ISOLATION, RESOURCE MANAGEMENT AND SHARING IN THE KAFFEOS JAVA RUNTIME SYSTEM

by

Godmar Back

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2002

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Godmar Back

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Wilson C. Hsieh
	John B. Carter
	Elmootazbellah N. Elnozahy
	Jay Lepreau
	Gary Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Godmar Back in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Wilson C. Hsieh
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Single-language runtime systems, in the form of Java virtual machines, are widely deployed platforms for executing untrusted mobile code. These runtimes provide some of the features that operating systems provide: interapplication memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other. Neither do they provide the ability to limit the resource consumption of applications. Consequently, the performance of current systems degrades severely in the presence of malicious or buggy code that exhibits ill-behaved resource usage.

In this dissertation, we show that Java runtime systems can be extended to provide a *process model*, and that such a process model can provide robust and efficient support for untrusted applications. Processes are an operating system abstraction in which each process executes as if it were run in its own virtual machine. We have designed and prototyped KaffeOS, which is a Java runtime system that provides support for processes. KaffeOS isolates processes and manages the physical resources available to them, in particular: CPU and memory. Each process is given its own heap, which can be separately garbage collected. Unlike existing Java virtual machines, KaffeOS can safely terminate processes without adversely affecting the integrity of the system, and it can fully reclaim a terminated process's resources.

The novel aspects of the KaffeOS architecture include the application of a user/kernel boundary as a structuring principle to runtime systems, the employment of garbage collection techniques for resource management and isolation, and a model for direct sharing of objects between untrusted applications. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing of objects.

We built a prototype of our design and ran the SPEC JVM 98 benchmarks to evaluate its performance for well-behaved applications. We found that for those applications, our KaffeOS prototype is no more than 8% slower than the freely available JVM on which it is based, which is an acceptable penalty for the safety that it provides. At the same time, our KaffeOS prototype can support more applications than a hardware-based approach on the same platform. We demonstrate that in the presence of malicious or buggy code that engages in a denial-of-service attack directed against resources, KaffeOS-based systems can contain the attack, remove resources from the attacked applications, and continue to provide robust service to other clients.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, I need to thank my advisor, Wilson Hsieh, for his immeasurable help and support not only in performing the research described in this dissertation but also in finding the motivation and strength to complete it.

I would also like to thank my committee members for their comments on drafts of this dissertation and their help on papers we published earlier. Jay Lepreau deserves credit for giving me the room and support to finish this dissertation.

Tim Wilkinson built the Kaffe system that I used as a foundation for the KaffeOS system described in this work. Patrick Tullmann contributed to many discussions about Java operating systems; he also proofread several chapters of this dissertation. Jason Baker contributed to KaffeOS's implementation; he also proofread Chapter 4. Tim Stack helped the implementation by updating it with a more recent version of Kaffe. I thank Eric Eide for his help with the much-hated LaTeX type setting system. Thanks go to the other members of the Flux research group from whom I have learned a great deal during my time as a graduate student in Utah and to the members of the operating systems research community who helped me by engaging in discussions at meetings and conferences and reviewing earlier papers.

Finally, last but not least, I would like to thank my fianceé Annette for her support and patience.

CHAPTER 1

INTRODUCTION

The need to support the safe execution of untrusted programs in runtime systems for type-safe languages has become clear. Language runtimes are being used in many environments to execute untrusted code that may violate a system's safety or security. Current runtime systems implement *memory protection* in software through the enforcement of type safety [9]. In addition, these systems provide *secure system services* through a number of mechanisms, including language-based access modifiers and a security model for granting access privileges to system services. They do not, however, sufficiently isolate untrusted code and are unable to control the computational resources used by such code. This dissertation presents KaffeOS, a design for a Java virtual machine (JVM) that allows for the robust execution of untrusted code. Like a hardware-based operating system that supports and manages multiple processes running on one physical machine, KaffeOS provides resource control and isolation to its processes within one virtual machine. KaffeOS provides these features without compromising the memory safety and access control existing systems already provide.

1.1 Motivation

To understand why memory safety and access control are insufficient to provide a robust environment for untrusted code, one can consider the following application scenarios that employ type-safe languages:

- With the advent of the World Wide Web, applets written in Java have become popular. An applet is a piece of mobile code that is referenced from within a web page. When a user visits that page, the code is downloaded to the client's machine, where it is executed using the client's computing resources.

There is no prior trust relationship between the originator of the code—who could be a malicious attacker—and the client who executes the code. A Java program's highly structured bytecode representation can be verified to ensure that the code will not compromise security on the client's machine, but it includes no defenses against denial-of-service attacks directed against computing resources such as memory and CPU. Six years after Java was first released to the public in 1995, industrial browsers still do not withstand even the simplest of these attacks [58]. As a result, browsers stop responding to user requests and, in some cases, even crash.

- Java has also become popular for many server-side applications. For instance, servlets are small programs that provide dynamic content to users, such as Java Server Pages [7]. Often, a server hosts many web sites from multiple different content providers (virtual hosts). Even though a servlet's code is usually trusted, a buggy servlet in one virtual host could cause the servlet engine to spend all its time collecting garbage and deny service to other servlets that serve a completely different web site and a completely different set of users.

- Active network technology [77] has emerged recently as an area that applies type-safe languages for networking protocol research and development. Whereas traditional networks only support a small, fixed set of networking protocols, support for active packets allows networks to be upgraded on the fly. Networks can so adapt to new requirements posed by new applications. These new protocols rely on the distribution and execution of customized code in routers along a packet's path. If this code is written in a type-safe language such as Java, Java's memory safety and its security model can be leveraged. However, if the resources used by the code cannot be controlled or if failing code cannot be contained, then it will be extremely unlikely that network providers will deploy this new technology.

- Oracle's JServer environment [55] uses Java to support such popular server-side technologies as Enterprise Beans (a Java-based component architecture geared towards business applications), CORBA servers (an object-oriented infrastructure for client-server communication), and stored procedures in databases. Since different sessions are able to share common data directly, systems like the JServer are highly scalable and support thousands of simultaneous sessions. However, they do not limit the resources a session can use, and they do not fully isolate sessions. Isolating sessions is useful even when the code running in them comes from trusted sources, because the code may be buggy or insufficiently tested. Any system that addresses these problems and provides resource control and isolation must allow for such direct sharing to maintain scalability and efficient use of resources.

- A final example is the use of kernel extensions in extensible operating systems. For instance, in the SPIN operating system [10], applications are allowed to download extensions into the kernel to adapt the system to application-specific needs. Because both the kernel and the extensions are written in Modula-3, a type-safe language, these extensions' access to kernel interfaces can be controlled. However, it is impossible to control the resources used by a given extension—for instance, to guarantee that one extension obtains a certain share of CPU time.

To address these problems, all of these applications require a runtime system that supports the following features:

- *Protection*: Protection includes confidentiality and integrity. Confidentiality requires that an application must not be able to read another application's data unless permitted. Integrity requires that an application must not be able to manipulate the data of another application or system-level data in an uncontrolled manner, or destroy the data of another application.

- *Isolation*: Applications must be isolated from each other. One application's failure must not adversely affect other, unrelated applications or the system.

- *Resource management*: First, resources allocated to an application must be separable from those allocated to other applications to ensure proper accounting. Second, resource management policies must guarantee that an unprivileged or untrusted application is not able to starve other applications by denying them resources.

- *Communication*: Since the system may include multiple cooperating applications, applications should be able to communicate with each other. For Java, an efficient way of sharing data should be supported that does not compromise protection and isolation.

Existing mechanisms such as type safety, language-based access control, and permission checks provide protection—our research shows how to support the remaining three features. Others have developed a number of systems that provide support for multiple, untrusted applications in Java during the last few years. However, they all fall short of our requirements in one or more ways.

An *applet context* [16] is an example of an application-specific approach that supports multiple applications on top of a single Java virtual machine (JVM). It is implemented as a layer on top of the JVM and provides a separate namespace and a separate set of execution permissions for untrusted applets. Applet contexts do not support resource management and thus cannot defend against denial-of-service attacks. In addition, they are not general-purpose: applet contexts are specific to applets and cannot be used easily in other environments. Similar *ad hoc layers* have been created for other application areas, such as servlets and mobile agents.

Ad hoc layers are unable to safely terminate the applications running in them; attempts to terminate applications can cause corruption in other applications or in the system's critical components. Because of this potential for corruption, recent Java systems do not support any way to forcefully terminate applications [75].

The general-purpose models for isolating applications in Java that do exist, such as the J-Kernel [42] or Echidna [40], also fall short. They superimpose an operating system kernel abstraction on Java without changing the underlying virtual machine.

As a result, it is impossible in those systems to account for the often significant resources spent by the JVM on behalf of a given application. For example, CPU time spent while garbage collecting a process's heap is not accounted for. Figure 1.1 depicts the basic structure of both ad hoc layers and superimposed systems.

An alternative approach to separating different applications is to give each one its own virtual machine and run each virtual machine in a different process on an underlying OS [47, 56], as shown in Figure 1.2. Most operating systems can limit a process's heap size or CPU consumption. Such mechanisms could be used to directly limit an entire VM's resource consumption, but they depend on underlying operating system support. Depending on the operating system has multiple drawbacks: the per-JVM overhead is typically high, and the flexibility with which resources can be managed may be limited. For instance, a typical JVM's memory footprint is on the order of 1-2 MB, which can severely restrict scalability on current machines. A JVM's startup costs, which include the cost of loading and linking the Java bytecode, are typically high. When different instances of a JVM run on the same machine, they typically do not share any runtime data structures, even on systems that provide support for shared memory. Finally, the option of dedicating one JVM process to each application does not apply to embedded or portable devices that may not provide OS or hardware support for managing

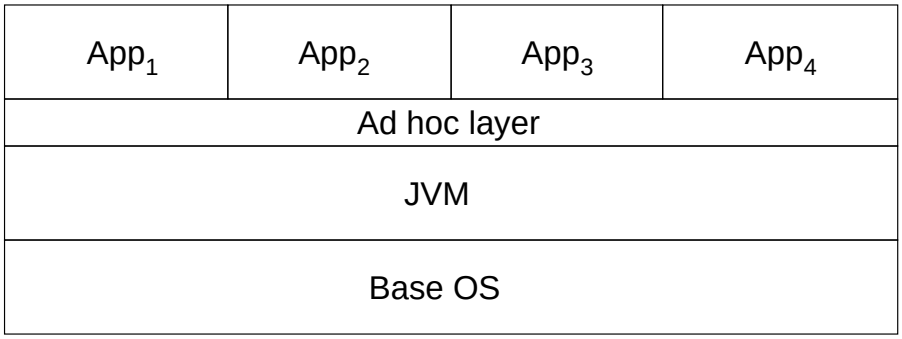


Figure 1.1. Single JVM model. In the single JVM model, applications run on top of an ad hoc layer in one JVM.

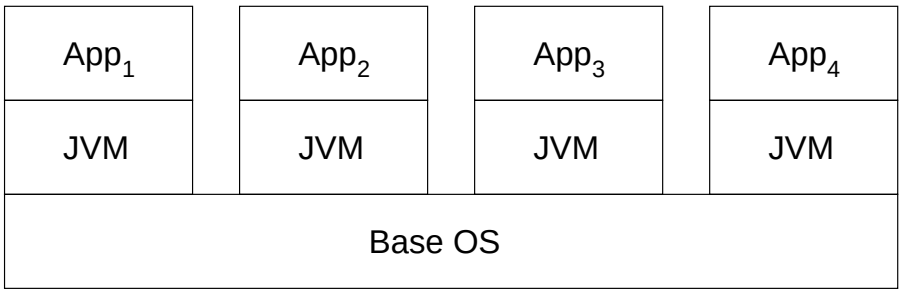


Figure 1.2. Multiple JVM model. In the multiple JVM model, each application runs in its own JVM as a separate process on top of the underlying base operating system.

processes [88].

Although we cannot directly rely on traditional operating systems to separate different Java applications in their respective JVMs, our approach to providing isolation and resource management exploits the same basic principles upon which operating systems are built. On a traditional operating system, untrusted code can be executed in its own process; CPU and memory limits can be placed on the process; and the process can be killed if it is uncooperative. In such a process model, a process is the basic unit of resource ownership and control; it provides isolation between applications. A "classical" property of a process is that each process is given the illusion of having the whole (physical or virtual) machine to itself. Therefore, if runtime systems support such a *process* abstraction, they can provide robust environments for untrusted code.

Compared to the alternative of using separate operating system processes to provide isolation to each Java application, designing runtime systems to support processes is a superior approach. First, it reduces per-application overhead. For example, applications can share runtime code and data structures in much the same way that an OS allows applications to share libraries.

Second, communication between processes can be more efficient in one VM. Processes can share data directly through a direct memory reference to a shared

object. Direct sharing is more efficient than exchanging data by copying it or sharing it indirectly through intermediate proxy objects. One of the reasons for using type-safe language technology in systems such as SPIN [10] was to reduce the cost of interprocess communication; this goal is one we want to keep. Therefore, supporting direct sharing of objects for IPC purposes between multiple processes in one VM is important.

Third, a JVM that does not rely on underlying operating system support can be used in environments where such support is missing. For example, such a JVM could be used on a portable or embedded device that may only have a minimal operating system, especially one that is not powerful enough to fully isolate applications. In other work, we have shown that a single JVM uses less energy than multiple JVMs on portable devices [35]. Finally, because an OS's protection mechanisms are typically process-oriented, it is also difficult (or impossible) to rely on OS support when embedding a JVM in an existing application, such as a web server or web browser.

1.2 KaffeOS

The structure of our system, KaffeOS, is depicted in Figure 1.3. By adding a process model to Java, we have constructed a JVM that can run multiple untrusted

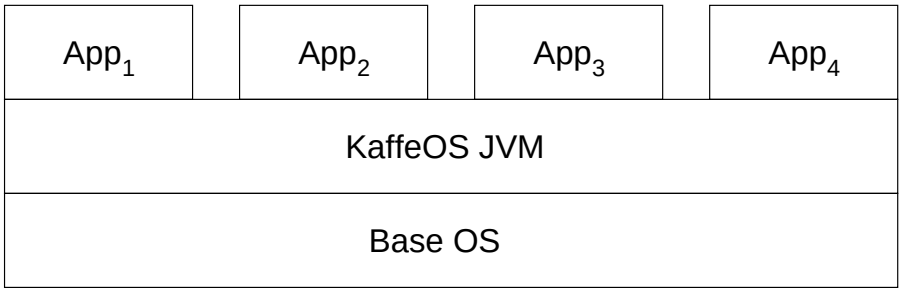


Figure 1.3. KaffeOS model. By supporting multiple processes within the JVM itself, KaffeOS achieves both the efficiency and ease of sharing of the single JVM model and the isolation provided by the multiple JVM model.

programs safely and still supports the direct sharing of resources between programs. A KaffeOS process is a general-purpose mechanism that can easily be used in multiple application domains. For instance, KaffeOS could be used in a browser to support multiple applets, within a server to support multiple servlets, or even to provide a standalone "Java OS" on bare hardware. We have structured our abstractions and APIs so that they are broadly applicable, much as the OS process abstraction is.

KaffeOS can terminate processes safely if they misbehave, fail, or exceed their resource limits. KaffeOS protects components that are essential to the integrity of the system by including them in a trusted kernel within the JVM. Safe termination is achieved by separating processes from the kernel and by structuring the kernel such that it is protected from corruption due to abrupt termination.

KaffeOS uses three different approaches for direct sharing of objects between processes. These approaches represent different points in the spectrum spanned by the conflicting goals of process isolation and resource management versus direct sharing. KaffeOS does not support the casual sharing of arbitrary objects between untrusted parties, because doing so would compromise isolation. However, untrusted parties can share dedicated objects in specially created shared heaps. Shared heaps are subject to a restricted programming model so as not to compromise isolation. Finally, the sharing of kernel objects, which provide shared services, is possible without restrictions.

Our design makes KaffeOS's isolation and resource control mechanisms comprehensive. We focus on the management of CPU time and memory, although other resources such as network bandwidth or persistent storage could be added in the future. The runtime system is able to account for and control all of the CPU and memory resources consumed on behalf of any process. We have dealt with these issues by structuring the KaffeOS virtual machine so that it separates the resources used by different processes as much as possible.

We paid particular attention to memory management and garbage collection, which proved to be the issues that posed the largest challenges in designing KaffeOS.

We have devised a scheme in which the allocation and garbage collection activities of different processes are separated, so that the memory consumption a of different process can be separately accounted for and so that the garbage collector does not become a source of priority inversion. This scheme borrows from distributed garbage collection techniques; however, instead of managing objects and collecting garbage across multiple, physically separated machines, we use it to manage objects across multiple processes within one JVM and to separately garbage collect their objects.

To evaluate the feasibility of KaffeOS's design, we have built a prototype and analyzed its behavior in various application scenarios. In the case of trusted code, there is a small performance penalty for using KaffeOS. We show that this penalty is reasonable; the total observed run-time overhead is less than 8%, relative to the freely available JVM on which our prototype is based.

We also examined our prototype's behavior when confronted with denial-of-service attacks by malicious code, if these attacks are directed against memory, CPU time, or the garbage collector. We found that KaffeOS can successfully thwart those attacks. As a consequence, a system based on KaffeOS can provide robust service to well-behaved applications even in situations in which otherwise faster commercial JVMs break down to the point of providing practically no service at all. We also compared KaffeOS to a configuration that is based on the multiple JVM model and found that KaffeOS's performance scales better. In our experiments, our prototype can support more processes while providing the same protection against misbehaved applications as an underlying operating system.

Finally, we evaluated the practicality of our programming model for shared heaps. We found that despite the restrictions our model imposes to maintain isolation, applications can easily and directly share complex data structures in realistic applications.

In summary, we claim that by combining operating system mechanisms with garbage collection techniques, runtime systems for type-safe languages can be structured to implement isolation and resource management. KaffeOS implements a

process model for Java that isolates applications from each other, provides resource management mechanisms for them, and also lets them share resources directly. However, the mechanisms we describe are not specific to Java and should be applicable to other type-safe languages as well.

Therefore, we put forth the thesis that runtime systems for type-safe languages can be extended to employ a process model that provides robust support for untrusted applications. Such a process model can be implemented efficiently using software mechanisms to provide both application isolation and direct sharing of resources.

1.3 Roadmap

In Chapter 2, we review some Java properties that are necessary to comprehend the discussion in the following chapters.

In Chapter 3, we discuss the principles underlying KaffeOS's design, and how they influenced the mechanisms we implemented. In particular, we discuss how KaffeOS reconciles the divergent needs of application isolation and sharing of resources and how it controls the resources used by its processes.

In Chapter 4, we discuss the implementation of our prototype. This discussion includes the structure of the kernel, the algorithm used by the garbage collection subsystem, and the language issues involved in using Java namespace in a multi-process environment.

Chapter 5 discusses the performance of our prototype, which includes both the overhead for trusted code and the performance improvement in the presence of untrusted code. Chapter 6 provides an in-depth discussion and comparison with related work. Chapter 7 suggests some directions for future work and summarizes our conclusions.

CHAPTER 2

THE JAVA PROGRAMMING LANGUAGE

The Java programming language is a concurrent, type-safe, class-based, and object-oriented language. It was born out of a project called Oak led by James Gosling at Sun Microsystems during the early 1990s. Originally intended as a programming language for set-top boxes, Java quickly became a general-purpose language that is used in a variety of applications. We chose Java as a base, because it is representative of a family of programming languages whose goal it is to make programming less error prone and to make programs more robust.

2.1 Language Overview

Java's syntax and procedural style were taken from C/C++. It uses the same control structures, same expressions, and similar primitive types and operators. However, Java is stricter in defining the behavior of types and operators. For example, unlike C, Java specifies the signedness and range of every primitive data type to ensure that a Java program has the same meaning on different platforms.

Objects: The basic unit of code is a Java class, which can contain fields and methods. As in C++, fields are either per-instance or (global) static fields. Java supports subclassing through single class inheritance. Polymorphism is provided through the use of virtual methods. Unlike C++, Java supports neither nonvirtual, nonprivate methods nor code outside of classes, which forces a more strict object-oriented style of programming. In addition, Java supports subtyping through interfaces. An interface is a collection of method declarations that an implementing class must provide. A class can inherit from only one base class, but it can implement an arbitrary number of interfaces. By disallowing fields in interfaces, Java avoids the difficulties associated with multiple inheritance.

Packages: Java classes are grouped in packages, which are roughly comparable to C++ namespaces. Packages are named in a hierarchical fashion. For instance, all classes that are part of the runtime library are found in the `java.*` hierarchy, such as `java.lang.String` for basic functionality such as Strings or `java.net.*` for the set of classes that provide access to standard network facilities such as sockets.

Type safety: Java is a type-safe language; Java programs cannot arbitrarily access the host computer's memory. All memory is tagged with its type, each of which has a well-defined representation. Java does not use pointers. Instead, objects are accessed using *references*, which cannot be cast into or from any primitive type, such as an integer. Arrays are first-class objects, and all accesses are bounds-checked. These properties avoid the pitfalls associated with pointer arithmetic.

Java's memory management is automatic. The programmer is relieved from the burden of having to explicitly manage memory. Instead, a garbage collector determines which objects the program is able to access in the future; it reclaims those objects that can not be accessed on any legal execution path. Consequently, there are no crashes due to dangling references that point to already freed memory. Note, however, that memory leaks can occur if references to objects remain stored unintentionally. Java allocates all objects on the heap—there are no locally declared static objects as in C++. This design avoids the complications associated with copy constructors and destructors in C++. Instead of destructors, Java supports finalization of objects. Objects can provide a `finalize()` method that is executed once the object has become unreachable but before its memory is reclaimed. `Finalize` methods can be used to free system resources, such as sockets, that may be associated with a Java object.

Language access modifiers: Java supports information hiding using access modifiers. Each class, method, or field can be marked with either *private*, *protected*, *public* or can have no modifying prefix at all. Private asserts that a field or method can be accessed only from within its class. Protected fields and methods are accessible to subclasses, and public fields and methods are accessible from anywhere. In the default case, a field or method can be accessed only from classes within the

same package as its class.

Access modifiers are enforced at runtime. Consequently, access to critical fields can be effectively protected by declaring them private. The flexibility of language access modifiers is limited, because there is no feature similar to C++'s friends that allows a class to grant access to a specific class or method. Therefore, accesses from outside the current package require that the data or method be public, which makes it accessible to all packages. Run-time checks are then required to determine whether the caller has sufficient privileges to perform the requested operation.

