Techniques for the Design of Java Operating Systems

Godmar Back Patrick Tullmann Leigh Stoller Wilson C. Hsieh Jay Lepreau

Department of Computer Science
University of Utah

Abstract

Language-based extensible systems, such as Java Virtual Machines and SPIN, use type safety to provide memory safety in a single address space. By using software to provide safety, they can support more efficient IPC. Memory safety alone, however, is not sufficient to protect different applications from each other. Such systems need to support a *process model* that enables the control and management of computational resources. In particular, language-based extensible systems should support resource control mechanisms analogous to those in standard operating systems. They need to support the separation of processes and limit their use of resources, but still support safe and efficient IPC.

We demonstrate how this challenge is being addressed in several Java-based systems. First, we lay out the design choices when implementing a process model in Java. Second, we compare the solutions that have been explored in several projects: Alta, K0, and the J-Kernel. Alta closely models the Fluke operating system; K0 is similar to a traditional monolithic kernel; and the J-Kernel resembles a microkernel-based system. We compare how these systems support resource control, and explore the tradeoffs between the various designs.

1 Introduction

Language-based extensible systems in the form of Java virtual machines are used to implement execution environments for applets in browsers, servlets in servers, and mobile agents. All of these environments share the property that they run multiple applications at the same time. For example, a user may load applets from different Web sites into a browser; a server may run servlets from different sources; and an agent server may run agents from across the Internet. In many circumstances these applications can not be trusted, either by the server or user that runs them, or by each other. Given untrusted applications, a language-based extensible system should be able to isolate applications from one another because they may be buggy or even malicious. An execution environment for Java byte code that attempts to provide such isolation is what we term a "Java operating system."

Conventional operating systems provide the abstraction of a *process*, which encapsulates the execution of an application. A *process model* defines what a process is and what it may do. The following features are necessary in any process model for safe, extensible systems:

- *Protection*. A process must not be able to destroy the data of another process, or manipulate the data of another process in an uncontrolled manner.
- *Resource Management*. Resources allocated to a process must be separable from those allocated to other processes. An unprivileged or untrusted process must not be able to starve other processes by denying them resources.
- *Communication*. Since an application may consist of multiple cooperating processes, processes should be able to communicate with each other. Supported communication channels must be safe and should be efficient.

These requirements on processes form one of the primary tradeoffs in building operating systems, as illustrated in Figure 1. On the right-hand side, processes can be protected from each other most easily if they are on completely separate machines. In addition, managing computational resources is much simpler, since the resources are completely separate. Unfortunately, communication is more expensive between processes on different machines. On the left-hand side, communication is much cheaper, since processes can share memory directly. As a result, though, protection and accurate resource accounting become more difficult.

Operating systems research has spanned the entire range of these systems, with a primary focus on systems in the middle. Research in distributed systems and networking has focused on the right side of the figure. Research on single-address-space operating systems such as

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement numbers F30602–96–2–0269 and F30602–99–1–0503.

Contact information: {gback,tullmann,stoller,wilson,lepreau}@cs.utah.edu. Department of Computer Science, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205. http://www.cs.utah.edu/flux/java/

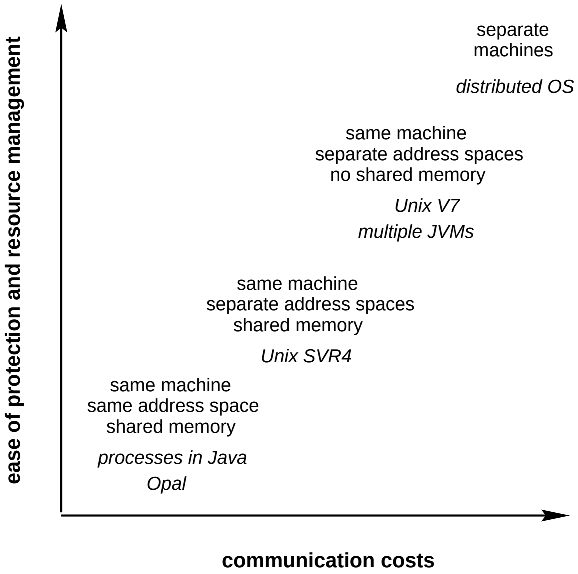

Figure 1: Trading off sharing and isolation between processes. On the right, running different processes on separate machines isolates them cleanly, but communication is more expensive. On the left, in theory a single-address-space operating system allows the most efficient communication between processes, but isolation is the most difficult.

Opal [13], as well as older work on language-based operating systems [38, 43] has focused on the left side of the figure. The reemergence of language-based extensible systems, such as SPIN [9, 32, 51] has focused attention back on the left side of the diagram. Such systems are single-address-space systems that use type safety instead of hardware memory mapping for protection. In this paper we discuss how resource management can be provided in language-based systems (in particular, in Java), and how the tradeoff between memory control and sharing is expressed in these systems.

We view Java as an example language-based extensible system for several reasons. First, Java's use of load-time bytecode verification removes the need for a trusted compiler. Second, Java's popularity makes it possible that a process model could be used widely. Finally, Java is general enough that the lessons we have learned in developing a process model for it should apply to other language-based extensible systems.

Systems such as Servlet Engines [4] and mobile agent servers [30] need to support multiple Java applications simultaneously. For safety, these systems use separate Java virtual machines to contain each application. While it is possible to run multiple Java applications and applets in separate Java virtual machines (JVMs), there are several reasons to run them within a single virtual machine. Aside from the overhead involved in starting multiple JVMs, the cost of communication between applications and applets is greater when applications are run in separate virtual machines (as suggested by Figure 1). Additionally, in small systems, such as the PalmPilot, an OS or hardware support for multiple processes might not be present. In such environments, a JVM must perform operating system tasks. A final reason to use a single JVM is that better performance should be achievable through reduction of context switching and IPC costs. Unfortunately, standard Java systems do not readily support multiprogramming, since they do not support a process abstraction. The research issues that we explore in this paper are the design problems that arise in implementing a process model in a Java virtual machine.

The hard problems in implementing a process model for Java revolve around memory management. Other hard problems in designing a process model are not unique to Java. In a Java system, protection is provided through the type safety of the language. Memory management is harder in Java than in conventional operating systems because the address space is shared. In a conventional operating system, protection is provided through a memory management unit. Process memory is inherently separated, and systems must be engineered to provide fast, efficient communication.

