The Structure of a Type-Safe Operating System

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Michael Golm

Erlangen - 2002

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	16. 09. 2002
Tag der Promotion:	17. 12. 2002
Dekan:	Prof. Dr. rer. nat. A. Winnacker
Berichterstatter:	Prof. Dr. rer. nat. F. Hofmann

Michael Golm

The Structure of a Type-Safe Operating System

The architecture of traditional operating systems relies on address-based memory protection. To achieve flexibility at a low cost operating system research has recently started to explore alternative protection mechanisms, such as type safety. This dissertation presents an operating system architecture that completely replaces address-based protection with type-based protection. Replacing such an essential part of the system leads to a novel operating system architecture with improved robustness, reusability, configurability, scalability, and security.

The dissertation describes not only the design of such a system but also its prototype implementation and the performance of initial applications, such as a file system, a web server, a data base management system, and a network file server. The prototype, which is called JX, uses Java bytecode as its type-safe instruction set and is able to run existing Java programs without modifications.

The system is based on a modular microkernel, which is the only part of the system that is written in an unsafe language. Light-weight protection domains replace the heavy-weight process concept of traditional systems. These domains are the unit of protection, resource management, and termination. Code is organized in components that are loaded into domains. The portal mechanism—a fast inter-domain communication mechanism—allows mutually distrusting domains to cooperate in a secure way.

This dissertation shows that it is possible to build a complete and efficient general-purpose time-sharing operating system based on type safety.

Part of the material presented in this dissertation has previously been published in the following conference proceedings:

Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. **The JX Operating System**. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, USENIX Association, pp. 45-58, June 10-15, 2002.

Michael Golm, Jürgen Kleinöder, and Frank Bellosa: **Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System**. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* , Schloß Elmau, Germany, IEEE Computer Society Press, pp. 1-6, May 20-23, 2001.

Meik Felser, Christian Wawersich, Michael Golm, and Jürgen Kleinöder. **Execution Time Limitation of Interrupt Handlers in a Java Operating System**. In *Proc. of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

Michael Golm and Jürgen Kleinöder. **JX: Eine adaptierbare Java-Betriebssystemarchitektur**. *GI-Herbsttagung: Mobile Computing*, Munich, Germany, November 16-17, 2000.

Contents

List of Figures

Chapter 1: Introduction

Chapter 2: Background

Chapter 3: Architectural Overview

Chapter 4: Domains

Chapter 5: Portals

Chapter 6: Components

Chapter 7: Processor Scheduling

Chapter 8: Memory Management

Chapter 9: Device Drivers

Chapter 10: Security Architecture

Chapter 11: Related Work

Chapter 12: Evaluation

Chapter 13: Conclusions

List of Tables

Chapter 1: Introduction

Chapter 2: Background

Chapter 3: Architectural Overview

Chapter 4: Domains

Chapter 5: Portals

Chapter 6: Components

Chapter 7: Processor Scheduling

Chapter 8: Memory Management

Acknowledgements

Building a complete operating system is nothing one person can accomplish alone. Many people have contributed to the JX project. JX grew out of my master's thesis about a meta architecture for Java, called MetaJava and later metaXa [73]. For this thesis I built a JVM and obtained first experiences with JVM internals. After I finished this thesis in 1997 I was employed as a research and teaching assistant at the Department of Computer Science (Operating Systems) of the University of Erlangen. Giving lectures and seminars allowed me to tell students about my ideas and convince them to do their thesis work in the context of the JX project. The first of those students was Hans Kopp who built a bytecode-to-x86 translator for metaXa as part of his master's thesis in 1998 [106]. Hans' compiler did considerably improve the performance of metaXa and using it as the foundation of an operating system became practical. At the end of 1998 I modified metaXa to run on the bare hardware and called it metaXaOS. At the same time I started to implement a new virtual machine, called JX, that provided multiple protection domains. In 1999 Andreas Weissel ported the IDE driver and the ext2 file system from Linux to metaXaOS [186]. Markus Meyerhöfer ported the driver for the 3Com 3C905B network interface card from Linux to JX and started to implement a TCP protocol stack [127]. In 2000 Christian Wawersich and Andreas Heiduk started to work on their master's thesis. Christian restructured Hans' translator and improved the quality of the generated machine code [185]. Christian's compiler allowed to run JX operating system components nearly as fast as an operating system written in C. Andreas wrote a driver for the Matrox G200 video card and the Brooktree BT878 frame grabber device [88] and a fascinating demo - an interactive puzzle of a live video source. At the end of 2000 Martin Alt and Meik Felser started their master's thesis. Martin wrote a bytecode verifier that performed the standard bytecode verification but could additionally be used to verify worst case execution times [5]. Using Martin's verifier we were able to guarantee an upper bound for response times without manually inspecting each interrupt handler. Meik focused on scheduling and multiprocessor support [63]. He ported JX to an SMP architecture and designed and implemented an interface to use Java schedulers. In 2001 Jürgen Obernolte, Christian Flügel, and Ivan Dedinski started to work on their semester thesis. Jürgen implemented a window manager [138]. Christian implemented a client for the Remote Desktop Protocol (RDP) [65] and Ivan implemented a database system [47]. Jörg Baumann restructured the garbage collector to allow domain-specific garbage collection and improved the performance visualization system. In 2002 Marco Winter implemented the AWT based on Jürgen's window manager [188]. I am grateful for the opportunity to work with all these talented students. Without them I would not have been able to prove that my design leads to a usable operating system.

I am deeply indebted to Jürgen Kleinöder who supported me from the beginning of the project. I thank Franz Hauck and the other members of the Ergoo group for the critical comments after my presentation dry-runs. This lead to considerable improvements of the final presentations. I thank Fridolin Hofmann and Bernd Hindel for referring my thesis and providing valuable comments and Wolfgang Schröder-Preikschat and Heinz Gerhäuser for being members of my committee.

Finally I owe thanks to my parents, my wife Anja, and my children Tobias and Rebecca for their patience and constant support during the often exhausting process of writing this dissertation.

CHAPTER 1 *Introduction*

This dissertation describes the architecture of a new class of operating systems: systems that are structured from the ground up to use a type-safe instruction set as their sole protection mechanism. In a type-safe instruction set instructions (operations) are applied to typed operands that refer to typed data entities. Strict rules describe what types of operands can refer to what types of data entities and what operations can be applied to them.

Using a type-safe instruction set instead of the traditional address-based protection has a number of advantages. The *robustness* of the system is improved because many kinds of programming errors can be detected at an early stage in the development by using the type system. An improved robustness has a positive effect on the *security* and *reliability* of the system, although I will show in this thesis that additional mechanisms are needed to maintain a secure state. The type-safe instruction set is a very fine-grained protection mechanism that allows to create *fine-grained protection* domains and enables an incremental *extensibility* of the system. The system design contains a communication mechanism that can be used for highly-efficient but unprotected communication inside a protection domain and for slower communication across protection boundaries. Because the programmer can use the same abstraction for intra-domain and inter-domain communication program module boundaries become independent from protection domain boundaries. The system can be *configured according to its intended use* on a scale that is bounded by the two extremes of placing each module in its own protection domain or placing all modules in the same domain. Systems that are dedicated to a very specific task, such as file-server appliances, benefit from the performance advantages of running all code in the same protection domain. Systems that run untrusted and potentially malicious code, such as an agent execution platform, benefit from the ability to spawn fine-grained protection domains. These systems must be able to completely isolate the untrusted code, which means *restricting access to information* that is available on the host as well as *restricting access to resources*. All these features must be realized without incurring excessive performance overhead compared to traditional operating system architectures.

1 Motivation

All hardware components of a computer have become more powerful in recent years: the access time of disks decreased while the throughput and capacity increased, CPU speed increased, memory access time decreased and memory capacity increased, accompanied by a steady reduction in hardware prices. This development enabled the ubiquitous use of computers for dedicated purposes, such as mobile devices, and lead to improved capabilities of general-purpose systems. While exploiting these capabilities software became more complex. To manage this complexity new software engineering techniques and methodologies have been developed. Examples for these techniques and methodologies are object-oriented design and implementation, component-based systems, and design patterns. But these developments had little impact on operating systems. Although there are object-oriented operating systems [38][80] and component-based systems [71], system software is still complex, difficult to comprehend, and expensive to change and extend. Different technologies for application development and kernel development make the interaction with the kernel cumbersome and inefficient. Reuse of designs and code between kernel and application is difficult if not impossible and usually requires two different interfaces [56]. This leads to several complications when the boundary between the application and the operating system becomes smaller or disappears. Examples for such systems are application-specific operating systems for databases, such as Oracle's "Raw Iron" database system, or operating systems for special- purpose hardware, such as routers or mobile consumer devices.

1.1 Robustness, security, and reliability

Complexity is the worst enemy of robustness, security, and reliability. Although current systems have reached a considerable level of complexity we assume that systems will even become more complex in the future. The most effective medicine for complexity is the principle of divide-and-conquer. A large system, such as an operating system, must be split into smaller, comprehensible parts.

1.2 Configurability, reusability, and dedicated systems

A general-purpose operating system is optimized for a specific workload. Using it for a different purpose leads to a phenomenon called *mapping dilemma* [104]. A mapping dilemma arises, when abstractions (modules, systems, services) are used in a way not anticipated by the designer of the abstraction. The repeatedly cited example is a windowing system where a heavy-weight window is used to implement one cell of a spread sheet program. While this leads to a short and concise spread sheet program, the performance and resource requirements render the program unusable. There are many examples for mapping dilemmas in current operating systems:

Database. High performance databases do not use the file system abstraction of current server operating systems, because the on-disk file layout does not match the file-usage pattern of database systems. Stonebraker et al. [169] describe the performance problems when running the INGRES database on top of a UNIX file system. They solve the problems by using a database-specific buffer cache and disk layout.

Web proxy. Gabber and Shriver [70] describe a similar situation for caching web proxies. Using a specialized file system library they were able to achieve a 6-10 times performance improvement compared to a web proxy that uses a general-purpose file system.

NFS. It is difficult to use a standard UNIX system as a high performance NFS server because of the general-purpose nature of the network stack and the file system. To achieve highest performance, the network code, the virtual memory code, and the file system code must be integrated. Examples are Auspex's functional multiprocessing system NS5000 [92], [94], Network Appliance NetApp filers [93] [184], and IBM's HA-NFS [21].

Multimedia. Multimedia applications need access to resources in short burst and with minimal jitter. This kind of service is not be provided by a general-purpose scheduler [137].

All of the above systems are dedicated to a specific purpose. A reusability framework is a critical requirement for the cost-effective construction of these dedicated systems. Different execution environments and implicit assumptions lead to bad reuse inside the operating system and between operating system code and user level code.

1.3 Extensibility and fine-grained protection

Besides being able to configure dedicated systems it should also be possible to extend the system at run time. An extensible OS is not completely created at built time but contains mechanisms that allow to add or replace system functionality at run time.

A special form of extensibility are *hot pluggability* and *hot swappability.* Hot pluggability means the extension, hot swappability means the exchange of a system component during normal operation and without the need to restart the system. Hot swappability allows to update a system component with a new version. System busses for embedded systems, such as Compact peripheral component interconnect (CompactPCI), allows to exchange hardware during normal system operation. The exchange of hardware components usually must be followed by an exchange of driver software, which must be hot swappable. Hot swappable OS components used, for example, in telecom switches to allow system updates without shutting down the switch. Several research projects investigated hot swappability. Examples include the Synthetix [179] system that uses hot-swappability to replace a general component with a specialized component, and the K42 OS [96].

Most operating systems today are extensible using loadable kernel modules but this completely lacks protection. The combination of extensibility with protection is one of the major advantages of a microkernel-based system that adders to a multi-server approach. Spreading the OS functionality among many servers means that a part of the functionality can be replaced by replacing a server.

2 Design objectives

To provide the described properties the system design has the following objectives:

Support for context-independent code. A flexible definition of protection and trust boundaries must be possible. This allows reuse of code between different protection domains. Even OS-level code, such as device drivers, file systems, and network protocol handlers, should be reusable in an application context.

Separation of concerns. Code that implements a certain functionality should be separated from code that manages resources or enforces a certain security policy. This is an essential requirement for reusability.

Shared-nothing. As hardware, especially CPU and main memory, constantly becomes cheaper and more capable it becomes more cost-effective to run applications on one host that formerly required separate hosts. In order to keep the illusion of separate hosts the operating system must allow to completely isolate such applications. They should not content for resources and should not share any resources. Typical examples for contention in traditional systems are the file system's buffer cache and the working set of a process.

Application-specific resource management. There is no single resource management strategy that is optimal for all applications. Thus applications should be able to implement their own resource management.

3 Contributions

The assumption that underlies this dissertation is that the root of the described problems is the address-based protection mechanism. I believe that this mechanism is not able to cope with current problems and that it should be replaced with a more flexible protection mechanism: type safety. The contribution of this dissertation is an operating system architecture that solely relies on a type-safe instruction set for protection. This architecture has been implemented and the prototype is able to run non-trivial applications, such as a file system, a network file system, a relational database system, a web server, and a window manager. The performance of these applications indicates that the system is usable even with today's hardware that is optimized for address-based protection.

This dissertation contains the first design, implementation, and performance evaluation of a multi-user timesharing operating system that relies on type-based protection[1].

4 Dissertation outline

The next chapter gives an overview of the research in operating system architectures. The discussion focuses on protection mechanisms, extensibility, and flexibility. Chapter 3 explains the overall architecture of JX. The following chapters describe the JX operating system in detail. Each chapter concentrates on one aspect, describes the design and implementation and evaluates the specific part of the system using micro benchmarks. They describe protection domains (Chapter 4), the communication system (Chapter 5), the component architecture (Chapter 6), CPU scheduling (Chapter 7), and memory management (Chapter 8). As these concepts are tightly interrelated it is sometimes necessary to use a detail that is explained in one of the following chapters. A forward reference is provided in such a circumstance. Chapter 9 describes the architectural support for device drivers. The security implications of the system design are studied in detail in Chapter 10. Chapter 11 surveys related work in Java-based operating systems and resource management. Chapter 12 evaluates the functionality and performance of the system. The final chapter gives a conclusion and perspectives for future work.

[1] It is not possible to state such a claim in this generality. Thus we state that we were not able to find the description of such a system in any of the major operating system journals, conference proceedings, or workshop proceedings.

CHAPTER 2 *Background*

By reviewing previous operating system research we attempt to understand the shortcomings of previous architectures and learn from their ideas. This chapter first gives an overview about the main trends in system architecture and then describes several protection mechanisms that have been used in operating systems. Protection is a very central part of the OS and to a largely determines the OS structure. Consequently, most projects that attempted to build extensible and flexible operating systems devote much effort to the protection mechanism. Other important aspects of extensible and flexible operating systems are resource management and modularization.

1 Operating system architectures

Since its early days operating systems struggled for an architecture with a modular structure that increases robustness and allows to adapt the system to new requirements (see Dijkstra [58], Parnas [143], and Haberman [78] for the early discussions about OS modularization). Current systems and research projects follow five architectural roads: monolithic systems with an internal component structure, microkernels with modularization of the system into protected services, object-oriented systems that emphasize ease of development and maintenance, single address-space systems with emphasis on performance and data sharing, and finally language based systems.

1.1 Monolithic systems

Even operating systems that are called monolithic usually have a modular structure and can be extended.

UNIX. Modern UNIX systems can be extended by using so-called modules. The module mechanism is used for device drivers but can also be used to load new file systems. Usually there is a defined interface between the kernel and the module that informs the module when it is activated and deactivated. The module implements functions of a specific function table, such as functions for network devices. This makes the module mechanism very similar to the inheritance mechanism of object-oriented systems. Modules run in the same address space as the system kernel and therefore require the same level of trust as the kernel. They can not be used for untrusted extensions and bugs in extension modules affect the whole system.

Windows NT/2000/XP. Microsoft Windows uses an extension mechanism called Component Object Model (COM). It is an object-oriented interface between components that can be written in different languages. As all components run in one address space there is no protection between them.

1.2 Microkernels

The major drawback of the module mechanisms that is used to extend monolithic systems is the lack of protection. Microkernels attempt to solve this problem be placing components in different address spaces.

Early systems. The concept of a microkernel was developed in the late 1960's. Hansen and colleagues built one of the first microkernels and described its philosophy in an influential paper [81]. The microkernel, called nucleus, makes no assumptions about the kind of applications that run on it. The nucleus provides the process abstraction, client-server-style communication between processes (IPC), creation, control, and removal of processes. IPC between processes is done using message passing. Every process has a message queue and message buffers are maintained in the nucleus in a common pool. Denial of service attacks on this pool are prevented by limiting the number of buffers a process can use simultaneously and by allowing a server to use the received request buffer for sending the reply. The designers of Hydra [190] argue for a separation of policy and mechanism [115]. This influenced other system designers to attempt to build "policy-free" kernels. The external pager mechanism of Mach is one example.

Mach. The Mach microkernel [99] has gained much popularity and is still used today, for example as the foundation of Apples MacOS X/Darwin [8]. Similar to the early microkernels Mach provides a few, hardware-independent abstractions: tasks, threads, memory objects, messages, and ports. Operating system functionality, such as file systems and network protocols runs as a server, called *task*. A task communicates with other tasks and the microkernel by sending a *message* to a *port*[1]. During port communication threads are switched but the timeslice is *handoff* from the client to the server thread. Mach version 3 uses continuations [55] to save memory. A *continuation* is a function that is executed by a re-scheduled thread and a data structure that encapsulates the state of the thread. This data structure can be seen as a reification of the functions execution state. In a synchronous, blocking invocation this state is equivalent to the stack frames.

A system configuration where the complete operating system runs in a single monolithic server is called a single-server system (Mach-UX). Building a single-server system can be accomplished by porting an existing monolithic operating system. Compared to the original monolithic system such a system has the advantage of being able to concurrently run many operating systems, called OS personalities, on the same machine and restarting one OS personality without restarting the whole system. Exchanging a part of the monolithic server is not possible in a single-server system. Therefore multi-server systems (Mach-US) [168] appeared that distribute the functionality of one operating system across many servers, such as tty server, file server, process manager. There is a file-system server, network server, etc. Security is based on ports, which are capabilities. In a multi-server system the communication between OS components is no longer a simple function call but a more expensive IPC. For performance reasons vital parts of the OS server were migrated back into the microkernel. A project at the OSF moved the OSF/1 UNIX server into the kernel [42]. They were able to achieve a performance only 8% slower than a monolithic UNIX. Another example is the network server that is integrated into the NORMA version of the Mach 3 microkernel [14] to achieve fast network IPC.

[1] Mach version 2.5 contained a "UNIX compatibility layer" within the microkernel. Release 3.0 [76] moved the UNIX kernel to the application layer.

L4. L4 [117] has a structure that is similar to Mach. It was designed to allow fine-grained servers by providing a fast IPC path. Abstraction by virtualization of the hardware is very important design goal for L4. The Clans&Chiefs [118] model, a custodial access control model [79], is used to validate IPC between protection domains.

Exokernel. While L4 can still be considered a microkernel in the tradition of Mach the Exokernel [61] follows a more radical approach. It assumes that every mechanism embodies policy decisions and thus separation of policy and mechanism is not possible. Therefore the Exokernel attempts to remove also the mechanisms from the kernel and strives for a kernel that only multiplexes hardware between protection domains and provides no higher-level abstractions or virtualization of hardware. Operating systems are deployed as libraries and linked to applications. This is a realization of the idea of an application-specific operating system first proposed by Anderson [174][1]. Engler [60] gives several examples how a library OS can be constructed and resources be safely multiplexed, including network packets and disk blocks. Network processing uses Dynamic Packet Filters (DPF) to dispatch packets to servers. DPFs are written in a simple loop-less language. DPFs are downloaded into the kernel and can be optimized. Disk processing dispatches disk blocks among the servers. To decide which server is allowed to access a certain disk block, the meta data of the file system is reused. As the file system is also implemented as a server the kernel has no information about the structure of the meta data. This problem is solved by using Untrusted Deterministic Functions (UDF). UDFs are written in a turing-complete language and are deterministic. Turing completeness allows the client to use any meta data layout and determinism allows the kernel to use a UDF once with a piece of meta data and when using the same UDF again it gives the same result, thereby guaranteeing that only those blocks are accessed for which access was granted during the initial run of the UDF.

1.3 Single address-space systems

Operating systems that use different address spaces to separate protection domains make it difficult for applications to share data. Single-address-space operating systems (SASOS) attempt to solve this problem by placing all applications in one large (usually 64bit) address space. Applications are protected by using segmentation or page tables with identical page mappings but different page protection. Examples for SASOS are Multics [46], Opal [39], and Mungi [90].

2 Protection mechanisms

An important responsibility of a multi-user operating system is the protection of the data and processes of different users from each other. We will look at five types of protection mechanisms: address spaces, software-fault isolation, proof-carrying code, virtual machines, and type-safe instruction sets.

[1] Building operating systems from libraries was common in embedded operating systems and process control systems, such as the IBM System/7. The difference between these systems and Anderson’s proposal is the requirement for protection using memory management hardware.

2.1 Address-space based protection

The first operating systems directly used physical memory. With the advent of systems that supported multiple concurrently running processes or tasks memory protection and memory fragmentation became an issue. Fragmentation leads to an under utilization of main memory. Processes could directly access and change the data of other processes, which makes debugging difficult and makes the whole system vulnerable for malicious or erroneous programs. The use of segmentation or virtual address spaces solves these problems. In such a system processes can compute arbitrary addresses but the hardware restricts access to a certain range of addresses and optionally translates addresses from a virtual address space to the physical address space. This allows to better utilize main memory by relocating data and using virtual addresses to access the data. It improves robustness by providing separate address spaces to separate processes. This introduced a new problem: it became difficult to share data. Single address space operating systems (see 1.3) attempt to solve the described sharing problems. They do not use separate address spaces but partition one address space between different protection domains. This is only practical for systems with a large virtual address space, as is provided by a modern 64bit CPU. SASOS continue to rely on MMU protection and continue to be restraint by its limits.

There is excellent support for address-space protection by current hardware. Highly optimized Memory Management Units (MMUs) are part of most processors. Address space protection is course grained and causes a performance overhead during inter-process communication. This motivated the development of other protection mechanisms, such as Software Fault Isolation and Proof-Carrying Code.

2.2 Software Fault Isolation

Software Fault Isolation (SFI) [180] provides memory protection by patching object code. The SFI patcher analyzes the binary code and statically verifies that memory accesses lie within a defined range or inserts instructions that perform this check at run time. SFI was used in VINO [161]) and analyzed in detail in [160]. The major SFI advantage is language independence. The disadvantages are: it must understand the processor instruction set, it uses registers for range checks and it causes runtime overheads.

2.3 Proof-carrying code

Proof-carrying code (PCC) [135] starts from the insight that is much more easier to check the correctness of a proof than to create the proof itself. In the case of PCC the proof states the memory safety of a fragment of code. The proof can be checked and the code can be safely run without address-space protection. PCC can be used to download extensions in a kernel [136].

2.4 Virtual machines

Virtual machines (VM) [126] have the same instruction set as the real hardware but emulate several instructions to provide resource isolation between VMs sharing the same hardware. A control program [126], also called hypervisor [25], is responsible for the isolation of the VMs. VMs experienced a renewed popularity with the VMWare PC emulator [199]. VMs have two problems: a large granularity and the lack of an efficient inter-VM communication mechanism. The VMWare emulator, for example, consumes a lot of resources to emulate a complete PC which makes it impossible to create fine-grained protection domains. A VM realizes a sandbox. The only interface of this sandbox to the external world are the emulated devices. This makes communication expensive and difficult to control, because the type of the communicated data is lost on the way from the application to the device interface.

2.5 Type-safe instruction set

In contrast to address spaces a type safe instruction set can not only be used to isolate different processes but also to improve the robustness and reliability of the tasks themselves. A type-safe instruction set allows to compute only valid addresses. By making several faults impossible or detecting them at development time a type-safe instruction set reduces the overall number of faults. There are no faults caused by the use of illegal addresses but something similar to faults is still possible. For example, a communication between two protection domains can be aborted because the domain was terminated. The other domain then would experience a fault (exception) during further communication attempts.

What is type-based protection?

Memory protection that relies on type safety must ensure the following minimal system invariants:

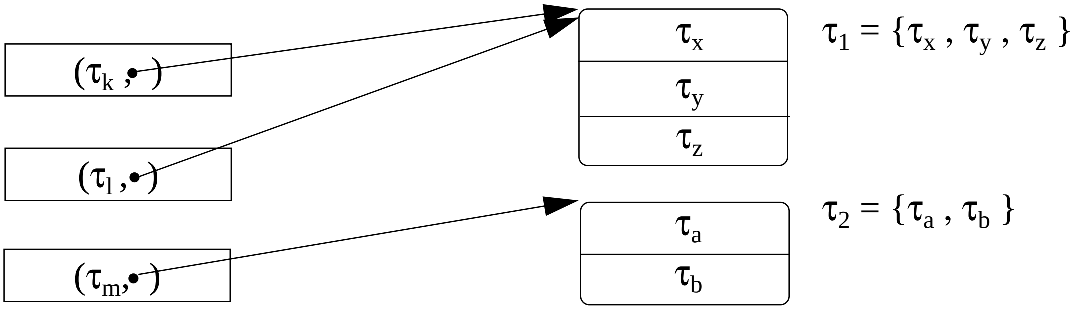
- Every pointer has a pointer type and a value (ptype, value). It is not possible to arbitrarily change the value or type of a pointer. The value of a pointer is changed by assignment.
- Every data entity has a data type (dtype). A data type consists of an ordered set of typed values.
- All instructions that access a data entity use a typed pointer and a numeric or symbolic index into the data entity. That the index is within the boundary of the data entity must be checked either statically or dynamically.

Most type systems have more rules. There are non-pointer data types and type conversions between these types. These rules are intended to avoid programming errors, but are not necessary for protection. Type systems can be classified into systems that check types dynamically whenever a pointer is used or statically at compile time. Static type checking usually is preferable because it does not cause runtime overhead and type errors are detected at development time. Systems that only allow static type checking are usually not expressive enough and therefore many systems use a combination of static and dynamic type checking. Figure 2.1 shows an example of type-based memory protection.

History. Using type safety as a protection started with the language Mesa that first was used on the Alto system [110] about 1973. The Alto supported four languages: BCPL (the ancestor of C), Mesa (a descended from Pascal), Smalltalk, and Lisp. Each language has its own microcoded instruction set. The Mesa language used a space efficient instruction set, also called Mesa [98]. It is an instruction set for a stack machine and is very similar to the Java bytecode instruction set. The development of one of Alto'successor, the Pilot [147], started in 1976. The Pilot exclusively used the Mesa instruction set for protection. As Pilot was a single-user operating systems it was mainly concerned with erroneous code and not malicious code.

Figure 2.1: Type-based memory protection

There are two data entities of type τ_1 and τ_2. Three pointers refer to the data. The first pointer has type τ_k, the second pointer has type τ_l , the third pointer has type τ_m. The system guarantees that pointer types τ_k and τ_l are compatible to data type τ_1 and pointer type τ_m is compatible to data type τ_2.



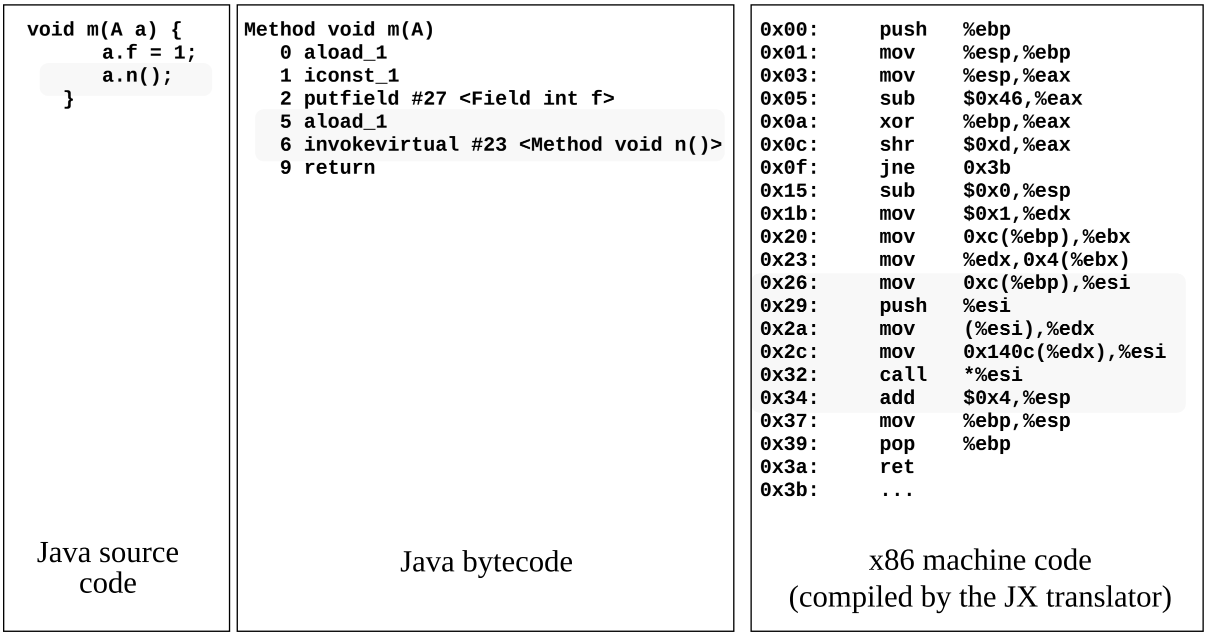
More recent operating systems that use safe languages are SPIN [18], which uses Modula3, and Oberon [189], which uses the Oberon language, a descendant of Modula2.

An operating system which used a type-safe language to achieve high security was KSOS [66], which used the Euclid language [108].

Intermediate instruction set. Using type-safety with off-the-shelf hardware requires the use of an intermediate instruction set. The first intermediate language that was designed but never implemented was UNCOL [170]. The motivation for UNCOL was the diversification of processors and the difficulties of porting machine language programs between these processors. Two of the first widely used intermediate instruction sets are PASCAL’s P code [166] and the Smalltalk bytecode [50]. The Inferno [54] system consists of a virtual machine, called Dis, and a language called Limbo. Limbo programs are compiled to the intermediate code of the Dis register machine. While the Limbo language is type safe, the intermediate code is not. Therefore, Inferno relies on a trusted compiler to sign the intermediate code. In the Mahler system [181] the intermediate language was used for code instrumentation and performance studies. The OmniVM system [2] used an intermediate instruction set for mobile code. Most of the intermediate instruction sets are designed for a specific high-level language. This often leads to inefficient code when compiling other languages to the intermediate instruction set. Typed Assembly Language (TAL) [132] is an attempt to create a low-level type-safe language that can be used as the target instruction set for many languages and efficiently compiled to a real CPU instruction set. Microsofts .NET architecture also relies on an intermediate instruction set, called Microsoft Intermediate Language (MSIL) [59], which is a type-safe instruction set. MSIL was designed as the target instruction set for different languages, such as C# and Visual Basic. The JX system uses Java bytecode. Java bytecode is an instruction set for the Java Virtual Machine (JVM) which is an abstract stack machine. Figure 2.2 shows how the Java source code language is translated to the bytecode instruction set and then to the instruction set of the real CPU, which is a x86. The instruction set of the JVM uses static typing with few dynamic type checks. There are two kinds of types: pointer types and numeric types. There are two data areas, the heap and the stack, and three

Figure 2.2: Relation between Java source code, Java bytecode, and machine code

A type-safe Java source program is compiled to a type-safe intermediate instruction set. This instruction set can be interpreted or compiled to the instruction set of the real processor. The shaded area shows the representation of a method invocation in Java source, bytecode, and machine code.



```
void m(A a) {
      a.f = 1;
      a.n();
   }
```

Java source code

```
Method void m(A)
   0 aload_1
   1 iconst_1
   2 putfield #27 <Field int f>
   5 aload_1
   6 invokevirtual #23 <Method void n()>
   9 return
```

Java bytecode

```
0x00:     push   %ebp
0x01:     mov    %esp,%ebp
0x03:     mov    %esp,%eax
0x05:     sub    $0x46,%eax
0x0a:     xor    %ebp,%eax
0x0c:     shr    $0xd,%eax
0x0f:     jne    0x3b
0x15:     sub    $0x0,%esp
0x1b:     mov    $0x1,%edx
0x20:     mov    0xc(%ebp),%ebx
0x23:     mov    %edx,0x4(%ebx)
0x26:     mov    0xc(%ebp),%esi
0x29:     push   %esi
0x2a:     mov    (%esi),%edx
0x2c:     mov    0x140c(%edx),%esi
0x32:     call   *%esi
0x34:     add    $0x4,%esp
0x37:     mov    %ebp,%esp
0x39:     pop    %ebp
0x3a:     ret
0x3b:     ...
```

x86 machine code
(compiled by the JX translator)

classes of instructions that must preserve typing of data: instructions that load a word from the heap and store it on the stack or vice versa and instructions that call methods that expect parameters on the stack and push the return value. From a memory-protection point of view type-safety must ensure that there is no data flow between memory cells (heap or stack) that store numeric types and memory cells that store pointer types. When the same memory cell is used to store both kinds of types, as are stack cells, it must be assured that they are re-initialized before being used with a new type. When data flows between cells with different pointer types (pointer assignment) the object-oriented subtyping rules apply. Pointers can be assigned from subtype to supertype but not in the opposite direction. When this is necessary a dynamic type check must be executed using the checkcast instruction. The bytecode verifier ensures that this instruction is present in the appropriate place.

Assuring type safety. Building an operating system that completely relies on the type safety of the JVM instruction set requires a high confidence in the correctness of the JVM specification with respect to type safety and the proper operation of the bytecode verifier. There have been many attempts to formalize the JVM specification (see for example [145]). These projects uncovered and fixed errors in the following areas of verification: ensuring that an object is initialized before it is used [150], typing of subroutines [167], dynamic class loading [153].

3 Resource protection and management mechanisms

In addition to protect the memory of tasks an operating system has to manage and protect resources. Tasks and untrusted kernel extensions consume resources and modify the state of a trusted component. Approaches to handle this are hierarchical resource management, resource containers, transactions, and path-based resource management.

3.1 Hierarchical resource management

The first microkernels also introduced a hierarchical resource management (HRM) scheme, called nested processes [81]. The process hierarchy, which is usually defined by the parent/child relationship, is used to manage and account for resources. A child obtains resources from its parent and these resources are deducted from the parent. HRM is used in Fluke and Alta [177]. ANTS decided against a hierarchical model because they consider it to be too restrictive [107]. The argument is that the resource permissions of an active code domain should be determined by its credentials and not by the permissions of the ANTS execution environment that creates the domain.

3.2 Resource containers

The purpose of resource containers [13] is to separate the two abstractions protection domain and resource principal and account for resources that are consumed inside the kernel on behalf of a process. Although Banga et al. only implemented resource containers to control CPU usage, they state that the resource container abstraction is more general and can as well be used for the management of other resources, such as disk bandwidth and TCP buffers. A resource principal can even be more fine grained than a process. As their major example is a web server it is beneficial to account resources per connection or per thread that handles a connection. This is of course only possible if the thread does not behave maliciously and only uses defined interfaces to access resources within its own process.

3.3 Lightweight transactions

Transactions are used in VINO [156] to prevent resource misuse of kernel extensions (grafts). VINO transactions are designed to be lightweight. Nevertheless the cost of transaction management may outweighed the benefits of the kernel extension.

3.4 Path-based resource management

Scout [134] uses the abstraction of a path for resource management. Resources are not accounted to a process, protection domain, or module, but to the execution path of an activity. This execution path can cross module boundaries. The Escort architecture [164] ensures that all dataflow in Scout is represented as a path. This is essential to guarantee security and the confinement of information. The Scout team ported a JVM to the Scout system (Joust [83]) and applied path-based resource control to Java programs.

3.5 User-defined resource management

A study of Mach performance measured that unnecessary multiprocessor locking (on a uniprocessor) was responsible for about 10 percent of the overhead of running the OSF/1 system on top of Mach compared with the OSF/1 integrated kernel [42]. This and the fact that a large number of all systems are uniprocessors make it worthwhile to allow application specific scheduling to avoid heavy-weight locking using mutual exclusion locks.

4 Modularity and reusability

A modular design and reusability have been important topics since the early days of operating systems. Habermann [78] proposed a layered design called “functional hierarchy”, where each layer provides a new “virtual machine” abstraction to the upper layers. Parnas [143] and Dijkstra’s “THE” system [58] also emphasize modularity.

Depending on the mechanism that is used to encapsulate modules a different communication mechanism is needed: procedure call or message passing. Lauer & Needham [113] observed that these are equivalent abstractions that even can be mapped to each other at a conceptual level.

Modularity is especially important for embedded systems. These systems have severe resource constraints and therefore cannot accept unneeded functionality. Furthermore they often must be customized to a specific application domain. Therefore it is not astonishing that many projects that emphasize modularity have an embedded system background, for example, MMLite [91], Pure [20], Pebble [71], and 2K [105]. Modularity and reusability is the main theme of the following systems:

x-Kernel. The x-kernel [97] is an operating system that focuses on the flexible construction of communication protocols. It allows to compose new protocols by using existing components. Everything in the x-kernel, including devices, is represented as a protocol and the system has strong supports for the composition of protocols. The x-kernel demonstrated that this flexibility does not come with a performance cost. The x-kernel protocols are faster than equivalent UNIX implementations.

Lipto. Lipto [57] is an OS derived from the x-kernel. It considers modularity and protection as two orthogonal concepts. Lipto uses proxies [158] to uniformly refer to local and remote objects. It uses a proxy compiler to generate C/C++ code from an interface definition. This complicates its use from a programmers perspective and restricts proxies to classes that are specially prepared at development time.

Microsoft COM. The Component Object Model (COM) [26] is the component model of Microsoft’s Windows operating systems. COM is a language-neutral, binary interface standard and an architecture that defines an execution environment for components. An important part of this architecture are so called apartments. The main purpose of apartments are to mix components that were written in a single-threaded style (without synchronizing access to the objects), called single-threaded apartments (STAs) and components that use multiple threads, called multithreaded apartments (MTAs). As COM was introduced 1993, at a time when Windows did not support threads, there was a lot of code written in a single-threaded style. The two types of apartments can be mixed in a single process. Calls between incompatible apartments are serialized by using a proxy instead of a direct object reference.

The programmer must be careful not to share a direct reference to an object between different incompatible apartments, for example by using a global variable. Although COM provides the ability to run a component in a separate process (out-of-process, EXE server), communication with such a component is orders-of-magnitude slower.

OSKit. The OSKit [67] attempts to leverage the huge amount of UNIX-like open source operating system code to build new operating systems. The kit provides glue code to use code from different systems in a single system, like a NetBSD network stack with a Linux file system. To achieve this reusability of independent components, existing systems are componentized and equipped with a COM interface.

MMLite. MMLite [91] is an architecture to build a complete OS out of object-based components. It is similar to the OSKit approach but allows to even replace the scheduler component or the virtual memory management component. COM is used as an interface between components, which usually are written in C or C++. Because modularity is at the object level, the memory of objects can not be deallocated explicitly. A reference counting garbage collector is used for the heap management, which has well-known limitations, such as the inability to collect cycles and an overhead for the maintenance of the reference counts. Address-space protection is used to provide firewalls but is otherwise orthogonal to the modular decomposition of the system. Its conceptual architecture is very similar to Lipto.

5 Extensibility and openness

As many of the previously presented concepts the idea of an open operating system is fairly old [109]. There are two categories of openness: systems that allow a gradual modification using an extension interface and systems that allow the wholesale replacement of components. The first category includes Mach external pagers [191], extensible file systems [122], as well as applications that are extensible by providing a so-called plug-in interface. Examples are QuarkXpress, Photoshop, Netscape, and Gimp. The extensions are loaded in the address space of the application and therefore must be as trustworthy as the application itself. Current UNIX kernels can be extended by using modules. As they run in the kernel address space, they must be trusted like the kernel. This situation motivated many projects to support untrusted extension modules. The most prominent of them are SPIN and VINO.

SPIN. SPIN [18] is an operating system that can be extended by using so-called event handlers. Event handlers are portions of untrusted code that can be downloaded into the kernel. Kernel operations trigger events [142] that can be processed by a handler. Memory protection is realized using a type-safe language to write handlers (Modula-3) and a trusted compiler that signs the handlers. Whether a handler is allowed to handle an event is decided by guards. If the handler runs longer than a certain time limit it can be terminated. Only handlers that are marked as EPHEMERAL can be terminated. Such a handler can only call other functions that are marked EPHEMERAL. No kernel function is marked ephemeral and therefore it is safe to abort an ephemeral handler.

Handlers are downloaded in the kernel as a SPINDLE. A system service in SPIN is decomposed into a SPINDLE, a library, and a user-level server. A SPINDLE can be downloaded into the supervisor address space, a library is loaded into the application address space and can contact the

SPINDLE via system calls, and the user-level server is used to maintain the state of the service that outlives the application.

VINO. VINO uses SFI for memory protection. (MiSFIT [161], the “Minimal i386 Software Fault Isolation Tool”). Extensions are called grafts. Resource consumption of grafts is monitored and an extension module can be aborted like a transaction. Running grafts in the kernel is usually motivated by performance concerns. The overhead of the transaction mechanism may render the extensions useless for the performance goal.

6 Summary

Current OS architectures are not able to cope with demands of today’s applications. We attribute most of the shortcomings to the used protection mechanism: the address space. Other protection mechanisms are better suited because they offer more flexibility and finer granularity. In this dissertation we develop an operating system structure using a type-safe instruction set as the only protection mechanism. Other protection mechanisms that use compiler extensions force the system to use a trusted compiler that signs the compiled code.We learn from Java that a general purpose language can be compiled to a type-safe instruction set. All of the extensible systems (SPIN, VINO, Exokernel) use a combination of several memory protection mechanisms. They use address-space protection in combination with type safety or Software Fault Isolation. We think that this unnecessarily complicates the architecture and advocate an architecture that relies on only one protection mechanism: type safety.

Using a different protection mechanism allows a new view at system problems of the past and reconsider if and how their solutions can be transferred to our new architecture. We learn from the Mach project to structure the system as a multi-server system to allow restartability, extensibility, and fault containment. We should not force components to use IPC if not necessary. We learn from SPIN that type-safe protection allows fine grained extensibility. Microkernels like Mach and especially the Exokernel emphasize that resources should be managed outside the kernel. Combining this with a type-safe protection system is a challenging task as such systems require support by a runtime system that contains many resource management mechanisms and policies.

Virtual machines, or hypervisors, are able to isolate systems running on the same hardware, but they have no support for sharing and efficient, typed communication.

Microsoft COM apartments demonstrate, that the discussed problems are not only of academic interest. Apartments subdivide a process in different execution environments. Objects must not be shared between incompatible apartments, i.e., apartments that use incompatible thread scheduling strategies. COM can not *enforce* the isolation between apartments, because a programmer error may lead to direct sharing of objects, for example when using a global pointer to access the same object in two or more apartments. A malicious component can not be isolated at all. This shows that a protection domain should not be separated from execution environment but instead the overhead associated with a protection domain should be reduced.

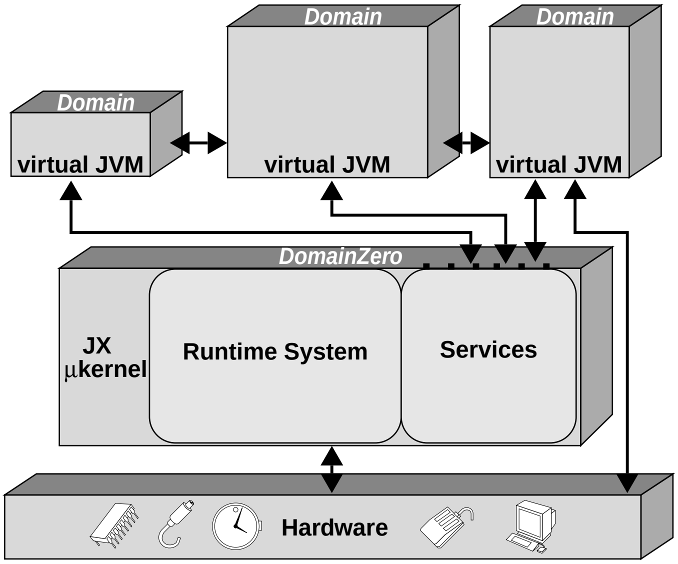
Previous operating systems that used type safety for protection were designed as single-user systems (Pilot, JavaOS) or used type-safety merely as an additional protection mechanism inside one address space and used address spaces to isolate the tasks of different users (SPIN). Using two protection mechanisms complicates the system architecture and application development and also requires two different resource protection mechanisms. We will show that type safety is sufficient as a protection mechanism to build a multi-user operating system.

CHAPTER 3 *Architectural Overview*

This chapter gives an overview of the architecture of the JX system. An in-depth description of design details and implementation aspects is left to the following chapters.

The JX system replaces hardware enforced address-based protection with protection based on the type-safety of the instruction set. This means that the system runs in a single-address space and never leaves the supervisor mode of the processor. The majority of the JX system is deployed in type-safe Java bytecode. A small microkernel, written in unsafe languages, such as C and assembler, contains the functionality that can not be provided at the level of the type-safe instruction set. This includes system initialization after boot up, saving and restoring CPU state, low-level protection-domain and component management, inter-domain communication, and garbage collection. The microkernel is logically split in two halves: a runtime system for Java bytecode and a number of system services (see Figure 3.1). The Java code is organized in components (Chapter 6) which are loaded into domains (Chapter 4), verified, and translated to native code. Domains can be thought of as a “virtual JVM”. Each domain provides a complete Java execution environment and is isolated from other domains. The effect is similar to running each JVM in its own process in a traditional operating system. Communication between domains is handled by using portals (Chapter 5). From a programmer perspective the portal mechanism is similar to Remote Procedure Call [1] or Remote Method Invocation [171], but it is more efficient and easier to use than these mechanisms.

Figure 3.1: Structure of the JX system



1 Modular microkernel

The JX microkernel is designed in the spirit of an Exokernel [61]. The microkernel is as minimal as possible. But in contrast to other Exokernels that use protection mechanisms provided by the hardware, the JX microkernel must contain a runtime environment to allow type-safe protection.

The JX microkernel (see Figure 3.2) contains a complete Java runtime system[1]. Therefore it is important that the kernel has a modular structure. We adhered to three important design principles while building the microkernel:

- avoidance of globally shared state
- avoidance of dynamic resource management
- moving as much functionality as possible to the level of the type-safe instruction set

The most important design principle is the avoidance of shared data structures in the microkernel. State is shared in domains that are outside the microkernel. Especially there are no dynamically growing data structures in the kernel; per domain structures are located in domain memory. This design is motivated by the fact that the microkernel is the only part of the system that can not be replaced while the system is running. Although different microkernels can be configured and built, deploying them requires restarting the complete system.

One design objective was the avoidance of dynamic resource management inside the microkernel. A traditional operating system implements many abstractions in its kernel; for example files and sockets. The implementation of these resources consumes a varying amount of resources. It is difficult to trace the resource consumption back to a resource principal.

A further design principle was to implement as much as possible of the system outside the microkernel at the Java-level. To accomplish this the microkernel contains “open spots” that are filled by invoking code that runs in a Java domain. Examples are resource management decisions, like CPU scheduling, interrupt handlers, and locks and condition variables. This reduces the amount of native code (C and assembler) and the number of transitions between native code and Java code. Such a transition must cooperate with the garbage collector, which makes it expensive and is a source of errors and robustness problems.

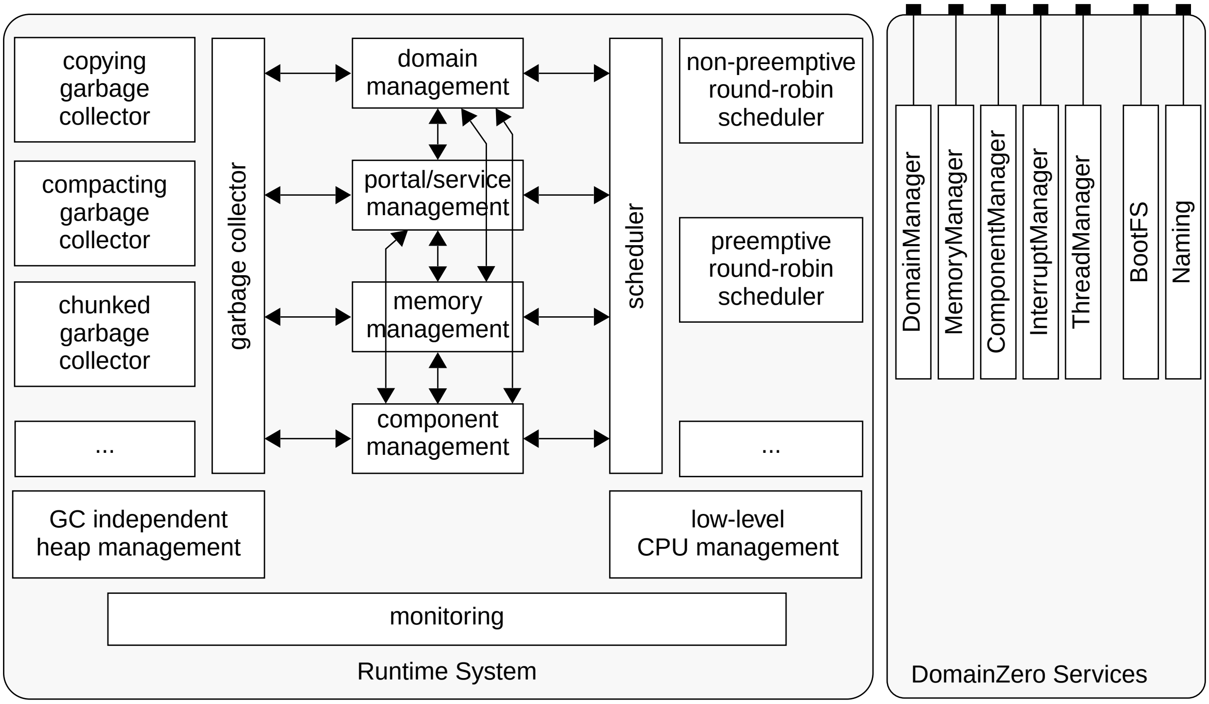
The microkernel does not use blocking mutual exclusion locks (mutexes) or semaphores. Critical regions are short and guarded by blocking interrupts on a single processor and using spinlocks on a multiprocessor.

There are several resources that must be managed by an operating system. Resources can be classified in physical resources and virtual resources. Physical resources correspond to real hardware resources, such as main memory, CPU, network bandwidth, and disk blocks. Virtual resources only exist as an artefact of the computational process. Virtual resources make it easier for applications to use the system and are usually realized using a complex interaction of physical resources. Examples of virtual resources are files, sockets, and database tables. The JX microkernel attempts not to provide any virtual resources and to limit its management of physical resources to a minimum. The only re-

[1] JX is a clean-room implementation of the JVM specification. We implemented it from scratch only using our expertise from a previous project [73]. To make it runnable on the bare hardware we adopted code from Linux and the OSKit [67], for example the printf implementation for debug messages.

Figure 3.2: Structure of the JX microkernel

The microkernel is logically split into two halves: a runtime system that implements a JVM and service implementations that allow other domains to communicate with the microkernel. All of these services are registered at the microkernel's name service. The name service itself is installed as the name service of DomainZero (the domain of the microkernel) and inherited to DomainInit, the initial Java domain.



source that is solely managed by the microkernel is main memory. Even the CPU management can be delegated to the Java-level.

2 Protection domains

The unit of protection and resource management is called a *domain*. All domains except DomainZero solely execute Java bytecode. *DomainZero* contains all the native code of the JX microkernel. It is the only domain that can not be terminated. There are two ways how domains interact with DomainZero. First, explicitly by invoking services that are provided by DomainZero. One of these services is a simple name service, which can be used by other domains to export their services by name. Secondly, implicitly by requesting support from the Java runtime system; for example, to allocate an object or to check a downcast.

Each domain has its own heap with its own garbage collector (GC). The collectors run independently and they can use different GC algorithms. Currently, domains can choose from four GC implementations. They are described in Chapter 8.

Each domain has its own threads. A thread does not migrate between domains during inter-domain communication. Memory for the thread control blocks and stacks is allocated from the domain's memory area.

Domains are allowed to share code - classes and interfaces - with other domains. But each domain has its own set of static fields, which, for example, allows each domain to have its own System.out stream. A task that runs in a protection domain can be an application program started on behalf of a user or a system service that is shared by many users.

To make domains completely independent from each other communication between domains is implemented as message passing and called *portal invocation*. Buffering messages is the responsibility of the source domain's execution environment. Synchronizing access to the message queue, flow control, and message acceptance is the responsibility of the target domain's execution environment.

3 Portals and services

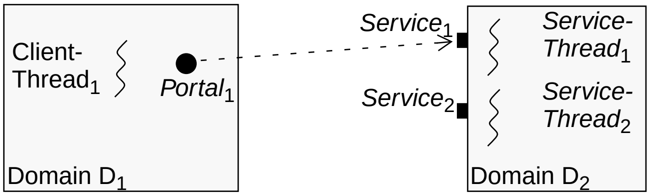
Portals are the fundamental inter-domain communication mechanism. The portal mechanism works similar to Java's RMI [171], making it easy for a Java programmer to use it. A portal can be thought of as a proxy for an object that resides in another domain and is accessed using remote procedure call (RPC).

An entity that may be accessed from another domain is called *service*. A service consists of a normal object, which must implement a portal interface, an associated *service thread*, and an initial portal. A service is accessed via a *portal*, which is a remote (proxy) reference. Portals are capabilities [48] that can be copied between domains. A domain that wants to offer a service to other domains can register the service's portal at a name server.

When a thread invokes a method at a portal, the thread is blocked and execution is continued in the service thread (see Figure 3.3). All parameters are deep copied to the target domain. If a parameter is itself a portal, a duplicate of the portal is created in the target domain. This means that the portal is passed by-reference.

Figure 3.3: Portal invocation

Domain D_2 exports services Service_1 and Service_2. Domain D_1 has obtained a portal to Service_1. When ClientThread_1 invokes a method at Portal_1 the thread is blocked and ServiceThread_1 is unblocked and executes the implementation of the service method.



As a convenience to the programmer the system also allows an object that implements a portal interface to be passed like a portal. First it is checked, whether this object already is associated with a service. In this case, the existing portal is passed. Otherwise, a service is launched by creating the appropriate data structures and starting a new service thread. This mechanism allows the programmer to completely ignore the issue of whether the call is crossing a domain border or not. When the call remains inside the domain the object is passed as a normal object reference. When the call leaves the domain, the object automatically is promoted to a service and a portal to this service is passed.

Every communication—even between a domain and the microkernel—is performed using portals. This uniformity makes the location of a service completely transparent to the functional code of a domain[1] and it allows kernel services, such as the name service, to be provided by a regular domain. It furthermore allows to enumerate all communication relationships of a domain.

4 Reusable system components

An important design principle of UNIX is the “small is beautiful” principle. An example of this principle is the combination of small programs to larger units by using pipes for communication between the processes that are created from the programs. Each process contains a simple processing step and delivers the data to the next process. The used communication mechanism, the pipe, is a powerful abstraction but it has several shortcomings:

- The unit of reuse (program) is identical to the unit of protection (process).
- It is not typed. All programs can be combined with each other but this includes combinations that make no sense and can even be dangerous. A mistake in connecting the programs can only be detected by analyzing the errors that are signaled by the programs that receive unexpected input.
- The mechanism is too much focused on data stream processing. A more advanced interaction between programs, for example a back channel, is difficult to realize.

All these points are addressed in the JX architecture. The first point is addressed in JX by using components as the unit to build more powerful abstractions and domains as the unit of protection and resource management. The component is the unit of reuse. It contains the complicated algorithms that should be reused rather than reinvented. The domain provides a runtime environment for components. It allows components to work together in one protection domain and shields them from attacks from outside the protection domain. Figure 3.4 shows two different system configurations that use the same set of components.

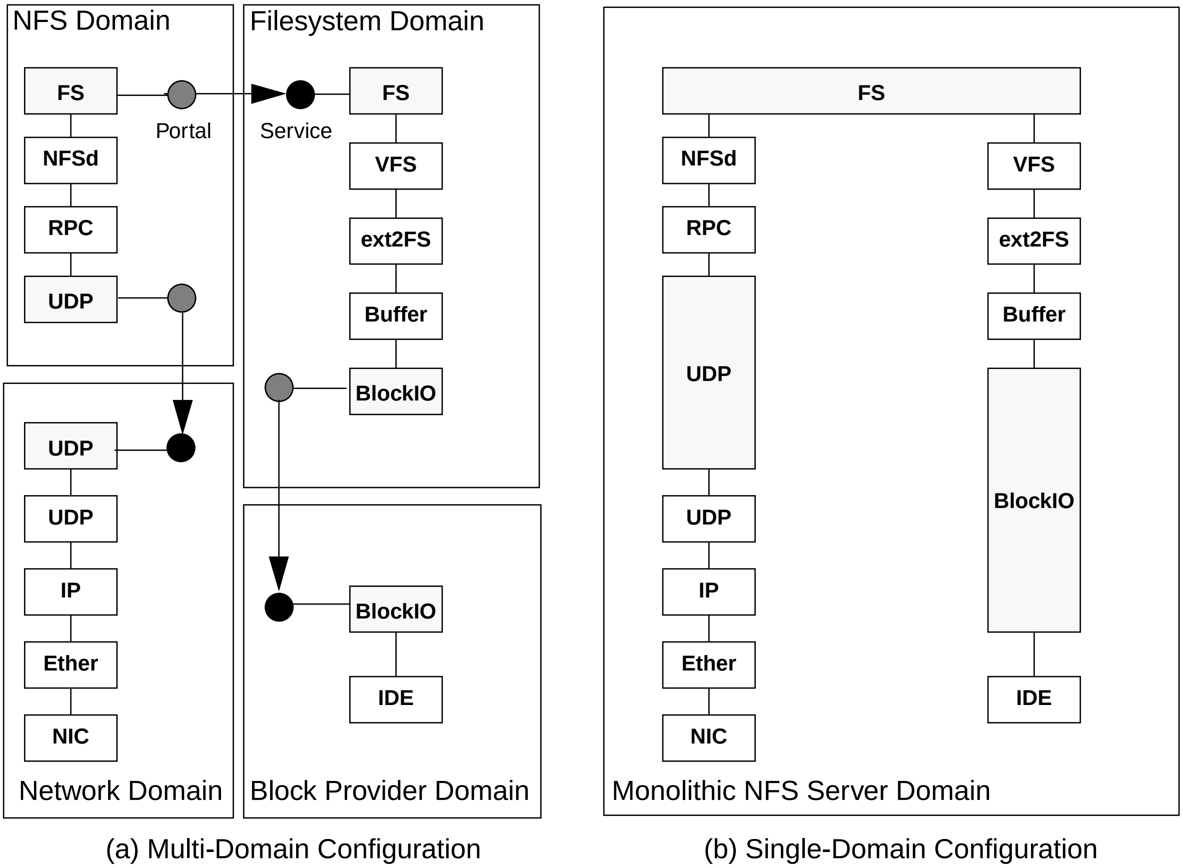
The second point is addressed by using a typed communication channel. This excludes several errors when connecting components. It is still possible to connect two components which use the same syntactical type but use it with a different semantics. These errors can be reduced by the careful design of types and an exact specification of the meaning of a type.