Exception handling: Java exceptions are used to signal errors that occur during program execution. Java adopted C++'s `try/catch` model for exceptions. If an exception is raised within a `try` block, execution branches abruptly from the current instruction to the corresponding `catch` clause for that exception. If there is no matching clause, the method terminates abruptly, and the exception is thrown in the caller. In this way, exceptions propagate up the call stack until either a matching catch clause is found or there are no more stack frames. Unhandled exceptions terminate the current thread.

An exception can be raised in different ways: either directly by executing a `throw` statement or indirectly as a side effect of executing a bytecode instruction or runtime operation. For instance, any array store instruction can throw an exception if the array index is out of bounds. Similarly, attempts to dereference a `null` pointer result in a null pointer exception. Some runtime operations can be interrupted on request; such interruption is also signaled via an exception.

There are two groups of exceptions: checked exceptions, which an application is expected to handle if they occur, and unchecked exceptions, from which an application is not expected to recover. Every method must declare the checked exceptions it can throw. The compiler checks that any caller of a method that can throw an exception is prepared to handle it—either by supplying a compatible catch clause or by declaring the exception itself. The rationale behind this rule is to avoid a common pitfall in languages such as C, where programmers tend to forget to check return codes for error conditions.

Unchecked exceptions need not be declared. Java's designers judged that having to declare such exceptions would not significantly aid in establishing the correctness of the code and would pointlessly clutter code ([49], §11.2). In addition, it would be difficult or impossible for a program to recover from them. An example of such an exception that is hard to recover from would be a loading error that occurs if parts of the program code cannot be loaded. Note that code in critical parts of the runtime system must handle all errors, including those errors that signal resource exhaustion, such as out-of-memory errors. Consequently, such code may not be able to take significant advantage of Java's exception checking rules.

Concurrency: Java is a concurrent language with built-in support for multi-threading. Programs can create new threads simply by instantiating an object of type `java.lang.Thread`. Threads have their own stacks for their local variables, and share data through objects on a common heap. Two kinds of synchronization are supported: mutual exclusion and unilateral synchronization. Mutual exclusion is provided through the `synchronized` keyword, which encloses blocks of code in a critical section that can be entered only by one thread. In addition, methods can be declared synchronized, which is equivalent to enclosing the entire method body in a synchronized block. Java's concurrency mechanisms do not prevent unsynchronized accesses to shared variables [41]; consequently, race conditions can occur. Unilateral synchronization is provided in a POSIX-like wait/notify style [46]; every object inherits a `wait()` and a `notify()` method, and therefore every object can be used as a condition variable.

Every object to which a program holds a reference can be used as a lock when it occurs as an argument to a `synchronized` clause. This feature can, however, cause problems when public fields of critical objects are used for synchronization. For instance, in an early version of the HotJava browser, an applet could effectively halt all activity in the browser by acquiring and not releasing a lock on its status bar object. Other applets and the system would get stuck as soon as they tried to display any messages in the browser's status bar.

Terminating uncooperative threads is not safe in current versions of Java. To

understand this problem, it is useful to look at the history of Java threads [75]. The original Java specification provided a method called `Thread.stop()` to stop threads. This method caused an exception to be thrown in the targeted thread's context. This asynchronous exception was handled like any other run-time error, in that locks were released while unwinding the stack. Later, JavaSoft realized that this procedure could lead to damaged data structures when the target thread holds locks. In particular, data structures vital to the integrity of the runtime system could be left in inconsistent states. As a result, JavaSoft deprecated `Thread.stop()`.

A later proposal for termination was also flawed. A different mechanism for termination, `Thread.destroy()`, was proposed by Sun that would have terminated a thread without releasing any locks it held. This proposal had the potential for deadlock, and JavaSoft never implemented it. There is currently no supported mechanism to ensure atomicity in the presence of asynchronous termination.

Native libraries: In many circumstances, it is necessary to interface Java code with code in other languages, usually C or assembly code. To access such code, Java allows methods to be declared `native`, i.e., as having an implementation in non-Java code. Since C code cannot be shown to maintain Java's safety properties, only trusted classes can have native code, although some systems have tried to avoid that restriction by placing the code in a different operating system process [24]. However, such approaches require each native call to use interprocess communication mechanisms.

Writing native code requires close cooperation with the JVM's garbage collector. The code must ensure that objects that are kept alive by native code are found by the collector. Two approaches are possible: a conservative collector can scan the data segment of the entire program and keep all objects that could be referenced alive, or the code is required to explicitly register and unregister references it needs to be kept alive. The latter approach is chosen in the Java native interface (JNI) specified by Sun Microsystems [53]. Native code is allowed access to Java objects only through a well-defined interface, which includes such operations as getting or setting fields in an object. This interface allows the JVM to closely monitor all

operations done by native code that are relevant to the garbage collector.

2.2 The Java Virtual Machine

A Java compiler translates Java programs into bytecode that is targeted at the Java virtual machine. Java bytecode provides the basis for Java's portability, because the same bytecode can be executed on all platforms that provide an implementation of the Java virtual machine.

Java uses late binding—Java bytecode is loaded and linked at run time. Code can be loaded from a filesystem or from user-specified sources such as web servers. After the bytecode is loaded, it is checked by a verifier before being linked. If the verification is successful, the bytecode is executed. Java bytecode can be executed by an interpreter, or it can be translated into native code by a just-in-time compiler (JIT) first, as illustrated in Figure 2.1. Most JVMs today employ JIT compilers; interpreters are typically used only where the space and complexity overhead of JIT compilation is not tolerable, such as in small or embedded systems. To avoid the overhead of JIT compilation for code that is executed only rarely, some VMs use an adaptive technique that interprets code first and translates those pieces that are invoked frequently during the execution of the program.

The Java virtual machine is a simple, stack-based virtual architecture that provides storage for the heap and each thread's stack and registers, and a CPU to execute bytecode instructions. Local registers, object fields, and static fields can be pushed on and popped off the stack. All arithmetic and logical operations are performed on stack elements; their results are also stored on the stack.

Bytecode verification: Bytecode verification ensures that the bytecode maintains the same safety properties that are present in the source language. Java bytecode is designed such that the Java runtime can verify its consistency and therefore need not trust the bytecode compiler to produce proper code. This property allows the use of Java bytecode in mobile code environments. A client can check safety properties of the code without access to the source code.

Bytecode instructions are strongly typed, so that the bytecode verifier is able to verify whether the type of the operands matches the expected type. For instance,

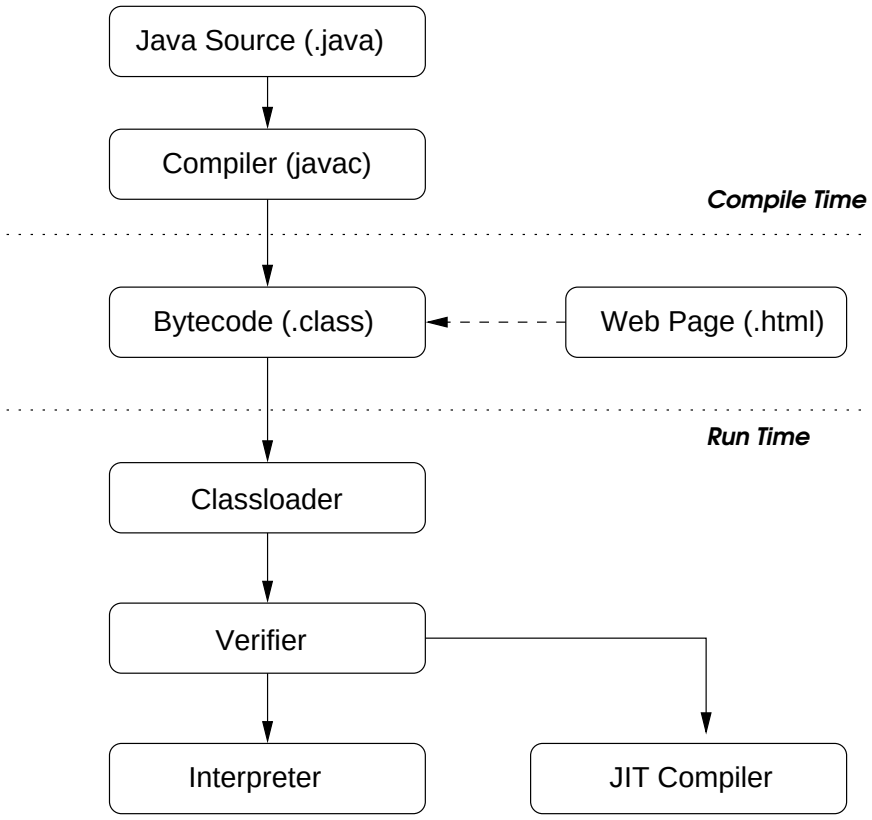


Figure 2.1. Java compilation and execution. In a typical scenario, Java source code, such as applet code, is first compiled into an intermediate representation called bytecode. This bytecode is embedded in webpages. A class loader that is part of the client's browser loads the bytecode, passes it on to the verifier for verification. The code is either interpreted or compiled to native code before being executed.

the bytecode instruction to add two doubles (`DADD`) is different from the instruction to add two ints (`IADD`). If an instruction requires a given type, the virtual machine has to make sure that a given local register or stack location contains a value of that type. The bytecode verifier ensures that this property holds no matter which execution path is taken to reach a given instruction. The verifier proves this property using simple dataflow analysis. Figure 2.2 provides a simple example.

Class loading: Java's class loading mechanism provides four major features: lazy loading, multiple namespaces, user-defined loading policies, and type-safe

```
class A {
    static A a;
    B b;
    void f(B x) {
        a.b = x;
    }
}
```

```
Method void f(B)
   0 getstatic #5 <Field A.a A>
   3 aload_1
   4 putfield #6 <Field A.b B>
   7 return
```

Figure 2.2. Example of Java bytecode. The right side contains a disassembled listing of bytecode produced by the Java compiler for the `A.f` method shown on the left. The `getstatic` instruction loads the static field `A.a` onto the stack. The method's argument, `x`, is kept in local register #1; the `aload_1` instruction pushes `x` on the stack. `putfield` operates on the stack operands and assigns the value of `x` to the field labeled `b` in `A.a`. Simple dataflow analysis tells the virtual machine that the value stored in `A.a` is indeed of type `B`.

linkage [54]. Lazy loading postpones class loading and linking until a class's first use; users can define class loaders to provide their own loading policies and define multiple namespaces. The JVM ensures that the linking process cannot be used to bypass Java's type-safety guarantees.

A *class loader* is an object that serves requests for class objects. A class object[1] is a runtime representation of a type. Each class object is tagged with the loader that loaded it. The VM maintains a mapping

$$\textit{(class loader, class name)} \rightarrow \textit{class object}$$

All class objects that have a loader in common form a namespace, for which the common loader acts as name server. Class objects with the same name that are loaded by different class loaders constitute different types: attempts to cast one to the other fail.

Multiple namespaces can be created by instantiating multiple class loaders. Supporting multiple name spaces is necessary for applications such as running

[1]Note that the term *class* is typically used for both the runtime class object and the syntactic `class` construct at the source code level.

applets from multiple sources in a browser, because there is no prior coordination between the different developers as to how to name their classes.

Class loaders *define* class objects by presenting a sequence of bytes in class file format to the JVM. The JVM parses and verifies the contained bytecode, establishes a new runtime type, and returns a reference to the new class object. The loader that defines a class is said to be that class's defining loader. Class files contain both a class's bytecode and symbolic information necessary to resolve interclass references.

If a class contains symbolic references to another class, the runtime system initiates the loading process of the referred class by requesting a class object from the referring class's defining loader. The initiating loader can either load the necessary class file itself, or it can in turn invoke another class loader to resolve the reference. This process is known as *delegation* [54]. Since the Java type is determined by the defining loader, delegation allows different class loaders to share types by delegating them to a common loader. One example is the system class loader, to which all loaders refer for system classes.

Because a class loader is a user-defined object, it has full control over the location from where class files are loaded. Ordinarily, class files are loaded from a local file system. A class loader can implement other options: for instance, a loader in a mobile agent system could load class files over the network from an agent's originating host. Some systems even create class files on the fly from other source languages, such as Scheme [15]. Class loaders can also restrict their classes' access to other classes by refusing to return an answer for a requested name. This mechanism can be used to enforce a crude form of access control.

Type-safe linkage: The virtual machine must guarantee type safety, even in the presence of class loaders that do not return consistent answers when asked for names. For instance, a class loader should return the same runtime class when queried for the same name. The JVM must be able to cope with user-defined loaders that violate this invariant. An insufficient understanding of the consistency assumptions made by the JVM compromised type safety in early JVM implemen-

tations. Ill-behaved class loaders were able to bypass language access modifiers or could even cause the JVM to crash. These vulnerabilities were fixed by defining and enforcing a set of constraint rules on the classes a loader defines [54]. For instance, one rule states that if a class references a field declared in a class that was defined by a different loader, then that field's type must have the same definition in both loaders' namespaces. If a loader attempts to define a class that would violate such constraints, a linkage error is raised.

Security model: Java's security mechanisms defend against threats that can arise from untrusted code, such as the possibility of system modification or corruption or leakage of sensitive information that could violate a user's privacy. Untrusted code is prevented from performing operations such as writing to or deleting local files or directories, or reading from files and sending their contents over the network.

The Java security model associates principals with code and controls access to resources such as files and sockets. Code that originates from the same source is considered to belong to the same principal and is given equal privileges.

Java's designers have tried to separate security mechanisms and policies by centralizing policy decisions in one location in the runtime system. Early versions of Java used a `SecurityManager` object for that purpose. Runtime classes consulted the security manager whenever an access control decision was to be made. If the attempted operation was disallowed, the security manager vetoed it by raising a `SecurityException`. The first policy implemented using security managers was the *sandbox* policy. The sandbox policy is an all-or-nothing approach: code from untrusted sources is sandboxed and has no privileges, and trusted code has all privileges. The security manager bases its decision on whether to allow a sensitive operation to proceed on the call stack of the current thread: if any activation record belongs to untrusted code, the operation is disallowed.

Because this approach was too inflexible, later versions of Java replaced it with an approach that does not require a security manager but supports making access control decisions directly in the VM. Classes are placed in different *protection domains*, and each domain possesses a set of permissions. When an access control deci-

sion must be made, the JVM examines the call stack and determines the protection domains in the current thread's call chain. Unlike in the security manager model, the JVM obtains the effective set of permissions by intersecting the permissions held by the domains that are part of the call stack. This algorithm implements the principle of least privilege: it prevents lower-privileged code from acquiring privileges by calling into higher-privileged code, and it requires higher-privileged code to effectively give up its permissions when calling into lower-privileged code.

CHAPTER 3

DESIGN OF KAFFEOS

Our discussion of the goals and decisions behind KaffeOS's design is split in four sections: first, we describe how we applied lessons from operating systems to protect and isolate processes in KaffeOS and how we guarantee safe termination of its processes. In the second section, we focus on the design of the memory management subsystem and explain how KaffeOS separates garbage collection activities. The third section is dedicated to sharing models: we discuss the trade-off between sharing and isolation, outline the possible design space for sharing models, and present KaffeOS's sharing model. In the final section, we describe KaffeOS's hierarchical resource management model for memory and CPU resources.

3.1 Protection and Isolation

Protection is a core function of any operating system. An operating system needs to protect the integrity of shared resources and control access to them. An operating system must also provide *isolation* for its processes, i.e., it must isolate their activities as much as possible to uphold the illusion that each process has the whole machine to itself. We discuss the different aspects of protection and isolation and show how operating systems provide these features. We then apply this discussion to show how we provide protection and isolation in KaffeOS.

3.1.1 The Red Line in Operating Systems

Many operating systems use a model in which processes can execute in two different modes, user mode and kernel mode, which are represented by a bit in a CPU register. In user mode, a process executes user code. User code may not access any memory or I/O ports outside the current process's address space. It also

cannot execute the privileged instructions needed to change the bit that indicates the current mode. This mechanism protects the kernel and other applications from malicious or buggy applications by preventing them from corrupting kernel or application data. If user code attempts to execute illegal instructions, a trap instruction enters the kernel, which can terminate the offending process safely. When a process needs to access kernel data, the process enters kernel mode by executing a trap instruction, which transfers execution to kernel code. Kernel code is trusted and has full access to all memory, all I/O ports, and all instructions.

Protection is the main function of the user/kernel boundary, which is also often colloquially referred to as the "red line" [20]. However, the red line serves several other functions: it enforces resource control and provides safe termination.

The user/kernel boundary is necessary to enforce resource control. A process running in user mode is subject to policy limitations on the resources that it can consume. The kernel, on the other hand, has access to all of the physical resources of the machine and enforces resource limits on user processes. For example, only the kernel can disable interrupts, which prevents processes other than the current process from running on the process's CPU. In most systems, the kernel is directly responsible for scheduling processes, which controls how many CPU cycles processes are allowed to consume. Similar restrictions apply to the use of memory: a process may use only the memory provided to it by the kernel.

Changes to resource allocations are done by entering the kernel. For example, UNIX processes use the `sbrk` system call to request a change in the amount of memory available to them. Some operating systems (such as exokernel systems [34]) move resource management to user level. Even in such systems, the kernel is responsible for providing the mechanisms for enforcing a given policy.

Kernel code must be written with resource control in mind. For instance, should an attempted kernel operation exceed a user process's resource limit, the kernel must be able to either gracefully reject the operation or terminate that process safely. The kernel must be structured so that a resource control violation does not cause abrupt termination that endangers the integrity of the kernel itself, because

that could affect the integrity of other applications (which would violate isolation). Aside from a resource control violation, a process can also be terminated because of an internal failure condition or because of an explicit request.

No matter why a process is terminated, the integrity of the system must not be harmed. Of course, the system cannot guarantee that processes that depend on the terminated process will continue to function, but it must ensure that unrelated processes are unaffected. Integrity must be preserved by ensuring that globally shared data structures are modified atomically, which is accomplished by enclosing those data structures in the kernel. The system's integrity must be maintained even when processes exceed their resource limits while manipulating kernel data structures and have to be terminated as a consequence.

Most operating systems provide a mechanism to defer explicit termination requests. Usually, this is a software flag that is set when a nonterminable region is entered and cleared when it is left. In traditional operating systems such as Unix, the nonterminable region encompasses the complete kernel. That is, crossing the red line enters code in which termination requests are deferred.

3.1.2 Language-based Protection and the Red Line

Java's type safety and language-based access control provide language-based protection without a red line. KaffeOS employs these mechanisms to prevent a process from accidentally or intentionally accessing another process's data. This property is enforced by several mechanisms, which work together. First, Java's type safety guarantees that programs cannot forge pointers to obtain access to foreign objects. A process cannot access arbitrary memory locations by casting an integer to a pointer. It can access only valid objects to which it holds references.

Second, system code uses Java's access modifiers, such as *private* and *protected*, to prevent processes from accessing fields in system objects to which they have access. (We assume that Java's type system is sound and that Java's bytecode verifier functions properly.)

Third, different processes use distinct classloaders. Each classloader provides a namespace with a distinct set of types; the details of the interaction of KaffeOS's

processes and types are discussed in Section 4.2. If a process were somehow able to get hold of an object in another process, attempts to access it would result in an exception, because the object's runtime class would not be recognized in the rogue process.

Fourth, we trust system code not to hand out references to a foreign process's objects or to internal system objects. Although we cannot guarantee these properties, the damage caused by violating them is limited, because language access modifiers and different namespaces restrict possible use of such leaked objects.

Another reason why the system must not expose certain internal objects is that Java allows any object to be used as a monitor. We must avoid a scenario in which a user process could acquire a lock on an object that is used internally for synchronization, hold on to it indefinitely, and possibly prevent other processes from making progress. In such a scenario, the system would hand out references to an internal thread control object, which system code uses internally to synchronize such thread operations as exiting a thread or joining other threads. If a process could acquire a lock on that object, it could prevent these thread operations from completing and possibly prevent itself or other processes from terminating.

Cases in which system objects that are used for synchronization are exposed are bugs. Experience with our implementation indicates that these bugs can often be detected during testing by the deadlocks they cause. These deadlocks occur if a thread is terminated while locking a system-level object. Since the object's lock is not released, all threads will eventually block on that object, which causes a detectable deadlock.

All of the protective measures mentioned so far can be implemented by employing existing mechanisms in Java. Consequently, in KaffeOS, we do not need a red line for protection. However, the red line is still needed for proper resource control and safe termination.

Proper resource control in Java requires a user/kernel distinction, because certain parts of the Java system must be written either to have access to all of the physical resources, or not to require memory allocation, or to be prepared if a

memory allocation fails. For example, the exception handling code in the Java runtime must not depend on the ability to allocate heap memory, since the inability to acquire memory is signaled via an exception. The developers of Java failed to distinguish those parts of the system that should be subject to policy-defined resource limits from those that should always have access to all of the physical resources. As a result, applications such as servlet engines fail fatally on current JVMs if they run out of resources in code that manipulates shared data structures, especially those structures whose consistency is critical to the functioning of the application. Such failures can be prevented by introducing the red line, which must protect such data structures, and making sure that code inside the kernel does not abruptly terminate when a process exceeds resource limits.

The integrity of critical data structures must also be preserved if a process is terminated by request—for instance, if a user wishes to kill an uncooperative or stuck process. This safety can be provided by ensuring that all manipulations of critical data structures are done in an atomic fashion. The red line provides this atomicity by deferring termination requests while such data structures are manipulated. To protect these data structures and to ensure correctness, Java runtime systems must draw a red line to encapsulate those parts of the Java runtime libraries in which threads must not be terminated. Mechanisms must be present to prevent abrupt termination in this code. By contrast, threads executing in user code can be terminated at will. The usefulness of the red line as a structuring principle does not depend on a specific JVM implementation; it can and should be applied to all JVMs that support multiple applications.

