Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System

Michael Golm, Jürgen Kleinöder, Frank Bellosa
University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{golm,kleinoeder,bellosa}@informatik.uni-erlangen.de

Abstract

Early type-safe operating systems were hampered by poor performance. Contrary to these experiences we show that an operating system that is founded on an object-oriented, type-safe intermediate code can compete with MMU-based microkernels concerning performance while widening the realm of possibilities.

Moving from hardware-based protection to software-based protection offers new options for operating system quality, flexibility, and versatility that are superior to traditional process models based on MMU protection. However, using a type-safe language—such as Java—alone, is not sufficient to achieve an improvement. While other Java operating systems adopted a traditional process concept, JX implements fine-grained protection boundaries. The JX System architecture consists of a set of Java components executing on the JX core that is responsible for system initialization, CPU context switching and low-level domain management. The Java code is organized in components which are loaded into domains, verified, and translated to native code.

JX runs on commodity PC hardware, supports network communication, a frame grabber device, and contains an Ext2-compatible file system. Without extensive optimization this file system already reaches a throughput of 50% of Linux.

1 Introduction

For several years there has been an ongoing discussion in the OS community whether software-based protection is a promising approach [3]. We want to support the arguments for software-based protection with the experience we gained while building the JX operating system.

While MMU-based protection is commonly used in today's operating systems it has some deficiencies [10], [3]. From the point of functionality it neither meets the actual requirements of fine grained protection (page size is too coarse), nor offers it appropriate abstractions for access control (page tags are not capabilities).

These deficiencies justify the exploration of alternative protection mechanisms. Java popularized a protection mechanism that is based on a combination of type-safe intermediate code and load-time program verification.

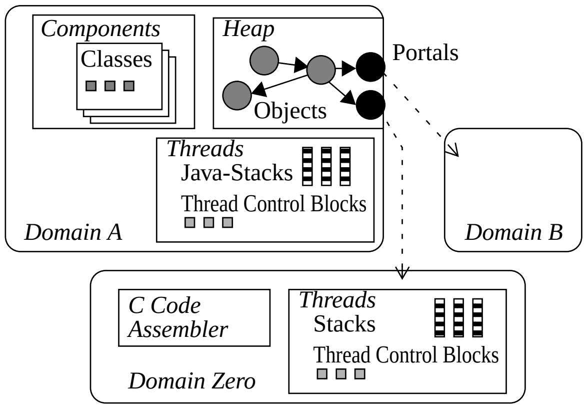
Several other research groups have been building Java-based operating systems: Sun's JavaOS [14], which was later replaced by "JavaOS for Business" [18], JN [16], J-Kernel [11], KaffeOS [2], and Joust [9]. But they are either limited by a monolithic structure or are built upon a full-featured OS and JVM. Furthermore, no performance figures for OS related functionality are published. KaffeOS and J-Kernel are two projects that try to overcome the monolithic structure by intruducing a process concept which is similar to the domain concept of JX. But their research is mainly concerned with introducing the traditional process concept and a red line [6] between user level and kernel into their Java operating system. While a red line between trusted and untrusted code is indeed important, we must free our mind from the MMU-enforced architecture of traditional operating systems. The aim of our research is a customizable and flexible [4] open OS architecture with fine-grained protection boundaries. Depending on functionality and deployment of a system there are different levels of trust and protection. An embedded real-time system needs a different red line than a single-user desktop system or a multi-user server system or an active network node OS [5]. In our architecture it is possible to draw red lines when and where they are needed.

While other Java operating systems require a microkernel, or even a full operating system including a JVM, JX runs on the bare hardware with only a minimal statically linked core (< 100kB). The remaining operating system functionality, including device drivers, is provided by Java components that are verified, compiled to native code, and optimized at load time.

The paper is structured as follows: In section 2 we describe the architecture of the JX system. The problems that appear when untrusted modules directly access hardware are discussed in section 3. Section 4 gives examples of the performance of IPC and file system access.

2 JX System Architecture

The JX system consists of a small core, written in C and assembler, which is less than 100 kilobytes in size. The majority of the system is written in Java and running in separate protection domains. The core runs without any protection and therefore must be trusted. It contains functionality that can not be provided at the Java level (system initialization after boot up, saving and restoring CPU state, low-level domain management, monitoring).