In this paper we compare several Java systems, and the process models that they support: Alta and K0, two projects at the University of Utah, and the J-Kernel, a project at Cornell. Alta is structured much like the Fluke microkernel [22], provides a hierarchical process model, and focuses on providing safe, efficient sharing between processes with potentially different type-spaces. K0 is structured much like a traditional monolithic kernel and focuses on stringent and comprehensive resource controls. The J-Kernel is structured like a microkernel-based system, with automatic stub generation for inter-task communication. It should not be surprising that language-based operating systems can adopt ideas from previous operating systems research: many of the design issues and implementation tactics remain the same. These systems support strong process models: each can limit the resource consumption of processes, but still permit processes to share data directly when necessary.

Section 2 overviews Java and its terminology. Section 3 describes the technical challenges in providing resource management in Java, and Section 4 compares the design and implementation of Alta, K0, and the J-Kernel. Section 5 describes related research in traditional operating systems, language-based operating systems, and Java in particular. Section 6 summarizes our conclusions.

2 Background

Java is both a high-level object-oriented language [26] and a specification for a virtual machine that executes bytecodes [32]. Java gives applications control over the dynamic linking process through special objects called *class loaders*. Class loaders support user-defined, type-safe [31] loading of new data types, object types, and code into a running Java system.

A JVM is an architecture-neutral platform for object-oriented, multi-threaded applications. The JVM provides a number of guarantees, backed by run-time verification and automatic memory management, about the memory safety of applications it executes. Specifically, the bytecodes that constitute an application must satisfy certain semantic constraints, and only the JVM-provided automatic garbage collector can reclaim storage.

A traditional JVM is structured as a trusted core, usually implemented in C, augmented with Java libraries. Together, the core and libraries implement the standard Java class libraries. Calls to the core C code are made through *native methods*.

Protecting a system from buggy or malicious code, and protecting clients from each other, requires more control than just the protection afforded by type safety. In particular, a JVM must also be able to provide security (control over data, such as information in files) and resource management (control over computational resources, such as CPU time and memory). A JVM is therefore analogous to a traditional operating system.

Although extensive investigation has been devoted to security issues in Java [25, 47], resource management has not been as thoroughly investigated. For example, a client can abuse its use of memory (either intentionally or accidentally) to compromise the overall functionality of a JVM. The design and implementation of robust Java operating systems that tightly control resource usage is an open area of research that we are addressing.

3 Resource Management

This section discusses the primary design choices for managing resources in a Java operating system. We divide the problem of resource management into three related subproblems:

- *Resource accounting*: the ability to track resource usage. Accounting can be exact or approximate, and can be fine-grained or coarse-grained.

- *Resource reclamation*: the ability to reclaim a process's resources when it terminates. Complex allocation management policies and flexible sharing policies can make reclamation difficult. Reclamation can be immediate or delayed.

- *Resource allocation*: the ability to allocate resources to processes in a way that does not allow processes to violate imposed resource limits. Allocation mechanisms should be fair and should not incur excessive overhead.

We discuss each of the previous issues with respect to several computational resources: memory, CPU usage, and network bandwidth. We do not currently deal with managing the use of persistent storage, because there is little specific to the management of such storage in language-based systems. Since Java encourages direct sharing of memory, the primary difficulty in supporting a process model in Java is in isolating processes' memory from one another.

3.1 Memory

The issues of memory accounting, memory reclamation and memory allocation within a Java process model can be divided into two discussions: memory accounting and the impact of the inter-process sharing model; and allocation and deallocation policies.

3.1.1 Sharing Model

A *sharing model* defines how processes can share data with each other. In a Java operating system, three choices are possible: copying, direct sharing, and indirect sharing. The sharing model in standard Java (without processes) is one of *direct sharing*: objects contain pointers to one another, and a thread accesses an object's fields via offsets from the object pointer. In Java with processes, the choice of sharing model affects how memory accounting and process termination (resource reclamation) can be implemented.

Copying. Copying is the only feasible alternative when address spaces are not shared: for example, when two processes are on different machines. Copying was the traditional approach to communication in RPC systems [10], although research has been aimed at reducing the cost of copying for same-machine RPC [8]. Mach [1], for example, used copy-on-write and out-of-line data to avoid extra copies.

If data copying is the only means of communication between processes, then memory accounting and process termination are straightforward. Processes do not share any objects, so a process's objects can be reclaimed immediately; there can be no ambiguity as to which process owns an object. Of course, the immediacy of reclamation depends on the garbage collector's involvement in memory accounting: reclaiming objects in Java could require a full garbage collection cycle.

In Java, the use of copying as a communication mechanism is unappealing because it violates the spirit of the Java sharing model, and because it is slow. There is enough support in a JVM for one process to safely share a trusted object with an untrusted peer; not leveraging this support for fine-grained sharing in a Java process model neutralizes the major advantage of using a language-based system. On the other hand, in a system that only supports copying data between processes, process termination and per-process memory accounting are much simpler.

Direct Sharing. Since Java is designed to support direct sharing of objects, another design option is to allow direct sharing *between* processes. Interprocess sharing of objects is then the same as intraprocess sharing. Direct sharing in single-address-space systems is somewhat analogous to shared memory (or shared libraries) in separate-address-space systems, but the unit of sharing is much finer-grained.

If a system supports direct sharing between processes, then process termination and resource reclamation are complicated. If a process exports a directly shared object, that object cannot be reclaimed when the exporting process is terminated. The type-safety guarantees made by the Java virtual machine cannot be violated, so any reference to the object must remain valid. To reclaim the object would require that all references to the object be located. In the presence of C code, such a search is impossible to do without extensive compiler support. Therefore, in order to support resource reclamation when a process is killed, either direct sharing needs to be restricted or the system must guarantee that all outstanding references to any object can be located.

Indirect Sharing. An alternative to direct sharing is *indirect sharing*, in which objects are shared through a level of indirection. When communicating a shared object, a direct pointer to that object is not provided. Instead, the process creates a proxy object, which encapsulates a reference to the shared object. It then passes a pointer to the proxy object. Proxies are system-protected objects. In order to maintain indirect sharing (and prevent direct sharing), the system must ensure that there is no way for a client to extract a direct object pointer from a proxy. Such second-class handles on objects are commonly called capabilities; analogues in traditional operating systems include file descriptors and process identifiers.