Deferring termination to protect data structure integrity is superior to the possible alternative of supplying cleanup exception handlers. In the alternative model, a termination request would be delivered as an exception, and catch clauses would be added as cleanup handlers. Cleanup handlers in the form of *catch* clauses have a slightly lower cost than deferring termination in the common case, but programming them is tedious and error prone. (See Appendix A for an illustrating example.) In addition, Marlow et al. [57] have pointed out that the use of *catch*

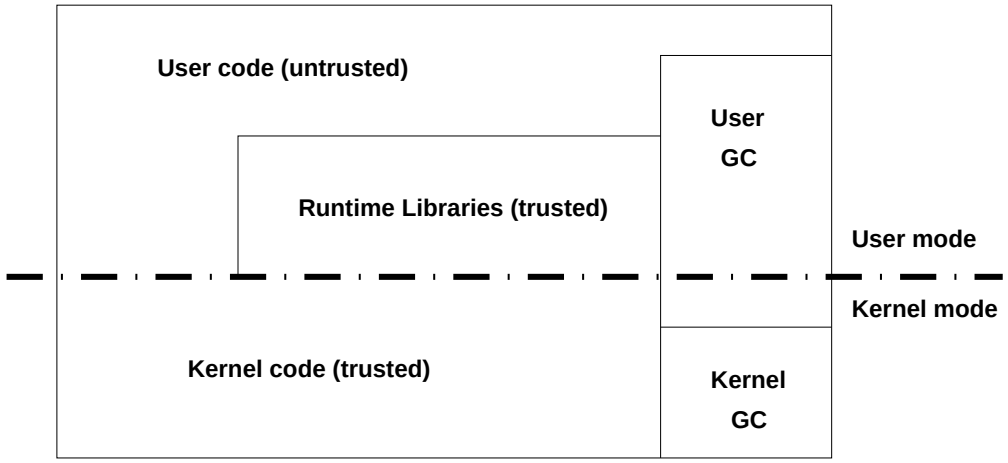
clauses can create race conditions.

Another alternative solution, put forth by the Real-Time Java proposal [14], would be to defer termination requests during synchronized sections. However, this alternative would also be problematic, because deferring termination is independent of synchronization. As discussed in Section 2.1, the *synchronized* keyword is intended to protect a data structure's consistency in the presence of concurrent access, not in the presence of termination. Equating these two concepts leads to decreased parallelism, higher potential for deadlocks, and confusing code, because programmers are then forced to use synchronization on objects that are clearly not shared with any other thread, just to prevent termination. In addition, termination would be unnecessarily deferred even when the synchronization object is not shared.

Defining the red line to protect the integrity of the system does not make asynchronous exceptions a usable programming tool for multithreaded applications, because it does not protect user data structures from corruption that is caused by abrupt termination. If untrusted user code were allowed to enter kernel mode, it could delay termination indefinitely. Hence, such applications must still rely on cooperation to know when it is safe to stop their own threads; they must ensure that their data structures are kept consistent when threads are stopped.

3.1.3 The User/Kernel Boundary in KaffeOS

Figure 3.1 illustrates the high-level structure of KaffeOS. User code executes in "user mode," as do some of the trusted runtime libraries and some of the garbage collection code. The remaining parts of the system must run in kernel mode to ensure their integrity. These parts include the rest of the runtime libraries and certain parts of the virtual machine, such as the garbage collector or the just-in-time compiler. A piece of code executes in kernel mode if and only if it accesses globally shared data structures. Such a structure echoes that of exokernel systems [34], where system-level code executes as a user-mode library. A language-based system allows the kernel to trust user-mode code to a great extent, because bytecode verification ensures that language access modifiers such as "private" or "protected" are respected at run time. Often, kernel mode is entered at the beginning and



	User Mode	Kernel Mode
Trusted	Runtime Code	Kernel Code
Untrusted	User Code	

Figure 3.1. User/kernel boundary in KaffeOS. System code, which includes the kernel, the runtime libraries and the garbage collector, can run either in kernel or user mode; such code enters kernel mode where necessary. By contrast, user code always runs in user mode. In user mode, code can be terminated arbitrarily; in kernel mode, code cannot be terminated arbitrarily. The bottom part of the figure shows that the trust dimension is orthogonal to the user/kernel mode dimension.

left at the end of a public method; methods called from that method assume (and `assert()` where appropriate) that they are called in kernel mode. We present concrete examples in Section 4.1.

Unlike in a hardware-based OS, *user mode* and *kernel mode* in KaffeOS do not indicate different protection domains. In particular, KaffeOS's use of the red line does not prevent user objects from directly accessing shared objects. Instead, user mode and kernel mode in KaffeOS indicate different environments with respect to termination and resource consumption:

- Resources consumed in user mode are always charged to a user process and not to the system as a whole. Only in kernel mode can a process consume resources that are charged to the kernel, although typically such use is charged to the appropriate user process.

- Processes running in user mode can be terminated at any time. Processes running in kernel mode cannot be terminated at an arbitrary time, because they must leave the kernel in a clean state.

The KaffeOS kernel is structured so that it can handle termination requests and exceptional situations that are caused by resource exhaustion cleanly. Explicitly issued termination requests are deferred, but kernel code must also not abruptly terminate due to exceptions. We achieve this goal by adopting a programming style that mostly avoids exceptions and uses explicit return code checking instead. This style has the advantage of being equally applicable to both code written in Java and in languages that do not support exceptions, such as C. In addition, it reduces the requirement that the exception handling facilities in the runtime system themselves do not terminate abruptly. Other than through testing, we do not currently have a means to verify that our kernel implementation follows these principles. We discuss the possible use of static analysis as future work in Section 7.1.1.

In some situations, kernel code has to call out to user code. When making such upcalls, kernel code must be prepared for the user code to take an unbounded amount of time, or not to return at all, or to complete abruptly. The thread executing the upcall cannot hold on to system-level resources that may be needed by other processes prior to making the call. Other than through visual inspection, we do not currently have a means to ensure that. For some resources, we could check this property at run time, but it would be hard to do in general unless we kept track of every lock a thread holds. This bookkeeping would incur an overhead that would nullify the performance benefits of fast locking algorithms, such as thin locks [2].

3.2 Memory Management in KaffeOS

Each process is given its own heap on which to allocate its objects. In addition, there is a dedicated *kernel heap* on which only kernel code can allocate objects. A heap consists of a memory pool, i.e., a set of memory pages, that is managed by an allocator and a garbage collector. Processes can allocate memory from their heaps without having an impact on other processes, and, as importantly, they can garbage collect their heaps separately. Separated heaps have been used in other contexts before. For instance, multiprocessor memory allocators such as Hoard [6] use per-CPU heaps to reduce lock contention and increase concurrency. By contrast, we use them for resource separation and accounting.

Our goal is to precisely and completely account for the memory used by or on behalf of a process. Therefore, we account not only for objects at the Java level but for all allocations done in the VM on behalf of a given process. For instance, the VM allocates data structures during the loading and linking process for a class's symbols and range tables for exception handling. On the other hand, bytecode-rewriting approaches, such as JRes [22, 25], can account only for object allocations, because they do not modify the virtual machine.

Kernel interfaces are coded carefully such that memory that is used on behalf of a process is allocated on that process's heap. When one considers, for example, the creation of a new process with a new heap; the process object itself, which is several hundred bytes, is allocated on the new heap. The handle that is returned to the creating process to control the new process is allocated on the creating process's heap. The kernel heap itself contains only a small entry in a process table.

As a matter of design, we try to map all logical resources to memory. This design has the advantage that we need not apply resource limits to logical resources, but only to memory. For example, a hardware-based operating system needs to apply limits on the number of files a process can have open at any given point in time. This limit is necessary because each file descriptor occupies kernel space and is therefore not subject to its process's heap limits. By contrast, KaffeOS does not need extra limits for system objects such as open files, because it can allocate them

completely on user heaps.

Although we would like to map all logical resources to objects that can be reclaimed via garbage collection, some resources require explicit deallocation (e.g., Unix network sockets must be closed using the `close(2)` system call to signal the end of the connection to the other party). In KaffeOS, such resources are always associated with trusted system objects. Those objects must register reclamation handlers with the kernel, which are executed upon a process's exit.

3.2.1 Full Reclamation

As in any operating system, all memory used by a process must be reclaimed when a process exits or is terminated. Memory must be reclaimed without sacrificing type safety, which implies that dangling pointers must not be created. Therefore, no foreign references must point to objects on the terminated process's heap, so that the garbage collector can fully recover a terminated process's objects. Such situations could potentially occur for two reasons: either because an object in a foreign process inadvertently acquired a reference or because two processes cooperated maliciously. The former case could occur if, for instance, we were to lazily allocate a runtime class object on a user heap and then attempted to refer to that class object later from a different process. In the malicious case, a process would pass a reference to one of its own objects to another process that outlives the process passing the reference. We refer to the latter case as a *sharing attack*.

In Java's execution model, an object can be referenced in one of three ways: either from another object on the heap, from a machine register, or from a thread's stack. Consequently, for a process's objects to be fully reclaimable, they must not be referenced from either another heap or a foreign thread's stack or registers. We discuss heaps in this section; an exhaustive discussion of how we treat references from thread stacks and registers is presented in Section 3.2.2.

To prevent cross-references from foreign heaps, we monitor writes to the heap. We use *write barriers* [87] for that purpose. A write barrier is a check that happens on every write of an object reference into the heap. As we show in Section 5.1, the cost of using write barriers, although non-negligible, is reasonable. Unlike for

software-fault isolation schemes [83], we do not need to instrument every "store" machine instruction—in Java, write barriers need to be inserted only for certain Java bytecode instructions. These instructions are `PUTFIELD`, `PUTSTATIC`, `AASTORE`, and any assignment within the VM or in native libraries that creates a connection between two garbage-collected objects. `PUTFIELD` and `PUTSTATIC` need only be instrumented if the type of the destination field is a reference to an object and not a primitive type such as a double or integer. This information is known at run time, because Java bytecode retains all type information.

KaffeOS's write barriers prevent illegal cross-references by interrupting those writes that would create them and raising an exception instead. We call such exceptions "segmentation violations." Although it may seem surprising that a type-safe language runtime could throw such a fault, it actually follows the analogy to traditional operating systems closely. However, whereas those systems guard against writes into another segment, we reject writes that would connect to other segments. As with all exceptions, kernel code must be able to gracefully back out of segmentation violations.

Not all cross-references between heaps are illegal: only cross-references between user heaps are, because such references would prevent full reclamation. Some cross-references between user and kernel heaps are required to be able to precisely account for the memory used by a process as discussed earlier and to allow for the provision of system services by the kernel. We discuss how we reclaim those references in Section 3.2.2. Figure 3.2 illustrates which cross-references are legal and which are not.

An alternative to preventing the creation of illegal cross-references would have been to revoke existing references after a process terminates. Such revocation could be implemented in hardware or in software. A hardware implementation would invalidate a reference by unmapping the memory at which the object is stored. Since we want our design to apply to a wide range of systems, we assume that the platforms on which it runs will not necessarily have a hardware memory management unit under its control. We also assume that the host may not have

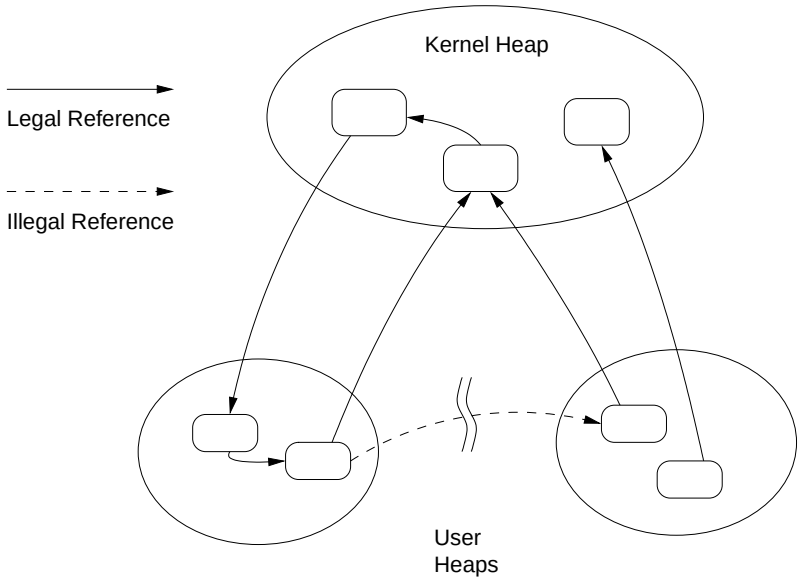


Figure 3.2. Valid references for user and kernel heaps. References between user and kernel heap are legal in both directions. Cycles involving a single user heap and the kernel heap may occur, as shown on the left. Multiple processes can refer to the same object on the kernel heap, thereby sharing it. References that would span across user heaps are illegal.

an operating system that supports virtual memory. A Palm Pilot is an example of such a host. Under these assumptions, memory cannot simply be revoked by unmapping it.

A software approach to revocation would involve resetting all locations that hold references to a shared object. For instance, a special value—analogous to the `null` value used to represent an empty reference—could be assigned to these locations. Future accesses through such a reference would trigger an access violation. To allow for revocation in this way, all references to the object would have to be located.

In the presence of C code, such a search is impossible to do, because any word in memory is a potential reference. Even a limited implementation that is restricted to Java references would require extensive compiler support. This problem is analogous to a problem that occurs in persistent object systems that use the garbage collector to pickle and unpickle objects [30]. In these systems,

when an object is written to persistent storage, all references to it have to be updated. Although feasible, this scheme requires extensive compiler support to keep track of references, which is expensive in both storage space and complexity of implementation.

3.2.2 Separate GC

Each process's garbage collection activity is separated. Each process must be able to garbage collect its own heap independently of others. If there were no cross-references at all between heaps, separate GC would be trivial. Since cross-references are required to share objects on the kernel heap, we use techniques from distributed garbage collection schemes [61] to take cross-references into account. Distributed GC mechanisms are normally used to overcome the physical separation of machines and create the impression of a global shared heap. In contrast, we use distributed GC mechanisms to manage multiple heaps in a single address space.

3.2.2.1 Use of Entry/Exit Items

We use write barriers not only to prevent illegal cross-references but also to detect and monitor legal cross-references. When a cross-reference is created, we create an *entry item* in the heap to which it points. During a garbage collection cycle, entry items are treated as garbage collection roots. We share the entry item in the case that multiple cross-references to the same object exist, as shown in Figure 3.3. In addition, we create a special *exit item* in the original heap to remember the entry item created in the destination heap. Unlike distributed object systems such as Emerald [50], entry and exit items are not used for naming nonlocal objects. Instead, they are used to decouple the garbage collection of different heaps.

Exit items are subject to garbage collection: if the garbage collector encounters a reference to an object outside the current heap, it will not traverse that object, but the corresponding exit item instead. Exit items that are not marked at the end of a GC cycle are garbage collected, and the reference count of the entry item to which it points is decremented. Entry items are reference counted: they keep track of the number of exit items that point to them. If an entry item's reference count

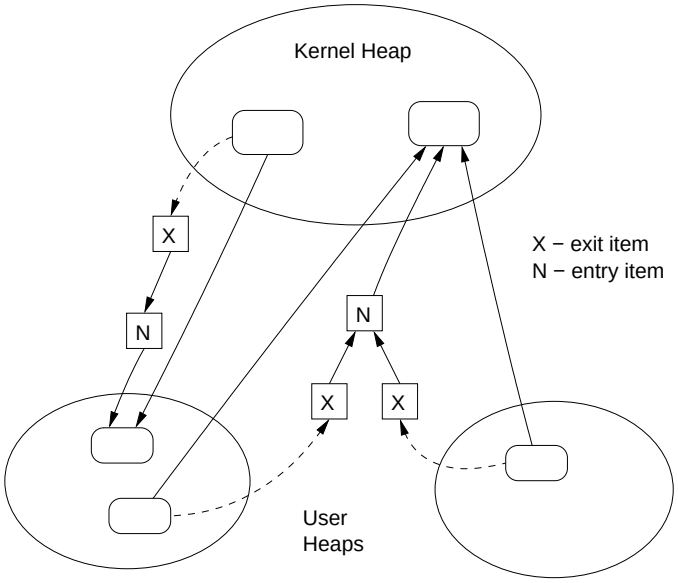


Figure 3.3. Use of entry and exit items. We keep track of each legal cross-heap reference through exit and entry items. Multiple exit items can refer to the same entry item if an object is shared by more than one heap. Dashed lines are used to denote that objects do not need to store a direct pointer to the exit items; instead, they can be found through a hashtable indexed by the remote address.

reaches zero, the entry item is removed, and the referenced object can be garbage collected if it is not reachable through some other path.

3.2.2.2 Interheap Cycles and Merging Heaps

Since cross-references between heaps are reference counted, we must pay particular attention to cycles. Reference counting cannot reclaim cycles of objects, even if they are not reachable through any path. Our rules for allowing cross-references restrict the possible types of interheap cycles that can occur. Since cross-references between user heaps are not legal, there can be no cycles between user heaps. The only form of interheap cycles that can occur is between the kernel heap and user heaps.

A user heap can acquire references to the kernel heap when a user process executes a system call that returns a reference to a kernel object. For example, such

references are returned to provide kernel services. Once the entry and exit items are registered, the user process is free to store references to that object anywhere on its heap. For a cycle to be created, however, a reference to an object allocated on the user heap would have to be reachable from a kernel object. Writes to kernel objects can only be done by trusted kernel code; therefore, we can control what cycles are created, modulo any bugs.

Our kernel code tries to avoid cycles where possible. However, some cycles are unavoidable: for instance, a process object is allocated on the user heap, while its process table entry is allocated on the kernel heap. These two objects must reference each other. Our kernel allows such cycles only if they include system objects whose expected lifetime is equal to the lifetime of their owning processes. An object's expected lifetime is evident from its purpose: for instance, a process object lives as long as the process.

Interheap cycles are collected after a process terminates or is killed. After we destroy all its threads, we garbage collect the process's heap and merge the remaining objects into the kernel heap. All exit items that now point back into the kernel heap are destroyed, and their corresponding entry items are updated and reclaimed if their reference counts reach zero. Garbage collection of the kernel can then collect any unreachable cycles. Figure 3.4 illustrates these steps.

Because we allow user-kernel cross-references, there is the theoretical possibility that larger cycles spanning *multiple* heaps could be created accidentally. Such a scenario would require intermediate objects on the kernel heap that connect different user heaps. Such occurrences would be considered kernel bugs. Such bugs do not compromise type safety; memory leaks are their worst consequence.

The design decision that heaps be merged upon termination does not unduly limit the options available when implementing the memory allocator and garbage collector. For nonmoving collectors, merging can be implemented merely by recoloring the objects (or pages of objects). Implementations that would have to move the objects will not have to move many objects, because we garbage collect the user heap before merging. Only objects that were kept alive by objects on the kernel

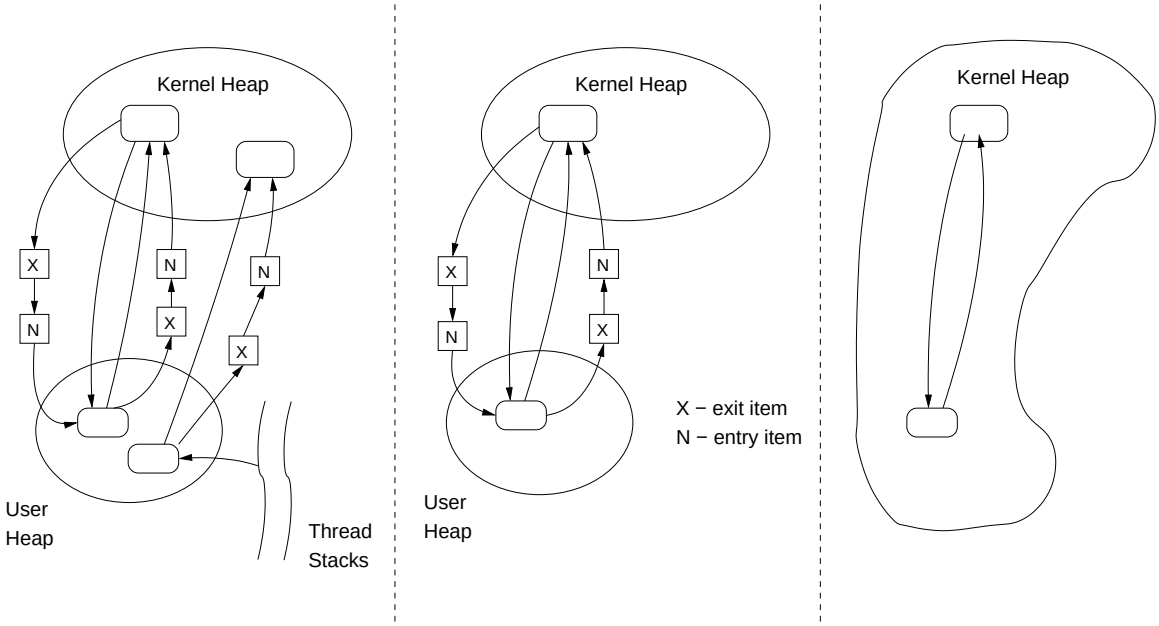


Figure 3.4. Terminating a process. First, all threads are terminated. Second, a GC cycle is run. Finally, the heaps are merged, and unreachable cycles can be collected.

heap have to be merged. For this reason, the collection of entry and exit items while merging heaps should not add substantial overhead.

3.2.2.3 Thread Stack Scanning

In a garbage-collected system, objects can be kept alive if they are referenced from any thread's activation records, as in the case of local variables that hold object references. For this reason, a garbage collector includes all valid references on thread stacks and in machine registers in its set of *roots*. All objects that are reachable on a directed path from the roots are considered alive. To obtain these roots, the threads generally have to be stopped for a brief period of time so that a snapshot of the roots located on their stacks can be taken. Depending on the garbage collection algorithm, this snapshot can be taken at the beginning (in *snapshot-at-beginning* algorithms), or at some point during the collection (in *incremental update* algorithms [87]).

The local variables and registers of all processes' threads must be included in the root set whenever a heap is garbage collected. This necessity follows from the following scenario in which a thread performs a system call that needs to gather information about another process, then during the execution of that system call the thread needs to access objects located on the other process's heap. Access to such objects requires references to them from that thread's local variables or registers. These references need to be taken into account when determining which objects are reachable. Our write-barrier mechanism does not eliminate this requirement. Write barriers prevent a process only from storing a reference to a foreign object in its own heap, but they do not prevent that process from otherwise using that object. For instance, invoking a method such as `Object.hashCode()` does not require that a reference to the object be stored on the heap, but type safety would be compromised if the object were not kept alive.

The price for examining all threads is not prohibitive. Foreign threads do not have to be stopped for more than a miniscule amount of time, which we show in Section 5.2. In our implementation, this amount of time is less than the granularity of the time slices with which the scheduler schedules threads. The expected time is small, because there are typically few references from foreign threads into the heap that is collected, which means that most references found on foreign thread stacks can be rejected quickly when scanning those stacks.