The Java code is organized in components (Sec. 2.2) which are loaded into domains (Sec. 2.1), verified (Sec. 2.4), and translated to native code (Sec. 2.5). A domain can communicate with another domain by using portals (Sec. 2.3).

The protection of the architecture is solely based upon the JX core, the code verifier, the code translator, and hardware-dependent components (Sec. 3). These elements are the *trusted computing base* [7] of our architecture.

2.1 Domains

A domain is the unit of protection, resource management, and typing.

Protection. Components in one domain trust each other. One of our aims is code reusability between different system configurations. A component should be able to run in a separate domain, but also together (co-located) with other components in one domain. This leads to several problems:

- The parameter passing semantics must be by-copy in inter-domain calls, but may be by-reference in the co-located case. This is an open problem.
- During a portal call a component must check the validity of the parameters because the caller could be in a different domain and is not trusted. When caller and callee are co-located (intra-domain call), the checks change their motivation—they are no longer done for security reasons but for robustness reasons. We currently parametrize the component whether a safety check should be performed or not.

Resource Management. JX domains have their own heap and own memory area for stacks, code, etc. If a domain needs memory, a domain-specific policy decides whether this request is allowed and how it may be satisfied, i.e., where the memory comes from. Objects are not shared between domains, but it is possible to share memory. Other Java systems use shared objects with the consequence of complicated and not interdependent garbage collection, problems during domain termination, and quality-of-service crosstalk [13] between garbage collectors.

Typing. A domain has its own type space, that initially contains exactly one type: java.lang.Object. Types (classes and interfaces) and code (classes) can then be loaded into the domain. Our type-space approach differs from the Java type spaces [12] as we do not use the class loader as type-space separator but tie type separation to resource management and protection. By this means a SecurityManager becomes redundant and protection boundaries are automatically enforced.

The C and assembler code of the JX core are encapsulated by a special domain, called *DomainZero*. All other domains contain only Java code. We do not allow native methods.

2.2 Components

Code is generally loaded as a component. JX does not support loading of single classes. A component is a collection of classes and interfaces. There are four kinds of components:

- *Library*: A simple collection of reusable classes and interfaces (example: the Java Development Kit).
- *Service*: A component that implements a specific service, e. g., a file system or a device driver. A service component is *started* after it has been loaded. To start a service means to execute a static method that is specified in a configuration file that is part of the component.
- *Interface*: Access to a service in another domain is always performed using an interface. An interface component contains all interfaces that are needed to access a service. An interface component also contains the classes of parameter objects. A special interface library *zero* contains all interfaces to access DomainZero.

•*Domain*: A domain is started by loading a domain component. An initial thread is created and a static method is executed.

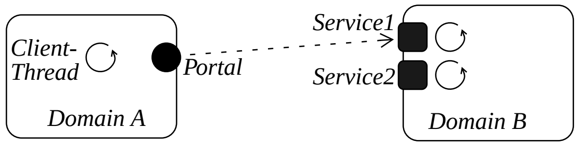
Components can be shared between domains. Sharing happens at two levels. At a logical level sharing establishes a window of type compatibility between two domains. At a lower level, sharing saves memory, because the (machine) code of the component has to be stored only once. While component sharing complicates resource accounting and domain termination, we believe that code sharing is an essential requirement for every real operating system. While code can be shared if the domains use the same type of execution environment (translator, memory layout), static variables are never shared. In JX this is implemented by splitting the internal class representation into a domain-local part, that contains the statics, and a shared part, that contains code and meta information.

2.3 IPC, Portals, and Services

Domains communicate solely by using portals. An object that may be accessed from another domain is called *service*. Each service is associated with a *service thread*.

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. Portals allow to establish the “principle of least privilege”. A domain gets only the portals it needs for doing its job.

A portal looks like a normal object reference. The portal type is an interface that is derived from the interface 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.



An object reference can be passed as parameter of a portal invocation only if the object is a service. In this case a portal to the service is transferred and the reference counter of the service is incremented. Other parameters are passed by value. When a portal is no longer referenced in a domain, it is removed by the garbage collector and the reference counter of the associated service is decremented.

