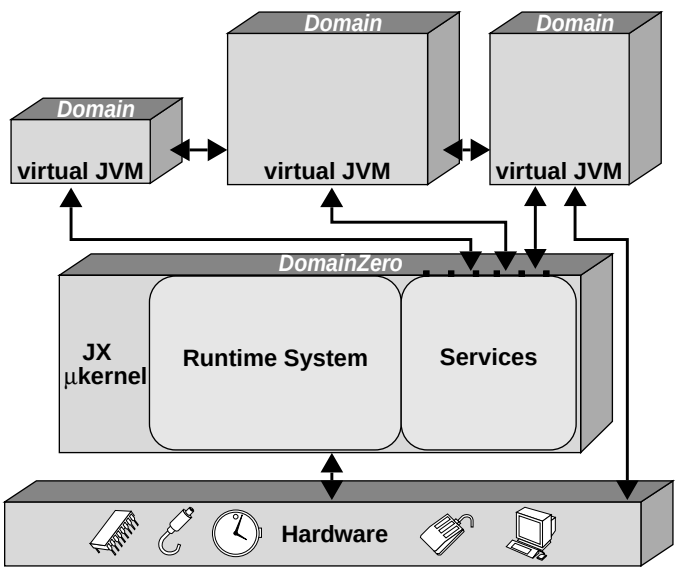
CHAPTER 3 *Architectural Overview*

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.

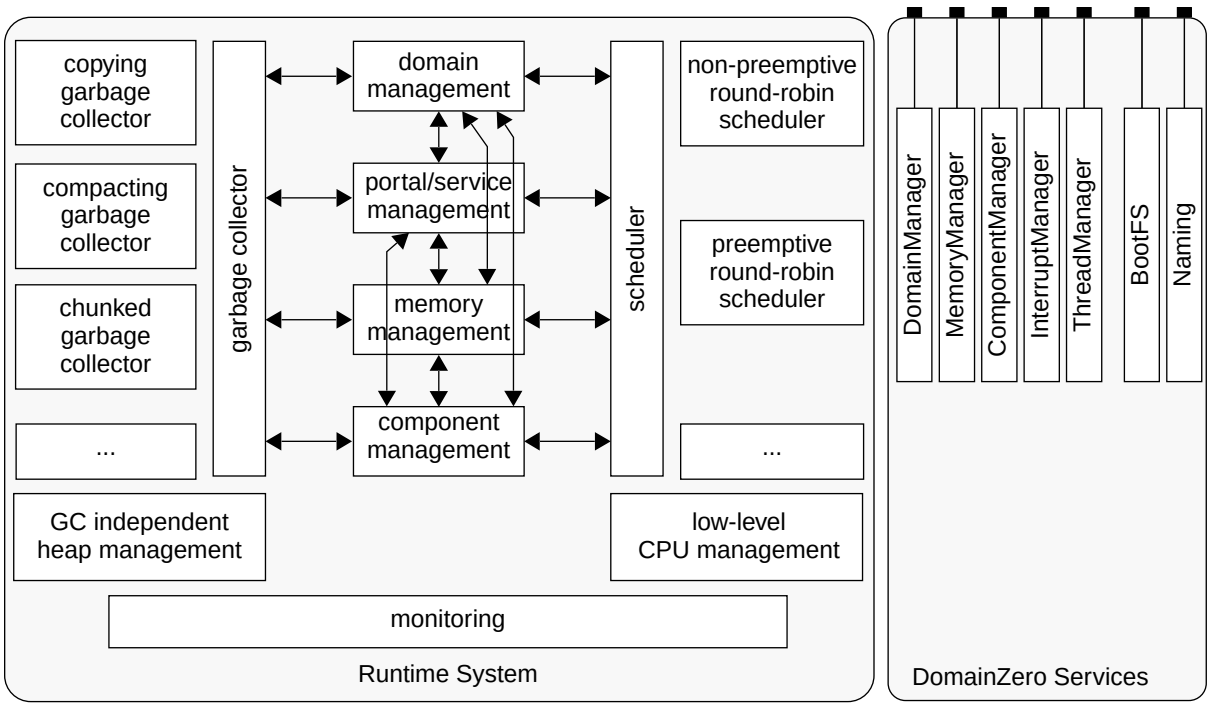
The microkernel does not use blocking mutual exclusion locks (mutexes) or semaphores. Critical regions are short and guarded by blocking interrupts on a single processor and using spinlocks on a multiprocessor.