Additional optimizations could be added. For instance, the information of a stack scan could be saved and made available to all collectors (a time-space trade-off). Incremental thread stack scanning schemes [19] could be used to reduce the time required to scan a stack, and a stack does not have to be scanned multiple times while it is suspended.

One disadvantage of the necessity to scan all threads is that it scales badly in the worst case. Theoretically, one process could deny service to other processes by creating a large amount of threads and increasing the other processes' garbage collection loads. However, creating a thread is possible only through the kernel. This vulnerability is similar to other attacks against kernel services, which can be

avoided by imposing limits on the number of threads a process can create. In addition, the optimizations described above would limit the problem.

3.3 Interprocess Communication

We must ensure that all communication mechanisms are subject to protection and resource control. This task also mirrors the responsibility of an operating system kernel. For example, a kernel must bound IPC port queues to ensure that one process does not deny communications service to another process. In systems that employ capabilities, such as Mach [1] or Fluke [36], the kernel must track capabilities to ensure that a process does not acquire communication rights that it should not have. In systems that provide shared memory or memory-mapped files, the kernel provides memory that is shared between processes. The memory mappings into the participating processes' address spaces must be established in kernel mode to ensure that processes cannot violate any protection guarantees and to maintain sufficient isolation between the processes.

There is a trade-off between the difficulty of achieving isolation and the costs of communication between processes. If resources are completely separated, e.g., no memory is shared, isolation is easier to guarantee, but the cost of communicating is higher. If resources are shared, communication costs are lower, but it becomes harder to provide isolation and resource management. Before describing the design and properties of the communication model used in KaffeOS, we discuss this trade-off between memory control and sharing for different options of data sharing.

3.3.1 Sharing Models

A *sharing model* defines how processes can share data with each other. We discuss three possible models: copying, indirect sharing, and direct sharing. In a runtime system that supports processes, the choice of sharing model affects how memory accounting and resource reclamation can be implemented, and vice versa.

3.3.1.1 Copying

To share data when address spaces are not shared, the data must be copied. For example, copying is necessary when two processes are on different machines. Copying was the traditional approach to communication in RPC systems [12], although research has been aimed at reducing the cost of copying for same-machine RPC [11]. Mach [1], for example, used copy-on-write and out-of-line data to avoid extra copies. Java's version of RPC, called remote method invocation (RMI [17]), uses copying in its object serialization framework.

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.

However, the use of copying as the sole 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 a major advantage of using Java.

3.3.1.2 Indirect Sharing

In the *indirect sharing* model, 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 contains a reference to the shared object and passes a pointer to the proxy object. Proxies are system-protected objects. 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. Indirect sharing is used in the J-Kernel [42, 43] system.

The advantage of indirection 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. One disadvantage of indirect sharing is that the use of proxies slightly increases the cost of using shared objects. Another disadvantage is that indirect sharing requires the programmer to pay careful attention to arguments passed to methods that are invoked on shared objects. Such arguments must be proxied to prevent the leaking of direct references.

3.3.1.3 Direct 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. 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 at a finer granularity.

If a system supports direct sharing between processes, then process termination and resource reclamation are complicated. If a process exports a directly shared object arbitrarily, it is possible that that object cannot be reclaimed when the exporting process is terminated. As discussed in Section 3.2.1, all exported references to a shared object must remain valid, so as not to violate the type-safety guarantees made by the Java virtual machine. As in the case of user/kernel sharing, we suitably restrict references while still maintaining most of the benefits of direct sharing. Unlike user/kernel sharing, however, our direct sharing model does not assume that one sharing party is trusted.

3.3.2 Direct Sharing in KaffeOS

In KaffeOS, a process can dynamically create a shared heap to communicate with other processes. A shared heap holds ordinary objects that can be accessed in the usual manner. Shared objects are not allowed to have pointers to objects on any user heap, because those pointers would prevent the user heap's full reclamation. This restriction is enforced by our write barriers; attempts to create such pointers

result in an exception. Figure 3.5 shows which references are legal between user heaps, shared heaps, and the kernel heap. References between a shared heap and the kernel heap are legal in both directions, similar to references between a user heap and the kernel heap.

Shared resources pose an accounting problem: if a resource is shared between n sharers, should all n sharers be charged $1/n$ of the cost? In such a model, the required contribution of each sharer would grow from $1/n$ to $1/(n-1)$ when a sharer exits. Sharers would then have to be charged an additional $1/(n-1) - 1/n = 1/n/(n-1)$ of the total cost. As a result, a process could run out of memory asynchronously through no action of its own. Such behavior would violate isolation between processes. Therefore, we charge all sharers in full for shared data when they obtain references to it and reimburse sharers in full when they are done using the shared data.

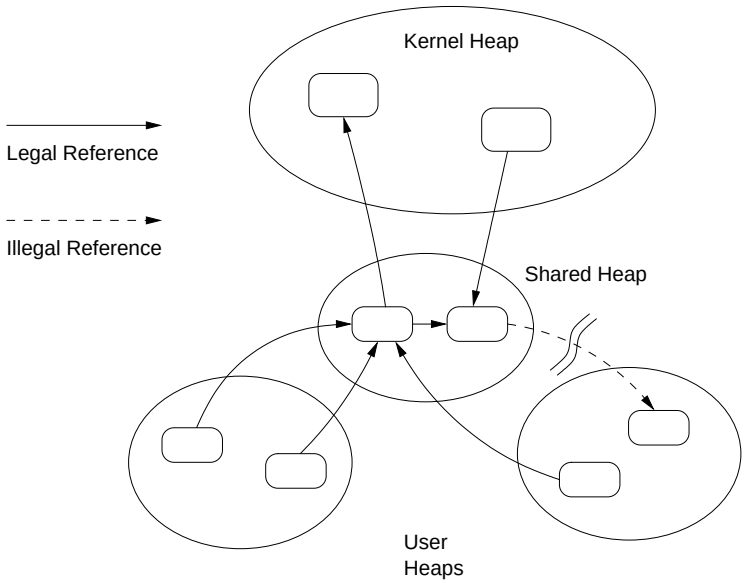


Figure 3.5. Valid cross-references for shared heaps. References from a user heap to a shared heap are legal and necessary to share objects. References from a shared heap to any user heap would prevent full reclamation and are therefore illegal. Shared objects may refer to kernel objects and vice versa.

A similar problem would occur if one party could expand the size of the data that is shared. If the size of the shared data increased over time, a process could asynchronously run out of memory. To avoid this problem, we allow processes to share only data of fixed size. This decision may lead to some waste of memory, because applications that do not know how much data they will share may have to overprovision the shared heaps they use.

Our model therefore guarantees three properties:

- One process cannot use a shared object to keep objects in another process alive.
- All sharers are charged in full for a shared heap while they are holding onto the shared heap, whose size is fixed.
- Sharers are charged accurately for all metadata, such as class data structures.

A shared heap has the following lifecycle. First, one process picks one shared class out of a central shared namespace, creates the heap, loads the shared class into it, and instantiates a specified number of objects of that shared class. During this instantiation, each object's constructor is invoked using the Java reflection API. A shared class can execute arbitrary code in its constructor, which includes instantiating objects of additional classes, which are also loaded onto the shared heap. Typically, a single object of a shared class is constructed, which serves as a wrapper to jump start the process of populating the shared heap. For instance, such a wrapper object could build a more complex data structure, such as a tree or hashmap, and insert objects whose content is read from a file.

While the heap is being created, its memory is charged to its creator. The creating process does not need to know the actual size of the shared objects in advance, which would often be impractical.

After the heap is populated with classes and objects, it is frozen, and its size remains fixed for its lifetime. Kernel code keeps track of what shared heaps are currently in use and provides a lookup service to user processes. Other processes can use this service to look up a shared heap; this operation returns a direct reference to

the objects in it. When a process looks up a shared heap, it is charged the amount established when that heap was frozen. The lookup fails if the amount exceeds the potential sharer's memory budget.

If a process drops all references to a shared heap, all exit items to that shared heap become unreachable. After the process garbage collects the last exit item to a shared heap, that shared heap's memory is credited to the sharer's budget. When the last sharer drops all references to a shared heap, the shared heap becomes orphaned. The kernel garbage collector checks for orphaned shared heaps at the beginning of each kernel GC cycle and merges them into the kernel heap. This per-heap cost must be bounded by placing limits on the number of shared heaps a process can create during a given time period. Otherwise, a process could repeatedly create and abandon shared heaps and deny service to other processes.

Metadata related to shared classes and objects are allocated on the shared heap. Because the shared heap is frozen after it is populated with objects, we need to ensure that no further allocations of metadata are necessary afterwards. For this reason, we eagerly perform allocations that would otherwise be performed lazily in Java, such as the compilation of bytecode to native code and the resolution of link-time references. We do not envision this to be a problem since data are not shared casually in KaffeOS, but for the purposes of interprocess communication. Therefore, it is unlikely that too much unnecessary work will be done—unnecessary work is done only if the shared classes contain dead code that a lazy compiler would not have compiled and its symbolic references would not have been resolved. This situation could happen if the class contains debugging code (such as a static `main` method for testing) or if the class was adapted from existing code, but we expect that overhead to be small.

3.3.2.1 Programming Model and Trust

Processes exchange data by writing into and reading from shared objects and by synchronizing on them in the usual way. Acquiring references to the objects requires the invocation of a special API, but making use of the objects does not.

The fixed size of a shared heap and the fact that attempts to write references to

user heap objects into shared objects will trigger segmentation violation exceptions impose some restrictions on the programming model used. For instance, certain dynamic data structures, such as variable-length queues or trees with a dynamically changing number of nodes, cannot be efficiently shared. Their fixed-sized counterparts can be shared, however. For instance, it is easily possible to share such data structures as read-only tries that are used for dictionary lookup. Section 5.4 provides a thorough discussion of the restrictions imposed by our shared programming model for a selected set of examples.

Our direct sharing mechanism is designed not to require the kernel to trust the code associated with any shared classes. Untrusted parties, without additional privileges, can share data; however, the communicating parties must trust each other. Our sharing mechanisms do not defend a server against an untrusted client. If an untrusted client is terminated while operating on a shared object, the shared object might be left in an inconsistent state. If the shared classes' code were trusted, the runtime's security policy could permit the code to enter kernel mode to protect the shared data structure. In this manner, we could implement trusted servers that protect the data structures they share with untrusted clients. Such an arrangement would be analogous to in-kernel servers in microkernel-based systems [52].

3.4 Hierarchical Resource Management

In KaffeOS, the kernel manages primary resources. Primary resources are those that must be shared by all processes, such as CPU time, memory, or kernel data structures. Kernel data structures can be managed simply by applying conventional per-process limits. In this section, we focus on the mechanisms for setting resource management policies for memory and CPU time.

We use hierarchical resource management schemes. In a hierarchical scheme, resources can be subdivided along a tree, such that the sum of resources used by a subtree corresponds to the amount of resources assigned to the root of that subtree. Such an arrangement is useful in many situations, as it mirrors the assignment of "real-life" resources. For instance, a server administrator may decide to assign 80%

of a resource to "premium" customers and the remaining 20% to the remaining customers. Within the premium group, different customers may get still different shares of that resource.

In our resource framework, we represent resources as objects. A resource object is like a capability, in that its holder can obtain a certain amount of a resource, or pass it to a new process it creates. Applications can employ different policies by creating different resource objects and assigning them to processes. For instance, a servlet engine can create resource objects for the servlets it spawns, and allot the desired amount of a resource to them.

3.4.1 Memory Management

To manage memory, we associate each heap with a *memlimit*, which consists of an upper limit and a current use. The upper limit is set when the memlimit is created and remains fixed for its lifetime; only the current use portion is updated. Memlimits form a hierarchy: each one has a parent, except for a root memlimit. All memory allocated in a heap is debited from its memlimit, and memory collected from a heap is credited to its memlimit. This process of crediting/debiting is applied recursively to the node's parents, according to the following rules.

A memlimit can be *hard* or *soft*.[1] This attribute influences how credits and debits percolate up the hierarchy of memlimits. A hard memlimit's maximum limit is immediately debited from its parent, which amounts to setting memory aside for a heap. Credits and debits are therefore not propagated past a hard limit after it has been established. For a soft memlimit's maximum limit, on the other hand, no memory is set aside, so a process is not guaranteed to be able to allocate its full limit. It cannot, however, individually allocate more than this limit. All credits and

[1]NB: The adjectives *hard* and *soft* were borrowed from hard and soft currencies in economics: liquidity in a hard currency typically guarantees that a sought after resource can be acquired. The purchasing power of a soft currency depends on how much of that resource is available and on how many buyers are competing for it. Hard and soft memlimits should not be confused with hard and soft quotas in filesystems; in particular, a memlimit's softness does not imply that its upper limit can be temporarily exceeded.

debits of a soft memlimit's current usage are immediately reflected in the parent. If the parent is a soft limit itself, the process is applied recursively such that an allocation fails only if it would violate the maximum limit of any ancestor node.

Hard and soft limits allow different memory management strategies. Hard limits allow for memory reservations but incur inefficient memory use if the limits are too high. Soft limits allow the setting of a *summary* limit for multiple activities without incurring the inefficiencies of hard limits, but they do not reserve memory for any individual activity. They can be used to guard malicious or buggy applications where high memory usage can be tolerated temporarily. If one considers the example shown in Figure 3.6; the 64 MB root memlimit is split into a hard memlimit with 16 MB on the left, of which 9 MB are used, and a soft memlimit with 48 MB on the right. The soft memlimit is further subdivided into two soft memlimits with 30 MB each. Because these memlimits are soft, the sum of the maximum limits of

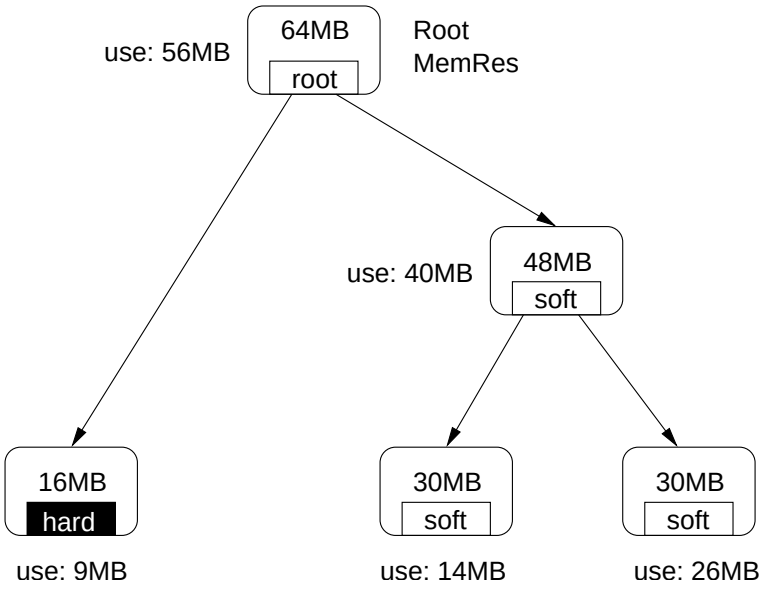


Figure 3.6. Hierarchical memory management in KaffeOS. The numbers inside the boxes represent limits; the numbers outside represent current use. A hard limit's limit is immediately debited from its parent; for soft limits, only the actual use is debited.

sibling nodes can nominally exceed the parent's maximum limit (30 MB + 30 MB = 60 MB is greater than 48 MB.) Doing so allows a child to use more than it would have been able to use had the memory been partitioned (for instance, one child uses 26 MB, which is more than the 48 MB / 2 = 24 MB it could have used had the 48 MB been evenly split.) The sum of the current use of these soft memlimits (14 MB + 26 MB = 40 MB) is the current use of their parent. However, the current use of the root memlimit is 56 MB, which is the sum of the current uses of its soft child node (40 MB) plus the maximum limit of its hard child node (16 MB).

Soft limits are also used during the creation of shared heaps. Shared heaps are initially associated with a soft memlimit that is a child of the creating process heap's memlimit. By default, their maximum limits are equal, but a smaller maximum limit can be chosen for the shared heap's memlimit. In this way, soft heaps are separately accounted for, but still subject to their creator's memlimit. This mechanism ensures that they cannot reach a size that is larger than their creator's resources.

3.4.2 CPU Management

Managing CPU time is both similar to and different from managing memory. Just as for memory, we must ensure that a process's CPU use is completely accounted for. Unlike memory, however, CPU time is committed only for short time quanta. Consequently, the issue of revocation does not arise for CPU time in the way it does for memory. Therefore, our scheduling mechanisms place no emphasis on limiting CPU consumption but use scheduling algorithms that allow processes to receive a guaranteed minimum share.

To completely account for a process's CPU use, all activity that is done on behalf of a process should be done in the context of that process. For this reason, we minimize the time spent in uninterruptible sections of code in the kernel. This is analogous to how packets should be processed by an operating system. For example, Druschel and Banga [32] 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. The use of a single kernel for system services, as

opposed to using a client-server architecture, as done in microkernel-based systems such as Mach [1], helps us to account for all activity in the proper process context.

We use a stride scheduler [84] for CPU scheduling. In stride scheduling, processes obtain CPU time that is proportional to an allotted share. Accordingly, our CPU resource objects represent a share of CPU. Stride scheduling is a work-conserving scheduling scheme: CPU time that a process does not use is not wasted but can be used by other processes. Therefore, it does not limit how much CPU time a given process uses under light load. Under full load, if all processes are runnable, a process uses only its assigned share.

The Java specification demands a priority-based scheduler for the threads within one application ([49], §17.12). Since our process model maintains the illusion that each process has a whole virtual machine to itself, we use stride scheduling to schedule processes and use a priority-based scheduler to schedule the threads within each process. A process's garbage collection thread runs at a higher priority than all other threads in that process. The garbage collection thread, the finalizer thread, and all other threads in the process are summarily subject to the stride scheduler's proportional share scheduling; i.e., the priority-based scheduler is active only when the global stride scheduler assigns CPU time to the process.

The kernel heap's collector and finalizer threads are the only threads not scheduled by the stride scheduler. They are always scheduled whenever they are runnable. This effect can be achieved by assigning an "infinite" number of tickets to them, which will cause the scheduler to always schedule them if they are runnable. As already mentioned, denial-of-service attacks against the kernel heap can be prevented by limiting the number of kernel operations a process can perform.

Figure 3.7 shows an example of a CPU resource hierarchy. The root CPU resource is subdivided into a 40% and a 60% share. The 40% share is further subdivided into a 20% and a 10% share (numbers given are with respect to the root resource), which leaves 10% in the original share. Shares are not assigned to threads, but to processes as a whole.

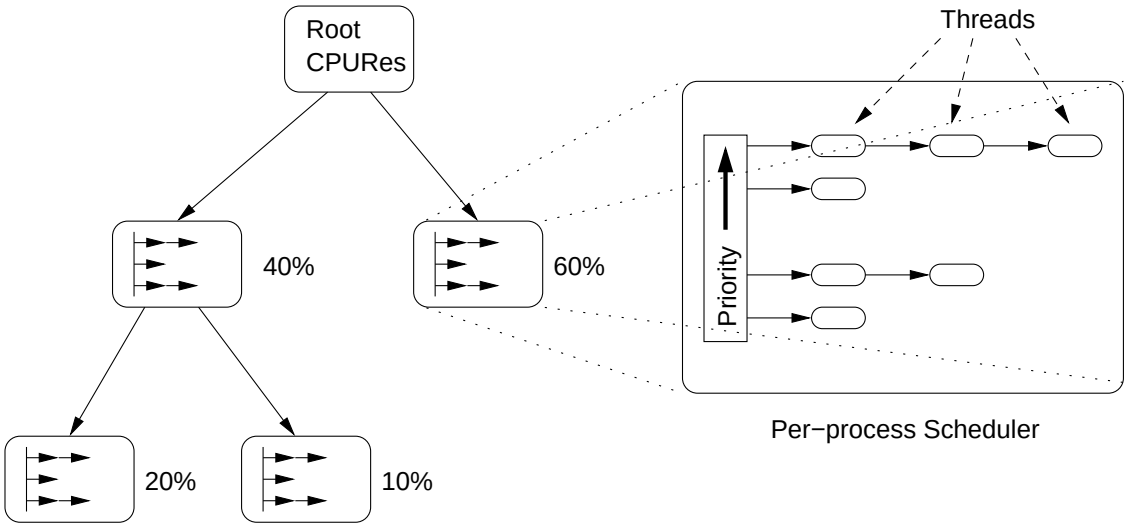


Figure 3.7. Hierarchical CPU management in KaffeOS. The root CPU resource is subdivided into a hierarchy. Each node constitutes a resource, which can be attached to a process. Within each process, a simple priority-based scheduler is used, which provides the same behavior as a stand-alone JVM.

3.5 Summary

Java's language-based mechanisms provide memory safety and security for KaffeOS user processes, which prevents them from accessing each other's data. However, they are insufficient to allow for safe termination of processes. Safe termination is provided by introducing a user/kernel distinction, or red line. This red line protects critical data structures by deferring termination during kernel code, which is written to avoid abrupt termination. Deferring termination is superior to the alternatives of using cleanup handlers.

Our memory management subsystem provides each process with a separate heap, which allows us to control the amount of memory available to a process. By implementing a process's logical resources as objects on that process's heap, we reduced the instances where limits on specific logical resources must be kept. Exceptions are instances either where resources of the underlying system are involved or where the creation of heaps themselves is concerned. The number of such

instances is smaller than the number of instances in which a hardware-based system would have to apply additional limits; we believe this reduction is an advantage of systems that employ software-based protection.

We use write barriers both to prevent illegal cross-references and to manage legal cross-references. Illegal cross-references are those that would allow sharing attacks and compromise the full reclamation of a process's memory. Legal cross-references are necessary for sharing; by using entry and exit items, we can separate the garbage collection activity of each process despite the presence of such cross-references.

KaffeOS's approach to sharing is best characterized by looking at the different situations in which objects could be shared, which can be summarized as follows:

- Sharing of *arbitrary objects between untrusted parties* is unsupported because the resulting violation of isolation and the compromise of full reclamation would trump possible benefits.

- Sharing of *common runtime data structures and common objects maintained by trusted code* is supported via the kernel heap, with no restrictions on the programming model.

- Sharing of *dedicated objects between untrusted parties* for the purposes of interprocess communication is supported via KaffeOS's shared heaps, albeit in a somewhat restricted model.