The third point is addressed by using the abstraction of a method invocation for communication between components.

[1] To support the concept of a trusted path between domains the system allows to identify the communication partner. This identification interface should only be used by code that supervises the security of the domain and is separated from the functional code of the domain. For more details see Chapter 10.

Figure 3.4: Two different system configurations using the same set of components

JX allows components to be used unchanged in different system configurations. A system can be configured with fine-grained protection domains that each contain very few—maybe even a single—components. Such a configuration isolates the components with respect to their resource consumption and fault properties. It also allows them to be restarted independently from each other and be replaced by a different implementation. In a system which does not requires these properties, all components can be placed in a single domain. In such a configuration communication overhead between components is removed and the compiler can further optimize the now monolithic system.



(a) Multi-Domain Configuration

(b) Single-Domain Configuration

5 Bytecode-to-nativecode translator

Because JX runs on an off-the-shelf CPU the type-safe instruction set must be translated to the instruction set of this CPU by a component called the *translator*. The translator is a program that runs in its own domain and that is trusted.

When a component is loaded its methods must be translated from bytecode to native code. There is a translator domain that is responsible for this translation. The microkernel communicates with the translator using portal calls. The translator receives the bytecode and delivers the native code component.

6 Fast portals

Several portals which are exported by DomainZero are *fast portals*. A fast portal invocation looks like a normal portal invocation but is executed in the caller context (the caller thread) by using a function call. This is generally faster than a normal portal call, and in some cases it is even necessary. For example, DomainZero provides a portal with which the current thread can yield the processor. It would make no sense to implement this method using the normal portal invocation mechanism.

A fast portal method can even by inlined because its semantic is part of the kernel interface and known to the translator. Instead of creating code to invoke the kernel function the translator creates code that is equivalent to the kernel function.

7 Bootstrapping

A boot loader loads the JX microkernel and all initial system components. After loading the kernel image and a boot module that contains the initial components the boot loader transfers control to the JX microkernel, which initializes global memory management, domain management, thread management, and portal and service management. Then DomainZero is created as a representative of the microkernel and the initial thread of DomainZero continues system initialization by initializing the interrupt controller and trying to detect a multi-processor system. Then DomainZero services are initialized and registered at the name service. Then the first domain executing only type-safe instructions, is created. We call this domain *DomainInit*. It usually executes a configuration script that lists the domains that should be started. A typical system starts domains for device drivers, file systems, network protocols, file servers, web servers, but also application programs.

Because the system components contain code in an instruction set that can not directly be executed on the CPU, the components must be translated to the native instruction set. The translator can either run offline or it can be run as a system service. Offline translation must be used for the translator itself and for components that are used for system initialization. In a static environment, such as an embedded system, offline translation may be sufficient for the whole system. In a desktop or server system new code must be loaded onto the system and the translator operates online.

8 Performance measurements

If not otherwise noted all performance measurements in this dissertation are performed using the following hardware:

- CPU: Pentium III (Katmai) Stepping 3, 500 MHZ
- Cache: L1 instruction cache 16 KB, L1 data cache 16 KB, L2 cache 512 KB
- DRAM: 384 MB
- 100 MHZ front side bus
- PCI BIOS: revision 2.10

- Chipset: Intel 440BX (82371AB PCI-to-ISA / IDE Xcelerator PIIX4)
- Video: Matrox Millenium G200 AGP
- Disk: Maxtor 91303D6, IDE bus
- NIC: 3Com 3C905B Fast Etherlink XL 10/100, 100BaseTX

The performance characteristics of simple Java operations on this platform when being compiled using the JX translator is presented in Table 3.1. Each operation was performed in a loop with 5,000,000 iterations. The time for running an empty loop was 11 ns per iteration and was subtracted from the measured time to obtained the numbers presented in Table 3.1.

Tab. 3.1 Performance of simple Java operations

Operation	Time (ns)
virtual method invocation[a]	40
non-virtual method invocation[a]	35
static method invocation[a]	33
assignment to an object field	5
creation of a new object	389

[a] About half of this cost is caused by the stack size check (see Chapter 8, Section 4.2).

For performance analyses of the microkernel and of the Java components the JX system contains an event logging facility. The Java components use a fast portal and the microkernel uses a macro to log an event. Table 3.2 shows the cycles required to log events. We executed a loop of 100 iterations that logged an event. The time between two events is 87 cycles. We disabled the first and second level cache and run the same loop. The time between two events increases dramatically to 3656 cycles. Assuming that the currently used part of the event log is in the cache with a high probability we subtract 87 cycles from each event transition in the following event diagrams.

Tab. 3.2 Event logging overhead

operation	time (cycles)	standard deviation (cycles)	n (iterations)
kernel logs event (caches enabled)	87	2	100
kernel logs event (caches disabled)	3656	66	100
Java component logs event (cache enabled)	96	4	1000

9 Summary

This chapter introduced the JX system architecture. JX is a single-address space operating system that allows protection without the use of memory management hardware (MMU). JX consists of a small microkernel and runtime system that provides hardware access and an runtime environment for Java programs. The system contains a bytecode-to-nativecode translator that is integrated with the kernel which allows certain optimizations, such as inlining of kernel code. The system is structured into domains which are independent JVMs. Code is loaded as a component. Inter-domain method invocation is performed by using portal calls. There is no object sharing between domains. Sharing is only possible via portals. Portals are stub objects with no visible state, i.e., no instance variables.

The JX architecture is based on a few abstractions that are explained in detail in the following chapters: domains (Chapter 4), portals and services (Chapter 5), components (Chapter 6), threads (Chapter 7), heap memory and memory objects (Chapter 8).

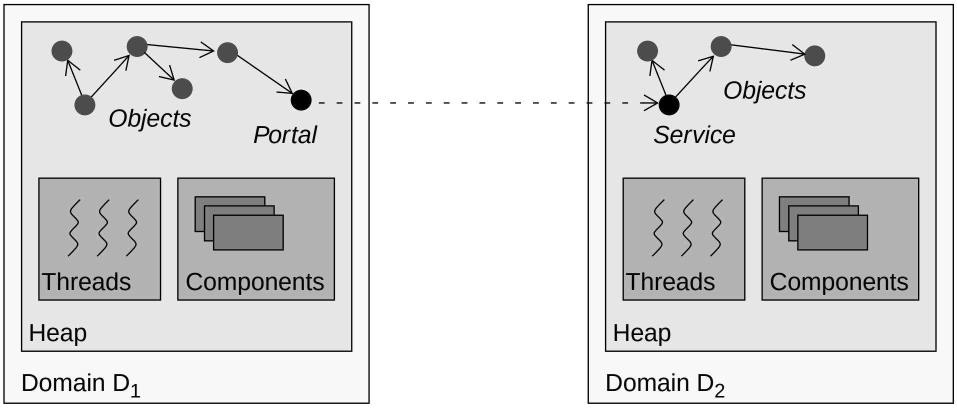
CHAPTER 4 *Domains*

This chapter describes the domain abstraction. It describes the lifecycle of a domain, its isolation properties, and the performance of domain related operations, such as domain creation.

1 Structure of a domain

Every domain contains a heap, a number of code components, threads, portals and services. The heap is used to store all data of the component, including thread controls blocks, stacks, portals, services, and components[1]. The heap memory is managed automatically by using a garbage collector.

Figure 4.1: Structure of a domain



2 Lifecycle of a domain

2.1 Creation

A domain is created by using a method of DomainZero's DomainManager service. Components are loaded into the domain or shared with other domains (see the Components chapter for details). After domain creation the domain is automatically activated by starting an initial thread. This thread starts

[1] In the current prototype the components are stored in a separate memory area.

executing an entry method that is specified during domain creation. An entry method is a static method that must be part of the components that are loaded into the domain. This entry method can be passed an array of String parameters and an array of Object parameters. As portals are subtypes of Object they can be passed in the Object array. It is also possible to pass the naming service to the domain which allows the domain to obtain more portals. In a former version of JX this was the only way how a domain could obtain a reference to the naming service. This required that the naming portal be passed around in the system either by passing it to newly created objects as a parameter of the constructor or by storing it in a static variable. This turned out to lead to very cluttered programs and we introduced an easier way to obtain the naming portal. There is a static method InitialNaming.getInitialNaming() that is part of the zero interface component. This method has an empty implementation but the translator emits a call to a microkernel function whenever it translates an invocation of this static method. The microkernel function returns the naming portal of the domain that issued the call. The interface of the Naming portal is described in Figure 4.2.

Figure 4.2: Naming portal

The Naming portal has two methods: a method to register a portal and a method to lookup a portal. This represents the minimal functionality of a name service. Additional methods that allow, for example, to unregister portals or receive a notification when a new portal is registered, can be provided by name services that run outside DomainZero and use an extended Naming interface.

```
public interface Naming extends Portal {
    void registerPortal(Portal portal, String name);
    Portal lookup(String name);
}
```

2.2 Termination

A domain can be terminated *explicitly* at any time[1] using a kernel service. The security system decides whether one domain is allowed to terminate another domain. A domain terminates *automatically* when there are

- no running threads,
- no services,
- no references to threads from the outside[2], and
- no registered interrupt handlers.

When all these four conditions hold the domain has no internal activity and can not be activated from the outside.

When a domain is terminated, all threads are stopped, all services are deactivated, and active service invocations are terminated by throwing an exception to the calling domain. Threads that wait for the completion of a portal invocation are terminated by signalling the termination to the called

[1] Device-driver domains can defer termination.

[2] References to threads from outside the domain could be used to unblock a blocked thread. One use of these references is to unblock a thread by a timer service that runs in another domain. See Section 3.3 of Chapter 9 for details about the timer service.

domain, which then can decide to terminate the call at once or when it reaches a save point to do so. The reference counts of all memory objects (see Chapter 8, Section 3.1) and services (see Chapter 5, Section 5) that are referenced by portals of the domain are decremented. The fixed memory area and the heap are released.

It is difficult to safely terminate threads or protection domains in systems that support fine-grained sharing of objects. A thread can hold a lock for an unlimited amount of time. Terminating the thread while it holds the lock may leave the data structure in an inconsistent state. Rudy et al. [149] present a technique based on bytecode rewriting that can safely terminate a thread. These problems only appear because threads and data structures are not unambiguously associated with a protection domain. In JX objects as well as threads are cleanly separated into domains and domains share information by using portals. The use of a portal—the invocation of a method—is clearly visible in the source code. Every portal invocation can throw an exception, for example, when the server domain is terminated. Whether the data structures of the server domain become inconsistent is not relevant, because it is terminated anyway. The client domain can detect and react to a server termination by catching and handling the thrown exception.

3 Isolation

The domain is the unit of protection and resource management. It is isolated from other domains with respect to resource usage and data access:

- *Sandbox*: Every domain can only access its own data or methods or invoke methods that are exported by other domains.
- *Faults*: Due to a software bug or hardware error a domain can fault. The domain must be removed or restarted without affecting the rest of the system. All resources of the domain, such as interrupts and memory, must be freed.
- *Resource usage*: Every domain can use a granted resource without affecting other domains.

Fault isolation can be used to guarantee non-stop system execution. A (reliable) monitor module and a (reliable) specialized TCP implementation could be used for remote administration even in case of a complete fault of the remaining system.

3.1 Resource management

The unit of resource accounting is a domain. Once a resource, such as memory or CPU, is committed to a domain it can be used without further accounting. For example, a domain manages its heap with large memory blocks. Once such a block is committed to the domain it can use an arbitrary memory management scheme to allocate objects in this block.

Resource types. There are two fundamental resources: memory and CPU time. Because these two resources are necessary for every computation we call them *primary resources*. Depending on the hardware of the system there are additional resources, such as a disk, a network interface, a display, and input devices. We call them *secondary resources* because they require primary resources to be used. Built upon primary and secondary resources are *virtual resources*. They are a virtualization of primary and secondary resources to allow (i) concurrent access by multiple users or programs and to allow (ii) easy and portable access to the resource by providing a high-level API.

The management of the two primary resources is explained in Chapters “Processor Scheduling” on page 73 and “Memory Management” on page 83.

When a domain obtains a portal to a service in another domain it acquires a virtual resource. When the domain terminates this resource must be released. This is done automatically when the reference count of the service drops to zero. The service may have allocated resources that are not released automatically. Therefore a method serviceFinalizer() is invoked at the service object when the reference count of a service drops to zero. This method is used for example in the window manager. The window manager creates windows for client domains. When such a client domain terminates the window service is deleted. All windows that belong to this client must be closed and associated resources must be released.

Shared services. When more than one domain obtains a portal to a service the service is shared between domains. Shared services usually implement high-level abstractions, such as files, or network connections. The implementation of these high-level abstractions maps the high-level virtual resource to lower-level virtual or physical resources. The domain that implements this mapping and provides the high-level resource is responsible for accounting the resource usage. How this accounting is done is outside the scope of the JX architecture. The resource container abstraction [13] can be used to separately account different services that run in the same domain but service different resource principals.

Protected data structures. Address-based systems must allocate data structures in the kernel address space[1], which requires a separate resource accounting mechanism for the memory that is consumed by these data structures. Type safety allows the kernel to maintain a data structure on the heap of a domain without the domain itself being able to access this data structure directly. We call these data structures protected data structures. Because protected data structures are stored on a domain’s heap they do not consume any global resources.

3.2 Execution environment

A domain provides an execution environment for components. Components with differing requirements are run in different domains. An execution environment includes

Memory management. A domain can use a specific garbage collection strategy (or no garbage collection at all).

Scheduling and synchronization. A domain can use a specific thread management strategy, including a custom implementation of mutexes and condition variables, and a scheduling strategy (non-preemptive or preemptive, priority-based, deadline based).

Portal and service management. It can be specified per domain and per service implementation class how incoming service requests are handled, whether a new thread pool is created for a service, and whether a new service thread is added to the thread pool if a service thread should block.

[1] An example are LPC message blocks in Windows 2000 [163].

Namespace. Code that runs in a domain needs portals to communicate with other domains. Initially a domain can access only a name service portal[1]. The name service for the initial domain, DomainInit, is provided by the microkernel, DomainZero. When a domain creates a new domain the creating domain can attach a new name service to the created domain, otherwise the name service of the creating domain is used. The current implementation of DomainInit creates a new name service and attaches it to all domains it creates. This guarantees that the name service of the microkernel is only used during bootstrapping the DomainInit naming service.

4 Domain ID

Several data structures in the microkernel and on a domains heap need to reference domains. Although it is sufficient to use a domain ID to unambiguously refer to a domain, using a direct pointer to the Domain Control Block (DCB) is much faster. In the current implementation DCBs are not moved but when a domain terminates the DCB of the domain is reused for domains that are newly created. This means that before a domain pointer can be used it must be checked whether it is still the DCB of the original domain. To be able to perform such a check the domain ID must be stored together with the domain pointer. Before the DCB is used this domain ID must be compared with the domain ID that is stored in the DCB. This requires the domain ID to be unique during the complete lifetime of the system, i.e. between two reboots. If a domain is created every nanosecond[2] a 64 bit range for domain IDs will last for 830 years. This means that the system must be booted at least once in 830 years.

5 Performance evaluation

Creation time. We measured the time to create a domain and compare it with the time to create a process in Linux. We created 100 domains and measured the time. On Linux we created 100 processes by using the fork system call and measured the time. We also used fork in combination with exec to load a new program. The results are presented in Table 4.1. JX domain creation includes the component loading and preparation (see Chapter 6 p. 59). It can be compared with a combination of fork and exec on UNIX. The factor of domain creation to Linux fork/exec is about 1.2 (Table 4.1).

Tab. 4.1 Domain creation time and destruction time

operation	time (μs)
JX domain creation	2245
Linux fork	263

[1] As has been described in Section 2, during domain creation a domain is passed an object array that also may contain portals. This feature is a recent addition to the system and it is only necessary to reduce the startup time of domains and saving costly name service lookups. In principle, passing only the naming portal is sufficient.

[2] Note that the domain creation time on our 500 MHZ PIII is about two milliseconds.

Tab. 4.1 Domain creation time and destruction time

operation	time (µs)
Linux fork/exec	1814

Memory footprint. Many applications, such as databases, agent platforms, and active networks must create large numbers of domains that each encapsulate a small amount of untrusted code. A small memory footprint of a protection domain is important for these applications. The memory overhead of a domain consists of the Domain Control Block, the domain specific code, the heap, and data structures that are allocated per-thread. Each thread needs a thread control block, which includes an area to save the CPU state and a stack. The stack consumes most memory and often the major part of this memory is unused. To avoid this situation a linked stack frame organization could be used. Stack frames are not taken from preallocated stack memory but allocated from the global memory pool and linked together. Although this approach does not waste memory for unused stack space it increases the method invocation overhead and can lead to fragmentation of the global memory system. Therefore we use a technique that is a combination of these two. We allocate a stack chunk that is relatively small. When there is no space to allocate a new stack frame we create a new stack chunk and link these two chunks together. When all stack frames of a stack chunk are freed, the chunk can also be released. To avoid allocation and deallocation of chunks that contain only one frame, deallocation must be delayed. It could be delayed until the previous chunk is half empty, but this would require a special check every time a stack frame is popped. Therefore we delay it, until the previous chunk is empty.

Figure 4.3 compares the memory requirements of a JVM process on Linux with the memory requirements of a JX domain. It is difficult to notice the JX line because a JX domain has a low memory footprint of 35 KB while a JVM process requires 3.4 MB. This data can be paged to secondary storage and only a part of it (the working set) needs to be resident in memory. But because of the large difference in DRAM and disk access speed paging severely affects performance.

The 35 KB can be split into 7 KB private code (including memory for static variables), 4 KB scratch memory used for portal parameter copying (see Chapter 5, Section 3.1), and 24 KB heap. The heap is populated by 10 KB character arrays that are created by the static initializers (for example by the Character class) and the “Hello World” application, less than 1 KB small objects, 8 KB for two stacks (4KB each for the initial thread and the garbage collector thread), 600 Bytes for two thread control blocks. This sums up to 20 KB. The remaining 4 KB are allocated but not used.These numbers indicate that scalability can still be improved by reducing the number of statically allocated character arrays and by reducing the stack size. One cost is not included in the 35 KB, that is the memory consumed by the Domain Control Block. The DCB contains storage space for the domain-local GC implementation (196 bytes), for the domain-local scheduler (128 bytes), and 484 bytes for domain management. Additionally it contains a list of services and a list of service thread pools[1]. These lists consume 4 * MAX_SERVICES + 4 * MAX_THREADPOOL bytes, where MAX_SERVICES is the maximal number of services exported by the domain and MAX_THREADPOOLS is the maximum number of thread pools for service threads (see Chapter 5 for details). Using a reasonable number of 50 services and 50 thread pools adds 400 bytes to the memory consumption of a domain.

[1] In the current prototype these lists are statically allocated when a domain is created. They should also be placed on the heap.

Figure 4.3: Protection domain memory footprint

The UNIX/JVM graph shows the memory consumption (data segment) of UNIX JVM processes. The JX domains graph shows the memory consumption of JX domains that execute a "Hello World" program.

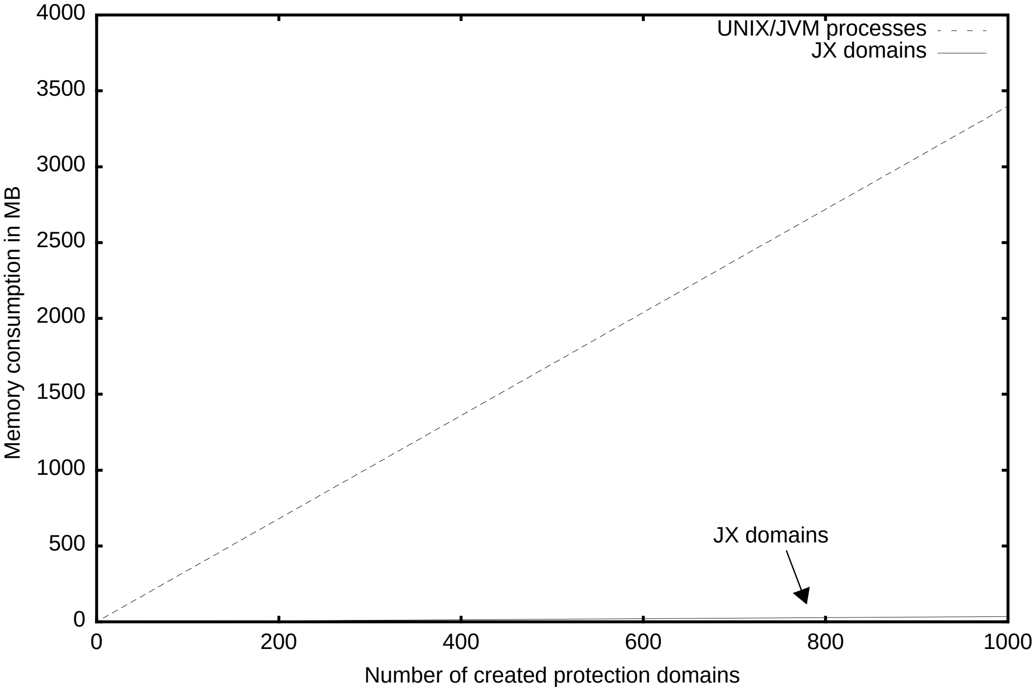
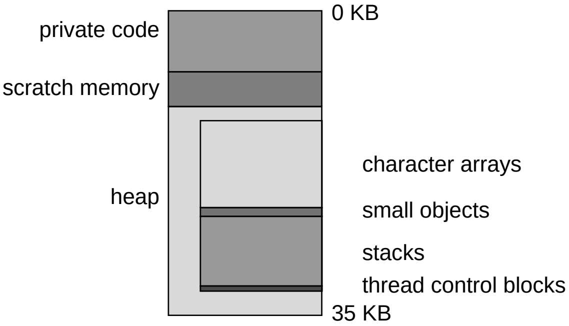


Figure 4.4: Break down of the memory footprint of a JX domain



6 Summary

This chapter described the domain abstraction. The domain is the unit of protection and resource management. A domain isolates code that is running in it against other domains. Each domain provides an execution environment that can be tailored to the needs of the code that runs in it. Domain creation is about 20% slower than fork and exec on Linux. The memory footprint of a domain is about one percent of the memory footprint of a JVM process on Linux.

CHAPTER 5 *Portals*

This chapter describes the portal invocation mechanism. The chapter first describes the mechanism from an application programmers perspective and then discusses its implementation and the interrelations between the portal invocation system and other parts of the system, such as the scheduler, translator, and garbage collector.

1 Design principles

The portal invocation is the central inter-domain communication mechanism. Its design was guided by the following principles:

- The communication mechanism must allow two domains to communicate but otherwise completely isolate them. The domains should depend on each other only during an exactly defined period of time.
- Communication must be possible between two mutually distrusting domains. Neither of them should be able to launch a denial-of-service attack or to obtain any data from the other domain without the permission of this domain.
- The communication mechanism must be similar to an abstraction that is well known to the programmer: the method invocation.
- It must be possible to extend the mechanism to a distributed communication mechanism.

There are several alternatives for inter-domain communication. A direct method call using the same thread and the same stack can be used. The problem is a bad isolation of domains. A thread would be able to run in different environments or protection domains. Another alternative is the use of asynchronous message passing. This is the typical model for IPC in message-passing microkernels. It is non-transparent and is usually used to build a synchronous message-passing mechanism.

2 Programming model

Portal invocation is designed adhering to two major principles: it must naturally fit into the object-oriented programming model and must be easy to use for the programmer. We use a communication mechanism that is similar to a Remote Procedure Call (RPC) and to Java’s Remote Method Invocation (RMI).

Domains communicate solely by using portals. An object that has the ability to be accessed from another domain is called *service object*. Each service is associated with a *service thread* or a thread pool. A *portal* is a remote reference that represents a service, which is running in another domain. Portals are capabilities that can be passed between domains.

A portal looks like a normal object reference. The portal type is an interface that is derived from the marker interface[1] jx.zero.Portal. A portal invocation behaves like a normal synchronous interface method invocation: The calling thread is blocked, the service thread executes the method, returns the result and is then again available for new service requests via a portal. The caller thread is unblocked when the service method returns. While a service thread is processing a request, further requests for the same service are blocked if the service uses only one thread to process requests.

The creation of a new service requires an interface and an implementation of the service (see Figure 5.1). The client can obtain a portal and invoke methods at the portal (see Figure 5.2). A server domain can provide an implementation of the service by implementing the portal interface and instantiating the implementation class (see Figure 5.3). The resulting instance is called a *service object*.

Figure 5.1: The interface of a portal

This example shows an interface MyService that is a portal interface. It contains a method sayHello(), which expects one parameter. The type of the parameter is a regular class.

```
interface MyService extends jx.zero.Portal {
      void sayHello(MyParam p);
}
class MyParam {
      int a;
}
```

Figure 5.2: Portal client code

This example shows how a client uses the portal defined in Figure 5.1.

```
MyService svc = ... ;
MyParam paramObj = ... ;

svc.sayHello(paramObj);
```

Figure 5.3: Service implementation

This example shows how the portal interface can be implemented.

```
class MyServiceImpl implements MyService {
      void sayHello(MyParam p) {...}
}
```

[1] A marker interface is an interface that contains no methods and is solely used to state that an object has a certain property. Marker interfaces are very common in Java. They are used to indicate that an object can be serialized (Serializable) or cloned (Cloneable).

Whether a method invocation at a portal interface is performed within the same domain or crosses a domain boundary is not visible by looking at the client code. This transparency is required for the reusability of the client code in different environments. But to be really reusable the client must be programmed in a defensive style. The code must not make assumptions about the sharing of parameter objects. They may be shared if the service runs in the same domain or copied otherwise. The code must also not make assumptions about the thread that executes the service code. It is the current thread if the service runs in the same domains or the current thread blocks and a thread in another domain executes the service method. This may lead to a deadlock situation if the client is itself a service that is called by the service in the other domain. To avoid such deadlock the portal invocation subsystem can spawn a new service thread whenever a service thread blocks. Blocking a client thread during a portal call also has consequences for object consistency in a non-preemptively scheduled domain. In such a domain no object locking is necessary but all objects must be in a consistent state when a thread blocks. This means that whenever a portal method is invoked, may it block or not, all objects must be in a consistent state.

3 Parameter passing

The fundamental question for an inter-domain communication mechanism is whether parameters should be copied or shared. The main motivation for data sharing is performance. Certain RPC implementations, such as Lightweight RPC (LRPC) [17], use shared communication buffers for parameter passing. Shared communication buffers, or more generally shared heaps, have several problems:

- They complicate memory and CPU accounting. Sharing a garbage collected heap leads to resource consumption that can not unambiguously be attributed to a domain.
- They lead to more complex programs. The programmer has to decide whether an object is used as an parameter and allocate this object on a shared heap.
- It is difficult to evaluate the data and fault containment properties of applications that share objects. Domains have communication relationships not only during the start and end of a method invocation but during the whole invocation and even beyond it. This would make it impossible to decide whether a fault in one domain leads to a fault in another domain without detailed analysis of the source code. The same argument applies to the containment of information, which is important for the security of the system.
- It would be difficult to transparently distribute such applications. Access to shared parameter objects would lead to message exchanges and does not scale well.
- It would require the use of write barriers to prevent illegal references between shared heaps and local heaps [10]. These write barriers slow down the very frequent operation of access to local objects. We expect that many applications with infrequent inter-domain communication will perform better with parameter copy and without write barriers.

JX does not use shared buffers for parameter passing because we consider accountability and maintainability more important than a small performance improvement for communication-intensive programs. The essential advantage of copying is a nearly complete isolation of the two communicating protection domains. The only time where two domains can interfere with each other is during portal invocation. This makes it easy to control the security of the system and to restrict information flow. Another advantage of the copying semantic is, that it can be extended to a distributed system without much effort. But copying parameters also poses several problems.

- It leads to duplication of data, which is especially problematic when a large transitive closure is copied.
- The object identity is lost during copying.

In practice, copying posed no severe performance problems, because only small data objects are used as parameters. Objects with a large transitive closure in most cases are server objects and are accessed using portals. Using them as data objects often is not intended by the programmer.

3.1 Copying algorithm

After the decision for a copying parameter passing semantics we now look at the details of the copying algorithm. The following questions must be answered:

When must objects be copied between domains? Objects must be copied during parameter passing and when passing the return value. When an exception is thrown during a service execution the exception object must be copied to the caller domain and thrown in the client thread. When a domain is created the name service portal must be copied to the new domain. In all cases we use the term *parameter graph* to denote the object graph that is copied between domains.

Who copies the parameters? Parameters can be copied by the caller thread (*client copying*) or the callee thread (*server copying*). We expect that many calls are waiting to be serviced and it wastes server heap space to copy parameters before a server thread is ready to process the call.

Are DoS attacks possible? The whole object graph that is reachable from the parameter reference is copied. A client domain can attack the server by using a large object graph. The same attack can be performed by the server using the return value. There are two solutions for this problem. The server can limit the amount of data that is copied per call or use a separate heap for objects that are copied into the domain from the outside. The separate heap complicates heap management and allows an attacker to cut off the domain from communication by flooding this heap. Therefore we use a per-call copy quota. Two other DoS attack would be possible if client copying would be allowed. A client could flood the server's heap by using many parallel portal calls each in its own thread and with parameters just below the per-call quota. A per-client quota would solve this problem. The second problem is more subtle. During parameter copying the heap must be locked. A DoS attack would be possible by performing calls with a sufficiently high frequency and parameter graph that is large but below the quota. Especially if the client is granted more CPU cycles and scheduled more frequently than the server domain the server could be locked out of its own heap. These attacks are not possible if the server copies the parameters and return values.

How are cycles detected? The parameter copying algorithm must transfer the whole object graph that is reachable from the parameter reference in the source domain and rebuild the object graph in the destination domain. This means that cycles in the object graph must be detected and reconstructed in the target domain.

During a portal communication object graphs are transferred in two directions: the parameters of the call are transferred from the client to the server and the return value is transferred from the server to the client. When we talk about the object graph transfer we call the domains source domain and target domain. When the client/server relationship is important we talk about client domain and server domain and distinguish between parameter passing and return value passing.

A simple algorithm (*COPY0*) uses a scratch memory to remember all references that are already copied and looks in this memory to detect already copied objects. The data structure of the scratch memory consists of pairs of references (source, destination). We call this data structure *mapping table*. If the reference is found as the source part of a pair the destination reference is used instead of copying the object.

As indicated by the performance measurements in Section 7.3 this algorithm exhibits poor scalability. A more sophisticated algorithm (*COPY1*) uses the source object to store a pointer to the destination object (forward reference). When storing the forward reference a part of the source object is overwritten. Thus this algorithm needs a second pass to correct all changed objects on the source heap. The problem is to obtain the overwritten data. The target object can not be used because source and target objects may not be identical. For example, a service object is promoted to a service and a portal is passed. The source graph then contains an object and the target graph contains a portal. Therefore the overwritten word must be remembered in the mapping table. But as the destination pointer is no longer needed the size of the mapping table is identical to COPY0. Forwarding pointers are removed by correcting all objects that are listed in the mapping table when copying is finished. Because this algorithm modifies source objects no other thread should be allowed to access the heap of the source domain during copying. If a GC of the target heap should become necessary, the copying can be aborted and restarted after the GC. Aborting the copying only requires resetting the forwarding pointers of the source heap. The copied objects on the target heap are garbage because they are not yet referenced.

3.2 Passing special objects

Portals. When a portal is passed as a parameter in a portal call, a corresponding portal is created in the target domain (see Figure 5.4).

Portals that return home. When a portal is passed to the domain in which its service resides, a reference to the service object is passed instead of the portal (see Figure 5.5).

Service objects. When a service object is copied to another domain it is automatically promoted to a service and a portal to the service is passed to the other domain. This substitution of a portal for the service object only works if the type of the reference is the interface and not the implementation class. To guarantee that this requirement holds, the target domain is not allowed to contain the class of the service object. If the target domain contains the implementation class an exception is thrown when a portal is transferred to this domain.

Passing objects by value increases the performance in a distributed system, but passing the “wrong” reference by value could duplicate the whole server. Only objects at the leaf of the object graph should be passed by value. The decision whether an object should be passed by reference or by copy should be left to the programmer. Marking a class as a service and the automatic service creation allow the programmer to make this decision while the details are handled by the system.

Figure 5.4: Portal to service in other domain

This parameter graph contains a portal to a service in a third domain (which is not the target domain). Such a portal is copied to the target domain and the reference count of the service is incremented.

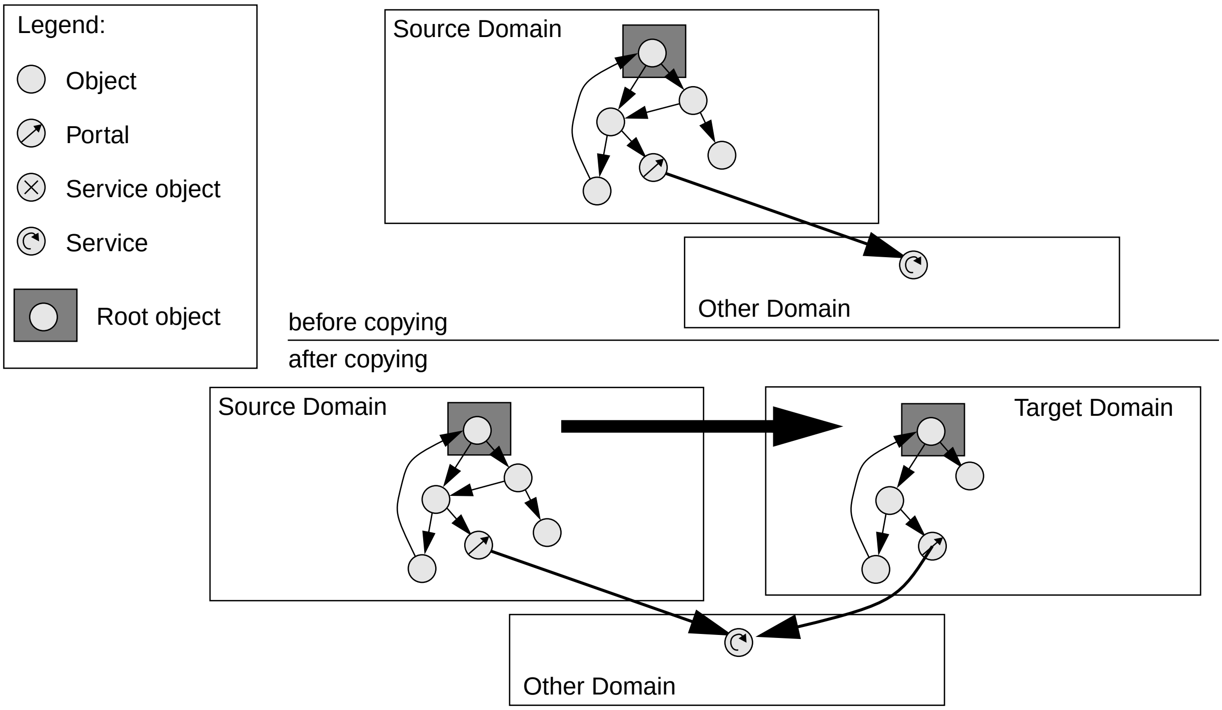


Figure 5.6 shows how an object graph that contains a service object is transformed during copying the graph to the target domain.

Several connection types between subgraphs can lead to problems or unexpected behavior (see Figure 5.7). In the following we call the object graph that is spawned by a service object *service graph* and an object graph that is spawned by a parameter object and pruned at service objects *parameter graph*.

We will look at the following types of references:

(a) Reference between service graphs

(b) Reference form service graph to parameter graph

(c) Reference from parameter graph to service graph

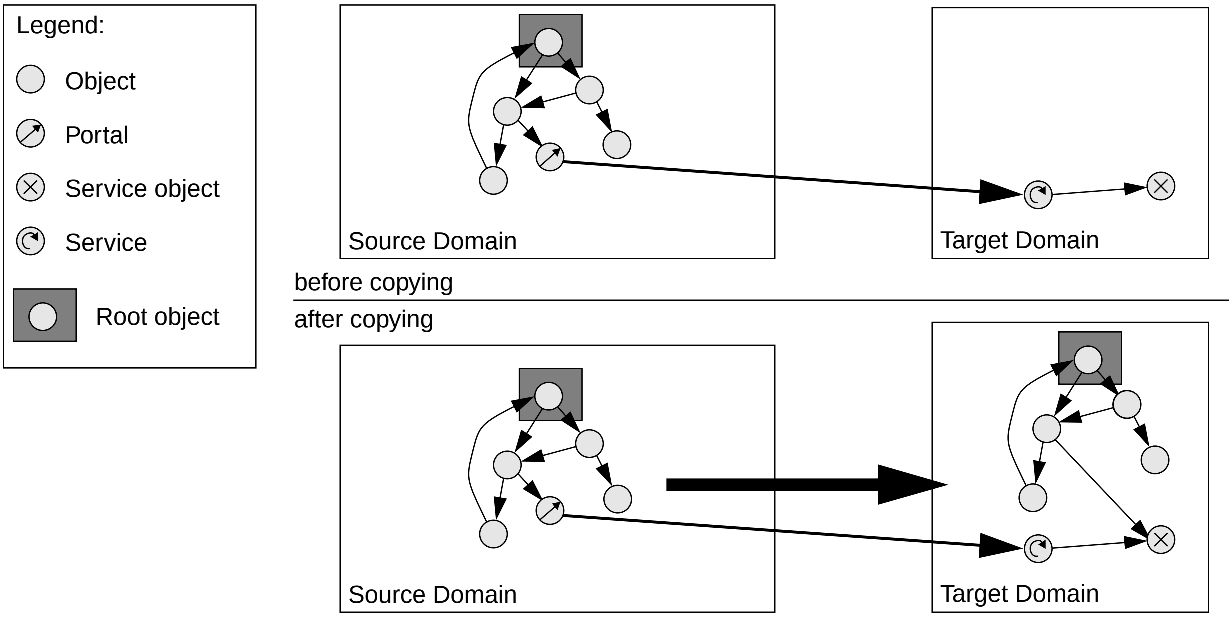
References of type (a) lead to no differences between the intra-domain and inter-domain behavior. Such references are no problem because objects that are behind a service object are not copied and therefore the reference between the service graphs is retained.

References of type (b) and (c) connect parameter graph and service graph through a regular object[1] This leads to a partial copy of the service graph to the target domain. Depending on the implementation this may lead to inconsistencies and therefore connections between a parameter graph

[1] Otherwise they are always connected through a service object.

Figure 5.5: Portal to service in target domain

The parameter graph contains a portal to a service running in the target domain. During graph copying every reference to the portal is replaced by a reference to the service object.



and a service graph should be avoided. We currently have no automatic mechanism to detect these connections at development time. But fortunately we also do not know any practical example for this kind of problems.

As a convenience to the programmer the system also allows an object that implements a portal interface to be passed like a portal. First it is checked, whether this object already is associated with a service. In this case, the existing portal is passed. Otherwise, a service is launched by creating the appropriate data structures and starting a new service thread. This mechanism allows the programmer to completely ignore the issue of whether the call is crossing a domain border or not. When the call remains inside the domain the object is passed as a normal object reference. When the call leaves the domain, the object automatically is promoted to a service and a portal to this service is passed.

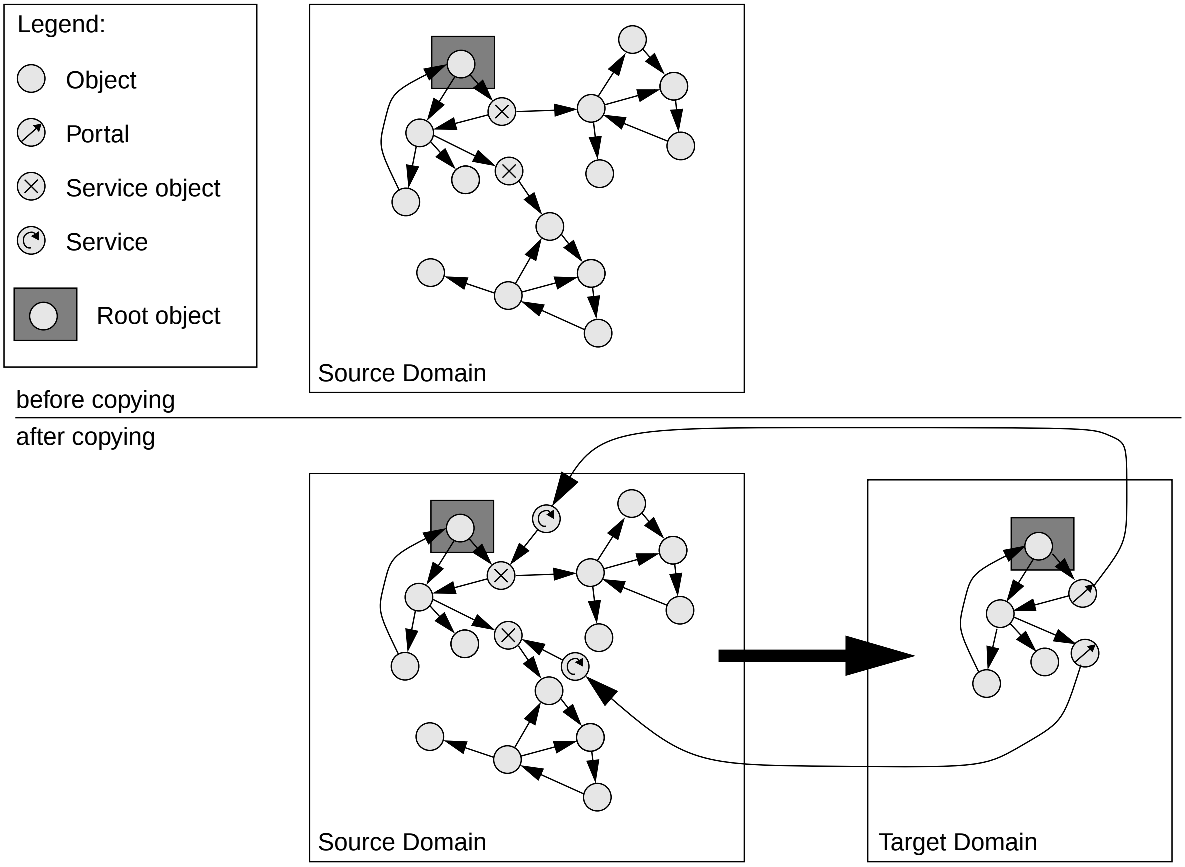
Automatic promotion of an object to a portal has a problem if the service object implements more than one portal interface. Which one should be used? There are at least two possible ways to resolve this ambiguity:

(a) When an object is passed during method call as argument or return value the type specified in the method signature could be used. This does not work with deep copied objects, because such an object can contain the class as the static type and not the interface. It also does not work when the portal method's parameter has a class type and not the interface type.

(b) Allow the programmer to choose by registering mappings. Type mappings must be identical for the type space that is shared between the two domains.

Figure 5.6: Pruning a large object graph at service objects

This figure shows an object graph that is spawned by the boxed object. The graph contains all objects that are transitively reachable from the root object.



(c) A service object must not implement two unrelated portal interfaces. It is, however, allowed to implement two or more portal interfaces that are in a subtype relation. In such a case the portal object is created by using the most specific type.

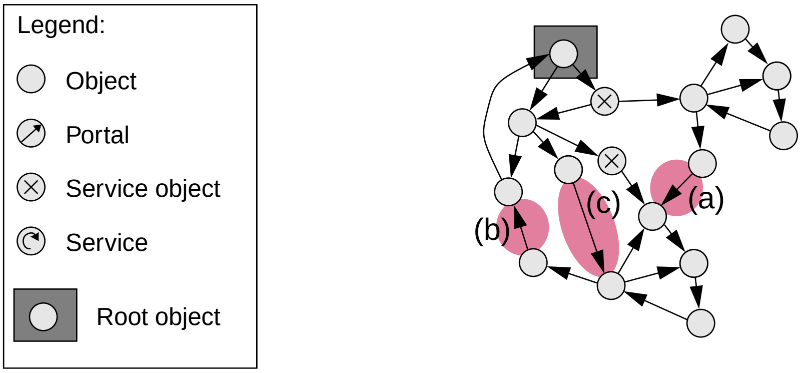
We choose variant (c) because it requires no programmer support (as variant (b)) and works in all practically relevant situations (unlike variant (a)).

3.3 Implicit portal parameters

In some situations additional information must be passed with a portal invocation. For this purpose it is possible to attach an object to the current thread. This object is passed as an implicit parameter during a portal call that is performed by this thread and the object is attached to the server thread. With this mechanism data can be passed along a portal invocation path. Implicit parameters can be used to pass credentials with a portal call. These credentials could be portals that testify that the client has

Figure 5.7: Problematic connections

There are three types of problematic connections: (a) a connection between service graphs, (b) a connection between a service graph and a parameter graph, (c) a connection from a parameter graph into a service graph.



certain rights. Not being forced to pass them explicitly makes the functional code more readable and easier reusable.

3.4 Summary

A service object is a regular object that is marked by using an empty interface. A service is a special object that has a reference to a service object and that can be referenced from outside the domain by a portal object.

4 Garbage collection and portal invocations

When parameters or return values are copied between domains the target domain may need a garbage collection. When a garbage collection occurs during parameter copying the copying is restarted.

During portal invocation the GC can run in the caller or in the callee domain. The portal call copies the object graph of the parameters from callee to caller domain. When the GC moves objects that are contained in this graph, it must also update the internal tables of the portal invocation system.

The parameter graph (in the client domain) may be changed during copying but objects must not be moved or deleted. When a GC happens in the client domain during parameter copying the mapping table (see 3.1) must be updated.

There are two occasions when a service thread has to wait for a garbage collection in another domain: during service creation and return value passing.

Service creation. A service is always created by passing a service object to another domain. During service creation new objects, such as the service control block, must be allocated. This allocation could trigger a garbage collection. Because parameters are copied by the receiver, the receiver thread waits for the completion of a garbage collection in the client domain.

Return value. The return value is copied from the server domain to the client domain in the service thread. If a garbage collection is necessary in the client domain the copying is restarted after garbage collection is finished.

5 Life-cycle management of services

As already mentioned portals and services can not be created explicitly by an application program. A portal is created during portal communication, when a service object is passed to another domain. Instead of copying the service object the portal invocation system creates a service in the source domain and a portal in the target domain. Portals and services are also automatically destroyed.

5.1 Service creation

As explained in Section 3.2 a service is automatically created when a service object is part of an object graph that is transferred between domains. All service data structures are created. The initial data structure is a service control block (SCB), which contains pointers to other data structures related to this service.. A former version of JX created a new thread for each newly created service, which renders a service a very heavy-weight abstraction and made it unusable for many puposes. The file system, for example, represents every open file as a service and using a thread, including a stack, for every open file is a lot of overhead. For this reason we changed the system to use thread pools and allow an newly created service to use an already existing thread pool or create a new one. When an existing thread pool is joined, the overhead of a service solely consists of 40 bytes to store the SCB.

5.2 Service termination

The service control block maintains a reference counter. Passing a portal to another domain increments the reference counter of the service. Garbage collecting a portal decrements the counter. Terminating a domain means “garbage collecting” all portals of this domain, which decrements the reference counters of their associated services. When the reference counter drops to zero the service is destroyed. When a service is destroyed the method serviceFinalizer() is invoked at the service object to inform the service about its destruction. The service can use this method to release all resources that it allocated to the service. In the window manager the window service uses this method to close the window. The serviceFinalizer() method is started in a new thread to release the resources of the service completely without waiting for the method to finish.

A service can also be destroyed even if its reference count is not zero. This happens when the user explicitly requests it or when the domain is explicitly terminated. All subsequent attempts to invoke the service will throw an exception in the caller thread. This essentially has the effect of revoking access to all portals to the service.

Using a reference counting technique has the advantage that garbage collection activity (incrementing reference counts) is performed during a portal call and during the garbage collection of a client domain (decrementing reference counts). There is no other interaction needed between domains. On the other hand reference counting can not detect cyclic data structures. A simple cycle that spawns two domains is created when an object graph of a service A contains a portal to a service B in another domain and the object graph of B contains a portal to service A.There is currently no mechanism to detect and resolve such cycles.

6 Implementation

Figure 5.8 shows the data structures that are used by the portal call implementation. A portal is an object that lives on the heap of a client. It contains the following information:

- A domain ID that identifies the domain in which the service is running.
- A domain pointer, which is used for a fast lookup of the domain control block (DCB). DCBs are reused after a domain terminated. If the domain pointer is valid the domain IDs from the DCB and from the portal must be identical.
- A service ID which denotes the service. We can not use a direct pointer to the service control block (SCB), because SCBs are located on the heap of a domain and may be moved by a compacting or copying GC. Using a direct pointer would require updating data of foreign domains during a GC cycle. It is also not an indirect pointer because the whole domain may be moved or swapped out, which would invalidate all pointers. Therefore each domain has a service table that consists of pointers to service control blocks (SCB). The service ID is an index into the this service table.

The SCB contains information which is relevant to maintain the service and perform service invocations. It contains a list of waiting threads, a reference to the service object and a thread in which the service execution takes place. To detect unreferenced services the SCB contains a reference counter that is incremented whenever a new portal to the service is created and decremented when a portal is destroyed.

Every TCB has a reference to it's current client (if any). The reference, called mostRecentlyCalledBy, is updated by the GC when the client TCB is moved in the client domain (see Chapter 8, Section 2.2).

During a portal communication the client thread (caller) is blocked and a service thread in the service domain (callee) is unblocked. A service is associated to a pool of service threads. To execute a call one thread is removed from this pool and added to the pool when the service execution is finished. During the execution of the service the client thread is in the state PORTAL_WAIT_FOR_RET. When no threads are available in the pool the client thread is blocked in the state PORTAL_WAIT_FOR_RCV. A thread in the thread pool is in the state PORTAL_WAIT_FOR_SND.

Figure 5.9 shows a flow diagram of the portal send operation. This operation is executed in the client thread and accesses the service data structures of the server domain. Atomic operations are implemented by disabling interrupts on a uniprocessor and using spinlocks on a multiprocessor. A previous version of send first checked whether the target domain is the current domain and executed the portal call as a normal method invocation. Then the functionality of passing a portal as the service object was added. This leads to the invariant that the heap contains no portals to its own domain and the *current-domain check* could be removed from send.

Figure 5.8: Portal data structures

A portal object contains a domain ID and a service ID. The portal does not depend on the physical location of the domain or the service within the domain. The serviceID is used to look up the service control block. The vtable of the portal object contains pointers to the implementation of the send() function.

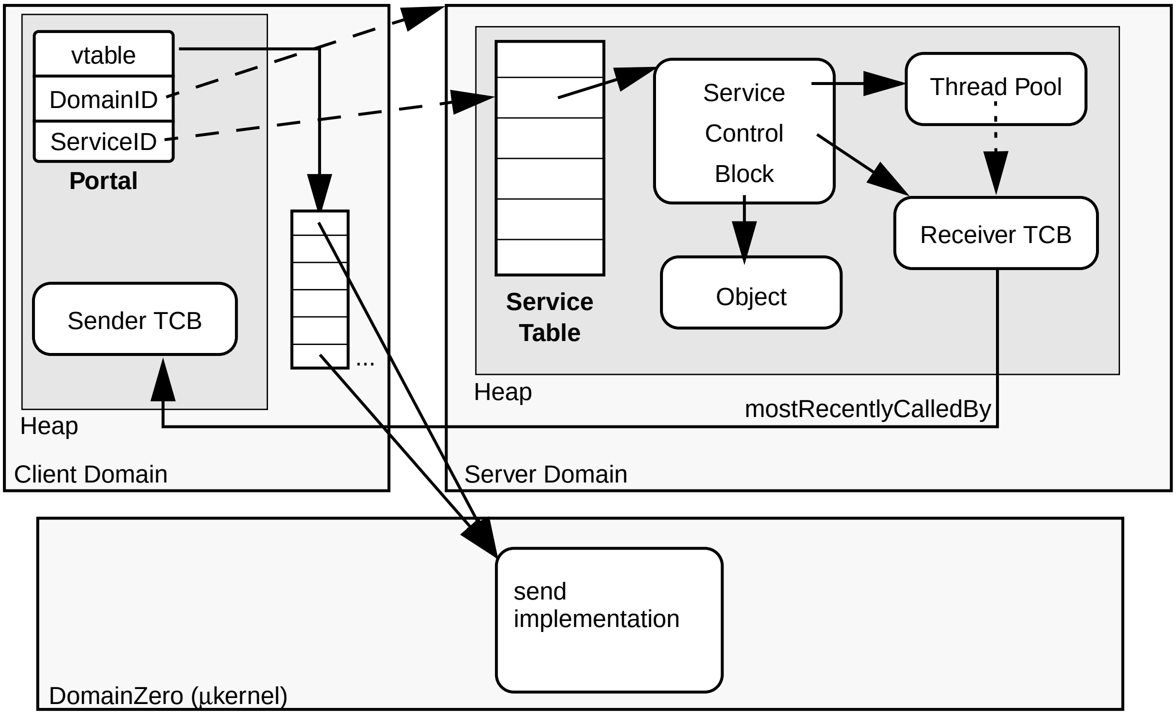


Figure 5.10 shows the flow diagram of the receive operation. This flow diagram has no end state because a receiver thread always blocks waiting for the next sender once it finished processing a request.

6.1 Thread state transitions during a portal call

The client thread changes its state during a portal call (see Figure 5.11). A single wait state is not sufficient because the different phases of the service thread execution impact the client domain in different ways. The first phase is the parameter copying phase (PORTAL_WAIT_FOR_PARAMCOPY). A garbage collection can be triggered during this phase. After copying the parameters the thread waits for the execution of the service to finish (PORTAL_WAIT_FOR_RET). No interaction between service and client thread happens in this phase. When the service is finsihed the return value is copied to the client (PORTAL_WAIT_FOR_RETCOPY).

Only threads that are in the state PORTAL_WAIT_FOR_PARAMCOPY or PORTAL_WAIT_FOR_RETCOPY interact with other domains.

Figure 5.9: Send operation

This send operation is executed by a thread that invokes a method at a portal object. The operation requires atomicity to check the state of the target domain, the target service, and to enqueue the thread at the service.

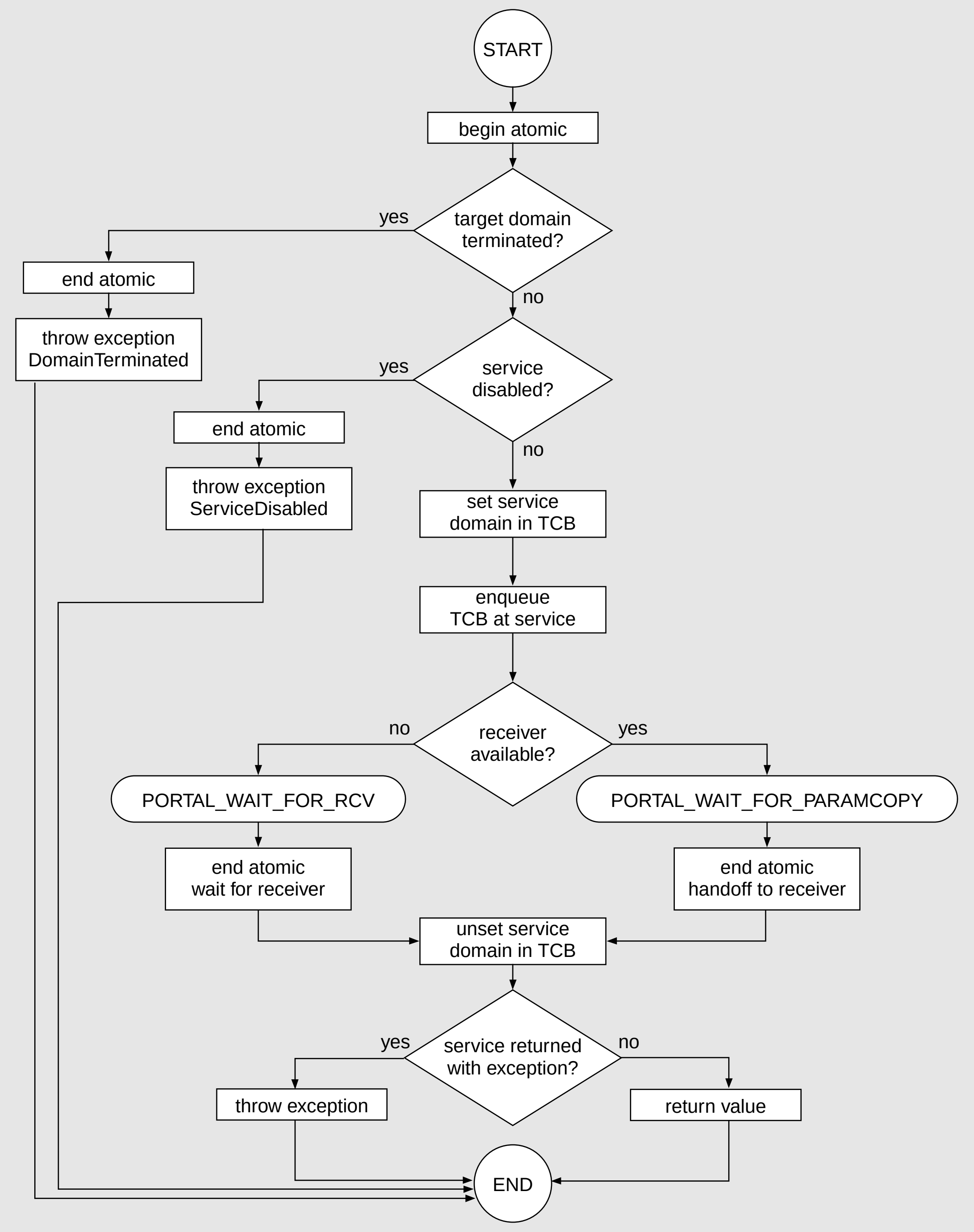

START
begin atomic
target domain terminated?
yes
no
end atomic
throw exception DomainTerminated
service disabled?
yes
no
end atomic
throw exception ServiceDisabled
set service domain in TCB
enqueue TCB at service
receiver available?
no
yes
PORTAL_WAIT_FOR_RCV
PORTAL_WAIT_FOR_PARAMCOPY
end atomic wait for receiver
end atomic handoff to receiver
unset service domain in TCB
service returned with exception?
yes
no
throw exception
return value
END

Figure 5.10: Receive operation

The receive operation is executed by threads that process service invocations. These threads are either executing a service method (the *process* box in the flow diagram) or are blocked in a service thread pool (caused by *block in pool*). The sequence *end atomic; handoff to sender; begin atomic* can be implemented very efficiently by storing the interrupt mask in the thread context. Saving or restoring this mask requires only two processor instructions on the x86.