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

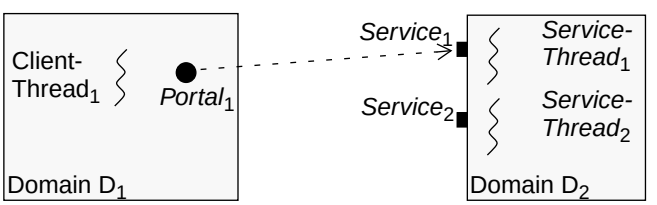
Portals are the fundamental inter-domain communication mechanism. The portal mechanism works similar to Java's RMI [171], making it easy for a Java programmer to use it. A portal can be thought of as a proxy for an object that resides in another domain and is accessed using remote procedure call (RPC).

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.

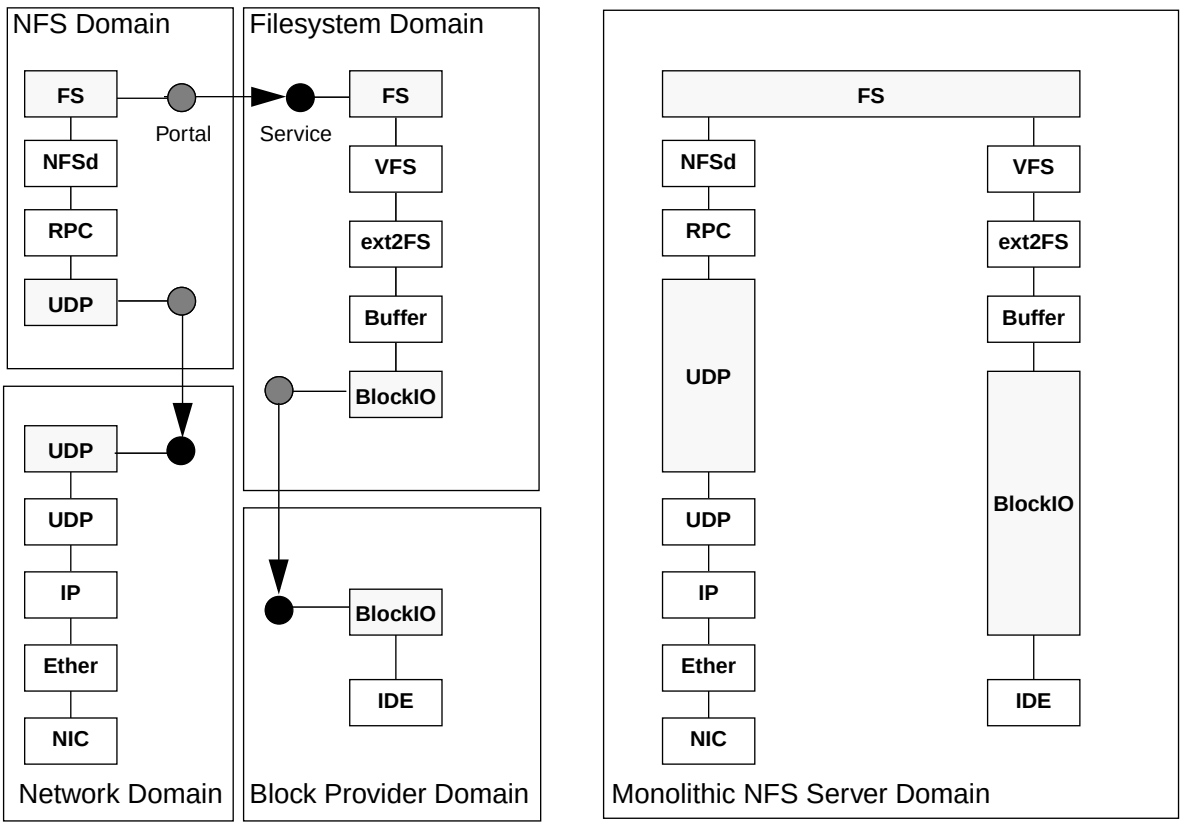
The second point is addressed by using a typed communication channel. This excludes several errors when connecting components. It is still possible to connect two components which use the same syntactical type but use it with a different semantics. These errors can be reduced by the careful design of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