specifying a function to execute, in the form of a server reference.

The second (lower-level) service enables the definition of schedulers. Threads created using this service are never executed automatically. Instead, they must be

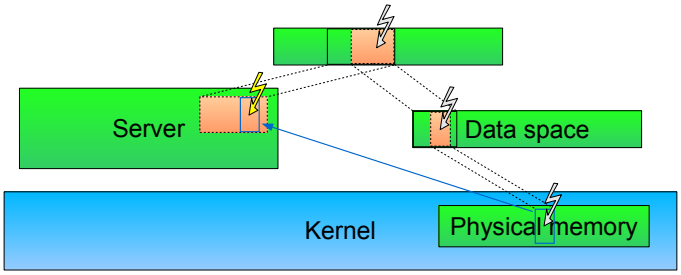

Figure 3.6: Page fault handling involving multiple data spaces

executed explicitly (dispatched) within the context of another thread. The service also includes facilities to preempt threads and to manage their state and accounting information. A scheduler can use this service to implement the higher-level "thread factory" service above. In order to actually preempt threads, a scheduler can register itself as a timer interrupt handler, and then call the preemption function from the context of the interrupt.

3.5.3 Memory Management

Facilities to allocate and map blocks of memory are important for any microkernel. Although mapping operations directly affect the address spaces of servers, they are not directly included in the server model. Instead, the facility to map data is provided as a pseudo-service which only the kernel itself can implement. For the kernel, there is no fundamental difference between this service and any other service, except that its implementation involves manipulating the caller's address space.

The actual data to map is represented by a separate "data space" service, which the "map" function takes as an argument. (The name "data space" comes from the SawMill operating system [7].) That service, which can be implemented by any server, formalizes accesses to data, thereby enabling the mapping of pages on demand based on page faults. The implementation of a data space involves redirecting every access to another data space, until a kernel-internal data space representing physical memory is reached (see figure 3.6). Although the data spaces themselves do not transfer any data, they indirectly tell the kernel which data to map.

The exact definition of these services has evolved significantly during the development of the server model, to support the secure and efficient implementation of diverse memory management subsystems such as anonymous memory with swapping and copy-on-write capabilities, memory-mapped files, and address spaces for legacy applications.

3.5.4 Legacy Compatibility

Traditionally, all applications running on a multi-server system are servers of their own, indistinguishable from other servers as far as the microkernel is concerned. If the microkernel API already includes the concept of address spaces and supports trapping of exceptions and system calls (as in L4 [28]), legacy applications can often be loaded and executed without modification. Otherwise they must be compiled specifically for the microkernel and multi-server system (as seen in MINIX [24] and Hurd [12], for instance).

BSS itself does not feature address spaces as a central concept. However, since the BSS server model differs significantly from the way regular applications are written, and because the importance of legacy compatibility cannot be overstated, the conversion of applications into BSS servers is often not an option.

Still, even though address spaces are not a central concept, they can be defined using appropriate services. In particular, data spaces are well-suited to describe the contents of address spaces, and all architecture-specific features can be included in the service definitions. To execute code within such an address space, a simple service call is sufficient, which returns when the execution is stopped (from a different thread, or from a system call or exception handler). Trapping of system calls and exceptions is easily mapped to callbacks.

3.6 Limitations

Our server model is intended to replace traditional microkernel design principles. First and foremost, this means that we must be able to develop a kernel that implements the model efficiently. If the abstractions used in the model are too high-level, we may end up with fine-grained, independent servers which meet all of our goals but require too many resources to actually be useful as an operating system. During the evolution of microkernels, their developers have realized that efficiency must be taken into account throughout the entire design process [31], but our case is special: Since we primarily designed a programming model without a specific microkernel in mind, we are concerned with the *potential* efficiency of the model's implementation, i.e. with the resource requirements and performance overhead that is implicit in the model, without reference to any particular implementation.

In section 3.1, we analyzed the basic requirements on a server model, which are essentially the least common denominator of all microkernel features. Actual microkernels offer far more specific functionality:

- *Isolation*: Minimizing the effect of a server malfunction.
- *Separation*: Splitting the system into separately administrated parts. This implies that servers can be created and destroyed independently of each other, and that resource utilization is strictly confined.
- *Modularity*: Restricting interaction between servers to well-defined interfaces.

- *Security*: Providing the necessary infrastructure so that multi-server systems can implement diverse security policies.
- *Portability*: Abstracting certain aspects of the hardware in a way that leads to portable servers.
- *Hardware access*: Letting multi-server systems leverage as many features of the hardware as possible.
- *Real-time support* (in some systems): Limiting the run-time of kernel operations, and providing the necessary scheduling framework for real-time multi-server systems.

In order for our programming model to act as a replacement for a microkernel API, we are ultimately concerned with the same goals, even though not all of these features are integral parts of our model. In fact, we consider all features except modularity and portability to be orthogonal to our server model, i.e. an actual microkernel implementing our server model is able to offer any subset of them independently of any server code. (While modularity was an explicit goal, portability of server code followed rather naturally.)

More specifically, we considered the goals of isolation, security, access to all hardware features, and real-time support when designing our server model (but not separation). However, they are only *potentially* covered by our model; use of the server model alone does not guarantee any of these properties (just as two servers adhering to an existing microkernel API are not necessarily isolated from each other; it is the actual kernel which ensures isolation and must be implemented accordingly). The consideration of these goals differentiates our programming model from regular module or component architectures. Yet, in contrast to microkernel APIs, the goals are not directly reflected in the model. Rather than that, for example, our specific approach to modularity ensures that two servers can be loaded into two separate protection domains (address spaces), and thus isolated from each other. The model does not, however, specify that this is indeed the case.

Because of this difference, the server model alone is not a substitute for an entire microkernel specification. A corresponding microkernel specification would also define that kernel's specific set of features and guarantees. (Strictly speaking, this is true for existing microkernels as well; for example, (earlier versions of) L4Ka and Fiasco implement the same API [28], but Fiasco provides real-time guarantees beyond those of L4Ka.)

Another difference is that other microkernels are designed to give the user complete control over the servers that are loaded at any time, similarly to the way entire operating systems let the user manage processes. In our server model, servers are deliberately not modeled as independently running processes, therefore such a feature is not immediately realizable in the same form, although other variants are unproblematic: For example, a "device manager" server could permit an administrator to load an unload drivers for specific devices, and then hand references to those drivers out to other servers. The difference stems from the fact that BSS servers are always loaded within a particular context, instead of being a global part

of a system.

More generally, the server model itself does not directly include any form of global resource management. Since fine-grained servers also require fine-grained resources, management of these resources naturally involves more layers of indirection. Fundamentally, every server manages the resources of all servers that it loads (if any), using its own resources as a basis. (Practically, few servers directly load other servers.) On the one hand, this restricts the user from managing most individual servers. On the other hand, given an appropriate hierarchy of servers, management of a few servers is sufficient to restrict the resources of all of their "children."

In any case, both from a developer and from a user perspective, the model deviates from traditional operating system concepts. One way to interpret this deviation is that the model disposes of established abstractions such as files and processes, and even threads and address spaces, operating at a lower level. From another point of view, these abstractions are simply not a core part of the model, but still available in the special cases where they are needed; server developers are simply forced to make all such requirements explicit instead of relying on a fixed set of microkernel features.

Chapter 4

Implementation

In this chapter, we will describe the most significant aspects of the implementation of our prototype multi-server system. The system consists of an IA-32 kernel that is able to load BSS server files, 31 individual servers, and test programs for various features (which are actually realized as servers implementing a dedicated "executable" service).