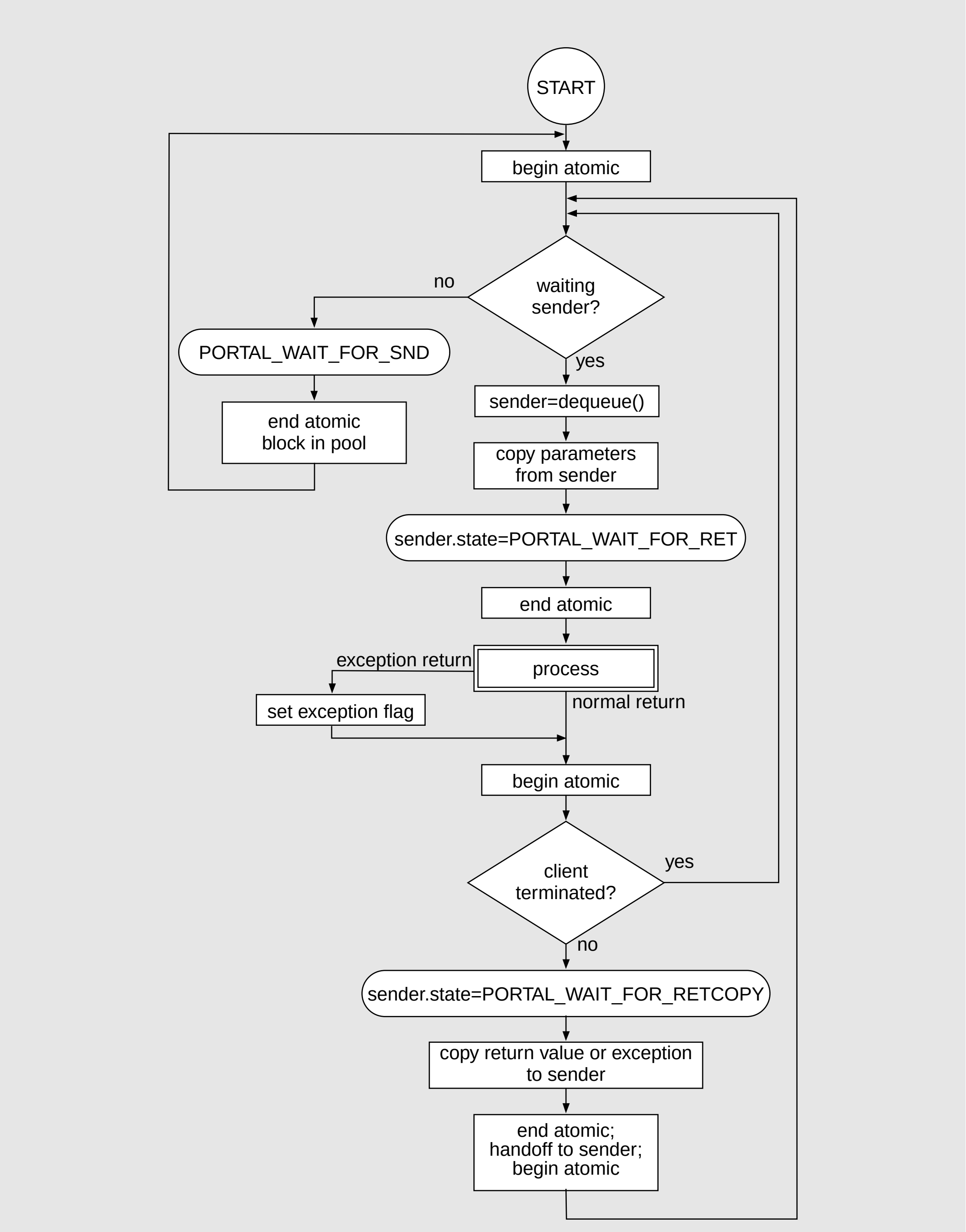
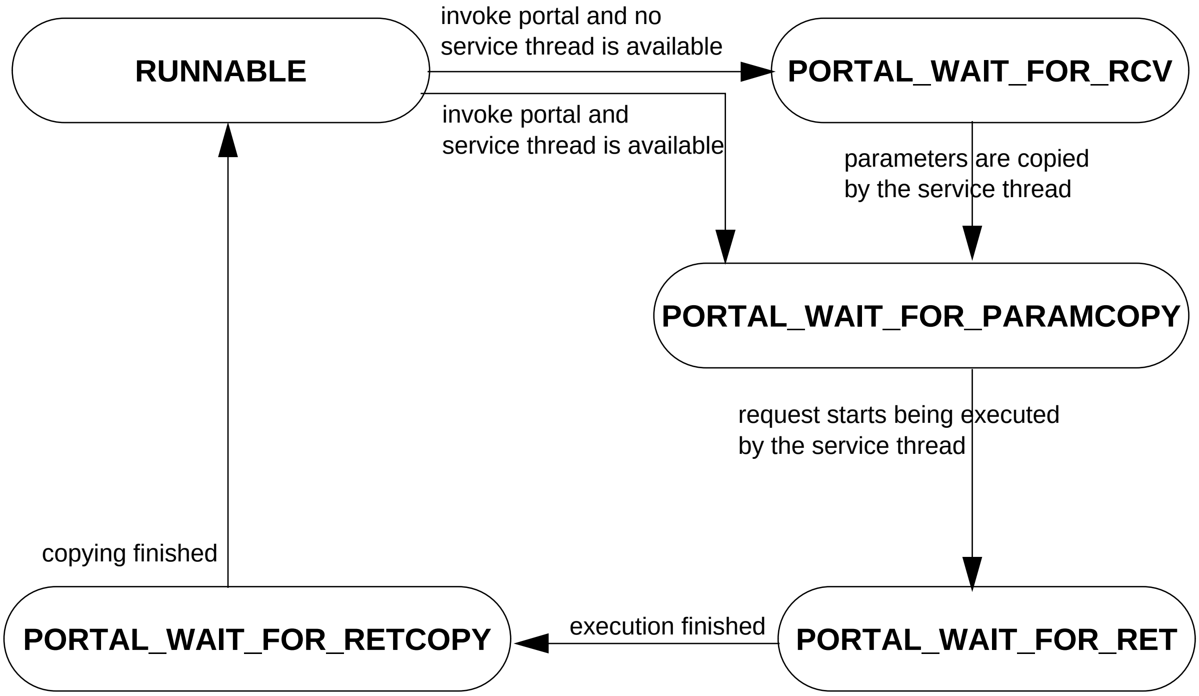


Figure 5.11: States of a client thread during a portal call


RUNNABLE
invoke portal and no service thread is available
PORTAL_WAIT_FOR_RCV
invoke portal and service thread is available
parameters are copied by the service thread
PORTAL_WAIT_FOR_PARAMCOPY
request starts being executed by the service thread
copying finished
PORTAL_WAIT_FOR_RETCOPY
execution finished
PORTAL_WAIT_FOR_RET

6.2 Premature call termination

A service execution can be terminated when the client domain terminates or requests an abort of the service execution. Systems that use a migrating thread RPC, such as LRPC, must cope with this special case. The server thread can not be terminated at arbitrary time because this could lead to inconsistent data structures in the server. LRPC provides a mechanism to initialize a new thread from a “captured” thread. The new thread returns as if the call was aborted with an exception. The original thread continues to execute in the server domain and is destroyed when the server returns.

As the portal invocation is not implemented using a migrating thread model it can abort a service execution by setting an abort flag in the thread control block of the service thread and raising an exception in the client thread. The service thread can check for the abort flag at a safe point and also terminate by raising an exception.

6.3 The translator and portal communication

The Java source code and the bytecode of components need not be changed when components are collocated in the same domain or dislocated in separate domains. But the behavior of a method call at a portal interface must be different in the two cases. In the first case the object that implements the portal interface should be called directly, in the second case the microkernel must be invoked to perform the portal call. In both cases the same bytecode instruction is used: invokeinterface. This instruction is also used for all other interface invocations. Should an interface invocation at a portal interface treated specially, in other words, should the translator emit special code when the interface is a subinterface of jx.zero.Portal?

In an initial JX implementation a portal method invocation was treated specially by the translator. The translator emitted machine instructions that directly transferred control to the microkernel. This method has several shortcomings. It limits flexibility because an ordinary object could not be used instead of the portal proxy and it complicates the translator.

In the next generation of JX the runtime system generated jump code (trampoline) when a new portal was created in a domain. This trampoline code contains the method table index. All method table entries are directed to this jump code. Now a portal proxy looks like a regular object. The major disadvantage of this implementation is the requirement to dynamically allocate code memory for the trampoline code. As this code can not be moved[1], it can not be allocated on the heap but must be allocated in the non-movable memory area.

The current generation of JX emits the same code for a portal invocation and an interface invocation and does not generate trampoline code. The entries in the vtable of a proxy directly point to the send() kernel function. This creates the problem that the send() function no longer has information about the method index and it therefore does not knwo which method it should invoke and how many parameters it should copy. We solved this problem by passing the method index in a well-known register to each interface method. As the method index is know when the bytecode is translated to the native code the translator emits instructions that move it into the register.

6.4 Fast portals

Several portals which are exported by DomainZero are *fast portals*. A fast portal invocation is executed in the caller thread. This is generally faster than a normal portal call, and in some cases it is even necessary. For example, DomainZero provides a portal with which the current thread can yield the processor. It would make no sense to implement this method using the normal portal invocation mechanism.

A fast portal has a vtable that is constructed by the kernel and points to functions in the kernel (see Figure 5.12). The vtable entries point to functions in another domain. These functions must have the following properties:

- The code must not be moved.
- The domain that hosts the functions must never be terminated.
- The must not depend on the context it is executing in. Because threads are not switched the code executes in the context of the caller thread.

As fast portals could be used to circumvent the isolation of domains we currently allow only the microkernel to create them.

In contrast to a regular portal a fast portal has no corresponding service object that contains the state of the service. If such information is necessary it must be directly stored in the portal object

[1] Moving it would require an update of the method table entries that point to the trampoline code. This means that the method tables must be scanned during a GC; they must be part of the root set. This is not supported in the current version of JX.

(additional data in Figure 5.12). As the portal object has an interface type this data can not be accessed by the client domain. Only the microkernel can use this data because it is not forced to obey typing rules.

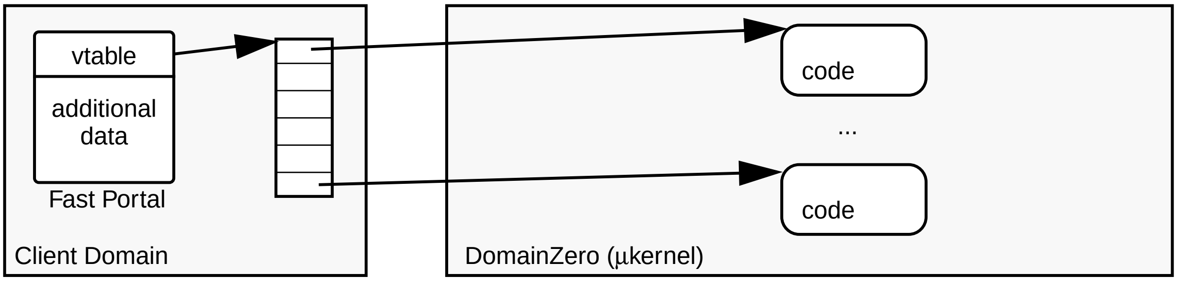
There are two reasons for the use of fast portals:

- the functionality can not be provided when threads and the contexts are switched
- the functionality is impractical if the thread switch overhead must be paid

Examples for fast portals are the ThreadManager and memory objects. The ThreadManager allows to influence the state of the current thread by yielding the processor or by blocking. This could not be realized if the ThreadManager would be a regular portal. Memory objects represent areas of memory. To be practical methods to read and write this memory must execute very fast.

Figure 5.12: Fast portals

A fast portal is structured like a regular object but uses a vtable that is constructed by the kernel and points to functions in the kernel. The fast portal object can contain additional data that is not accessible from Java code.



7 Performance evaluation

7.1 Null portal invocation

Table 5.1 shows the cost of a portal invocation and compares it with other systems. This table contains very different systems with very different IPC mechanisms and semantics. L4Ka is an L4[116]-based microkernel developed at the University of Karlsruhe. Fiasco is a C-implementation of the L4 interface. As the JX portal invocation code is also written in C and not hand-tuned assembler code it is more appropriate to compare the portal invocation cost with the Fiasco IPC cost than with one of the kernels with highly tuned IPC path. The J-Kernel is capability system implemented on top of an unmodified JVM. The IPC operation is a method invocation at a capability. This invocation is very fast because it does not include a thread switch. The resource management problems that are caused by such a design are discussed in Chapter 11. Alta is a extended JVM developed at the University of Utah that supports a process concept for Java and an IPC operation to communicate between such processes[1].

Tab. 5.1 IPC latency (round-trip, no parameters)

System	IPC(cycles)
L4Ka (PIII, 450MHz) [194]	818
Fiasco/L4 (PIII 450 MHz) [196]	2610
J-Kernel (LRMI on MS-VM, PPro 200MHz) [87]	440
Alta (PII 300 MHz) [11]	27270
JX (PIII 500MHz)	750

7.2 Portal invocation path

By instrumenting the microkernel to generate events during the portal invocation we can analyze the invocation cost in more detail. Figure 5.15 shows the event diagram. The most common sequence of events, which is also measured in the portal invocation performance test, is denoted by a dotted line in Figure 5.15. The total sum of the cycles consumed along this path is 691, which corresponds well with the 750 cycles we measured in the performance test. The most cycle consuming operations are the thread switches between sender and receiver and back with 168 cycles and 179 cycles. These 347 cycles or 694 ns could be reduced by performing the thread switch lazily: the service method continues to run on the client stack. Only when this optimization would become visible, a regular thread switch is performed. The optimization would become visible if, for example, the thread blocks or one of the domains or threads is terminated. All these places must be identified and modified to cope with the lazy thread switch. As this requires substantial changes to the system we did not yet attempt to implement this optimization.

7.3 Parameter copying cost

To evaluate how parameter passing affects portal call performance we performed the following experiment. We implemented a portal method that has one parameter. This parameter is an object that contains only one field: a reference to an object of the same class. We used this class to create linked lists of different length and passed them as parameter. This test measures the scalability of the parameter copy implementation, its ability to handle very large and/or deep object graphs. The heap size of the target domain was 140 MB. The heap of the source and target domains are garbage collected before each test. The target domain did only provide the single service we used in the measurements. For the Java RMI test we used Sun JDK 1.1.8 for Linux. This JVM's object serialization implementation is not able to transfer large object graphs because it uses a recursive implementation[2]. Copying

[1] The excessive cost of IPC in Alta was probably be one of the reasons that a follow-on project of the University of Utah, the KaffeOS [10], completely abandoned the IPC primitive and used shared heaps.

[2] The same problem exists in other JVM implementations, for example IBM's J2RE 1.3.0 for Linux.

Figure 5.13: Events logged during the send operation

To analyze the cost of a portal invocation we instrumented the portal invocation code to emit events at interesting positions. This figure show the flow diagram of Figure 5.9 and marks the points where events are logged.

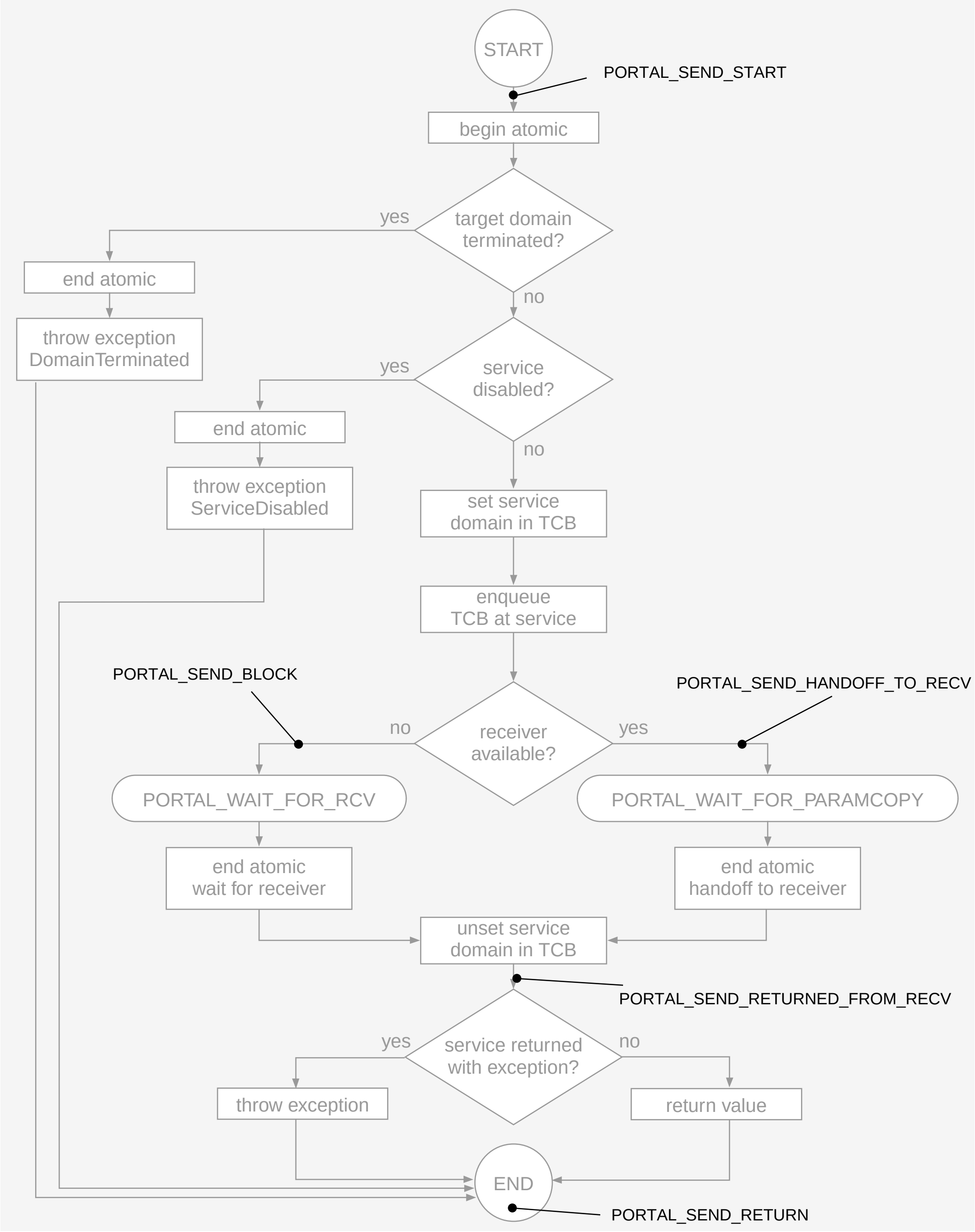

START
PORTAL_SEND_START
begin atomic
target domain terminated?
yes
no
end atomic
throw exception DomainTerminated
service disabled?
yes
no
end atomic
throw exception ServiceDisabled
set service domain in TCB
enqueue TCB at service
PORTAL_SEND_BLOCK
PORTAL_SEND_HANDOFF_TO_RECV
receiver available?
no
yes
PORTAL_WAIT_FOR_RCV
PORTAL_WAIT_FOR_PARAMCOPY
end atomic wait for receiver
end atomic handoff to receiver
unset service domain in TCB
PORTAL_SEND_RETURNED_FROM_RECV
service returned with exception?
yes
no
throw exception
return value
END
PORTAL_SEND_RETURN

Figure 5.14: Events logged during the receive operation

This figure shows at which points of the receive operation events are logged.

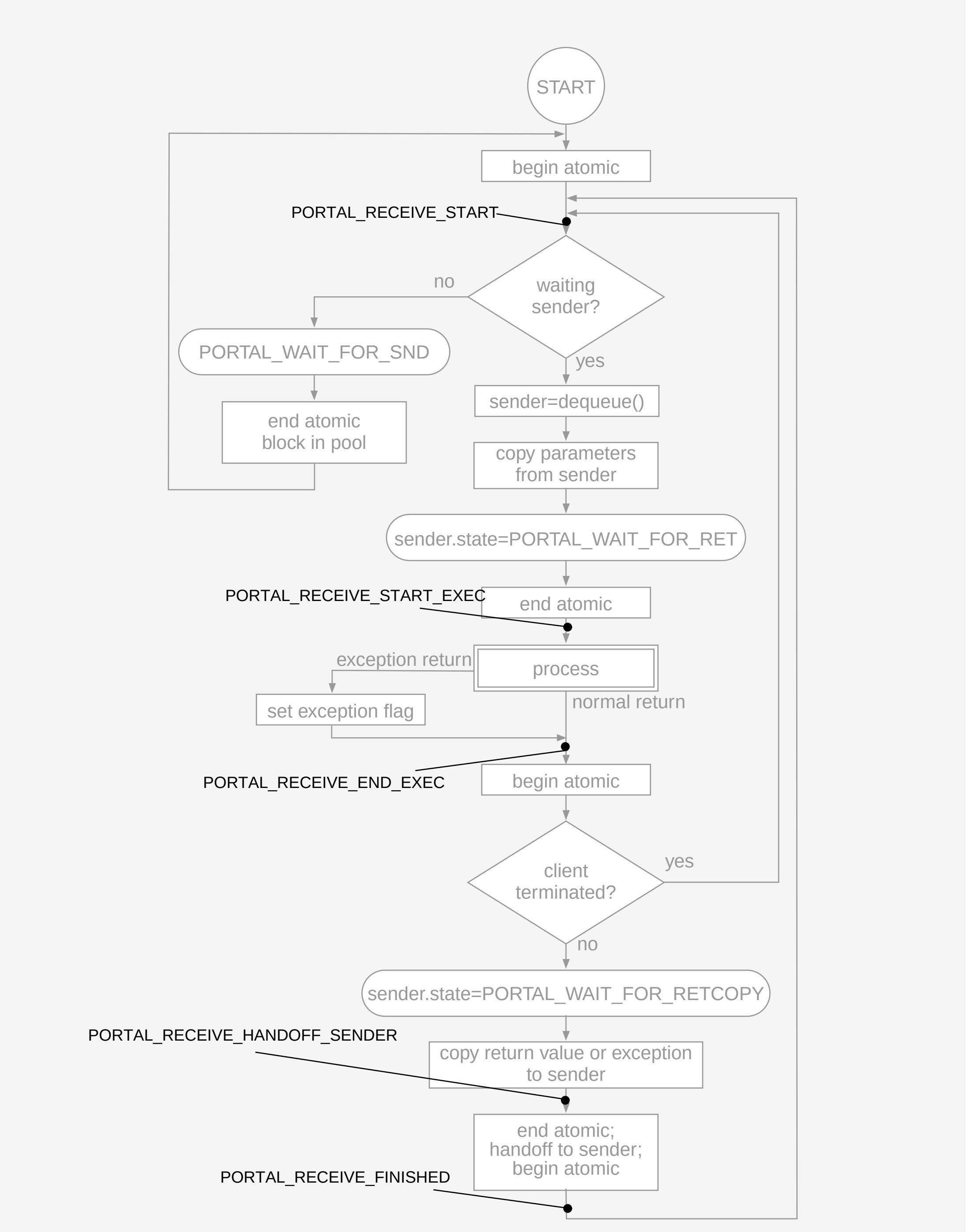
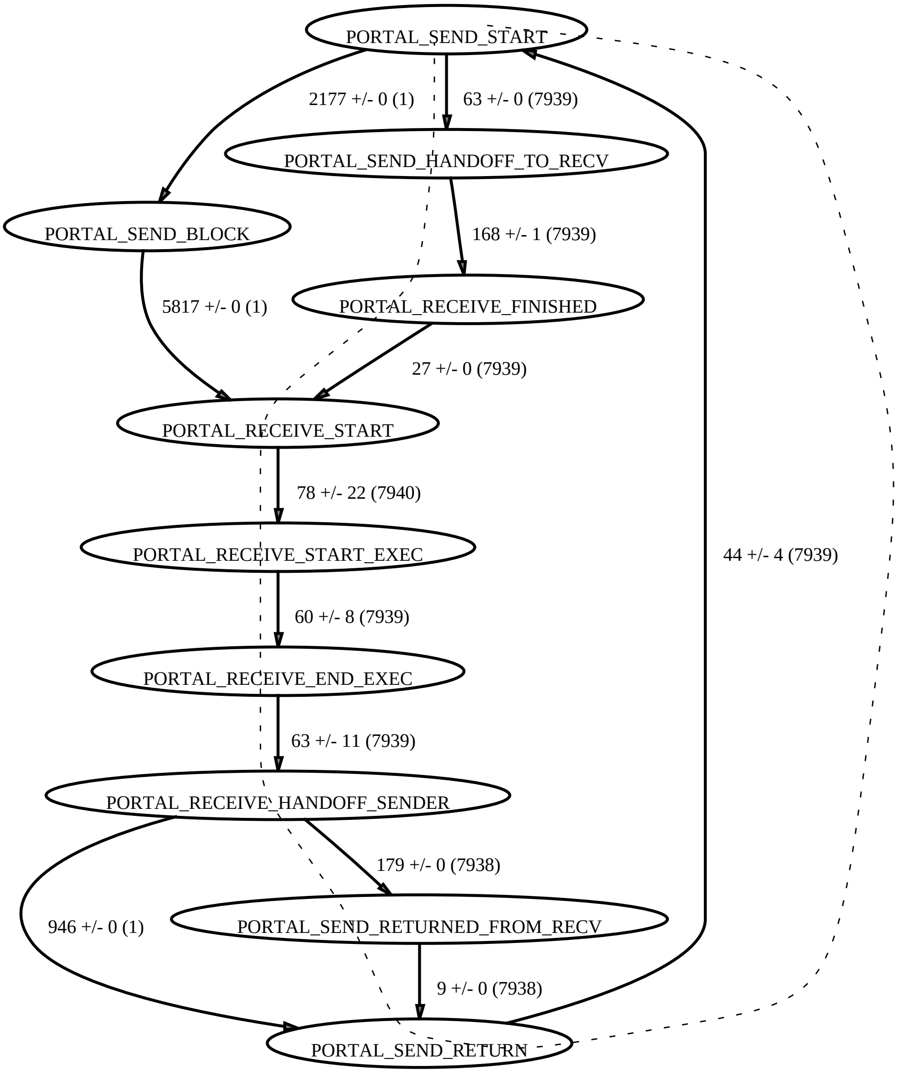


Figure 5.15: Event sequences during a loop of portal invocations.

Times are given in cycles with standard deviation All times are corrected by 87 cycles. The number in parenthesis is the number of transitions between two events. The dotted line marks the common path of a portal invocation (maximum number of transitions). For an description of the events see Figure 5.13 and Figure 5.14.



a deep object graph, such as a long linked list, overflows the stack. In our experiments Java is able to serialize a list of 256 elements but throws a StackOverflowException with 512 elements. To prevent stack overflows we increased the stack size to 20 MB. We also increased the heap size to 140 MB by using the following command line parameters: “-ms140m -mx140m -oss20m -ss20m”.

Each test was repeated 1000 times. The results are presented in Figure 5.16. It can be seen that the COPY0 algorithm has an exponential complexity. This is caused by the cycle detection part that scans a list of already copied objects to check whether an object has already been copied (see Figure 5.16). But as can be seen in Figure 5.17 the exponential complexity is negligible when only a small number of objects is transferred. For up to 100 objects both algorithms perform nearly identical. The COPY1 algorithm has a linear complexity and scales well to 10,000 and more parameter objects. The RMI serialization graph is not shown in Figure 5.17 because it is orders of magnitude slower than our two copying algorithms and it does not fit in the figure.

8 Summary

This chapter explained the portal abstraction in detail. Portals are used for inter-domain communication. They allow communication between domains but weakens the domain isolation only at dedicated points: the call and the return from the call. From a programmers perspective a portal invocation is a method invocation at a capability object. The most important properties of the portal mechanism are:

- Parameters are copied to the service domain as well as return values and exceptions are copied to the client domain on return from the call.
- Services are created automatically when a service object is passed to another domain.
- A service can either use an existing thread pool or create a new one.
- The thread pool can consist of one or more threads.
- If a service thread blocks a new thread can be created and added to the pool to avoid deadlock.

The performance of a portal invocation is in the range of IPC performance of optimized microkernels, although the portal invocation code is not written in hand-optimized assembler. Parameter copying is orders of magnitude faster than parameter copying using RMI.

Figure 5.16: Portal parameter copying cost

"COPY1 List" and "COPY1 Array" have nearly identical performance and therefore they collapse to a single graph. The COPY0 algorithm can only copy 4000 objects until the stack overflows.

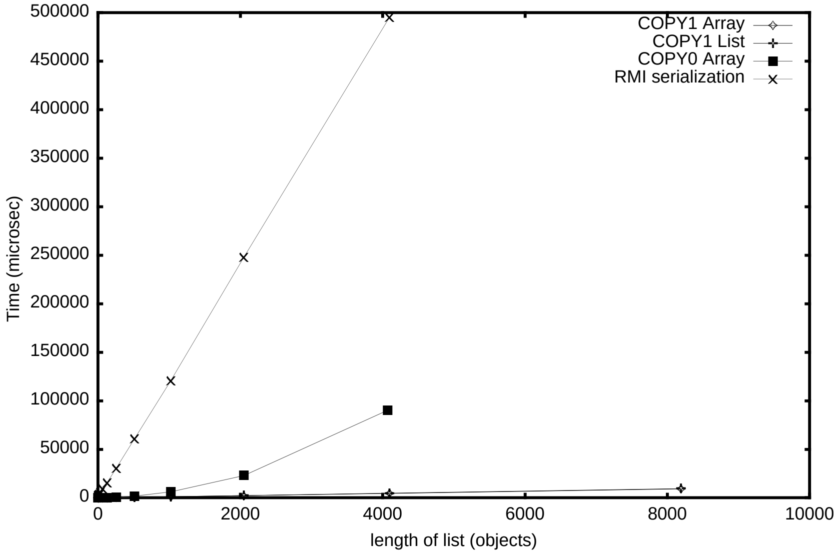
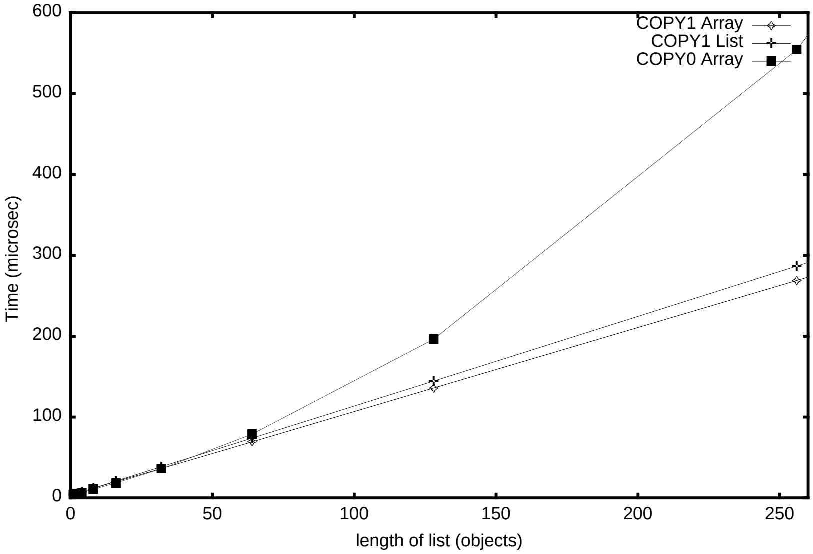


Figure 5.17: Small number of object parameters

RMI is not shown because it is orders of magnitudes slower than the JX parameter copying mechanism. (An RMI call with one parameter requires about 2000 μs)



CHAPTER 6 *Components*

All code in JX is organized in components [172]. The goals of the JX component system is to ease the maintenance and extension of the system. Modularizing the code that runs inside a domain into components eases configuration and administration of domains. Classes and methods are appropriate abstractions for a software developer to reason about a system. For everyone else, including system administrators, these abstractions are too fine-grain. Therefore we group related classes into components and reason about relationships between components. When a system evolves over time, not the single class is changed, but a whole component. To support a gradual system evolution new and old versions of a component can coexist in the system at the same time, but in one domain only one version of a component is allowed.

This chapter first describes the different types of components, followed by a discussion of component sharing and component relationships.

1 Component types

All JX components have the same structure: they consists of a set of classes and a description file that contains information about the component and the necessary environment to deploy the component. The environment specification includes the type of thread scheduler that is expected by the component code, for example a non-preemptive or preemptive scheduler.

Components can be classified into three types: library components, service components, and interface components.

Library component. A library component is a collection of classes and interfaces that are to be used by other components. They contain reusable code and data structures, such as lists and hash tables. The JDK component is an example for a library component.

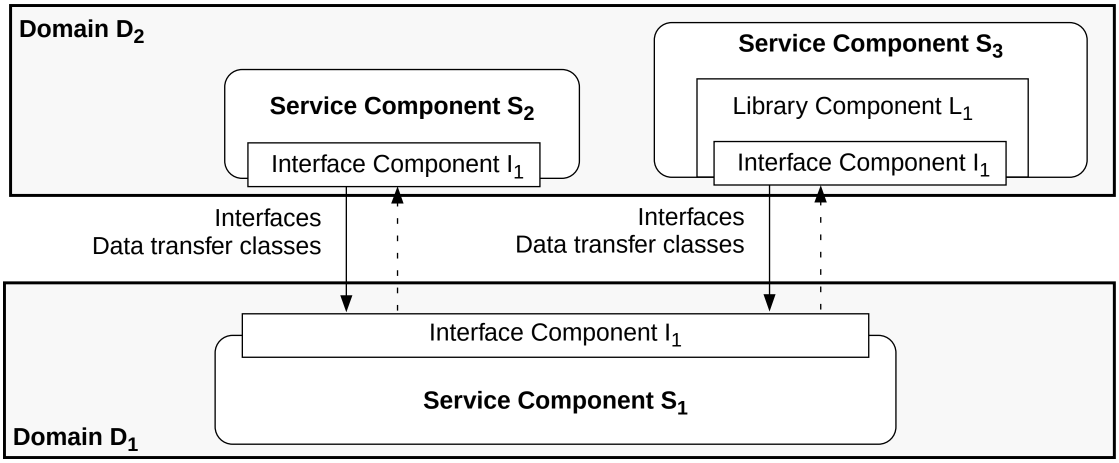
Service component. A service component contains the implementation of a specific service. Examples are device drivers, network protocols, file servers, database servers, or web servers.

Interface component. An interface component contains all classes and interfaces necessary to use a service component. An interface component usually contains the portal interface and classes or interfaces of the parameters. Additional classes, that make it easier to access the service but are not essential should be placed in a library component. Such a separation allows a domain to only import a minimal amount of code to access the service. If the interface component contains only interfaces and classes without methods no code must be imported at all. This allows to access an untrusted service without the need to use untrusted code.

To visualize a system configuration we use the notation described in Figure 6.1.

Figure 6.1: Notation for system configurations

In this example service component S_2 uses service component S_1 via interface component I_1. The arrorws are annotated with the names of the interfaces that are used to access the service and the classes that are used to transfer data. The solid line is used to show the direction of the communication. The dotted line shows the direction of return value transfers and makes the bidirectional nature of the communication explicit. Service component S_3 uses library component L_1 to access service component S_1. A library component is usually used to provide a more convenient interface to the service or to hide the capability-style service access behind an interface that conforms to an external standard (for example the JDK).

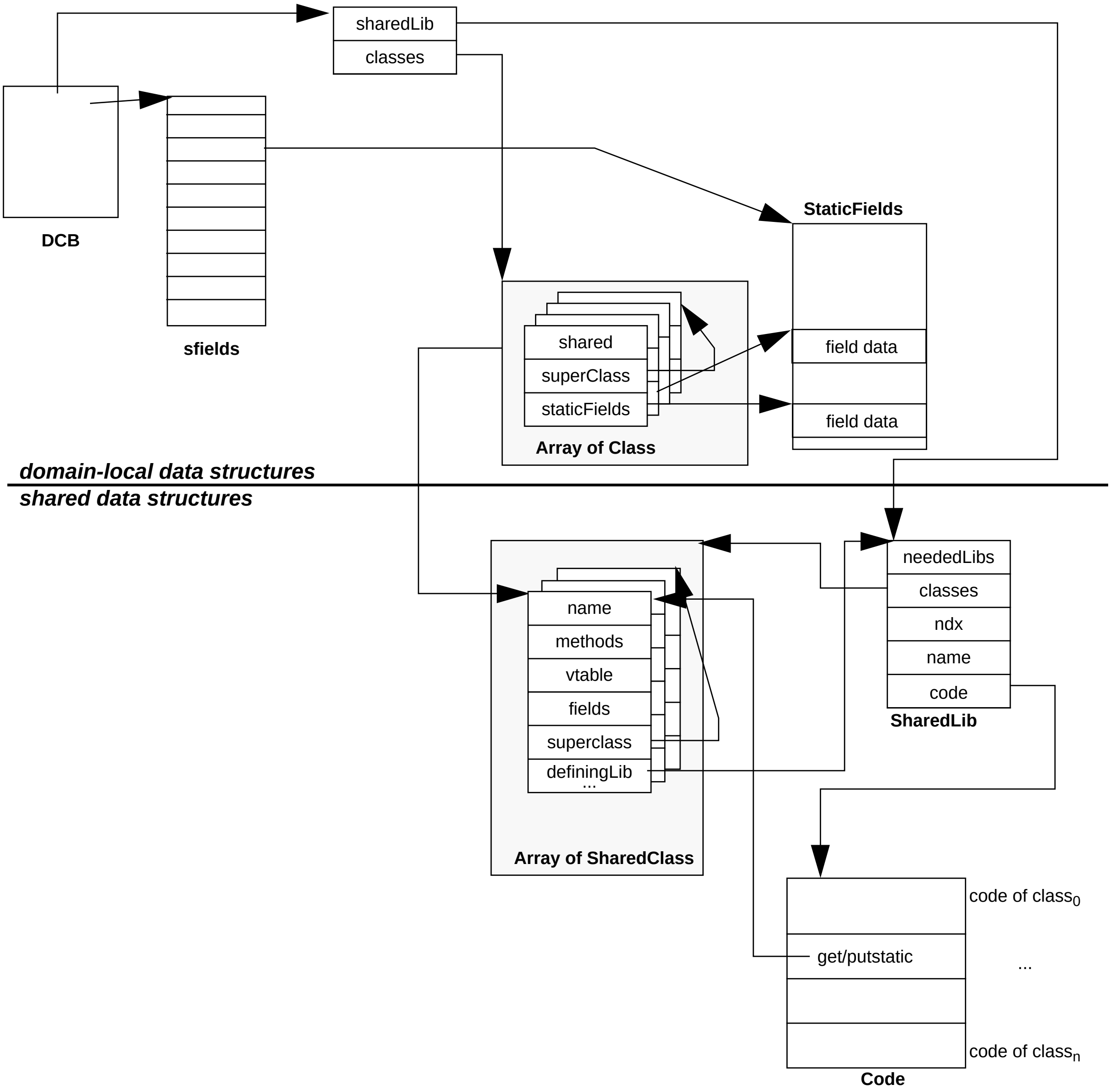


2 Shared components

Code sharing between protection domains has been used since the early timesharing systems [178]. Shared libraries have been introduced in UNIX SVR3 [9] and SunOS [72]. A shared library consists of code, constant data, and mutable data. Similar to shared libraries JX components can be shared between domains. A component consists of classes. Each class has a code part (the bytecode or the native code that was translated from bytecode), a constant data part (string objects and numeric constants which are part of the constant pool of a class), and a mutable data part (static fields of a class). Code and constant data can be shared between domains, while mutable data must be instantiated per domain. Figure 6.2 shows the data structures that are used to manage components and share them between domains. It is not possible to have a pointer from a shared data structure to a domain-local data structure. To find the domain-local data (static fields) starting from shared code it is required that the components are enumerated in a domain. Every shared component has a unique id (ndx field in the SharedLib structure). Every domain contains a table that maps this ID to the domain-local part of the component, the static fields. To access a static field, the shared code first looks up the domain that the currently running thread belongs to. In the domain control block it uses the component ID to find the static fields pointer in the sfields table.

Figure 6.2: Data structures for the management of components

Components are split in a domain-local part and a shared part. The shared part contains the majority of the data, such as method code and meta info. The domain-local part contains a data structure to find the static fields of the component's classes. Other domain-local data can be added if necessary.



Component sharing between domains is mandatory when the domains interact using portals. The domain that provides the service does also provide the interface that must be used to access the service. Type safety requires that either the method signature in the callee and caller domains are identical or the parameters are mapped to the other signature appropriately.

3 Code reuse between collocated/dislocated configuration

An overall objective of object orientation and object-oriented operating systems is code reuse. JX has all the reusability benefits that come with object orientation. But there is an additional problem in an operating system: the protection boundary. To call a module across a protection boundary in most operating system is different from calling a module inside the own protection domain. Because this difference is a big hindrance on the way to reusability, this problem has already been investigated in the microkernel context [57].

Our goal was a reuse of components in different configurations without code modifications. Although the portal mechanism was designed with this goal the programmer must keep several points in mind when using a portal. Depending on whether the called service is located inside the domain or in another domain there are a few differences in behavior. Inside a domain normal objects are passed by reference. When a domain border is crossed, parameters are passed by copy. To write code that works in both settings the programmer must not rely on either of these semantics. For example, a programmer relies on the reference semantics when modifying the parameter object to return information to the caller; and the programmer relies on the copy semantics when modifying the parameter object assuming this modification does not affect the caller.

In practice, these problems can be relieved to a certain extent by the automatic promotion of portal-capable objects to services. By declaring all objects that are entry points into a component as service object[1] a reference semantics is guaranteed for these objects.

4 Component relationships

Figure 6.3 shows the relationships of components in an evolving system. Components *depend-on* other components. In the example of Figure 6.3 component F depends on components B and C. Components also evolve over time and new versions of the component appear that may be used instead of the old version. The new components *extend* the old components. Component B in the example exists in versions B_1, B_2, and B_3. Components depend on specific versions of other components, for example, component B_3 depends on A_2, while B_2 and B_1 depend on A_1.

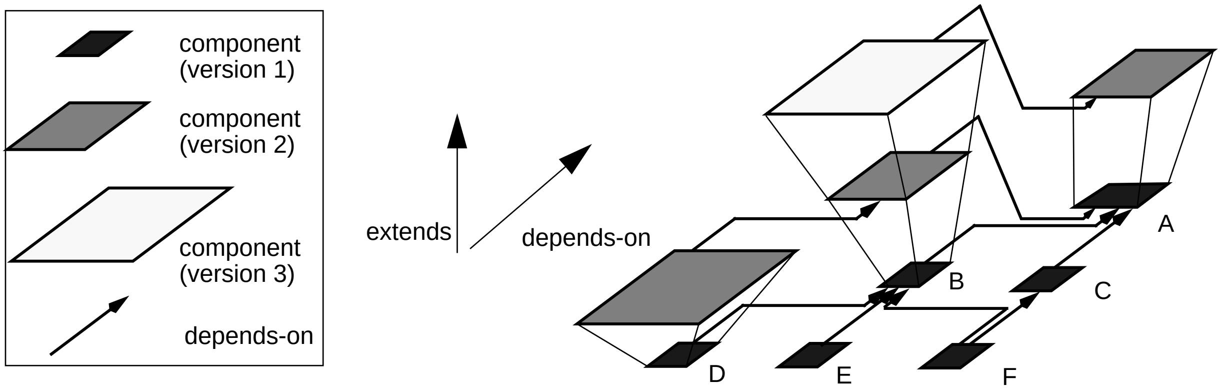
4.1 Dependencies

Components may depend on other components. We say that component B has an *implementation dependence* on component A, if the method implementations of B use classes or interfaces

[1] As explained in Chapter 5 a service object is an instance of a class that implements a portal interface.

Figure 6.3: Component compatibility and dependability

Components can depend on other components (depends-on relation) and can be an extension of an existing component (extends relation). Both relations are non-circular.



from A. Component B has an *interface dependence* on component A if the method signatures of B use classes or interfaces from A or if a class/interface of B is a subclass/subinterface of a class/interface of A, or if a class of B implements an interface from A, or if a non-private field of a class of B has as its type a class/interface from A.

Component dependencies must be non-cyclic. This requirement makes it more difficult to split existing applications into components (Although they can be used as one component!). A cyclic dependency between components usually is a sign of bad design and should be removed anyway. When a cyclic dependency is present, it must be broken by changing the implementation of one component to use an interface from an unrelated component while the other class implements this interface. The components then both depend on the unrelated component but not on each other. The dependency check is performed by the verifier and translator.

We used Sun's JRE 1.3.1_02 for Linux to obtain the transitive closure of the depends-on relation starting with java.lang.Object. The closure of the implementation dependency consists of 625 classes; the closure of the interface dependency consists of 25 classes. This means that each component that uses the Object class (i.e., every component) depends on at least 25 classes from the JDK. We think, that even 25 classes are a too broad foundation for OS components and define a compatibility relation that allows to exchange the components.The JDK is implemented as a normal component. Different implementations and versions can be used. Some classes of the JDK must access information that is only available in the runtime system. The class Class is an example. This information is obtained by using a portal to DomainZero. In other words, where a traditional JDK implementation would use a native method, JX uses a normal method that invokes a service of DomainZero via a portal. All of our current components use a JDK implementation that is a subset of a full JDK and, therefore, can also be used in a domain that loads a full JDK.

4.2 Compatibility

The whole system is build out of components. It is necessary to be able to improve and extend one component without changing all components that depend on this component. Only a component B that is compatible to component A can be substituted for A. A component B is binary compatible to a component A, if

- for each class/interface C_A of A there is a corresponding class/interface C_B in component B
- class/interface C_B is binary compatible to class C_A according to the definition given in the “Java Language Specification” (see [77] Chapter 13).

When a binary compatible component is also a semantic superset of the original component, it can be substituted for the original component without affecting the functionality of the system.

Szyperski [173] discusses the problems of independently extensible component-based systems. Hoelzle [95][1] and [84] use a concept called type adaptation to allow the integration of independently developed components into single application. While this is not the problem we are facing the type adaptation idea is used in the following section to allow communication between domains that use different components.

5 Portal communication and components

When two domains communicate using portals they exchange typed data. This is no problem as long as the type spaces of the two domains are completely identical, i.e. as long as they use exactly the same components. For many reasons such a configuration is neither practical nor possible. It is not possible because a client domain is not allowed to load a service implementation of a portal it uses as client (see Chapter 5, Section 5.1). It is not practical because different applications use different components and they may even use different versions of the same component. A communication between two domains should be possible if they use compatible component versions (see Section 4.2). Without this it is not possible to use an untrusted implementation of the JDK class library. The microkernel, which is trusted, uses JDK classes, such as String and OutOfMemoryException. The standard compliant implementation of these classes depends on many other classes (see Section 4.1). Therefore we need an implementation of these classes that is minimal but not standard compliant (jdk_0). This version is used in DomainZero. Other domains that require a standard-compliant JDK use a JDK component (jdk_1) that is compatible to jdk_0. Although these domains usually trust jdk_1 the microkernel does not. Figure 6.4 illustrates the relationships of the components and domains.

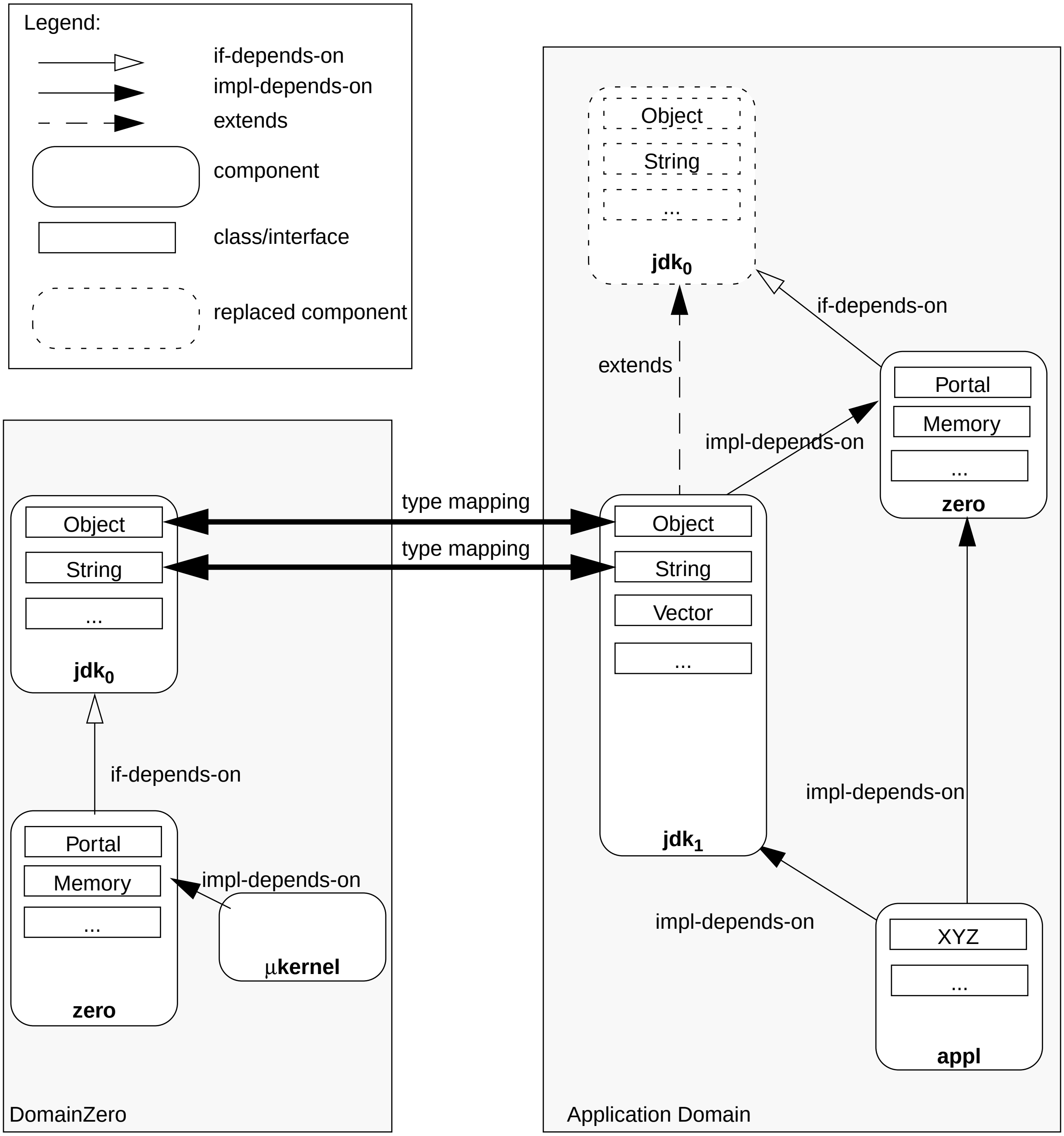
When an object is passed between the two domains and the class of the object is in different but compatible components the object’s state must be copied according to a mapping table. This mapping table describes which field in one class corresponds to which field in the other class. Mapping must be possible in both directions because objects can be passed either as parameters or as return values.

[1] Keller & Hoelzle [102] developed this mechanism further and called it Binary Component Adaptation.

Figure 6.4: Component compatibility and portal communication

During portal communication objects are copied between domains. The object's state must be copied according to the type mapping that exists between two components.

In the Application domain jdk_0 is subsumed by jdk_1. Although jdk_1's implementation depends on zero, it's method tables can be computed before zero's method tables are available, because it is not interface-dependent on zero.



To translate components to machine code and compute the layout of objects and method tables a component path is needed through the three-dimensional tree that is spawned by the *depends-on* and *extends* relations. This path must fulfill the following requirements:

- There are never two versions of the same component in one domain.
- If a component is present in a domain and this component depends on a specific version of another component, then the other component is present in the domain with at least that version.

To fulfill these requirements the following algorithm is used: The path is created by following the depends-on and extends relations until there is a component that depends on no other component. Because a component may directly depend on more than one component, more than one path may be created for the component (component F in Figure 6.5 depends on B and C). If a path includes a component that is also included in a previously created path the maximum of the component version previous path and the required component version is used. The first path of domain D_1 in Figure 6.5 is $F_1B_3A_2$ the second path is $F_1C_1A_2$. The depends-on relation between C and A only requires A_1 but the first path already contains A_2 which is therefore used.

As depicted in Figure 6.5 different domains can use different component path's. Therefore object layout and method tables are different even if the object is an instance of a class of a shared component. When an object must be converted from one domain to another one during a portal call, the different object layout is not only caused by the different versions of the class's components but also by the different path's. The following example which uses the component graph of Figure 6.5 may illustrate this. The two domains D_1 and D_2 share component C which may contain a class C with fields x and y. This class C may be a subclass of class A in component A. In the first version of component A (A_1) class A may contain a field k. In the second version (A_2) class A may have two additional fields l and m. In domain D_1 an instance of class C has the fields (k, x, y) while in domain D_2 such an instance has the fields (k, l, m, x, y). Even if the bytecode component C is shared between domains D_1 and D_2, the linked native code component can not be shared because of the different object layout.

JDK compatibility. To run existent Java applications on JX the system must provide an API that this application expects. This is usually jdk1.1, J2ME, J2SE, J2EE. The needed API is provided as a library component. For most of the operating system components even the J2ME provides too much unneeded functionality. Besides the zero interface library they only need some additional classes, like String, Integer, and System. Furthermore, only a subset of the methods of these classes is needed.

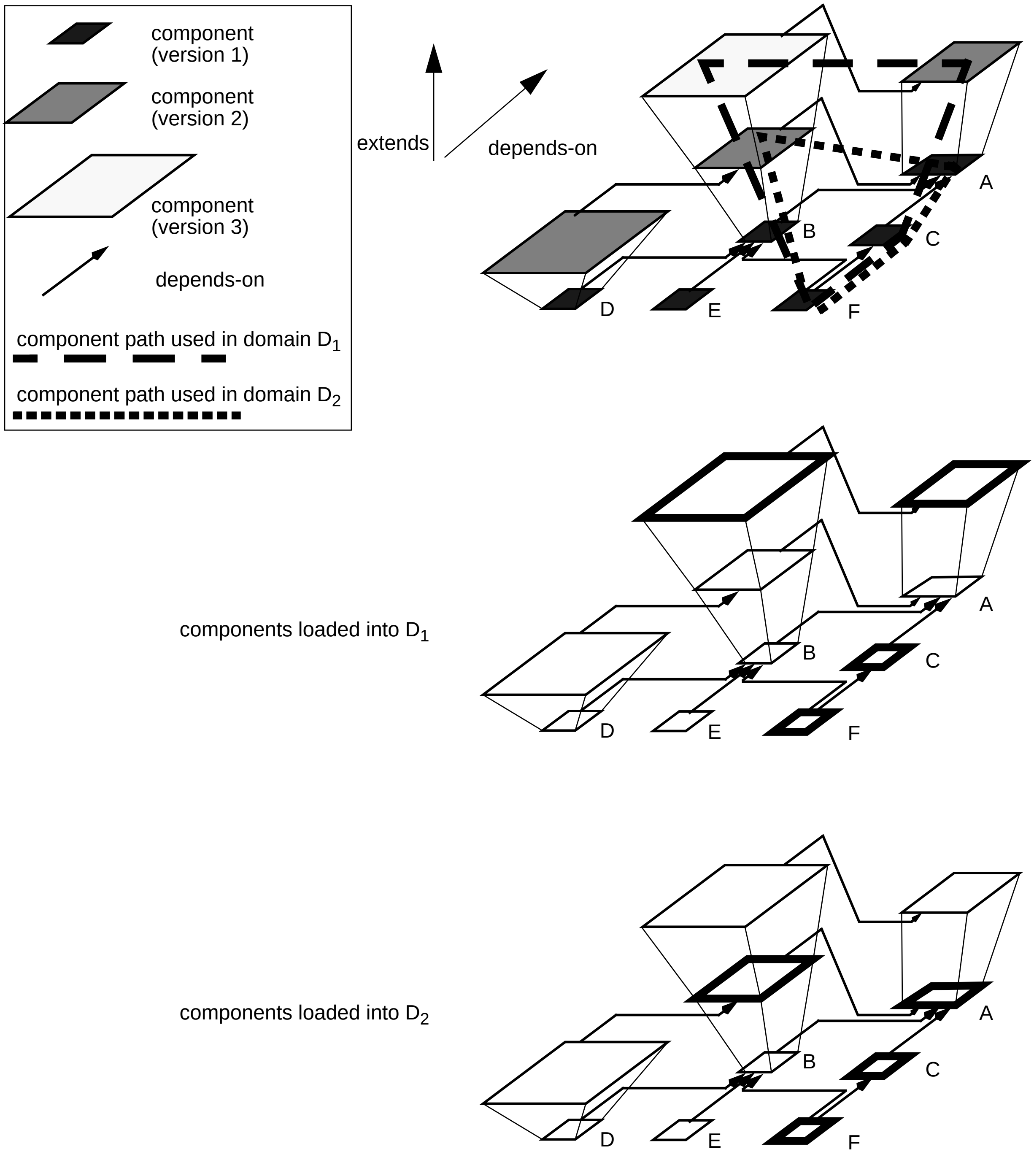
On the other hand, OS components should be collocatable with arbitrary application code in the same domain, to enable several performance optimizations, like inlining. This creates the problem of running code inside a domain that was developed using two different APIs. To solve this problem, we defined a compatibility relation over components.

6 Translation to machine code

All bytecode is compiled to machine code by the translator. The translator usually runs in an own domain and is highly processor specific.

Figure 6.5: Path through components

When computing the object layout and the vtable layout components must be arranged in a total order. In the figure this order is visualized as a path along the component relations. Because the path can be different in different domains, marshalling is required when copying objects between domains.



component (version 1)
component (version 2)
component (version 3)
depends-on
component path used in domain D_1
component path used in domain D_2
extends
depends-on
A
B
C
D
E
F
components loaded into D_1
A
B
C
D
E
F
components loaded into D_2
A
B
C
D
E
F

6.1 Method table creation and interfaces

JX extensively uses interfaces to achieve abstraction and decouple operating system components. Unfortunately, interface invocations have been the source of performance problems since the introduction of Java. Non-cyclic dependencies and the compilation of whole components opens up a way to compile very efficient interface invocations. Usually, interface invocations are a problem because it is not possible to use a fixed index into a method table to find the interface method. When different classes implement the interface, the method can be at different positions in their method tables. Components are sorted according to their dependency relationship. Consistent ordering of non-dependent components uses lexical order by sorting components according to their names. Within a component classes and interfaces are sorted by their subtype relationships. All interfaces come first and a supertype comes before any of its subtypes. There is a global interface dispatch table that contains all interface methods and holes for non-interface virtual methods. Whenever an index is used for a non-interface method it can no longer be used for an interface method, because a subclass may implement the interface and the interface method and non-interface method would require the same index. The technique makes no closed-world assumptions: components can be added without requiring a recompilation of existing components.

Our method table creation scheme makes interface invocations as fast as method invocations at the cost of (considerably) larger method tables. The size of the x86 machine code in the complete JX system is 1,010,752 bytes, which was translated from 230,421 bytes of bytecode. The method tables consume 630,388 bytes. These numbers show that it would be worthwhile to use a compression technique for the method tables or a completely different interface invocation mechanism. One should keep in mind, that a technique as described in [4] has an average-case performance near to a virtual invocation, but it may be difficult to analyze the worst-case behavior of the resulting system because of the use of a caching data structure.