Finally, KaffeOS provides a hierarchical resource framework for CPU time and memory to allow for flexible resource management policies. CPU and time memory are two physical resources that must be managed differently. For CPU scheduling, KaffeOS uses a work-conserving stride scheduler. Our memory management scheme assumes that virtual memory mechanisms are unavailable or inaccessible. Under this assumption, committed memory cannot be revoked and must therefore be limited. Reserving memory for processes, however, raises the specter of inefficient resource use because of partitioning. To avoid such inefficient use, KaffeOS's memory limits do not always imply a reservation. Instead, KaffeOS supports both

hard and soft limits on memory. Hard limits allow for reservations of memory, but this memory is not available to other applications. Soft limits do not provide reservations, but they make multiple applications subject to a common soft parent limit, which avoids the problem of partitioning.

CHAPTER 4

IMPLEMENTATION

We implemented our KaffeOS prototype by extending Kaffe [85], an existing Java virtual machine. Kaffe is publicly available under an open source license and has a small user community that actively develops it. It is not a very fast JVM, compared to industrial Java virtual machines. Its low performance stems mainly from its just-in-time compiler, which performs little to no optimizations. The just-in-time compiler does not inline methods or use sophisticated register allocation algorithms. Its memory allocation subsystem is also not optimized. However, Kaffe provides a sufficient base for our prototype, because it is mature and complete enough to run real-world applications. Our design does not rely on any specific properties of Kaffe and should apply to faster JVMs as well.

4.1 Kernel

The original Kaffe code used a nonpreemptive, cooperative threading system. We developed a preemptive threading system, integrated it, and made the JVM runtime thread-safe in the face of preemptive threads.

The original Kaffe source does not include a user/kernel boundary (red line). Instead, it assumes that it is safe to throw an exception at almost any place. This assumption regularly leads to corrupted data structures and frequent deadlocks when threads exit abnormally. Such abnormal situations include not only those with which a regular JVM is not normally expected to deal, such as resource exhaustion, but also those that arise when other errors occur, such as ordinary loading or linking errors. The insight that a red line is necessary stems from our attempts to find a fix for these problems.

We completely restructured the runtime system to establish the red line and create a kernel that can safely back out of all exceptional situations. We examined all parts of the Kaffe runtime to see whether they could be run in user-mode or must be run in kernel mode. We also established a system of JVM-internal calling conventions that included a way for errors to be signaled in kernel mode, which are then propagated to a caller in user mode, where they can be converted into exceptions.

Although the need for deferred termination does not generally coincide with the need for synchronization, as discussed in Section 3.1.3, we found that the two needs overlap in certain cases in which global locks are acquired. For instance, the just-in-time compiler uses global data structures that must be protected against concurrent access and abrupt termination. In such cases, we combine acquiring a lock and disabling termination.

We implemented basic kernel services as classes in a package `kaffeos.sys.*`. We implemented as many methods as possible in Java and resorted to native methods only when necessary, mainly for glue code to the portions of the virtual machine written in C. Most of the kernel-related services are implemented in the classes `Kernel`, `Heap`, and `Process`. Other kernel services, such as opening a file or network socket, are implemented by adapting the existing classes in the Java runtime libraries, such as `java.net.Socket`.

`Kernel` implements basic functionality to bootstrap and shutdown the system or to enter and leave kernel mode; it also provides methods to control various internal kernel properties. `Heap` implements access to the (native) implementation of multiple heaps, it allows for the creation of new heaps, and it provides methods to manipulate existing heaps. It also provides functionality for deep copying of objects between heaps. Deep copying duplicates an object and the transitive closure of the objects to which that object refers; for instance, when deep copying an array of elements, each of its elements must be copied. Deep copies are necessary when arguments to a system call must be passed from one process to another. For instance, a process's command line arguments must be deep copied, because a

child process is not allowed to refer directly to the arguments stored in its parent process.

The `Process` class provides the following functionality:

- bootstrapping of new processes,
- code to safely kill a process and invoke any necessary cleanup handlers to free its logical resources,
- handles that can be used to control running processes,
- process-local properties, such a timezone or language-specific output settings

As discussed in Section 3.1.2, we changed all kernel code not to synchronize on objects that are exposed to user code. A typical code transformation is shown in Figure 4.1. The `Thread` class provides a method called `join()`, which can be used to wait for a thread to complete execution. The current state of the thread is stored

```
/* original code */
class Thread {
  private int state;

  public synchronized void join() {
    while (state != Thread.DONE) {
      this.wait();
    }
  }

  private synchronized void finish() {
    state = Thread.DONE;
    this.notifyAll();
  }
}
```

```
/* KaffeOS version */
class Thread {
  private int state;
  private Object synch = new Object();

  public void join() {
    try {
      Kernel.enter();
      synchronized (synch) {
        while (state != Thread.DONE) {
          synch.wait();
        }
      }
    } finally {
      Kernel.leave();
    }
  }
}
```

Figure 4.1. Protecting kernel-internal locks. Instead of using a `synchronized` method, an internal private field is used. The lock is acquired and released in kernel mode. The transformed version of `finish()` is not shown.

in a variable called `state`, which is set to `DONE` when the thread finishes. Because this variable is accessed by both the finishing thread and any threads waiting for the thread to finish, it must be protected by a lock. Since `join()` and `finish()` are `synchronized` methods, the publicly exported `Thread` object functions as this lock. One thread could prevent another thread from finishing by not releasing the lock. In addition, if a thread is killed while holding the lock, the lock may not be released. Finally, if the thread is killed in `finish()` between setting the state and signaling the state change via `notifyAll()`, joining threads may not be woken up and would subsequently hang. By using the internal private field `synch` as a synchronization object and by acquiring and releasing the lock in kernel mode, both problems are avoided.

4.2 Namespace and Class Management

We use Java's class loading mechanism to provide KaffeOS processes with different namespaces. This use of Java class loaders is not novel but is important because we have tried to make use of existing Java mechanisms when possible. When we use standard Java mechanisms, we can easily ensure that we do not violate the language's semantics. Other language-based systems provide different name spaces in other ways: for example, the SPIN project used a mechanism called domains, implemented by a dynamic linker [70].

As discussed in Section 2.2, Java represents its runtime types as class objects, and each class object is distinguished by its defining class loader. We exploit this fact to our advantage in two ways: First, we use multiple loaders if we cannot allow or do not want processes to share classes. Second, we delegate to a common loader for processes to share classes.

Reloading classes: *Reloading* is the multiple instantiation of a class from identical class files by different loaders. Although these instantiations constitute different types, they use the same code, and thus their visible behavior is identical.

Reloaded classes are analogous to traditional shared libraries. Reloading a class gives each instance its own copies of all static fields, just as a shared library uses a

separate data segment in each process in which it is mapped. Reloaded classes could share text, although our current implementation does not support such sharing. Sharing text could be accomplished by changing the just-in-time compiler to emit code that follows shared-library calling conventions. Such code would use indirect calls through a jump table for calls that lead outside the library's code segment.

Sharing classes: We can share classes between different processes' namespaces by delegating requests for them to a common loader. We share classes in two cases: either because the class in question is part of the runtime library or because it is the type of a shared object located on a shared heap. We refer to the former as system-shared and to the latter as user-shared. System sharing of classes makes more efficient overall use of memory than reloading them, because only a single class object is created. Even if our implementation shared the text and constants of reloaded classes, the advantage of having system-shared classes would likely still be significant, because a system-shared class needs to maintain only a single copy of its linking state. However, system-shared classes are not subject to per-process accounting, because we assume that such sharing benefits all processes. Their number and size are bounded, because applications cannot add system-shared classes. Our goals in developing our class-loading policy were to gain efficiency by maximizing system sharing and to maintain correctness and ease of use for user-shared classes. At the same time, we had to take into account the sometimes conflicting constraints imposed by the Java API and Kaffe's linker implementation, which we discuss below.

Process loaders: Each KaffeOS process has its own class loader, which manages that process's namespace. The process loader is a trusted system object that implements the loading policy. When asked for a class, it decides whether the class is shared or not. If so, the request is delegated to either the system loader (for system-shared classes) or a shared loader (for user-shared classes). If the class is not shared, it is either a system class that must be reloaded or a regular user class. For user classes, our default implementation uses the standard Java convention of loading classes from a set of directories and compressed archives that are listed in

a `CLASSPATH` variable.

To ensure full namespace control, the process loader must see all requests for classes, including those from user-created class loaders in that process. By default, user-created class loaders in standard Java first attempt to delegate to the system loader to ensure that system classes are not replaced by user classes. We changed the class loader implementation to delegate to the process loader whenever a user-created loader would delegate to the system loader in a conventional JVM. KaffeOS's model of delegation is depicted in Figure 4.2.

System-shared classes: To determine which classes can be system-shared, we examined each class in the Java standard libraries [17] to see how it acts under the semantics of class loading. In particular, we examined how classes make use of static fields. Because of some unfortunate decisions in the Java API design, some classes export static fields as part of their public interfaces. For example, `java.io.FileDescriptor` exports the public static variables `in`, `out`, and `err` (which correspond to stdin, stdout, and stderr.) Other classes use static fields internally.

To system-share classes with static fields, either we must conclude that it is safe to leave the fields unchanged, we must eliminate the fields, or we must initialize the

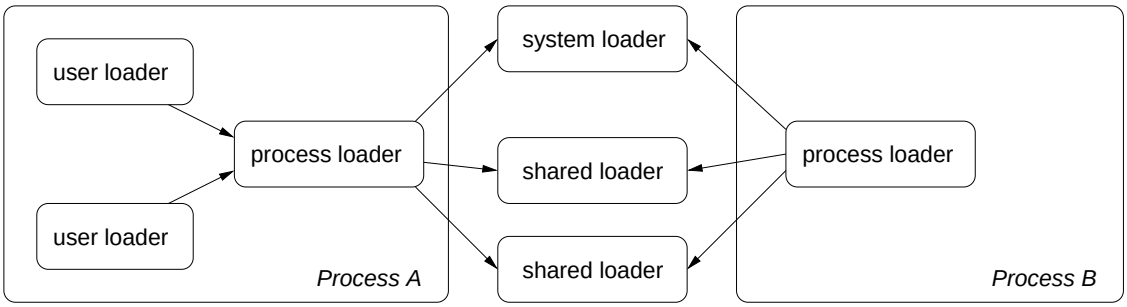


Figure 4.2. Class loader delegation in KaffeOS. Each process has its own process loader, which is that process's default loader. The process loader fields all requests for system classes by user-defined loaders in the same process; such loaders cannot directly delegate to the system loader. The process loader either reloads a class or delegates to the system loader or a shared loader.

fields with objects whose implementation is aware of KaffeOS's process model. Final static fields with a primitive type such as `int` are constants in Java and can be left untouched. Elimination of static fields is sometimes possible for singleton classes, which can be made to use object fields instead of static fields. If static fields cannot be eliminated, we examine whether they can be initialized with objects whose implementation is process-aware. If an object stored in a static field is used only by invoking its methods, we can provide a substitute implementation that provides the same methods but modifies their behavior to take the currently executing process into account. For instance, a process-aware version of the `System.out` stream maintains a stdout stream for each process and dispatches output to the appropriate stream.

If the object's fields are directly accessed by other classes, then creating a substitute implementation becomes more complicated, because it requires changes to the code that uses that object. As a practical matter, we tried to make as few code changes as possible, to allow for easy inclusion of upgrades and bug fixes developed for the code base upon which KaffeOS is built. Reloading classes allows us to use a class with static fields practically unchanged in a multiprocess environment. Consequently, we reload some classes that could have been rewritten to be shared.

Linker constraints: The decision of whether to share or reload a class is subject to linker-specific implementation constraints as well. First, shared classes cannot directly refer to reloaded classes, because such references are represented using direct pointers by the runtime linker and not through indirect pointers as in shared libraries. Second, certain classes must be shared between processes. For example, the `java.lang.Object` class, which is the superclass of all object types, must be shared. If this type were not shared, it would not be possible for different processes to share generic objects. Consequently, the transitive closure of classes pointed to by `Object` has to be shared.

We used the JavaClass bytecode engineering library [26] to develop a small tool that determines the transitive closure of classes referenced by a class. JavaClass has

the ability to parse class files and provide access to the symbolic linking information contained therein. We used the tool on the classes we needed to share. For `java.lang.Object`, we found that its closure turns out to be quite large: `Object` refers to `java.lang.Throwable`, which is the base class of all exceptions, which in turn refers to `java.lang.System` in its `printStackTrace` method. `System` is a complex class with a multitude of references to other classes.

Fortunately, we can rewrite classes to remove direct references. This transformation requires the use of reflection to access fields or methods in a class. An example of such a transformation is given in Figure 4.3, which depicts the initialization code sequence for the `System` class. In the original version, the `out` variable was implemented as a simple `PrintStream` layered on top of a buffered file stream, which in turn relied on a `FileOutputStream` constructed using the stdout file descriptor `FileDescriptor.out`. As discussed earlier, `System.out` is replaced with a process-aware object that dispatches the output to the current process's stdout stream; `println()` is shown as an example. The current process's output stream is initialized with the current process's `FileDescriptor.out` in a method called `kaffeos_init()`. Because `FileDescriptor` is reloaded, reflection is used to resolve the name `java.io.FileDescriptor` in the context of the current process's loader, which avoids a direct reference from `System` to `FileDescriptor`. By convention, the static `kaffeos_init()` method, if it exists, is invoked for every shared class at process initialization time. It serves as an analogue to static initializers in reloaded classes. Code that would otherwise be in a static initializer is moved to `kaffeos_init()` if process-local data must be initialized.

When writing native methods for reloaded classes, we must ensure that references to class objects and static fields are properly handled. In standard Java, a native library can be loaded only by a single class loader ([53], §11.2.4). This restriction was introduced to prevent native code from mistakenly intermixing classes and interfaces from different class loaders. Such intermixing could occur if native libraries cached pointers to class objects and static fields between calls. Such caching is safe if classes that contain native methods are not reloaded. KaffeOS

```
public final class System {
    final public static PrintStream out;

    static {
        // direct reference to reloaded FileDescriptor class
        FileOutputStream f = new FileOutputStream(FileDescriptor.out);
        BufferedOutputStream b = BufferedOutputStream(f, 128);
        out = new PrintStream(b, true);
    }
}
```

```
/* KaffeOS version */
public class ProcessStdoutPrintStream extends PrintStream {
    /* process-aware version */
    void println(String s) throws IOException {
        Process.getCurrentProcess.getStdout().println(s);
    }
}

public final class System {
    final public static PrintStream out = new ProcessStdoutPrintStream();

    private static void kaffeos_init() {
        Class fdclass;
        fdclass = Class.forName("java.io.FileDescriptor", true,
                                Process.getRootLoader());
        Field f = fdclass.getField("out");
        ((ProcessStdoutPrintStream)out).createStdoutStream(f.get(null));
    }
}
```

Figure 4.3. Transforming library code. The original version of `System.out` was implemented as a simple `PrintStream` layered on top of a buffered file stream, which in turn relied on a `FileOutputStream` constructed using the stdout file descriptor `FileDescriptor.out`. Because `System` must be shared, `System.out` is replaced with a process-aware version that dispatches output to the current process's stdout stream. As in the original version, this stream is initialized with the current process's `FileDescriptor.out`. Because `FileDescriptor` is reloaded, reflection is used to avoid a direct reference from `System` to `FileDescriptor`.

does not impose this restriction and reloads classes with native methods. Therefore, we must ensure that its native libraries do not perform such caching.

Adapting the runtime libraries to KaffeOS was a compromise between our desire to share as many classes as possible to increase efficiency, yet at the same time exploit reloading to avoid code changes. Out of roughly 600 classes in the core Java libraries, we are able to safely system-share 430. This high proportion (72%) was achieved with only moderate changes to a few dozen classes and more extensive changes to a few classes, such as `java.lang.System`. The rest of the classes are reloaded, which requires no changes at all. The set of runtime classes that an application uses is, unsurprisingly, application-dependent. Table 4.1 shows the distribution for the runtime classes used by the SPEC JVM98 benchmarks. (See Section 5.1.2 for a detailed discussion of these benchmarks.) The proportion of shared classes lies above 72%, although we note as a caveat that these benchmarks do not exercise very many runtime classes; the results may therefore not be representative.

User-shared classes: To ensure that every process that has access to a shared heap sees the same types, process loaders delegate the loading of all user-shared types to shared loaders. Each shared heap has its own shared loader, which is created at the same time as the heap. Process loaders use a shared class's name to determine the shared loader to which the initial request for a shared class should be delegated. Java's class loading mechanism ensures that subsequent requests are

Table 4.1. Number of shared runtime classes for SPEC JVM98 benchmarks.

Benchmark	check	compress	jess	db	javac	mpegaudio	mtrt	jack
reloaded	8	8	8	8	8	8	8	8
shared	55	46	54	49	60	49	51	47

This table shows that of the runtime classes used by the SPEC JVM98 benchmarks, only a small number cannot be shared and must be reloaded. For all benchmarks, `java.io.FileDescriptor`, `java.io.FileInputStream`, and `java.io.FileOutputStream` are among the eight reloaded classes.

delivered to the shared loader that is the defining loader of the initial class. If we did not delegate to a single loader, KaffeOS would need to support a much more complicated type system for its user-shared objects.

The namespace for user-shared classes is global, which allows communicating partners to look up shared classes by name. We use a simple naming convention for this shared namespace: the Java package `shared.*` contains all user-shared classes and packages. The table that maps names to loaders is a global resource, and limits on the number of entries a process can create could be applied to prevent that table from growing indefinitely.

Like system-shared classes, user-shared classes cannot directly refer to reloaded classes. Because most classes that are part of the runtime library are shared, this limitation is not severe. As with system-shared classes, a developer of user-shared classes can use reflection to access reloaded classes. Failure to use reflection results in write barrier violations during linking, because the linker would attempt to create a cross-reference from a class object on a shared heap to a class object on a user heap.

4.3 Memory Management

We improved and adapted Kaffe's memory management subsystem to work in KaffeOS's multiprocess environment. The original code provided only a single heap and mostly used global data structures. We examined which data structures had to remain global and which had to be encapsulated and replicated for each heap. On this basis, we implemented cross-references and the merging of heaps. In this section, we focus on the *process-specific* implementation aspects of the memory management subsystem of our KaffeOS prototype.

The implementation consists of two parts: the memory allocator and the garbage collector. The memory allocator provides explicit memory management facilities similar to `malloc/free`. The garbage collector invokes the memory allocator to first allocate and later free objects.

4.3.1 Memory allocator

The memory allocator is logically divided into two components: the small object allocator and the primitive block allocator. The small object allocator maintains a pool of pages from which small objects are allocated. Each heap has its own pool of small object pages. The primitive block allocator maintains a single global pool of continuous memory regions that are comprised of one or more contiguous pages. Objects that are larger than one page and small object pages are allocated from the pool of primitive blocks.

A small object page is subdivided into units of equal size. This design has the advantage that it is possible to determine in constant time whether a pointer points to the start of an object, which is a frequently performed operation in our conservative garbage collector. The regular layout of objects also speeds up exception handling, because it allows the exception handler to quickly map a stored program counter to the just-in-time compiled code in which it is contained. (Since most Java methods are small [63] and since our just-in-time compiler does not perform any inlining, the size of the compiled code is often far smaller than a page, which is why our allocator stores them in small object pages.)

For each heap, the small object allocator keeps free lists for a number ($n = 19$) of small object sizes. These sizes were determined using a heuristic that minimizes the slack that occurs if the requested size is smaller than the next-largest small object size. We determined this heuristic by examining a histogram of the object sizes allocated during the execution of selected applications.

Keeping the free lists for small objects on a per-heap basis has the advantage that all internal fragmentation is fully accounted for. Once a page is taken from the primitive block list and prepared for use in a given heap, it is charged to that heap. As a result, it is impossible for a process to launch a fragmentation attack against KaffeOS by holding on to a few objects that are spread over multiple pages. A second advantage is that no cross-process locking is necessary during allocation, which reduces contention. Finally, because all objects within one page belong to the same heap, we need to store the heap identifier only once for all objects on a

page.

On the flip side, because a page will be charged to a heap even if only a single object is used, processes may be overcharged for memory. If we consider only live objects, we can expect an average overcharge of $n * pagesize/2$, where n is the number of freelists. For $n = 19$ small object sizes, the overhead amounts to 38 KB on a machine with 4KB pages. Picking a smaller n would reduce this overhead but would increase the slack that is wasted per object. Internal fragmentation can also lead to an overcharge when a heap is not garbage collected frequently enough, so that it allocates long-lived objects in new blocks instead of garbage collecting and reusing older blocks. The possible impact of this overcharge is difficult to predict, because it depends on an application's allocation and garbage collection patterns. However, overcharging because of internal fragmentation does not require an application to have a memlimit with a maximum limit greater than the maximum total size of its live objects, because a garbage collection is triggered when a heap reaches its maximum limit. This fragmentation issue occurs only because our collector does not move objects, and it is not inherent in our design.

Although internal fragmentation can be accounted for, our implementation is subject to external fragmentation of global memory. In practice, this fragmentation caused problems with lazily allocated kernel data structures, because scattered small object pages with long-lived kernel data fragmented the space of primitive block, which reduced the maximum size of primitive blocks available. To prevent the kernel from causing external fragmentation in this way, we adopted a work-around: when possible, we allocate kernel pages from a large continuous block at the bottom of the global memory pool. Figure 4.4 illustrates how blocks are allocated. The preferred area for kernel heap pages is shown on the top left. The primitive block allocator allocates blocks of memory whose sizes are multiples of a pagesize, and external fragmentation can occur. Like the internal fragmentation discussed earlier, this external fragmentation could be avoided if a moving collector were used.