The main implementation goal was to build a system that would allow us to evaluate the server model with respect to several different criteria:

- The system's set of features should be large and diverse enough to be able to determine whether the abstractions of the server model are sufficient.
- We should be able to test in how far the reuse of existing operating system code is possible.
- At least some features of the system should be benchmarkable, so that we can estimate the performance overhead caused by our design.

Therefore, we decided to keep the kernel as simple as possible, and to focus on a set of servers implementing a particular widely-used feature. We chose the networking subsystem because it places a lot of demands on the server model due to high-volume data transfers, and because it enables us to measure and compare its performance using external equipment.

4.1 Kernel

When developing a kernel that is able to load BSS servers, the first implementation decision is whether servers execute in user mode or kernel mode. While other microkernel APIs define abstractions that implicitly assume user-mode servers, our server code makes few assumptions about its environment, in particular whether the code is running in user or kernel mode.

Since we wanted to keep the kernel as simple as possible, we decided to load servers into kernel mode. The decision does not rule out the possibility of user-mode servers; in fact, our system would enable us to develop a specific kernel-mode server to load servers into separate address spaces and interact with them.

However, since we did not consider this issue to be of vital importance, all servers currently run in kernel mode.

The main reason why a kernel-mode implementation is simpler is that the entire kernel can use "low-level" mechanisms corresponding to the concepts of our server model:

4.1.1 Servers

Servers are represented by compound data structures, each consisting of a fixed-size "system" part and a variable-size "user" part. The system part contains a list of function pointers for the service functions, a pointer to a "parent" server, a reference counter, and an optional destructor function pointer. The user part contains the internal data of the server.

This data structure can be used to represent both regular and local servers, and also server-like objects defined internally by the kernel. The kernel can invoke a server function by making an indirect call to one of the functions in the list and passing a pointer to the user part of the server as the first argument.

The actual service call arguments of the server model are divided into three classes: raw data, (temporary) server references, and pointers (see section 3.3.1). In our implementation, temporary references, which are opaque to servers, are internally realized as pointers to server data structures. Therefore, no processing is needed for service call arguments: Both local references and pointers are equally valid in all servers. In particular, although the server model stipulates that data must be transferred from one server to another when pointer arguments are used, we never actually need to perform any copying operations, since the data transfer semantics are trivially fulfilled if we use the same pointers.

When we load a server file, we set up a server data structure so that the function pointers correspond to the entry points defined in the server file. We cannot, however, let the kernel jump to these entry points directly: First, we always need to know which server is currently running, so we need to change an internal variable automatically before we enter a server. Second, whenever a service function returns, we need to clean up all temporary references it has acquired. To handle both of these requirements, we dynamically generate a short code sequence that registers the server as running, calls the actual server function of the loaded server, and then jumps to a globally defined cleanup function. If the calling convention includes arguments on the stack (on IA-32, this is the case if there are more than two arguments in addition to the server pointer, by default), we also need to copy these arguments to a different place on the stack before making the call. The reason is that the first copy of the arguments is already followed by a return address.

Furthermore, we need to relocate all outgoing calls of the server as part of the loading process. The calls that are defined are either system calls or service calls, system calls being related to permanent and temporary references, error handling, thread synchronization, and debug output. For error handling and debug output,

we basically "wire" the calls to the internal kernel facilities. In contrast, references and synchronization are always specific to a particular server: If a server converts a temporary reference to a permanent one, the permanent reference is tied to the server and must be released automatically when the server is destroyed. Therefore, the loader extends the data structure of each regular (i.e. non-local) server with a permanent reference list. For the rest of the kernel, the list is simply part of the "user data" of the server; only the loader and the dynamically generated system-call code are aware of its contents.

Similarly, we must generate code for the outgoing service calls of the server, since the server code itself does not have any knowledge about the internal server data structures of the kernel, particularly the function pointer lists. In a server file, all service calls are realized as direct calls to a relocatable address, whereas in our implementation, we always need to load the appropriate function pointer from the data structure of the target server, set up the first argument to point to the server's user data, and then call the function indirectly via its pointer. Moreover, for simplicity, we directly support the use of both temporary and permanent references as target servers. For calls to permanent references, we first need to load the pointer to the target server data structure from the caller's reference list; this is also done in the dynamically generated code.

In short, when we load a server, we essentially adapt it to our internal representation of BSS concepts, by generating a "wrapper" around it that makes it appear exactly like an internal kernel component.

4.1.2 Threads

Our kernel directly employs threads as a concept. They are represented by data structures with a fixed alignment in memory, followed by their stack (in the direction of stack growth). This way, a pointer to the current thread data can always be obtained quickly by masking the stack pointer. After the stack, there is an unmapped page as a rudimentary protection against stack overflows.

These thread data structures actually correspond to thread *contexts* in BSS terminology (see section 3.4.2). There is no in-kernel scheduler, but only a main thread (for bootstrapping) as well as a single thread for each CPU in the system (although multiple CPUs are not supported yet). All other thread contexts must be executed (directly or indirectly) within one of these threads. There is a "dispatch" kernel function which carries out this operation by saving the current stack pointer in the current thread's data structure, registering the current thread as a "host" in the target thread's structure, and loading the stack pointer from the target thread's structure.

This scheme makes blocking operations very simple to implement: Instead of switching to some global scheduler, a thread can block itself simply by returning to its host thread, i.e. by saving the current stack pointer to its own data structure and loading the previously saved stack pointer from the host data structure. From

the host thread's point of view, this operation appears as if the "dispatch" function had returned normally. Once a thread is unblocked, it can be dispatched again.

The thread data structure carries all data that BSS defines as being "thread-local." Most notably, this includes a stack of temporary references, as a list of pointers to servers whose reference counter is decreased when the corresponding service function returns. Furthermore, all errors are local to a thread. When an error is thrown, either from a server or from within the fixed kernel code, the kernel restores some of the processor state from a location on the stack, pointed to by the thread data.

4.1.3 Memory

The kernel needs to be able to allocate fine-grained variable-size blocks of memory for server data structures. At the moment, a fixed-size chunk of kernel memory is reserved for this purpose, to keep the implementation as simple as possible. It is part of the kernel executable; thus, it is a contiguous part of physical memory, mapped directly into the region of virtual addresses belonging to the kernel. Such memory is never handed out to regular servers.

In contrast, BSS stipulates that memory is represented by "data space" servers (see section 3.5.3). BSS defines a service to map arbitrary portions of data spaces into a server's address space – which, in this context, does not necessarily refer to a hardware address space, but simply means that the server can access the data using pointers. Since all servers execute in kernel mode and can therefore use kernel pointers, the kernel can implement this service easily. The only catch is that the data spaces to be mapped are not necessarily kernel servers; they can be regular servers that redirect accesses to kernel data spaces. Different regions of a data space may be redirected to different kernel data spaces. Thus, in some cases, the actual physical memory belonging to a data space may not be contiguous. Consequently, to obtain a valid pointer to the entire memory represented by a data space, the kernel must create a contiguous view of the data space contents in virtual memory. This also requires that kernel data spaces representing physical memory are always page-aligned.

4.1.4 Hardware Interaction

The kernel exports both IA-32 legacy I/O and memory-mapped I/O capabilities as kernel servers. Moreover, it delivers interrupts to registered servers. From an implementation perspective, only interrupts are somewhat complicated.

Unlike other microkernels, our kernel does not manage the interrupt controller present in a system. So, instead of providing IRQs as an abstraction, it enables servers to register interrupt handlers directly in the IA-32 interrupt descriptor table (IDT). More precisely, when a server requests to associate a certain IDT entry with an interrupt handler server, the kernel builds a custom IA-32 interrupt handler that, after setting up appropriate thread-related state, directly calls this particular server.