6.2 Optimizations

Inlining. An object-oriented programming style leads to many small methods. Most of these methods will not be used polymorphically and can be inlined. The JX translator can inline private, final, and static methods without difficulties.

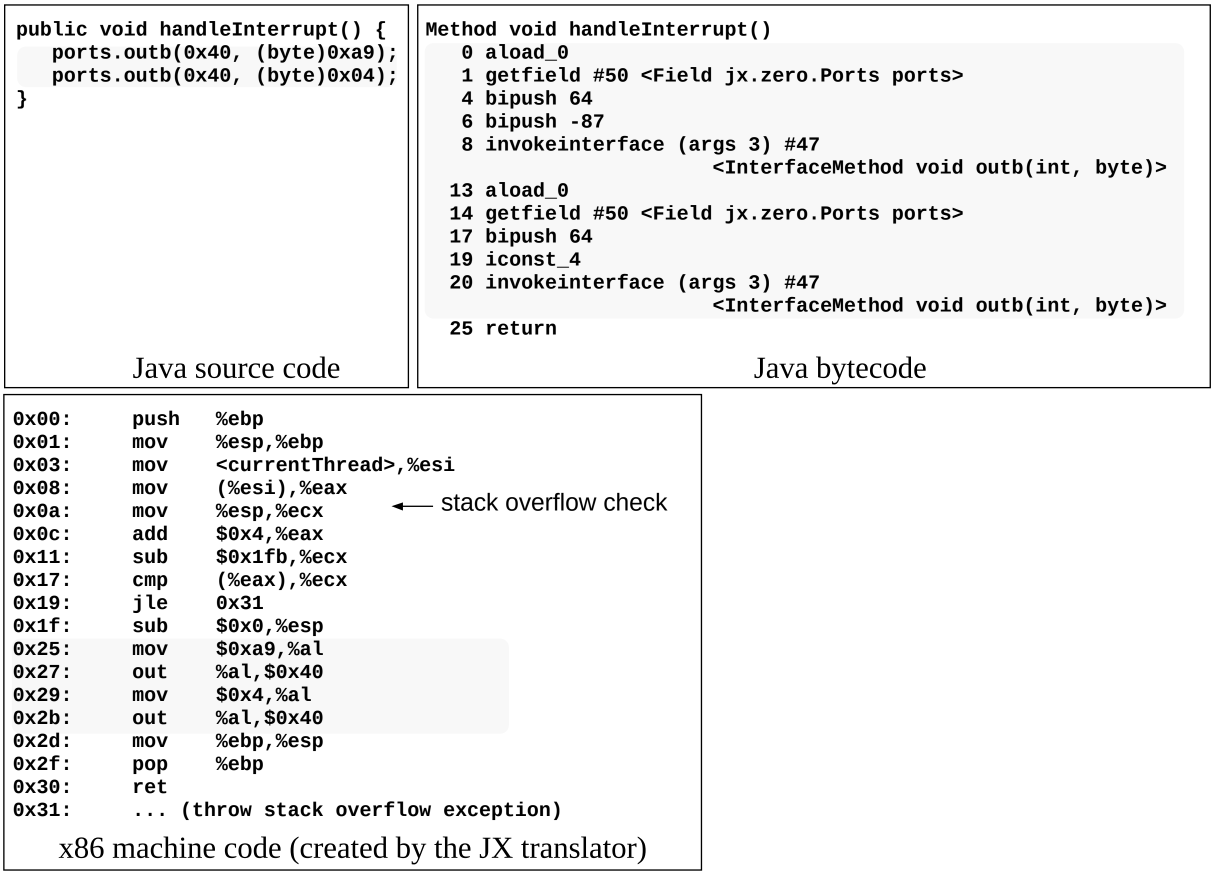
Inlining of virtual methods is only possible when the method is not overwritten. To detect this the translator needs a global view of the system. In JX it only needs a domain-local view, because components are loaded per domain. Even if another domain loads a component with a class that overwrites the method, the method can be inlined, because the overwriting method is part of the other domain's component and will never be executed in the current domain

Inlining virtual methods is only possible if either the domain is not allowed to dynamically load components or components are recompiled and inlining is removed if a new component is loaded that contains a subclass that overrides the inlined method.

Inlining fast portals. It is essential to generate optimized code for several fast portal calls. Figure 6.6 shows an example of this technique. In this example a two-line Java program is compiled to a bytecode sequence of 10 instructions (not counting the return statement), which is compiled to a machine code sequence of four instructions. This example demonstrates that inlining of fast portals can reduce the size of the generated machine code.

Figure 6.6: Inlining of fast portal methods

This figure shows the implementation of the interrupt handler of a driver for the PC's programmable interval timer (PIT). The interrupt handler needs two outb instructions to set the counter of the timer to a new value. This code sequence requires two lines Java code, is compiled to 25 bytes bytecode (without constant pool), and 8 bytes of machine code.



```
public void handleInterrupt() {
   ports.outb(0x40, (byte)0xa9);
   ports.outb(0x40, (byte)0x04);
}
```

Java source code

```
Method void handleInterrupt()
   0 aload_0
   1 getfield #50 <Field jx.zero.Ports ports>
   4 bipush 64
   6 bipush -87
   8 invokeinterface (args 3) #47
                     <InterfaceMethod void outb(int, byte)>
  13 aload_0
  14 getfield #50 <Field jx.zero.Ports ports>
  17 bipush 64
  19 iconst_4
  20 invokeinterface (args 3) #47
                     <InterfaceMethod void outb(int, byte)>
  25 return
```

Java bytecode

```
0x00:     push   %ebp
0x01:     mov    %esp,%ebp
0x03:     mov    <currentThread>,%esi
0x08:     mov    (%esi),%eax
0x0a:     mov    %esp,%ecx
0x0c:     add    $0x4,%eax
0x11:     sub    $0x1fb,%ecx
0x17:     cmp    (%eax),%ecx
0x19:     jle    0x31
0x1f:     sub    $0x0,%esp
0x25:     mov    $0xa9,%al
0x27:     out    %al,$0x40
0x29:     mov    $0x4,%al
0x2b:     out    %al,$0x40
0x2d:     mov    %ebp,%esp
0x2f:     pop    %ebp
0x30:     ret
0x31:     ... (throw stack overflow exception)
```

stack overflow check

x86 machine code (created by the JX translator)

Interface elimination. During profiling of the file system we realized that several small methods of the buffer cache are called very often by the file system. Although these methods were declared as final and therefore are inlinable, the translator did not inline them, because the file system uses the buffer cache via an interface. Using an interface allows for an easy replacement of the buffer cache implementation. In cases where there is only one class that implements the interface loaded into the domain, the translator is now able to replace the interface invocation (invokeinterface instruction) by a non-virtual method invocation (invokespecial instruction), which then can be inlined.

Inlining portals. Placing components into one domain would even allow inlining of portal calls. In a single domain NFS server, for example, this would allow to inline the code path that starts at the network driver, goes through the protocol stack, to the NFS server, through the file system and ends at the buffer cache or disk driver. This would enable enormous compiler optimizations. By using the interface elimination technique we can transform the interface invocation into a method invocation. But there is an additional problem. Even if there is only one class in the domain that implements the interface, the portal invocation may be an invocation at a real portal. To solve this problem, a domain must be forbidden to obtain portals of the inlined type. Only then, inlining of (local) portal calls would be possible.

Compact code. The translator for the Pentium processor tries to generate compact code by appending exception handling code at the end of the method. While this does not reduce the total code size it leads to better utilization of the instruction cache and the instruction pipeline. The underlying assumption is that exception handling code is rarely executed. By moving it to the end of the method it is not unnecessarily loaded into the instruction cache. The code that checks for the exception condition uses a conditional forward branch to the exception handler. The Pentium branch predictor assumes that a forward branch is not taken and loads the pipeline with instructions from the normal method.

6.3 Java-to-native control transfer

The JX system consists of a large and growing number of Java classes and a small and fixed core of code that is compiled from C and assembler, called native code. Most of the time the system executes Java code, but there are certain points where Java code invokes native code. At the following points the control flows from Java code to C code:

- The Java code contains a complex operation, for example a checkcast instruction, that must call a support function in the runtime system. These support functions are all contained in a function table *vmsupport*. This function table is used by the linker to resolve a symbol to a function address.
- The Java code contains a portal invocation. As far as the translator is concerned, a portal invocation is an interface invocation. No special code is generated. The method table of the portal object contains pointers to a short machine code sequence that was generated at portal creation time and that performs an invocation of the send() function in the core. The send() function blocks the current thread and activates the service thread (see Chapter 5).
- The Java code contains a fast portal invocation of a DomainZero service. This again is either compiled as an interface invocation or special code is generated. The method table contains pointers to the C functions. The C function that implements the service is executed in the same thread.

At the following points control flows from C to Java code:

- A service is invoked by receive().
- A new string is created by the newString() function and the String constructor is invoked.

There are no user-defined native methods. Native methods implicitly separate the system into two domains - a Java domain and a native domain. This roughly corresponds to the separation into a user domain and a system domain in operating systems. This two-level architecture conflicts with the multi-domain Java operating system we envisioned. Furthermore, it complicates the implementation of the garbage collector, that has to look for references held by the native code. A native method threatens the stability of the whole system.

7 Summary

This chapter described the mechanisms that are used to manage code. All code is organized in components. The problems of component sharing, component dependability, and component evolution have been discussed. The most important points to remember are:

- Components can transparently be shared between domains.
- Domains can use different versions of the same component or completely different components that implement the same functionality.
- When components are not shared between domains portal communication becomes more expensive because objects must be marshalled.
- The domain-specific translation of components to native code allows a number of optimizations that are not possible if the code is shared.
- Although there are situations where components must directly interact with the C code of the kernel without using portals, the transition points between a component and the microkernel can be exactly enumerated.

CHAPTER 7 *Processor Scheduling*

There are two important primary resources in a computational system: memory and processor. Without one of these resources no computation is possible. This chapter describes the processor scheduling framework of JX and discusses issues that are related to processor scheduling.

The system can be configured to use several scheduler configurations (see Figure 7.1):

(a) There is a global scheduler built into the kernel that schedules all threads.

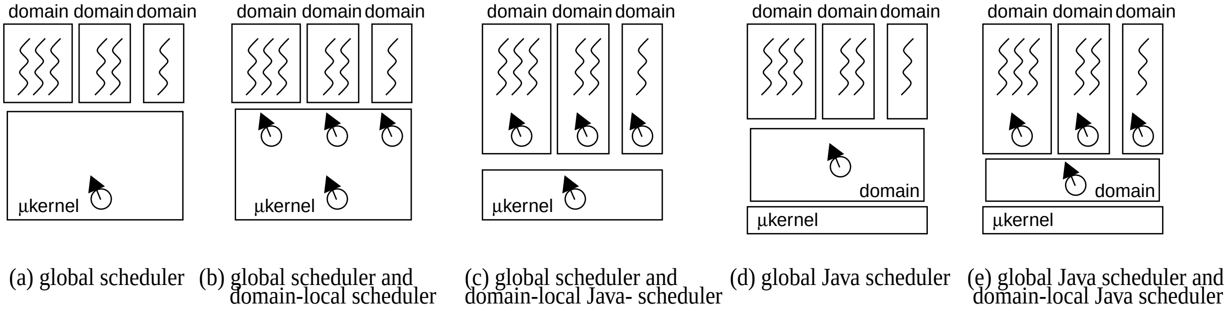
(b) The global scheduler schedules domains instead of threads. The global scheduler decides what domain to run next, schedules the domain-local scheduler, and the domain-local scheduler decides what thread to run next.

(c) The domain-local schedulers can be written in Java and run in a normal domain outside DomainZero. This allows to use user supplied schedulers that can be loaded at runtime.

(d) Even the global scheduler can run outside the microkernel. All scheduling decisions lead to the invocation of a scheduler implementation which is written in Java. In this configuration there is one (Java) scheduler that schedules all threads of all domains.

(e) The global Java scheduler does not schedule threads, as in the previous configuration, but domains. This is similar to configuration (c) with the global scheduler outside the kernel.

Figure 7.1: Scheduler configurations


domain domain domain
µkernel
domain
(a) global scheduler
(b) global scheduler and domain-local scheduler
(c) global scheduler and domain-local Java- scheduler
(d) global Java scheduler
(e) global Java scheduler and domain-local Java scheduler

1 Microkernel scheduling support

The microkernel contains low-level support functions to save and restore the CPU state and to detect a multiprocessor hardware (called “low-level CPU management” in Figure 3.2 on page 19). There is a defined interface between the scheduler and other parts of the microkernel. This interface includes a definition of all states of a thread (see Figure 7.2). After a thread is initialized it is put in the runnable state. It can leave this state by explicitly calling a block() method at a DomainZero portal. It also leaves this state during a portal communication (see Figure 5.11 on page 49). There is a special kind of threads, called one-shot threads, that run until completion without being interrupted. Their thread control block and stack is reused between runs. Currently the garbage collector and interrupt threads are realized as one-shot threads.

Figure 7.2: Thread states

This figure shows the states of a thread. The rectangular box contains the states that are specific to the portal invocation mechanism and not shown in this figure.

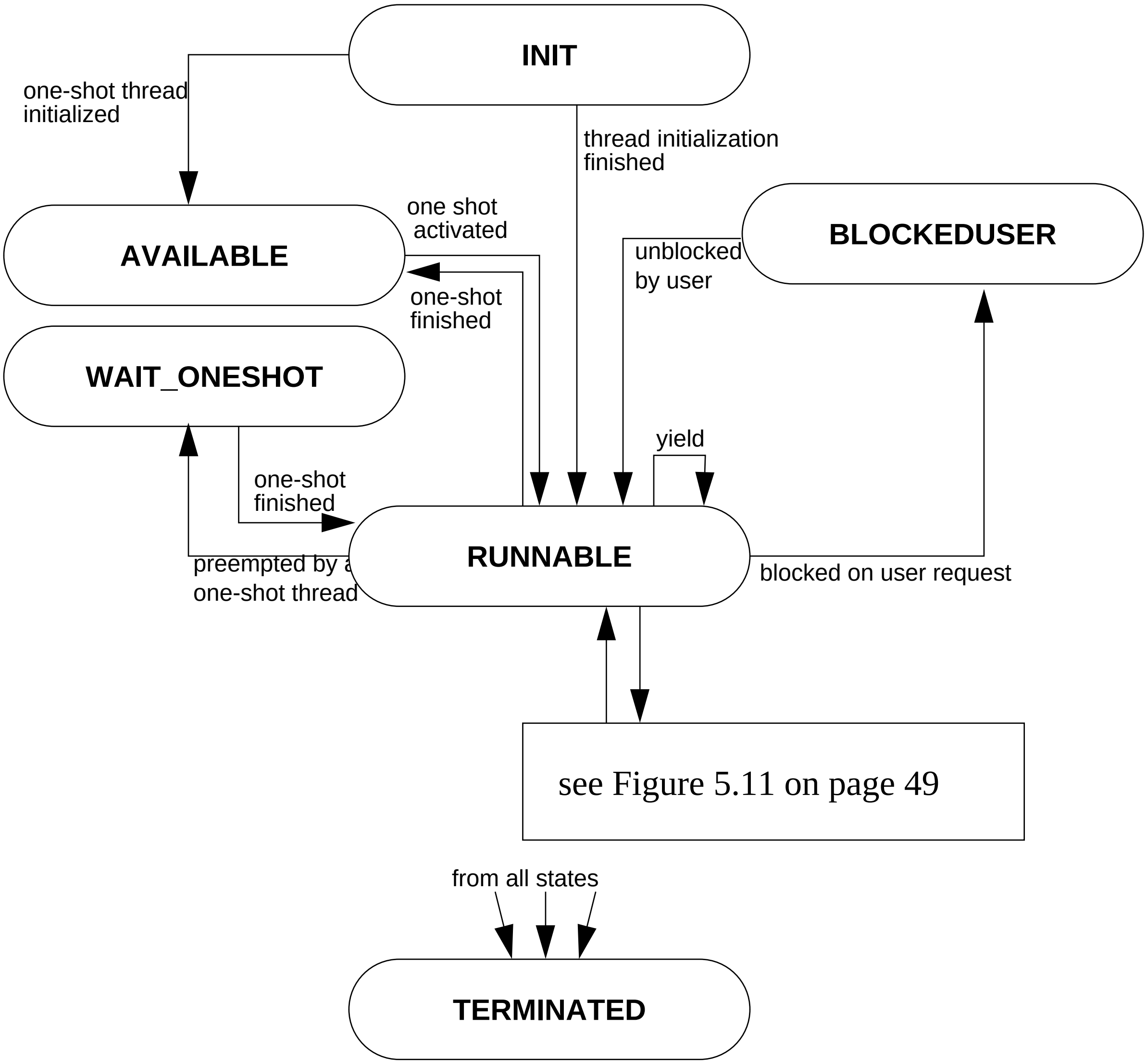


Figure 7.3: Interface between the VM and the global scheduler

void Sched_preempted();

The time slice of the currently running thread is over. The global scheduler must select and switch to the next thread. This function is called in the timer interrupt context with interrupts disabled.

void Sched_created(ThreadDesc * thread, int param);

A new thread has been created. The parameter param tells the scheduler whether the thread should be scheduled or not. The scheduler is expected to schedule this thread according to the scheduling strategy.

void Sched_destroyed(ThreadDesc * thread);

A thread was destroyed and must be removed from the scheduling data structures. The passed thread control block is valid until the function returns.

void Sched_destroyed_current(DomainDesc * currentDomain);

The currently running thread was destroyed. The current TCB is not available. A new thread must be scheduled.

void Sched_yield();

The currently running thread yields the processor. If no other thread is runnable this function returns and the current thread continues to run.

void Sched_unblock(ThreadDesc * thread);

The passed thread becomes runnable and should be scheduled according to the scheduling strategy.

void Sched_domainEnter(DomainDesc * domain);

The domain enters the scheduler. This function is called when a new domain is created.

void Sched_domainLeave(DomainDesc * domain);

The domain leaves the scheduler. This function is called when the domain terminates or is suspended. The DCB and related data structures, such as TCBs, are valid until this function returns.

void Sched_global_gc_rootSet(DomainDesc * domain, HandleReference_t handler);

The domain is garbage collected and the scheduler must invoke the handler function with all references it holds to objects that are located on the domain's heap.

void Sched_global_gc_tcb(DomainDesc * domain, ThreadDesc *thread, HandleReference_t hand);

The domain is garbage collected and the scheduler must invoke the handler function with all scheduler maintained references that are contained in the passed thread control block.

void Sched_block_portal_sender();

A thread invokes a portal method but the service domain is not ready to accept the service invocation. The usual cause is that there is no available receiver thread. The portal mechanism already appended the thread to the wait queue of the service. The scheduler should set this thread's state to PORTAL_WAIT_FOR_RCV and switch to the next runnable thread.

void Sched_portal_waitfor_sender();

The receiver thread is available and there are no pending requests. The scheduler should set this thread's state to PORTAL_WAIT_FOR_SND and switch to the next runnable thread.

void Sched_portal_handoff_to_receiver(ThreadDesc *receiver);

The service thread calls this function to handoff the timeslot to the thread receiver.

void Sched_portal_handoff_to_sender(ThreadDesc *sender, int isRunnable) ;

This function is called by a portal receiver that wishes to handoff the timeslot to the thread sender. The flag isRunnable states whether the current thread is runnable (RUNNABLE) or must wait for the next sender. (PORTAL_WAIT_FOR_SND).

2 Scheduler configurations

This section describes problems that are specific to the different scheduler configurations.

2.1 Global kernel scheduler

In this configuration all threads of all domains are scheduled by a single scheduler. Many of the functions of Figure 7.3 may have a very simple and short implementation. The most import examples are the Sched_portal_* functions. These functions can be inlined and therefore this configuration results in a kernel with the minimal possible scheduling related overhead.

2.2 Global kernel schedulers and domain-local kernel schedulers

The interface between the global scheduler and the local schedulers (Figure 7.4) is similar to the interface between the VM and the global scheduler (Figure 7.3) because most scheduling decisions are off-loaded to the local schedulers. The interface of a local scheduler has five groups of methods: information methods, activation methods, querying methods, garbage collection methods, and initialization methods.

Figure 7.4: Interface between global scheduler and domain-local scheduler

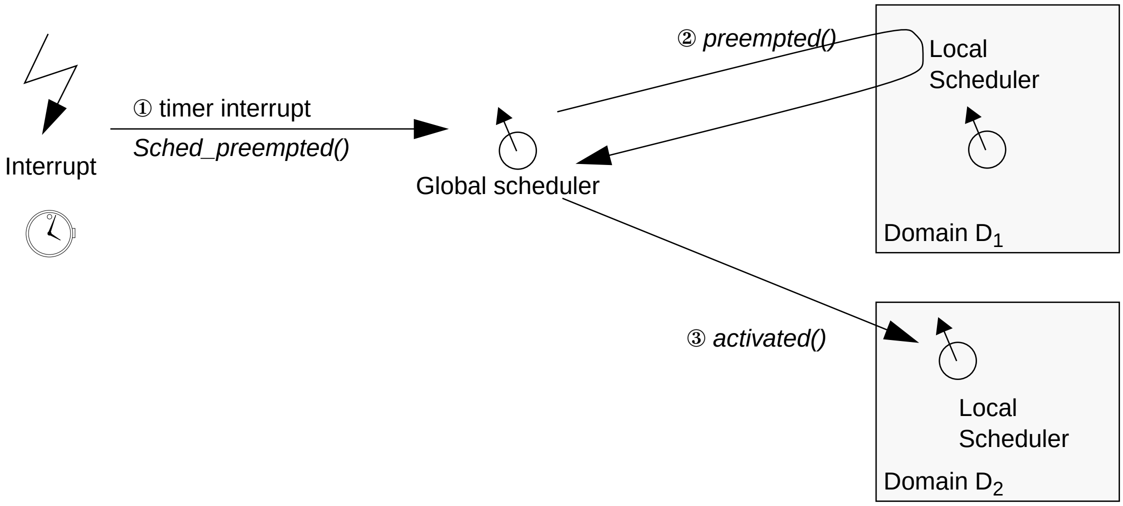
```
Information methods:
void preempted(DomainDesc * domain, ThreadDesc * thread);
void created(DomainDesc * domain, ThreadDesc * thread);
void destroyed(DomainDesc * domain, ThreadDesc * thread);
void blocked(DomainDesc * domain, ThreadDesc * thread);
void unblocked(DomainDesc * domain, ThreadDesc * thread);
void yielded(DomainDesc * domain, ThreadDesc * thread);
void blockedInPortal(DomainDesc * domain, ThreadDesc * thread);
Activation methods:
void activated(DomainDesc * domain);
void switchTo(DomainDesc * domain, struct ThreadDesc *thread);
void interrupted(DomainDesc * domain, ThreadDesc * thread);
Querying methods:
boolean isRunnable(DomainDesc * domain);
boolean portalCalled(DomainDesc * domain, ThreadDesc * thread);
Garbage collection methods:
void walkDomainSpecials(DomainDesc * domain, HandleReference_t hndl);
void walkThreadSpecials(DomainDesc * domain, ThreadDesc *thread, HandleReference_t hndl);
Initialization methods:
void init(DomainDesc *domain, CallbackFunction becomesRunnable);
void initTCB(DomainDesc * domain, ThreadDesc *thread);
```

Information methods are used to inform the scheduler about a state change related to its domain. The scheduler is informed whenever one of its threads is preempted or interrupted, created or destroyed, blocked or unblocked, or yielded the processor. A dedicated method informs the scheduler that a thread was blocked waiting for a portal invocation to complete.

Activation methods request the scheduler to activate one of its threads. The switchTo() method has an additional thread parameter that gives a hint to the scheduler that scheduling this specific thread may help another domain to progress. The local scheduler is free to ignore this hint. Especially schedulers that guarantee a strict round-robin scheduling must ignore this hint if the domain was preempted or interrupted. The switchTo() method is used to implement handoff during a portal call. Figure 7.5 shows the interaction between the interrupt subsystem, the global scheduler, and the local scheduler when a timer interrupt signals the end of the time slice. First the global scheduler is informed. If the global scheduler implements the two-level scheduling that is described in this section it informs the preempted domain about the preemption and selects and activates the next domain. If a local scheduler needs smaller time-slices than the global scheduler, the local scheduler must be interrupted without being preempted. For this purpose, the local scheduler has a method interrupted() which is called before the time-slice is fully consumed. This method operates similar to the method activated().

Figure 7.5: Example interaction between global scheduler and local scheduler

A timer interrupt leads to the invocation of the global scheduler's Sched_preempted function (see Figure 7.3). If the global scheduler uses domain-local schedulers (configurations (b,c,e) in Figure 7.1) it first invokes the preempted() method of the currently running domain, selects a new runnable domain and activates this domain by invoking the activate() method of its local scheduler.



Querying methods are used by the global scheduler to obtain information about the state of a specific domain. The isRunnable() method tests whether the domain can be activated and the portalCalled() method tells the local scheduler that the thread invoked a portal and asks whether the remaining time of the time slice should be donated to the called domain.

Garbage collection methods are called during a garbage collection. The local scheduler can store state in the domain control block (DomainDesc) and in the thread control block (ThreadDesc). The contents of this data can only be interpreted by the respective local scheduler and may contain object references. When the walkDomainSpecials() method is called the local scheduler must enumerate and update its roots by invoking the passed handler function with all object references that it stored in the domain control block. The walkThreadSpecials() method is used to enumerate and update the references that are stored in the thread control block of the passed thread.

Initialization methods are called when a new domain is created and the scheduler must initialize data structures in the domain control block and when a new thread is created and the scheduler must initialize its part of the thread control block. The DCB initialization method is passed a callback function that can be invoked to inform the global scheduler when the domain becomes runnable.

The information and activiation methods are similar to scheduler activations [174] in a UNIX-like system, which notify the user-level scheduler about events inside the kernel, such as I/O operations. There are very few scheduling related operations inside the JX kernel. Scheduling is affected when a portal method is invoked. First, the scheduler of the calling domain is informed, that a thread started a portal call. The scheduler can now delay the portal call, if there is any other runnable thread in this domain. But it can as well handoff the processor to the service domain. The scheduler of the service domain is notified of the incoming portal call and can either activate the service thread or let another thread of the domain run. Not being forced to schedule the service thread immediately is essential for the implementation of a non-preemptive domain-local scheduler.

2.3 Global kernel scheduler and domain-local Java schedulers

While the separation in global and local kernel schedulers improves the structure of the kernel moving the local scheduler up to the Java level allows to dynamically replace schedulers and allows to use user supplied schedulers. There are two important differences between local kernel schedulers and local Java schedulers: Java schedulers are supplied in type-safe bytecode and they are not trusted. This means, that the global scheduler can *not* assume, that an invocation of the local scheduler returns after a certain time. During the execution of the information methods interrupts are disabled. An upper bound for the execution time of these methods must be verified during the verification phase.

The difference to the interaction illustrated in Figure 7.5 are additional thread switches. The global scheduler invokes a scheduler thread of domain D_1 to execute the method preempted()[1]. The preempted() method is executed with interrupts disabled. This avoids race conditions and the need to program the preempted method to be reentrant. But it requires that an upper bound of its execution time is verified similar to interrupt handlers (see Chapter 9, Section 4). When the preempted() method returns, the system switches back to the thread of the global scheduler. The global scheduler then decides, which domain to run next (in the example D_2 is selected) and activates the domain-local scheduler using the method activated(). For each CPU that can be used by a domain the local scheduler of the domain has a CPU portal. This portal can be used to activate another thread by calling the method switchTo(). The switchTo() method can only be called by a thread that runs on the CPU which is represented by the CPU portal. The global scheduler does not need to wait for the method activated() to

[1] This thread is a one-shot thread (see Section 1).

finish. Thus, an upper time bound for method activated() is not necessary. The activated() method makes the scheduling decision and it can be arbitrarily complex.

3 Interaction between domains and the scheduler

The functional code of a domain should not depend on the scheduler configuration. This code interacts with the scheduler by using a fast portal that is provided by DomainZero: the ThreadManager portal. The ThreadManager is used to block or unblock threads, or to yield the processor. The CPUState portal is used to represent a thread[1]. These portal interfaces are described in Figure 7.6.

Figure 7.6: ThreadManager and CPUState interface

ThreadManager can be used to yield the processor, block the current thread, unblock another thread, get a reference to the current thread, and create a new thread. The ThreadEntry interface is equivalent to the JDK's Runnable interface.

To avoid races leading to lost wakeups between the invocation of block() and another thread's invocation of unblock(), the ThreadManager provides additional methods to block only if no other thread unblocked the current thread. Each invocation of unblock sets a flag unblocked in the thread control block. The blockIfNotUnblocked() method blocks only if this flag is not set. The CPUState portal is not only to represent threads when interacting with the ThreadManager. It can also be used to check whether a thread is a service thread. It furthermore has two functions to create a linked list of CPUState objects. While it isof course possible to manage CPUState objects using a normal container, such as Vector, these two methods are much faster. As CPUState is a fast portal the compiler can inline these two methods which results in a few machine instructions to manipulate the next field in the CPUState portal.

```
public interface ThreadManager extends Portal {
    void yield();
    void block();
    void blockIfNotUnblocked();
    void clearUnblockFlag();
    boolean unblock(CPUState state);
    CPUState getCPUState();
    CPUState createCPUState(ThreadEntry entry);
}
public interface CPUState extends Portal {
    boolean isServiceThread();
    CPUState getNext();
    CPUState setNext(CPUState next);
}
```

[1] To avoid confusion with the class java.lang.Thread that is part of the (untrusted) JDK we did not use the name Thread.

4 Concurrency control

To allow to control the concurrency between threads and domains in the presence of interrupts and preemptive scheduling the system must provide locking and synchronization primitives. There are three distinct areas were concurrency control is necessary: within the microkernel, within a domain, and between domains.

4.1 Kernel-level locking

There are very few data structures that must be protected by locks inside the microkernel. Some of them are accessed by only one domain and can be locked by a domain-specific lock. Others, for example, the domain control blocks, need a global lock. Because the access to this data is very short, an implementation that disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor is sufficient and is currently used.

4.2 Domain-level locking

Domains are responsible for synchronizing access to objects by their own threads. Because there are no objects shared between domains there is no need for inter-domain locking of objects. Java provides two facilities for thread synchronization: mutex locks and condition variables. When translating a component to native code, an access to such a construct is redirected to a user-supplied synchronization class. How this class is implemented can be decided by the user. It can provide no locking at all or it can implement mutexes and condition variables by communicating with the (domain-local) scheduler. Every object can be used as a monitor (mutex lock), but very few actually are. To avoid allocating a monitor data structure for every object, traditional JVMs either use a hashtable to go from the object reference to the monitor or use an additional pointer in the object header. The hashtable variant is slow and is rarely used in today’s JVMs. The additional pointer requires that the object layout must be changed and the object header be accessible to the locking system. Because the user can provide an own implementation, these two implementations, or a completely application-specific one, can be used.

Java allows to apply locking to a class. Although classes can be shared between domains they are represented by distinct class objects and the classes can be independently locked. This design avoids that one domain starves another domain by holding the lock of a shared class or that the lock is used as a covert communication channel.

Another very cheap way of guaranteeing safety in the presence of multiple threads on a uniprocessor is to avoid arbitrary thread switching.

4.3 Inter-domain locking

Memory objects (see Chapter 8, Section 3) allow sharing of data between domains. When two domains want to synchronize they can use a portal call. Several capability-based systems contain primitives to lock an object to perform a transaction that involves a series of method invocations. Such a mechanism introduces the possibility of deadlock and denial of service. A malicious domain could obtain such a lock but never release it. For these reasons JX does not provide blocking inter-domain locks.

Atomic code. The JX kernel contains a mechanism to avoid locking at all on a uniprocessor. The atomic code is placed in a dedicated memory area. When the low-level part of the interrupt system detects that an interrupt occurred inside this range the interrupted thread is advanced to the end of the atomic procedure. This technique is fast in the common case but incurs the overhead of an additional range check of the instruction pointer in the interrupt handler. It increases interrupt latency when the interrupt occurred inside the atomic procedure, because the procedure must first be finished. But the most severe downside of this technique is, that it inhibits inlining of memory accesses. Similar techniques are described in [19], [134], [128], [159]. We implemented atomic code and evaluated its performance (see Figure 12.5 in Chapter 12). We measured no significant difference between disabling interrupts and using atomic code. As the atomic code implementation adds an considerable complexity to the kernel we will avoid its use.

5 Multiprocessor support

The system already contains basic multiprocessor support. CPUs are managed by a CPU-specific global scheduler. Portal calls between threads that run on different CPUs are only allowed, when the domain has the associated CPU portal. In a multiprocessor system the portal call mechanism could be extended to allow the service create one thread per available CPU.

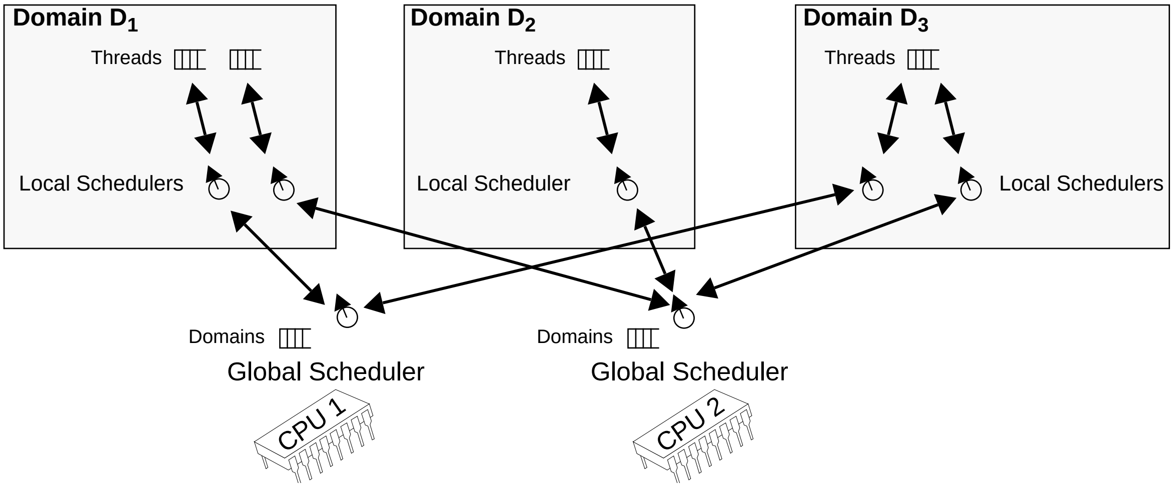
Figure 7.7 displays an example scheduler configuration for an SMP system. Each CPU has its own global scheduler. Each domain has one domain-local scheduler per available CPU. A domain may not be allowed to use all CPUs. To use a CPU, the domain must obtain a CPU portal for the specific CPU. Restricting domains to run a specific CPUs conflicts with the handoff scheduling strategy that can be used during a portal call. When the client thread runs on a CPU that is not available for the server domain, handoff scheduling can not be used.

6 Performance evaluation

We are interested in the effect of the scheduler on the performance of portal invocation. Table 5.1 of Chapter 5 stated 750 cycles for a portal invocation. This was measured using a global kernel scheduler, which is configuration (a) in Figure 7.1. The extra communication between the global kernel scheduler and the local Java-schedulers in configurations (c) and (e) is not for free. The time of a portal call increases from 750 cycles to 920-960 cycles if Java schedulers are used and either the calling domain or the called domain is informed about the portal call (see Section 2.3). If both involved domain schedulers are informed about the portal call the required time increases to 1180 cycles. If the schedulers are not informed, a portal invocation does not involve scheduling overhead, because of the use of handoff scheduling. When using handoff scheduling the scheduler configuration does not influence the cost of a portal invocation.

Figure 7.7: Multiprocessor scheduling in JX

On an SMP system each CPU runs its own global scheduler. A domain that is allowed to run on the CPU has a local scheduler that is connected to the global scheduler of the CPU. If a domain runs on more than one CPU it has more than one local scheduler. Whether the domain-local scheduler uses a single runqueue or separate runqueues for each CPU depends on the implementation of the local schedulers. This figure uses scheduler configuration (c) of Figure 7.1.



7 Summary

This chapter described the mechanisms of the JX system that are related to processor scheduling. It described the possible scheduler configurations and the interface between the microkernel and the scheduler. A design goal was to allow each domain to control scheduling and use domain specific scheduling strategies. Three of the scheduler configurations allow to use domain-specific schedulers. In these configurations the scheduler is split into a global and a domain-local scheduler. The global scheduler schedules domains and the local scheduler schedules the threads of a domain. The global, as well as the domain-local scheduler can be deployed in the type-safe instruction set. If the domain-local scheduler is not trusted, an additional thread switch to the thread of the local scheduler is required. This scheme is similar to scheduling in UNIX with user-level threads: the kernel schedules processes and each process may use an arbitrary user-level threads library to schedule its threads. An important design goal of the local scheduler interface was the ability to implement different preemptive and non-preemptive scheduling strategies. Using a non-preemptive scheduler in a domain avoids locking overhead on a uniprocessor. Although the program is structured to use multiple threads these threads are switched only when they explicitly block or yield the processor. The two-level scheduler configurations proved useful when porting the system to a multiprocessor. The difference to the uniprocessor is that there is a global scheduler per CPU and domains may contain more than one local scheduler. We finally analyzed the impact of scheduler configurations on the cost of a portal invocation.

CHAPTER 8 *Memory Management*

Besides the CPU memory is the other fundamental primary resource. This chapter describes the memory managment architecture of JX. The first part of the chapter explains the problems of the global memory management, the second part explains the memory object abstraction.

1 Global and domain-local memory management

Memory protection in JX is based on the use of a type-safe instruction set. No memory management hardware (MMU) is necessary. The whole system, including all applications, runs in one physical address space. This makes the system ideally suited for small devices that lack an MMU. But it also leads to several problems. In a traditional system fragmentation is not an issue for the user-level memory allocator, because memory that is allocated but not actively used, is paged to disk. In JX unused memory is wasted main memory. So we face a similar problem as kernel memory allocators in UNIX, where kernel memory usually also is not paged and therefore a scarce resource. In UNIX a kernel memory allocator is used for vnodes, proc structures, and other small objects. In contrast to this the JX kernel does not create many small objects. It allocates memory for a domain's heap and the small objects live in the heap. The heap is managed by a garbage collector. In other words, the JX memory management has two levels, a global management, which must cope with large objects and avoid fragmentation, and a domain-local garbage-collected memory.

Global memory management. The global memory is managed by using a bitmap allocator [187]. This allocator was easy to implement, it automatically joins free areas, and it has a very low memory footprint. On the other hand there is nothing in the system's design or implementation that prevents us to use another allocator.

Domain-local memory management. A domain has two memory areas: an area where objects may be moved by the garbage collector and an area where they are fixed. In the future, a single area may suffice, but then all data structures that are used by a domain must be movable. Currently, the fixed area contains the code and class information. Moving these objects requires an extension of the system: all pointers to these objects must be known to the GC and updated; for example, when moving a code component the return addresses on all stack frames must be adjusted.

Stack overflow detection and null pointer check. A system design without MMU means that several of their responsibilities (besides protection) must be implemented in software. One example is the stack overflow detection, another one the null pointer detection.

- Stack overflow detection is implemented in JX by inserting a stack size check at the beginning of each method. This is feasible, because the required size of a stack frame is known before the method is executed. The size check has a reserve, in case the Java method must trap to a runtime function in DomainZero, such as *checkcast*. A stack size check must be performed whenever a method is entered. To store its local and temporary variables the method needs a stack frame of a known size. The check code must test whether the stack has enough space for the frame and otherwise throw an exception or enlarge the stack. Enlarging the stack is only possible with a more sophisticated stack management that currently is not implemented. The check code must be fast because it is executed very often. We implemented two versions of the check. Version 1 (STACK0) aligns all stacks at an address that is a multiple of the stack size, which must be a multiple of 2. The check code adds the frame size to the stack pointer and sets the lower bits of the result to zero. If the result is larger than the original stack pointer the frame would overflow the stack and an exception is thrown. Version 2 (STACK1) uses the current thread pointer to access the stack boundary that is stored in the TCB. It compares the stack pointer plus the frame size to the stack boundary and throws an exception if it is greater.
- The null pointer check currently is implemented using the debug system of the Pentium processor. It can be programmed to raise an exception when data or code at address zero is accessed. On architectures that do not provide such a feature, the compiler inserts a null-pointer check before a reference is used.

2 Heap

Objects in Java and other type-safe runtime systems, such as Inferno and ML, are stored on a garbage collected heap [100]. The garbage collector usually is able to move the objects to compact the heap. Moving low-level data structures, such as thread control blocks and stacks, was a special challenge that we had to cope with.

Several different garbage collection algorithms can be used. No single GC algorithm will fit all applications. The Inferno VM, for example, uses reference counting[1] for predictable response times. A reference counting GC has disadvantages, such as high runtime overhead, bad multiprocessor scalability, and the need for an additional cycle collecting mechanism, that make it unsuitable for many applications. Therefore we decided not to use a single GC but to define an interface between the GC and the rest of the runtime system to allow different GCs to be used. Currently four collectors are implemented: a copying collector with fixed heap size (COPY), a copying collector with dynamically changing heap size (CHUNKED), and a compacting collector (COMPACTING).

COPY. This collector is an exact, copying, non-generational GC. An exact GC always knows whether a word is a reference or a data value[2]. A copying collector starts with a root set of object references, copies the object graph that is spawned by these references to a new heap and deallocates the old heap.

[1] And an additional mechanism to periodically collect cycles.

[2] This is in contrast to consevative GCs that do not have this information and must assume that a word is either a reference or a data value. They can be used for unsafe languages, such as C and C++. The shortcoming of these collectors is the inability to move objects, because they can not update references to objects.

The JX implementation of this algorithm does not need any additional data structures and is not recursive (see Algorithm 8.1).

Figure 8.1: Copying garbage collection

- Initial setup:
    - There are two semi-spaces S_1 and S_2
    - S_1 contains the original objects; S_2 is empty
    - There is a root set of references consisting of all stacks, all static variables, and all object references that are stored by the microkernel
- Algorithm:
    - Shallow copy all objects that are directly referenced by the root set to semi-space S_2
        - When copying an object (old object) mark the object in semi-space S_1 as copied and write a forwarding pointer to the object of semi-space S_2 into the old object
    - Scan all objects of semi-space S_2
        - If the object contains an object reference and the object has already been copied, replace the original object reference by the value of the forwarding pointer
        - If the object contains an object reference and the object has not been copied, copy the object and create a forwarding pointer

Two so called semi-spaces are used as a heap. Objects are allocated in one semi-space. When this semi-space runs full the garbage collector copies all live objects into the other semi-space, which then is also used to allocate objects.

This collector has some interesting properties:

- After a GC run the heap is in compact form. There is no need to search for free space during an allocation. Objects are always allocated at the top of the heap by simply advancing a pointer, which is a very fast operation.
- Deallocation is done by discarding the original semi-space. Therefore the time of one GC is independent from the number of dead objects, it is proportional to the number of live objects.
- Because objects are moved, pointers to objects must be updated. It is necessary to be able to locate all pointers to objects. This requires special attention within the microkernel (see Section 2.2).

CHUNKED. The heap consists of small linked chunks instead of a large block. This allows the heap to grow and shrink according to the memory needs of the domain. A group of domains can use a certain amount of memory cooperatively. A limit can be specified for the sum of the heap sizes of such a group. If this limit is reached and a domain needs to allocate additional memory a garbage collection is started in this domain and if the collection does not releases the necessary amount of memory the domain is blocked, waiting for the other domains in the group to release memory.

COMPACTING. The compacting collector is also an exact collector that moves objects. But in contrast to a copying GC it does not use two semispaces and does not change the order of the objects. The collector operates in four phases. In the first it marks all reachable (live) objects, in the second phase it calculates the new address of all live objects. In the third phase it corrects all references to live objects, and in the final phase it copies the contents of the object to the new location. The second phase uses two pointers into the heap: top and current. Top marks the end of processed live objects, current marks the start of unprocessed objects (the current position of the heap scan). Initially top and current are set to the start of the heap. If the object at the current address is marked it is moved to the top address and the top address is set to the end of the moved object. If it is not marked it is not moved. The current pointer is then set to the next object. At the end of this phase the current pointer points to the original end of the heap and the top pointer to the new end of the heap.

2.1 GC interface

GC runs of JX domains are completely independent from each other. The independence allows for different GC implementations that even use different objects layouts. Some GCs need additional information per object, for example a mark bit or a generation number, some GCs will reorder the fields of an object to better use CPU caches, some GCs will even compress objects or store a group of objects, such as an array of booleans, in a space efficient manner. For this to work the garbage collector implementation must be hidden behind an implementation independent interface. The interface is realized as a domain-specific function table (see Figure 8.2).

Figure 8.2: Garbage collector interface

ObjectHandle allocDataInDomain(struct DomainDesc_s * domain, int objsize, int flags);

> This function is called to allocate a new object on the domains heap.

void done(struct DomainDesc_s * domain);

> This function is called when the domain terminates. The garbage collector should release all resources, especially the memory allocated for the heap.

void gc(struct DomainDesc_s * domain)

> A garbage collection should be performed.

boolean(*isInHeap) (struct DomainDesc_s * domain, ObjectDesc * obj)

> The garbage collector tests whether the given object pointer is inside its heap.

void (*walkHeap) (struct DomainDesc_s * domain, HandleObject_t handleObject)

> The garbage collector applies the handler function to all objects on its heap. This function can be used by the garbage collector independent code to operate on all objects on the heap without knowing the organization of the heap.

2.2 Moving special objects

TCB. The GC must know all references to thread control blocks (TCB) and update them. Threads are linked to threads in other domains. During a portal call, for example, the sender thread is appended to a wait queue in the service control block. When the TCB is moved also the external references to it must be updated. For this purpose the TCB has a link to the SCB it waits for.

A TCB object cannot simply be copied to another domain. When a TCB object crosses a domain border (for example from domain D_1 to domain D_2 in Figure 8.4) a proxy for the TCB is created on the target heap. The proxy, which has the type ForeignCPUState, references the domain of the real TCB (domain D_1) by using the domain pointer / domain ID mechanism (see Section 4 of Chapter 4). To reference the TCB it is not possible to simple use a pointer to the TCB object, because this object could be moved during a GC in domain A. Therefore the proxy uses the unique thread ID to reference the TCB. Because it is very expensive to find the TCB given the thread ID we use the following optimization. The proxy additionally contains a direct pointer to the TCB and the number of the GC epoch of domain A. The GC epoch of a domain is a strictly monotonic increasing number. It is incremented when a GC is performed in the domain. It is guaranteed that if the GC epoch number has not changed objects are not moved.

Every pointer to a TCB must be known to the garbage collector. The GC must be able to find and update these pointers when moving the TCB. For scalability reasons these pointers must be found without scanning the heap of other domains. An example for such a pointer is the link between a service thread and its client, which is created during a portal invocation. To return the result of a portal call the service thread needs a reference to the client thread. There are two ways to implement such a reference.

- The server's TCB contains the thread ID of the client thread. On return from a portal call the corresponding client TCB must be found. As this may involve a linear search over all threads of the client domain the mapping from thread ID to TCB could be very slow. Therefore an optimization must be used. Together with the thread ID a direct TCB pointer, a GC epoch number, and a pointer to the client's DCB are stored in the server's TCB. The direct TCB pointer can be used as long as the GC epoch number is identical to the client domain's GC epoch. The current GC epoch is stored in the client's DCB. If the epoch numbers differ a GC in the client domain could have moved the client TCB. In this case the thread ID is used to find the TCB and the TCB pointer and GC epoch number can be updated.
- The reference to the client TCB is stored in the server's TCB as a direct pointer called mostRecentlyCalledBy (see Figure 8.3). This pointer must be updated when the client TCB is moved during a garbage collection in the client domain or when the client thread or the client domain is terminated. To detect whether a TCB is referenced via a mostRecentlyCalledBy pointer the client TCB has a pointer to the server's TCB (blockedInServiceThread). This pointer must be updated similar to the mostRecentlyCalledBy pointer.

The current implementation uses the second alternative, because it uses fewer space in the TCB and is faster in the common case that no GC occurred during a portal invocation.

A similar problem is caused by fast portals that represent a thread of another domain. These portals, called ForeignCPUState, also need a reference to the TCB of a thread in another domain. The second alternative can not be used because potentially there are many ForeignCPUState portals referencing the same TCB. Therefore ForeignCPUState portals are implemented using the first alternative (see Figure 8.4).

Stack. A previous implementation of the stack overflow check required that stacks are aligned at a multiple of their size (see Section 1). This alignment requirement prohibited allocating them on the heap because of heap fragmentation. The current implementation (called STACK1 in Section 1) does not need aligned stacks and allocates all stacks on the heap. When the stacks are moved the frame

Figure 8.3: Management of Thread Control Blocks during a portal communication

During a portal communication the TCBs of the client and the server thread are connected in both directions. The connection from the server TCB to the client TCB (mostRecentlyCalledBy) is required for delivering the results of the invocation. The connection from client to server TCB (blockedInServiceThread) is required to update the mostRecentlyCalledBy link during a garbage collection.

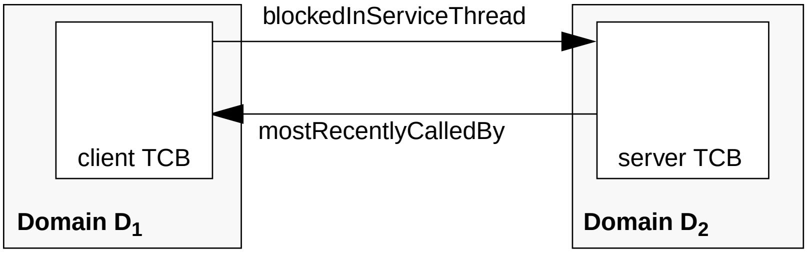
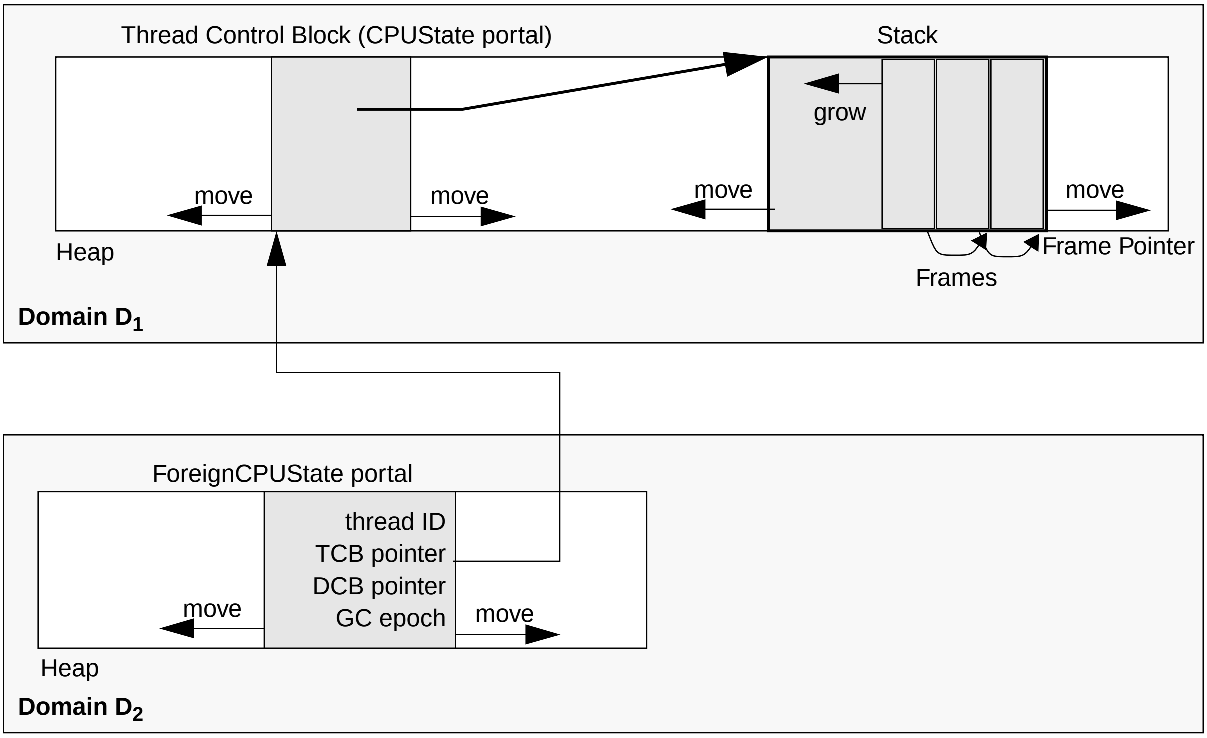


Figure 8.4: Relation between TCBs, inter-domain TCB references, and stacks

Thread Control Blocks and stacks live on the heap of a domain. A stack consists of linked stack frames. Another domain can hold a reference to a thread control block by using ForeignCPUState portal. Because TCBs as well as stacks can be moved during a garbage collection, the ForeignCPUState portal contains the thread ID of the foreign TCB, the GC epoch and a direct pointer to the TCB.



pointers (which link the stack frames), the CPU context in the TCB (stack pointer and frame pointer registers), and the stack pointers in the TCB must be corrected. A complication occurs when the collector thread moves its own stack. It must detect this and switch to the new stack before releasing the old heap. As the stack was copied in the shallow copy phase (see Figure 8.1) it must copy the stack again to reflect the actual execution state. The current implementation switches to the new stack immediately after correcting the frame pointers of this stack and runs the rest of the collection on the new stack.

SCB. Service Control Blocks (SCBs) and Service Pools are also allocated on the heap. They contain references to TCBs which must be updated.

Domain. Domain Control Blocks could be allocated on the heap of DomainZero. To allow direct pointers to DCBs we used a dedicated memory area for DCBs and domain portals that contain a DCB pointer and a domain ID. When the ID in the DCB equals the ID in the domain portal the DCB pointer is valid otherwise the DCB has been reused.

2.3 Garbage collection and interrupt handling

All interrupt handlers are written in Java. All of the current GC implementations operate non-incremental, i.e., no mutator thread[1] is allowed to run during a collection because the heap is not consistent during this time. Therefore all interrupts that are serviced by the domain must be blocked. On a PC architecture the programmable interrupt controller (PIC) is used to disable dedicated interrupts. A blocking in software by remembering the interrupt (set a flag) and returning from the core IRQ handler is not possible with level-triggered interrupts that must be acknowledged at the interrupting device to be deactivated.

To avoid that a garbage collection becomes necessary during interrupt handling the heap contains a reserve that is only available for object allocation in interrupt handler threads. As first-level interrupt handler should be very short and should not allocate much memory this reserve should be sufficient. If it is not, an exception is thrown.

2.4 Garbage collecting and timeslicing

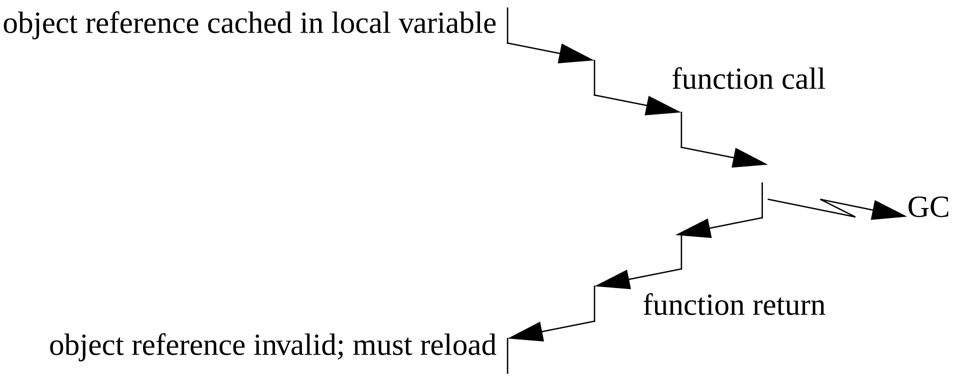
Scheduling threads of one domain preemptively means that a thread may be at an arbitrary instruction when another thread requires a collection. A technique that is also used in JX is to advance all threads that are located in Java code to a safe point [3]. A safe point is a point where the execution state of the thread is known: the types of the saved registers and the types of all stack positions are known. It is difficult to advance a thread that currently executes a core function, because to obtain a stack map, which is necessary for a exact collector, the C compiler must be modified to generate such maps. Therefore C code must either disable interrupts or register references.

A collection may also be necessary in kernel code when the kernel allocates objects. The code must be carefully written to check whether a GC occurred and reload variables (see Figure 8.5).

[1] All threads, besides the GC thread, that run in the domain and modify the heap are called mutator threads.

Figure 8.5: GC in kernel functions

Kernel code must be programmed very carefully to not hold object references across function calls. They should assume that a garbage collection can invalidate the reference and should reload all references after a function call. An exception to this rule are functions that are specified not to cause a garbage collection.


object reference cached in local variable
function call
GC
function return
object reference invalid; must reload

2.5 Garbage collection of portals and memory objects

Portals and memory objects are shared between domains. Both refer to a central data structure. Portals have a reference to a service control block (SCB) in another domain and memory objects have a reference to a memory control block (MCB) in DomainZero. Reference counters are used to reclaim the SCB and MCB. When a domain terminates or portals or memory objects become garbage the respective reference counter must be decremented during a finalization cycle. How this is actually realized depends on the installed garbage collector. The copying GC moves live objects to a second heap and sets a flag in the original object's header to indicate that the object was copied. At the end of the GC cycle all objects that have not set this flag are dead. During the following finalization phase the heap is scanned using the walkHeap() function of Figure 8.2 and the reference counters of all dead portals and memory objects are decremented. When the domain is in the process of being terminated all objects are considered dead.

2.6 Future work

Similar to scheduling garbage collection should be removed from the microkernel. This is much more difficult, because it must be guaranteed that the untrusted GC preserves the type of objects while copying them. Although there are recent advances [183] several open problems have to be solved before the technology can be used for a real system.

Most commercial JVMs use generational garbage collection. Generational GC assumes that most objects die young and if an object survives a GC it lives very long. The GC algorithm tries to separate young and old objects in two heaps and garbage collect the heap of the young objects more often than the heap of the old objects. The heap of the young objects often is called incubator or nanny. All objects are allocated in the nanny. When the nanny is garbage collected all live objects are copied to the old object’ s heap. Because old objects are not traversed GC time is reduced. To avoid that old objects contain references to new objects write barriers can be used.

2.7 Summary

This chapter discussed issues that are related to the management of main memory in the JX system. The memory hierarchy was described. This hierarchy consists of a global memory management at the lowest level and on top of this a domain-local memory management and a heap management. The heap memory is managed by a garbage collector, which can be selected on a per-domain basis.

3 Memory Objects

Efficiently managing buffers is one oft the challenging tasks of an operating system. This section describes the memory management facilities of the JX operating system. In a “pure” JVM there are only objects and arrays. A JVM that must serve as a foundation for an efficient operating system must provide a richer interface for buffer management. For this purpose JX provides memory objects. They have four non-orthogonal properties: sharing, revocation, splitting, and mapping. Memory objects can be shared between domains. There is a revoke() method that atomically returns a new memory object representing the same range of memory and revokes access to the old one. A memory object can be split in two memory objects that together represent the original memory range. Access to the original object is thereby revoked. And a memory object can be mapped to a class structure and accessed like an object.