A portal/service connection between two domains requires that these domains have overlapping type spaces, i.e. the interface component must be logically shared. If the interface component depends on other components, they must be shared, too.

2.4 Component Verifier

When a component is loaded into a domain, its bytecode is verified before it is translated into machine code. As in the normal Java bytecode verifier, the conformance to the Java rules is checked. Basically this guarantees type safety. Furthermore the verifier performs additional JX-specific checks regarding interrupt handlers (Sec. 2.6), memory objects (Sec. 2.7), and schedulers (Sec. 2.9).

A type-safe operating system has the well-known advantages of robustness and ease of debugging. Furthermore, it is possible to base protection and optimization mechanisms on the type information. This is extensively employed in JX by using well-known interfaces (contained in a trusted library) and restricting the implementability of these interfaces (Sec. 2.6 and 2.7).

2.5 Component Translator

Components are translated from bytecode into machine code. The translator is a crucial element of JX to get a reasonable performance. The translator is domain-specific, so it can be customized for a domain to employ application-specific translation strategies. The same component may be translated differently in different domains. As the translator is a trusted component, this facility has to be used carefully because it affects the protection of the whole system.

Furthermore the translator is used to “short-circuit” several portal invocations. Special portals that are exported by DomainZero often do not need the domain context of DomainZero. Invocations of such portals can be inlined directly at the call site.

2.6 Interrupts

An interrupt is handled by invoking the handleInterrupt method of a previously installed interrupt handler object. The method is executed by a dedicated thread while interrupts on the interrupted CPU are disabled. This would be called the *first-level interrupt handler* in a traditional operating system. To guarantee that the handler can not block the system forever, the verifier checks all classes that implement the InterruptHandler interface whether the handleInterrupt method has certain time bounds. 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.

2.7 Memory Management

Heap and Garbage Collection. The memory of the objects within a domain is managed by a heap manager with garbage collector. Currently, the heap manager is part of the JX core. It cooperates with the translator to obtain information about the object structure and stack structure. So far we are working with only one heap manager implementation and one translator implementation, but it is also possible to build domain-specific heap managers. They can even be written in Java and run in their own domain. The heap manager is a trusted part of the system.

Memory objects. To handle large amounts of data, Java uses arrays. Java arrays are useless for operating system components, because they do not provide access control and it is not possible to share only a part of an array. JX uses Memory objects instead. The memory that is represented by such a Memory object can be accessed via method invocations. These invocations are inlined by inserting the machine instructions for the memory access instead of the method invocation. This makes memory access as fast as array access. A Memory object can represent a part of the memory of another Memory object and Memory objects can be shared between domains like portals. Sharing memory objects between domains and the ability to create subranges are the fundamental mechanisms for a zero-copy implementation of system components, like the network stack, the file system, or an NFS server.

Avoiding range checks by object mapping. A memory range can be mapped to a (virtual) object that implements a marker interface (an interface without methods that is only used to mark a class as MappedLittleEndian or MappedBigEndian). The verifier ensures that a class that implements one of these interfaces is never instantiated by the new bytecode. Instead the objects are created by mapping and the translator generates code to directly access the memory range for access to instance variables. This makes the range check redundant.

2.8 Domain Termination

When a domain terminates, all resources must be released. Further interaction with the domain raises an exception.

All services are removed by stopping the service thread. A service contains a reference counter, that is incremented each time a portal to this service is passed to another domain. It is also incremented when a client domain passes the portal to another client domain. It is decremented, when the portal object in a client domain is garbage collected or when the client domain is terminated. As long as the reference counter is not zero, the service can not be completely removed when its domain terminates. Until all reference counters drop to zero, the domain remains in a *zombie* state.

Interrupt handlers are uninstalled. All threads are stopped and the memory (heap, stacks) is released.

2.9 Scheduling

CPU scheduling in JX is split into two scheduler levels. The *low-level scheduler* decides which domain should run on the CPU. Each CPU has its own low-level scheduler. The *high-level scheduler* is domain-specific - each domain has one high-level 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. The high-level schedulers are responsible for scheduling the threads of a domain.

The high-level scheduler may be part of the domain or may be located in a different domain.