In our implementation, we reuse the thread data structure of the thread that was running before the interrupt occurred, for efficiency reasons. However, this is transparent to the interrupt handler server: For example, when an error is thrown inside the interrupt handler and not caught, the interrupt is aborted, but the thread resumes normally. From a logical point of view, each interrupt is essentially a thread of its own, with two special properties: First, as long as an interrupt handler is executed, interrupts are disabled – even when the thread enters user level. If interrupts were re-enabled before the interrupt handler has instructed the device to clear the interrupt, the same interrupt would occur again immediately. Second, interrupt-handling code is not allowed not block.

There are three different possibilities for enabling and disabling interrupts in the kernel: The first is a fully preemptible kernel, where interrupts are enabled virtually at all times. Since, with the appropriate protection mechanisms in place, BSS servers can always be preempted, full preemptibility is well-suited for a kernel implementing BSS. The downside is that it is quite complicated to implement. A simpler scheme is to enable interrupts briefly on every service call, when the kernel is in a well-defined state and no server locks can be held. Finally, we can enable interrupts only when the processor is either idle or running in user mode.

We decided on the last option because it is trivial to implement, and because it does not cause any overhead. This way, in-kernel scheduling is always cooperative, but we would not gain anything with respect to our goals if we implemented preemptive scheduling in the prototype. Moreover, very little kernel code directly depends on this decision; changing the interrupt-handling code to support one of the other two possibilities is quite feasible.

4.2 Bootstrapping

When we developed the system, it soon became apparent that we needed to load and connect servers in a way that is not hard-coded, since with more and more servers, the system structure becomes somewhat complicated and volatile. Also, most of the time, the reason we need to load a server is that some other server which we want to load requires the service implemented by the first server. We can even view the entire system as a single executable server (similarly to an “init” process in a traditional system) whose required services are resolved in such a way that the appropriate system servers, drivers, etc. are loaded.

When we load a server, its required services are resolved by a “query” server we specify. What we needed, then, was a query server that would load other server files, as configured by the user. Therefore, we implemented a file parser that reads configuration files in a specific format, and implements a query server for each configuration file. The features of this format have evolved over time; for example, it is possible to load other configuration files and use them to resolve the required services of servers that are loaded. Every subsystem is now fully described by such configuration files; no server other than the configuration file parser ever loads

server files.

When booting the kernel, the user needs to specify three file names: The "init" executable, the configuration file parser, and the main configuration file, which resolves the required services of the init executable. Other servers are automatically loaded as needed, based on the main configuration file.

4.3 Driver Framework

The core of the driver framework is a "PC hardware" server, which knows about the components of a standard PC, and loads (or rather, requests) drivers for individual components. Examples include the standard PS/2 keyboard (as present or emulated in every PC), the screen in text or standard VGA mode, PIT and RTC timers, and also the PCI bus (if present).

The PCI bus driver, in turn, scans the bus for devices, configures devices if necessary, and attempts to locate a driver for each device, based on the vendor and device ID and other information specified by PCI. For this purpose, the configuration file system has proven to be useful as a driver database: None of the information about suitable drivers for specific devices is encoded directly in any server; instead, we were able to realize the search for a suitable driver as a series of service queries, which can be handled by regular configuration files.

4.4 Network Device Driver

For a realistic evaluation of networking capabilities in our system, we decided to port an Ethernet adapter driver from the Linux kernel. We chose the Realtek RTL-8139 100 MBit/s adapter [41] because of its widespread availability and because it is supported by the QEMU [8] emulator.

The interface of an Ethernet driver mainly consists of functionality for packet sending and receiving. Ethernet packets contain hardware address fields for the sender and receiver, a higher-level protocol (or "type") identifier, the actual data, and a checksum. The RTL-8139 chip handles checksums automatically; all of the other fields are read and written directly from/to DMA buffers in the order they appear in the Ethernet packet (or "frame").

Therefore, it makes sense to pass the data received from the device directly to the next layer of the networking subsystem, and to pass Ethernet packets built by that layer directly to the device. Consequently, when defining an "Ethernet" BSS service, we decided not to treat the different fields of each packet separately, but to handle Ethernet packets (without checksums) as raw data.

When sending packets, this raises the question of where these packets are allocated. In our system, not all memory can be used for DMA, since data spaces can be user-defined and therefore are not necessarily contiguous in physical memory (see section 3.5.3). DMA-capable memory is allocated using a specific service,

which also returns the physical address of the memory so that this address can be written into a device register. As a special restriction, the RTL-8139 chip can handle exactly four send buffers, whose addresses must be set at initialization time. Therefore, if we want to avoid having to copy packets to driver-internal buffers prior to sending, the driver itself must be responsible for the allocation of send buffers, and must hand them out to the next layer when they become available (after a send operation).

The natural way of defining an appropriate "Ethernet" service is to include three functions: One of them retrieves a single packet from the device, and blocks if no such packet has arrived yet, another allocates an empty packet to be sent, and the third sends a packet. In our implementation, such a definition would work well, but it would not be future-proof: If, as intended, the device driver and the rest of the networking code were loaded into different address spaces, every function call would entail an address space switch, resulting in a significant performance overhead for every individual packet.

Therefore, the service we defined does not directly include these functions. Instead, it contains three functions returning server references, which can, indirectly, be used in the same manner. The corresponding services define generic "queue" functionality, which can be implemented outside of the driver, and especially in the kernel. To the user of the "Ethernet" service, this is transparent, except for the additional indirection. The driver, however, can request queue servers via its requirements, use one end of each queue for itself, and hand out the other end via its interface. This way, references to the packets are accumulated in the queue server, so the number of address space switches can be reduced.

4.5 TCP/IP Stack

In the rest of the networking subsystem, we encounter the same situation if we attempt to divide the networking code into different servers according to networking layers and protocols. Therefore, we decided to continue the same scheme along the entire networking hierarchy. Specifically:

- Incoming Ethernet packets must be distributed to different servers based on the protocol ("type") field (IPv4, IPv6, ARP, etc.). Since we want the device driver to handle packets transparently, a dedicated server is responsible for the registration of protocol servers and distribution of packets to those servers.
  However, since the vast majority of packets carries the same protocol (IPv4 at the moment, possibly IPv6 in the future), the distribution process adds unnecessary overhead. Therefore, we decided on a compromise: The driver can, at the developer's discretion, forward packets of a certain type directly to the corresponding server, by requesting special queues from the distribution server. The actual registration of protocols, as well as the distribution

of less-frequent packets, is still handled in the dedicated server. This approach eliminates the overhead for frequently-used protocols while keeping the amount of specialized code in the driver minimal.

- The next server in the chain, in our case an IPv4 implementation, still receives and sends raw Ethernet packets, albeit with a fixed protocol ID. Thus, its purpose is not only the implementation of an IPv4 layer, but specifically the implementation of IPv4 on top of Ethernet. In addition to dealing with Ethernet IP packets, the server answers ARP queries for its own address(es) and sends out ARP requests for addresses on the local network.
  Since IP routing is not Ethernet-specific and possibly concerns several devices, we defined our services so that it can be implemented as a separate server. This prompted us to introduce the concept of "endpoints" describing servers on the local network. In the Ethernet-specific IPv4 implementation, an endpoint corresponds to a specific hardware address.
  The service implemented by the IPv4 server (i.e. the IPv4 service, without reference to the Ethernet protocol), is defined in the same spirit as the Ethernet service: IPv4 packets are transferred via queues and treated as raw data. The IPv4 implementation does, however, reassemble fragmented incoming IP packets.
- Similarly to the Ethernet-specific IPv4 implementation, our TCP implementation is specific to IPv4, as it needs to deal with raw IP packets. It enables the user to open TCP connections and to set up TCP servers on specific ports. Connections are represented as pairs of streams, which, like queues, can accumulate data to reduce the number of potential address space switches.