3.1 Lifecycle of memory objects

3.1.1 Creation

Memory objects are created by the microkernel as fast portals. The MemoryManager portal allows to allocate a memory object of a specified size, a ReadOnlyMemory object with a specified size, or a DeviceMemory object at a specified address with a specified size.

3.1.2 Destruction

An explicit deallocation was not an option, because this would contradict the whole philosophy of the JX system. We need an automatic mechanism - a garbage collector for memory objects. Memory objects are shared between domains and the memory GC is not allowed to stop all domains and should not posses any global knowledge. This makes the memory GC similar to a distributed GC. One of the simplest distributed GC is reference counting. Reference counting can be used, because there are no cycles in memory references (a memory object only contains a reference to its parent memory object).

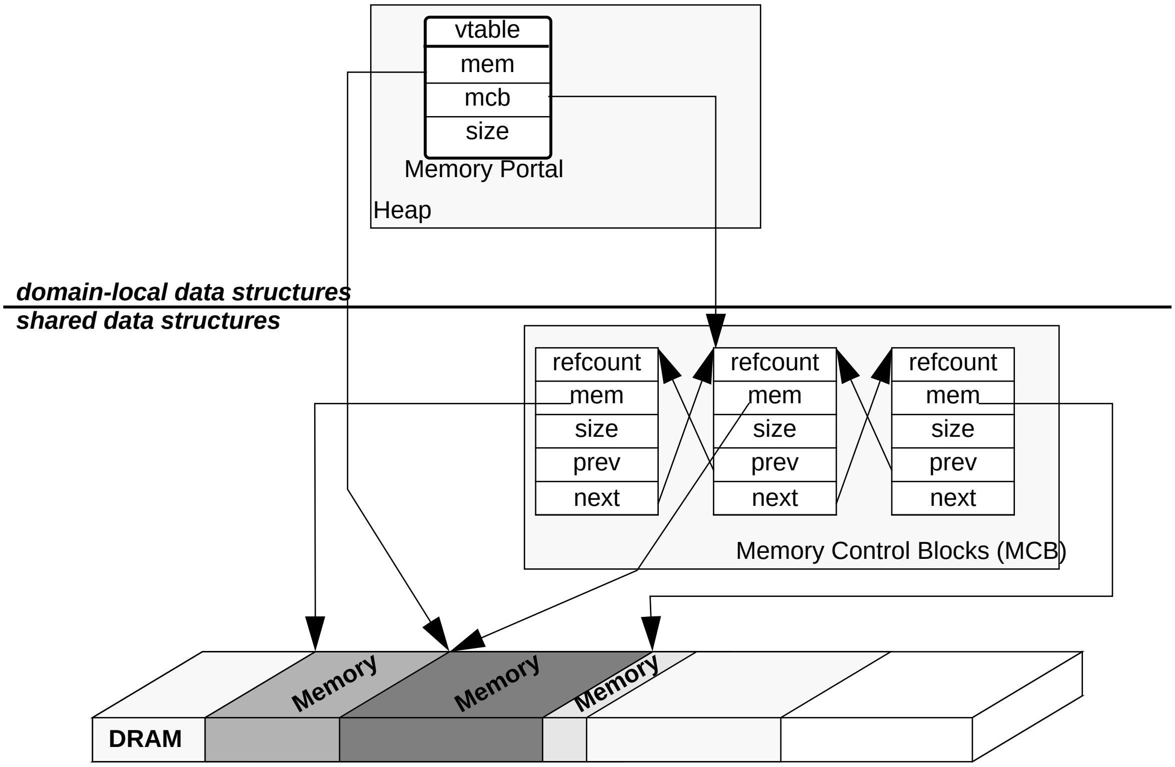
3.2 Implementation

This section describes one possible implementation of memory objects. As the data structures used to maintain memory objects are only accessed by the memory object implementation different implementations are possible. We experimented with implementations that improve efficiency by providing only a subset of the four features sharing, revocation, splitting, and mapping. The results of these experiments are presented in the performance evaluation section at the end of this chapter.

Sharing. Because memory objects can be shared between domains a global data structure (Memory Control Block MCB) is required to maintain the state of the memory object (see Figure 8.6). This shared data structure is referenced by memory portals that are located on the heaps of different domains. This state or meta information of the memory object consists of a flag used for revocation and a reference count used to garbage collect memory regions.

Figure 8.6: Data structures for the management of memory objects

Domains can access "plain" memory by using a memory portal. The memory portal is a fast portal that contains a pointer to a memory control block (MCB), size information, and a pointer to the memory area. This information is stored in the "additional data" part of the fast portal (see Figure 5.12 of Chapter 5). MCBs are maintained in a doubly-linked list. Each MCB is responsible for an area of memory that does not overlap with the area of any other MCB.



Revocation. If a memory object supports revocation the revocation check and memory access must be performed in an atomic operation. Systems that do not perform these operations atomically are said to have the time-of-check-to-time-of-use (TOCTTOU) flaw. An example of this kind of flaw is the STOP-PROCESS-ERROR of Multics described in [23]. Operations of memory objects are short (with the exception of memory copy) and do not block. To realize atomicity efficiently (this means without using mutual exclusion locks) on a uniprocessor interrupts can be disabled or *atomic code* (see Section 4.3 of Chapter 7) can be used. On a multiprocessor a spinlock is used. The spinlock is implemented using an atomic bus transaction to read and write a data word. Several CPU instructions are available for this purpose, such as test-and-set or compare-and-swap (CAS). When using CAS the revocation flag and the lock flag of the spinlock can be arranged in the same word and checked with a single CAS instruction.

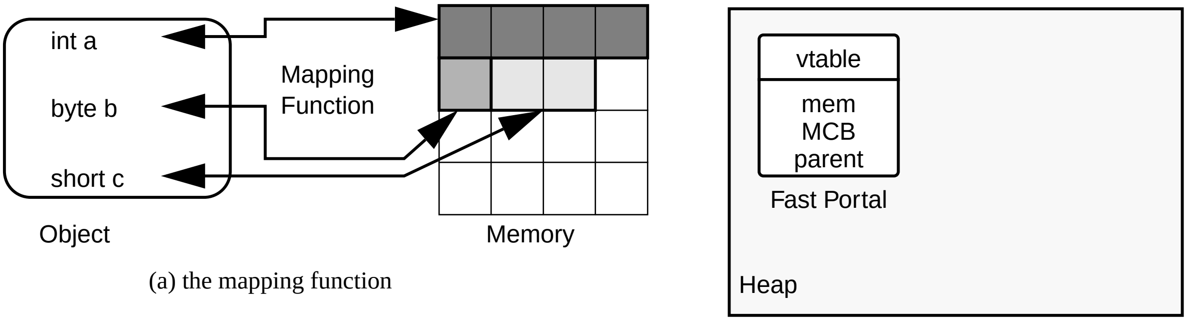
Splitting. Often it is desirable to pass only a subrange of a certain memory to another domain. Each memory type supports the creation of subranges. Splitting and revocation are not orthogonal. A natural semantics for revocation says that the contents of the memory can only be changed by the revoker after a revocation. If a memory is splitted there are still references to the unsplitted memory. A split must therefore revoke access to the original memory.

Mapping. A memory object is a data container without data structure. Many memory objects contain data that is structured. For example, a memory object holding a disk block of inodes has the structure of a field of inodes. A memory object that holds a network packet can be structured in a header and a payload. To allow structured access to a memory object JX supports the abstraction of mapping an object to a memory object. The memory object contains the state of the object. An access of a field of the object accesses the underlying memory object. The object is an "instance" of a class that is marked using the marker interface MappedObject. The bytecode verifier ensures that no instances of such a class can be created by using the regular object creation mechanism (the new operation). To specify the byte order that should be used when mapping a sequence of bytes to an integer or short data type, the mapping class uses a subtype of MappedObject: either MappedLittleEndian or MappedBigEndian. The bytecode-to-nativecode translator is then able to generate the necessary machine code instructions to access the memory in the correct byte order.

Figure 8.7: Mapping a memory range to a class

(a) The mapping function describes how a field access is translated into a memory access.

(b) A mapped memory is represented by a fast portal. The portal object contains a pointer to the begin of the memory area, a reference to the memory control block (MCB), and a reference to a parent mapping if this object is part of a larger aggregation (otherwise the parent link is null).



(a) the mapping function

(b) the fast portal representing a mapped memory

The mapping function currently only supports primitive integer types (byte, short, char, int, long). The usefulness of the mapping mechanism would be improved considerably when aggregated data types are supported, i.e., an object that contains not only primitive fields but also other objects. The way Java handles the aggregation—also called parts/whole—relation makes this very difficult. The *whole* object contains not the data of the *part* object but only a reference to it. This reference has no pointer to its (logically) enclosing object and can be used as an independent reference. This problem can be solved by introducing hierarchical mappings: each mapped object can be part of a larger aggregation and the portal object contains a reference to the enclosing mapped object.

3.3 Problems

Efficiency. As already mentioned in Section 3.2 not all features of a memory object are needed by all applications. Several applications do not pass memory objects to other domains but require the revocation feature within their own domain. We implemented a memory object that creates the central data structure only when the memory object crosses a domain boundary. The revocation flag is part of the heap-managed memory proxy until a global MCB object is created.

Global resources. Memory objects require a central data structure to allow garbage collection and revocation of shared memory. This contradicts our design principle of not allowing a domain to allocate global resources. We alleviate this problem by using per-domain quotas that limit the number of MCB objects and the amount of memory a domain is allowed to allocate.

Hardware references. Memory objects are also used in device drivers. A device driver may use the memory for DMA transfers to and from a device. This means that the hardware has an implicit reference to the memory object and the memory should not be garbage collected, even if there are no other references to it. There are two solutions for this problem. The programmer of the device drivers ensures that the memory object stays alive by making it reachable from the root set of live references. This will usually be the case because the memory object that is involved in the DMA transfer will be part of a data structure to manage the currently executing device operation. A more robust solution is to support the programmer in keeping the memory object alive. An implicit hardware reference can only be created by asking the memory object for the start of its physical memory area. This start address is then written in a device register or in a DMA table. The system can support the programmer by incrementing the reference count in the MCB object whenever the physical start address of the memory is queried. This requires that the programmer invokes a method to decrement the reference count once the memory is no longer used by the hardware.

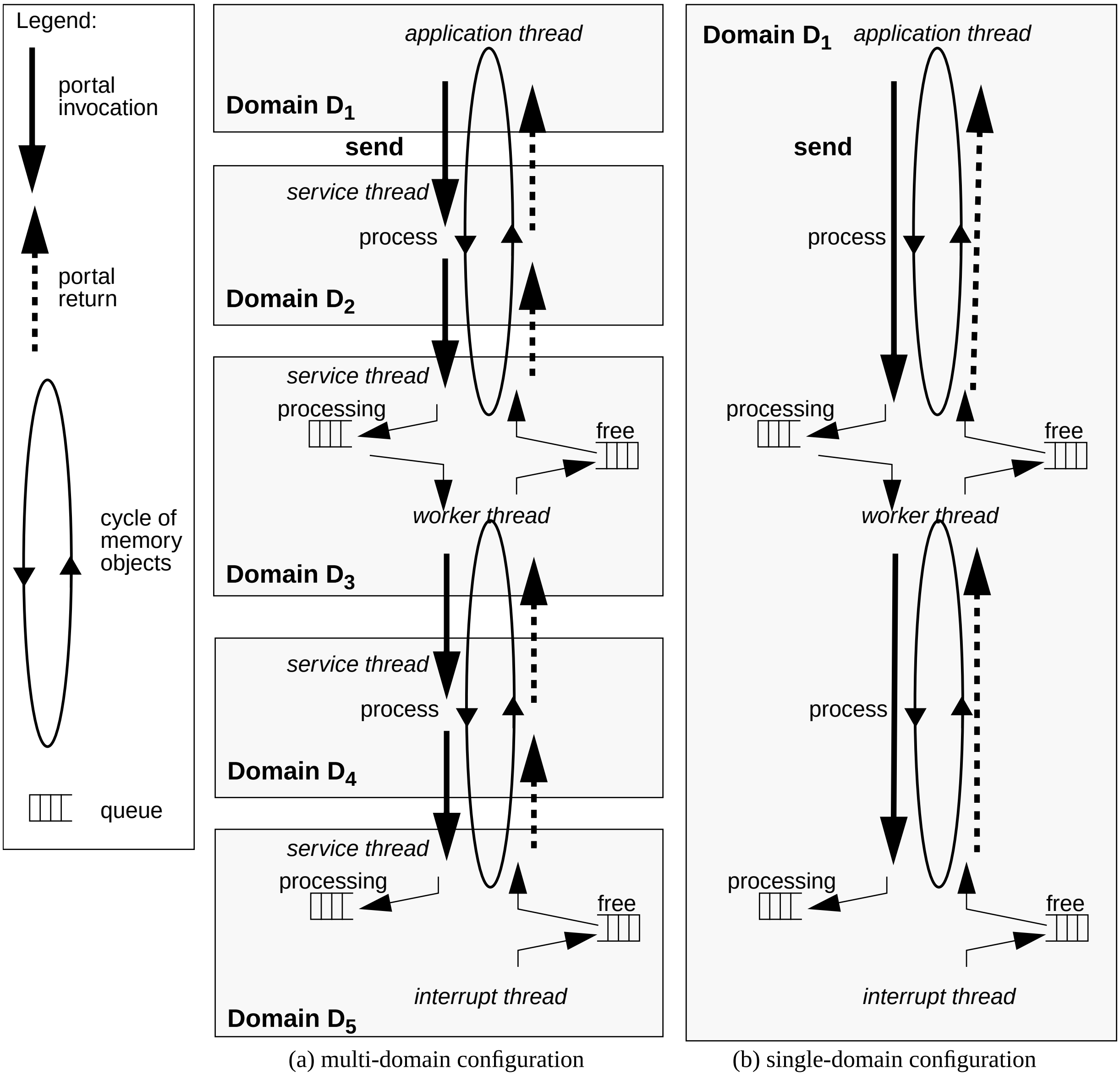
3.4 Buffer management

Memory objects are used as buffers, for example in the network stack. Network packet processing requires an efficient buffer management. Allocating a buffer and passing a buffer to another domain must be fast operations. Some kind of flow control is needed to avoid that a domain runs out of buffers. A domain that wants to receive a packet passes a buffer to the network domain. In exchange it gets a (different) buffer that contains the received data. The send operation returns an empty buffer in exchange for the buffer that contains the data. The principle of this operation is illustrated in Figure 8.8.

Figure 8.8: Principle of buffer management

(a) This figure illustrates the principle of buffer management if the network protocol stack is distributed across multiple domains. Domain D_1 is the application domain. Domain D_2 contains the application, domain D_5 the lowest layer of the network stack. During the send operation a memory object is passed from D_1 to D_2 to D_3 where it is enqueued in a processing queue. The service thread of D_3 dequeues a memory object from the free queue and passes it as return value to domain D_2, which passes it to D_1. Another thread (*worker thread*) asynchronously dequeues the memory object from the processing queue and passes it on to D_4, which processes it and passes it to D_5. The service thread of D_5 makes the memory available for the DMA hardware and inserts it in a processing queue. The device asserts an interrupt when it completes the DMA transfer and the memory object can be moved from the processing queue to the free queue.

(b) The same as (a) but all components run in a single domain. The operation is identical, only there are no service threads.



(a) multi-domain configuration (b) single-domain configuration

4 Performance evaluation

This section evaluates the performance of the global memory manager, memory objects, and stack size checks.

4.1 Global memory management

The bitmap-based global memory manager has a low memory footprint. Using 1024-byte blocks and managing about 128MBytes or 116977 blocks, the overhead is only 14622 bytes or 15 blocks or 0.01 percent.

4.2 Stack size check

We used a microbenchmark to measure the overhead of the stack size check. A method is called in a loop of 100,000 iterations. There is no measurable difference between the two check variants in this microbenchmark.

Tab. 8.1 Virtual method invocation with different stack size checks

Operation	Time (ns)
no check	22
check STACK0	40
check STACK1	40

4.3 Memory objects

Table 8.2 shows the performance of the memory operations create, revoke, split.

Tab. 8.2 Performance of memory operations

Operation	Time (ns)
Create Memory	2,496
Create DeviceMemory	750
Revoke Memory	900
Split Memory	1,780
set32, 1 KB block	18,400

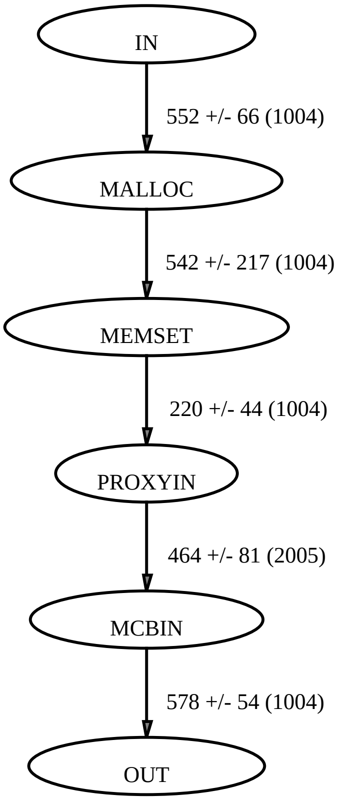
Tab. 8.2 Performance of memory operations

Operation	Time (ns)
set32	50
get32	45
Create mapping	570

Figure 8.9 displays the time in nanoseconds between event log points in the memory allocation code.

Figure 8.9: Time between events during the creation of a memory object.

Time is given in nanoseconds with standard deviation. The number in parenthesis is the number of transitions between two events. The time between IN and MALLOC is spent allocating the memory from the global memory management. The time between MALLOC and MEMSET is spent filling the memory range with zero. The time between PROXYIN and MCBIN is spent creating the proxy object (memory portal). The time between MCBIN and OUT is spent creating the MCB.



4.3.1 Delayed MCB creation

When a memory object is used only within one domain (no sharing feature) no central data structures must be created for this memory object. Table 8.3 shows the effect of creating MCBs lazily. The saved time of about 500 nanoseconds is consistent with the event trace of Figure 8.9.

Tab. 8.3 Memory creation time with and without the optimization *delayed MCB creation*

Operation	Time (ns)
w/o delayed MCB creation	2,496
with delayed MCB creation	1,904

4.3.2 MappedMemory

Figure 8.10 shows an example of the use of MappedMemory. Figure 8.11 shows an equivalent C program. The C program is *not* compiled with an optimization option. By inspecting the assembler code we recognized that when using the “-O2” optimization option the gcc compiler moves data accesses out of the loop. The mapped access and get/set test was compiled with inlined memory access. The results of the benchmark are presented in Table 8.4

Tab. 8.4 Memory access time of mapped memory vs. get/set

This table shows the performance of different memory access mechanisms. The column titled "get-set/mapped" contains the ratio between the access time when using the get-set interface and the access time when using mapped memory. The last two columns compare these interfaces with time of the Linux benchmark.

operation	get-set interface (µs)	mapped (µs)	get-set/ mapped	C prog. on Linux (µs)	get-set/ C-Linux	mapped/ C-Linux
4,000,000 write	73474	61010	1.20	42212	1.74	1.45
4,000,000 read	48000	32687	1.47	24664	1.95	1.33
1 write	0.018	0.015	1.20	0.010	1.74	1.45
1 read	0.012	0.008	1.47	0.006	1.95	1.33

The table shows the improvement of mapping compared to the set/get interface. This improvement is due to removed range checks. It also shows how the get/set interface and the mapped interface compare to a C program running on Linux. While access to a mapped object is still slower than access to a C struct the difference is rather small and is due to the additional indirection of MappedMemory compared to a C struct (see Section 4.3.2). We also measured the time required to create a mapping. This time is also important because mapping is performed on the performance critical path, for example, when receiving or sending network packets. The time to create 1000 mappings was 552 microseconds (or 552 nanoseconds per mapping). This means that creating a mapping is slightly slower

Figure 8.10: Measured code sequence for JX MappedMemory access

```
class MyMap implements MappedLittleEndianObject {
    int a; char b; short c; int d;
}

Memory mem = memoryManager.alloc(12);
VMClass cl = componentManager.getClass("test/memobj/MyMap");
MyMap m = (MyMap) mem.map(cl);
// -- START TIME --
for(int i=0; i<ntries; i++) {
     m.a = 42; m.b = 43; m.c = 44; m.d = 45;
}
// -- END TIME --
```

Figure 8.11: Measured code sequence for C struct access

```
struct MyMap {
    signed long a; unsigned short b; signed short c; signed long d;
};

char *mem = malloc(12);
struct MyMap *m = (struct MyMap*) mem;
// -- START TIME --
for(i=0;i<n;i++) {
      m->a = 42; m->b = 43; m->c = 44; m->d = 45;
}
// -- END TIME --
```

than creating an object (see Chapter 3, Section 4). This is reasonable because a mapping is represented by a proxy object that is created during the mapping process. These numbers show that mapping pays off only when the data is accessed very often. A mapped read saves 6 nanoseconds, a mapped write saves 8 nanoseconds. The time to map an object is equivalent to 92 read accesses or 69 write accesses. A network packet header is usually not accessed that often, therefore mapping will not improve performance but lead to more readable programs. Only an order of magnitude performance improvement of object allocation will make mapping practical.

5 Summary

This chapter described the memory management of the JX system. Similar to two-level scheduling memory management is separated into a global management and a domain-local management. The global management allocates the available main memory to domains. It uses a bitmap allocator with low memory footprint. The domain-local management uses almost all its memory for a garbage collected heap. All data is allocated on this heap[1]. This includes thread control blocks and stacks. A domain can select from a number of garbage collectors to manage this heap. A defined interface be-

[1] As already mentioned the current prototype uses a so-called fixed memory area that is used to store data structures that are not yet prepared to be moved by a garbage collector. This includes the domain-local part of a component.

tween the kernel and the garbage collectors allows to use different collectors. To cope with large amounts of memory and to share memory between domains memory objects have been introduced. They allow sharing, revocation, splitting, and mapping. The performance of several operations of memory objects and the performance impact of delayed memory control block creation was measured.

CHAPTER 9 *Device Drivers*

The purpose of a device driver is to hide the details of a specific hardware and present this hardware with an interface that can easily be used by other parts of the oparating system. In an object-oriented system such an interface is an object and in the specific case of JX it is a service.

Due to the enormous amount of new hardware that appeared in the last years, operating system code is dominated by device drivers. The overall goal of JX is to minimize the static and trusted part of the system. An investigation of the Linux kernel has shown that most bugs are found in device drivers [40]. Because device drivers will profit most from being written in a type-safe language, all JX device drivers are written in Java. An alternative to writing device drivers at the Java level would be resuing existing drivers that are written in C and available for other open source operating systems, such as Linux and FreeBSD. While in the short-run resuing existing drivers seems attractive we suspect it to be a source of instability and performance problems in the long run. A C driver can not be integrated into the system as seamlessly as a Java driver. It will always be a contaminant.

This chapter starts by reviewing the device driver architecture of UNIX and Windows2000, describes our device driver framework and the problems we encountered keeping the device drivers outside the trusted part of the system.

1 Device-driver architecture of traditional systems

We first outline the device driver architecture of two traditional OS architectures - Linux, as an example for UNIX-like systems, and Microsoft Windows 2000. From an high-level perspective device drivers in Linux and Windows are very similar. Devices are represented in a global, hierarchical name space. Linux, as most other UNIX clones, uses a device inode created by the mknod() system call in the virtual file system. Windows has a namespace that includes the file system as well as directories outside the file system that contain entries for devices (\Dosdevices and \device). A device entry is created by using the CreateFile() API call. In both systems it is possible to load a driver either at boot time or when the system is already running. While device management looks similar on the surface the details of the driver architectures differ considerably.

Linux. The driver is split into a top half and a bottom half. The top half is the first-level handler and runs with the actual interrupt blocked or all interrupts blocked[1]. The top half can schedule work to run in a bottom-half handler. The top and bottom halves are not executed in a process context. It is not allowed to block and can only perform atomic memory allocations. Bottom half functions are invoked in the scheduler and on return from a system call if they are marked by they top half. Because

[1] Note that the meaning of top-half and bottom-half is the opposite of their meaning in BSD UNIX [125], where the bottom-half is the first-level handler.

the number of bottom-half handlers is limited to 32 the kernel contains an additional mechanism for delayed execution: task queues. A task queue is a list of function pointers. A first-level interrupt handler can add an entry to this list. A bottom-half handler executes the functions contained in this list. Interrupt handlers synchronize by using interrupt priority levels (IPLs). Code that is executed under a specific IPL can only be preempted by an interrupt of higher priority.

Windows 2000. Windows 2000/XP/ME contains a driver model, called the Windows Driver Model (WDM) [195] [163] that is a superset of the Windows NT driver model. An I/O manager controls device drivers. Applications or kernel components communicate with the I/O manager to request I/O. The I/O manager communicates with drivers by using I/O Request Packets (IRPs). Drivers are arranged in a stack. The high-level drivers are the device drivers, the low-level drivers are the bus drivers. The stack contains at least two drivers, the so called function driver, and one bus driver. The I/O manager passes IRPs down the stack until the request is finalized. Interrupts are assigned priorities by using interrupt request levels (IRQLs), the equivalent of UNIX's interrupt priority levels. Interrupts are handled in an interrupt service routine (ISR). Complex work is not done in the ISR, instead a deferred procedure call (DPC) is used. A DPC is very similar to a bottom-half handler in Linux. DPCs run at DPC IRQL, which is lower than the IRQL of ISRs and allows low interrupt response time. DPC IRQL is higher than the IRQL of any thread which leads to high worst- case scheduling latencies [15].

2 JavaOS Driver Architecture

JavaOS [154], Sun's Java-based operating system, uses three levels of interrupt handlers. The first-level handler executes immediately, the second-level and third-level handler execute asynchronously similar to bottom-halves or DPCs. Although device drivers in JavaOS are written in Java, first-level and second-level interrupt handlers are not. Only third-level handlers are written in Java and execute in a regular Java thread. This requires that the native microkernel contains a native handler for each supported device. The reason for this design is the single-heap architecture of JavaOS which automatically leads to a single resource management domain. When using a mark & sweep garbage collector in such a design, garbage collection pauses are considerably and would lead to data losses when using devices with small buffers.

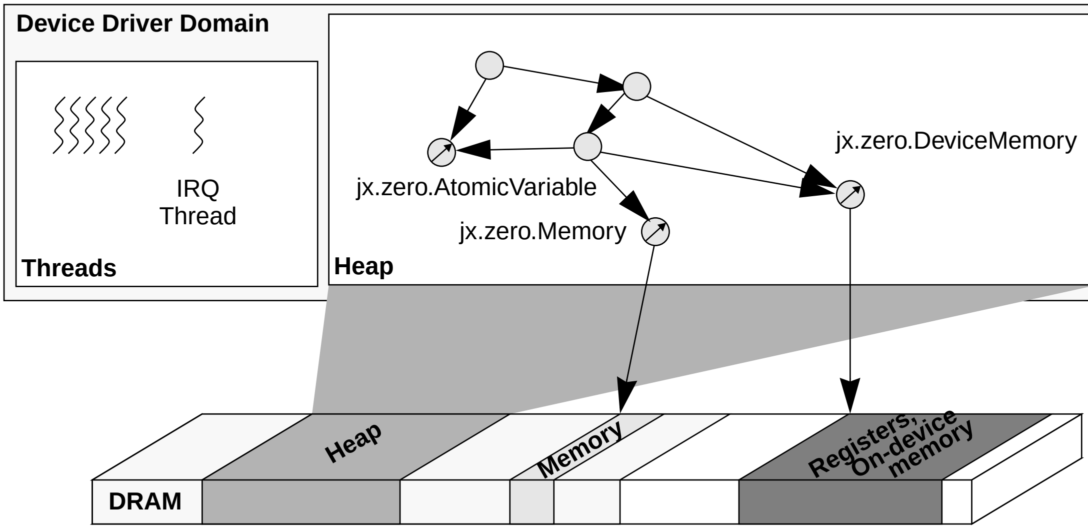
3 JX Driver Architecture

As the UNIX/Linux and Windows 2000 driver architectures are similar from a high-level perspective, the JX driver architecture is similar to them. This is not surprising, because it is enforced by the hardware architecture. And again the details are fairly different because of the completely different operating system architecture.

Device drivers can also be components that do not use the hardware at all, but emulate the functionality of the device. These drivers can be treated like normal applications. This section describes the environment for a device driver that needs to directly interact with the hardware to control a real device. These drivers have to handle interrupts, access device registers, access physical memory at a specific location, and control DMA transfers. An example structure of such a driver is displayed in Figure 9.1.

Figure 9.1: Example memory layout and structure of a JX device driver

The device driver uses a DeviceMemory portal to access the memory-mapped device registers and device memory. It uses a Memory portal to access main memory and an AtomicVariable (see Section 4) to synchronize its interrupt thread with other threads running in the domain.



3.1 Interrupt handling

Interrupts are handled in dedicated interrupt handler threads. These one-shot threads (see Chapter 7, Section 1) are created by the system when an interrupt handler is registered. Using the IRQManager service of DomainZero (see Figure 9.2), device drivers can register an object that contains a handleInterrupt() method. On a multiprocessor the installHandler() installs the handler for the CPU that is executing the installation method. An interrupt is handled by invoking this handleInterrupt() method. The method is executed in a dedicated thread while interrupts on the interrupted CPU are disabled. This would be called a *first-level interrupt handler* in a conventional operating system. A handleInterrupt() method usually acknowledges the interrupt at the device and unblocks a thread that handles the interrupt asynchronously. This thread is similar to a bottom-half handler in Linux or a DPC in Windows 2000. In contrast to these systems the thread is scheduled by the regular thread scheduler and has *not* automatically a priority that is higher than any other non-interrupt handler thread. Such a design is necessary to run tasks with real-time requirements on JX. In such a system user threads performing time critical tasks must have precedence over device handler threads. Although JX was designed with an eye towards real-time capabilities it is not yet possible to run real-time tasks out-of-the box. A real-time scheduler is missing as well as semaphores that support a protocol to cope with priority inversion.

Interrupt priorities. There is currently no need for interrupt priority levels as first-level interrupt handlers are very short. Making them preemptable only seems to introduce complicated synchroni-

Figure 9.2: InterruptManager and InterruptHandler interfaces

The InterruptManager is used to install and uninstall interrupt handlers. These handlers must implement the interface InterruptHandler and whenever the interrupt with the specified number is raised the handleInterrupt() method is executed in a dedicated interrupt thread.

```
package jx.zero;

public interface InterruptManager extends Portal {
    void installHandler(int irq, InterruptHandler handler);
    void uninstallHandler(int irq, InterruptHandler handler);
}
public interface InterruptHandler {
    void handleInterrupt();
}
```

zation problems (code that runs at different levels can not share data) and adds another “scheduling policy” that must be maintained and understood by the driver developer. Deciding which second-level handler thread to run next is a regular scheduling problem and not hard-wired into the device-driver architecture. Preemptable first-level handlers may be necessary if the device is very slow to access or requires complicated computations before interrupts can be acknowledged.

3.2 Register access

Drivers use DeviceMemory to access the memory-mapped registers of a device and the memory-mapped on-device memory. On some architectures there are special instructions to access the I/O bus; for example, the in and out processor instructions of the x86. These instructions are available via the fast portal jx.zero.Ports. As other fast portals, these invocations can be inlined by the translator (see Figure 6.6 of Chapter 6).

3.3 Timer

Almost all device drivers need a time source to timeout an unsuccessful operation or to schedule work that must be done at a specific time. This timeing service is provided by a portal of type jx.timer.TimerManager. A device driver expects to find such a TimerManager service in the name service. Figure 9.3 lists the TimerManager interface. It allows to install a handler that is executed once or periodically or unblock a thread once or periodically. A TimerManager implementation typically uses a linked list of handler objects and a specific thread the removes elements from this list and invokes the handler method timer(). To allow the client of the TimerManager to run in a different domain the TimerHandler interface is a portal. If the TimerManager runs in a different domain as its client, the invocation of the handler method is performed as a portal call otherwise as a regular method invocation. In both cases the timer manager thread is blocked until the handler method returns. As this allows a client to block the TimerManager forever by using a handler that does not return (runaway handler) we must remove the handler part of the TimerManager interface and only provide the thread unblocking functionality. The handler functionality must be implemented in a component that is loaded into the client domain. This component implements the linked list of handler objects and uses a thread that is unblocked by the TimerManager. A runaway handler does only affect the time service of its own domain. Other clients of the TimerManager are unaffected. Such an implementation requires a reference to a thread of another domain. The implementation of such references is explained in Chapter 8, Section 2.2.

Figure 9.3: TimerManager interface

The TimerManager is used to schedule an action to be performed at a later time. It can be used to invoke a method at a handler object or to unblock a thread at the specified time.

```
public interface TimerManager extends Portal {
    Timer addMillisTimer(int expires, TimerHandler handler, Object argument);
    Timer addMillisIntervalTimer(int expires, int interval, TimerHandler h, Object a);
    boolean deleteTimer(TimerHandler t);
    int getTimeInMillis();
    int getTimeBaseInMicros();
    int getCurrentTime();
    int getCurrentTimePlusMillis(int milliSeconds);
    void unblockInMillis(CPUState thread, int timeFromNow);
    void unblockInMillisInterval(CPUState thread, int expires, int interval);
}
public interface TimerHandler extends Portal {
    public void timer(Object arg);
}
```

3.4 Garbage collection

Because all device drivers are written in Java and run in a normal JX domain they use garbage collected memory. The garbage collection algorithms currently used in the JX prototype do not allow threads to run concurrently with the garbage collector. If an interrupt occurs during garbage collection of the domain that contains the interrupt handler, interrupt handling is deferred until garbage collection completes. As this increases interrupt response time it must be coped with if the system should support real-time tasks. There are at least two solutions: the device driver domain could use an incremental GC or does not allocate dynamic memory at all. An incremental GC [12] can run concurrently with mutator threads but mutator threads must be suspended if they allocate memory faster than the GC thread can collect. For hard real-time processing it seems more attractive not to allocate any memory after the domain has initialized its data structures. As object allocation is a very common operation in a Java program this seems to be a difficult endeavor but one that can be accomplished very reliably, because it can be detected automatically whether a certain method performs an object allocation before running the method. Thus it is possible to separate methods into *allocating methods* and *non-allocating methods*. To be certain that after a defined point in time—called *time X*—only non-allocating methods are executed, the bytecode verifier must check that starting with a *set of initial methods*, for example all handleInterrupt() methods, only non-allocating methods are invoked. This is possible by using the control-flow analyses of JX's extended bytecode verifier. This control-flow analyses is used to compute the worst-case execution time of interrupt handlers based on an execution time definition for every bytecode instruction. By setting the "execution time" of an instruction to 1 if the instruction allocates memory and 0 if it does not, we can guarantee that a method does not allocate memory if its “execution time” is 0. Not being able to allocate objects restricts the programming in the following ways:

- When a thread is created a new thread control block and a new stack must be allocated on the heap. Therefore the program can not create new threads after time X. If the program needs multiple threads it must create them before time X, block them, and unblock them after time X. The method that is used to continue the execution of this thread must be included in the set of initial methods.
- A portal invocation copies parameters and return values between domains and needs to allocate these objects on the heap. After time X methods are not allowed to invoke methods at portal interfaces if the parameter or return values are object types. Even with this restriction a dynamic allocation may be required if the portal invocation throws an exception. But this is a special case of the next problem.
- Exceptions can be thrown at nearly every position of a Java program. Throwing an exception does *not* mean that any dynamic object allocation must be performed. An exception can be thrown using a preallocated object. But as such an object is unable to capture the actual state of the execution (the stack trace) it is of limited use to debug a program. It can, however, be used to transfer control to the exception handler, which is sufficient in most cases. A special kind of exceptions are the exceptions that are thrown by the runtime system. These exceptions include NullPointerException and ArrayIndexOutOfBoundsException, but also the exceptions that are thrown during a portal call when the (other) domain is terminated or the service disabled. To avoid dynamically allocating these objects the JX kernel can be configured to preallocate them.

Although these restrictions make programming more difficult it is possible to write useful programs if their task is not too complex. A complex program, such as a window manager, should be run in a non-real-time domain and communicate with the real-time domain using fast portals, such as Memory and AtomicVariable, or simple user-defined portals.

4 Robustness

Device drivers in JX are programmed in Java and are installed as service components in a domain. JX aims at only trusting the hardware manufacturer (and not the driver provider) in assuming that the device behaves exactly according to the device specification. When special functionality of the hardware allows bypassing the protection mechanisms of JX, the code for controlling this functionality must also be trusted. This code can not be part of the JX core, because it is device dependent. One example for such code is the busmaster DMA initialization, because a device can be programmed to transfer data to arbitrary main memory locations. Generally speaking there are two main problems: access to memory by the device and blocking interrupts for an indefinite period of time.

Memory access. Most drivers for high-throughput devices will use busmaster DMA to transfer data. These drivers, or at least the part that accesses the DMA hardware, must be trusted. To reduce the amount of critical code, the driver is split into a (simple) trusted part and a (complex) untrusted part (see Figure 9.5). The IDE controller and network card basically use a list of physical memory addresses for busmaster DMA. The code that builds and installs these tables is trusted. We also developed a driver for a frame grabber card based on the Bt848 chip [88]. This chip can execute a program in a special instruction set (RISC code). This program writes captured scanlines into arbitrary memory regions. The memory addresses are part of the RISC program. To allow an untrusted component to download RISC code a verifier for this instruction set would be necessary. For these reasons we decided to make the RISC generator a trusted component.

Interrupts. To guarantee that the handler can not block the system forever, the verifier checks all classes that implement the InterruptHandler interface. It guarantees that the handleInterrupt method does not exceed a certain time limit. To avoid undecidable problems, only a simple code structure is allowed (linear code, loops with constant bound and no write access to the loop variable inside the loop). A handleInterrupt method usually acknowledges the interrupt at the device and unblocks a thread that handles the interrupt asynchronously.

Synchronization. JX does not allow device drivers to disable interrupts outside the interrupt handler. Drivers usually disable interrupts as a cheap way to avoid race conditions with the interrupt handler. Code that runs with interrupts disabled in a UNIX kernel is not allowed to block, as this would result in a deadlock. Using locks also is not an option, because the interrupt handler - running with interrupts disabled - should not block. We use the abstraction of an AtomicVariable (Figure 9.4) to solve these problems. An AtomicVariable contains a value, that can be changed and accessed using set and get methods. Furthermore, it provides a method to atomically compare its value with a parameter and block if the values are equal. Another method atomically sets the value and unblocks a thread. To guarantee atomicity the implementation of AtomicVariable currently disables interrupts on a uniprocessor and uses spinlocks on a multiprocessor. Using AtomicVariables we implemented, for example, a producer/consumer list for the network protocol stack.

Figure 9.4: AtomicVariable interface

An AtomicVariable contains one Object reference. The value of this reference can be accessed using the set() and get() methods. While these methods simply write or read the value, the method atomicUpdateUnblock() sets the value and unblocks a thread atomically with respect to the methods blockIfEqual() and blockIfNotEqual().

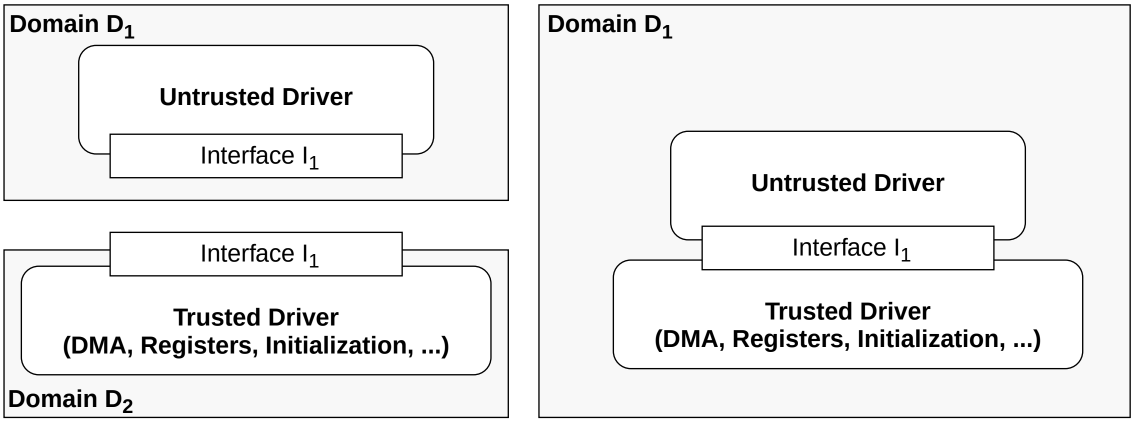
```
package jx.zero;

public interface AtomicVariable extends Portal {
    void set(Object value);
    Object get();
    void atomicUpdateUnblock(Object value, CPUState state);
    void blockIfEqual(Object testValue);
    void blockIfNotEqual(Object testValue);
}
```

5 Device driver framework

The device driver framework is *not* part of the JX core architecture. It is one example for the device management in an operating system that is build on top of JX.

Figure 9.5: Split device driver

To reduce the amount of trusted code the device drivers can be split in two halves. a trusted part and an untrusted part. The trusted part accesses critical components, such as the DMA engine, certain registers, and initializes the device. The untrusted part contains the code to manage the device, such as scheduling I/O requests. Both parts can be run in the same (b) or in different domains (a). If they run in the same domain the untrusted part can attempt to circumvent the integrity of the trusted part by exhausting the domain's resources. The trusted part must be programmed to cope with such a situation. It should, for example, assume that every instruction can be its last one.



(a) multi-domain configuration

(b) single-domain configuration

5.1 Device identification

A device driver must be able to detect a supported device. Every device driver implements an interface jx.devices.DeviceFinder that contains a single method find(). This method returns an array of supported devices. The interface jx.devices.Device (see Figure 9.6) is the supertype of all devices. It contains methods open() and close() to activate and deactivate the device. The getSupportedConfigurations() method returns a device-specific array of supported configurations. A fixed configuration must be passed to the open() method when activating the device.

Figure 9.6: Device and DeviceFinder interface

```
package jx.devices;

public interface Device {
    DeviceConfigurationTemplate[] getSupportedConfigurations ();
    void open(DeviceConfiguration conf);
    void close();
}

public interface DeviceFinder {
    Device[] find(String[] args);
}
```

5.2 Example classes of device drivers

Devices are categorized into different classes, such as network devices, disk devices, and framebuffer devices. There is an interface that must be implemented by a device driver. The communication between the “application” (which may be a file system or a network protocol) and the device driver takes place via this interface. The interface can of course be implemented by a class that (i) emulates a device or a class that (ii) intercepts requests to a real device to provide additional functionality. An example for the first category is a disk driver that sends “disk blocks” over the network instead of using a local disk. Examples for the second category are a network driver that logs all packets or filters packets, or a disk driver that only provides read-only access to a disk by throwing an exception in the method(s) that write to the disk.

5.2.1 The network driver

The network device interface is listed in Figure 9.7. The method setReceiveMode() configures the device to either receive packets that are addressed to the device (individual), all packets on a broadcast medium (promiscuous), or multicast packets. The transmit() method sends the contents of the memory object. The registerNonBlockingConsumer() method is used to install a packet receiver.

Figure 9.7: Network device interface

```
package jx.devices.net;

public interface NetworkDevice extends Device {
    static final int RECEIVE_MODE_INDIVIDUAL = 1;
    static final int RECEIVE_MODE_PROMISCOUS = 2;
    static final int RECEIVE_MODE_MULTICAST  = 3;

    void setReceiveMode(int mode);
    Memory transmit(Memory buf);
    boolean registerNonBlockingConsumer(NonBlockingMemoryConsumer consumer);
    byte[] getMACAddress();
    int getMTU();
}

public interface NonBlockingMemoryConsumer {
    Memory processMemory(Memory data, int offset, int size);
}
```

5.2.2 The disk driver

A disk driver provides the functionality of persistently storing data and is accessed by using the interface jx.bio.BlockIO (see Figure 9.8). The method getSectorSize() returns the size of a sector in bytes; getCapacity() returns the number of sectors of this device. The methods readSectors() and writeSectors() perform the actual I/O operation. These methods operate in a synchronous mode, i.e. even if the device could write the data asynchronously using DMA operations the writeSectors() method blocks until all data is written to the device.

Figure 9.8: BlockIO device interface

```
package jx.devices.storage;

public interface BlockIO extends Device {
    int getSectorSize();
    int getCapacity();
    void readSectors(int startSector, int numberOfSectors, Memory buf);
    void writeSectors(int startSector, int numberOfSectors, Memory buf);
}
```

5.2.3 The framebuffer driver

A frame buffer device driver implements the interface shown in Figure 9.9. The methods open(), close(), and getSupportedConfigurations() are inherited from the Device interface. The framebuffer interface allows to use devices that need a trigger to make the update visible to the user. An example for such a device is a remote display that needs to know when the current contents of the (virtual) display must be transferred to the host that provides the physical display.

Figure 9.9: Framebuffer device interface

```
package jx.devices.fb;

public interface FramebufferDevice extends Device {
    int getWidth();
    int getHeight();
    int setMode (int width, int height, ColorSpace colorSpace);
    void startFrameBufferUpdate();
    void endFrameBufferUpdate();
    void startUpdate();
    void endUpdate();
    DeviceMemory getFrameBuffer();
    int getFrameBufferOffset();
    int getBytesPerLine();
    ColorSpace getColorSpace();
    int drawLine(PixelRect draw, PixelRect clip, PixelColor color, DrawingMode mode);
    int fillRect(PixelRect rect[], int count, PixelColor color, DrawingMode mode);
    int bitBlt(PixelRect oldPos[], PixelRect newPos[], int count);
}
```

6 Performance evaluation

One of the most important performance characteristics related to device drivers are interrupt latency and interrupt response time. *Interrupt latency* is defined as the time between the device asserts the interrupt and the CPU is ready to deliver it (time A in Figure 9.7). This gap appears because the currently executed instruction can not immediately preempted. Some CPUs contain instructions that need long time to complete. *Interrupt response time* is defined as the time between the creation of the interrupt by the device and the time the interrupt handler thread starts running (time (A+B+C) in

Figure 9.10). Devices with small buffers, such as the serial line device of a PC, do not tolerate large delays between the time they raise an interrupt and get service. Otherwise they loose data.

Figure 9.10: Interrupt latency in JX

The time periods in the diagram have the following meaning: *A* is the interrupt latency, *A+B+C* is the interrupt response time, in the interval B interrupts are disabled due to other system activity (for example other device drivers) and in the interval C+D interrupts are disabled due to the currently executing device driver. The length of interval E, the latency of the second-level handler, is solely determined by the scheduler.

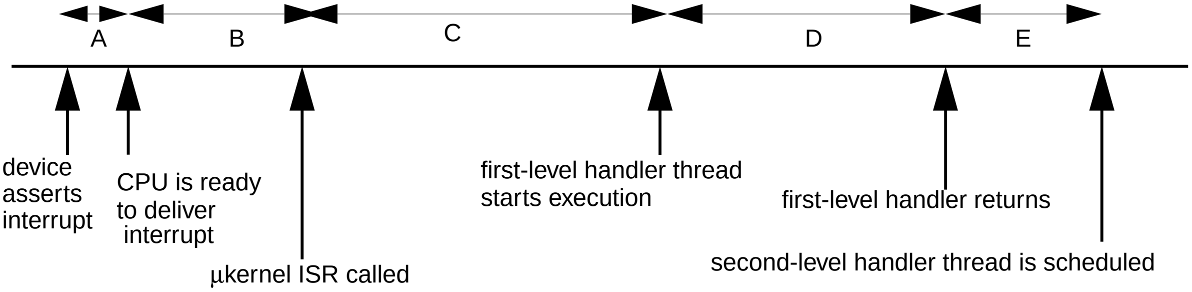
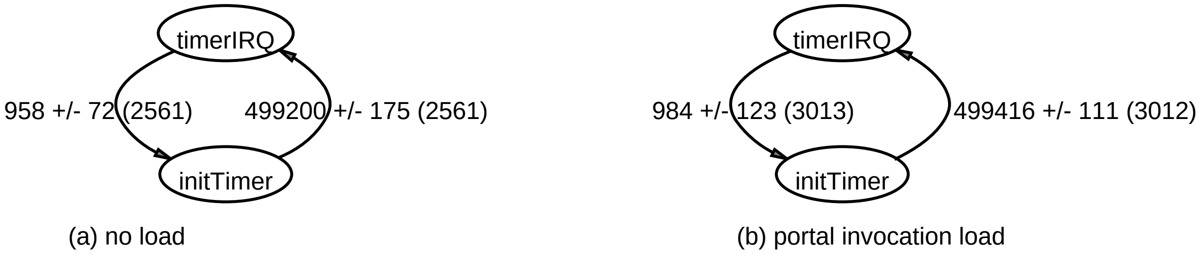

A
B
C
D
E
device asserts interrupt
CPU is ready to deliver interrupt
μkernel ISR called
first-level handler thread starts execution
first-level handler returns
second-level handler thread is scheduled

Figure 9.11: Measured interrupt response time

The time is given in cycles (1 cycle = 2 ns) and is corrected by the time required to log an event from the Java-level interrupt handler (96 cycles). The time is given as mean with standard deviation. The number of transitions is denoted in parenthesis.



The interrupt response time is determined by the system architecture, i.e. by the maximal duration the system runs with interrupts disabled. As the OS is written at the Java level and is not allowed to disable interrupts interesting operations are kernel operations that disable interrupts. One of these operations is the portal invocation.

We use the following experiments to measure interrupt response time. We use the programmable interval timer (PIT) as the interrupt source and setup the PIT in one-shot mode and program the timer to fire in 1 millisecond. At the beginning of the interrupt handler we record the time by generating an event (*timerIRQ*). Then we re-arm the timer and generate another event (*initTimer*). Figure 9.11 shows the event timing diagram once without additional load (a) and with a concurrently running endless loop that performs portal invocations (b). The transition between timerIRQ and initTimer gives the time required to re-arm the PIT. Although re-arming the PIT requires only two outb instructions it is a very slow operation (1.92 µs). The transition between initTimer and timerIRQ is the time between leaving the interrupt handler and entering it again. Without additional load this time is 998.4 µs. Although we programmed the PIT to fire in 1000 µs it either is not very accurate or starts to count within the 1.92 µs re-arming period[1]. If we assume the latter case the response time of the unloaded system is between 320 ns and 0 ns. In the loaded system the time between initTimer and timerIRQ is 998.8 µs. As this is an increase of 400 ns compared to the unloaded case we conclude that the mean response time is between 720 ns and 0 ns. This increase is caused by the atomic code sequences during a portal call.These sequences are marked with (begin atomic ... end atomic) in the flow diagram of the portal invocation mechanism in Figure 5.9 and Figure 5.10 of Chapter 5.

7 Lessons learned

We learned several lessons from the implementation of the IDE, the NIC, and Bt848 drivers:

- The code that uses the DMA engine depends on the code that initializes the device. As the initialization code makes up a large part of the driver the amount of code that can be removed from the trusted is rather small.
- We learned that device drivers profit very much from object orientation and the abstraction that is possible with interfaces. Even the implementation of a device driver can be structured in an object-oriented way. Due to the limited number of device drivers we have no practical experiences of code reuse between different drivers but we expect that the object-oriented structure will promote the reuse of driver parts.
- The device driver framework contains many classes in many components. To limit the amount of trusted code the trusted low-level part of the driver must not depend on the device driver framework.

8 Summary

We demonstrated that it is possible to write device drivers completely in a type-safe high-level language. Parts of the driver must be trusted, because they access the hardware in a way that is unpractical to make secure. Problematic operations include the setup of DMA data structures and interrupt acknowledgement. Furthermore, because for economic and performance reasons the hardware usually performs few validity and safety checks and trusts the driver to access the device according to the device specification. It is necessary to include a thin layer between the hardware and the untrusted part of the driver. This layer mediates access and performs the necessary checks.

All microkernel-based systems, where drivers are moved into untrusted address spaces run into the same problems, but they have much weaker means to cope with these problems. Using an MMU does not help, because busmaster DMA accesses physical RAM without consulting page ta-

[1] After we discovered this problem we learned about a more reliable technique of measuring interrupt response time. The performance counter registers of the Pentium III can be used to count events and raise an interrupt when the counter overflows. As event source the CPU cycle can be used. Compared to the PIT an access to the performance counters is fast.

bles. In JX these problems are addressed by using type safety, special checks of the verifier, and split drivers.

CHAPTER 10 *Security Architecture*

Security is an important requirement for a multi-user timesharing system. It is even more important if the system is able to execute mobile untrusted code. This chapter describes the security architecture of the JX system. It analyses the security implications of the system design and compares it with related architectures.

1 Security problems in traditional systems

1.1 Operating systems

There are two categories of errors that cause the easy vulnerability of current systems. The first are implementation errors, such as buffer overflows, dangling pointers, and memory leaks, which are caused by the prevalent use of unsafe languages in current systems. This becomes dangerous when an OS relies on a large number of trusted programs. From the top ten CERT notes (as of January 2002) with highest vulnerability potential six are buffer overflows [28], [29], [30], [31], [32], [33], two relate to errors checking user supplied strings that contain commands thus allowing the user to run arbitrary commands with root privilege [34], [35], one executes commands in emails [36], and one is an integer overflow [37]. The six buffer overflow vulnerabilities could have been avoided by using techniques described by Cowan et al. [43]. However, not all overflow attacks can be detected and the authors recommend the use of a type-safe language.

The second category of errors—the architectural errors— is more difficult to tackle. The three CERT notes related to the execution of commands in strings and emails are critical because the vulnerable systems violate the principle of least-privilege [152]. Thus, in current mainstream systems it is not the question whether the proper security policy is used, but whether security can be enforced at all [120]. Violations of the principle of *least-privilege*, an uncontrolled cumulation of functionality, many implementation errors, complex system architectures, and poorly understood interrelations between system components make current systems very vulnerable. This is a problem that affects all applications, because applications are built on top of an operating system and can be only as secure as its trusted programs and the underlying kernel.

Microkernels are well suited as the foundation of a secure system. On the other hand we believe that the security of a microkernel architecture will only improve over a monolithic architecture if a type-safe language is used for the trusted servers. Java allows developing applications using a modern object-oriented style, emphasizing abstraction and reusability. On the other hand many security problems have been detected in Java systems in the past.

1.2 Java Security

Java security is based on the concept of a sandbox, which relies on the type-safety of the executed code. Untrusted but verified code can run in the sandbox and can not leave the sandbox to do any harm. Every sandbox must have a kind of exit or hole, otherwise the code running in the sandbox can not communicate results or interact with the environment in a suitable way. These holes must be clearly defined and thoroughly controlled. The holes of the Java sandbox are the native methods. To control these holes, the Java runtime first controls which classes are allowed to load a library that contains native code. These classes must be trusted to guard access to their native methods. The native methods of these classes should be non-public and the public non-native methods are expected to invoke the SecurityManager before invoking a native method. The SecurityManager inspects the runtime call stack and checks whether the caller of the trusted method is trusted.

Java version 1 distinguishes between trusted system classes, which were loaded using the JVM internal class loading mechanism, and untrusted classes, which were loaded using a class loader external to the JVM. Implementations of the SecurityManager can check whether the classes on the call stack—the callers of the method—are trusted or untrusted classes. When the caller was a system class the operation usually is allowed otherwise the SecurityManager decides, depending on the kind of operation and its parameters, whether the untrusted class is allowed to invoke the operation[1].

Java version 2 also relies on stack inspection but can define more flexible security policies by describing the permissions of classes of a certain origin in external files.

To sum up, Java security relies on the following requirements:

(1) Code is kept in a sandbox by using an intermediate instruction set. Programs are verified to be type-safe.

(2) The package-specific and/or class-specific access modifiers must be used to restrict access to the holes of this sandbox: the native methods of trusted classes. As long as the demarcation line between Java code and native code is not crossed, the Java code can do no harm.

(3) The publicly accessible methods of the trusted classes must invoke the SecurityManager to check whether an operation that would leave the sandbox is allowed.

The SecurityManager is similar to a reference monitor, but has a severe shortcoming: it is not automatically invoked. A trusted class must explicitly invoke the SecurityManager to protect itself. The mere number of native methods makes it difficult to assure this. We counted 1312 native methods in Sun's JRE 1.3.1_02 for Linux, which are 2.9 percent of all methods. From these native methods 34 percent are public and even as much as 16 percent are public static methods in a public class. This means that the method can be invoked directly from everywhere without the SecurityManager having a chance to intercept the call. Two of these methods are java.lang.System.currentTimeMillis() and java.lang.Thread.sleep() which provides an interesting opportunity to create a covert timing channel. The fact that covert channels are not exploited can be attributed to the existence of many overt channels. Public, non-final, static variables in public system classes are only one example (we counted 31 of these fields in Sun's JRE).

[1] The real implementation uses the abstraction of *classloader-depth*, which is the number of stack frames between the current stack frame and the first stack frame connected to a class that was loaded using a classloader.

A further problem is that the stack inspection mechanism only is concerned with access control. It completely ignores the availability aspect of security. This lack was addressed in JRes [45]. By rewriting bytecodes, JRes creates a layer on top of the JVM. In our opinion, this is the wrong layer for resource control, because resources that are only visible inside the JVM can only be accounted inside the JVM. Examples are CPU time and memory used for the garbage collector (GC) or just-in-time compiler or memory used for stack frames. Furthermore, rewriting bytecodes is a performance overhead in itself and it creates slower programs. Often, Java is perceived as inherently insecure due to the complexity of its class libraries and runtime system. JX avoids this problem by not trusting the JDK class library.

2 Requirements for a security architecture

In this section we review requirements for a security architecture.

2.1 Saltzer & Schroeder

Saltzer and Schroeder [152] collected a comprehensive list of requirements for a secure system:

Economy of mechanism. The security mechanisms must be simple and small to be subject to verification.

Fail-safe defaults. Access should be rejected if not explicitly granted.

Complete mediation. All accesses must be checked.

Open design. The system design must be published.

Separation of privilege. Do not concentrate all privileges at one principal.

Least privilege. A system component should be granted the minimal privileges necessary to fulfil its task.

Least common mechanism. No unnecessary sharing between system components should be allowed.

Psychological acceptability. When the security system communicates with the human user it must respect the mental model of the user and must not annoy the user with too many questions.

2.2 Confinement

A lot of work on security has been conducted in the context of military security. Although there are more recent standards [192] the Orange Book [49] contains abstract principles for the design of secure systems. Systems are classified according to the security they are able to enforce. Requirements for higher levels include secrecy and modularity:

Secrecy. Systems rated B1 must support sensitivity labeling of all objects and clearance labeling of all subjects. Systems that aim at a B2 or higher rating must analyze covert channels.

Modularity. The trusted computing base must be as small as possible, because it must be verified to obtain high assurance. It must not only be small in size, but the whole system architecture must be simple and clean.

2.3 Integrity

While most of the work done in the context of military security focused on secrecy, the security model developed by Clark & Wilson [41] assumes that in commercial systems integrity of data is equally or more important than secrecy. This model allows data access and modification only via *transformation procedures* and requires an integrity monitoring using *integrity verification procedures*.