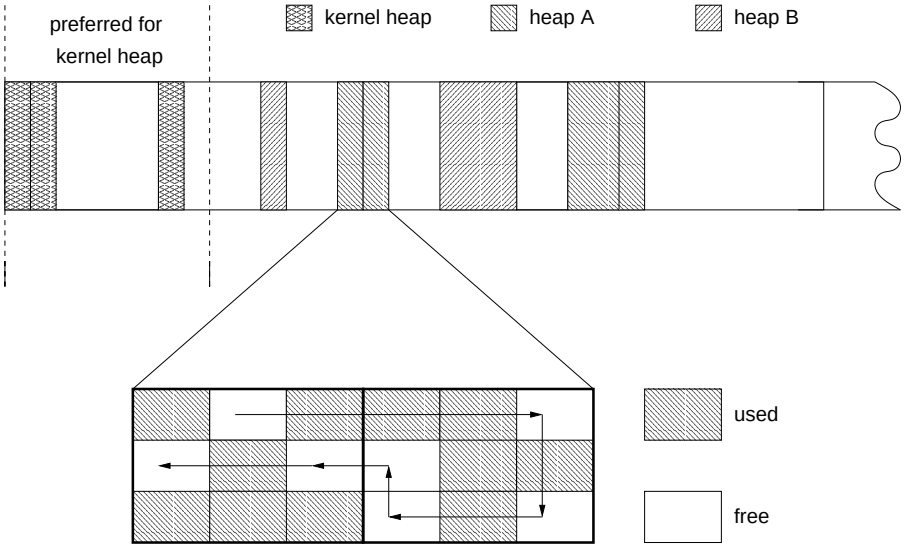


Figure 4.4. Internal and external fragmentation in KaffeOS. At the top, the primitive block allocator allocates blocks of memory whose sizes are multiples of a pagesize. External fragmentation can occur. The preferred area from which to allocate primitive blocks that are used for objects on the kernel heap is at the low end of the managed memory area, so as to reduce external fragmentation. At the bottom, the figure shows two small object pages for a given object size. All unused cells are kept in a singly linked freelist whose entries can span multiple small object pages. All internal fragmentation is accounted for.

4.3.2 Garbage collector

The original Kaffe provided only a conservative, nongenerational, nonincremental collector that walked all stacks and all objects on the heap conservatively. We improved its performance by using type information to walk the heap precisely,[1] but stacks are still walked conservatively for easier integration with native code. We use the tricolor scheme [28], which was originally developed to reason about incremental collectors, to explain how it works. In this scheme, one of three colors (black, grey, and white) is assigned to each object during garbage collection.

Initially, all objects start out white. Garbage collection roots are colored grey.

[1]Most programs allocate more than half of their space for nonreferences [27].

During the mark phase, all grey objects are walked and marked black. All white objects that are reachable from a grey object are greyed before the grey object is marked black. This process is repeated until there are no more grey objects. At the end of the mark phase, all black objects are kept and all white objects are either freed or become eligible for finalization. If an object is subject to finalization, its finalizer will be executed, the object will be marked as having been finalized, and it will be freed as soon as it again becomes unreachable. Objects that are waiting to be finalized are treated as garbage collection roots to ensure that they and all objects to which they refer stay alive. When walking the heap, we do not need to treat finalizable objects differently from other grey objects.

For correctness, we must ensure that the set of objects that we identify as reachable does not change during the mark phase, even if the reachability graph should change. In the simplest case, we could prevent all threads from running during the full GC cycle. We always stop all threads that belong to that heap's process. As discussed in Section 3.2.2.3, though, it would violate the separation between different processes if we prevented all remote threads from running for the full duration of the collection. The following argument shows that we do not need to stop all remote threads.

If an object is connected to the reachability graph only by a single edge, we call that edge a "last reference." Such references constitute the worst-case scenario for the garbage collector. Figure 4.5 depicts two objects, A and B, in a heap, as well as two remote threads, RT1 and RT2. In the figure, B's last reference is stored in A. The last reference can move to RT1's (or RT2's) stack if RT1 or RT2 reads the reference onto its stack and writes a `null` value into A. We focus on B as the object that we must keep alive, even if the location of its last reference changes between A, RT1, or RT2.

For either RT1 or RT2 to obtain a reference to B, there must have been an entry item R with some path to B (unless the heap is being bootstrapped, during which time no garbage collection takes place). In addition, since RT1 and RT2 are different threads, they can pass references to B between each other only by temporarily

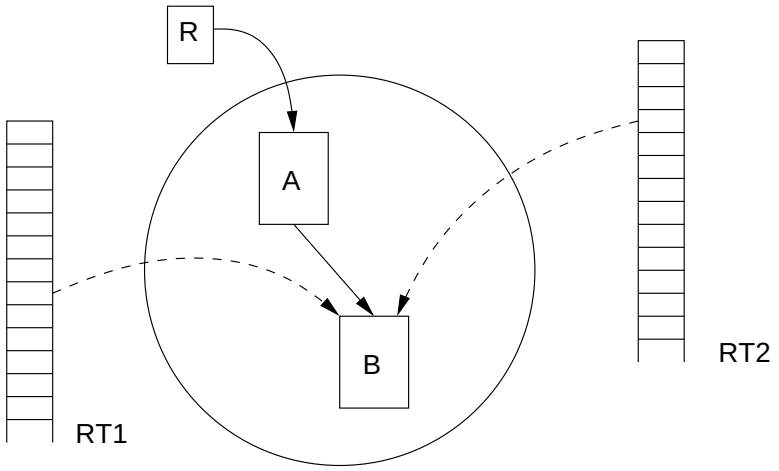


Figure 4.5. Worst-case scenario for remote thread stacks in KaffeOS. In this scenario, the last reference to object B can be stored in either A, RT1, or RT2. In either case, the garbage collector must keep B alive without stopping either RT1 and RT2 for the duration of the scanning phase.

storing them in an intermediate object, which we assume to be A without loss of generality. We need to consider only the case in which this intermediate object is located on the same heap as B: if the reference were stored into an object on the kernel heap, an entry item for B would be created, which would ensure that B is kept alive. In addition, write barriers prevent the storing of a reference to B into objects on other user or shared heaps.

KaffeOS does not trigger a write barrier violation when a remote thread manipulates A, because our write barrier policy allows writes that create intraheap references. In our example, it allows any—including remote—threads to write a reference to B into A. It will also allow the assignment of `null` to fields in A. This policy of not depending on the heap association of the executing thread is necessary to allow bootstrapping of new heaps by remote threads. Therefore, we cannot rely on write barrier violations to detect manipulations by remote threads.

We must ensure that object B is marked, independent of how the activities of threads RT1 and RT2 change the location where the last reference to B is stored.

We could miss B if the last reference is not on any thread's stack while we scan the stack and if the last reference is not reachable from A while we scan this heap's entry items. This situation can occur only if the last reference moves between these locations.

One option to address this problem would have been to implement incremental garbage collection, which can deal with concurrent mutators. However, this would have been expensive to do, given the infrastructure with which we worked. Our solution does not require a fully incremental collector.

Instead, we block those foreign threads that attempt writes into a heap when a garbage collection for that heap has been started. Usually, this blocking affects only few, if any, threads. Only foreign threads in kernel mode have a legitimate reason to write into a process's heap, and only if they perform kernel operations directly related to that process. The write barrier checks a per-heap flag called `gc_running` that each heap sets when a garbage collection has started (see Figure 4.6).

However, the write barrier code, because it is executed before the actual write, cannot perform this check atomically with the write. In other words, there is a time window in which a foreign thread could check whether a heap is being collected, see that its `gc_running` flag is false, and complete the write later during the garbage collection. Every thread can have at most one such outstanding write; consequently, the last reference cannot "ping-pong" between RT1's and RT2's stack, which would require two outstanding writes.

We adapt our collector's scan phase to take such belated writes into account. Our algorithm is shown in Figure 4.7. In Part 1, we first walk and mark all entry items. If B is reachable from R during that time, it is marked. One can assume it is not, and the last reference is on RT1's stack. In Part 2, we walk all remote thread stacks. If B is still reachable from RT1's stack when we walk it, it is marked. However, RT1 might have moved the last reference into A. For this reason, we recolor all black objects to be grey and rewalk them in Part 3. During this time, B will be marked. It is not possible for the last reference to have moved to RT2's stack, because RT2 cannot write `null` into A if it read the last reference written

```
/*   dst->field = src  */
write_barrier(dst, src)
{
    dcoll = dst.collector;
    while (dcoll->gc_running) {
        synchronize (dcoll) {
            while (dcoll->gc_running) {
                wait();
            }
        }
    }
    if (src == 0) return;  /* proceed with write */
    scoll = src.collector;
    if (check_cross_ref(scoll, src, dcoll, dst))
        return;            /* proceed with write */
    throw new SegmentationViolation();
}
```

Figure 4.6. Pseudocode for write barrier. Our write barrier code first determines the heap on which the destination object is located. It determines whether that heap is being collected and waits if it is. Otherwise, it determines the collector of the source object and checks whether either null is assigned or the cross-reference is legal. Exit and entry items are created or updated during `check_cross_ref()` as required.

by RT1 after the garbage collection started: the race could not occur. The last reference could not have moved back onto RT1's stack, either, because the race cannot occur twice in the same thread.

In summary, our thread stack algorithm needs to perform more work than in the single process case. In particular, it needs to walk each remote entry twice, as well as all objects reachable from them. Because entry items to user heaps can be created only by kernel code, there are generally few of them. For instance, a simple "Hello, World" program creates only two entry items. The collection of the kernel heap itself is a special case, where we stop all threads for the entire duration of GC.

4.4 Summary

We implemented a prototype of KaffeOS by extending and modifying the existing code base of the Kaffe JVM. The core aspects of our implementation covered in

```
begin_mark_phase(Collector *coll)
{
    coll->gc_running = true;
    atomic {                                        /* Part 1 */
        coll->mark_entry_items();
        coll->walk_grey_list();
    }

    forall (t in coll->RemoteThreadSet())           /* Part 2 */
        atomic {
            t->scan();
        }

    atomic {                                        /* Part 3 */
        coll->turn_black_objects_grey();
        coll->walk_grey_list();
    }
                                                    /* Part 4 */
    // walk local thread stacks
    // walk remaining gc roots
    // ...
}
```

Figure 4.7. Pseudocode for mark phase. Remote threads can run outside of blocks marked `atomic`. On a uniprocessor, `atomic` could be implemented simply by disabling interrupts. The time spent inside `atomic` blocks is bounded and typically small, because we expect few entry items for user heaps in Part 1 and because we expect few valid references from remote heaps in Part 2.

this chapter include the kernel, the namespace and class management subsystem, and the memory subsystem. This discussion is not exhaustive of the work we did to create our prototype; we also had to perform substantial implementation work in such areas as the implementation of the threading system, the stride scheduler, and the core class loading mechanism.

We implemented the KaffeOS kernel as a set of Java classes with corresponding native methods. Implementing the kernel involved implementing classes for process and heap management and introducing the red line to ensure safe termination. We introduced the red line by redesigning some sections of code and by applying transformations to others so that critical data structures are manipulated in kernel

mode and so that kernel code handles all exceptional situations carefully. This experience shows that the user/kernel boundary can be introduced into an existing code base; nevertheless, it should be used as a structuring principle from the beginning when designing JVMs or other runtime systems.

KaffeOS uses Java class loaders to provide its processes with separate name-spaces. The use of class loaders proved to have both advantages and disadvantages. Some advantages are that they are integrated into the language, that they do not compromise type safety, and that their impact on the type system is well-researched. Another advantage is that they allow for easy sharing of types via delegation to a common loader, which we use for both system-shared and user-shared classes. On the other hand, class loaders are an inefficient way to reload a class merely to obtain a process-local set of static variables, because our current implementation does not share text. We were able to mitigate this disadvantage by sharing many system classes; however, our KaffeOS implementation does not share any user classes.

We adapted Kaffe's memory allocator to provide support for our heap model, which includes the user heaps, shared heaps, and the kernel heap. The Kaffe allocator consists of two layers: a small object allocator and a primitive block allocator. We provided each process with its own small object allocator but kept a single primitive block allocator. As a result of this implementation decision, our implementation can defend against fragmentation attacks; however, our kernel heap may suffer from external fragmentation. We expect that different allocators, in particular those designed to work with precise or moving collectors, will require different implementation decisions.

CHAPTER 5

EVALUATION

We focus on three areas in the evaluation of our KaffeOS prototype. The first area is the run-time overhead KaffeOS introduces, when compared with a JVM that does not provide robust support for multiple processes. One particular source of concern is the use of write barriers for isolation, because it might be expected to cause substantial overhead. For instance, Dillenberger et al. rejected such use in a similar situation, reasoning that "intervention on every 'putxxx' operation code would degrade performance" ([29], page 206). We show that the overhead is reasonable when compared to the Kaffe VM upon which KaffeOS is based; we also project that the overhead would still be reasonable if KaffeOS were implemented on top of an industrial JVM.

Second, we evaluate KaffeOS's effectiveness in the area for which it was designed: supporting untrusted or buggy applications that may intentionally or inadvertently engage in denial-of-service attacks directed at primary computing resources. We show that KaffeOS can defend against denial-of-service attack directed at memory, CPU time, and the garbage collector and also show that KaffeOS's integrity is not compromised by such attacks. We compare KaffeOS to two competing existing approaches: running applications on top of an ad hoc layer in one JVM and providing multiple JVMs for multiple applications. In addition to effectiveness, we also compare scalability, as measured in the number of applications a given system can support, and find that KaffeOS can outscale the approaches using multiple JVMs.

Finally, we evaluate the practicality of KaffeOS's model for interprocess communication by way of shared heaps. As discussed in Section 3.3.2, our programming

model introduces some restrictions. We address the question of whether these restrictions could outweigh the advantages of being able to directly share objects. We illustrate the impact of these restrictions on the programming model using concrete examples. We demonstrate that KaffeOS's shared heaps allow easy sharing of data structures as complex as hashmaps or linked lists.

Our prototype runs as a user-mode application on top of the Linux operating system. All our measurements were taken on a 800MHz "Katmai" Pentium III, with 256 Mbytes of SDRAM and a 133 MHz PCI bus, running Red Hat Linux 6.2. The processor has a split 32K Level 1 cache and combined 256K Level 2 cache. We used the GNU C compiler (Version egcs–1.1.2) to compile the VM, and we used IBM's jikes compiler (Version 1.12) to compile the Java portions of the runtime libraries.

5.1 Write Barrier Overhead

KaffeOS's use of write barriers for isolation introduces some overhead when running applications. In this section, we evaluate this overhead both using microbenchmarks and real application benchmarks.

To measure the cost of write barriers in KaffeOS, we implemented several versions, which we labeled *No Write Barrier*, *No Heap Pointer*, *Heap Pointer*, and *Fake Heap Pointer*. The *No Write Barrier* version is our baseline for determining the write barrier penalty. The *No Heap Pointer* version is the default implementation, which has no space overhead. The *Heap Pointer* version is faster than the *No Heap Pointer* version but increases the size of each object's header; the *Fake Heap Pointer* version serves to isolate the space impact of the *Heap Pointer* version from its speed impact.

- *No Write Barrier*: We execute without a write barrier and run everything on the kernel heap. We implemented this version by instructing the just-in-time compiler not to insert a write barrier when compiling bytecode to native code. We used C preprocessor macros to ensure that no write barrier would be inserted in native libraries. We changed the function that creates new

heaps to return a reference to the kernel heap instead. The merging of heaps upon a process's exit is an empty operation in this version. The *No Write Barrier* version accounts for any possible performance impact of the changes made to Kaffe's runtime and provides the baseline version for examining the write barrier overhead.

- *No Heap Pointer*: This version is the default version of KaffeOS. For each small object page and for each large object, we store a heap ID in a block descriptor. To avoid cache conflict misses, the block descriptor is not kept at a fixed offset on the same page. Instead, block descriptors are stored in an array, whose index values are computed as affine transformations of the address of an object. At each heap pointer write, the write barrier consists of a call to a routine that finds both the source and the destination object's heap ID from their addresses and performs the barrier checks.

- *Heap Pointer*: In this version, we trade some memory for speed: we increased each object's header by 4 bytes, which we use to store the heap ID. In this case, extracting the heap ID is substantially faster, because the block descriptor does not have to be read from memory to retrieve the heap ID. We coded the fast path (i.e., where our policy allows the write to complete) for this barrier version in assembly language.

- *Fake Heap Pointer*: To measure the impact of the 4 bytes of padding in the *Heap Pointer* implementation, we padded the object header by 4 bytes but did not store the heap ID in them. Instead, we determine the heap ID as in the *No Heap Pointer* version. In other words, we impose the memory overhead of the *Heap Pointer* version but do not take advantage of its faster write barrier implementation.

The KaffeOS JIT compiler does not inline the write barrier routine. Inlining the write barrier code might improve performance, but it could also lead to substantial code expansion. For instance, in the *No Heap Pointer* version, the

write barrier code includes 29 instructions, plus an additional 16 instructions for procedure calling conventions, which an inlined version may not require. Ideally, the intermediate representation of the write barrier code should be available to a just-in-time compiler, so that the compiler can apply heuristics to decide whether to inline or outline on a per-call site basis.

5.1.1 Microbenchmarks

We devised microbenchmarks to determine the best-case overhead of performing a write to a heap location. The microbenchmarks assign a reference to a location in a tight loop using each of the `PUTFIELD`, `PUTSTATIC`, and `AASTORE` opcodes. We used the Pentium cycle counter register to measure the number of cycles spent. We subtracted the loop overhead, which we determined by counting the number of cycles spent in an empty loop.

Table 5.1 shows the results. The overhead for the *No Heap Pointer* version is roughly 43 cycles per write. The optimization used by the *Heap Pointer* version substantially reduces this overhead; in the case of the `PUTFIELD` instruction, the reduction is from 43 to 11 cycles. `PUTFIELD` and `PUTSTATIC` take roughly the same number of cycles for all versions. The `AASTORE` instruction has a higher base cost, but its write barrier overhead is only slightly larger. As expected, the *Fake Heap Pointer* version and the *No Heap Pointer* take roughly the same number of cycles for all three instructions.

Table 5.1. Best-case overhead per write barrier by type.

Version	PUTFIELD		PUTSTATIC		AASTORE	
(all numbers are in cycles)	Cost	Overhead	Cost	Overhead	Cost	Overhead
Base cost (No Write Barrier)	2		2		65.4	
No Heap Pointer (Default)	45.0	+43.0	43.7	+41.7	110.2	+44.8
Heap Pointer	13.0	+11.0	13.0	+11.0	79.0	+13.6
Fake Heap Pointer	44.4	+42.4	44.4	+42.4	111.9	+46.5

These cycle numbers represent the best case, a tight loop executing with a hot cache.

5.1.2 Application Benchmarks

We used the SPEC JVM98 benchmarks [72] to evaluate the performance of KaffeOS for real-world, medium-sized applications. These benchmarks were released in 1998 by the System Performance Evaluation Corporation (SPEC), an organization of hardware vendors whose goal is to provide a standardized set of relevant and application-oriented benchmarks to evaluate the efficiency of JVM implementations. The benchmarks measure the efficiency of the just-in-time compiler, runtime system, operating system, and hardware platform combined. They were chosen based on a number of criteria, such as flat execution profile (no tiny loops), repeatability, and varying heap usage and allocation rate.

SPEC JVM98 consists of seven benchmarks: compress, jess, db, javac, mpegaudio, mtrt, and jack. All except db are real-life applications that were developed for purposes other than benchmarking. An eighth benchmark, check, is used to test the proper implementation of Java's features in the VM; for example, check tests that the JVM performs array bound checks. Our KaffeOS implementation passes the check benchmark. We briefly discuss each benchmark:

- **compress:** This application compresses and decompresses data from a set of five tar files using a modified Lempel-Ziv method (LZW). Each file is read in, compressed, and written to memory. The result is read and uncompressed and the size is double-checked against the original size. Compress is not a typical object-oriented application in that it does not use many objects; it spends most of its execution operating on two large byte arrays that are allocated for its input and output.

- **jess:** This benchmark is a Java Expert Shell System (JESS) that is based on NASA's CLIPS expert shell system. An expert shell system continuously applies a set of rules to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. This benchmark is computationally intensive and allocates many objects, though most are short lived.

- **db:** This synthetic benchmark simulates database operations on a memory-resident database. It reads in a 1 MB file that contains records with names, addresses and phone numbers. It then performs a stream of operations on the records in the file, such as adding and deleting an address or finding and sorting addresses.

- **javac:** This application is the Java compiler from Sun Microsystem's Java Development Kit (JDK) version 1.0.2. Its workload is the compilation of the jess benchmark, which is several thousand lines of Java code.

- **mpegcompress:** This program decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. The workload consists of about 4 MB of audio data. This benchmark performs very little memory allocation and makes frequent use of floating point operations.

- **mtrt:** This raytracer program renders a small scene depicting a dinosaur. It is the only benchmark that is multithreaded; the scene is divided between two threads.

- **jack:** This parser generator is based on the Purdue Compiler Construction Tool Set (PCCTS), now known as JavaCC. The workload consists of a file that contains instructions for the generation of jack itself.

In addition to the four different write-barrier implementations mentioned above, we include in our comparison the version of Kaffe on which KaffeOS is based. We use a development snapshot from June 2000 for that purpose; we label this version "Kaffe 2000" in our benchmarks. We also include IBM's JVM from the IBM JDK 1.1.8, which provides one of the fastest commercial JIT compilers [73] available for JVMs that implement Java version 1.1.x. Our results are not comparable with any published SPEC JVM98 metrics, as our measurements are not compliant with all of SPEC's run rules. Complying with the run rules would have required support for the abstract windowing toolkit (AWT), which our prototype does not provide.

We instrumented KaffeOS to determine the number of write barriers that are executed in a single run of each benchmark. Except in the *No Write Barrier* case, we run each benchmark in its own process. Table 5.2 shows how many write barriers are executed for a single run. Except for jack, these numbers confirm the numbers presented in Table 1 of an independent study by Dieckmann that investigated the allocation behavior of the SPECjvm98 benchmarks [27]. We note that the total number of writes includes not only the benchmark program but also includes writes executed in methods that belong to the runtime libraries. We found that when the implementation of certain runtime classes, such as `java.util.Hashtable`, changes, the number of writes can change substantially. For instance, when we updated our libraries from an earlier version of KaffeOS to the version used to create Table 5.2, we observed a decrease from about 33.0 M to 30.1 M for the db benchmark, and an increase from about 15.5 M to 20.8 M for javac. The lesson is that the number of write barriers depends not only on the application but also on implementation decisions in the runtime libraries.

Table 5.2 also shows that for almost all write barrier executions counted, the source and destination objects lie in the same heap, which confirms our expectations. Only a miniscule fraction of write barriers (less than 0.2%, except for

Table 5.2. Number of write barriers executed by SPEC JVM98 benchmarks.

Benchmark	Barriers	Cross-heap	
		Number	Percent
compress	47,969	3336	6.95%
jess	7,939,746	4747	0.06%
db	30,089,183	3601	0.01%
javac	20,777,544	5793	0.03%
mpegaudio	5,514,741	3575	0.06%
mtrt	3,065,292	3746	0.12%
jack	19,905,318	6781	0.03%

"Cross-heap" counts those occurrences where source and destination do not lie in the same heap, i.e., those for which entry and exit items must be created or updated.

compress, which does hardly any writes) are "cross-heap," in that they result in the creation or update of entry and exit items. The other possible outcome produced by a write barrier, a segmentation violation error, should not and does not occur in these benchmarks. Differences in the number of "cross-heap" write barriers result when applications use different kernel services. For instance, javac opens many more files than compress.