We had originally planned to reuse most of the code from the lwIP [14] project in our TCP/IP implementation. However, the decomposed networking subsystem, as described above, has little in common with the design of lwIP. We regard decomposition as more important than the reuse of existing code, especially since in this case, the distinction between the IP layer and the TCP (or UDP, ICMP, etc.) layer is already inherent in the protocol. Therefore, we implemented the (relatively simple) IP layer without code from lwIP, but reused individual pieces of code from lwIP in the TCP layer. The code we were able to reuse concerns the Nagle [14] algorithm and the congestion control implementation, which determines how much data to send at any given time.

Chapter 5

Evaluation

In this chapter, we will evaluate the server model described in chapter 3 with respect to our goals. Since the abstract goal of "improving operating system decomposition" is largely immune to direct empirical analysis, we have built a concrete prototype system as an example, as described in chapter 4. Our experience with the implementation of this prototype system serves as an indicator for the fitness of the server model for the purpose of system decomposition. Furthermore, during the development of the prototype system, we were able to identify concrete aspects of the server model that needed (or still need) improvement.

5.1 Goals

Our server model was designed specifically for the purpose of fine-grained decomposition of operating systems. We can break down this goal into several aspects, and determine our success with respect to each:

- First of all, the model needs to have sufficient expressive power to develop real operating systems.
- At the same time, it must operate on a level low enough to be implemented by a microkernel.
- It should support fine-grained modularity, with servers that are strictly separate from each other, with little architectural overhead.
- Fine-grained modularity also requires that the model can be implemented very efficiently.
- Software interfaces used in existing operating system code must be formalizable as services according to the model.
- And finally, we want to be able to convert existing code from other operating systems into servers with as little effort as possible.

5.2 Methodology

Most of the subgoals above are of qualitative nature. To determine whether we were able to achieve them, we will analyze our prototype implementation, and compare it to alternatives where such a comparison is feasible. Quantitative measurement is possible when determining the system's granularity, for example in terms of the average size of a server. However, such a figure is not particularly meaningful without an estimation of the overhead caused by modularization, in terms of both effort and efficiency. For lack of a direct comparison, we need to make subjective statements about both kinds of overhead.

Our main focus lies in the effort and increase in code complexity involved with the conversion of existing code into BSS servers. To this end, we have ported a network device driver and a TCP/IP stack from two different sources, as examples representing more general classes of OS components. However, "complexity" itself is a relatively vaguely defined term: To us, it basically means the difference between "ideal" code that solves a particular, precisely specified problem, and the actual server code which is adapted to the restrictions of our programming model.

It is evident that established software complexity metrics such as Cyclomatic Complexity [35] are not very useful for our analysis, due to a very narrow definition of "complexity." We could, in theory, calculate the cyclomatic complexity of the original and ported code – and arrive at exactly the same values, as the cyclomatic complexity metric assesses the complexity of the abstract *problem* solved by the code. All of the changes we needed to make, however, are on a more architectural level; they manifest themselves in modifications to data structures and functions, as well as single lines of code, but not to branches and loops.

The situation is further complicated by the fact that the original code is written according to a specific programming model, without consideration for microkernel issues. Especially, it is not the "ideal" code which we would like to measure against; it is simply not possible to write code which *only* solves a particular, precisely specified problem. For evidence, consider how little code device drivers for the same device, but written for different operating systems, have in common, even though they solve exactly the same problem.

Although this does not prevent us from comparing the original and ported code and quantifying the changes we needed to make, it shows that we must also take into account *why* we needed to make a change. Most of the time, the situation is ambivalent: It is obvious that a certain implementation detail works only in a monolithic kernel, but still not all possible microkernel-compatible solutions are equal in terms of complexity.

This type of complexity is more subtle and less well-suited for quantitative analysis. It is known by the name of "accidental complexity" [10], as opposed to the "essential complexity" originating from the problem to be solved. The original use of the term refers to programming languages, and how well an abstract solution to a problem can be translated into specific language constructs. In any multi-server

system, the server model as defined by the microkernel introduces an additional requirement for such a translation: The resulting code must not only solve the problem, but follow the specific paradigms of the server model in doing so.

Thus, we can analyze our server model in the same way in which we would analyze a programming language – except that no obvious or generally accepted metric exists for this purpose. Therefore, we confine ourselves to the example of the network device driver, but tackle the question from multiple sides: First of all, we count and classify the modifications we needed to make. Secondly, for individual results of such modifications, we determine which constructs would have been necessary on a traditional microkernel. And thirdly, we check how much of the newly introduced code has a direct relation to the problem domain, and how much is introduced merely to meet the requirements of our server model.

5.3 Results

We analyze our model based on each individual subgoal as described in section 5.1.

5.3.1 Expressiveness

The basic requirement for a microkernel specification is that actual operating systems can be developed on top of it. We do not necessarily refer to multi-server systems or even to systems which follow any existing paradigms, but a server model must at least enable the definition of mechanisms for hardware access, as an interface between the microkernel and the system.

In principle, the fact that we were able to build a prototype operating system, with selected but diverse features found in other systems, shows that the server model fits this requirement. On the hardware side, we are able to access the screen, keyboard, and serial console directly and via BIOS calls, and we have developed drivers for the PCI subsystem and a network device. On the software side, in addition to our test programs, our system contains a (partial) emulation layer for the Linux ABI. All of our servers are thread-safe, which was easy to ensure because of the automatic synchronization based on service calls (see section 3.4.1).

We shall, however, describe how individual hardware features can be expressed in terms of services:

- The kernel gives a server access to a CPU by calling a specific service function. (There is no restriction in the model that would prevent upcalls from the kernel into servers.) For the first CPU, the kernel simply calls the main server. For each additional CPU, the main server creates a local server and asks the kernel to call this server from a separate thread. Since the server can be declared thread-safe, server code can be executed simultaneously on multiple CPUs.

- All physical memory is exported in terms of data spaces (see section 3.5.3). The server can use a service call to map and unmap parts of it in its address space.
- I/O (both IA-32 legacy I/O and memory-mapped I/O) can be abstracted using simple function calls, and therefore is implemented as a regular service.
- There is no difference between memory used for DMA and other types of memory, except that a fixed physical address is needed and that special memory attributes may need to be set in hardware.
- Interrupts can, at the lowest level, be mapped to function calls from the kernel to a registered server. The functions are called directly from the kernel interrupt handler.
- The creation of hardware address spaces, and the execution of user-level code in those address spaces, is definable as a service (see section 3.5.4). Data spaces can describe the contents of address spaces. System calls, software interrupts, and exceptions happening in an address space result in service calls to a registered server. All specialties of an architecture can be supported.

In short, services and service calls are sufficiently expressive to define a hardware abstraction layer. If the entire operating system is built as a single server, it can essentially be programmed like a monolithic operating system.

5.3.2 Implementability

The server model must be implementable by a microkernel. Therefore is must rely on abstractions that are simple and low-level enough for the implementing kernel to actually deserve that name.

This issue has two sides: While the general abstractions used in the server model (servers, services, references, etc.) are defined on a significantly higher level than usual microkernel primitives, the kernel can abstract the hardware at a very low level, as described above. For instance, there is no need for in-kernel scheduling or sophisticated interrupt logic.

The first result is that we have successfully designed and implemented a kernel which can load servers written according to our server model. At a code size of 53 KiB, we would classify it as a microkernel. This shows that the concepts of the server model are not too high-level for a light-weight implementation. We do, however, need to put this result into perspective:

- The implementation loads all servers into the kernel. Under this design decision, we were able to design the entire kernel according to a lower-level variant of the server model: The kernel internally uses mechanisms for threading, error handling, reference counting, etc. which are compatible with the concepts of the server model. A user-level implementation would have been significantly more complex, especially because a lot of special optimizations are needed to achieve acceptable performance at user level.