2.4 Separation of policy and enforcement

Separation of policy from mechanism is a software engineering principle that leads to a modular system structure with evolvable and replaceable policies. Several security architectures follow this principle. The DTOS system [130] and its successor Flask [165] concentrated on policy flexibility in a microkernel-based OS. In some systems, security decisions are spread over the whole system, which makes it difficult to understand what kind of security policy the system as a whole actually enforces [112]. Centralizing the policy facilitates adaptations to new security requirements and enhances manageability. The policy can be changed without changes to the fundamental security and system architecture and without changes to the object servers. Furthermore, a central security manager is a requirement for the enforcement of complex security policies that are more than access decisions. The policy could, for example, state that all email must be encrypted, an alert must be activated for three unsuccessful login attempts, or that users of a certain clearance must store all their data in encrypted form. These policies can only be enforced by a security manager that has complete control over the system.

2.5 Suitable programming language

The importance of the programming language for a secure system was recognized in early systems, such as KSOS [66]. A study of the Secure Computing Corporation evaluates five microkernel-based operating systems with respect to security [155]. This study contains a list ([155] pp. 24) of properties of a programming language that affect assurability of code. To improve assurability a programming language should allow a high abstraction level, support strong data typing, modularization, and prohibit pointer manipulation.

2.6 Performance

Although current advances in processor performance will keep a moderate performance degradation below the perceptional threshold of a typical user, a slowdown of central mechanisms may have a dramatic effect for the performance of the whole system.

3 Capabilities

Conceptually a capability is a reference to an object together with a list of allowed operation. The capability concept was introduced by Dennis and Van Horn in the 1960s [48]. Capability based protection experienced a renewed attention in the research community with the introduction of the Java language in 1995.

Several operating systems are based on capabilities and use three different implementation techniques: partitioned memory, tagged memory [62], and password capabilities. Early capability systems used a tagged memory architecture (Burroughs 5000 [140], Symbolics Lisp Machine [131]), or partitioned memory in data-containing and capability-containing segments (KeyKOS [68] and EROS [157]). All these implementations relied on specific hardware features. To become hardware-independent, password capabilities [6] have been invented and are used in several systems (Mungi [89], Opal [39], Amoeba [175]). In Amoeba, for example, capabilities are 128bit values consisting of a port number to identify the server and an object id to identify the object, an 48 bit random number and 8 bit access rights. The access right bits are used to encode which operation of the object can be invoked, which means that there is a maximum of 8 different classes of operations per server. The check field contains the result of a one-way function that uses the access bits as one of its input parameters. There is no definite agreement on how secure password capabilities really are. There is a non-zero chance that passwords can be guessed. Security relies on the strength of the cryptographic function. Password capabilities can be transferred over an external channel, for example, on a desktop system the user reads the capability in one window and types it in another window. Furthermore, using cryptographic methods adds an overhead when using password capabilities.

Type-safe instruction sets, such as the Java intermediate bytecode, are a fourth way of implementing capabilities. The main advantages of this technique are that it is hardware-independent, capability verification is performed at load time, access rights are encoded in the capability type and not stored as a bitmap in the capability, and capabilities can not be transferred over uncontrolled channels.

4 JX as a capability system

Portals are capabilities [48]. A domain can only access other domains when it possesses a portal to a service of the other domain. The operations that can be performed with the portal are listed in the interface.

Although the capability concept is very flexible and solves many security problems, such as the confused deputy [82], in a very natural way, it has well known limitations. The major concern is that a capability can be used to obtain other capabilities, which makes it difficult, if not impossible, to enforce the *-property [24]. To decide whether a given system is able to enforce the *-property, Kain and Landwehr [101] developed a taxonomy for capability systems. The Kain&Landwehr paper assumes a segmented memory architecture, where capabilities are used to control access to segments of memory. We try to translate their questions and answers to an architecture where capabilities control access to services and unforgeable capabilities are realized by type safety.
What happens

(1) ... when a portal is created?

(2a) ... to an executing portal invocation when the security attributes of the service are modified?

(2b) ... to a portal when the security attributes of the service are modified?

(3) ... when a portal is copied?

(4) ... when a method at a portal is called?

(5) ... when a service is invoked via a portal?

For the JX system as described up to now the questions can be answered as follows (the letters in parenthesis denote the answer in the Kain&Landwehr taxonomy):

(1) In JX a capability (portal) can only be created by the microkernel. It is created when an object that is marked to act as a service is passed to another domain. Although there are no explicit checks when a capability is created or when an object is accessed, the type safety of the system disallows access to arbitrary objects and the creation of arbitrary capabilities. (b)

(2a) The only change of security attributes is service deactivation and in such a case the service execution is aborted with an exception. (no answer fits)

(2b) When the service is deactivated, an access via a portal will throw an exception. (no answer fits)

(3) A portal is passed without modification between domains. (a)

(4) No access checks are performed. (a)

(5) A portal can be revoked (by deactivating the service). When the portal is used to access the service, it is checked whether the service is still available. (b)

JX as described up to now can be classified as *b??aab*. Such a system can not enforce the *-property; thus an additional mechanism is needed: a reference monitor that is able to check all portal invocations and the transfer of portals between domains.

5 The reference monitor

A reference monitor must be tamper-proof, mediate all accesses, and be small enough to be verified. A reference monitor for JX must at least control incoming and outgoing portal calls. There are two alternatives for the implementation of such a reference monitor:

Proxy. Initially a domain has access only to the naming portal that is passed during domain creation. To obtain other portals the name service is used. The parent domain can implement this name service to not return the registered portal but a proxy portal which implements the same interface. This proxy can then invoke a central reference monitor before invoking the original portal.

Microkernel. The portal invocation mechanism inside the microkernel invokes a reference monitor on each portal call and passes sender principal, receiver principal, and call parameters to the reference monitor.

These two implementation alternatives have the following advantages and drawbacks. The proxy solution needs no modification of the microkernel and thus avoids the danger of introducing new bugs. As long as no reference monitoring is needed, the proxy solution does not cause any additional cost. The microkernel solution must check in every portal invocation sequence whether a reference monitor is attached to the domain. Because the domain control block, which contains this information, is already in the cache during the portal invocation, this check is nearly for free. On the other hand, the proxy solution requires the name service to create a proxy for each registered portal. During a method invocation at such a portal the whole parameter graph must be traversed and when a portal is found it must be replaced by a proxy portal.

We rejected the proxy approach, because it requires a rather complex implementation and it is difficult to assure that each portal is “encapsulated” in a proxy portal.

We modified the microkernel to invoke the reference monitor when a portal call invokes a service of the monitored domain (inbound) and when a service of another domain is invoked via a portal (outbound). The internal activity of a domain is not controlled. The same reference monitor must control inbound and outbound calls of a domain, but different domains can use different monitors. A monitor is attached to a domain when the domain is created. When a domain creates a new domain, the reference monitor of the creating domain is asked to attach a reference monitor to the created domain. Usually, it will attach itself to the new domain but it can - depending on the security policy - attach another reference monitor or no reference monitor at all.

It must be guaranteed, that while the access check is performed, the state to be checked can only be modified by the reference monitor. When this state only includes the parameters of the call, these parameters could be copied to a location that is only accessible by the reference monitor. When the state includes other properties of the involved domains, the activity of these domains must be suspended. For these reasons the access check is performed in a separate domain, not in the caller or callee domain.

The list of parameters is accessed using an array of VMObject portals. VMObject is a portal which allows access to an object of another domain. The reference monitor furthermore gets the Domain portal of the caller domain and the callee domain. To accelerate the operation of the reference monitor, the Domain portal is a portal which can be inlined by the translator. On an x86 it takes only two machine instructions to get the domain ID given the Domain portal.

The main problem is to obtain a consistent view of the system during the check. One way is to freeze the whole system by disabling interrupts during the check. This would work only on a uniprocessor, would interfere with scheduling, and allow a denial-of-service attack. Therefore, our current implementation copies all parameters from the client domain to the server domain up to a certain per-call quota. These objects are not immediately available to the server domain, but are first checked by the security manager. When the security manager approves the call the normal portal invocation sequence proceeds.

The addition of the reference monitor places JX in another class in the Kain&Landwehr taxonomy. The actual classification depends on the implementation of the security manager.

5.1 Making an access decision

Spencer et al. [165] argue that basing an access decision only on the intercepted IPC between servers forces the security server to duplicate part of the object server’s state or functionality. We found two examples of this problem. In UNIX-like systems access to files in a file system is checked when the file is opened. The security manager must analyze the file name to make the access decision, which is difficult without knowing details of the file system implementation and without information that is only accessible to the file system implementation. The problem is even more obvious in a database system that is accessed using SQL statements. To make an access decision the reference monitor must parse the SQL statement. This is inefficient and duplicates functionality of the database server.

There are three solutions for these problems:

(1) The reference monitor lets the server proceed and only checks the returned portal (the file portal).

(2) The server explicitly communicates with the security manager when an access decision is needed.

(3) Design a different interface that simplifies the access decision.

Approach (1) may be too late, especially in cases where the call modified the state of the server.

Approach (2) is the most flexible solution. It is used in Flask with the intention of separating security policy and enforcement mechanism [165]. The main problem of this solution is, that it pollutes the server implementation with calls to the security manager. The Flask security architecture was implemented in SELinux [121]. In SELinux, the list of permissions for file and directory objects have a nearly one-to-one correspondence to an interface one would use for these objects. This makes approach (3) the most promising approach. Our two example problems would be solved by parsing the path in the client domain. In an analogous manner the SQL parser is located in the client domain and a parsed representation is passed to the server domain and intercepted by the security manager. This has the additional advantage of moving code to an untrusted client, eliminating the need to verify this code. Section 14 gives further details about the design of the file server interface.

5.2 Controlling portal propagation

In [111] Lampson envisioned a system in which the client can determine all communication channels that are available to the server *before* talking to the server. We can do this by enumerating all portals that are owned by a domain. As we can not enforce a domain to be *memoryless* [111], we must also control the future communication behavior of a domain to guarantee the confinement of information passed to the domain.

Several alternative implementations can be used to enumerate the portals of a domain:

(1) A simple approach is to scan the complete heap of the domain for portal objects. Besides the expensive scanning operation, the security manager can not be sure, that the domain will not obtain portals in the future.

(2) An outbound intercepter can be installed to observe all outgoing communication of the domain. Thus a domain is allowed to posses a critical portal but the reference monitor can reject it’s use. The performance disadvantage is that the complete communication must be checked, even if the security policy allows unrestricted communication with a subset of all domains.

(3) The security manager checks all portals transferred to a domain. This can be achieved by installing an inbound interceptor which inspects all data given to a domain and traverses the parameter object graph to find portals. This could be an expensive operation if a parameter object is the root of a large object graph. During copying of the parameters to the destination domain, the microkernel already traverses the whole object graph. Therefore it is easy to find portals during this copying operation. The kernel can then inform the security manager, that there is a portal passed to the domain (method createPortal()). The return value of createPortal() decides whether the portal can be created or not. The security manager must also be informed if the garbage collector destroys a portal (destroyPortal()). This way reference monitor can keep track of what portals a domain actually possesses.

Confinement can now be guaranteed with two mechanisms that can be used separately or in combination: (i) the control of portal communication and (ii) the control of portal propagation.

Figure 10.1 shows the complete reference monitor interface. Figure 10.2 shows the information that is available to the reference monitor.

Figure 10.1: Reference monitor interface

```
public interface DomainBorder {
      boolean outBound(InterceptInfo info);
      boolean inBound(InterceptInfo info);
      boolean createPortal(PortalInfo info);
      void destroyPortal(PortalInfo info);
}
```

Figure 10.2: Information interfaces

```
public interface InterceptInfo extends Portal {
    Domain getSourceDomain();
    Domain getTargetDomain();
    VMMethod getMethod();
    VMObject getServiceObject();
    VMObject[] getParameters();
}

public interface PortalInfo extends Portal {
    Domain getTargetDomain();
    int getServiceID();
}
```

6 Principals

A security policy uses the concept of a *principal* [48] to name the subject that is responsible for an operation. The principal concept is not known to the JX microkernel. It is an abstraction that is implemented by the security system outside the microkernel, while the microkernel only operates with domains. Mapping a domain ID to a principal is the responsibility of the security manager. We implemented a security manager which uses a hash table to map the domain ID to the principal object. We first considered an implementation where the microkernel supports the attachment of a principal object to a domain. The biggest problem of such a support would be the placement of the principal object. Should the object live in the domain it is attached to or in the security manager domain? Both approaches have severe problems. As the security manager must access the object it should be placed in the security manager's heap. But this creates domain interdependencies and the independence of heap management and garbage collection, which is an important property of the JX architecture, would be lost. Thus, a numerical principal ID seemed to be the only solution. But having a principal ID has no advantages over having a domain ID, so finally we concluded that the microkernel should not care about principals at all.

The security manager maps the unique domain ID to a principal object. Once the principal is known, the security manager can use several policies for the access decision, for example based on a simple identity or based on roles [64].

To service a portal call the server thread may itself invoke portals into other domains. To avoid several problems (trojan horse, confused deputy [82]) the server may want to downgrade the rights of these invocations to the rights of the original client. The most elegant solution of these problems is a pure capability architecture. In the JX architecture this would mean that the server uses only portals that were obtained from that particular client. This requirement is difficult to assure in a multi-threaded server domain that processes requests from different clients at the same time. Because the server threads use the same heap, a portal may leak from one server thread to another. A better solution is to allow the reference monitor to downgrade the rights of a call. To allow the reference monitor to enforce downgrading rights to the rights of the invoker, each service thread (a thread that processes a portal call) has the domain ID of the original client attached to it. This information is passed during each portal invocation. The reference monitor has access to this information and can base the access decision on the principal of the original domain, instead of the principal of the immediate client.

7 Revocation

7.1 Revocation of portals

When the security policy is changed, some capabilities must be revoked. In order to support revocation, each access to a capability must be checked. When the capability was revoked, the invocation must be rejected. It is more complicated to cope with invocations that are executing when the capability is revoked. There are two ways to handle this problem:

(i) the invocation is immediately terminated. This could leave server data structures in an inconsistent state. Even when the invocation is terminated by throwing an exception in the server, all code must be protected against inconsistencies by a catch/finally.construct.

(ii) the revocation request is blocked until the invocation returns. This could potentially lead to a indefinite delay of the invocation and policy change. To avoid such a situation, the server should check if the capability was revoked and then finish the call as fast as possible.

For performance reasons, no access control for each access to a memory object can be done, although it would be possible that the microkernel calls the reference monitor during a fast portal method invocation.

7.2 Revocation of fast portals and memory objects

As described in Chapter 5 there is a special kind of portals, called *fast portals*. Fast portals can only be created by DomainZero. They are executed in the context of the caller. The semantics of a fast portal is known to the system and it's methods can be inlined by the translator. An example for a fast portal is the Memory portal. We solved the confinement problems of capabilities by introducing a reference monitor that is invoked when a portal is used. Although a reference monitor could be used for fast portals, this is not practical for performance reasons. Therefore memory portals support revocation. When the reference monitor detects that a portal is passed between two domains (createPortal()) it could revoke the access right to the memory object for the source domain or reject passing of the memory portal.

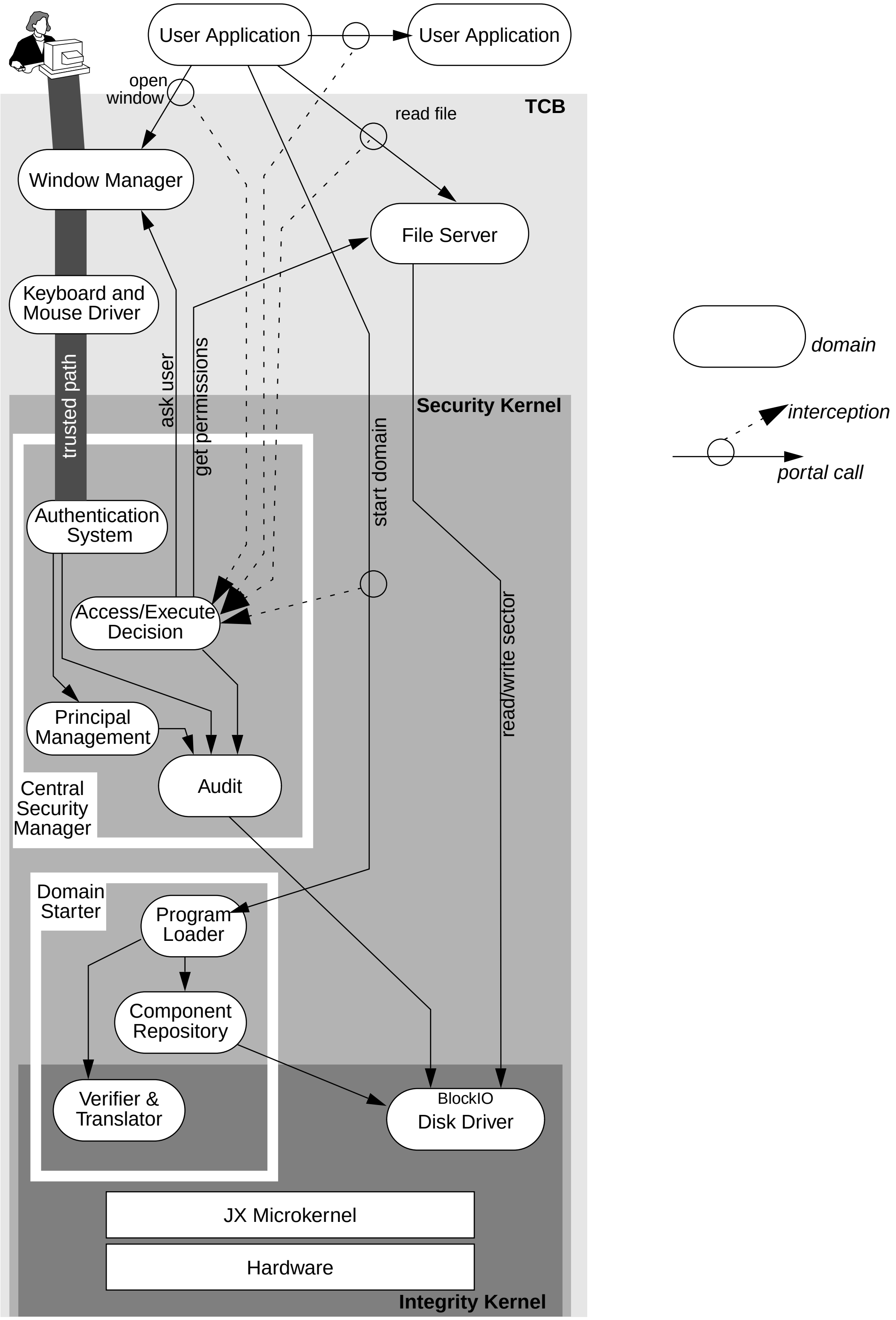
8 Structure of the Trusted Computing Base

Figure 10.3 shows the structure of the Trusted Computing Base (TCB). In the TCB we include all system components that the user trusts to perform a certain operation correctly. The central part of the system is the *integrity kernel*. Compromising the integrity kernel allows an intruder to crash the whole system. Built on the integrity kernel is the *security kernel*. The security kernel represents the minimal TCB. In a typical system configuration the TCB will include the window manager and the file system. Users will trust the file system to store their data reliably. Compromising the security kernel or the rest of the TCB leads to security breaches, such as disclosure of confidential data or unauthorized modification of data, but not to an immediate system crash. It may lead to a system crash when a compromised security kernel allows access to the integrity kernel. This design is reminiscent of the protection rings of Multics.

JX is a component-based system. All components specify which other components they depend on. Unnecessary functionality can easily be removed with a few configuration changes. In a server system the window manager may not be part of the TCB, while in a thin client system the file system may not be needed. A user may even decide not to trust the file system and store the data in an own data base.

It is important that there are no dependencies between the inner kernels and the outer ones. The security manager, for example, must not store its configuration in the file system.

Figure 10.3: Typical TCB structure

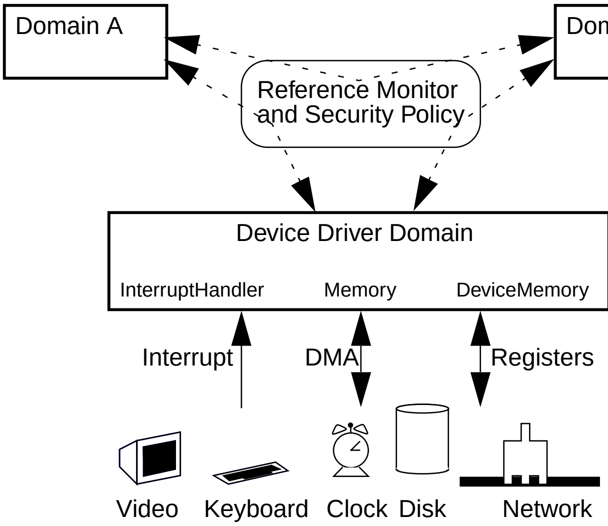


9 Device drivers

There is little work in the security literature which describes how to cope with device drivers. Most systems ignore the problem by placing complete trust in device drivers. We think that such an approach is inappropriate. An investigation of a recent Linux kernel [40] identified device drivers as that part of the kernel that contains most of the bugs. Drivers have an error rate that is seven times higher than the error rate in the rest of the Linux kernel. This problem is multiplied by the fact that device drivers are by far the largest part of the kernel (see Table 10.1). Microsoft has also recognized this problem and bundles a “Driver Verifier” [129] with Windows 2000 which helps to find bugs in device drivers. We think, that a better way is to design an interface that makes these bugs impossible in the first place.

Many of the bugs that are investigated in [40] are not possible in JX, because of the use of a type-safe language and several restrictions in the interface that is available to device drivers. This interface does not allow to arbitrarily disable interrupts. Code that runs with interrupts disabled - the interrupt handler - is checked by an verifier for a upper runtime bound. The busmaster DMA engine of a device can be used to circumvent memory protection. In [75] we advocated to split drivers in an untrusted and a trusted part (which, for example, contains the DMA initialization code). In practice this proved more difficult than expected, because of the tight integration of the DMA engine in most devices. Therefore, the current JX drivers are not split and several device drivers must be part of the integrity kernel (see Figure 10.3).

Figure 10.4: Device driver information flow



Besides being a source of instability, drivers can leak information. The communication between domains is controlled by the reference monitor, but the communication between a device driver and the hardware, using DMA or registers, can not be controlled (see Figure 10.4). Device drivers are able to (persistently) store information at a device or read information from a device.

In an MLS system data that comes from a multi-level device must be labeled as early as possible. Components that work with multi-level data must be verified, components that only operate at a single level not. Using different domains for device drivers for networks of different classification (e.g., intranet/internet or secure/unclassified) makes it possible to label information as early as possible. Furthermore labeling is done automatically by labeling the driver domain appropriately. If possible, network data should be labeled at the packet level, disk data at the block level. This may not be possible when the system uses existing communication protocols, like TCP, to transfer multi-level data.

10 Tamper-resistant auditing

Auditing is a means of defense-in-depth. The system must assure that all security relevant events are persistently stored on disk and cannot be modified. To be certain that the audit trail is tamper-proof we use a separate disk and write this disk in an append only mode. We do not use a file system at all but write the messages unbuffered to consecutive disk sectors. We do not use any buffering and the audit call only returns when the block was written to disk. Writing at consecutive disk sectors avoids long distance head movements and gives a rate of 630 audit messages per second. Writing one audit message needs 1582 µseconds[1]. Given that a file access which can be satisfied from the buffer cache is in the tenth of µseconds auditing each file access adds considerable overhead. The size of a typical audit message is between 35 and 40 bytes. The disk is used as a ring buffer: when the last sector is reached we wrap to the first one and overwrite old logs. This avoids a problem often encountered when logging to a file system: when the file system is full logs get lost. Usually, the most recent logs are the most valuable. With the above mentioned message rate of 630 messages/second and a message size of 40 bytes we have a time window of 110 hours using a 10 GB disk. Under normal operation the time window is much larger, because the message rate is well below its maximum.

11 Trusted path

According to the Orange Book [49] a trusted path is the path from the user to the TCB. Depending on the user interface the TCB must include the window manager or the console driver.

Recent literature generalizes the notion of a trusted path to any communication mechanism within the system. To trust a communication path it is essential to identify the communication partner and provide a communication channel that can not be overheard or modified. Portal communication is such a mechanism.

Usually, the reference monitor limits communication according to a certain security policy. This mechanism works automatically and is transparent to domains. But it is even possible for a domain to explicitly consider portal communication as being performed on a trusted path, because the target domain of a portal can be obtained and this identity can not be spoofed.

[1] If the disk is used for other purposes than writing the audit log this time will increase due to higher seek times and rotational latencies.

12 JDK - System classes and protection domains

Other Java operating systems, such as KaffeOS and the J-Kernel include a large part of the JDK in the TCB. In order to avoid the security problems of a complex TCB, each JX protection domain gets its own set of untrusted system classes. These may or may not conform to the one of the standards that is set by Sun's JDK (JDK1.1, J2EE, J2SE, J2ME). No sandboxing needs to take place at this level, as sandboxing is done by encapsulating domains. Therefore no SecurityManager is needed. The minimal requirements for JDK classes are codified in the jdk_0 component (see Chapter 6, Section 5). Important classes of the JDK are handled as follows:.

- The static variables in, out, and err in the System class define the standard input, output, and error streams. These streams can be redirected, e.g. to a network connection, by setting the value of these variables appropriately. But notice that because they are final, they can only be set once. We do not require that these variables exist. We do not even assume the existence of the System class.
- The classes RandomAccessFile, File, FileInputStream, and FileOutputStream in the java.io package define, how the file system is accessed. By replacing them a domain can utilize a different file system. When this file system is implemented in another domain, the file classes use portals to access the file system domain. These classes are implemented in a library component.
- The classes Thread and ThreadGroup are the interface to the thread implementation in the lower level execution environment.
- The String class must be present and contain a constructor which can be passed a character array. This is called by DomainZero when loading the domain into memory and creating constant strings. Notice, that we do not require, that String objects be immutable. If a domain chooses to implement mutable Strings it is free to do so, but most existing code assumes immutable Strings and will not function properly. The security of our system is not based on the immutability of strings.
- The Object class must exist and must be the only class without a superclass. No further requirements exist for this class.

Most domains will use the same system classes, that provide an API that is compatible to the Sun implementation. The compiled native code of these classes is not duplicated in each domain but provided by the compiler domain that compiled these domains. The code need not even be present in main memory but can be loaded on demand when the method is called. For this purpose, the entry in the vtable points to the loader program instead of the real method.

The JDK is not part of the TCB. However, there are some classes, whose definition is very tightly integrated with the JVM specification [119][77].

Although these classes (except Object) are implemented outside the runtime system, the runtime system must know about their existence or even know part of their internal structure (fields and methods). These structural requirements are checked by the verifier.

The class Object is the base class of all classes and interfaces. It contains methods to use the object as a condition variable, etc. In JX Object is implemented by the runtime system. The class String is used for strings. Because String is used inside the runtime system, it is required that the String class does exist in a domain and that the first field is a character array. The runtime system needs to throw several exceptions, such as ArrayIndexOutOfBoundsException, NullpointerException, OutOfMemoryError, StackOverflowError. It is required that these classes and their superclasses RuntimeException, Exception and Throwable exist in a domain. There are no structural requirements for these classes. Arrays are type compatible to the interfaces Cloneable and Serializable. These interfaces also must exist in a domain.

Classes are represented by the portal jx.zero.VMClass. But because Object contains a method getClass(), it is required that java.lang.Class exists and contains a constructor which has one parameter of type VMClass.

13 Maintaining security in a dynamic system

An operating system is a highly dynamic system. New users log in, services are started and terminate, rights of users are changing, etc. To maintain security in such a system, the initial system state must be secure and each state transition must transfer the system into a secure state.

There are two issues to be considered here: the system issue and the security policy issue.

The system issue is concerned with secure boot process (for example by booting from a tamper-proof device like a CD-ROM), the initial domain correctly starts the security services and attaches them to the created domains. Each domain is started with no rights (only the naming portals and no trusted code) and is associated to a principal. When a domain obtains new portals or communicates using existing portals the security system is involved.

The policy issue is concerned with secure changes of the access rights, additions of principals, etc. How this is done depends on the used security policy. Example of how to securely maintain right changes are described in [176][69].

14 Securing servers

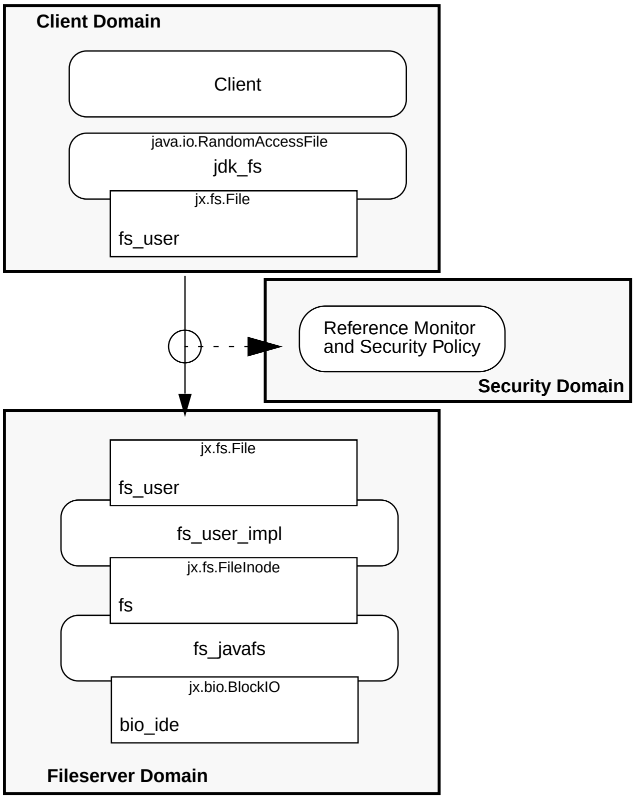
We use the file system server to illustrate how our security architecture works in a real system. As we discussed in Section 5.1 we use the server interface to make access decisions. For this to work servers must export *securable interfaces*. A securable interface must use simple parameters and provide fine-grained simple operations.

Many servers have a built-in notion of permissions, for example the user/group/other permissions in a UNIX file system. We will call them *native permissions*. These permissions can be supplemented or replaced by a set of *foreign permissions*. These permissions could, for example, be access control lists. Because foreign permissions are not supported by the server, there must be a way to store them. The SELinux system [121] uses a file hierarchy in the normal file system to store foreign permissions.

14.1 Securing the file server

There is some scepticism whether a capability-based system can be compatible to the JDK (see the discussion of capabilities in [182]). We proved that this is possible by implementing a component that implements the java.io.* classes in terms of our capability-based filesystem interface (Figure 10.5). Code that uses the java.io classes can run unmodified on top of our implementation of java.io. But the advantages of a capability-based system are lost: files must be referenced by name and problems similar to the Confused Deputy [82] are possible. An application can avoid the problems by using the (not JDK-compatible) capability-based file system interface.

Figure 10.5: Filesystem layers



In an MLS system in which the file system is part of the TCB, the file system must be verified to work correctly - which may be a difficult task as file systems employ non-trivial algorithms. We used a configuration which eliminates the need for file system verification. Our system creates different instances of the file system for the different security levels, each file server being able to use a disjunct range of sectors of the disk. Assuring correct MLS operation can now be reduced to the problem of verifying that the disk driver works correctly; that is, it really writes the block to the correct position on the disk. The file system may run outside the TCB with a security label that is equivalent to the data it stores.

14.2 Securing the database server

In contrast to the file system, a database system usually is not part of a traditional operating system. It must implement and enforce its own security policy. In a microkernel system, database security is not that different from file system security. The database must export an interface that allows to derive access decisions. An interface that accepts SQL statements does not have this property.

Certain policies that emphasize data integrity, such as Clark-Wilson, require that the data is not directly exposed to users (as would be the case in a FS), but only available via trusted processes or functions. A data base is better suited for this task than a file system.

For a relational database it seems very natural to formulate the security policy itself in the relational model [144]. It must be stored in the database and the security manager must access the database.

15 Evaluation

This section discusses how the described architecture meets the requirements stated in Section 2.

Economy of mechanism. The microkernel is small and as simple as possible. The concept of stack inspection is no longer needed. Even untrusted code can obtain a capability to do useful work in a restricted way. JX relies on the type safety of the Java Bytecode language. If a flaw in the type system is found the whole system is compromised. We assume that Java is type-safe. There is a lot of ongoing and finished work on formally proving the type safety of Java. Using a simpler intermediate language could make this proof easier and also reduce the size and complexity of the bytecode-to-nativecode translator [132].

JX security does not rely on specific features of the Java language, like access modifiers for fields and methods, thus eliminating a possible class of attacks [151].

Fail-safe defaults. Basing access decisions on fail-safe defaults is mainly the responsibility of the security manager (which is not part of the basic architecture). As an example, we implemented a security manager that allows communication between dedicated principals and automatically rejects all other communication attempts.

Complete mediation. All accesses to a portal can be checked by the reference monitor.

Open design. The design and even the implementation of JX is completely open.

Separation of privilege. The microkernel-based architecture supports a system design where privileges are not centralized in one component but distributed through the system in separate domains. Domains do not trust each other; therefore breaking into one domain has a strictly limited effect for the overall system security.

Least privilege. A domain starts with the privilege to lookup portals. What portals can be obtained by a domain is limited by the name service and also by the reference monitor that is consulted when a portal comes into a domain or is passed to another domain. If the file system is compromised file data can be modified and disclosed, but a database or file system that run in another domain can still be trusted - as long as it does not trust the compromised file system domain.

Least common mechanism. JX allows controlled sharing between applications (domains) using portals. Domains do not share resources that are implemented by the microkernel. All resources, like files, sockets, and database tables, are implemented by domains and shared using portals. Domains have separate heaps and independent garbage collection.

Psychological acceptability. Whether the user accepts a security policy depends on the formulation and implementation of the policy and on the user interface. This is outside the scope of this thesis.

Secrecy. Portals are the main mechanism to restrict data flow. The reference monitor is able to enforce a wide range of access control policies, including MLS.

But data can also flow on covert channels, either on storage channels or on timing channels [124], [103].

Storage Channels. A file system potentially contains many covert storage channels, such as file names, access permissions, and the amount of free disk space. To avoid these channels we use different file systems for different security domains or security clearances. They all access the BlockIO interface, which is implemented in the TCB and uses a fixed partitioning of the block device.
This architecture is an example of polyinstantiation, which is often used to remove covert channels. There is an obvious storage channel when using portal calls between domains of different clearances. Even if only portals and no data parameters flow upward, the dynamic type of the portal parameters can be used as a covert channel.

Timing channels. These channels are more difficult to avoid. In the example above, the two file systems could use the shared block I/O as a timing channel by moving the disk arm and measuring the difference in disk access time thus timing disk arm movements [141]. The bandwidth of such a channel can be reduced by doing random disk accesses. Timing channels could be reduced further by making it more difficult for the domain to measure real time at a high resolution.

JX is good in coping with the timing channel problem, because there already is “noise” on most channels. The use of an intermediate instruction set makes it difficult to exploit cache effects or to time instructions. Automatic memory allocation and garbage collection eliminate memory as covert channel. There is only a minimal shared state in the microkernel. To reduce the bandwidth of channels that exploit scheduling latencies, a randomized scheduling strategy can be used in the global scheduler.

Modularity. One requirement for a security architecture is a small and modular TCB. Table 10.1 gives an estimate of the complexity of several systems by counting lines of code (kLOC = 1000 lines of code). When comparing the numbers one should keep in mind that different programming languages are used: the Translator and Verifier of JX are written in Java, the kernel of JX and all other systems are written in C and assembler. A number of programming errors that are possible in C and assembler are not possible in Java, such as memory management and pointer manipulation errors.Therefore we assume that Java programs contain fewer bugs per LOC.

All systems have between 30 and 120 kLOC. The largest part of the Linux source code are device drivers. But only few drivers are normally linked to the kernel statically or as a module. The Linux number only contains the absolutely necessary part of the sources. The number would be higher in one of the standard distributions where the kernel contains additional file systems, network protocols, or other services.

An interesting observation is that the memory management (mm) part of the Linux kernel is larger than our bytecode verifier, which shows that so called hardware-based protection needs as much software as software-based protection.

Using a Java processor the translator can be eliminated from the TCB. This would reduce the size of the integrity kernel to 37 kLOC.

System	Parts	kLOC	total kLOC
SecureJava [52]	Paramecium Kernel Java Nucleus	11 22	33
Linux 2.4.2	kernel mm ipc arch/i386/kernel arch/i386/mm fs fs/ext2 net/ipv4	13 15 3 25 1 23 5 34	119
Linux 2.4.2 drivers			1711
Linux 1.0 & drivers			105
LOCK [162]	TCB	87	87
KeyKOS [146]	Kernel Domain code	25 25	50
JX integrity kernel (no drivers)	Microkernel Translator Verifier	25 40 12	77

Tab. 10.1 Operating system code sizes

Integrity. The methods of a portal can be regarded as transformation procedures. Access to a domain and to the data that is represented by a domain is only possible using portals. Moving the data from one consistent state to the next and monitoring the integrity of the data is the responsibility of the domain

Separation of policy and enforcement. The security policy is not part of the servers. Even the enforcement is separated from the functional part of the servers.

Programming language. Many security flaws are due to language defiances, like buffer overflows, that simply cannot happen in a language like Java. We tried to keep the non-Java portion of the system as small as possible. As can be seen in Table 10.1 the microkernel, which is the only part of the system that is written in an unsafe language, is rather small (25 kLOC) compared to the other parts of the TCB.

Performance. We present a performance evaluation of the JX system in Chapter 12. Section 3.5 of Chapter 12 contains measurements of the impact of the described reference monitor mechanism on application performance.

16 Summary

We described the security architecture of JX, which can be seen as a hybrid of language-based protection and operating system protection. It reconciles the performance and integrity of a type-safe protection mechanism with the strong isolation and complete mediation of operating systems. JX avoids typical Java security problems, such as native methods, execution of code of different trustworthiness in the same thread, and a huge trusted class library. The JX TCB consists of three rings: the integrity kernel guarantees essential system properties, like type-safety and domain isolation; the security kernel enforces a certain security policy; the outer ring of the TCB contains all services that the user trusts.

JX provides a number of security mechanisms of different invasiveness. The capability mechanism is inherent in the architecture and guarantees a minimal level of security. On a per-domain basis this mechanism can be supplemented by a monitor that controls propagation of capabilities between domains and, if necessary, a reference monitor that mediates access to these capabilities.

We described the system modifications that were necessary to support confinement. The measured performance overhead of the reference monitor indicates that this mechanism should not be used if not needed. We believe that for most commercial systems the pure capability system, with proper interface design (e.g., a read-only interface), supplemented by the capability propagation monitor will provide sufficient security and guarantee confinement at a low cost.

CHAPTER 11 *Related Work*

This chapter compares the JX architecture to other projects that extend Java with capabilities that are important for operating systems, such as isolation of protection domains, inter-domain communication, sharing between domains, resource accounting and user-defined resource management, reusability, scalability, and security. Our evaluation of the related systems is guided by a number of questions:

Isolation. How are protection domains implemented? How well can domains be isolated from each other? Can domains be completely confined? Can domains be terminated independently from each other? Can domains use different JDK implementations?

Communication. Is fast inter-domain communication possible? Does communication creates interdependencies between protection domains?

Sharing. Is code and/or data sharing possible? How transparent can code be shared? How are special cases, such as static fields or synchronized methods, treated?

Resource accounting. How exact is resource accounting and how well does the system cope with specific problems? When accounting CPU time a specific problem is the considerations of garbage collection time and time spent in interrupt handlers. When accounting memory specific problems are the account for stack and thread control block usage.

User-defined resource management. Is it possible to use different garbage collectors in different domains and are the GC runs independent from each other. Does the system provide any support for the implementation of independent garbage collection? Is it possible to use different schedulers for different protection domains?

Reusability. How does the addition of protection domains affect reusability?

Scalability. How expensive are protection domains. Is it possible to efficiently create small as well as large domains?

Security. How secure is the system with respect to the criteria presented in Chapter 10? What security mechanisms are provided by the system?

The following projects either attempt to provide an operating system structure or attempt to improve one of the aspects in our list (the projects are listed in no particular order):

Conversant [16] by The Open Group Research Institute was one of the first attempts to enhance Java with separate protection domains and resource accounting. Whenever a new name space is created by using a new class loader a new heap is created and used by the code that is loaded by the class loader. System classes are loaded by the internal class loader and shared between protection domains. Inter-heap references can be created by passing references through static variables of shared system classes. A runtime check (write barrier) is used to detect attempts to create inter-heap references.

KaffeOS [10] by the University of Utah is an extension of the Kaffe JVM to support a process abstraction. The design is similar to Conversant. Heaps are separated and a write barrier is used to detect attempts to create inter-heap references. There is no IPC primitive, but shared heaps are used for communication. The system provides a process abstraction that is similar to UNIX, with kernel-mode code and user-mode code.

JRes [45] by the Cornell University is a pure Java system that uses bytecode rewriting in class loaders to support resource accounting. Before each object allocation instruction and in each finalizer a call to the resource management system is inserted.

J-Kernel [87], [86] by the Cornell University was developed by the same research team that developed JRes. It supports a task structure and capability-like inter-task references on top of an unmodified JVM. The J-Kernel is layered on top of RMI and extends RMI by a capability mechanism. Capabilities are realized by generating proxy classes “on-the-fly” and using a reference to an instance of such a proxy class instead of the original reference.

Luna [85], [86] by the Cornell University is a follow-on project to the J-Kernel. It extends the Java type system and introduces a new type for remote pointers. Remote pointers are references to objects of a different task. Remote pointers allow fine-grained sharing between tasks and they can be revoked either explicitly or when the target task terminates.

Secure Java [52] by the Vrije Universiteit Amsterdam and IBM T. J. Watson Research Center is a JVM that runs on top of the Paramecium [53] kernel.

Aurora [148], [139] by Oracle is the central component of JServer, the Java runtime environment that is embedded in Oracle’s database. The main goal of the Aurora JVM is scalability, i.e. the support of many concurrent database sessions.

MVM [44] by Sun Microsystems is an extension of the HotSpot JVM that allows to run multiple JVMs in a single process. When possible, data is shared between the JVMs.

Scalable JVM OS/390 [51] by IBM is a port of a JVM to the OS/390 operating system. As this port aimed at using a JVM to run server programs, it focused on reliability, availability, security, and scalability. These concerns were addressed by using facilities of the underlying OS/390 system: reliability and availability by using performance monitoring, security by using the principal-based access control of OS/390, and scalability by using OS/390 processes and shared memory to store the resolved system classes.

JavaSeal [27] by the University of Geneva and its follow-on project J-SEAL2 [22], by the Technische Universität Wien is a pure Java implementation of resource accounting and management.

JavaOS [154] by Sun Microsystems is a complete Java operating system. Similar to the Pilot OS [147] (see Section 2.5 of Chapter 2) JavaOS is a single-user system. As JavaOS is the only real Java-based operating system besides JX, it is very related to the work described in this dissertation.

All systems besides JavaOS are no real operating systems but extensions of Java runtime environments with functionality that usually is associated with operatings systems, such as task termination and resource control.

1 Architectural aspects

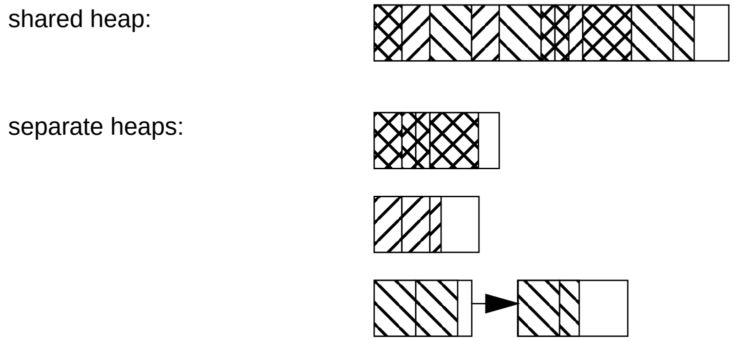
1.1 Isolation

Systems that provide protection domains on top of an unmodified JVM use the name spaces that are created by class loaders. Inside a JVM a class is uniquely identified by its class name and the class loader that loaded the class. A class contains references to other classes in its constant pool. When these class names are resolved by the JVM internal linker the class loader that loaded the class is requested to load the new class. Using a new class loader to load a class creates a new, separate name space because the new class can not use previously loaded classes. An exception are system classes that are loaded by a JVM internal class loader and that are accessible in all name spaces.

An unmodified JVM uses a shared heap for all protection domains (name spaces). This leads to an optimal memory utilization but allows for denial-of-service attacks. To make these attacks impossible systems either keep the shared heap model and account object allocations (JRes) or use separate heaps. Fragmentation can be reduced by using heap fragments (as illustrated by the third domain in Figure 11.1). A shared heap model usually is combined with a shared thread model. The thread contains frames of different protection domains (see Figure 11.2).

Figure 11.1: Shared heap vs. separate heaps

The different patterns represent objects of different protection domains. In a shared heap model objects of different domains are mixed which unnecessarily complicates resource accounting and reclamation.



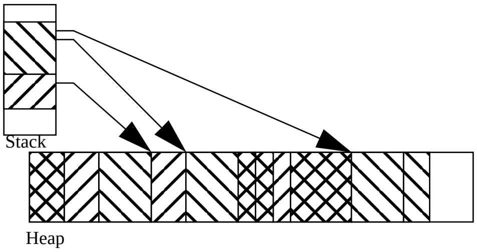
Heap. It is relatively easy to share data between protection domains in a Java system. A multi-tasking system must prevent DoS attacks that simply fill the heap or lead to a fragmentation of the heap. Systems that use a single heap and that do not account allocations are subject to simple heap exhaustion attacks. These systems include JavaOS, JKernel, Aurora.

Conversant and KaffeOS use separate heaps and write barriers to detect attempts to create inter-heap references. This check slows down the putstatic, putfield, and aastore bytecode to about 50% of the original speed in the Conversant system.

KaffeOS uses a separate heap per process and shared heaps for interprocess communication. Instances of system classes are allocated on a shared heap. System classes must be modified to protect (hide)

Figure 11.2: Stacks with mixed frames

A shared heap model is usually combined with a shared thread model. A single thread executes code that belongs to different protection domains and thus the thread contains frames that belong to different domains. These frames contain references to objects of their domain. Although an inter-domain invocation has nearly zero overhead, the shared stack makes clean and safe termination nearly impossible.



shared kernel locks (example is the Thread class). They must also be modified to multiplex state between processes (example is the static System.out variable that represents the standard output stream and should be process local). KaffeOS allows sharing between processes and allows access to the fields of shared objects. We think that this kind of sharing is fundamentally flawed because it can be used to violates the contract of the abstract data type (ADT). To circumvent that a malicious process violate the ADT contract by directly modifying the state of the object KaffeOS programs will end up encapsulating this state and allowing access only through methods, which will be equivalent to the JX sharing model.

The J-Kernel [86] uses the Java classloading mechanisms to add a task structure (termination, revocation, resource accounting) to an unmodified JVM.

JSEAL uses the Java Resource Accounting Facility (JRAF) [3] that is implemented in pure Java. Therefore, it can be used with standard JVMs. J RAF is based on bytecode rewriting techniques. Memory allocation instructions are redirected to a controller object that denies object allocation if a limit is exceeded.

The Aurora JVM uses different heaps for different object lifetimes. But different sessions are not isolated; neither in their memory consumption nor in their CPU consumption. As a deadlock therefore may affect all users of the database the system integrates a deadlock detection mechanism.

Code and static fields. It should be possible to share code between protection domains. In order to preserve the illusion of independent virtual machines static variables should not be shared. KaffeOS loads classes multiple times. This wastes memory and requires recompilation by the JIT compiler, which increases application startup latency and wastes additional memory. The MWM supports code sharing with domain-specific static fields.

1.2 Communication

There is no fast communication mechanisms for applications inside one MVM. Communication between processes in KaffeOS is done using a shared heap. Our goal was to avoid sharing between domains as much as possible and we, therefore, use RPC for inter-domain communication

1.3 Sharing

Allow field access to shared object may break the object abstraction.Allowing access via public method requires synchronization. Allowing synchronization allows DoS attacks by acquiring but never releasing the lock. Therefore all locks must be private as in the KaffeOS system classes.

In KaffeOS objects can be shared between protection domains. References to objects of domain A can be on the stack of a thread of domain B. It is therefore necessary to include the stacks of all domains in the garbage collector's root set of a domain. Although KaffeOS contains some clever mechanisms that allow the GC to run concurrently with threads of foreign domains, all threads of the system must be scanned. This is extremely unscalable and will render KaffeOS unusable when many domains, i.e. many threads, are used. To make use of the shared heap the interface of standard classes, such as java.util.Hashtable, must be changed.

Synchronization. Systems that share data between protection domains must provide a mechanism to synchronize access to this data. This mechanism must be carefully designed to prevent denial-of-service attacks. When using a simple mutual exclusion lock as synchronization primitive a malicious domain could hold the lock forever thereby blocking the progress of other domains. This is a problem in systems that share classes and static fields. The static method java.lang.Math.random(), for example, is synchronized, i.e. it uses a static lock. In KaffeOS the kernel code must not use a publicly accessible object (including class objects) for synchronization. Otherwise user code can block the kernel forever. The proposed technique is to create an object that is only accessible within the trusted kernel class.

1.4 Resource accounting

The pure Java systems (JRes, J-Kernel, JavaSeal) are not able to account for stack and thread control block memory and for GC time. There are no means for resource control for applications inside one MVM.

1.5 User-defined resource management

There is no provision for user-defined resource management in any of the systems with the exception of JX.

1.6 Scalability

To achieve scalability the Aurora VM reduces the minimum incremental per-user session footprint, shares read-only data, such as Java bytecodes, and uses different garbage collection algorithms, depending on the type of memory.

A regular JVM has a minimal footprint of about 2 to 3 MB data. Running 10,000 JVMs therefore consumes between 20 and 30 GB memory for data. The Aurora VM reduces this amount to a minimum of 40-60KB per session and 200 KB for medium sized program, such as an ORB [139]. This means that 10,000 sessions fit into 2GB of memory and there is no need to use slow secondary storage.

There are three different memory types with different lifetime durations: call, session, and global. *Call* memory persists during a (possibly recursive) call into the database, *session* memory exists for the duration of the session connected to the database, and *global* persists as long as the database instance is running.Different garbage collectors are used for the different memory areas. The short-living call area is collected by using generational scavenging, session memory is collected by using a copying collector that compacts memory. The call duration memory contains the C stacks, the Java stacks, the newspace and the oldspace [148]. To avoid timeslicing overhead threads are scheduled non-preemptively.

While JX domains are a more general concept they provide the same advantages. Depending on the purpose of the domain a specific garbage collection implementation can be selected. Read-only state, such as compiled code, is shared between domains, which leads to a low per-domain memory footprint and good scalability.

The MVM and IBM's JVM port to OS/390 also allow sharing of class data to improve scalability.

1.7 Security

Secure Java [52] aimed at reducing the TCB of a Java VM to its minimum. The bytecode verifier and just-in-time compiler are outside the TCB. The JIT can be inside the TCB to enable certain optimizations. The garbage collector is inside the TCB, but because the JIT and verifier are not trusted the integrity of the heap can not be guaranteed. We think that this is the main problem, because not relying on the integrity of the heap complicates the GC implementation and complex implementations should be avoided in a secure system. Heap integrity is important when reasoning about security of higher level applications. Systems that are layered on top of a JVM and/or require a large part of the JDK library (all systems besides SecureJava and JX) have the problem of a very large TCB.

Containment of information is not a problem many system seems to care about. Especially the systems that support shared heaps (Conversant, KaffeOS) may leak information through shared objects.

1.8 Reusability

One of the objectives of the JX project was an enhanced reusability of operating system components. To enable the reuse of components between different configurations, either operating within the same domain or across a domain boundary, inter-domain communication must be transparent for the functional part of the component. Systems that require an explicit action to create an inter-domain communication channel make reuse impossible. On the other hand, a domain or component must be able to control the communication activity. A conceptual model for this kind of separation is a meta-level structure, also called reflection. The JX portal interception architecture was influenced by the ideas of the metaXa system [73], which was the first JVM supporting behavioral or computational reflection [123].

The management of capabilities in the J-Kernel is done explicitly. The capability system is not orthogonal to application code which makes reuse in a different context (using a different security policy) difficult. A capability is created by calling Capability.create() and passing a Remote object as parameter. The Capability.create() method dynamically creates a stub class that implements the Remote interface. The created methods perform a revocation check, switch stack segments, and copy parameters. The returned capability object is an instance of the dynamically created class.

2 Summary

By evaluating related systems according to architectural aspects that are required to build a general-purpose time-sharing operating system this chapter demonstrated that none of the related systems can be used to build such a system.

The projects can be classified according to their changes to the Java runtime environment. There are projects that extend or modify the Java compiler instrument class files, projects that modify the runtime system, and projects that modify the operating system. Depending on the amount of modification different levels of isolation, resource management flexibility, and resource control can be achieved. Only a solution that integrates the operating system and the runtime system can achieve full isolation, maximal flexibility, and exact resource control. JavaOS, as the only operating system besides JX, is the only system that is able to completely control the resource consumption But as it misses a protection domain concept it can not be classified as a multi-user or timesharing system.

CHAPTER 12 *Evaluation*

This chapter evaluates whether the JX architecture is suitable as the foundation of general purpose or dedicated operating systems. The first part of this chapter describes several typical operating system services, such as a file system, a web server, a network file system, a window manager, and a database system. The second part of the chapter compares the performance of JX with a traditional UNIX system (Linux). We do not use synthetic benchmarks, such as JVM98 [198], but real operating system components: a Linux-ext2-compatible file system, a network file server speaking the NFS protocol, relational database system, and a web server. The final two sections analyze the file system performance and the NFS performance in more detail.

1 Functional Evaluation

We present four examples to demonstrate the suitability of the JX architecture for the construction of general-purpose and dedicated operating systems. The examples are a file system, an NFS server, a window manager and a database management system.

1.1 Filesystem

The file system consists of four service components with their respective interface components: the device driver, the block buffer cache, the ext2 file system, and the file system user interface (see Figure 12.1).

An application interacts with the file system by using the FSApplicationInterface component, either directly or via the JDK_FS library component. When an application opens a file, a file object is returned. The file object provides methods to access and modify the file.

The file system interface is a capability oriented abstraction layer for the file system. It accesses the inode based information of the file system component. The file system component contains the actual implementation of a specific file system. We currently use an Ext2 compatible system that was ported from Linux [186]. The file system component uses the block buffer to get the raw data from a storage device. The block buffer reads and writes sectors from the device driver. More precisely it uses a reference to an object implementing the BlockIO interface. The BlockIO component that is accessed by the block buffer component need not necessarily be a device driver for a local disk. It may also transfer blocks over the network, or store the blocks in main memory.

Usually, the file system is used by more than one application. Therefore a reasonable system configuration is to use a file system domain that provides the file system as a service and is accessed by other domains.

If the disk is shared between different file systems or applications that require access to raw blocks the File System Domain can be split into two domains: one domain provides disk access and contains the device driver: the other domain contains the file system implementation.

Files are represented by the portal interface FSObject and its sub-interfaces RegularFile, ReadOnlyRegularFile and Directory. Depending on the file system other interfaces may be used, such as Symlink. When opening a file, the application can specify, that read-only access is sufficient. Then a ReadOnlyRegularFile object, that lacks the methods to modify the file, is passed to the application. When the application wants to write data to a file, it can get a RegularFile object that contains modification operations. There is no way to change the type of a portal from ReadOnlyRegularFile to RegularFile, because dynamic type checks are performed by using the type of the portal object and not the type of the service object. Even if the service object implements both types the type of the portal can not be changed to the other type.

Opening a new file creates a new service in the file system domain. To make this practical service creation must be a low overhead operation and consume few resources. Chapter 5 described how shared thread pools reduce the resource requirements of a service. The file services share a thread pool that usually contains only one thread. It contains more than one thread if the service thread blocks and a new service thread is started (see Chapter 5, Section 6).

The number of files is limited by the number of services that can be created by a domain. Currently there is a static limit on the number of services because of the service table. Using a different data structure to map service ID to service control block would remove this limit. The number of open files would then only be limited by the available heap memory in the file server domain.

1.2 Network File System

The Network File System (NFS) protocol uses SUN RPC for communication. JX contains an RPC layer and a stub code generator that creates RPC stub and skeleton class from Java interfaces. The complete network stack, from the device driver up to the NFS server is deployed in the form of type-safe JX components. The NFS server uses the file system that was described in Section 1.1. The file system components, the network protocol processing components, the NFS server components, and the device driver components can either be collocated in the same domain or run in different domains.

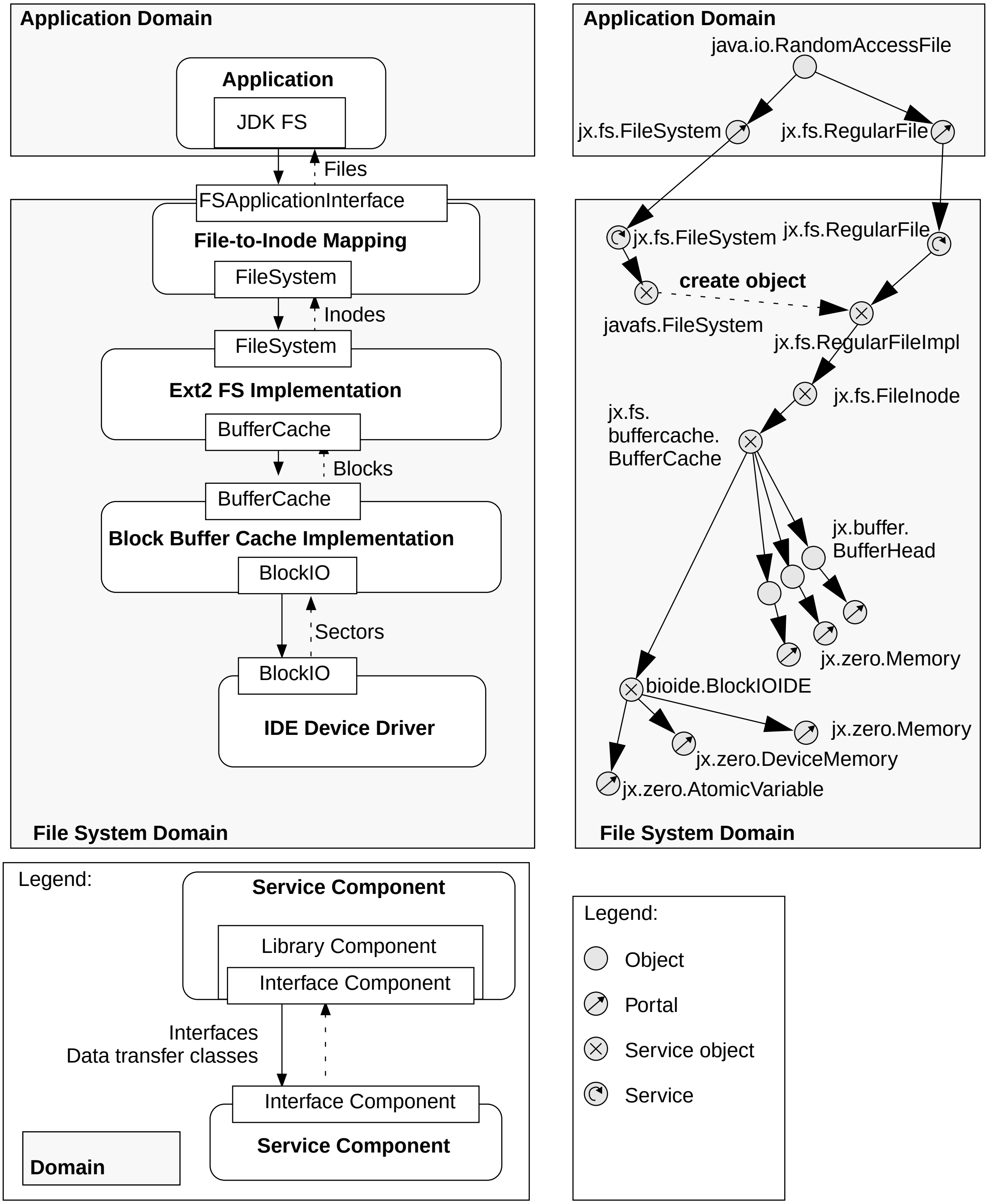
1.3 Window manager

The JX window manager [138] uses the frame buffer component to access a frame buffer device and provides windows as services. Each window has an event queue that is maintained by the window manager.