Compared to direct sharing, indirect sharing is less efficient, since an extra level of indirection must be followed whenever an interprocess call occurs. The advantage of indirection, however, is that resource reclamation is straightforward. All references to a shared object can be revoked, because the level of indirection enables the system to track object references. Therefore, when a process is killed, all of its shared objects can be reclaimed immediately. As with copying, immediate revocation is subject to the cost of a full garbage collection cycle in Java.

3.1.2 Allocation and Deallocation

Without page-protection hardware, software-based mechanisms are necessary to account for memory in a Java operating system. Every allocation (or group of allocations) must be checked against the allocating process's heap limit. Stack frame allocations must be checked against the executing thread's stack limits.

Memory is necessarily reclaimed in Java by an automatic garbage collector [50]. It seems obvious to use the garbage collector to do memory accounting as well. The simplest mechanism for keeping track of memory is to have the allocator debit a process that allocates memory, and have the garbage collector credit a process when its memory is reclaimed.

In the presence of object sharing (whether direct or indirect), other memory accounting schemes are possible. For example, a system could conceivably divide the "cost" of an object among all the parties that keep the object alive. This model has the drawback that a process can be spontaneously charged for memory. For example, suppose a process acquires a pointer to a large object, and is initially only charged for a small fraction of the object's memory. If the other sharers release their references, the process may asynchronously run out of memory, because it will be forced to bear the full cost of the entire object.

Another potential scheme is to allow processes to pass memory "credits" to other processes. For example, a server could require that clients pass several memory credits with each request to pay for the resources the server allocates. Such a scheme is analogous to economic models that have been proposed for resource allocation [46]. A similar system might permit a process to transfer the right to allocate under its allowance. A similar effect is possible in the simple allocator-pays model by having a client allocate an object and pass it to the server to be "filled in."

An important issue in managing memory is the relationship between allocation and accounting schemes. In particular, a system that charges per object, but allocates memory in larger chunks, might be subject to a fragmentation attack. A process with a small budget could accidentally or maliciously cause the allocation of a large number of blocks. One solution is to provide each process with its own region of physical or virtual addresses from which to allocate memory. While this solution guarantees accurate accounting for internal fragmentation, it may introduce external fragmentation.

3.2 CPU Usage

The two mechanisms necessary for controlling CPU usage are accounting and preemption. The system must be able to account accurately for the CPU time consumed by a thread. The system must also be able to prevent threads from exceeding their assigned CPU limits by preempting (or terminating) them. Desirable additional features of CPU management are multiple scheduling policies, user-providable policies, and support for real-time policies.

3.2.1 CPU Accounting

The accuracy of CPU accounting is strongly influenced by the way in which processes obtain services. If services are implemented in libraries or as calls to a monolithic kernel, accounting simply amounts to counting the CPU time that a thread accrues.

CPU accounting is difficult with shared system services, where the process to bill for CPU usage is not easily determined. Examples of such services include garbage collection and interrupt processing for network packets. For such services, the system needs to have a means of deciding what process should be charged.

Garbage Collection. The simplest accounting policy for garbage collection is to treat it as a global system service. Unfortunately, such a policy is undesirable because it opens the system to denial-of-service attacks. For example, a process could trigger garbage collections frequently so as to slow down other processes. In addition, treating garbage collection as a universal service allows priority inversion to occur. If a low-priority thread allocates and deallocates large chunks of memory, it may cause a high-priority thread to wait for a garbage collection.

We see two approaches that can be taken to solve this problem. First, the garbage collector could charge its CPU usage to the process whose objects it is traversing. However, since this solution would require fine-grained measurement of CPU usage, its overhead would likely be prohibitive.

The second alternative is to provide each process with a heap that can be garbage-collected separately, such that the GC time can be charged to the owning process. Independent collection of different heaps requires special treatment of inter-heap references if direct sharing is to be allowed. In addition, distributed garbage collection algorithms might be necessary to collect data structures that are shared across heaps.

Packet Handling. Interrupt handling is another system service, but its behavior differs from that of garbage collection, because the "user" of an external interrupt cannot be known until the interrupt is serviced. The goal of the system should be to minimize the time that is needed to identify the receiver, as that time cannot be accounted for. For example, Druschel and Banga [20] showed how packets should be processed by an operating system. They demonstrated that system performance can drop dramatically if too much packet processing is done at interrupt level, where normal process resource limits do not apply. They concluded that systems should perform *lazy receiver processing* (LRP), which is a combination of early packet demultiplexing, early packet discard, and processing of packets at the receiver's priority. The use of LRP improves traffic separation and stability under overload.

3.2.2 Preemption and Termination

Preempting a thread that holds a system lock could lead to priority inversion. As a result, it is generally better to let the thread exit the critical section before it is preempted. Similarly, destroying a thread that holds a system lock could lead to consistency problems if the lock is released or deadlock if the lock is never released. Preemption and termination can only be safe if the system can protect critical sections against these operations.

By making a distinction between non-preemptible, non-killable code, and "regular" code a Java system effectively makes a distinction between user-mode and kernel-mode [5]. In a traditional, hardware-based system, entry to (and exit from) the kernel is explicit: it is marked with a trap instruction. The separation between kernel and user code is not as clear in Java, since making a call into the kernel might be no different than any other method invocation.

In addition to providing support for non-preemptible (and non-killable) critical sections, a Java operating system needs to have a preemption model within its kernel. The design choices are similar to those in traditional systems. First, the kernel could be single-threaded, and preemption would only occur outside the kernel. Alternatively, the system can be designed to allow multiple user threads to enter the kernel. In the latter case, preemption might be more immediate, but protecting the kernel's data structures incurs additional overhead.

3.3 Network Bandwidth

While bandwidth can be important for certain applications of Java, such as active networks [48], there is little about controlling network bandwidth that is specific to Java. A range of approaches, from byte counting to packet scheduling, is available.

The J-Kernel experimented with a special version of the WinSocket DLL to count bytes in outgoing network streams. The implementations of K0 and Alta could easily provide access to packet scheduling facilities provided by the infrastructure on which they run.

4 Comparison

In this section we describe in detail our two prototype systems, K0 and Alta, and the prototype of a third

Java operating system, J-Kernel, that has been built at Cornell. These systems represent different sets of design tradeoffs:

- The J-Kernel disallows direct sharing between processes, and uses bytecode rewriting to support indirect sharing. Because it consists of Java code only, it is portable across JVMs. As a result, though, the resource controls that the J-Kernel provides are approximate. J-Kernel IPC does not involve a rendezvous: a thread migrates across processes, which can delay termination.