- Several aspects of the model are not exactly implemented as intended due to a lack of development resources. For example, preemption of servers is not possible (resulting in cooperative multithreading), the references of every server are stored in fixed-size arrays, and the thread stack size is simply chosen sufficiently large instead of an intended automatic stack switching on service calls. Non-essential hardware features, such as multiple CPUs, are not supported.

On the other hand, even the kernel-mode implementation is reasonably fault-tolerant: Most invalid memory accesses are automatically translated into regular BSS errors; they do not necessarily crash the kernel (or even the server which caused them, whether that is desirable or not). Stack overflows are not fatal either; the worst scenario is that some resources allocated by the function causing the overflow are not deallocated properly.

Testing the server model in practice revealed several details that needed modification:

- Originally, the server model did not contain the concept of permanent vs. temporary references (see section 3.3.3); in our current terms, every reference was "permanent" – its lifetime was that of the server. As it turned out, in a fine-granular system, passing references from one server to another becomes a very frequent operation. As an extreme example, in our networking framework, every packet is modeled as a data space, which is a (possibly local or kernel-internal) server.
  Passing a permanent reference from one server to another means that the kernel needs to update the reference table associated with the receiving server, i.e. search for a free entry and fill it appropriately. Furthermore, it must increase the referenced server's reference counter. On multiprocessor systems, both operations must happen atomically, causing some overhead due to locking.
  In contrast, temporary references can be managed in a thread-local data structure, avoiding both costly searches and locking. Moreover, when a temporary reference is used as an argument or return value, the reference counter does not need to be modified. Aside from these technological advantages, temporary references also lead to simpler server code because they do not need to be released explicitly.
- We took several attempts trying to find a good solution for error handling (see section 3.4.3). The dilemma is that while errors can happen at any time, often as part of normal system behavior (e.g. "file not found"), the caller of a function is usually prepared to deal only with a very limited set of errors (none, most of the time). Our exception-based approach at error handling provides a good way to abort an operation when an error occurs, but the situation when the caller of a function expects the function to fail is more complicated, since it is difficult to specify exactly *how* the caller expects the function to fail.

We decided that when a server throws an error, it should be able to attach all possibly useful information to the error (and this is still the case). Originally, we forwarded this information to the server that caught the error, under the assumption that this server would either analyze the information and continue or display the information to the user and abort. Transferring the information from one server to another turned out to be somewhat problematic, as the information needed to be stored at a thread-local location, without even knowing its size in advance. In practice, we never found any use for the additional information obtained from caught errors, except for a standardized error ID number. Consequently, in our current model, the rest of the data is no longer involved in the error handling process. Instead, the system can transfer it to a central entity responsible for the display and recording of error messages.
In two minor points, the current situation is still not fully satisfactory: First, there is no 1:1 correspondence between our error IDs and POSIX error codes (nor do we believe there should be). If the caller is a POSIX emulation layer (for example, our Linux ABI implementation), it needs to convert the codes before passing them to the application, which can lead to nonstandard results. One possible solution is that when a server throws an error, it always specifies a POSIX error code in addition to the BSS error ID, with overridable compile-time defaults. That way, when an existing POSIX-based system is decomposed, the error codes will stay the same.
Second, an error ID can only be a hint at the actual error that happened. For example, when calling an "open" function of a file system, a "file not found" error does not necessarily refer to the file being opened. In theory, it can also refer to some internal file, for example a server that needs to be loaded to handle the call. Ideally, there would be a distinction between such "internal" errors and regular errors, but there are no obvious criteria to determine the boundary where an error turns from "regular" to "internal".

- The preferred synchronization method, where servers are always locked when they are executing code but unlocked at each service call (see section 3.4.1), works well in many common cases but fails when several service calls are required to perform a single atomic operation. Even though such a situation should never occur in high-level code, it becomes common if all I/O accesses are mapped to service calls, since many interdependent I/O operations are often necessary for a single hardware action. Since I/O services are usually (but not always) implemented directly by the kernel, and are always intended to execute quickly, keeping the server locked is unproblematic. Therefore, we decided to simply mark certain services as atomic, instructing the kernel not to release the lock of the calling server.

- Currently, a server is never destroyed automatically as long as another server holds a reference to it (see section 3.3.3). This model can be implemented easily using a reference counter for each server. However, circular references

can prevent servers from being destroyed. At first, none of the services we defined required the use of circular references, but as the system evolved, such references became common: Quite often, in low-level code, a server offers another server to register itself in some way. Internally, the registration process involves obtaining a permanent reference to the registered server – while at the same time, the registered server already possesses a reference to the callee. The most prominent example is the registration of interrupt handlers with an interrupt controller driver, where it is also apparent that technically, the behavior makes sense: As long as a server is registered as an interrupt handler, the interrupt controller driver needs to be able to call it whenever an interrupt occurs.
Since the problem is deeply rooted in the BSS design and implementation, we decided to ignore it for this thesis. To solve the problem, the reference from the interrupt controller driver to the registered handler must, in some sense, be "weak:" Its presence must not affect the reference counter of the registered handler, so that when the handler (or its parent, if it is a local server) is destroyed, the reference becomes invalid. Some notification mechanism for the referencing server is necessary; an interrupt controller driver would, for instance, disable the corresponding interrupt.

5.3.3 Fine-grainedness

Fine-grained modularity was the main focus of our microkernel API, under the assumption that a rigorously decomposed system provides a good basis for further development towards other design goals (see section 3.6).

Subjectively, we can say that whenever we identified a part of a server that did not need to share any state with the rest of the server, we were able to split that part off into a separate server (or to implement the component as two separate servers in the first place). Especially, if the part in question used only a subset of the server's resources, the server model and the concrete services permitted us to limit the new server's resources in such a way that the server was confined exactly to its particular task.

We can also attest that encapsulating a server in this way does not lead to any substantial increase in code size. Part of the reason is that the "outer" server does not need to deal with actual server files; this job is handled generically by a carefully designed configuration mechanism. Another contributing factor is that the split-off candidates are often local servers, defined in their own compilation unit, and implementing services that are already defined.

To assess the granularity we achieved, we measured the minimum, maximum, and average sizes of our prototype servers, both in lines of code and in bytes of the resulting binary. In addition, we can regard local servers as sub-modules: On the one hand, there is no well-defined interface between a server and its local servers, but on the other hand, local servers are defined as separate compilation units (files).

	Minimum	Maximum	Average
Files/Server	1	10	3.6
LOC/File	6	620	86
LOC/Server	27	1363	306
Bytes/Server	280	13588	2907

Table 5.1: Minimum, maximum, and average server sizes in our prototype system

Therefore, we also determined the number of files per server and the lines of code per file (i.e. per regular or local server). The results are shown in table 5.1.

We left the configuration file parser (see section 4.2) out of the analysis. Since the configuration files include names of services and functions, the parser contains a database of all services that are defined. The database is generated automatically; thus, it should not count towards the lines of code, but it does result in a binary size of almost 100 KiB – making the server substantially larger than any other. Moreover, we did not count our test executables as servers, since they would skew the result in the other direction.

If the resulting numbers are any indication, we have achieved our goal of fine-grained modularity. A particularly interesting observation is that some empirical evidence suggests an optimal component size of approximately 250 to 400 LOC, depending on the programming language [23]. In the study, the optimal size is determined by measuring the relative number of defects per LOC, depending on the component granularity. A "component" is characterized by a well-defined interface, so in our case, a complete server would classify as a component, whereas a local server would not. It seems that our average number of lines of code per server (306) matches the empirical optimal component size quite well, although we arrived at our server size simply by decomposing the system at all places where it made sense from a technological point of view. Of course, a single empirical study may not be representative, but it does suggest that we have arrived at a limit where even more fine-grained decomposition would no longer improve a system.