When a domain is terminated all windows should be closed and the window manager releases all resources associated with these windows. The automatic service lifecycle management mechanism,

Figure 12.1: Components structure and simplified object graph of the file system

The left half of this figure shows the component structure of the file system and an application that uses the file system. The right half of the figure shows a simplified object graph of the two domains.



which was explained in Chapter 5, is used. Portals to a window can be passed to other domains. Therefore the window remains open although it is not completely functional.

1.4 Database Management System

The JX database [47] uses the following abstractions: Database, Table, Index, and Key. These abstractions are represented as portals. The database implementation is hidden behind these portals. The implementation uses fairly standard database techniques, including B-Trees. The details of the implementation are described in [47].

1.5 Webserver

As our JDK class library misses functionality that is required to run an off-the-shelf Java web server, such as tomcat [197], we wrote a simple Java web server. The JX web server accepts a connection, creates a either a new domain or a new thread and passes the portal that represents the TCP connection and a portal to the file system to the new domain/thread.

The web server creates a new domain “Web worker” for each new HTTP connection. It passes a java.net.Socket object and a jx.fs.FS portal object to the new domain. The java.net.Socket is implemented by using a jx.net.TCPSocket portal which is passed to the new domain as part of the Socket object graph. The “Web worker” domain parses the HTTP stream and eventually accesses the file system to read the requested html file. The file is represented by a jx.fs.FileInode portal.

2 Performance evaluation

In this section we evaluate the performance of the JX system using selected components: the file system, the NFS server, and the database management system.

2.1 Filesystem

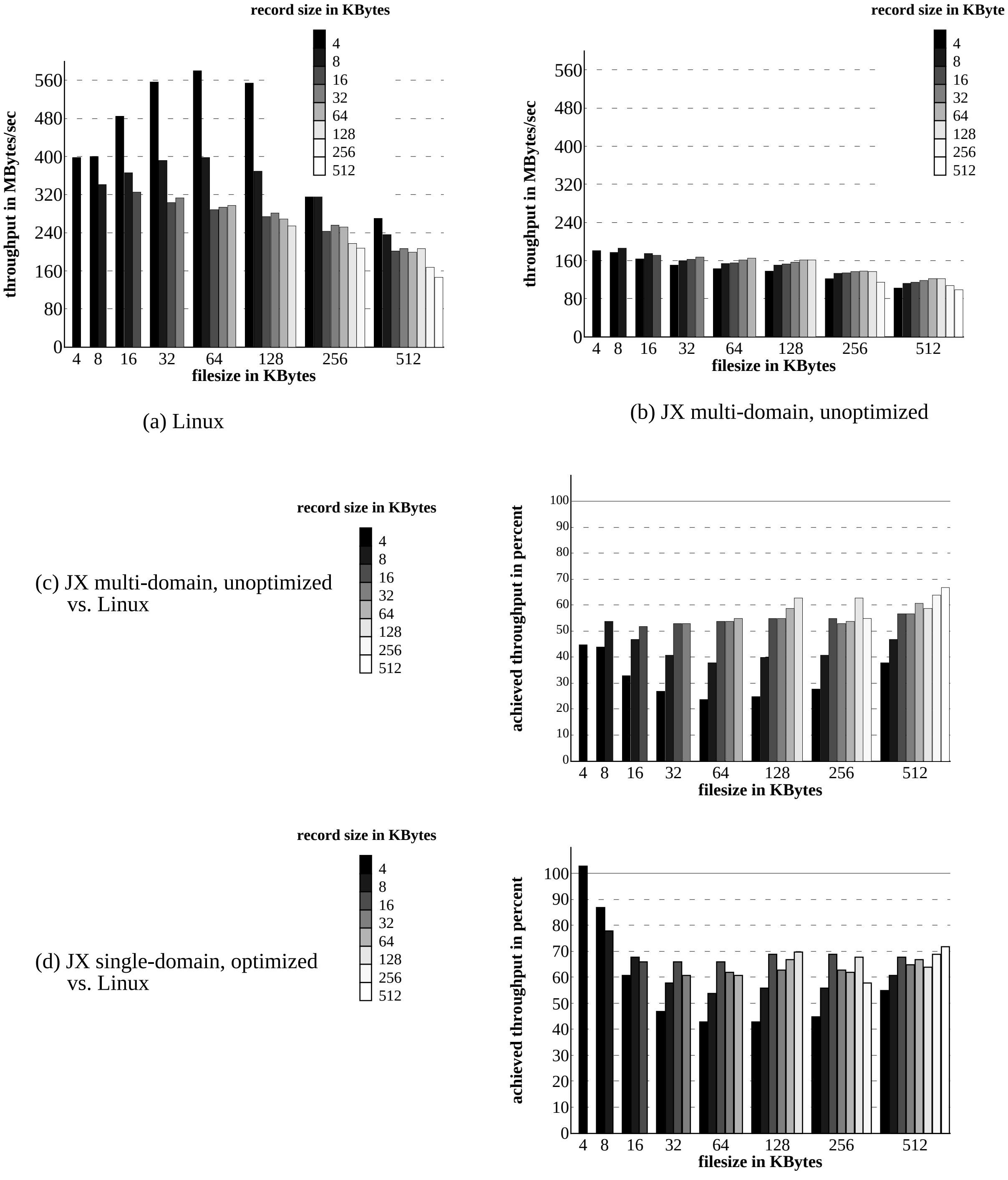
To evaluate the performance of the JX file system we used the IOZone benchmark [200]. As JX is a pure Java system, we can not use the original IOZone program, which is written in C. Thus we ported IOZone to Java. The JX results were obtained using our Java version and the Linux results were obtained using the original IOZone.

Figure 12.2 shows the results of running the IOZone reread benchmark on Linux.

Our Java port of the IOZone contains the write, rewrite, read, and reread parts of the original benchmark. In the following discussion we only use the reread part of the benchmark. The read benchmark measures the time to read a file by reading fixed-length records. The reread benchmark measures the time for a second read pass. When the file is smaller than the buffer cache all data comes from the cache. Once a disk access is involved, disk and PCI bus data transfer times dominate the result and no conclusions about the performance of JX can be drawn. To avoid these effects we only use the reread benchmark with a maximum file size of 512 KBytes, which means that the file com-

Figure 12.2: IOZONE: JX vs. Linux

This figure shows the results of running the IOZone benchmark on Linux and on different JX configurations. Figures (a) and (b) display absolute throughput, while figures (c) and (d) compare the JX throughput with the Linux throughput of figure (a).



(a) Linux

(b) JX multi-domain, unoptimized

(c) JX multi-domain, unoptimized vs. Linux

(d) JX single-domain, optimized vs. Linux pletely fits into the buffer cache. The JX numbers are the mean of 50 runs of IOZone. The standard deviation was less than 3%. For time measurements on JX we used the Pentium timestamp counter which has a resolution of 2 ns on our system.

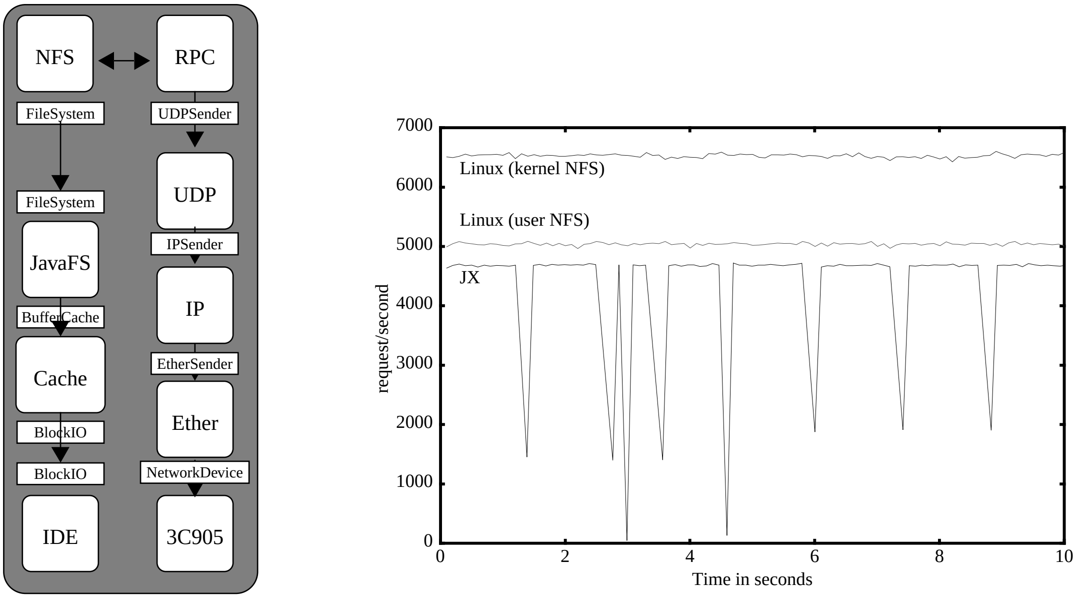
Figure 12.2 shows the results of running the IOZone benchmark on Linux (a) and JX (b). Figure 12.2 (c) compares JX performance to the Linux performance. Most combinations of file size and record size give a performance between 20% and 50% of the Linux performance. Linux is especially good at reading a file using a small record size. The performance of this JX configuration is rather insensitive to the record size. Figure 12.2 (d) compares an optimized JX system to Linux. We will explain the optimization options in detail in Section 3.

2.2 Network File System

To assess NFS performance we use a home brewed *rate* benchmark. The rate benchmark sends *getattr* requests to the NFS server as fast as possible and measures the achievable request rate.

Another benchmark is the rate benchmark, which measures the achievable NFS request rate by sending *getattr* requests to the NFS server. Figure 12.3 shows the domain structure of the NFS server: all components are placed in one domain, which is a typical configuration for a dedicated NFS server. Figure 12.3 shows the results of running the rate benchmark with a Linux NFS server (both kernel and user-level NFS) and with a JX NFS server. There are drops in the JX request rate that occur very periodically. These drops are caused by garbage collector runs. The GC is not interrupted, because it disables interrupts as a safety precaution in the current implementation. The pauses could be avoided by using an incremental GC [12], which allows the GC thread to run concurrently with threads that modify the heap.

Figure 12.3: NFS server configuration and request rate



2.3 Database Management System

We compare the performance of a database running on top of JX with the performance of a database running on top of Linux. The JX database was implemented as part of a semester thesis [47]. As Linux DB we use the freely available MySQL system. The MySQL distribution contains benchmarks that we adapted for the JX database. When comparing the results one must consider the following functional differences between MySQL and the JX database:

- The JX database does not contain an SQL parser. SQL statements are translated manually to an operator tree. MySQL performs this translation automatically which costs time especially when using complex SQL statements. We therefore use only benchmarks with few operators that cause many database operations.
- A MySQL client communicates with the database server by using TCP[1] while the JX client uses portals to communicate with the server. We therefore selected benchmarks that do not transfer large amounts of data.

The following benchmarks are used. All benchmarks fit completely in main memory:

Insert. Insert 10000 rows in a table. Each row consists 4 integer columns. Create three indexes. No query optimization is necessary because there is only one operator. Data must be transferred to the server.

Select. Select operations are performed on the created table. Compared to the amount of computation necessary to find the data only a simple query must be optimized by the MySQL query optimizer. No data is transferred to the client.

Update. Update one column in each row. A simple query optimization and data transfer is necessary.

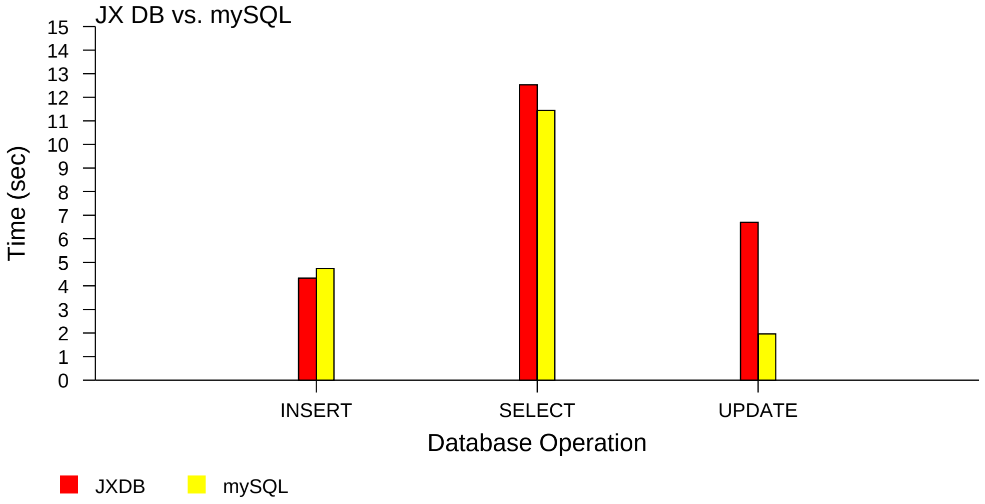
Figure 11.18 shows the results of running the benchmark. While there is no large difference between the insert and select results, the update results differ significantly. Each update triggers the regeneration of the index structures. When turning off index regeneration the update benchmark requires only 0.05 seconds. An obvious optimization would be to generate indexes lazily, i.e., only when they are needed.

2.4 Web server

To see how well the JX web server performs we wrote an equivalent web server in C and measured its performance on Linux (Table 12.1). The Linux web server accepts a connection, and either forks a process that parses the http request, reads the requested file and sends a reply, or processes the request without forking a new process. The Linux and JX/thread numbers are not much different. Cre-

[1] Although client and server run on the same host using TCP sockets for communication creates protocol overhead.

Figure 12.4: Database benchmark results



ating a new domain to processes each request is considerably more expensive, but allows to execute arbitrary untrusted code to process the request.

Tab. 12.1 Web server request rate

Benchmark	http request rate (req/sec) (mean of four runs each sending 1000 requests)
JX web server using threads	459
JX web server using domains	142
Linux web server using fork	381
Linux webserver without fork	445

3 Detailed performance analysis of the file system

JX provides a wide range of flexible configuration options. Depending on the intended use of the system several features can be disabled to enhance performance. Figure 12.5 shows the results of running the IOZONE benchmark on JX with various configuration options. These results are discussed in further detail below. The legend for the figures indicates the specific configuration options used in each case. The default configuration used in Figure 3 was MNNSCR, which means that the configuration options used were multi-domain, no inlining, no inlined memory access, safety checks enabled, memory revocation check by disabling interrupts, and a Java round-robin scheduler. The modifications described in this section are pure configurations. Not a single line of code is modified.

3.1 Domain structure

How the system is structured into domains determines communication overheads and thus affects performance. For maximal performance, components should be placed in the same domain. This removes portal communication and parameter passing overhead, and allows further optimizations, such as inlining. Figure 12.5 (a) shows the improvement of placing all components into a single domain. The performance improvement is especially visible when using small record sizes, because then many invocations between the IOZone component and the file system component take place. The larger improvement in the 4KB file size / 4KB record size can be explained by the fact that the overhead of a portal call is relatively constant and the 4KB case is very fast, because it completely operates in the L1 cache. So the portal call time makes up a considerable part of the total time. The contrary is true for large file sizes: the absolute throughput is lower due to processor cache misses and the saved time of the portal call is only a small fraction of the complete time. Within one file size the effect also becomes smaller with increasing record sizes. This can be explained by the decreasing number of performed portal calls.

3.2 Translator configuration

The translator performs several optimizations. This section investigates the performance impact of each of these optimizations. The optimizations are inlining, inlining of fast portals, and elimination of safety checks.

3.2.1 Inlining

One of the most important optimizations in an object-oriented system is inlining. We currently inline only non-virtual methods (final, static, or private). We plan to inline also virtual methods that are not overridden, but this would require a recompilation when, at a later time, a class that overrides the method is loaded into the domain. Figure 12.5 (b) shows the effect of inlining.

3.2.2 Inlining of fast portals

A fast portal interface that is known to the translator can also be inlined. To be able to inline these methods that are written in C or assembler the translator must know their semantics. Since we did not want to wire these semantics too deep into the translator, we developed a plugin architecture. A translator plugin is responsible for translating the invocations of the methods of a specific fast portal interface. It can either generate special code or fall back to the invocation of the DomainZero method. We did expect a considerable performance improvement but as can be seen in Figure 12.5 (c) the difference is very small. We assume, that these are instruction cache effects: when a memory access is inlined the code is larger than the code that is generated for a function call. This is due to range checks and revocation checks that must be emitted in front of each memory access.

Figure 12.5: IOZONE performance

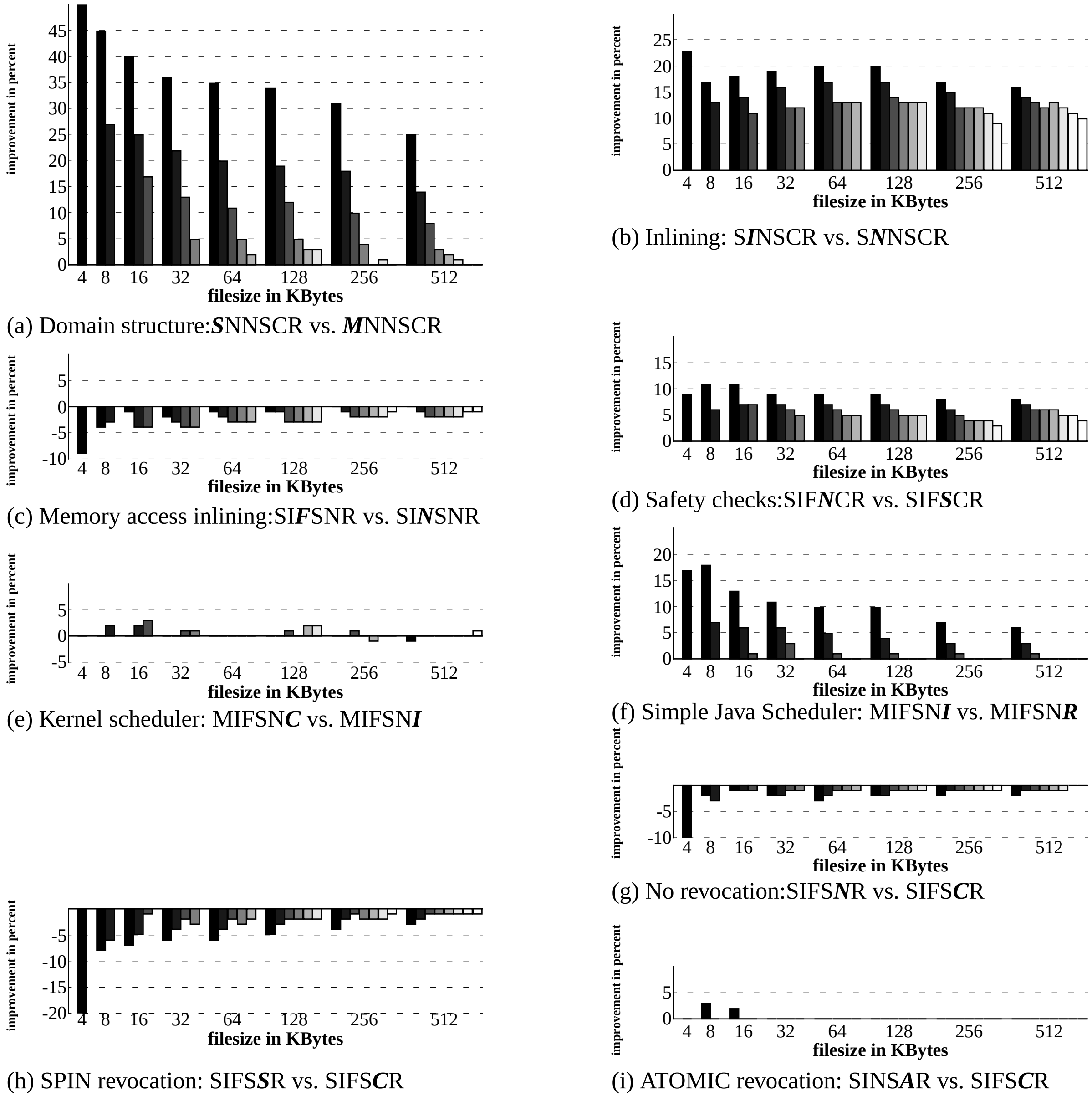
Legend for all figures on this page:

record size in KBytes

■ 4
■ 8
▨ 16
▨ 32
▨ 64
□ 128
□ 256
□ 512

Encoding of the measured configuration:

1. domain structure:	S (single domain), M (multi domain)
2. inlining:	I (inlining), N (no inlining)
3. memory access:	F (inlined memory access), N (no inlined memory access)
4. safety checks:	S (safety checks enabled), N (safety checks disabled)
5. memory revocation:	N (no memory revocation), C (disable interrupts), S (spinlock), A (atomic code)
6. scheduling:	C (microkernel scheduler), R (Java RR scheduler), I (Java RR invisible portals)



(a) Domain structure:***S***NNSCR vs. ***M***NNSCR

(b) Inlining: S***I***NSCR vs. S***N***NSCR

(c) Memory access inlining:SI***F***SNR vs. SI***N***SNR

(d) Safety checks:SIF***N***CR vs. SIF***S***CR

(e) Kernel scheduler: MIFSN***C*** vs. MIFSN***I***

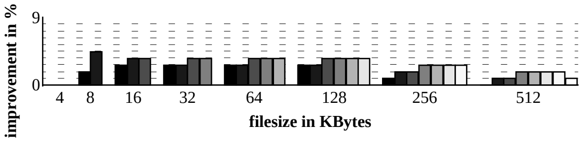
(f) Simple Java Scheduler: MIFSN***I*** vs. MIFSN***R***

(g) No revocation:SIFS***N***R vs. SIFS***C***R

(h) SPIN revocation: SIFS***S***R vs. SIFS***C***R

(i) ATOMIC revocation: SINS***A***R vs. SIFS***C***R

Figure 12.6: IOZONE performance:Stack size check V2 vs. V1



3.2.3 Safety checks

Safety checks, such as stack size check and bounds checks for arrays and memory objects can be omitted on a per-domain basis. Translating a domain without checks is equivalent to the traditional OS approach of hoping that the kernel contains no bugs. The system is now as unsafe as a kernel that is written in C. Figure 12.5 (d) shows that switching off safety checks can give a performance improvement of about 10 percent.

3.3 Memory revocation

Portals and memory objects are the only objects that can be shared between domains. They are capabilities and an important functionality of capabilities is revocation. Portal revocation is implemented by checking a flag before the portal method is invoked. This is an inexpensive operation compared to the whole portal invocation. Revocation of memory objects is more critical because the operations of memory objects - reading and writing the memory - are very fast and frequently used operations. The situation is even more involved, because the check of the revocation flag and the memory access have to be performed as an atomic operation. JX can be configured to use different implementations of this revocation check:

- **NoCheck**: No check at all, which means revocation is not supported.
- **CLI**: Saves the interrupt-enable flag and disables interrupts before the memory access and restores the interrupt-enable flag afterwards.
- **SPIN**: In addition to disabling interrupts a spinlock is used to make the operation atomic on a multiprocessor.
- **ATOMIC**: The JX kernel contains a mechanism to avoid locking at all on a uniprocessor. The atomic code is placed in a dedicated memory area. When the low-level part of the interrupt system detects that an interrupt occurred inside this range the interrupted thread is advanced to the end of the atomic procedure. This technique is fast in the common case but incurs the overhead of an additional range check of the instruction pointer in the interrupt handler. It increases interrupt latency when the interrupt occurred inside the atomic procedure, because the procedure must first be finished. But the most severe downside of this technique is, that it inhibits inlining of memory accesses. Similar techniques are described in [19], [134], [128], [159].

Figure 12.5 (g) shows the change in performance when no revocation checks are performed. This configuration is slightly slower than a configuration that used the CLI method for revocation check. We can only explain this by code cache effects.

Using spinlocks adds an additional overhead (Figure 12.5 (h)). Despite some improvements in a former version of JX using atomic code could not improve the IOZone performance of the measured system (Figure 12.5 (i)).

3.4 Cost of the open scheduling framework

Scheduling in JX can be accomplished with user-defined schedulers. The communication between the global scheduler and the domain schedulers is based on interfaces. Each domain scheduler must implement a certain interface if it wants to be informed about special events. If a scheduler does not need all the provided information, it does not implement the corresponding interface. This reduces the number of events that must be delivered during a portal call from the microkernel to the Java scheduler.

In the configurations presented up to now we used a simple round-robin scheduler (RR) in each domain. The domain scheduler is informed about every event, regardless whether being interested in it or not. Figure 12.5 (f) shows the benefit of using a scheduler which implements only the interfaces needed for the round-robin strategy (RR invisible portals) and is not informed when a thread switch occurred due to a portal call.

As already mentioned, there is a scheduler built into the microkernel. This scheduler is implemented in C and can not be exchanged at run time. Therefore this type of scheduling is mainly used during development or performance analysis. The advantage of this scheduler is that there are no calls to the Java level necessary. Figure 12.5 (e) shows that there is no relevant performance difference in IOZone performance between the core scheduler and the Java scheduler with invisible portals.

3.5 Security

Depending on the configuration, using a reference monitor causes an additional overhead. Figure 12.7 shows the overhead relative to the multidomain configuration that is created by using a monitor that intercepts and allows all invocations.

We implemented a reference monitor which imitates the discretionary access policy of UNIX. Each domain is owned by a principal. The credentials of a principal consist of user ID and a list of group IDs. Each read and write access to the file portal is validated against the user credentials. During this check the security manager asks the file server for the file permissions. As can be seen in Figure 12.8 this configuration is expensive. Using a pure capability architecture is much faster, because only portal creation must be checked but not portal access. This creates, however, the problem of cached access decisions. When the security policy or the security attributes of an object are changed, the portal (capability) still allows access.

3.6 Summary: Fastest safe configuration

After we explained all the optimizations we can now again compare the performance of JX with the Linux performance. The most important optimizations are the use of a single domain, inlining, and the use of the core scheduler or the Java scheduler with invisible portals. We configured the JX system to make revocation checks using CLI, use a single domain, use the kernel scheduler, en-

Figure 12.7: multidomain with null monitor

This diagram shows the overhead of using a null reference monitor as described in Chapter 10..

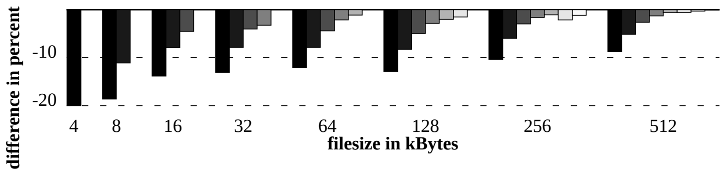
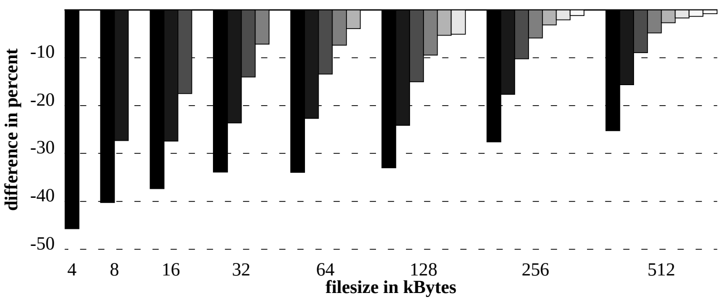


Figure 12.8: multidomain with access check at every read/write

This diagram shows the overhead of using a reference monitor that performs security checks.



abled inlining, and disabled inlining of memory methods. With this configuration we achieved a performance between about 40% and 100% of Linux performance (Figure 12.2). By disabling safety checks we were even able to achieve between 50% and 120% of Linux performance.

4 Detailed performance analysis of the NFS server

We now present a detailed analysis of the behavior and performance of the NFS server. We use visualizations that we developed to understand the behavior of JX components without inspecting their source code. We will explain how we obtained the data for the visualizations, describe how and interpret them

4.1 Method timing

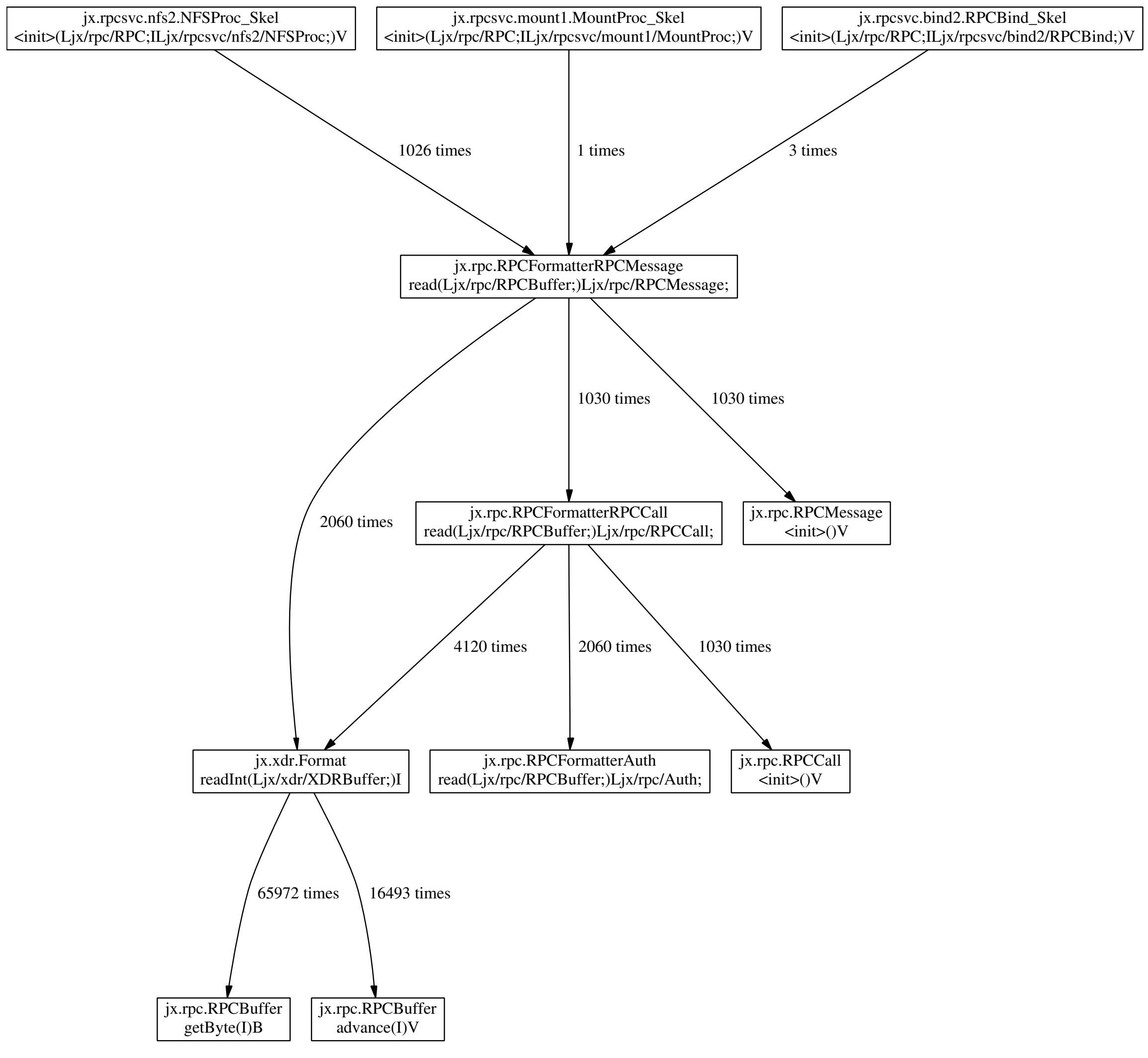
To write faster code, it is necessary for the programmer to identify the most time-consuming parts of a program. These *hot spots of execution* often indicate poor code or bottlenecks. The hot spots are the parts of the code that benefit most from optimizations. We instrumented the bytecode-to-nativecode translator to insert code that measures the execution times of methods.

Hardware supported measuring. Method timing uses the time stamp counter register (TSC) of the Pentium processor family. The TSC contains the elapsed clock ticks since system boot. To measure the time consumed by a method, we save the TSC in the prolog of the method and subtract the TSC in the epilog of the method. The result is accumulated for each pair of (caller, callee) method. The sum is saved together with the number of invocations and the caller and callee address in a data structure per thread. Collecting the data separately per thread avoids locking overhead and allows to analyze the behavior of a dedicated thread.

Analyzing the results. To get a better overview of the executions paths we produce diagrams that show the execution times.

A call graph display (Figure 12.9) is produced by using the *dot* program [193]. After having produced several call graphs we concluded that this visualization is not well-suited to give a good picture of the consumed time. This was the reason to develop another visualization (Figure 12.10) that shows a box for each method, where the size of the box is proportional to the time consumed by this method.

Figure 12.9: Call graph


jx.rpcsvc.nfs2.NFSProc_Skel
<init>(Ljx/rpc/RPC;ILjx/rpcsvc/nfs2/NFSProc;)V
jx.rpcsvc.mount1.MountProc_Skel
<init>(Ljx/rpc/RPC;ILjx/rpcsvc/mount1/MountProc;)V
jx.rpcsvc.bind2.RPCBind_Skel
<init>(Ljx/rpc/RPC;ILjx/rpcsvc/bind2/RPCBind;)V
1026 times
1 times
3 times
jx.rpc.RPCFormatterRPCMessage
read(Ljx/rpc/RPCBuffer;)Ljx/rpc/RPCMessage;
1030 times
1030 times
2060 times
jx.rpc.RPCFormatterRPCCall
read(Ljx/rpc/RPCBuffer;)Ljx/rpc/RPCCall;
jx.rpc.RPCMessage
<init>()V
4120 times
2060 times
1030 times
jx.xdr.Format
readInt(Ljx/xdr/XDRBuffer;)I
jx.rpc.RPCFormatterAuth
read(Ljx/rpc/RPCBuffer;)Ljx/rpc/Auth;
jx.rpc.RPCCall
<init>()V
65972 times
16493 times
jx.rpc.RPCBuffer
getByte(I)B
jx.rpc.RPCBuffer
advance(I)V

4.2 Event tracing

User-generated events. JX allows Java components to log component-specific events. The Java code can register an event name at the microkernel and gets an event number. This number is used for subsequent logging. Time and event number are stored in main memory. The overhead of logging an event is 45 CPU clock cycles.

Figure 12.11 shows the event trace that is produced when the system boots and acts as an NFS server. The system first sends a bootp request and receives a reply packet (at about 48.75 sec.). Then the client mounts the NFS file system, and starts to read the same file in an endless loop. This creates RPC and NFS traffic starting at about 53.6 sec. 0.2 sec. later there starts a dense block of events. These are the *getattr* requests that are sent by the client during the execution of the benchmark program. The block of events is interrupted three times by the garbage collector, that in the current version of JX disables all interrupts. This will be changed when the protocol between GC and intra-domain scheduler is finished.

To improve the NFS request rate, we have to look at the events in more detail, so we zoom into the 3.4 millisecond interval starting at about 53.791 sec. There is an obvious pattern of consecutive events (dotted line) that starts and ends with a *3C905interrupt* event. A complete cycle needs about 905 µsec.

Method enter/exit events. For debugging purposes it is often interesting to know which methods are executed in which order. Our translator can be configured to insert a prologue that logs TSC and method pointer. The same event-recording facilities as with user-generated events are used. Therefore the diagram looks similar.

Scheduling events. To visualize the scheduling behavior of the system we use a thread activity diagram. The kernel is instrumented to collect events during thread switching. The TSC and target thread ID are logged. Figure 12.12 shows the thread activity during boot and the NFS test. The y-axis shows either the thread name or the thread ID for unnamed threads. Some background information is necessary to understand this diagram. “Idle” is the thread that runs when no other thread is runnable. The “Main-Test” thread is the thread that starts the NFS server. “IRQThread8” is the first-level interrupt handler for the RTC interrupt. It runs periodically and fires timers by unblocking other threads. As “IRQThread8” is a first-level handler it runs with interrupts disabled and must complete in a bounded time. Time-consuming tasks are delegated to the “TimerManager” thread. At about 53.79 sec. the system starts to receive *getattr* NFS requests and the thread switches start to occur more frequently. “IRQThread9” is the first level interrupt handler for the NIC (network interface card) interrupt. It acknowledges the received packet with the NIC, places it in a queue, and unblocks the “Etherpacket-Queue” thread that processes that queue.

4.3 Memory

Java’s automatic memory management is very convenient for the programmer. But during performance debugging or in order to find memory leaks the programmer wants to know exactly when and where objects are allocated and destroyed.

Object creation: time and code position. Figure 12.13 shows at which times and in which libraries, classes, and methods objects are allocated. It is possible to identify several time periods with very specific allocation behavior, which are in most cases separated by periods with no allocations. At the beginning there is a large number of allocations within a short period of time but at many different code positions. This pattern is caused by the static initializers, which run when a component is loaded. Until about 44.0 sec. the system components are initialized with object allocations mostly in the JDK library and in the ethernet device driver (net_3c905.jll). The next starts at about 45.3 sec. with many allocations in the components that have a name starting with net. These components implement the network protocols IP, UDP, BOOTP, and ARP. At about 48.7 sec. there are allocations in the network stack and JDK which are caused the execution of the BOOTP protocol (see the corresponding time in the event diagram of Figure 12.11). This is the point where the system learns its IP address. This network activity is followed by an activity in the file system components. An inspection of the test program's source code reveals that the initialization code waits for the initialization of the network protocols to complete before initializing the file system components and formatting the file system. The next period of heavy allocations in nearly all components starts at about 53.6 sec. As can be seen in the event diagram of Figure 12.11 the system starts to receive *mount* and *getattr* requests at this time.

Object creation and destruction. Figure 12.14 shows the lifetime of objects. The y-axis shows the instruction pointer, representing the position of the object allocation (equally distributed). The color relates to the object type (class). At one allocation point always the same type of object is created. At about 45.65 sec., 49.08 sec. and 53.86 sec. we can recognize a garbage collector run destroying many objects.

Object aging. We developed several visualizations to capture whether objects are long- or short-living and what kind of objects are more likely to have a long or short lifetime. Figure 12.15 shows the age of objects and where they are allocated. The time is measured in allocated memory, that means an object grows older when memory is allocated.The number of instances of a class with a certain age is encoded in the size of the dot. The red line is the average number of instances with the specific age.

Cache behavior. Figure 12.12displays the values of the Pentium III performance counters and the thread activity diagram of Figure 12.12 in the background. During this test the performance counters are configured to measure the number of lines evicted from the L2 cache (which should be equal to the number of lines loaded into L2 and therefore the number of L2 misses) and the number of L2 requests (which is equivalent to the number of L1 misses). Together with the thread activity diagram the performance counter diagram tells us that there is lot of cache activity (many accesses and misses) during the boot phase. Then the code has only few L1 misses. Most requests seem to be satisfied by L1. There is a four seconds interval (between about 49.5s and 53.5s) and where the line is absolutely horizontal. During this period the system is idle after the initialization of the NIC. The there are two kinds of activity with different characteristics: the NFS server processing the getattr request and the garbage collector. During GC there are more L2 misses than during request processing. This is a result of the copying GC algorithm that traverses all live objects and copies them to the second semispace. As the GC touched all live objects the chances are good that objects that are needed during request processing are already in the L2 cache and therefore the removed-lines-curve flattens. Interestingly the number-of-requests-curve has the opposite characteristics. During request processing there are more L1 misses per unit of time than during GC. This can be explained by a better spatial locality of the garbage collector. The GC inspects all reference fields of an object and the chances are good that they are all in the same cache line.

5 Summary

Very different applications demonstrated the feasibility of using type safety and the JX architecture as the foundation of a general-purpose operating system. All applications benefit from the abstractions that are provided by JX, like portals and services and memory objects. The performance of the applications is between 50% and 100% of similar applications running on Linux.

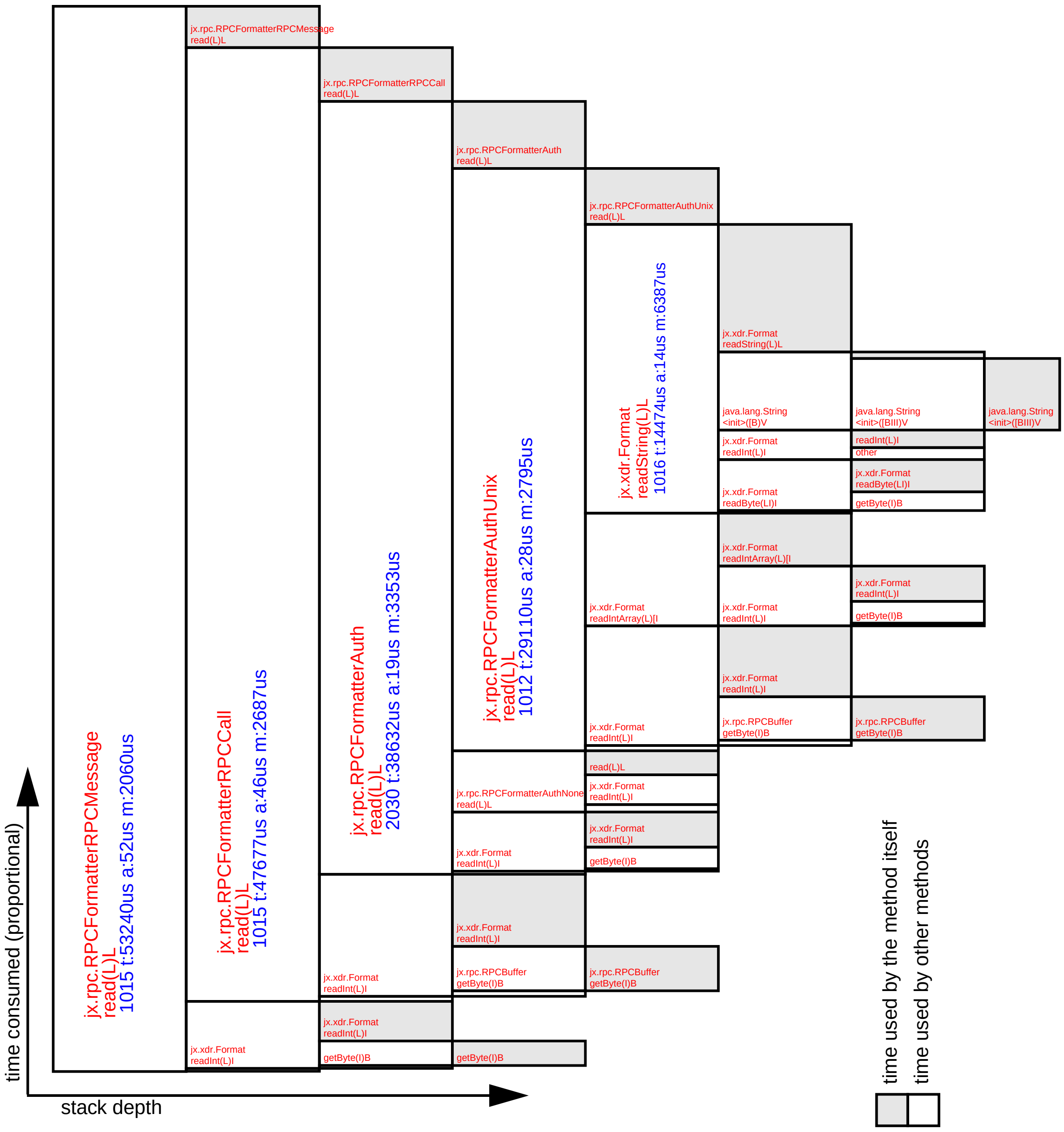
JX suffers a performance overhead because of the immaturity of the JX optimizer compared to today's C compilers and optimizers. We expect this problem to gradually disappear in future versions, because there are no optimization barriers in the JX architecture: components can be loaded privately in a domain and the translator has global information about all components loaded in a domain; no stack inspection is used, i.e. methods can be inlined; a domain can use it's own non-preemptive scheduler and can be restricted to one processor, i.e. memory accesses can be liberally reordered, as long as they only affect the own heap of a domain.

The analysis of the NFS server furthermore show that the NFS server creates a large number of objects and most of them are very short living. The garbage collector runs that are required to collect these objects introduce long pauses. One reason for the long execution time of the garbage collector is its bad locality and high number of L2 cache misses.

.

Figure 12.10: Method timing of RPC processing

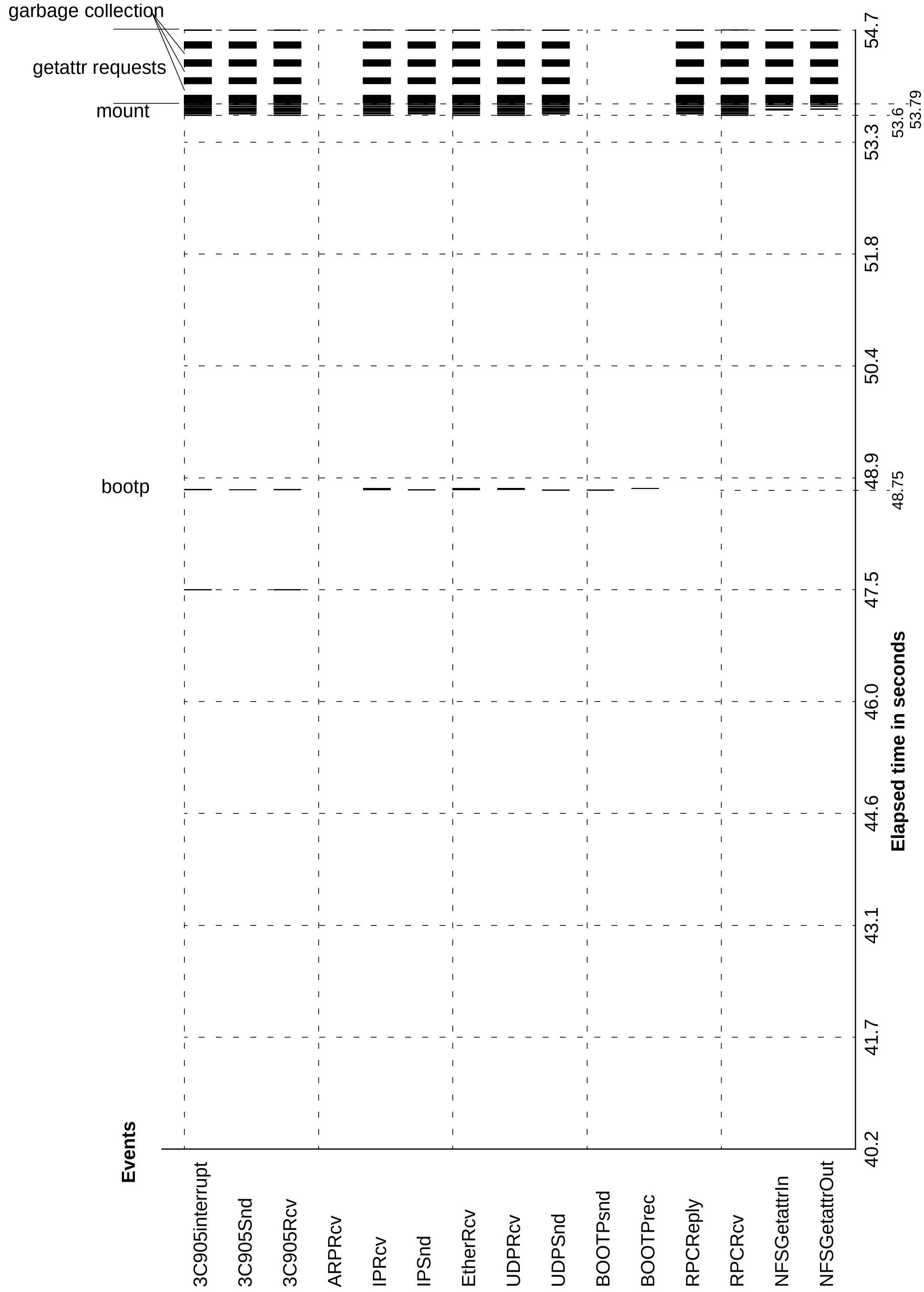
This figure displays the time distribution of RPC protocol processing. The gray boxes represent the time spent executing the method itself while the white boxes denote the time spent in invoked methods.



jx.rpc.RPCFormatterRPCMessage read(L)L
jx.rpc.RPCFormatterRPCMessage read(L)L 1015 t:53240us a:52us m:2060us
jx.rpc.RPCFormatterRPCCall read(L)L
jx.rpc.RPCFormatterRPCCall read(L)L 1015 t:47677us a:46us m:2687us
jx.xdr.Format readInt(L)I
jx.rpc.RPCFormatterAuth read(L)L
jx.rpc.RPCFormatterAuth read(L)L 2030 t:38632us a:19us m:3353us
jx.xdr.Format readInt(L)I
jx.xdr.Format readInt(L)I
getByte(I)B
jx.rpc.RPCFormatterAuthUnix read(L)L
jx.rpc.RPCFormatterAuthUnix read(L)L 1012 t:29110us a:28us m:2795us
jx.rpc.RPCFormatterAuthNone read(L)L
jx.xdr.Format readInt(L)I
jx.xdr.Format readInt(L)I
jx.rpc.RPCBuffer getByte(I)B
getByte(I)B
jx.xdr.Format readString(L)L 1016 t:14474us a:14us m:6387us
jx.xdr.Format readIntArray(L)[I
jx.xdr.Format readInt(L)I
read(L)L
jx.xdr.Format readInt(L)I
jx.xdr.Format readInt(L)I
getByte(I)B
jx.rpc.RPCBuffer getByte(I)B
jx.xdr.Format readString(L)L
java.lang.String <init>([B)V
jx.xdr.Format readInt(L)I
jx.xdr.Format readByte(L)I
jx.xdr.Format readIntArray(L)[I
jx.xdr.Format readInt(L)I
jx.xdr.Format readInt(L)I
jx.rpc.RPCBuffer getByte(I)B
java.lang.String <init>([BIII)V
readInt(L)I
other
jx.xdr.Format readByte(L)I
getByte(I)B
jx.xdr.Format readInt(L)I
getByte(I)B
jx.rpc.RPCBuffer getByte(I)B
java.lang.String <init>([BIII)V
time consumed (proportional)
stack depth
▨ time used by the method itself
□ time used by other methods

Figure 12.11: Application-generated events

This diagram shows the time of occurrence of application generated events.



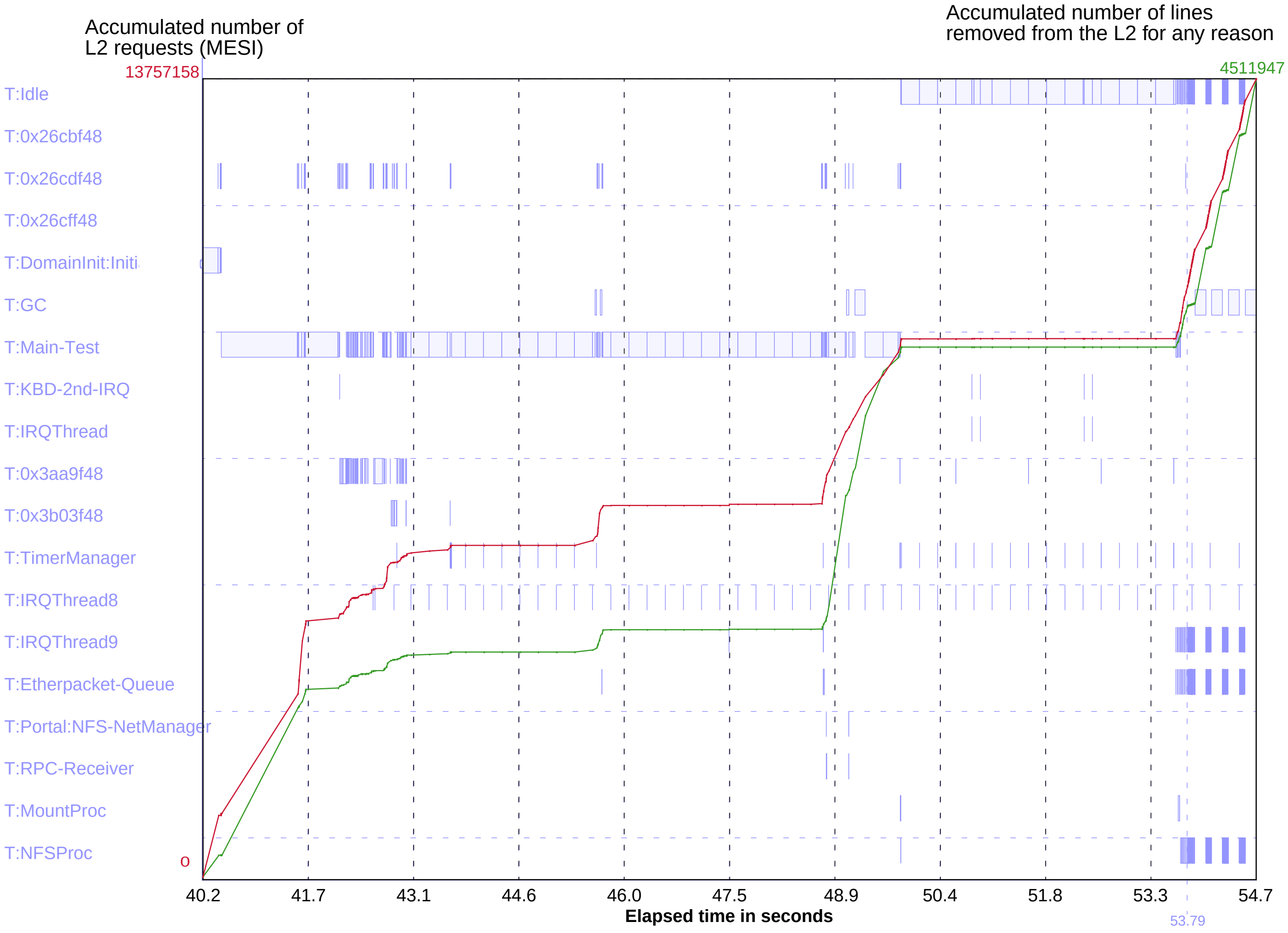


Figure 12.12: Thread activity diagram and P6 performance counters

This diagram shows thread activity and cache activity. The upper one of the two graphs is the number of L2 requests (maximum value 13757158), the lower one is the number of lines removed from the L2 (maximum value 4511947).

Figure 12.13: Object allocations over time

This figure shows the time and code position of object allocations during system startup and the NFS rate benchmark. Several patterns are clearly visible. A detailed explanation of these patterns is given in Section 4.3.