We also examined the distribution of the instructions that trigger write barriers. The results are shown in Table 5.3. This distribution is again very application-dependent. For instance, db makes significant use of the `java.util.Vector` class, which explains its high number of `AASTORE` instructions. Assignments to static fields are typically rare, except for javac. A small number of write barriers, labeled as "other," are triggered inside the VM or in native library code. Since the best-case overheads measured in Section 5.1.1 are roughly about the same for the different bytecode instructions that trigger write barriers, the distribution turns out not to be very important.

Figure 5.1 shows the results of running the SPECjvm98 benchmarks. We ran each benchmark three times in the test harness provided by SPEC (in a run that

Table 5.3. Type of write barriers executed SPEC JVM98 benchmarks.

Benchmark	Barriers	By Type			
		PUTFIELD	AASTORE	PUTSTATIC	other
compress	47,969	37,326	7,199	149	3,295
jess	7,939,746	4,037,788	3,897,103	153	4,702
db	30,089,183	3,119,214	26,966,222	188	3,559
javac	20,777,544	16,885,470	3,256,128	630,194	5,752
mpegaudio	5,514,741	5,502,039	8,986	182	3,534
mtrt	3,065,292	1,300,487	1,760,914	190	3,701
jack	19,905,318	16,368,829	3,529,429	320	6,740

The "By Type" columns list the different bytecode instructions that trigger write barriers. The type "other" refers to assignments inside of native libraries, or within the JVM itself. The distribution is very application-dependent.

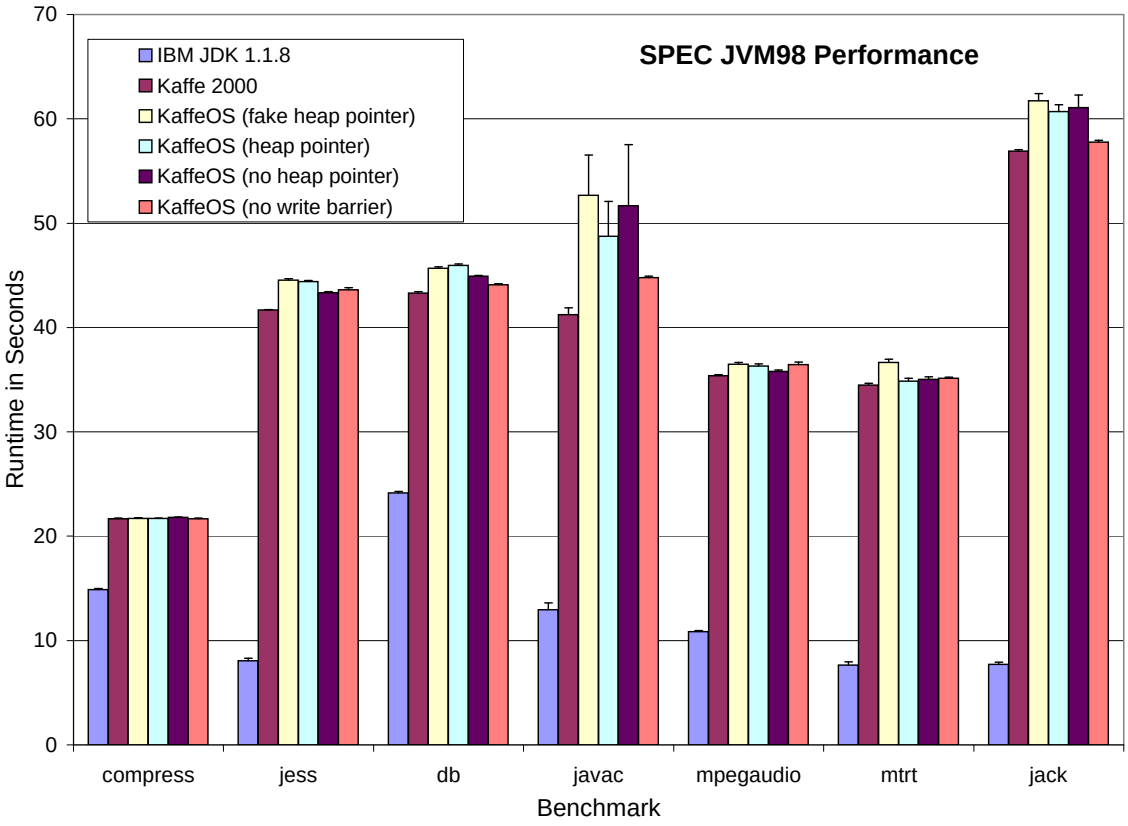


Figure 5.1. KaffeOS performance in SPECjvm98 benchmarks. This chart displays the time needed to run SPECjvm98 using the provided harness. Kaffe 2000 is the version of the Kaffe VM upon which KaffeOS is based. All Kaffe-based versions are roughly two to eight times slower than IBM's industrial VM. The differences between Kaffe 2000 and KaffeOS are small, and the overheads of all write barrier versions with respect to the *No Write Barrier* version are tolerable. The error bars shown reflect the 95% confidence interval.

follows all of SPEC run rules, each benchmark is run two to four times). We set the maximum heap size to 64 MB for each JVM. The bars show the average runtime as displayed by the test harness, and the error bars are used to display an estimate of the 95% confidence interval. Because of the small number of runs, the interval does not provide a good estimate, but it demonstrates that the variance between runs is small for most benchmarks, except for javac. Such variance can result from different causes. For instance, initialization work in system classes has to be done

only once during the first run. In addition, a conservative collector such as the one used in Kaffe and KaffeOS might hold on to floating garbage only to collect it later, which stretches the time required for a subsequent run. The SPEC benchmarks are intended to capture these effects, which is why their run rules demand that the JVM is not restarted after each run.

Overall, IBM's JVM is between two to eight times faster than Kaffe 2000; we will focus on the differences between Kaffe 2000 and the different versions of KaffeOS. For those benchmarks that create little garbage, compress and mpegaudio, the difference in total run time is small. For the other benchmarks, we observe a larger difference, even between Kaffe 2000 and the *No Write Barrier* version of KaffeOS. This difference is in part because of the changes we made to the runtime system and in part because the time spent in garbage collection differs, as is shown in Figure 5.2.

The time spent during garbage collection depends on the initial and maximum heap sizes, the allocation frequency, and the strategy used to decide when to collect. For instance, we found that IBM's performance varies widely depending on its heap management: if we do not specify a maximum heap size of 64 MB, it will aggressively try to keep its heap very small (less than 5 MB), which increases the time to run the jess benchmark from 8 to 24 seconds. IBM's collector is precise and copying, which means that its runtime is asymptotically proportional to the size of the live heap. jess allocates many objects, but very few of them are long-lived [27], which can explain this large variation in run time. This dependency on the garbage collection strategy limits our ability to compare the run-time overhead of JVMs that use different strategies.

Kaffe 2000 and KaffeOS use a simple strategy: a collection is triggered whenever newly allocated memory exceeds 125% of the memory in use at the last GC. However, whereas Kaffe 2000 uses the memory occupied by objects as its measure, KaffeOS uses the number of pages as its measure, because KaffeOS's accounting mechanisms are designed to take internal fragmentation into account. In addition, KaffeOS's decision mechanism is applied to each heap independently.

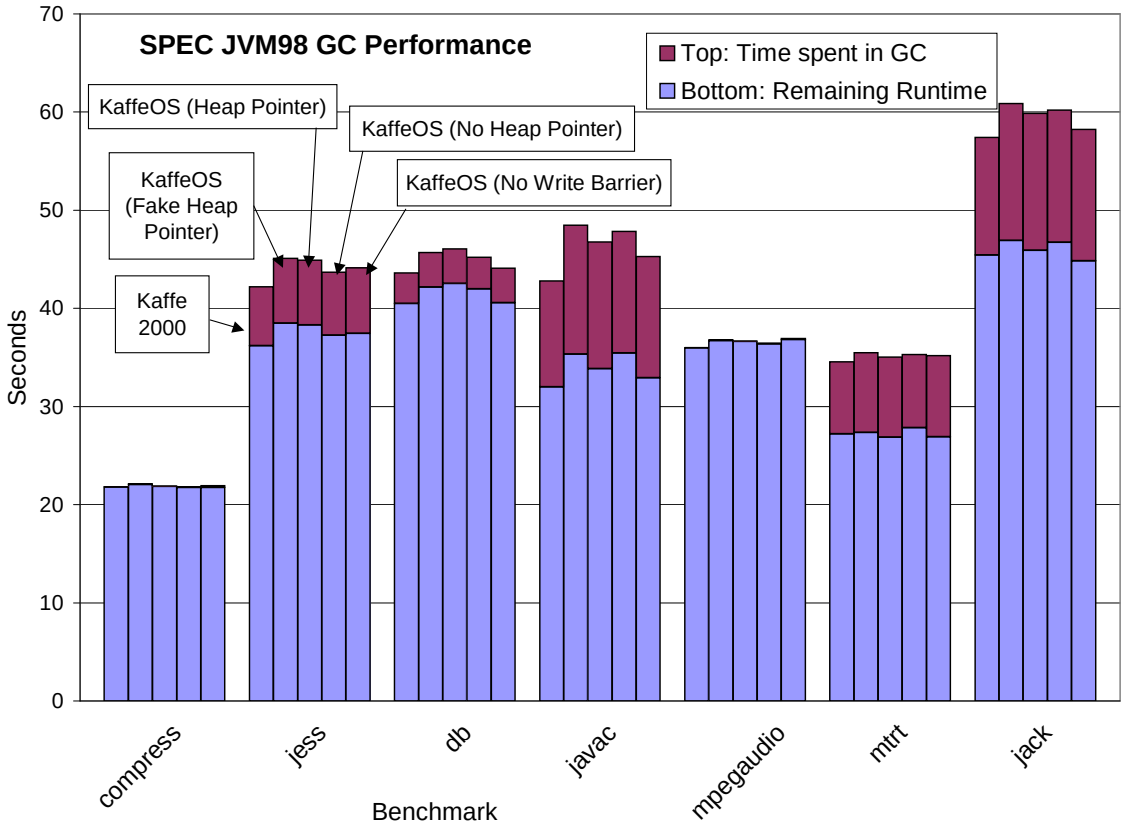


Figure 5.2. Time spent in GC for SPECjvm98 benchmarks. This chart displays the time needed to run a single SPECjvm98 spec benchmark from start to exit. The time spent in garbage collection is displayed in the top part of each bar. The heap pointer optimization is effective for some applications, but not for others.

To reduce the impact of different GC strategies on our estimate of the write barrier penalty for the SPEC benchmarks, we instrumented Kaffe 2000 and KaffeOS to use the Pentium cycle counter to measure how much time is spent during GC. Figure 5.2 compares the results of these experiments. Each group of bars corresponds to Kaffe 2000, KaffeOS with a fake heap pointer, KaffeOS with heap pointer, KaffeOS with no heap pointer, and KaffeOS with no write barrier, in that order. The full bar displays the time between the start and the exit of the JVM, which includes time spent in the SPEC harness. These results are for a single run

only. The upper part of the bar shows the time spent in garbage collection. The lower part of the bar represents the time not spent in garbage collection, which is computed as the difference between the total execution time and the time spent in GC. In this figure, the differences between Kaffe 2000 and KaffeOS *No Write Barrier* are very small, which suggests that we do not pay a runtime penalty for the changes we made to make the runtime libraries process-aware (as described in Section 4.2).

Table 5.4 compares the measured overhead to the overhead that could be expected from the write barriers alone, assuming the best-case cycle counts. The best-case overhead was obtained by multiplying the vector of cycles measured in Table 5.1 with the distribution vector in Table 5.3 for each benchmark. The worst overhead measured is 5.6% with respect to runtime excluding GC time; 7.7% with respect to the total runtime without the heap pointer optimization; and 4.5% and 4.8%, respectively, if the optimization is applied. For two benchmarks that perform a substantial number of writes, java and jack, the actual penalty is predictably larger than the estimate obtained using a hot cache. For these two benchmarks, the *heap pointer* optimization is effective in reducing the write barrier penalty.

Table 5.4. Measured vs. best-case overhead.

Benchmark	Total runtime		Runtime excluding GC time		Total
	No Heap Ptr	*Heap Ptr*	*No Heap Ptr*	*Heap Ptr*	IBM**
compress*	-0.5%(0.0%)	0.0%(0.0%)	0.1%(0.0%)	0.5%(0.0%)	0.7%
jess	-1.1%(1.0%)	1.7%(0.3%)	-0.5%(1.2%)	2.3%(0.3%)	10.5%
db	2.5%(3.8%)	4.5%(1.1%)	3.5%(4.1%)	4.8%(1.2%)	8.1%
javac	5.6%(2.5%)	3.2%(0.7%)	7.7%(3.4%)	2.9%(0.9%)	7.3%
mpegaudio*	-1.4%(0.8%)	-0.7%(0.2%)	-1.2%(0.8%)	-0.5%(0.2%)	-1.7%
mtrt*	0.3%(0.5%)	-0.4%(0.1%)	3.5%(0.6%)	-0.1%(0.2%)	-0.2%
jack	3.4%(1.9%)	2.8%(0.5%)	4.2%(2.4%)	2.4%(0.6%)	14.1%

* small number of write barriers. ** projected maximum overhead
In each table cell, the first number is the measured overhead; the number in parentheses is the best-case overhead relative to the *No Write Barrier* version, which was obtained by multiplying the vector of cycles measured in Table 5.1 with the distribution in Table 5.3.

Excluding GC, KaffeOS *Fake Heap Pointer* performs similarly to KaffeOS *No Heap Pointer*; however, its overall performance is lower because more time is spent during GC.

There are some anomalies: jess runs faster with write barriers than without, and the overhead of db is lower than the expected best-case overhead. The relative performance order is not consistent. Without nonintrusive profiling, which we do not have available for Kaffe, we can only speculate as to what the reasons might be. It is possible that cache effects are to blame, since both versions have completely different memory allocation patterns. Some internal data structures, such as the table of interned strings, are instantiated only once in the *No Write Barrier* cases, whereas they are duplicated for each heap in all other cases. Finally, since the garbage collector collects only the user heap, we may see a small generational effect, since fewer objects need to be walked during each process's GC.

On a better system with a more effective JIT, the relative cost of using write barriers would increase. The last column in Table 5.4 shows a projection for IBM's JDK. We used the run-time overhead (without garbage collection) and added it to IBM's run time and computed the overhead as a percentage of the total run time. The projected overhead in this case lies between 7% and 14% for these benchmarks with a large number of writes. This projection, combined with our numbers for the frequency of writes in the SPEC JVM98 benchmarks suggest that even if the relative cost should grow slightly, the absolute cost would very likely be tolerable.

In addition, a good JIT compiler could perform several optimizations to remove write barriers. A compiler should be able to remove redundant write barriers, along the lines of array bounds checking elimination. It could even perform method splitting to specialize methods and thus remove useless barriers along frequently used call paths. If a generational collector were used, we might also be able to combine the write barrier code for cross-heap checking with the write barrier code for cross-generation checking.

5.2 Overhead for Thread Stack Scanning

As outlined in Section 3.2.2.3, each KaffeOS process must scan all threads when garbage collecting its heap. In this section, we show that this requirement does not lead to priority inversion between processes. It can, however, impose significant overhead for each process, which we quantify for our current prototype.

Priority inversion could occur if one process prevented a second process from running because it needed to stop the second process's threads to scan them. As shown in Figure 4.7, we stop remote threads on three occasions during the mark phase of a heap's garbage collection, which correspond to the three code sections labeled "Part 1," "Part 2," and "Part 3." These sections correspond to marking entry items, scanning remote thread's stacks, and recoloring and rewalking the list of black items. The potential for priority inversion depends on the maximum amount of time remote threads must be stopped. This maximum is the maximum of the amounts of time spent in Part 1 and Part 3, combined with the maximum amount of time spent on scanning any individual thread in Part 2. The overall maximum, however, does not depend on the number of remote threads.

All of the aforementioned times are application-specific. In particular, the amount of time spent in Part 1 depends on how many entry items exist in a given heap. The time spent in Part 2 on each remote thread depends on the size of the thread stack and on how many remote references are found on a stack. For a downward-growing stack, the size of the thread stack is measured from the upper end of the allocated stack area to the lowest used address in this area; the size depends on the number and size of activation records a thread has allocated on it. Our implementation limits a thread stack's size to some maximum value, which is a configurable parameter that is set to 64 KB by default. A `StackOverflowError` is thrown when a thread attempts to use more stack space.

To measure the number of cycles spent in Parts 1 to 3, we started one process that constantly collects garbage on its own heap. In parallel with that process, we run the applications of the SPEC JVM98 benchmarks one after another in different processes. Table 5.5 shows the results: the variance is relatively small, and the

Table 5.5. Cycles spent on walking entry items for SPEC JVM98 Benchmarks

	Walking of entry items (Part 1)	Rewalking of marked objects (Part 3)
Number	27,123	27,123
Mean	383,071	291,348
Maximum	500,405	495,196
Standard Deviation	34,569	24,244

average amount of cycles spent in Part 1 is 383,000 cycles and in Part 3 is 291,000 cycles. On the 800 MHz machine on which we ran this experiment, this amounts to about 4.7% and 3.6% of one time slice of 10 ms.

To measure the number of cycles spent scanning stacks in Part 2, we used two scenarios. In addition to the SPEC JVM98 benchmarks, we crafted a benchmark in which a single thread executes a recursive function, pausing for 1 second every 100 steps. The execution of this function results in a monotonically increasing stack size.

Figure 5.3 shows both measurements. The top graph shows the recursion benchmark. For a maximum size of 221,000 bytes, the time it takes to scan the thread is about 3,100,000 cycles, which is about 39% of a time slice on our 800 MHz machine. This number of cycles corresponds to a maximum delay of about 4.8 ms. The bottom graph shows the distribution for the applications of the SPEC JVM98 benchmarks. The threads of these applications typically have a much smaller stack size than in the recursion benchmark. In the vast majority of cases, less than four pages per thread must be scanned.

The correlation between stack size and cycles to scan the stack is linear, as could be expected. For the SPEC JVM98 applications, the estimated slope ranges between 23 and 28 cycles per byte. The trend lines shown in Figure 5.3 do not capture the range of sizes between 8,000 and 10,000 bytes as well as they capture the range between 400 and 6,000 bytes, because there are many data points for smaller sizes. The slope for the SPEC JVM98 applications appears to be higher

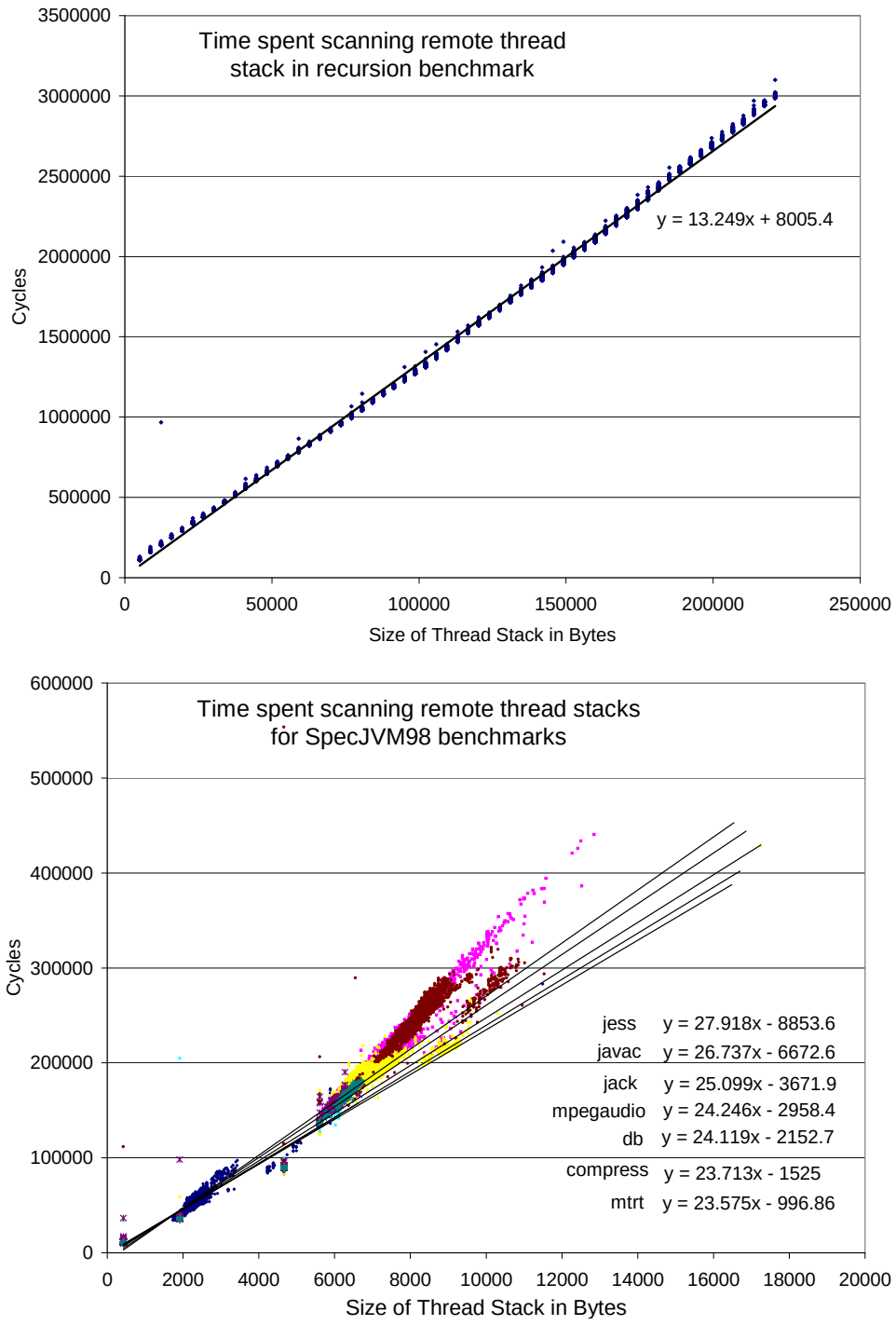


Figure 5.3. Time spent scanning remote thread stacks. These figures show the linear dependency of the time it takes to scan a stack to the size of the stack area. The top figure shows a synthetic benchmark that scans a single thread engaged in successive recursive invocations. The bottom figure shows the time required to scan each individual thread that is run during the executions of the SPEC JVM98 benchmarks.

than the slope for the synthetic recursion benchmark. Basing our estimate on the SPEC benchmarks, we can give an estimate for the expected overhead. For n threads with stack sizes $s_i, i = 1..n$, we expect an overhead of roughly $\sum_{i=1}^{n} 28 * s_i$ cycles. As an example, for 100 remote thread stacks of 8 kilobytes average size, we could expect about 23 * 10^6 cycles or about 29 ms per garbage collection. This overhead is substantial, which is why limits on the number of threads must be applied. In addition, there are several possible optimizations that are discussed in Section 3.2.2.3 that could reduce the $O(mn)$ complexity for m processes and n threads exhibited by our current implementation.