- K0 partitions the Java heap so as to isolate resource consumption. In addition, restricted direct sharing is permitted through the system heap. Garbage collection techniques are put to interesting use to support this combination. CPU inheritance scheduling is used as a framework for hierarchical scheduling of CPU time.

- Alta uses hierarchical resource management, which makes processes responsible for (and gives them the capability of) managing their subprocesses' resources. Direct sharing between sibling processes is permitted because their parent is responsible for their use of memory. The hierarchy also is a good match for CPU inheritance scheduling.

4.1 J-Kernel

The J-Kernel [15, 28] implements a microkernel architecture for Java programs, and is itself written in Java. It supports multiple protection domains that are called tasks. Names are managed in the J-Kernel through the use of *resolvers*, which map names onto Java classes. When a task creates a subtask, it can specify which classes the subtask is allowed to access. Class loaders are used to give tasks their own name spaces.

4.1.1 System Model

Communication in the J-Kernel is based on capabilities. Java objects can be shared indirectly by passing a pointer to a *capability* object through a "local RMI" call. The capability is a trusted object containing a direct pointer to the shared object. Because of the level of indirection through capabilities to the shared object, the capabilities can be revoked. A capability can only be passed if two tasks share the same class. Making a class shared is an explicit action that forces two class loaders to share the class.

All arguments to inter-task invocations must either be capabilities, or be copied in depth, i.e., the complete tree of objects that are reachable from the argument via direct references must be copied recursively. By default, standard Java object serialization is used, which involves marshaling into and unmarshaling from a linear byte buffer. To decrease the cost of copying, a fast copy mechanism is also provided. Specialized code for a class creates a direct copy of an object's fields. Both the specialized fast copy code and the stubs needed for cross-domain calls are generated dynamically.

The J-Kernel supports thread migration between tasks: cross-task communication is not between two threads. Instead, a single thread makes a method call that logically changes protection domains. Therefore, a full context switch is not required. To prevent malicious callers from damaging a callee's data structures, each task is only allowed to stop a thread when the thread is executing code in its own process. This choice of system structure requires that a caller trust all of its callees, because a malicious or erroneous callee might never return.

4.1.2 Resource Management

The J-Kernel designers made the explicit decision not to build their own JVM. Instead, the J-Kernel is written entirely in Java. As a result of this decision, the J-Kernel designers limited the precision of their resource control mechanisms. The lack of precision occurs because the JVM that runs under the J-Kernel cannot know about processes. As a result, it cannot account for the resources that it consumes on behalf of a process.

Memory Management. In order to account for memory, the J-Kernel rewrites the bytecode of constructors and finalizers to charge and credit for memory usage. Such a scheme does not take fragmentation into account. In addition, memory such as that occupied by just-in-time compiled code is hard to account for.

CPU Management. The NT version of the J-Kernel uses a kernel device driver to monitor the CPU time consumed by a thread. This mechanism is reactive: threads can only be prevented from consuming further resources after they already exceeded their limits. In addition, it is difficult to add custom scheduling policies for tasks.

4.1.3 Implementation Status

A version of the J-Kernel that does not support resource controls is freely available from Cornell [45]. The advantage of their implementation approach is a high degree of portability: the J-Kernel can run on most JVMs. Since it uses class reloading, there are some dependencies on the specific interpretation of gray areas in the Java language specification.

The J-Kernel is distributed with two additional pieces of software. The first is JOS, which uses the J-Kernel to provide support for servers. The second is J-Server, a Web server that safely runs client-provided Java code.

4.1.4 Summary

The J-Kernel adopts a capability-based model that disallows direct sharing between tasks. As a result, its capabilities are directly revocable, and memory can be completely reclaimed upon task termination. In addition, the J-Kernel exploits the high-level nature of Java's bytecode representation to support the automatic creation of communication channels.

4.2 K0

K0's design loosely follows that of a traditional monolithic kernel. K0 is oriented toward complete resource isolation between processes, with the secondary goal of allowing direct sharing. As in a traditional operating system, each process is associated with a separate heap, and sharing occurs only through a special, shared heap.

K0 can run most JDK 1.1 applications without modification. It cannot run those that assume that they were loaded by a "null" class loader.

4.2.1 System Model

A K0 process consists of a name space, a heap, and a set of threads. K0 relies on class loaders to provide different processes with separate name spaces. Each process is associated with its own class loader, which is logically considered part of the kernel. To provide different processes with their own copies of classes that contain static members, K0 loads classes multiple times. Unlike other JVMs, K0 allows safe reloading of all but the most essential classes, such as `Object` or `Throwable`. To reduce a process's memory footprint, classes that do not contain shared data may be shared between processes, akin to how different processes map the same shared library into their address spaces in a traditional OS. However, since all shared classes must occupy a single name space, sharing is a privileged operation.

Threads access kernel services by calling into kernel code directly. The kernel returns references to kernel objects that act as capabilities to such things as open files and sockets. In order to support the stopping or killing of threads, K0 provides a primitive that defers the delivery of asynchronous exceptions until a well-defined cancellation point within the kernel is reached. This primitive does not solve all of the problems with thread termination, but it enables the kernel programmer to safely cancel user processes without compromising the integrity of the kernel.

Each K0 process is associated with its own heap. Shared classes and other shared data reside in a distinct heap called the shared heap. K0 supports comprehensive memory accounting that takes internal allocations by the JVM into account. Because K0 controls inter-heap references, it is able to support independent collection of individual heaps and it is able to charge garbage collection time to the appropriate processes. The use of separate heaps has the additional benefit of allowing K0 to avoid priority inversions: it is not necessary to stop higher-priority threads in other processes when performing a collection.

4.2.2 Resource Management

Memory Management. The use of separate heaps simplifies memory accounting because each heap is subject to its own memory budget, and simplifies CPU accounting because each heap can be collected separately. In order to preserve these benefits while still allowing for efficient process communication, K0 provides limited direct sharing between heaps. If two processes want to share an object, two criteria must be met. First, the processes must share the type of the object. Second, the object must be allocated in the shared heap. The creation of a shared object is a privileged operation. An object in a process heap can refer to a shared object, and a shared object can refer to an object in a process heap. However, K0 explicitly disallows direct sharing between objects in separate processes' heaps, and uses write barriers [50] to enforce this restriction.