To avoid that one domain monopolizes the CPU, the computation can be interrupted by a timer interrupt. The timer interrupt leads to the invocation of the low-level scheduler. The low-level scheduler first informs the high-level scheduler of the interrupted domain about the preemption. For this purpose it invokes a method of the high-level scheduler with interrupts disabled. An upper bound for the execution time of this method has been verified during the verification phase. When the method returns, the system switches back to the low-level scheduler. The low-level scheduler then decides, which domain to run next. After ensuring that it will be reactivated with the next (CPU-local) timer interrupt, the low-level scheduler activates the high-level scheduler of the selected domain. The high-level scheduler chooses the next runnable thread. It can switch to this thread by calling a method at the CPU portal. This method can only be called by a thread that runs on the corresponding CPU.

3 Device Drivers

Due to the enormous amount of new hardware that appeared in the last years, operating system code is dominated by device drivers. While it is rather straight forward to move most operating system parts, such as file systems or network protocols, out of the trusted kernel, it is very difficult for device drivers.

Developers of commodity hardware do not assume that their products are directly accessed by untrusted code. Although the Nemesis project has demonstrated that it is possible to build user-safe hardware [17], we do not expect such hardware to become commercially available in the near future.

Device drivers in JX are programmed in Java and are installed as service component 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.

To reduce the amount of critical code, the driver is split into a (simple) trusted part and a (complex) untrusted part.

To understand the issues related to device drivers, we have developed drivers for the IDE controller, the 3C905B network card, and the Bt848 framegrabber chip. 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. The Bt848 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. We currently trust the RISC generator and thus limit extensibility. To allow an untrusted component to download RISC code, we would need a verifier for this instruction set.

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 tables. JX uses type-safety, special checks of the verifier, and splitted drivers to address these problems.

4 Performance

IPC. We measured the performance of a portal call. Table 1 compares the IPC round-trip performance of JX with fast microkernels and other Java operating systems.

System	IPC (cycles)
L4KA (PIII, sysenter, sysexit) [8]	800
Fiasco/L4 (PIII 450 MHz) [http://os.inf.tu-dresden.de/fiasco/status.html]	2610
J-Kernel (LRMI on MS-VM, PPro 200MHz) [11]	440
Alta/KaffeOS [1]	27270
JX/hosted (Linux 2.2.14, PIII 500MHz)	7100
JX/native (PIII 500MHz)	650

Table 1: IPC latency (round-trip)

Comparing IPC times for these systems is not easy because they were measured on different hardware (cache size, cache bandwidth, memory bandwidth, etc.), and, more importantly, they have different protection models. IPC is usually more expensive on a system with better protection. Currently the IPC path in JX is implemented in C and not optimized. It may be better compared with the Fiasco implementation of L4 than with L4KA. The emphasis of our work was on getting the architecture right and enabling performance, but not achieving it. The bad performance of Linux-hosted JX can be attributed to the use of sigprocmask to disable/restore signals.

The IPC cost of J-Kernel does *not* include thread switching costs, because the J-Kernel uses a “segmented” stack. IPC without switching threads complicates resource accounting, garbage collection, termination, and type separation.

File System. We have implemented the ext2fs in Java [19]. We reused the algorithms that are used in Linux-ext2fs.

We used the iozone benchmark to measure the Linux ext2fs re-read throughput (file size: 4 kB, record length: 4 kB — iozone -r 4 -s 4 -i 0 -i 1). To measure JX re-read throughput we wrote a Java benchmark, similar to iozone.

The system configuration that we measured works as follows: The virtual file system, the buffer cache, and the ext2 file system run in one domain (FSDomain). The IDE device driver runs in another domain. The client runs in a third domain. A service thread in the FSDomain accepts client requests. The client domain gets a portal to the virtual file system and calls lookup to get a FileInode portal. FSDomain uses one thread to asynchronously receive data from the block device driver. Only the service thread is active in this benchmark, because all data comes from the buffer cache.

System	Throughput (MByte/s)	Latency (µsec/4kB)
Linux (PIII 500 MHz)	400	10.0
JX (PIII 500MHz)	201	19.9
JX co-located (PIII 500MHz)	213	18.7

Table 2: File system re-read throughput and latency

We now try to estimate the necessary performance improvement to reach Linux throughput. The latency can be broken down as shown in table 3.