The largest server in all terms (except for the aforementioned configuration file parser) belongs to the Linux ABI compatibility layer. The reason that we are not able to decompose this server further is that the Linux ABI itself inherits all of the complicated requirements prescribed by the POSIX standard. In particular, there are strong dependencies between all abstractions defined by POSIX, such as processes, files, memory management, synchronization, etc. (`fork` and `exec` semantics are a prime example.) The fact that this is the largest server, i.e. that all other servers are smaller, indicates that we were able to avoid such complex dependencies in the rest of our multi-server system.

Our numbers have to be taken with a grain of salt, though, since the implementation of many servers is incomplete and/or not scalable. In a real-world system, servers will likely be substantially larger. However, in most cases, the corresponding services are already sufficient; only the implementation of certain servers needs

	Pentium 4	Core 2
Indirect function call/return	11	9
Service call/return	46	23
Temporary save/release	60	31
Permanent save/release	$70 + 4x$	$23 + 2x$

Table 5.2: Number of cycles of common BSS operations

to be completed. If we do that, the resulting system cannot be said to be less well-structured than the current prototype system, since the structure of a system is defined by its interfaces, not its implementation.

In any case, fine-grained decomposition of our networking subsystem already exhibits a practical benefit: In a system with several network devices, we are able to (but do not necessarily have to) load entirely separate TCP/IP stacks for each device. For example, if one device belongs to an internal network, we can ensure *by design* that certain network services are visible only on that network, whereas in most systems, this is a matter of application and/or firewall configuration.

5.3.4 Efficiency

In order for fine-grained decomposition to be viable, it must be efficient. In fact, some multi-server operating systems on first-generation microkernels are said to have failed mainly because of the performance overhead of communication between servers (see section 2.4.1).

In these systems, the overhead actually comes from two sources: Since servers run in separate hardware protection domains, there is a certain hardware cost associated with every transition between servers. Moreover, the IPC mechanism defined by a microkernel API always has an inherent minimum performance overhead, depending, for example, on the complexity of the mechanism.

In our implementation, we restricted ourselves to servers running in kernel mode, although our server model is specifically designed so that servers can be loaded into different protection domains (address spaces). The kernel-mode implementation is unaffected by the hardware cost of address-space switching, as only a single address space is ever involved. Thus, the performance numbers we obtain purely describe the overhead of our implementation.

First, we measured the cost of the most common operations of the server model, when executed repeatedly on our prototype kernel. The results on an Intel Pentium 4 and Core 2 system are shown in table 5.2. The specific operations are:

- Calling a service function. We create a server with a service function that immediately returns. The function does not have any parameters or return values. (On the IA-32 architecture, up to two arguments and return values can be passed in registers and therefore do not alter the service call path.) We

	Pentium 4	Core 2
BSS	9.83	10.18
Linux	10.84	10.15

(a) Throughput (MiB/s)

	Pentium 4	Core 2
BSS	41%	21%
Linux	16%	4%

(b) CPU load

Table 5.3: TCP/IP performance (100 MBit/s full-duplex)

read the time stamp counter of the CPU, call the service function repeatedly, and read the time stamp counter again, to calculate the average number of cycles for a complete round-trip call/return operation.

- Obtaining and releasing a temporary reference. An existing permanent reference is converted into a temporary reference, and the temporary reference is immediately released again. As a consequence, the kernel needs to update the reference counter of the target server and update the temporary reference stack of the thread.
- Obtaining and releasing a permanent reference. A temporary reference is converted into a permanent reference, which is immediately released again. As a consequence, the kernel needs to update the reference counter of the target server and update the permanent reference table of the server. In our current implementation, the exact number of cycles depends on the other references the server possesses; it can increase substantially if there is an unused entry in the table followed by a lot of used entries (denoted by x in the table).

We also determined the number of cycles consumed by a repeated regular (non-BSS) indirect function call. A service call currently takes approximately 2.5 to 4 times as many cycles as a regular call (at least in this particular case). The main reason is that the kernel needs to keep track of the currently running server and clean up temporary references when a service function returns. The number of cycles can possibly be reduced slightly by inlining some of the code responsible for this, but in general, the data provides a good estimate for the minimum overhead of the operations defined by the server model.

The exact impact of these numbers obviously depends on the frequency of service calls. Therefore, we have used our TCP/IP stack as an example to evaluate the actual performance of our system. Table 5.3 shows the TCP/IP throughput and corresponding CPU load for our system in comparison with Linux. While the difference in throughput is likely a result of Linux's TCP/IP implementation using the link more efficiently, the high CPU load in our system is not acceptable.

The reason for this discrepancy can lie either in the TCP/IP implementation, in the overhead of service calls and other operations, or in a more general infelicity of our implementation that causes a large performance penalty in the processor. Therefore, we measured the duration of all lengthy operations (such as memory copying) and estimated the service call overhead based on the numbers above. The

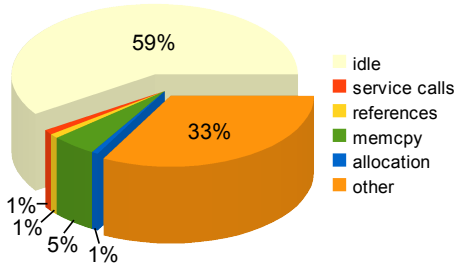

Figure 5.1: Estimated distribution of TCP/IP overhead

result for the Pentium 4 case is shown in figure 5.1. (We should remark that the amount of data that is copied is no larger than in a monolithic kernel.)

The calculated overhead caused by BSS operations only accounts for a CPU load of approximately 2% (i.e. 5% of the total load). This estimation indicates that the actual server model is not responsible for the inefficiency – but neither are the operations which are known to take some nontrivial amount of time. (We should add that we were able to identify one aspect of the server model which caused a high overhead: In outgoing pointer arguments, if the caller specified the transfer size in advance but the callee wished to fill only a portion of the buffer, the callee was forced to overwrite the rest of the buffer with zeros for security purposes. This problem has been fixed.) Further research is needed to find out what, exactly, leads to the high CPU load.

Fine-grained decomposition in general may be a factor, for example due to inefficient use of the instruction and data caches. Indeed, the Pentium 4 performance measurement counters indicate L1 load misses every 70 instructions on average. However, with the CPU instrumentation facilities that are available, we cannot find out exactly how many cycles are wasted due to these misses. Further, more tightly controlled experiments (i.e. servers that are less complex than a TCP/IP stack) would be necessary to find out the actual cause of the performance overhead. At the moment, we can only conclude that a substantial overall performance improvement is needed.

5.3.5 Interface Portability

Decomposition of existing operating system code can be divided into two aspects: A formalization of internal interfaces as services according to the server model, and an adaption of the actual code to the modified interfaces and the microkernel API. At first, we will briefly discuss how well interfaces can be converted from a monolithic kernel.

Clearly, not all possible interfaces are directly translatable to a multi-server system. The most problematic case we encountered is a frequent construct in the Linux kernel: Linux modules rarely implement a specified interface directly; instead, they have a single initialization function which is called at load time. This

function then registers a structure containing function pointers with some kernel subsystem, instructing the kernel to call the functions on specific events (e.g. when a device is found). Often, those functions will do the same for a different interface (e.g. to register a high-level abstraction of the device, or an interrupt handler).