Acquiring a reference to a shared object is only possible by invoking the system, and K0 ensures that resources allocated within the system heap on behalf of an process are subject to a specific limit. For instance, each process may only open a certain number of files, since the kernel part of a file descriptor is allocated in system space. K0 must be careful to not hand out references to objects that have public members, or objects it uses for internal synchronization.

Shared objects have a restricted execution model. During their construction, they can allocate objects on the system heap. After the objects are constructed, threads that methods on them are subject to normal segment limits: if a thread attempts to use a shared object to write a reference to a foreign heap into its own heap, a segmentation violation error will be triggered.

To allow for separate garbage collection of individual heaps, K0 implements a form of distributed GC [37]. For each heap, K0 keeps a list of *entry items* for objects to which external references exist. An entry item consists of a pointer to the local object and a reference count. The reference count denotes the number of foreign heaps that have links to that object. The garbage collector of a heap treats all entry items as roots. For each heap, K0 also keeps a list of *exit items* for non-local objects to which the heap refers. An exit item contains a pointer to the entry item of the object to which it refers. At the end of a garbage collection cycle, unreferenced exit items are collected and the reference counts in the corresponding entry items are decremented. An entry item can be reclaimed if its reference count reaches zero.

Write barriers are used to automatically create and update exit and entry items, as well as to maintain the heap reference invariants described previously. If a write barrier detects a reference that is legal, it will lookup and create the corresponding exit item for the remote object. In turn, the corresponding entry item in the foreign heap is updated. The same write barrier is used to prevent the passing of illegal cross-heap references. If the reference that would be created by a write is illegal, a segmentation violation error is thrown. The use of a write barrier is similar to the use of write checks in Omniware [2]. Although it may seem odd to use another protection mechanism (software fault isolation) in a type-safe system, the motivation is resource management, not memory safety.

Finally, to improve the use of the JVM's memory as a whole, K0 does not reserve non-overlapping, contiguous memory regions for each heap. Instead, memory accounting is done on a per-block basis. Small objects are stored in page-sized blocks, whereas larger objects are stored in dedicated blocks. Heaps receive new memory in blocks, and the garbage collector only reimburses a heap if it frees a whole block.

CPU Management. In traditional Java, each thread belongs to a thread group. Thread groups form a hierarchy in which each thread group has a parent group. The initial thread group is the root of the group hierarchy. K0 adapts the thread group classes such that all threads belonging to a process are contained in a subtree. Process threads cannot traverse this tree past the root of this subtree.

K0 combines the thread group hierarchy with CPU inheritance scheduling [23]. CPU inheritance scheduling is based on a directed yield primitive: a scheduler thread donates CPU time to a specific thread by yielding to it, which effectively schedules that thread. Since the receiver thread may in turn function as a scheduler thread, scheduler hierarchies can be built. Each non-root thread has an associated scheduler thread that is notified when that thread is runnable. A scheduler may use a timer to revoke its donation, which preempts a scheduled thread. Using CPU inheritance scheduling allows K0 to do two things. First, K0 can provide each process with its own scheduler that may implement any process-specific policy to schedule the threads in that process. Second, thread groups within processes may hierarchically schedule the threads belonging to them.

Each thread group in K0 is associated with a scheduler, which is an abstract Java class in K0. Different policies are implemented in different subclasses. At the root of the scheduling hierarchy, K0 uses a fixed priority policy to guarantee that the system heap garbage collector is given the highest priority. At the next level, a stride scheduler divides CPU time between processes. To provide compatibility with traditional Java scheduling, the root thread group of each process by default is associated with a fixed-priority scheduler that is a child of the stride scheduler.

4.2.3 Implementation Status

We have prototyped a K0 kernel that is composed of a modified JVM that is based on Kaffe 1.0beta1. It is supplemented by classes in binary format from JavaSoft's JDK 1.1.5, and a package of privileged classes that replace part of the core java packages. K0 ran as a stand-alone kernel based on the OSKit [21] (a suite of components for building operating systems). Additionally, K0 ran in user-mode with libraries that simulate certain OSKit components such as interrupt handling and raw device access. We implemented separate heaps, as well as write barriers. Our initial prototype did not support separate garbage collection nor class garbage collection. The prototype supported CPU inheritance scheduling in the way described above, although it only supported schedulers implemented as native methods in C. We implemented four different policies: fixed-priority, rate-monotonic scheduling, lottery, and stride-scheduling.

We are currently working on a successor system called KaffeOS, which is based on a much improved base JVM, supports separate garbage collection, and will provide full resource reclamation.

4.2.4 Summary

K0's design is oriented towards complete resource isolation between processes, with the secondary goal of allowing direct sharing. By giving each process a separate heap, many memory and CPU management resource issues become simpler. Sharing occurs through a shared system heap, and distributed garbage collection techniques are used to safely maintain sharing information.

4.3 Alta

Alta [44] is an extended Java Virtual Machine that provides a hierarchical process model and system API modeled after that provided by the Fluke microkernel. Fluke supports a *nested process model* [22], in which a process can manage all of the resources of child processes in much the same way that an operating system manages the resources of its processes. Memory management and CPU accounting are explicitly supported by the Alta system API. Higher-level services, such as network access and file systems, are managed by servers, with which applications communicate via IPC. Capabilities provide safe, cross-process references for communication.

Each process has its own root thread group, threads, and private copies of static member data. Per-process memory accounting in Alta is comprehensive. For access control purposes, Alta expands the Fluke model by providing processes with the ability to control the classes used by a sub-process. Alta also extends the Java class model in that it allows a process to rename the classes that a subprocess sees. As a result, a process can interpose on all of a subprocess' interfaces.

The Alta virtual machine does not change any of the interfaces or semantics defined by the JVM specification. Existing Java applications, such as `javac` (the Java compiler), can run unmodified as processes within Alta.

4.3.1 System Model

Communication in Alta is done through an IPC system that mimics the Fluke IPC system. Inter-process communication is based on a half-duplex, reversible, client-server connection between two threads (which may reside in different processes). Additionally, Alta IPC provides immediate notification to the client or server if the peer at the other end of the connection is terminated or disconnects.

Alta permits sibling processes to share objects directly. Objects can be shared by passing them through IPC. Sharing is only permitted for objects where the two processes have consistent views of the class name space. Enforcing this requirement efficiently requires that the classes involved are all final, i.e., that they cannot be subclassed. While this is somewhat restrictive, all of the primitive types — such as `byte[]` (an array of bytes) and `java.lang.String` — and many of the core Alta classes meet these requirements.