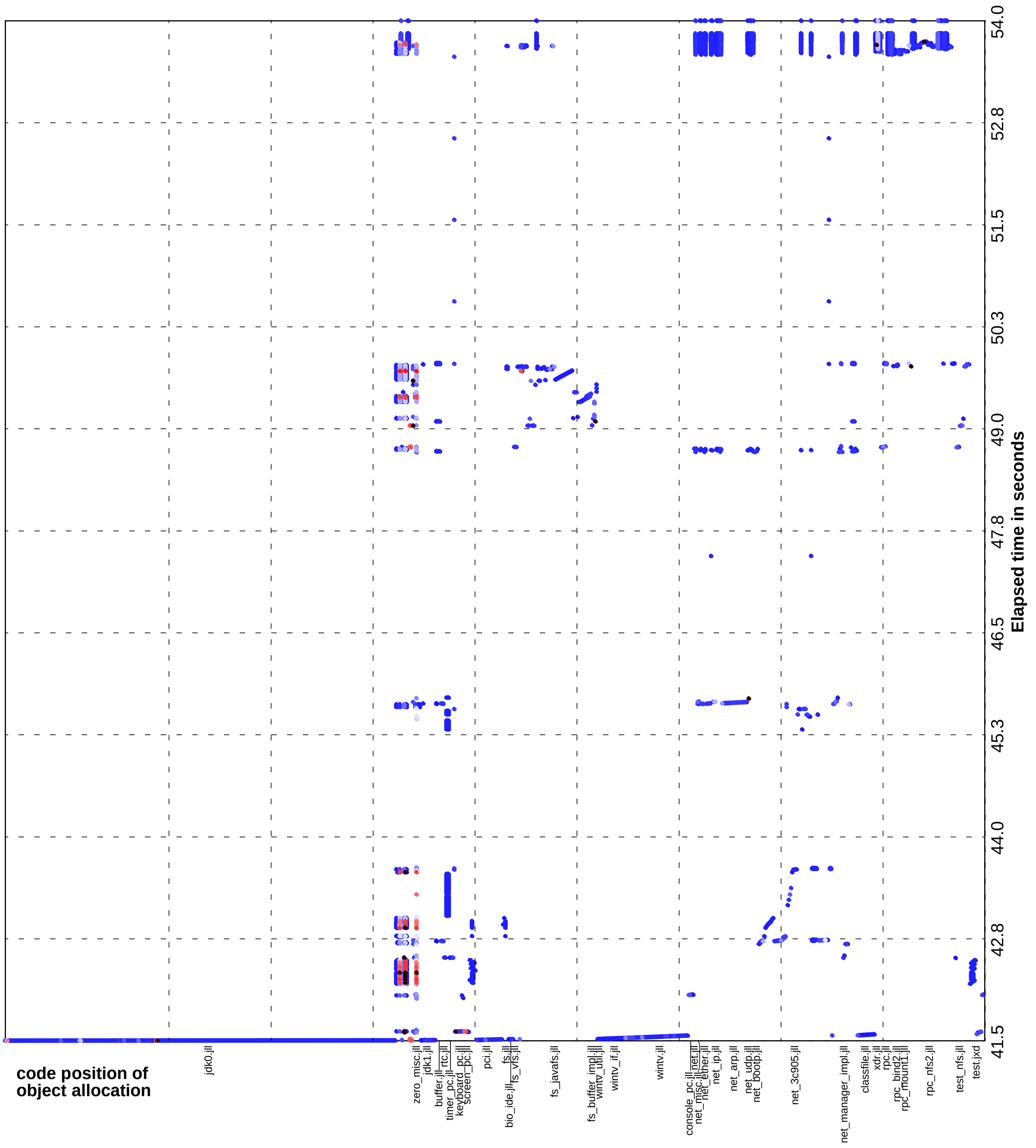
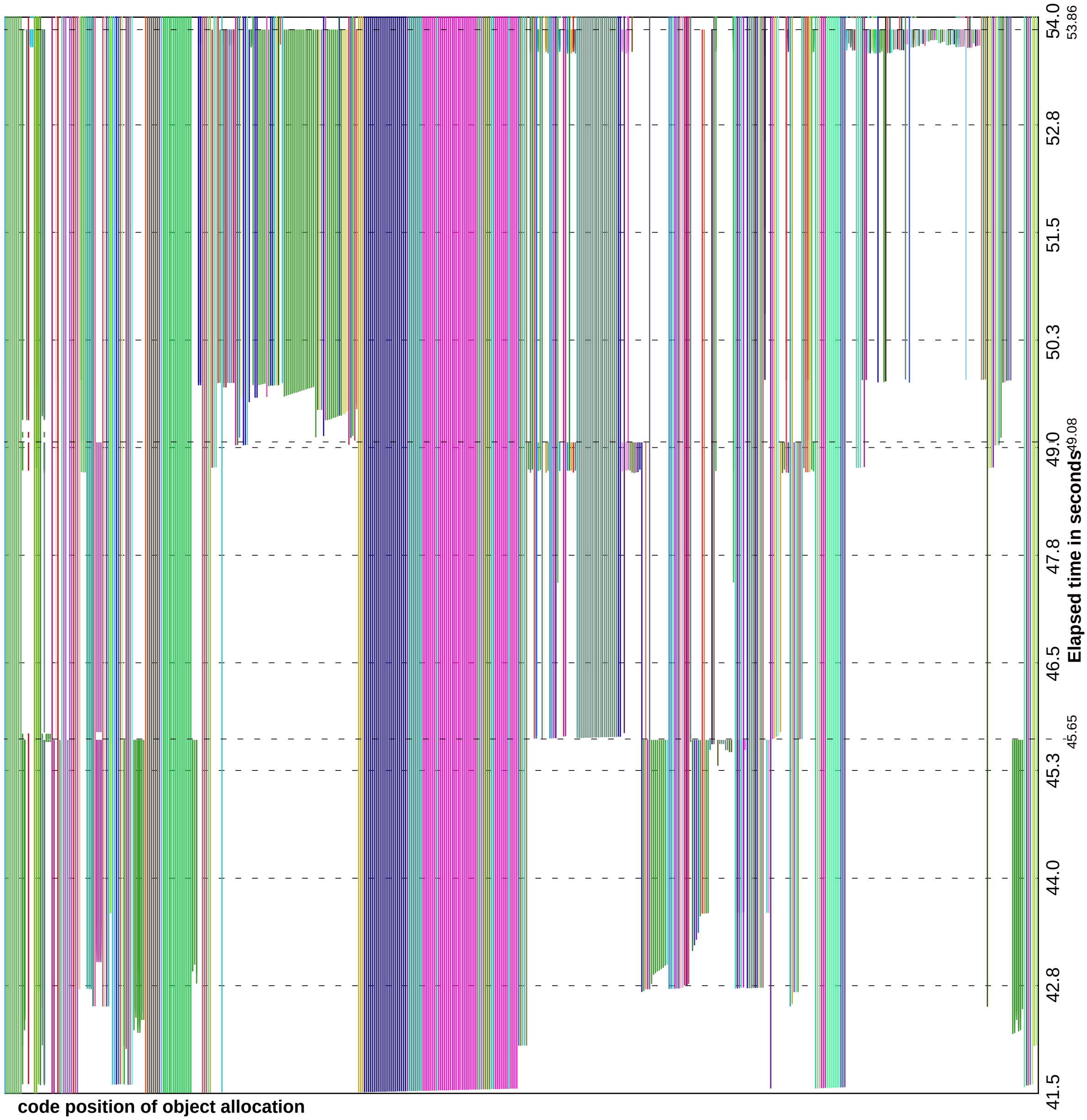


Figure 12.14: Object lifetimes

This figure visualizes the lifetime of objects. There is a line for each object in the system. The line starts when the object is created and end when it is garbage collected. The y-position of the line is the code position of the allocation statement.



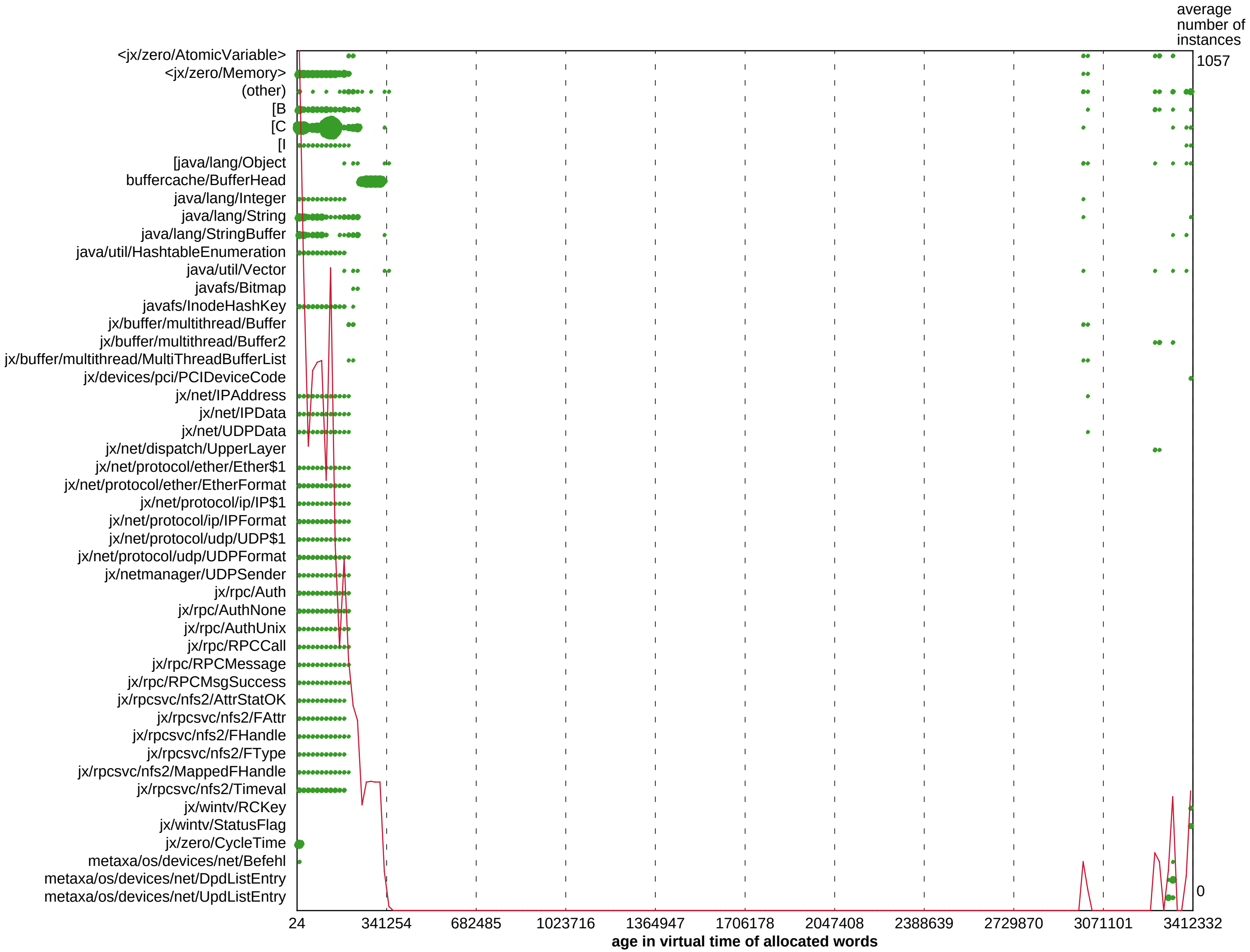


Figure 12.15: Age distribution of class instances

This diagram visualizes the ages of objects sorted by their type.The size of the dots corresponds to the size of the object. The graph shows the average number of instances of a certain age.

CHAPTER 13 *Conclusions*

1 Contribution

This thesis describes a novel approach to build operating systems. The contribution consists of

- an analysis of operating system properties required by today's and future applications
- an analysis of shortcomings and useful approaches of previous systems
- the design of an operating system that is build around a protection mechanism previously only used for language runtimes
- the implementation of a prototype that demonstrates the feasibility of the design
- a detailed performance analysis of the prototype with respect to its functional properties and performance

2 Meeting the objectives

The system meets the objectives stated in Chapter 1:

- **Robustness, Security, Reliability**. The amount of unsafe C and assembler code is reduced to a minimum. The complete operating system, including the device drivers, is build from typesafe code. This simplifies the system and makes it more robust and reliable. A security architecture was developed that allows the precise control of information and control flows within the system.
- **Configurability, Reusability**. With the exception of the microkernel all code is organized in components which can be collocated in a single protection domain or dislocated in separate domains without touching the component code. This reusability across configurations enables to adapt the system for its intended use, which may be, for example, an embedded system, desktop workstation, or server.
- **Extensibility, Fine-grained Protection**. Operating system code and application code is separated in protection domains with strong isolation between the domains. A domain has a low memory footprint which allows to extend the system by creating new domains.
- **Performance**. Performance is between 50% and 100% of monolithic UNIX performance for computational-intensive operating system tasks. The difference is even smaller when I/O from a real device is involved.

- **Uniformity**. All virtual resources, such as files, sockets, windows, and database connections are reduced to a single unifying abstraction: the portal.The portal is a lightweight capability-like construct to access a service. This simple but powerful architecture allows the introduction of new virtual resources and facilitates the adaptation to a new environment, such as a distributed system.
- **Productivity**. The existence of the prototype demonstrates that using type safety and keeping the architecture simple it is possible for a small group of people to build a fairly complete OS in short period of time.

3 Design summary

The JX design was inspired by previous operating system architectures. It’s shared-nothing principle is similar to the vertically structured architecture of the Nemesis OS [114]; orthogonality of protection and modularity was the main objective of Lipto [57]; application-level resource management was emphasized in many projects, the most prominent being the Exokernel [61].

In the spirit of a microkernel the JX system has five central abstractions: domain, thread, component, portal, and service. Additional abstractions are provided as portals to services of DomainZero or fast portals. Examples for fast portals are memory objects and atomic variables. The JX system has the following unique properties:

Type-safe protection. Protection is based on the type-safe instruction set. No protection hardware, such as an MMU is necessary, which allows JX to create protection domains even on embedded devices. Communication between domains is very efficient as there is no need to switch to another hardware context.

Independent heaps. Each domain has its own heap that is completely independent from the heaps of other domains. This is in contrast with namespace-based domains of traditional Java systems and allows independent garbage collection runs, domain-specific garbage collection implementations, and domain termination with immediate resource reclamation.

Multiple GC implementations. The performance of garbage collection largely depends on the characteristic of the application. Each domain can be created with one of several GC implementations that are provided by the kernel.

Independent scheduling. Threads of one domain can be scheduled independent of other domains. It is possible to use domain-specific schedulers that are supplied in the type-safe instruction set.

Security and containment. Domains are isolated and can communicate only via dedicated channels by using portals. JX security system can control the propagation of all portals as well as their use. The security system that contains the security policy is separated from the domains and can be changed without changing the application domains.

Automatic management of memory areas. JX provides a special abstraction, called memory portals, to cope with large memory or special memory areas, such as memory-mapped device registers. Memory portals can be shared between domains and are garbage collected.

Transparent code sharing. Immutable state can be shared between domains. This includes compiled code and string constants. Mutable state, such as static variables, is created per domain which make code sharing completely transparent.

Automatic service creation. A service is automatically created when a service object is passed to another domain. The other domain receives a portal to the service.

Flexible configuration. The protection domain structure is independent of the component structure. Components can be placed into the same or in separate domains without changing them.

Termination with immediate resource reclamation. Except device driver domains all domains can be terminated and the resources that are allocated to the domain can be released immediately.

Robust device drivers. All device drivers are deployed in the type-safe instruction set and benefit from the robustness that is provided by type safety. Robustness is further supported by special portal interfaces that allow drivers to access device registers and interrupt handler threads to synchronize with other threads without disabling interrupts.

Independent class libraries. Each domain can have its own implementation of class libraries. This also applies to the classes of the Java Development Kit.

4 Future work

The JX architecture can be used as a platform to build

- **general-purpose systems** that allows a user to perform everyday tasks, such as editing texts or working with spreadsheets. The window manager will probably be the central component of such a system.
- **dedicated systems** that perform a very specific task, for example a file server appliance. They could be build by reusing components from general-purpose systems and configuring them for the specific task, for example by collocating them in a single domain
- **embedded systems** that need to run untrusted code but run on hardware that lacks protection mechanisms. They can use the fine-grained protection provided by JX domains.
- **distributed systems** that go far beyond the concept of a single-system image that is the goal of UNIX-based distributed systems. Moving actively executing domains through the network could be used for load distribution or fault tolerance. We made the first steps to this goal by implementing an active storage system [74] that allows to move a file system agent to the file server host.

5 Final remarks

I hope that the JX system will increase the awareness of the operating systems community to the many benefits of moving to a type-based approach of system building. By providing a running prototype and demonstrating its functionality and performance I should have eliminated any doubts about the feasibility of such an endeavor.

Bibliography

[1] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. In *ACM Transactions on Computer Systems, 2(1)*, pp. 39-59, Feb. 1984. (page 17)

[2] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. of the ACM SIGPLAN 96ConferenceonProgrammingLanguageDesignandImplementation(PLDI)'*, pp. 127-136, May 1996. (page 10)

[3] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Conference on Programming Language Design and Implementation 98'*, pp. 269-279, 1998. (page 89)

[4] B. Alpern, A. Cocchi, S. J. Fink, D. P. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: invokeinterface Considered Harmless. In *OOPSLA 01'*, Oct. 2001. (page 68)

[5] M. Alt. *Ein Bytecode-Verifier zur Verifikation von Betriebssystemkomponenten*. Diplomarbeit, available as DA-I4-2001-10 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, July 2001. (page 17)

[6] M. Anderson, R. Pose, and C. S. Wallace. A password-capability system. In *The Computer Journal, 29*, pp. 1-8, 1986. (page 119)

[7] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. on Computer Systems, 10(1)*, pp. 53-79, Feb. 1992. (page 7) (page 78)

[8] Apple Computer. *Inside Max OS X: Kernel Programming*. May 2002. (page 6)

[9] J. Q. Arnold. Shared libraries on UNIX System V. In *Summer USENIX Conference*, 1986. (page 60)

[10] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of 4th Symposium on Operating Systems Design & Implementation*, Oct. 2000. (page 52) (page 37) (page 138)

[11] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *2000 USENIX Annual Technical Conference*, June 2000. (page 52)

[12] H. G. Baker. List processing in real time on a serial computer. In *Communications of the ACM, 21(4)*, pp. 280-294, Apr. 1978. (page 105) (page 150)

[13] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Symposium on Operating Systems Design and Implementation 99'*, 1999. (page 12) (page 30)

[14] J. S. Barrera. A Fast Mach Network IPC Implementation. In *Proc. of the USENIX Mach Symposium*, pp. 1-12, Nov. 1991. (page 6)

[15] P. Bernadat. *An evaluation of Windows NT with respect to real-time capabilities in the context of the QUITE project*. Technical Report, The Open Group, May 1999. (page 102)

[16] P. Bernadat, D. Lambright, and F. Travostino. Towards a Resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pp. 101-111, Dec. 1998. (page 137)

[17] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Operating Systems Review, 23(5)*, pp. 102-113, Dec. 1989. (page 37)

[18] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th Symposium on Operating System Principles*, pp. 267-284, Dec. 1995. (page 10) (page 14)

[19] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 223-233, Sep. 1992. (page 81) (page 155)

[20] D. Beuche, A. Guerrouat, H. Papajewski, W. Schrder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC99)'*, May 1999. (page 13)

[21] A. Bhide, E. N. Elnozahy, and S. P. Morgan. A highly available network file server. In *Proc. of the USENIX Winter Conference*, pp. 199-205, Jan. 1991. (page 2)

[22] W. Binder, J. Hulaas, A. Villazon, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA01)'*, Oct. 2001. (page 138)

[23] R. Bisbey II and D. Hollingworth. *Protection Analysis: Final Report*. Technical Report ISI/SR-78-13, University of Southern California/Information Sciences Institute, Marina Del Rey, CA, May 1978. (page 93)

[24] W. E. Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proc. of the 7th DoD/NBS Computer Security Conference*, pp. 291-293, Sep. 1984. (page 119)

[25] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *ACM Transactions on Computer Systems, 14(1)*, pp. 80-107, 1996. (page 9)

[26] K. Brockschmidt. Inside OLE, Second Edition. Microsoft Press, Redmond WA, 1995. (page 13)

[27] C. Bryce and J. Vitek. The JavaSeal Mobile Agent Kernel.In *Proc. of the 1st International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASAMA99)'*, pp. 176-189, ACM Press, May 1999. (page 138)

[28] CERT/CC. *VU#16532: BIND T_NXT record processing may cause buffer overflow* . Nov. 1999. (page 115)

[29] CERT/CC. *VU#5648: Buffer Overflows in various email clients*. 1998. (page 115)

[30] CERT/CC. *VU#970472: Network Time Protocol ([x]ntpd) daemon contains buffer overflow in ntp_control:ctl_getitem() function*. Apr. 2001. (page 115)

[31] CERT/CC. *VU#745371: Multiple vendor telnet daemons vulnerable to buffer overflow via crafted protocol options*. July 2001. (page 115)

[32] CERT/CC. *VU#28934: Sun Solaris sadmind buffer overflow in amsl_verify when requesting NETMGT_PROC_SERVICE*. Dec. 1999. (page 115)

[33] CERT/CC. *VU#952336: Microsoft Index Server/Indexing Service used by IIS 4.0/5.0 contains unchecked buffer used when encoding double-byte characters*. June 2001. (page 115)

[34] CERT/CC. *VU#29823: Format string input validation error in wu-ftpd site_exec() function*. June 2000. (page 115)

[35] CERT/CC. *VU#789543: IIS decodes filenames superfluously after applying security checks*. May 2001. (page 115)

[36] CERT/CC. *VU#17215: SGI systems may execute commands embedded in mail messages*. Apr. 1998. (page 115)

[37] CERT/CC. *VU#945216: SSH CRC32 attack detection code contains remote integer overflow*. Feb. 2001. (page 115)

[38] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices : An Object-Oriented System in C++. In *Communications of the ACM, 36(9)*, pp. 117-126, Sep. 1993. (page 2)

[39] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. In *ACM Trans. on Computer Systems, 12(4)*, pp. 271-307, Nov. 1994. (page 7) (page 119)

[40] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating System Principles 01'*, 2001. (page 101) (page 127) (page 127)

[41] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. of the IEEE Symposium on Security and Privacy*, May 1987. (page 118)

[42] M. Condict, D. Bolinger, E. McManus, D. Mitchell, and S. Lewontin. *Microkernel modularity with integrated kernel performance*. Technical Report , OSF Research Institute, Cambridge, MA, Apr. 1994. (page 6) (page 13)

[43] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, Jan. 2000. (page 115)

[44] G. Czajkowski and L. Daynes. Multitasking without Compromise: A Virtual Machine Evolution. In *Proc. of the OOPSLA01'*, pp. 125-138, Oct. 2001. (page 138)

[45] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications 98'*, pp. 21-35, ACM Press, 1998. (page 117) (page 138)

[46] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in Multics. In *Communications of the ACM, 11(5)*, pp. 306-312, May 1968. (page 7)

[47] I. Dedinski. *Design und Implementierung einer hochperformanten Datenbank f¸r das Java-Betriebssystem JX (Design and implementation of a high-performance data base system for the Java operating system JX)*. Studienarbeit, available as SA-I4-2002-01 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Jan. 2002. (page 17) (page 148) (page 148) (page 151)

[48] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *Communications of the ACM, 9(3)*, pp. 143-155, Mar. 1966. (page 20) (page 119) (page 119) (page 124)

[49] Department of Defense. *Trusted computer system evaluation criteria (Orange Book)*. DOD 5200.28-STD, Dec. 1985. (page 117) (page 128)

[50] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. ACM Press, 1983. (page 10)

[51] D. Dillenberger, R. Bordawekar, C. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. Oliver, F. Samuel, and R. St. John. Building a Java virtual machine for server applications: The JVM on OS/390. In *IBM Systems Journal, 39(1)*, pp. 194-210, 2000. (page 138)

[52] L. v. Doorn. A Secure Java Virtual Machine . In *Proc. of the 9th USENIX Security Symposium* , pp. 19-34, Aug. 2000. (page 134) (page 138) (page 142)

[53] L. v. Doorn. *The Design and Application of an Extensible Operating System*. Ph.D. thesis, 2001. (page 138)

[54] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. In *Bell Labs Technical Journal, 2(1)*, pp. 5-18, 1997. (page 10)

[55] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Thirteenth ACM Symposium on Operating Systems Principles*, pp. 122-136, Oct. 1991. (page 6)

[56] R. Draves and S. Cutshall. *Unifying the User and Kernel Environments*. Technical Report MSR-TR-97-10,, Microsoft Research , Mar. 1997. (page 2)

[57] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proc. of Twelfth International Conference on Distributed Computing Systems*, pp. 512-520, 1992. (page 13) (page 62) (page 170) (page 190)

[58] E.W. Dijkstra. Structure of the ‘THE‘-Multiprogramming System. In *Communications of the ACM, 11(5)*, pp. 341 - 346, May 1968. (page 5) (page 13)

[59] ECMA. *Common language Infrastructure, Part 3: IL Instruction set specification* . May 2000. (page 10)

[60] D. R. Engler. *The exokernel operating system architecture*. Ph.D. thesis, Massachusetts Institute of Technology, Oct. 1998. (page 7)

[61] D. Engler, F. Kaashoek, and J. O‘Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th Symposium on Operating System Principles*, pp. 251-266, Dec. 1995. (page 7) (page 18) (page 170) (page 190)

[62] R. S. Fabry. Capability-based addressing . In *Communications of the ACM, 17(7)*, pp. 403-412 , July 1974. (page 119)

[63] M. Felser. *Integration einer Mehrprozessorunterst¸tzung in das Java-Betriebssystem JX (Integration of Multiprocessor-Support into the Java Operating System JX)*. Diplomarbeit, available as DA-I4-2001-09 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, July 2001. (page 17)

[64] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. of the 15th National Computer Security Conference*, pp. 554-563, Oct. 1992. (page 124)

[65] C. Flügel. *Entwurf und Implementierung eines RDP-Clients f¸r das Java-Betriebssystem JX (Design and implementation of an RDP client for the Java operating system JX)*. Studienarbeit, available as SA-I4-2002-11, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, C. Wawersich, and M. Felser, July 2002. (page 17)

[66] Ford Aerospace. *Secure Minicomputer Operating System (KSOS) Executive Summary: Phase I: Design of the Department of Defense Kernelized Secure Operating System*. Technical Report WDL-781, Palo Alto, CA, 1978. (page 10) (page 118)

[67] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *16th SOSP*, Oct. 1997. (page 14) (page 18)

[68] B. Frantz. KeyKOS - a secure, high-performance environment for S/370. In *Proc. of SHARE 70*, pp. 465-471, Feb. 1988. (page 119)

[69] T. Fraser and L. Badger. Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE. In *Proc. of the 1998 IEEE Symposium on Security and Privacy (S&P 1998)*, 1998. (page 130)

[70] E. Gabber and E. Shriver. *LetsputNetAppandCacheFlowoutofbusiness!*'In *SIGOPS European Workshop*, 2000. (page 2)

[71] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *USENIX 1999 Annual Technical Conference*, pp. 267-282 , June 1999. (page 2) (page 13)

[72] R. Gingell, M. Lee, X. Dang, and M. Weeks. Shared libraries in SunOS. In *USENIX Summer Conference*, pp. 375-390, 1987. (page 60)

[73] M. Golm. *Design and Implementation of a Meta-Architecture for Java*. Master‘s thesis, University of Erlangen, 1997. (page 17) (page 18) (page 142)

[74] M. Golm, C. Wawersich, M. Felser, and J. Kleinöder. ActiveStorage: A Storage Server with Mobile Code Engine. In *GI Fachgruppentreffen Betriebssysteme 2001*, Nov. 2001. (page 171) (page 192)

[75] M. Golm, J. Kleinoeder, and F. Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems*, pp. 1-6, May 2001. (page 127)

[76] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proc. of USENIX Summer Conference*, pp. 87-95, June 1990. (page 6)

[77] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Aug. 1996. (page 64) (page 129)

[78] A. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. In *Communications of the ACM, 19(5)*, pp. 266-272, May 1976. (page 5) (page 13)

[79] H. Haertig, W. E. Kuehnhauser, and O. C. Kowalski. The BirliX Security Architecture. In *Journal of Computer Security, 2(1)*, pp. 5-21, IOS Press, 1993. (page 7)

[80] G. Hamilton and P. Kougioris. The Spring Nucleus: a Micro-kernel for objects. In *Proc. of Usenix Summer Conference*, pp. 147-159, June 1994. (page 2)

[81] P. B. Hansen. The nucleus of a multiprogramming system. In *Communications of the ACM, 13(4)*, pp. 238-241, Apr. 1970. (page 6) (page 12)

[82] N. Hardy. The confused deputy. In *Operating Systems Review, 22(4)*, pp. 36-38, Oct. 1988. (page 119) (page 124) (page 131)

[83] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Experiences building a communication-oriented JavaOS. In *Software--Practice and Experience, 30(10)*, Apr. 2000. (page 12)

[84] F. Hauck. Typen, Klassen und Vererbung in verteilten objektorientierten Systemen. Sep. 1994. (page 64)

[85] C. Hawblitzel and T. v. Eicken. *Tasks and Revocation for Java (unpublished draft)*. Nov. 1999. (page 138)

[86] C. Hawblitzel. Adding Operating System Structure to Language-Based Protection. Ph.D. thesis, Cornell University, June 2000. (page 138) (page 138) (page 140)

[87] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. v. Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, pp. 259-270, June 1998. (page 52) (page 138)

[88] A. Heiduk. *Entwurf und Implementierung eines Framegrabber-Treibers f¸r das Java-Betriebssystem JX (Design and Implementation of a Framegrabber Driver for the Java Operating System JX)*. Diplomarbeit, available as DA-I4-2001-06 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Apr. 2001. (page 17) (page 107)

[89] G. Heiser, K. Elphinstone, S. Russel, and J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. In *17th Australiasion Computer Science Conference*, pp. 271-280, Jan. 1994. (page 119)

[90] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. In *Software: Practice and Experience, 28(9)*, pp. 901-928, Aug. 1998. (page 7)

[91] J. Helander and A. Forin. MMLite: A Highly Componentized System Architecture. In *Eight ACM SIGOPS European Workshop*, Sep. 1998. (page 13) (page 14)

[92] D. Hitz, J. Lau, and M. Malcom. Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers. In *Proc. of Winter 1990 USENIX Technical Conference*, pp. 285-295, Jan. 1990. (page 2)

[93] D. Hitz, J. Lau, and M. Malcolm. *File System Design for an NFS File Server Appliance*. Technical Report TR3002, Network Appliance, Inc, Mar. 1995. (page 2)

[94] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of Winter 1994 USENIX Conference*, pp. 235-246 , Jan. 1994. (page 2)

[95] U. Hoelzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *Proc. of European Conference on Object-Oriented Programming 93'*, Lecture Notes in Computer Science 512, July 1993. (page 64)

[96] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm. Position Summary: Supporting Hot-Swappable Components for System Software. In *HOTOS 2001*, 2001. (page 3)

[97] N. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. In *IEEE Transactions on Software Engineering, 17(1)*, pp. 64-76, 1991. (page 13)

[98] R. K. Johnsson and J. D. Wick. An overview of the Mesa processor architecture. In *ACM Sigplan Notices, 7(4)*, pp. 20-29, Apr. 1982. (page 9)

[99] M. B. Jones and R. F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 67-77, Oct. 1986. (page 6)

[100] R. Jones and R. Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester (GB), 1996. (page 84)

[101] R. Kain and C. Landwehr. On access checking in capability-based systems. In *IEEE Transactions on Software Engineering, 13(2)*, pp. 202-207, Feb. 1987. (page 119)

[102] R. Keller and U. Hölzle. Binary Component Adaptation. In *Proc. of European Conference on Object-Oriented Programming 98'*, pp. 307-329, 1998. (page 64)

[103] R. A. Kemmerer. Shared resource matrix methodology: A practical approach to identifying covert channels. In *ACM Trans. on Computer Systems, 1(3)*, pp. 256-277, Aug. 1983. (page 133)

[104] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The Need for Customizable Operating Systems. In *Fourth Workshop on Workstation Operating Systems*, Aug. 1993. (page 2)

[105] F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. Ballesteros. 2k: A reflective, component-based operating system for rapidly changing environments. In *ECOOP98WorkshoponReflectiveObject-OrientedProgrammingandSystems'*, 1998. (page 13)

[106] H. Kopp. *Design und Implementierung eines maschinenunabh‰ngigen Just-in-time-Compilers f¸r Java (Design and Implementation of a machine-independent Just-in-time Compiler for Java)* . Diplomarbeit, available as DA-I4-1998-04, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Sep. 1998. (page 17)

[107] L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS Interface for Active Routers. In *IEEE Journal on Selected Areas in Communications*, 2001. (page 12)

[108] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. Report on the programming language Euclid. In *ACM SIGPLAN Notices, 12(2)*, Feb. 1977. (page 10)

[109] B. Lampson and R. Sproull. An open operating system for a single-user machine. In *Proc. of the Seventh ACM Symposium on Operating Systems Principles*, pp. 98-105, Dec. 1979. (page 14)

[110] B. W. Lampson. *Personal Distributed Computing: The Alto and Ethernet Software*. pp. 291-344, Addison-Wesley, 1988. (page 9)

[111] B. W. Lampson. A Note on the Confinement Problem. In *Communications of the ACM, 16(10)*, pp. 613-615, Oct. 1973. (page 122) (page 122)

[112] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A Security Model for Military Message Systems. In *ACM Trans. on Computer Systems, 2(3)*, pp. 198-222, Aug. 1984. (page 118)

[113] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. In *ACM Operating Systems Review, 13(2)*, pp. 3-19, Apr. 1979. (page 13)

[114] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. In *IEEE Journal on Selected Areas in Communications, 14(7)*, pp. 1280-1297, Sep. 1996. (page 170) (page 190)

[115] R. Levin. Policy/Mechanism Separation in HYDRA. In *5th Symposium on Operating System Principles*, pp. 132-140, Nov. 1975. (page 6)

[116] J. Liedtke. On micro-kernel construction. In *Proc. of the 15th ACM Symposium on OS Principles*, pp. 237-250, Dec. 1995. (page 51)

[117] J. Liedtke. Towards Real u-Kernels. In *CACM , 39(9)*, 1996. (page 7)

[118] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pp. 294-305, Springer Verlag, 1992. (page 7)

[119] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sep. 1996. (page 129)

[120] P. A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *21st National Information Systems Security Conference*, pp. 303-314, Oct. 1998. (page 115)

[121] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Usenix 2001 Freenix Track*, 2001. (page 122) (page 130)

[122] C. Maeda. A Metaobject Protocol for Accessing File Systems. In *Proc. of the 2nd International Symposium on Object Technologies for Advanced Software*, Mar. 1996. (page 14)

[123] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications 87'*, pp. 147-155, 1997. (page 142)

[124] J. McHugh. Covert Channel Analysis (Chapter VIII). In *Navy Handbook for the Computer Security Certification of Trusted Systems*. 1995. (page 133)

[125] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley , May 1996. (page 101)

[126] A. Meyer and L. Seawright. A Virtual Machine TimeSharing System. In *IBM Systems Journal, 9(3)*, pp. 199-218, 1970. (page 8) (page 9)

[127] M. Meyerhöfer. *Design und Implementierung eines Ethernet-Treibers und TCP/IP-Protokolls fuer das Java-Betriebssystem JX*. Studienarbeit, available as SA-I4-2000-16 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Oct. 2000. (page 17)

[128] M. Michael and M. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. In *Journal of Parallel and Distributed Computing, 54(2)*, pp. 162-182, 1998. (page 81) (page 155)

[129] Microsoft. Windows Driver Development Kit. 2001. (page 127)

[130] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proc. of the 5th USENIX Security Symposium* , June 1995. (page 118)

[131] D. A. Moon. Symbolics Architecture. In *IEEE Computer, 20(1)*, pp. 43-52, IEEE, Jan. 1987. (page 119)

[132] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Conference Record 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 85-97, 1998. (page 10) (page 132)

[133] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of the Second Usenix Symposium on Operating System Design and Implementation*, pp. 153-168, 1996. (page 12) (page 81) (page 155)

[134] D. Mosberger, P. Druschel, and L. L. Peterson. Implementing Atomic Sequences on Uniprocessors Using Rollforward. In *Software---Practice and Experience, 26(1)*, pp. 1-23, Jan. 1996. (page 12) (page 81) (page 155)

[135] G. C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106-119, Jan. 1997. (page 8)

[136] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Symposium on Operating Systems Design and Implementation 96'*, 1996. (page 8)

[137] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications . In *Sixteenth ACM Symposium on Operating Systems Principles*, Oct. 1997. (page 3)

[138] J. Obernolte. *Entwurf und Implementierung eines Windowmanagers f¸r das Java-Betriebssystem JX (Design and implementation of a window manager for the Java operating system JX)*. Studienarbeit, available as SA-I4-2002-04, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Feb. 2002. (page 17) (page 146)

[139] Oracle. *Delivering the Promise of Internet Computing: Integrating Java With Oracle8i*. Technical Report An Oracle Technical White Paper, Apr. 1999. (page 138) (page 141)

[140] E. I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, Inc., New York, 1973. (page 119)

[141] P.A. Karger and J.C. Wray. Storage channels in disk arm optimization. In *Proc. of the 1991 IEEE Symposium on Security and Privacy*, pp. 52-61, May 1991. (page 133)

[142] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proc. of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 201-212, Oct. 1996. (page 14)

[143] D. Parnas. On a "buzzword": Hierarchical structure. In J. Rosenfeld (ed.) *Information Processing 74, volume Software*. In *Proc. of IFIP (International Federation for Information Processing) Congress 74*, pp. 336-339, North-Holland, Amsterdam, 1974. (page 5) (page 13)

[144] G. Pernul, W. Winiwarter, and A. M. Tjoa. The Entity-Relationship Model for Multilevel Security. In *12th Int. Conference on the Entity-Relationship Approach*, LNCS 823, Springer Verlag, Dec. 1993. (page 132)

[145] Z. Qian. *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines*. Technical Report, Bremen Institute for Safe Systems (BISS), FB3 Informatik, Universitat Bremen, 2000. (page 11)

[146] S. Rajunas, N. Hardy, A. Bomberger, W. Frantz, and C. Landau. Security in KeyKOS. In *Proc. of the 1986 IEEE Symposium on Security and Privacy*, Apr. 1986. (page 134)

[147] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. In *Communications of the ACM, 23(2)*, pp. 81-92, ACM Press, New York, NY, USA , Feb. 1980. (page 9) (page 138)

[148] D. Rosenberg. Bringing Java to the Enterprise: Oracle on Its Java Server Strategy. In *IEEE Internet Computing, 2(2)*, pp. 52-59 , Mar. 1998. (page 138) (page 142)

[149] A. Rudys, J. Clements, and D. S. Wallach. Termination in Language-based Systems. In *Proc. of the Network and Distributed Systems Security Symposium*, Feb. 2001. (page 29)

[150] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. In *ACM Transactions on Programming Languages and Systems, 21(6)*, pp. 1196-1250, Nov. 1999. (page 11)

[151] A. Sabry and S. Fickas. *Java Security in Parallel Universes*. Technical Report CIS-TR-98-03, Dept. of Comp. and Inform. Science, Univ. of Oregon, 1998. (page 132)

[152] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE , 63(9)*, pp. 1278-1308 , Sep. 1975. (page 115) (page 117)

[153] V. Saraswat. *Java is not type-safe*. Technical Report, AT&T Research, 1997. (page 11)

[154] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System* . Addison Wesley Longman, 1999. (page 102) (page 138)

[155] Secure Computing Corporation. *DTOS General System Security and Assurability Assessment Report*. 1997. (page 118) (page 118)

[156] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *2nd Symposium on Operating Systems Design and Implementation*, 1996. (page 12)

[157] J. S. Shapiro, J. M. Smith, and D. J. Farber. *EROS: a fast capability system*. In *Symposium on Operating Systems Principles*, pp. 170-185, 1999. (page 119)

[158] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *ICDCS 1986*, pp. 198-204, 1986. (page 13)

[159] O. Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *ACM Sigplan International Conference on Functional Programming (ICFP99)'*, Sep. 1999. (page 81) (page 155)

[160] S. M. Silver. *Implementation and Analysis of Software Based Fault Isolation*. Technical Report PCS-TR96-287, Dartmouth College, June 1996. (page 8)

[161] C. Small. A Tool for Constructing Safe Extensible C++ Systems. In *Proc. of the 3rd USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1997. (page 8) (page 15)

[162] R. E. Smith. *Cost Profile of a Highly Assured, Secure Operating System*. Sep. 1999. (page 134)

[163] D. Solomon and M. Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, Sep. 2000. (page 30) (page 102)

[164] O. Spatscheck. *Escort: Securing Scout Paths*. Ph.D. thesis, University of Arizona, 1999. (page 12)

[165] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of the 8th USENIX Security Symposium* , Aug. 1999. (page 118) (page 121) (page 122)

[166] St. Pemberton and M. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, Chichester, UK, 1982. (page 10)

[167] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th ACM Symposium on Principles of Programming Languages*, Jan. 1998. (page 11)

[168] J. M. Stevenson and D. P. Julin. Mach-US: UNIX On Generic OS Object Servers . In *Usenix Technical Conference*, Jan. 1995. (page 6)

[169] M. Stonebraker, J. Woodfill, J. Ranstrom, M. Murphy, M. Meyer, and E. Allman. Performance enhancements to a relational database system. In *ACM Transactions on Database Systems, 8(2)*, pp. 167-185, 1983. (page 2)

[170] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. In *Communications of the ACM, 1(8)*, Aug. 1959. (page 10)

[171] Sun Microsystems. *Java Remote Method Invocation Specification*. 1997. (page 17) (page 20)

[172] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, Reading, MA, 1998. (page 59)

[173] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *19th Australian Computer Science Conference*, Australian Computer Science Communications, 1(18), pp. 203-212, 1996. (page 64)

[174] T.E. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, pp. 92-94, 1992. (page 7) (page 78)

[175] A. Tanenbaum. Chapter 7. In *Distributed Operating Systems*. Prentice Hall, 1995. (page 119)

[176] J. E. Tidswell and T. Jaeger. Integrated constraints and inheritance in DTAC . In *Proc. of the 5th ACM Workshop on Role Based Access Control*, pp. 93-102, 2000. (page 130)

[177] P. A. Tullmann. *The Alta Operating System*. Master‘s thesis, The University of Utah , Dec. 1999. (page 12)

[178] V. A. Vyssotsky, F. J. Corbató, and R. M. Graham. Structure of the Multics Supervisor. In *Proc. of the 1965 AFIPS Fall Joint Computer Conference*, 1965. (page 60)

[179] P. Wagle. *The Synthetix Kernel ProgrammersInterfaceforRepluggingProcedureSpecializationsatRun-Time'*. In *The Synthethix Toolkit Workshop*, June 1997. (page 3)

[180] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pp. 203-216, Dec. 1993. (page 8)

[181] D. W. Wall and M. L. Powell. The Mahler Experience: Using an Intermediate Language as the Machine Description. In *Proc. of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 100-104, 1987. (page 10)

[182] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *16th Symp. on Operating System Principles*, pp. 116-128, Apr. 1997. (page 131)

[183] D. C. Wang and A. Appel. Type-preserving garbage collectors. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pp. 166-178, Jan. 2001. (page 90)

[184] A. Watson and P. Benn. *Multiprotocol Data Access: NFS, CIFS, and HTTP*. Technical Report TR3014, Network Appliance, Inc., May 1999. (page 2)

[185] C. Wawersich. *Design und Implementierung eines Profilers und optimierenden Compilers f¸r das Betriebssystem JX (Design and Implementation of a Profiler and Optimizing Compiler for the Java Operating System JX)*. Diplomarbeit, available as DA-I4-2001-05 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Apr. 2001. (page 17)

[186] A. Weissel. *Ein offenes Dateisystem mit Festplattensteuerung fuer metaXaOS (An open file system and disk driver for metaXaOS)*. Studienarbeit, available as SA-I4-2000-02 , Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Golm, Feb. 2000. (page 17) (page 145)

[187] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In. In *Proc. of International Workshop on Memory Management*, Sep. 1995. (page 83)

[188] M. Winter. *Design und Implementierung der AWT-Schnittstelle f¸r das Java-Betriebssystem JX*. Studienarbeit, Univ. of. Erlangen, Dept. of Comp. Science, Lehrstuhl 4, supervised by M. Felser and C. Wawersich, Oct. 2002. (page 17)

[189] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992. (page 10)

[190] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. In *Communications of the ACM, 17(6)*, pp. 337-345, June 1974. (page 6)

[191] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th Symposium on Operating System Principles*, pp. 63-76, Dec. 1987. (page 14)

[192] *Common Criteria for Information Technology Security Evaluation (CCITSE) Version 2.1*. 1999. (page 117)

[193] AT&T Labs-Research: dot-Tool, part of Graphviz - open source graph drawing software. , http://www.research.att.com/sw/tools/graphviz/. (page 158)

[194] L4Ka Hazelnut evaluation, http://l4ka.org/projects/hazelnut/eval.asp. (page 52)

[195] http://www.microsoft.com/hwdev/driver/wdm/default.asp. (page 102)

[196] Status page of the Fiasco project at the Technical University of Dresden, http://os.inf.tu-dresden.de/fiasco/status.html. (page 52)

[197] The Jakarta Project, http://jakarta.apache.org/tomcat/. (page 148)

[198] http://www.spec.org/osg/jvm98/. (page 145)

[199] Webpage of VMWare, http://www.vmware.com/. (page 9)

[200] Webpage of the IOZone filesystem benchmark, http://www.iozone.org/. (page 148)

Index

Kurzfassung

Die Architektur eines typsicheren Betriebssystems

Die Architektur herkömmlicher Betriebssysteme baut auf adressbasiertem Speicherschutz auf. Um Flexibilität mit niedrigen Kosten zu verbinden, ist die aktuelle Betriebssystemforschung dazu übergegangen, andere Schutzmechanismen zu untersuchen, unter anderem Typsicherheit. Diese Dissertation beschreibt eine Betriebssystemarchitektur, die den adressbasierten Schutz vollständig durch einen typbasierten Schutz ersetzt. Der Austausch eines so zentralen Bestandteils des Betriebssystems führt zu einer neuartigen Betriebssystemarchitektur, welche sich durch verbesserte Robustheit, Wiederverwendbarkeit, Konfigurierbarkeit, Skalierbarkeit und Sicherheit auszeichnet.

Die Dissertation beschreibt nicht nur das Design dieses Systems, sondern auch die Implementierung eines Prototyps und die Performanz erster Anwendungen, wie Dateisystem, Webserver, Datenbanksystem und Netzwerk-Dateiserver. Der JX genannte Prototyp verwendet Java Bytecode als typsicheren Instruktionssatz. JX ist in der Lage, existierende Java-Programme ohne Änderungen auszuführen.

Das System basiert auf einem modularen Mikrokern. Dieser Mikrokern ist der einzige Teil des Systems, der in einer unsicheren Sprache geschrieben ist. Leichtgewichtige Schutz-Domänen ersetzen das schwergewichtige Prozesskonzept herkömmlicher Betriebssysteme. Schutz, Ressourcenverwaltung und Terminierung sind an diese Domänen gekoppelt. Der Programmcode ist in Komponenten organisiert, welche in die Domänen geladen werden. Der Portalmechanismus - ein schneller Inter-Domän-Kommunikationsmechanismus - erlaubt es gegenseitig misstrauenden Domänen sicher zu kooperieren.

Die Dissertation zeigt, dass es möglich ist, ein universelles und effizientes Mehrbenutzer-Betriebssystem auf Typsicherheit aufzubauen.

KAPITEL 1 *Einleitung*

Diese Dissertation beschreibt die Architektur einer neuen Klasse von Betriebssystemen: Systeme die einen typsicheren Befehlssatz als einzigen Schutzmechanismus verwenden. In einem typsicheren Befehlssatz werden Befehle (Operationen) auf typisierte Operanden angewendet welche auf typisierte Datenentitäten verweisen. Es existieren strenge Regeln die beschreiben, welche Typen von Operanden auf welche Typen von Datenentitäten verweisen dürfen und welche Operationen auf die Operanden angewendet werden dürfen.

Die Verwendung eines typsicheren Befehlssatzes anstelle des traditionellen addressbasierten Schutzmechanismus' hat eine Reihe von Vorteilen. Die *Robustheit* des Systems verbessert sich, da viele Arten von Programmfehlern in einer frühen Entwicklungsstufe durch Verwendung des Typsystems entdeckt werden können. Verbesserte Robustheit hat einen positiven Effekt auf *Sicherheit* und *Zuverlässigkeit*. Allerdings werde ich in dieser Dissertation zeigen, dass zusätzliche Mechanismen erforderlich sind, um einen sicheren Systemzustand zu erhalten. Ein typsicherer Befehlssatz ist ein sehr feingranularer Schutzmechanismus, welcher es erlaubt, *feingranulare Schutzdomänen* zu erzeugen und eine inkrementelle *Erweiterbarkeit* des Systems ermöglicht. Mein Systemdesign beinhaltet einen Kommunikationsmechanismus, der hocheffiziente aber ungeschützte Kommunikation innerbalb einer Schutzdomäne und etwas langsamere Kommunikation über Schutzgrenzen hinweg erlaubt. Da der Programmierer dieselbe Abstraktion für Intra-Domän- und Inter-Domän-Kommunikation verwendet, werden Programmmodulgrenzen unabhängig von Schutzdomängrenzen und das System kann *entsprechend seines gewünschten Einsatzzwecks konfiguriert* werden. Die Konfigurationsmöglichkeit erstreckt sich zwischen den beiden Möglichkeiten, für jede Komponente eine eigene Domäne zu verwenden und alle Komponenten in einer Domäne zu platzieren. Systeme, die nur für eine bestimmte Aufgabe eingesetzt werden, wie zum Beispiel dedizierte Dateiserver, profitieren von den Performanzvorteilen die entstehen, wenn alle Komponenten in derselben Domäne laufen. Systeme, die nicht vertrauenswürdige und potentiell bösartige Programme ausführen, wie zum Beispiel Agentenplatformen, profitieren von der Möglichkeit, feingranulare Schutzdomänen zu erzeugen. Solche Systeme müssen in der Lage sein, nichtvertrauenswürdige Programme vollständig zu isolieren. Das bedeutet, sowohl den *Zugriff auf Informationen als auch den Zugriff auf Ressourcen einzuschränken*. Alle der genannten Eigenschaften des Systems müssen ohne übermässige Performanzeinbussen gegenüber traditionellen Systemen realisiert werden.

KAPITEL 2 *Zusammenfassung*

1 Beitrag zur Wissenschaft

Die wissenschaftliche Leistung dieser Arbeit beteht darin, eine neue Art und Weise der Konstruktion von Betriebssystemen aufgezeigt zu haben. Der Beitrag zur Wissenschaft besteht konkret

- in der Analyse der Eigenschaften, die ein Betriebssystem aufweisen muss, um heutige und absehbare Anforderungen zukünftiger Anwendungen erfüllen zu können,
- in der Analyse der Unzulänglichkeiten und in dem Aufzeigen der nützlichen Ansätze bisher bestehender Systeme,
- in der Entwicklung eines Designs, dessen Kern darin besteht, einen bisher nur für Laufzeitsysteme von Programmiersprachen verwendeten Schutzmechanismus als zentralen Schutzmechanismus des Betriebssystems zu verwenden,
- in der prototypischen Implementierung dieses Designs und der detailierten Evaluierung des Prototypen hinsichtlich dessen funktionalen und Performanzeigenschaften.

2 Erfüllung der Anforderungen

Die beschriebene Architektur erfüllt die in Kapitel 1 aufgestellten Anforderungen:

- **Robustheit, Sicherheit, Zuverlässigkeit**. Der Anteil von unsicherem C- und Assembler-Code konnte minimal gehalten werden. Das komplette Betriebssystem, einschliesslich der Gerätetreiber, wird in typsicherem Code erstellt. Das vereinfacht die Systemarchitektur und verbessert die Robustheit und die Zuverlässigkeit des Systems. Eine Sicherheitsarchitektur wurde entwickelt, die eine genaue Steuerung des Informations- und Kontrollflusses innerhalb des Systems erlaubt.
- **Konfigurierbarkeit, Wiederverwendbarkeit**. Ausser dem Mikrokern ist der komplette Code, einschliesslich des Betriebssystems, in Komponenten organisiert. Diese Komponenten können in einer gemeinsamen oder in unterschiedlichen Schutzdomänen ablaufen, ohne den Komponentencode modifizieren zu müssen. Diese Wiederverwendbarkeit in verschiedenen Konfigurationen erlaubt es, das System entsprechend seines Einsatzzwecks anzupassen. Dieser Einsatzzweck kann ein eingebettetes System, ein Arbeitsplatzrechner oder ein Server sein.
- **Erweiterbarkeit, feingranularer Schutz**. Betriebssystemcode und Applikationscode arbeiten in verschiedenen, streng voneinander isolierten Schutzdomänen. Der Grundspeicherbedarf

einer Schutzdomäne ist gering und erlaubt die Erweiterbarkeit des Systems durch Erzeugen neuer Domänen.

- **Performanz**. Die Performanz des Systems liegt für rechenintensive Betriebssystemaufgaben zwischen 50 und 100 Prozent eines traditionellen UNIX Betriebssystems. Der Unterschied wird noch geringer, wenn Ein-/Ausgabeoperationen von realer Hardware beteiligt sind.
- **Uniformität**. Alle virtuellen Ressourcen, wie zum Beispiel Dateien, Endpunkte von Netzwerkverbindungen, Fenster einer graphischen Benutzeroberfläche, sowie Datenbankverbindungen werden auf eine einzige Abstraktion reduziert: dem Portal. Das Portal ist einleichtgewichtiges, Capability-ähnliches Konstrukt, um auf einen Dienst zuzugreifen. Diese einfache aber auch mächtige Architektur erlaubt die Einführung neuer virtueller Ressourcen und erleichtert damit die Anpassung an neue Umgebungen, wie zum Beispiel an ein verteiltes System.
- **Produktivität**. Die Existenz des Prototyps demonstriert, dass es - durch die Verwendung von Typsicherheit gepaart mit einfacher Architektur - einer kleinen Gruppe von Personen in kurzer Zeit möglich ist, ein nahezu vollständiges Betriebssystem zu entwickeln.

3 Zusammenfassung des Designs

Das Design des JX-Systems wurde durch vorangegangene Betriebssystemarchitekturen inspiriert. Sein Prinzip der vollständigen Trennung ist ähnlich der vertikalen Strukturierung des Nemesis Betriebssystems [114]. Orthogonalität der Abstraktionen, Schutz und Modularität war das wesentliche Ziel des Lipto-Systems [57]. Ressourcenverwaltung auf Anwendungsebene wurde in unterschiedlichen Projekten verfolgt. Das prominenteste dieser Projekte ist sicherlich der Exokernel [61].

Im Sinne eines Mikrokernes besitzt das JX-System fünf zentrale Abstraktionen: Domänen, Fäden, Komponenten, Portale und Dienste. Weitere Abstraktionen werden als Portale zu Diensten der DomainZero oder als schnelle Portale bereitgestellt. Beispiele für schnelle Portale sind Speicherobjekte und atomare Variablen. Das JX-System besitzt die folgenden einzigartigen Eigenschaften:

Typsicherer Schutz. Der Schutz basiert auf der Verwendung eines typsicheren Instruktionssatzes. Schutzhardware, wie zum Beispiel eine MMU, ist nicht erforderlich. Dies erlaubt es dem JX-System, selbst auf eingebetteten Systemen Schutzdomänen zur Verfügung zu stellen. Die Kommunikation zwischen Schutzdomänen ist sehr effizient, da kein Wechsel eines Hardwarekontextes stattfinden muss.

Unabhängige Halden. Jede Domäne hat seine eigene, von anderen Domänen unabhängige Halde. Dies steht im Kontrast zu anderen Systemen, die den Namensraum-Mechanismus verwenden, der von Java-Klassenladern erzeugt wird. Die unabhängigen Halden ermöglichen unabhängige Speicherbereinigungzyklen, die Implementierung domänspezifischer Speicherbereinigung und die Terminierung einer Domäne mit sofortiger Ressourcenfreigabe.

Mehrfache Speicherbereinigungsimplementierungen. Die Performanz der Speicherbereinigung hängt massgeblich von der Charakteristik der einzelnen Anwendung ab. Jede Domäne kann mit einer Speicherbereinigungsimplementierung verwendet werden, die aus einer durch den Mikrokern bereitgestellten Menge von verschiedenen Implementierungen ausgewählt werden kann.

Unabhängige Fädenablaufplanung. Fäden einer Domäne können unabhängig von anderen Domänen disponiert werden. Es ist möglich, domänenspezifische Ablaufplaner zu verwenden, die in Form des typsicheren Instruktionssatzes bereitgestellt werden.

Sicherheit und Informationsschutz. Domänen sind isoliert und können nur über dedizierte Kanäle unter Verwendung der Portale kommunizieren. Das Sicherheitssystem von JX kann die Verbreitung und die Nutzung der Portale kontrollieren. Das Sicherheitssystem, welches die Sicherheitspolitik enthält, ist von den Domänen separiert und kann ausgewechselt werden, ohne die Anwendungs-Domänen modifizieren zu müssen.

Automatisches Speichermanagement. JX stellt als besondere Abstraktion Speicherportale bereit, um große Speichereinheiten oder spezielle Speicherbereiche (z.B. Speicherabbildungen von Geräteregistern) verwenden zu können. Speicherportale können von Domänen gemeinsam genutzt werden und werden automatisch durch einen Referenzzählmechanismus verwaltet.

Transparentes gemeinsames Nutzen von Programmcode. Nicht-modifizierbare Daten können von Domänen gemeinsam genutzt werden. Das schließt übersetzten Programmcode und Zeichenketten-Konstanten ein. Veränderliche Daten - zum Beispiel statische Variablen - werden für jede Domäne separat angelegt. Dies macht das gemeinsame Nutzen von Programmcode vollständig transparent.

Automatische Erzeugung von Diensten. Ein Dienst wird automatisch erzeugt, wenn ein Dienstobjekt in eine andere Domäne bewegt werden soll. Die Domäne erhält anstelle des Dienstobjektes ein Portal zu dem neu erzeugten Dienst.

Flexible Konfiguration. Schutzdomängrenzen sind unabhängig von Komponentengrenzen. Komponenten können in ein und dieselbe oder in verschiedene Domänen platziert werden, ohne sie dafür modifizieren zu müssen.

Terminierung mit sofortiger Ressourcenfreigabe. Außer Gerätetreiber-Domänen können alle Domänen unverzüglich terminiert werden. Ressourcen, die von diesen Domänen in Anspruch genommen worden sind, können sofort freigegeben werden.

Robuste Gerätetreiber. Alle Gerätetreiber liegen in dem typsicheren Instruktionssatz vor und profitieren von der Robustheit, die durch die Typsicherheit gegeben ist. Robustheit ist ebenfalls durch spezielle Portalschnittstellen garantiert, die den Gerätetreibern erlauben, auf Geräteregister und auf die Fäden zur Handhabung von Unterbrechungen zuzugreifen, um sich zu synchronisieren, ohne Unterbrechungen abzuschalten.

Unabhängige Klassenbibliotheken. Jede Domäne kann ihre eigene Implementierung einer Klassenbibliothek besitzen. Das gilt auch für die Klassen des Java Development Kit.

4 Zukünftige Arbeiten

Die JX Architektur kann als eine Plattform genutzt werden, um

- **Mehrzwecksysteme** zu bauen, die den Benutzer bei der täglichen Arbeit unterstützen. Das umfaßt beispielsweise das Editieren von Texten oder das Arbeiten mit Tabellen. Der Window-Manager wird dabei die zentrale Komponente eines solchen System sein.

- **dedizierte Systeme** zu erstellen, die in der Lage sind, Code wieder zu benutzen und feingranularen Schutz ohne Hardwareunterstützung bereitstellen.
- **eingebettete Systeme** welche nichtvertrauenswürdige Programme auf Hardware laufen lassen müssen, die keine Schutzmechanismen zur Verfügung stellt. Diese Systeme können den feingranularen Schutz verwenden, der durch JX-Domänen zur Verfügung gestellt wird.
- **verteilte Syteme** zu entwickeln, die weit über das Konzept eines verteilten Systems hinausgehen, als es derzeitige UNIX-basierte verteilte Systeme zum Ziel haben. Das Bewegen einer aktiven Domäne durch ein Netzwerk kann beispielsweise für Lastverteilung oder zur Erhöhung der Fehlertoleranz benutzt werden. Wir habe die ersten Schritte in diese Richtung unternommen, indem wir ein aktives Speichersystem [74] implementiert haben, dass es erlaubt, einen Dateisystemagenten zum Dateiserver zu bewegen.

5 Abschließende Bemerkungen

Ich hoffe, dass das JX System die Aufmerksamkeit der Betriebssystemgemeide vermehrt auf die vielen Vorteile der typbasierten Herangehensweise beim Bau von Betriebssystemen lenkt. Mit der Vorstellung eines lauffähigen Prototypen und der Demonstration dessen Funktionalität und Performanz sollte jeder Zweifel an der Durchführbarkeit solch eines Unterfangens ausgeräumt sein.

Inhaltsverzeichnis

Lebenslauf

Persönliche Daten

Michael Golm

geboren am 05.10.1970 in Leipzig, verheiratet, zwei Kinder

Ausbildung

seit Jan. 1997	Promotionsstudium am Lehrstuhl für Betriebssysteme der Universität Erlangen-Nürnberg (Prof. Hofmann);
Jan. 1997	Abschluss des Informatik-Studiums als Dipl.-Informatiker
	Titel der Diplomarbeit: "Design and Implementation of a Meta Architecture for Java"
Sept. 1991 - Jan. 1997	Studium der Informatik an der Universität Erlangen-Nürnberg
Aug. 1991	Abitur am Albert-Schweitzer Gymnasium Erlangen

Berufstätigkeit

seit Jan. 1997	wissenschaftlicher Mitarbeiter am Lehrstuhl für Betriebssysteme der Universität Erlangen-Nürnberg;
	Ausbildung von Studenten in den Bereichen objektorientierte Programmierung, verteilte Systeme / Java und systemnahe Programmierung unter Unix / C;
seit 1998	Schulungen Objektorientierung, Java, JDBC, RMI für Sun Microsystems, Siemens Erlangen/Nürnberg/Karlsruhe, Astrum, 3Soft, Bizerba