In general, this construct translates to the creation of a local server, implementing the service corresponding to the monolithic interface. However, we strongly want to avoid loading drivers when no corresponding device is present; therefore, rather than exporting a global initialization function, our drivers directly export the corresponding high-level interface. Moreover, in Linux, some of the function pointer structures are built dynamically at run time, whereas our local servers are always defined at compile time. As a result, the conversion is not as straightforward as we would like. It does have the positive impact of reducing the code size, since the initialization and structure-building code is no longer needed.

A different kind of conversion problem became apparent in the control flow of the networking subsystem: In the lwIP TCP/IP stack, each network packet successively travels through all layers between the hardware and the application. The transition from one layer to the next is a simple function call. Therefore, the processing order is fixed (unless each packet is handled by a separate thread, which is unrealistic). Although in principle, this behavior can be translated to our server model quite easily, we also have to consider the possibility that each of the servers involved is loaded into its own address space.

To accommodate that case, we want to handle as many packets as possible in one layer before dealing with the same packets in the next layer. Instead of using function calls to pass packets from one layer to the next, we need to store the packets in queues and use individual threads in each server to dequeue and process them. The resulting interfaces are so substantially different from those in the original code that no interface reuse was possible.

5.3.6 Code Portability

We were able to achieve better results reusing individual pieces of code, particularly in the network device driver we ported. Figure 5.2 shows the amount of unmodified, adapted, and new code in the resulting server. 75% of the code was reusable either without modification or with an – often straightforward – adaption to the server model (for example, the use of specific BSS services instead of direct calls to kernel subsystems).

To us, a more interesting question is whether the new code constitutes an improvement over existing server models in terms of the aforementioned “accidental complexity” pseudo-metric. In order to answer this question, we categorize the added code according to its purpose. For each category, we discuss why it is necessary, and how the same problem would have to be solved on a traditional microkernel:

- The list of required services is unique to a BSS server. In the driver, we need to reference several basic OS services such as anonymous memory, DMA,

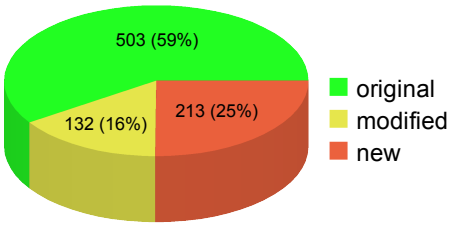

Figure 5.2: Code reuse in the RTL-8139 driver

short-time delays, etc. Moreover, the actual PCI device is represented by a required service.
Having such a list is unavoidable unless all of these features are part of the server model itself (as it is, in part, the case for other microkernels). We believe that an explicit list actually makes the driver more generic, as all requirements are well-defined. Memory, DMA, and delays are all part of the problem domain. Moreover, these code lines are not instructions but metadata. Their exact format constitutes accidental complexity, but their content is essential.
In microkernels that do not feature requirement lists, there needs to be actual code in each server to determine which other servers to talk to. Such code has no connection to the problem domain and is therefore less acceptable.

- We need to create three different local servers: An interrupt handler, a thread that waits for packets to be sent, and a server that represents the Ethernet link (because the link is a separate concept in the "Ethernet device" service we have defined). Each of these servers has its own local data, leaving us with two options: Either we distribute the driver's state across the different local servers as appropriate, or we use only the main server and keep a pointer to it in all of the other servers. Distributing the state makes sense because some of the state is used mainly in one of the servers (for example, in the interrupt handler). However, if we also want to access that state from the main server, we need to store a reference to the local server in the main server data, leading to additional code.
  The fact that we have four different data structures – while the original code has only one – can be regarded as a shortcoming of our server model. A remedy is certainly possible; in principle, there is no reason why the data structures of the main server and its local servers cannot be the same. It would, however, make the implementation more complex and possibly slower, as regular and local servers could no longer be treated the same way (see section 4.1.1). Moreover, distributing the state across the local servers makes the internal code of a server more modular, which has proven to have a positive impact in all cases except for the reuse of existing code. Therefore we regard the (relatively little) additional complexity as being justified.
  On a traditional microkernel, the equivalent of a local server would be a

communication endpoint, such as a port in Mach or a thread in L4. The exact problem does not arise because no state is explicitly associated with such an endpoint. However, the problem is not actually the *existence* of the data structure but the fact that the functions defined in a local server are given a pointer to it (instead of a pointer to the main server). Thus, in a traditional microkernel, the problem becomes an IDL issue. In fact, we would certainly be able to solve the problem exclusively on the server side by generating the appropriate server code in an IDL-like fashion, as an alternative to changing the server model – if we actually regarded it as a problem.

- As we mentioned before, we needed to change the networking architecture from a function-based to a queue-based model. 79 lines of code (37% of all new code, 9% of all driver code) are a direct consequence of this change. We must admit that in our eyes, this code counts as "accidental complexity," since it is not needed in a monolithic kernel (though it did permit us to remove some other code). Indeed, by changing the way in which packets are handled, we introduced bugs in the driver at first, due to ambiguous hardware specifications. However, as much as we would like to avoid making such changes, performance concerns force us to if we want to isolate the driver from other networking code at some point.
  It is important to note that this dilemma applies equally to other microkernels, since there is a fixed minimum overhead associated with hardware address-space changes. In fact, in microkernels with synchronous IPC (e.g. L4), the burden of implementing a queuing infrastructure lies on the driver itself (for example, using shared memory).
- We needed to make one particular addition to deal with a special situation that occurs only in our system: If there is no space left in the receive queue when a packet is inserted, the queue implementation throws an error. For performance reasons, we did not want to insert any code in the interrupt handler to catch this error. However, the interrupt must be acknowledged *after* all packets have been processed. This means that if we decide not to catch the error, the interrupt is not acknowledged and simply triggers again, possibly causing a live lock (depending on how, exactly, interrupts are handled).
  Our solution is to disable the interrupt for some time when we detect the problem via a receive buffer overflow. The question whether this constitutes accidental complexity is difficult to answer: On the one hand, it would not be necessary if we simply caught (i.e. ignored) the error, which makes sense because packets that cannot be delivered should be dropped. On the other hand, temporarily disabling the interrupt is certainly meaningful in the problem domain. In any case, the change is only a performance optimization.

To conclude, while interface reuse is problematic, code reuse works well, with a certain amount of additional complexity that cannot be avoided. Much of that complexity is the result of multi-server-specific performance considerations. At the same time, some of the original code is no longer needed.

We can also attest that the requirement to define all interfaces in terms of the rather restrictive server model has always led to a system structure that subjectively seems very easy to understand. For every server, the expected behavior is fully defined by the service it implements. This property greatly simplifies the conversion of code from an existing system, as the expected result of the conversion is well-specified in advance.

Chapter 6

Conclusion

In this thesis, we aimed to show that fine-grained decomposition of operating systems into multiple servers running on top of a microkernel is feasible. Since the server programming models of existing microkernels bring about a lot of additional code complexity when systems are decomposed into small servers, we defined a new server model called "BSS" based on simple abstractions specifically designed for decomposition. To evaluate the model, we implemented a prototype operating system consisting of a kernel and several servers.

The results are encouraging: Our server model proved suitable for the implementation of a wide range of system components. Whenever we encountered a situation where a part of a server was largely independent of the rest of the server, we were able to split this part off into a server of its own – often without any significant effort. This point is strengthened further by a close correspondence between the average code size of a server in our system and the optimal module size in software projects according to an empirical study.

Unlike any other system we are aware of, BSS servers are completely self-contained: Their purpose is precisely defined by the service they implement, and all of the requirements on their environment are specified formally. Knowledge of the overall system structure is not necessary in order to understand and modify each individual server of a system. No hidden interdependencies between servers can exist.