4.3.2 Resource Management

The strongest feature of the Alta process model is the ability to "nest" processes: every process can manage child processes in the same way the system manages processes. Resource management in Alta is strictly hierarchical. Any process can create a child process and limit the memory allowance of that process.

Memory Management. Alta supports memory accounting through a simple allocator-pays scheme. The garbage collector credits the owning process when an object is eventually reclaimed. When a process is terminated, any existing objects are "promoted" into its parent's memory. Thus, it the responsibility of the parent process to make sure that cross-process references are not created if full memory reclamation is necessary upon child process termination. It is important to note that Alta enables a process to prevent child processes from passing Java object references through IPC.

Memory reclamation is simple if a process only passes references to its children. In the nested process model, when a process is terminated all of its child processes are necessarily terminated also. Therefore, references that are passed to a process's children will become unused.

To support clean thread and process termination, Alta uses standard operating system implementation tricks to prevent the problem of threads terminated while executing critical system code, just like in K0. For example, to avoid stack overflows while executing system code, the entry layer will verify sufficient space is available on the current thread stack. This check is analogous to the standard technique of pre-allocating an adequate size stack for in-kernel execution in traditional operating systems. Additionally, Alta is structured to avoid explicit memory allocations within "kernel mode." A system call can allocate objects before entering the kernel proper. Thus, all allocation effectively happens in "user mode." Since the notion of the system code entry layer is explicit, some system calls (for example, `Thread.currentThread()`) never need enter the kernel.

CPU Management. Alta provides garbage collection as a system service. This leaves Alta open to denial-of-service attacks that generate large amounts of garbage—which will cause the garbage collector to run. Given the memory limits on processes, and limits on the CPU usage of a process, GC problems like this can be mitigated.

4.3.3 Implementation Status

Alta's implementation is based on a JDK 1.0.2-equivalent JVM and libraries (Kaffe [49] version 0.9.2 and Kore [14] version 0.0.7, respectively). The bulk of the system is implemented entirely in Java. The internals of the VM were enhanced to support nested processes. A number of the core library classes were modified to use Alta primitives and to make class substitution more effective. In addition to `javac`, Alta supports simple applications that nest multiple children and control their class name spaces, along with a basic shell and other simple applications.

In terms of code sharing, a process in Alta is analogous to a statically linked binary in a traditional systems — each process has its own JIT'd version of a method. The Kaffe JIT could be modified to provide process-independent, sharable code, just as compilers can generate position-independent code for shared libraries. Like the version of Kaffe on which it is based, Alta does not yet support garbage collection of classes.

Alta does not implement CPU inheritance scheduling. Because Alta and K0 share a common code base, the CPU inheritance scheduling that is implemented in the K0 should be easy to migrate to Alta. In addition, like K0, Alta runs as a regular process on a traditional operating system and could be made to run on top of bare hardware using the OSKit.

4.3.4 Summary

Alta implements the Fluke nested process model and API in a Java operating system. It demonstrates that the nested process model can provide Java processes with flexible control over resources. Because of the hierarchical nature of the model, direct sharing between siblings can be supported without resource reclamation problems.

4.4 Performance Evaluation

We ran several microbenchmarks on our two prototype systems, Alta and K0, and a port of the J-Kernel to Kaffe to measure their baseline performance. These benchmarks demonstrate that no undue performance penalties are paid by "normal" Java code, in any of these systems, for supporting processes. In addition, they roughly compare the cost of IPC and Java process creation in all three systems.

The Alta, J-Kernel, and basic Kaffe tests were performed on a 300 MHz Intel Pentium II with 128MB of SDRAM. The system ran FreeBSD version 2.2.6, and was otherwise idle. The K0 tests were performed on the same machine, but K0 was linked to the OSKit and running without FreeBSD.

Table 1 shows the average time for a simple null instance method invocation, the average cost of allocating a `java.lang.Object`, the average overhead of creating and starting a Thread object, and the average cost of creating a Throwable object. All of the benchmarks were written to avoid invocation of the GC (intentional or unintentional) during timing. For K0 and Alta the benchmarks were run as the root task in the system. For the J-Kernel, the benchmarks were run as children of the J-Kernel RootTask, `cornell.slk.jkernel.std.Main`.

None of the systems significantly disrupts any of the basic features of the virtual machine. Previously published results about the J-Kernel [28] used Microsoft's Java virtual machine, which is significantly faster than Kaffe. The Alta null thread test is significantly more expensive than the basic Kaffe test because Alta threads maintain additional per-thread state for IPC, process state, and blocking.

Table 2 measures the two critical costs of adding a process model to Java. The first column lists the overhead of creating a new process, measured from the time the parent creates the new process to the time at which the new process begins its `main` function. The Kaffe row lists the time required for Kaffe to fork and exec a new Kaffe process in FreeBSD. The J-Kernel supports a more limited notion of process—J-Kernel processes do not require an active thread—so the J-Kernel test simply creates a passive Task and seeds it with a simple initial object.

The subsequent columns of Table 2 show the time required for cross-task communication. Alta IPC is significantly slower because it is a rendezvous between two threads and uses ports, whereas J-Kernel IPC is simply cross-process method invocation. K0 IPC is implemented using a shared rendezvous object and is based directly on wait/notify. The IPC cost in K0 reflects its unoptimized thread package that is different than the thread package in the other JVMs.

Our performance results indicate that our systems need substantial optimization in order to realize the performance potential of language-based operating systems. The performance benefits from fine-grained sharing in software can be dominated by inefficiencies in the basic JVM implementation. As the difference to previously published J-Kernel results demonstrates, the future performance of Java systems will likely be spurred by advances in just-in-time compilation, which is orthogonal to the research issues we are exploring.

To analyze the implementation costs of our decision to build our own JVM, we examined each system in terms of useful lines of code (i.e., non-blank, non-comment lines of source). As a reference point, the original version of Kaffe v0.9.2 contains 10,000 lines of C, while Kaffe v1.0beta1 is comprised of just over 14,000 lines of C and 14,000 lines of Java. (Much of this increase is due to the move from JDK 1.0 to JDK 1.1.) Alta is comprised of 5,000 lines of Java and adds approximately 5,000 lines of C to Kaffe v0.9.2 (a significant fraction of this C code consists of features from later versions of Kaffe that we ported back to Kaffe v0.9.2). K0 adds approximately 1,000 lines of C code to the virtual machine and almost 2,000 lines of Java code to the basic libraries. The additional C code consisted of changes to the garbage collector to support K0's separate heaps.