Operation	JX	JX goal
memory copy	5.2	5.2
IPC	1.3	1.3
file system logic	13.2	3.5

Table 3: Latency breakdown (in µsec)

Memory copy and IPC are relative constant costs in JX. The poor performance of the file system logic is not a problem of the JX architecture but of our non-optimizing compiler. With an improvement of factor 4 in Java performance, we would reach the Linux performance level. Although safety-related overhead cannot be avoided completely, recent research in JIT compiler technology has shown that an optimizing compiler can improve the performance of a Java program significantly. Performance differences of factor 10 are not unusual between non-optimizing and optimizing Java compilers.

5 Status and future work

The system runs either on standard PC hardware (i486, Pentium, and embedded PCs with limited memory) or as a guest system on Linux. The JX Java components also run on a standard JDK (with an emulation for Memory objects). When running on the bare hardware, the system can access IDE disks [19], 3COM 3C905 NICs [15], and Matrox G200 video cards. The network code contains IP, TCP, UDP, NFS2 client, and SUN RPC. JX also runs on a PIII SMP machine.

We have already implemented a heap manager that runs in its own domain and manages the heap of another domain. This heap manager is always called, when the managed domain tries to create a new object or array. Creating a new object with the build-in mechanism costs 250 cycles, calling another domain adds at least 650 cycles. This is not practical until we further improve IPC performance. There are also efforts to improve the quality of the machine code generated by the translator.

The JX architecture supports a broad spectrum of OS structures — from pure monolithic to a vertical structure similar to the Nemesis OS [13]. We are going to investigate the issues that are involved when reusing components between these diverse operating system configurations.

6 References

[1] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proc. of the 2000 USENIX Annual Technical Conference*, pp. 197-210, June 2000

[2] G. Back, W. C. Hsieh, J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of the 4th OSDI*, pp.333-346, Oct. 2000

[3] B. Bershad. S.Savage, P. Pardyak. Protection is a Software Issue. In *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, pp. 62-65, 1995

[4] V. Cahill. *Flexibility in Object-Oriented Operating Systems: A Review.* Technical Report TCD-CS-96-05, Dep. of Comp. Science Trinity College Dublin, 1996

[5] K. Calvert (ed.), *Architectural Framework for Active Networks*, Version 1.0, Active Networks Working Group, July 1999

[6] D. R. Cheriton. Low and High Risk Operating System Architectures. In *Proc. of OSDI*, pg. 197, Nov. 1994

[7] Department of Defense. *Trusted computer system evaluation criteria*. DOD Standard 5200.28, Dec. 1985

[8] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. of the 9th SIGOPS European Workshop*, Sep. 2000.

[9] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheckti. Experiences building a communication-oriented JavaOS, *Software--Practice and Experience*, 30 (10), Apr. 2000

[10] C. Hawblitzel, T. von Eicken. *A case for language-based protection*. Technical Report TR-98-1670, Dep. of Comp. Science, Cornell University, March 1998

[11] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998

[12] S. Liang, G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proc. of OOPSLA '98*, October 1998

[13] 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. *IEEE Journal on Selected Areas in Communications*, 14(7), pp. 1280-1297, Sept. 1996

[14] P. Madany, et. al. *JavaOS : A Standalone Java Environment*. White Paper, Sun Microsystems, May 1996

[15] M. Meyerhöfer. *Design und Implementierung eines Ethernet-Treibers und TCP/IP-Protokolls für dasJava-Betriebssystem JX*, Studienarbeit (supervised by M. Golm), University of Erlangen, IMMD4, Oct. 2000

[16] B. R. Montague. *JN: An Operating System for an Embedded Java Network Computer*, Technical Report UCSC-CRL-9629, University of California, Santa Cruz, 1996

[17] I. A. Pratt. *The User-Safe Device I/O Architecture*. Ph.D. thesis, King's College, University of Cambridge, 1997

[18] Sun Microsystems, IBM. *JavaOS for Business, Reference Manual*, Version 2.1, Oct. 1998

[19] A. Weissel. *Ein offenes Dateisystem mit Festplattensteuerung für metaXaOS*. Studienarbeit (supervised by M. Golm), Univ. of Erlangen, IMMD4, Feb. 2000