They are also thread-safe, based on our novel approach to synchronization: Only one thread executes code of a particular server at a given time, but whenever the server makes a call to another server, a different thread can enter the first server. This approach makes explicit locks unnecessary while still leveraging the performance benefits of a threaded system. Its simplicity can be a reason to consider BSS server development even more convenient than monolithic kernel development in this regard.

A somewhat alarming result is that our system is about three to four times slower at processing network packets than the Linux kernel, even though all of our code runs in kernel mode. Calculations based on our microbenchmarks indicate

that this discrepancy is probably not caused by common operations of our programming model, but we were not able to determine the exact cause. We cannot rule out the possibility that the act of decomposition itself – in our strict sense of the term – is responsible for the high overhead.

We succeeded in reusing most of the code of a network device driver and certain parts of a TCP/IP stack, but discovered that interfaces in monolithic kernel code cannot be translated to services in any straightforward fashion, especially when performance considerations have to be taken into account. To decompose the TCP/IP stack into separate servers for the IP and TCP layers, we essentially had to rewrite the IP layer and large parts of the TCP layer. We do not consider these problems to be specific to our server model, but rather conclude that a one-to-one translation from monolithic to multi-server code is simply not possible in some cases.

However, we should remark that in the resulting system, network layers are indeed separated more strictly than in the original code. This leads us to the observation that we cannot really measure the most positive aspect of our programming model: the fact (or belief) that individual servers from a decomposed operating system are more generic than the system they were derived from. Even though we were not able to eliminate all instances of "accidental complexity" in our system, the most important benefit of BSS can be summarized as follows:

Few components in the software world are completely self-contained, isolatable, and reusable in any given context. BSS servers are, to the maximum extent we deem possible.

6.1 Future Work

We have established the feasibility of fine-grained decomposition in principle. We also have reason to believe that our results scale up to operating systems large enough to be of practical value. However, in a practical system, a performance overhead as high as we measured would usually not be acceptable. Since all operating system research is ultimately concerned with practical application, we regard this as the most important aspect of our thesis that requires further research. Independently of BSS, our results even raise the question whether such a high overhead might be unavoidable in fine-grained multi-server systems.

Assuming the performance problems can be fixed, our server model still needs to be improved in several details (see section 5.3.2). We are confident that, after these modifications are carried out, the model will at some point reach a "stable" state characterized by the ability to define any type of OS component in terms of servers. In theory, this makes BSS a potential candidate for standardization, for example as an OS-independent driver framework.

Since BSS servers do not contain any microkernel-specific code, they can be loaded on top of virtually any other operating system. Although not relevant to this thesis, we have implemented appropriate run-time systems on top of Linux and

L4. This shows that BSS can be used as a generic component framework, even though it was designed specifically for the development of operating systems. An advantage of BSS over other frameworks is that servers can be used across different platforms without recompilation, as long as the CPU architecture is the same. On the down side, BSS servers require a rather extensive support layer, compared to simple language-based approaches without the requirement of (potential) isolation. Nevertheless, a port of this layer to other operating systems may not be a bad investment.

An even more far-fetched idea is the definition of a custom programming language for BSS servers. Currently, we map certain concepts such as services and server references to regular programming languages (presently C and C++) as well as possible. The exact mapping is the main source of "accidental complexity" within BSS. In a programming language with integrated BSS support, even the distinction between temporary and permanent references would no longer be necessary, as the compiler could infer the type of a reference automatically.

Since the server model specifies rather precisely how a server must or must not behave, even the definition of a *safe* programming language for BSS does not seem entirely out of reach. This would take BSS closer to the Microsoft Singularity project (see section 2.4.5), except that BSS would still operate at the machine-code level and employ simpler abstractions.

In any case, even in its current state, we see BSS as a contribution to the OS research community that enables the development of diverse operating systems without starting entirely from scratch. We believe that most of the design criteria of existing research systems can be met by BSS-based systems. For this reason, a pool of BSS servers implementing various system components would be valuable for OS research in general.

Bibliography

[1] GNU Hurd operating system: first user program run using L4 microkernel. http://en.wikinews.org/wiki/GNU_Hurd_operating_system:_first_user_program_run_using_L4_microkernel. Accessed December 15, 2008.

[2] GNU Mach. http://www.gnu.org/software/hurd/microkernel/mach/gnumach.html. Accessed December 15, 2008.

[3] SourceForge.net: ndiswrapper. http://sourceforge.net/projects/ndiswrapper/. Accessed December 15, 2008.

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

[5] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[6] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42, 2002.

[7] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[9] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN—an extensible microkernel for application-specific operating system services. *SIGOPS Operating Systems Review*, 29(1):74–77, 1995.

[10] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[11] John Bruno, Jos Brustoloni, Eran Gabber, Avi Silberschatz, and Christopher Small. Pebble: A component-based operating system for embedded applications. In *Proceedings of the USENIX Workshop on Embedded Systems*, pages 55–65, 1999.

[12] Thomas Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), January 1994.

[13] Alan Dearle, Francis Vaughan, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, and John Rosenberg. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[14] Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of the first international conference on mobile applications, systems and services (MOBISYS 2003)*, San Francisco, May 2003.

[15] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[16] Yasuhiro Endo, James Gwertzman, Margo Seltzer, Christopher Small, Keith A. Smith, and Diane Tang. VINO: The 1994 fall harvest. Technical Report 34, Harvard Computer Center for Research in Computing Technology, 1994.

[17] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[18] Brett D. Fleisch and Mark Allan A. Co. Workplace microkernel and OS: a case study. *Software—Practice & Experience*, 28(6):569–591, 1998.

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

[20] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 97–114, 1994.

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

[22] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, St. Malo, France, October 5–8 1997.

[23] Les Hatton. Reexamining the fault density–component size connection. *IEEE Software*, 14(2):89–97, 1997.

[24] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[25] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[26] F. Rawson Iii. Experience with the development of a microkernel-based, multi-server operating system. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[27] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system, 2006.

[28] The L4Ka Team. *L4 Kernel Reference Manual (Version X.2)*. System Architecture Group, University of Karlsruhe, Germany, 2006.

[29] Paul Leroux. Microkernel RTOSs simplify software testability, 2005.

[30] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, System Architecture Group, University of Karlsruhe, Germany, November 2005.

[31] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 175–188, Asheville, NC, December 5–8 1993.

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

[33] Jochen Liedtke. μ-kernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, WA, October 27–28 1996.

[34] Lucy. Inside the Mac OS X kernel. In *24th Chaos Communication Congress 24C3, Berlin*, 2007.

[35] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[36] Microsoft Corp. The Component Object Model: Technical overview. *Dr. Dobbs Journal*, December 1994.

[37] Rob Pike. Systems software research is irrelevant, 2000.

[38] Bernhard Poess. *Binary Device Driver Reuse (Diploma Thesis)*. System Architecture Group, University of Karlsruhe, Germany, March 22 2007.

[39] Richard Rashid, Robert Baron, Ro Forin, David Golub, and Michael Jones. Mach: A system software kernel. In *Proceedings of the 34th IEEE Computer Society International Conference (COMPCON 89)*, pages 176–178, 1989.

[40] Richard Rashid, Robert Baron, Ro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems; a position paper. In *Proceedings of the 2nd Workshop on Workstation Operating Systems. IEEE*, pages 109–113, 1989.

[41] Realtek Semiconductor Corp. Realtek 3.3V single chip fast Ethernet controller with power management – RTL8139C(L), rev. 1.4, 2002.

[42] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *Operating Systems Review*, 41(3), July 2007.

[43] Neal H. Walfield and Marcus Brinkmann. A critique of the GNU Hurd multi-server operating system. *SIGOPS Operating Systems Review*, 41(4):30–39, 2007.