In comparison, the J-Kernel consists of approximately 9,000 lines of Java. Building the J-Kernel as a layer on top of a JVM was probably an easier implementation path than building a new JVM. The primary difficulty in building the J-Kernel probably lay in building the dynamic stub generator.

5 Related Work

Several lines of research are related to our work. First, the development of single-address-space operating systems — with protection provided by language or by hardware — is a direct antecedent of work in Java. Second, a great deal of research today is directed at building operating system services in Java.

5.1 Prior Research

A great deal of research has been done on hardware-based single-address-space operating systems. In

Virtual Machine	Method Invocation	Object Creation	Null Thread Test	Exception Creation
Kaffe 1.0beta1	0.16μs	1.9μs	480μs	12μs
K0	0.16μs	3.1μs	725μs	18μs
Alta	0.16μs	2.5μs	1030μs	15μs
Kaffe 0.10.0	0.17μs	1.8μs	470μs	10μs
J-Kernel	0.17μs	1.8μs	480μs	29μs

Table 1: Despite the fact that we have five distinct Java virtual machines based around different versions of the Kaffe virtual machine, base performance of the versions are not very different. The J-Kernel is run on Kaffe 0.10.0, because of deficiencies in object serialization in Kaffe 1.0beta1.

Virtual Machine	Process Creation	Null IPC	3-integer request	100-byte String request
Alta	120ms	90μs	109μs	138μs
K0	89ms	57μs	57μs	183μs
J-Kernel	235ms	2.7μs	2.7μs	27μs
Kaffe	300ms	N/A	N/A	N/A

Table 2: Process Tests. Note that numbers in the first column are reported in ms, while the other columns are reported in μs. Alta and K0 IPC is between separate threads while the J-Kernel IPC uses cross-process thread migration. The 3-integer request and 100-byte String request operations include the time to marshal and unmarshal the request. The J-Kernel uses object serialization to transmit a String while K0 and Alta use hand-coded String marshal and unmarshal code.

Opal [13], communication was accomplished by passing 256-bit capabilities among processes: a process could *attach* a memory segment to its address space so that it could address the memory segment directly. Because Opal was not based on a type-safe language, resource allocation and reclamation was coarse-grained, and based on reference counting of segments.

Many research projects have explored operating systems issues within the context of programming languages. For example, Argus [33] and Clouds [16] explored the use of transactions within distributed programming languages. Other important systems that studied issues of distribution include Eden [3], Emerald [11], and Amber [12]. These systems explored the concepts underlying object migration, but did not investigate resource management.

Language-based operating systems have existed for many years. Most of them were not designed to protect against malicious users, although a number of them support strong security features. None of them, however, provide strong resource controls. Pilot [38] and Cedar [43] were two of the earliest language-based systems. Their development at Xerox PARC predates a flurry of research in the 1990's on such systems.

Oberon [51] has many of Java's features, such as garbage collection, object-orientation, strong type-checking, and dynamic binding. Unlike Java, Oberon is a non-preemptive, single-threaded system. Background tasks like the garbage collector are implemented as calls to procedures, where "interruption" can only occur between top-level procedure calls.

A related project, Juice [24] provides an execution environment for downloaded Oberon code (just as a JVM provides an execution environment for Java). Juice is a virtual machine that executes "binaries" in its own portable format: it compiles them to native code during loading, and executes the native code directly. The advantage of Juice is that its portable format is faster to decode and easier to compile than Java's bytecode format.

SPIN [9] is an operating system kernel that lets applications load extensions written in Modula-3 that can extend or specialize the kernel. As with Java, the type safety of Modula-3 ensures memory safety. SPIN supports dynamic interposition on names, so that extensions can have different name spaces.

Inferno [19], an OS for building distributed services, has its own virtual machine called Dis and its own programming language called Limbo. Inferno is a small system that has been ported to many architectures: it has been designed to run in resource-limited environments, such as set-top boxes. In order to minimize garbage collection pauses, Inferno uses reference counting to reclaim memory, avoiding a number of accounting issues related to garbage collection in an operating system.

VINO is a software-based (but not language-based) extensible system [40] that addresses resource control issues by wrapping kernel extensions within transactions. When an extension exceeds its resource limits, it can be safely aborted (even if it holds kernel locks), and its resources can be recovered.

5.2 Java-Based Research

Besides Alta, K0, and the J-Kernel, a number of other research systems have explored (or are exploring) the problem of supporting processes in Java.

Balfanz and Gong [6] describe a multi-processing JVM developed to explore the security architecture ramifications of protecting applications from each other, as opposed to just protecting the system from applications. They identify several areas of the JDK that assume a single-application model, and propose extensions to the JDK to allow multiple applications and to provide inter-application security. The focus of their multi-processing JVM is to explore the applicability of the JDK security model to multi-processing, and they rely on the existing, limited JDK infrastructure for resource control.

IBM [18] released a JVM for its OS/390 family of systems that is targeted towards server applications such as Enterprise Java Beans. Their system puts each transaction into a separate worker JVM that initialize from and execute out of a shared heap. This shared heap holds those classes and objects that are expected to survive a transaction. Worker JVMs that leave no resources behind can be reused for multiple transactions. If a transaction does leave resources behind, the worker JVM process is terminated and the OS is used to free those resources. IBM's motivation for providing a quasi-process model in Java are faster startup times attributable to the savings in class loading and processing, which increases transaction throughput. However, they do not consider the case of malicious and uncooperative applications because there is no control over what data individual applications can store on the shared heap. In addition, the shared heap is not garbage collected.

One approach to resource control is to dedicate an entire machine to the execution of client code. For instance, AT&T's "Java Playground" [34] and Digitivity's "CAGE" Applet Management System [17] define special Java applet execution models that require applets to run on dedicated, specially protected hosts. This execution model imposes extremely rigid limits on mobile code, by quarantining applets on isolated hosts. As a result, richer access is completely disallowed. Although the above-mentioned systems guarantee the integrity of the JVM, they do not provide any inter-applet guarantees beyond that offered by the underlying "stock" JDK. These systems are similar to Kimera [41], which uses dedicated servers to protect critical virtual machine resources (e.g., the bytecode verifier) but not to protect applications from each other.

Luna [29] is a recent system from one of the J-Kernel developers. Luna extends the Java language and runtime with explicit, revocable remote pointers. Remote pointers can be dynamically revoked, and processes can safely share fine-grained data without compromising type-safety.

Sun's original JavaOS [42] was a standalone OS written almost entirely in Java. It is described as a first-class OS for Java applications, but appears to provide a single JVM with little separation between applications. It is being replaced by a new implementation termed "JavaOS for Business" that also only runs Java applications. "JavaOS for Consumers" is built on the Chorus microkernel OS [39] in order to achieve real-time properties needed in embedded systems. Both of these systems require a separate JVM for each Java application, and all run in supervisor mode.

Joust [27], a JVM integrated into the Scout operating system [35], provides control over CPU time and network bandwidth. To do so, it uses Scout's path abstraction. However, Joust does not provide memory limits.

The Open Group's Conversant system [7] is yet another project that modifies a JVM to provide processes. It provides each process with a separate address range (within a single Mach task), a separate heap, and a separate garbage collection thread. Conversant does not support sharing between processes, unlike our systems and the J-Kernel. Its threads are native Mach threads that support POSIX real-time semantics. Conversant provides some real-time services. Another real-time system, PERC [36], extends Java to support real-time performance guarantees. The PERC system analyzes Java bytecodes to determine memory requirements and worst-case execution time, and feeds that information to a real-time scheduler.

6 Conclusions

In order to support multiple applications, a Java operating system must control computational resources. The major technical challenges that must be addressed in building such a system are managing memory and CPU usage for shared code. Some of these challenges can be dealt with by adapting techniques used in conventional systems to language-based systems. Other challenges can be dealt with by adapting language technology, such as garbage collection, to fit into an operating system framework.

We have described two prototype Java operating systems that are being built at Utah: Alta and K0. These two prototypes and Cornell's J-Kernel illustrate tradeoffs that can be made in terms of system structure, resource management, and implementation strategies. We have shown that many design issues from conventional operating systems resurface in the structural design of Java operating systems. Java operating systems can be built with monolithic designs, as K0; or they can be built with microkernel designs, as Alta or the J-Kernel. Finally, we have shown how garbage collection techniques can be used to support resource management for Java processes.

Acknowledgments

We thank Kristin Wright, Stephen Clawson, and James Simister for their efforts in helping us with the results. We thank Eric Eide for his great help in editing and improving the presentation of this material, and Massimiliano Poletto for his comments on drafts of this paper. We thank Chris Hawblitzel for his clarifications of how the J-Kernel works. Finally, we thank the Flux group for their work in making the OSKit, which was used in some of this work.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation (PLDI)*, pages 127–136, May 1996.

[3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, Jan. 1985.

[4] The Java Apache project. http://java.apache.org, Apr. 2000.

[5] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.

[6] D. Balfanz and L. Gong. Experience with secure multiprocessing in Java. In *Proc. of the Eighteenth International Conf. on Distributed Computing Systems*, May 1998.

[7] P. Bernadat, L. Feeney, D. Lambright, and F. Travostino. Java sandboxes meet service guarantees: Secure partitioning of CPU and memory. Technical Report TOGRI-TR9805, The Open Group Research Institute, June 1998.

[8] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[10] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[11] A. P. Black, N. Huchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans. on Software Engineering*, SE-13(1):65–76, 1987.

[12] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[13] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[14] G. Clements and G. Morrison. Kore — an implementation of the Java(tm) core class libraries. ftp://sensei.co.uk/misc/kore.tar.gz OR http://www.cs.utah.edu/projects/flux/java/kore/.

[15] G. Czajkowski, C.-C. Chang, C. Hawblitzel, D. Hu, and T. von Eicken. Resource management for extensible internet servers. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[16] P. Dasgupta et al. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1), Winter 1990.

[17] Digitivity Corp. Digitivity CAGE, 1997. http://-www.digitivity.com/overview.html.

[18] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal*, 39(1):194–210, 2000. Reprint Order No. G321–5723.

[19] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *Proceedings of the 42nd IEEE Computer Society International Conference*, San Jose, CA, February 1997.

[20] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, Oct. 1996.

[21] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[22] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151. USENIX Association, Oct. 1996.

[23] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, Oct. 1996. USENIX Association.

[24] M. Franz. Beyond Java: An infrastructure for high-performance mobile code on the World Wide Web. In S. Lobodzinski and I. Tomek, editors, *Proceedings of WebNet '97*, pages 33–38, October 1997.

[25] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, Dec. 1997. USENIX.

[26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[27] J. H. Hartman et al. Joust: A platform for communication-oriented liquid software. Technical Report 97–16, Univ. of Arizona, CS Dept., Dec. 1997.

[28] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of the USENIX 1998 Annual Technical Conf.*, pages 259–270, New Orleans, LA, June 1998.

[29] C. Hawblitzel and T. von Eicken. Tasks and revocation for Java (or, hey! you got your operating system in my language!). Describes Luna, Nov. 1999.

[30] J. Kiniry and D. Zimmerman. Special feature: A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4), July/August 1997.

[31] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of OOPSLA '98*, Vancouver, BC, Oct. 1998. To appear.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Jan. 1997.

[33] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[34] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure execution of Java applets using a remote playground. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, CA, May 1998.

[35] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, Oct. 1996. USENIX Association.

[36] K. Nilsen. Java for real-time. *Real-Time Systems Journal*, 11(2), 1996.

[37] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, Sept. 1995.

[38] 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. *Communications of the ACM*, 23(2):81–92, 1980.

[39] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, Dec. 1989.

[40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Association.

[41] E. Sirer, R. Grimm, A. Gregory, and B. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 202–216, Kiawah Island Resort, SC, Dec. 1999.

[42] Sun Microsystems, Inc. JavaOS: A standalone Java environment, Feb. 1997. http://www.javasoft.com/products/-javaos/javaos.white.html.

[43] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.

[44] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.

[45] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, and D. Hu. J-Kernel. Source code available at http://www.cs.cornell.edu/slk/jk-0.91/doc/Default.html.

[46] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta. Spawn: A distributed computatational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb. 1992.

[47] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Oct. 1997.

[48] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH '98*, San Francisco, CA, April 1998.

[49] T. Wilkinson. Kaffe—a virtual machine to compile and interpret Java bytecodes. http://www.transvirtual.com/-kaffe.html.

[50] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 1992.

[51] N. Wirth and J. Gutknecht. *Project Oberon*. ACM Press, New York, NY, 1992.