5.3 Denial of Service Scenarios

We evaluate KaffeOS's ability to prevent denial-of-service attacks against memory and CPU time using a Java servlet engine. A Java servlet engine provides an environment for running Java programs (servlets) at a server. Servlets' functionality subsume that of CGI scripts at Web servers: for example, servlets can create dynamic content, perform computations, or run database queries. A servlet engine typically supports multiple, different servlets that provide web services to different clients.

Our first goal is to ensure that in situations in which one of these servlets misbehaves or fails, the service provided by the rest of them is unaffected. This property depends on the ability of KaffeOS to isolate applications, and showing this property demonstrates the effectiveness of KaffeOS's isolation mechanisms. A second goal is to show that KaffeOS's approach of running multiple applications in one JVM allows us to support more servlets than the approach that relies on underlying OS support for isolation.

We used the off-the-shelf Apache 1.3.12 web server, its JServ 1.1 servlet engine [48], and Sun's JSDK 2.0 release of the Java servlet extensions to run our tests. The setup is shown on the left-hand side of Figure 5.4. We modified a version of the Apache benchmark program *ab* to simulate multiple clients on a single machine. Each client issues HTTP GET requests for a specified URL on the server. On the

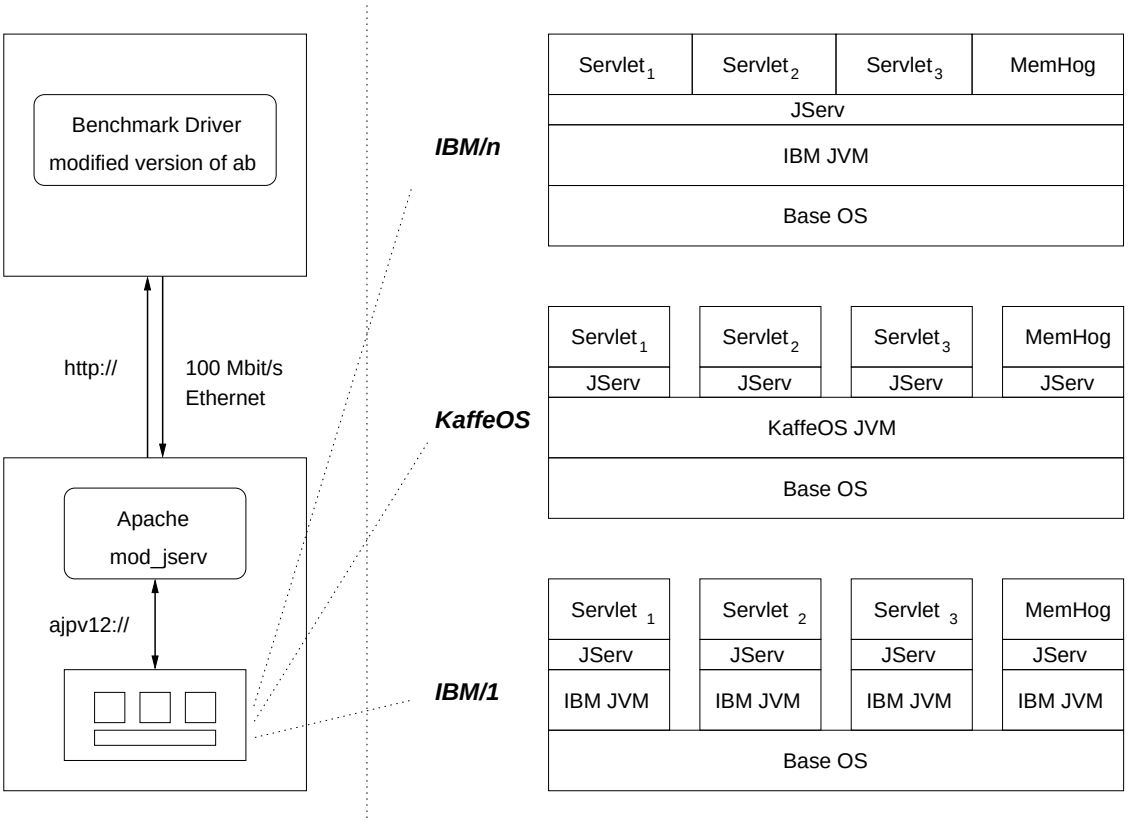


Figure 5.4. Configurations for MemHog. The left-hand side shows our experimental setup: a modified version of Apache's benchmark program *ab* runs on a client machine that is connected via 100 MBit/s Ethernet to a server. The server hosts both the Apache server with the *mod_jserv* module and the JServ engines in which the servlets run. The configurations are displayed on the right-hand side: *IBM/n* runs n servlets per JVM; *IBM/1* runs 1 servlet per JVM and uses n JVMs; and *KaffeOS* runs each servlet in its own KaffeOS process.

server machine, the Apache server processes and forwards these requests to one or more JServ servlet engines. An Apache module (*mod_jserv*) communicates with the servlet engines using a custom protocol called ajpv12.

Each JServ instance can host one or more *servlet zones*, which are virtual servers. Although a servlet zone can host multiple, different servlets, for simplicity each servlet zone hosts exactly one servlet in our experiment. A URL is mapped to each servlet; the servlet is loaded when a user issues the first request for its corresponding

URL. Servlets are kept in memory for subsequent requests; if no requests arrive for a certain time period, they are unloaded and garbage collected.

The right-hand side of Figure 5.4 shows the different configurations we compared:

- **IBM/n**: We run multiple servlets in one servlet engine; the servlet engine runs in a single instance of the IBM JVM.
- **KaffeOS**: We run each servlet in a separate engine; each servlet engine runs in its own process on top of KaffeOS.
- **IBM/1**: We run each servlet in a separate engine; each engine runs in a separate instance of the IBM JVM in a separate process on top of the Linux operating system.

We consider three scenarios that involve different kinds of denial-of-service attacks. In the first scenario, we attempt to deny memory to other servlets by running a servlet that allocates large amounts of memory until it exceeds its limit and is terminated. We use this scenario for a second purpose, namely, to experimentally verify that KaffeOS can terminate such applications safely. Finally, in the second and third scenario, we examine attacks against CPU time and the garbage collector.

5.3.1 MemHog Scenario

In this scenario, we use small "Hello, World" servlets as examples of well-behaved servlets that provide a useful service. Alongside these well-behaved servlets, we run a "MemHog" servlet that attempts to deny memory to them. This servlet, when activated, spawns a thread that enters a loop in which it repeatedly allocates objects. The objects are kept alive by linking them in a singly linked list. In the IBM/1 and IBM/n configurations, allocations will fail once the JVM's underlying OS process reaches its maximum heap size. In the KaffeOS configuration, allocations will fail when the process reaches its set limit. In all three configurations, an

`OutOfMemory` exception is eventually thrown in the thread that attempts the first allocation that cannot succeed.

Because the JServ engine is a heavily multithreaded Java program and because Java programs continually allocate objects, the `OutOfMemory` exception can occur at seemingly random places and not only in the thread spawned by the MemHog servlet. By examining backtraces we observed that these exceptions occur also at inopportune places. In particular, a thread can run out of memory in the code that manipulates data structures that are shared between servlets in the surrounding JServ environment, or it can run out of memory in code that manipulates data structures that are kept over successive activations of one servlet. Eventually, these data structures become corrupted, which results in an unhandled exception in one or more threads. In the KaffeOS scenario, this event causes the underlying KaffeOS process to terminate. In the IBM/1 and IBM/n scenarios, the underlying JVM terminates. In some instances, we observed the IBM JVM crash, which resulted in segmentation violations at the OS process level.

When simulating this denial-of-service attack, we did what a system administrator concerned with availability of their services would do: we restarted the JVMs and the KaffeOS processes, respectively, whenever they crashed because of the effects caused by a MemHog servlet.

We counted the number of successful responses our clients received from the "Hello, World" servlets during a certain period of time (30 seconds). We consider this number an effective measure for the amount of service the system provides, because it accounts for the effects caused by the denial-of-service attack on the Apache/JServ system as a whole.

For each configuration, we measured the amount of service provided with and without a denial-of-service attack. We ran the experiment with 2, 4, 8, 16, 32, 40, 48, 56, 64, 72, and 80 servlets. We allowed up to 16 concurrent connections from the client; increasing that number further did not increase throughput. The client rotated these connections among all servlets, so that all servlets were activated shortly after the start of the experiment and remained active throughout the

experiment.

Figure 5.5 shows the results. This figure shows that the KaffeOS configuration, as well as the IBM/1 configuration, can successfully defend against this denial-of-service attack, because the impact of the single MemHog servlet is isolated. However, the graph shows that running each of the servlets in a single JVM, as done in the IBM/1 approach, does not scale. We estimate that each IBM JVM process takes about 2 MB of virtual memory upon startup. Starting multiple JVMs eventually causes the machine to thrash. An attempt to execute 100 IBM JVMs

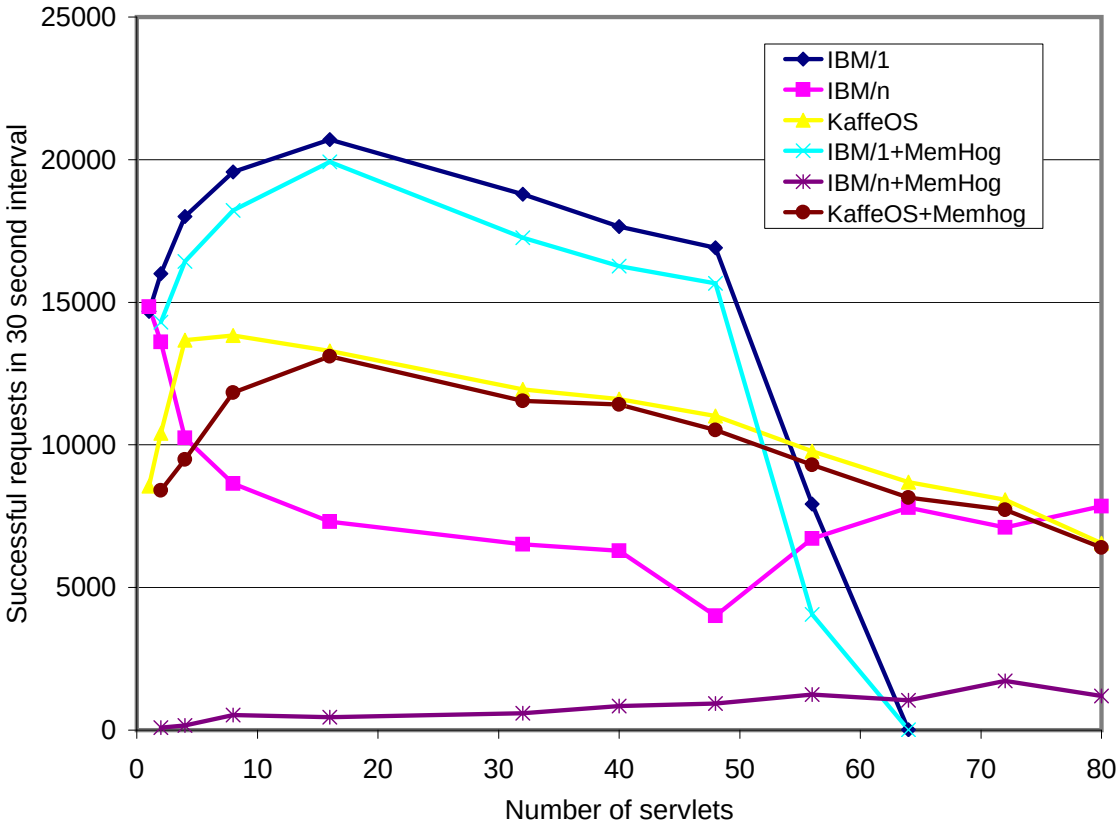


Figure 5.5. Throughput for MemHog servlet. This chart displays the number of successful requests received in 30 seconds. *IBM/1* is successful at isolating the MemHog servlet, whereas *IBM/n* is not. *IBM/n* and KaffeOS can support more than 64 servlets, whereas *IBM/1* cannot.

running the Apache/JServ engine rendered the machine inoperable.

The IBM/n configuration, on the other hand, can easily support 80 servlets. However, if the MemHog servlet is added, this configuration exhibits a severe decrease in performance. This degradation is caused by a lack of isolation between servlets. As the ratio of well-behaved servlets to malicious servlets increases, the scheduler yields less often to the malicious servlet. Consequently, the service of the IBM/n,MemHog configuration shown in Figure 5.5 improves as the number of servlets increases. This effect is an artifact of our experimental setup and cannot be reasonably used to defend against denial-of-service attacks.

In addition to being able to defend against the denial-of-service attack, KaffeOS can support as many servlets as IBM/n, but its performance does not scale very well. We identified two likely sources for the behavior. The first source are inefficencies in the signal-based I/O mechanisms used in our prototype's user-level threading system. The second source is the deficiencies in KaffeOS's memory allocation subsystem, which we discussed in Section 4.3. Specifically, we found that increased external fragmentation slightly increased the garbage collection frequency of individual processes, because it became harder for them to expand their heap even though they had not reached their limits. As discussed earlier, a moving collector should be able to alleviate this limitation.

We conclude from these experiments that KaffeOS's approach of supporting multiple applications in a single JVM can effectively thwart denial-of-service attacks directed against memory. The operating system-based approach (IBM/1) has the same ability, but its scalability is restricted when compared to KaffeOS.

5.3.2 MemHog Stress Test

One of KaffeOS's design goals is the ability to safely terminate an application without affecting the integrity of the system and to fully reclaim an application's resources when that application is killed. We claimed in Section 3.1.3 that the introduction of a user/kernel boundary is crucial to achieving safe termination. The MemHog scenario can be used to verify this claim. We developed a stress test in which we repeatedly activate a single MemHog servlet and kill it once it exceeds

its memory limit.

We use a debugging version of KaffeOS that includes a series of internal integrity checks for this test. For instance, we check the consistency of the internal class table, the consistency of the primitive block table, and the consistency of the heap free lists frequently at various points in the JVM code. If any inconsistency is detected during one of these tests, we abort the JVM. After each kill of a MemHog servlet, we record the overall number of bytes allocated by all heaps, as well as

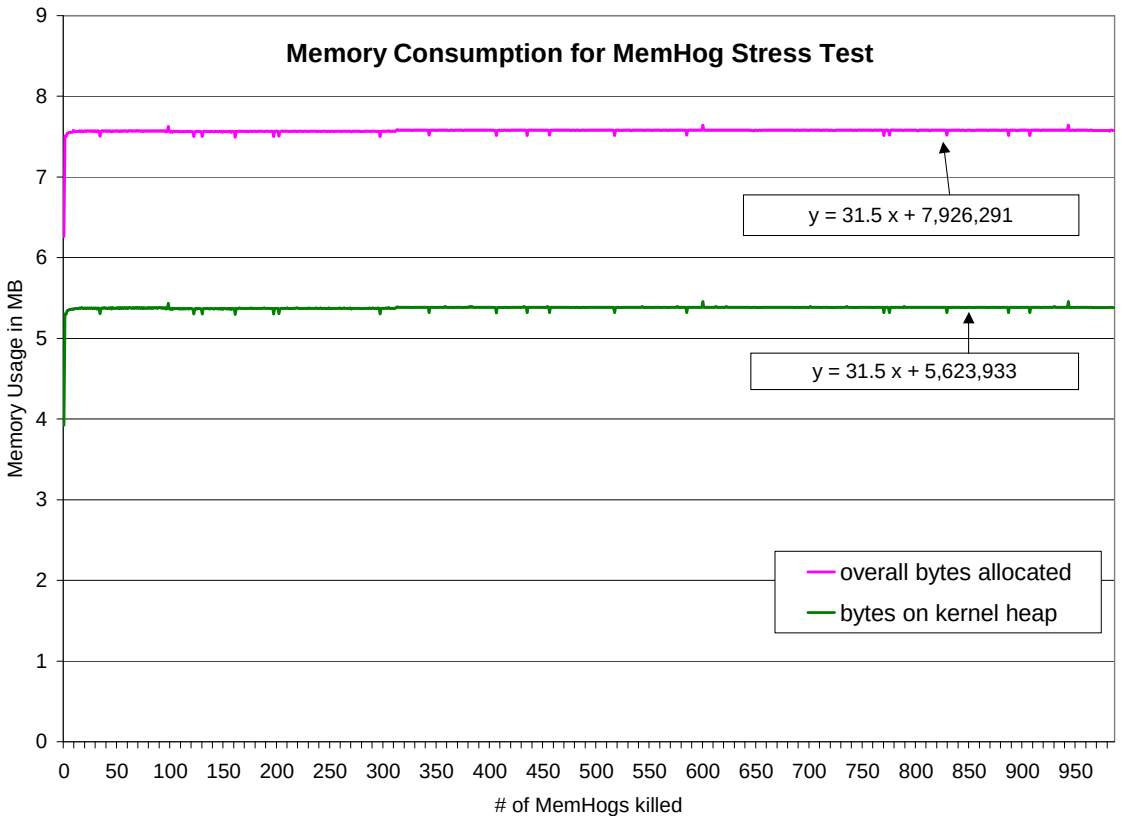


Figure 5.6. MemHog stress test. In this test, our driver repeatedly starts a JServ engine, which is killed by a MemHog servlet. We plot the total number of bytes allocated by all heaps of the system, and the number of bytes in live objects on the kernel heap. Both lines are nearly parallel to the x-axis, which implies that KaffeOS reclaims the memory used by a killed process almost in full. Linear regression analysis shows a leakage of 31.5 bytes per killed process.

the number of bytes in objects on the kernel heap. Figure 5.6 shows the results for about one thousand kills. We found that both byte numbers remained almost constant over time—linear regression analysis revealed a leakage of 31.5 bytes per kill, which indicates that KaffeOS can reclaim virtually all of the memory used by the MemHog servlet. The rise between MemHog #0 and #1, which can be seen at the very left of the graph, is due to an increase in the size of the kernel heap. This expansion occurs when the JServ servlet engine running the MemHog servlet causes the lazy loading of shared system classes that the driver process did not need.

The MemHog stress test demonstrates full reclamation when termination is caused by the violation of resource limits. The other possible case is explicitly requested termination. It does not have to be considered separately, because the circumstances under which both cases occur are sufficiently similar. In KaffeOS, an application can be killed explicitly by invoking the `ProcessHandle.kill()` method on its handle, which is equivalent to issuing a `kill(2)` system call in Unix. The reason for this similarity is that the MemHog servlet spawns a thread that allocates memory on each request. When the application is killed because it reaches its limit, several threads will typically run. As we found during debugging, any thread can reach the memory limit first. This thread then executes the code to kill the entire application, which includes destroying the other threads that are running—which is just what would happen if an outside process issued a `ProcessHandle.kill()` on the MemHog's process handle. During KaffeOS's development, we empirically observed that the destroy requests arrive so randomly that they are almost certain to cause corruption if we erroneously manipulate critical data structures in user mode. When we found such bugs, we moved the corresponding code parts into the kernel, which prevented the corruption.

We cannot claim that our prototype has the robustness and maturity of an industrial product. MemHog will crash it at some point (between 1000 and 5000 kills), because of race conditions, overflows, and other unhandled situations or bugs in our prototype. However, when compared to *IBM/n*, which typically survives only

a few MemHogs throwing `OutOfMemory` exceptions, this test provides strong evidence that the introduction of a user/kernel boundary is useful for the construction of a system that is robust enough to safely terminate ill-behaved applications.

5.3.3 CpuHog Scenario

A denial-of-service attack against CPU time is harder to detect than an attack against memory, because it can be difficult to determine whether a servlet's use of the CPU provides a useful service or is merely wasting CPU time. This dissertation does not aim to provide a general solution to this problem. Instead, we show how our CPU management mechanisms can be used to ensure that a servlet's clients obtain the share of service that is provisioned for them, even in situations in which another servlet attempts to use more than its share.

We adopt the view that using the CPU when it would otherwise be idle is harmless. In systems that bill users for CPU time used, this view would not be appropriate. We consider a successful denial-of-service attack against the resource CPU a situation in which an important task cannot get done or is being done much slower, because the CPU is used by some other, less important task. Our scheduling scheme cannot limit the amount of CPU time a process consumes: it can only guarantee a certain amount to processes that are runnable.

We developed an MD5-Servlet as an example of a servlet that performs a computationally intensive task. MD5 is a one-way hash function that is used to digitally sign documents [65]; hence, this servlet could be seen as representative of an application that provides notary services over the World Wide Web. For each request, the servlet computes the MD5 hash function over the first 50,000 words of the `/usr/dict/word` dictionary file. As a measure of service, we use the number of successfully completed requests per second. Our modified version of *ab* estimates this number of requests per second simply by inverting the time difference between consecutive responses. In addition to the MD5 servlet, we used a "CPUHog" servlet that executes an infinite loop in which it does not voluntarily yield the CPU to other threads.

For the purposes of this experiment, we added a small scripting mechanism to

the servlet driver that spawns JServ servlet engines for the different servlet zones. A script sets up a possible scenario in which a hog is activated and in which CPU shares must be adjusted so as to thwart a possible denial-of-service attack. Each entry in a script specifies a set of CPU shares that are assigned to the servlet engines spawned by the driver.

Table 5.6 shows one such script for three MD5 servlets (A, B, and C) and one CPUHog. Initially, all servlets start out with equal shares of 1/4 each. Because the fourth servlet zone has no servlet running, we expect the three servlets to split the excess CPU time among them, which will give each MD5 servlet $1/4 + 1/4/3 = 1/3$ of the CPU. After a few seconds have passed, we manually activate the CPUHog servlet by issuing a request for its associated URL. Once the CPUHog servlet is activated, it immediately uses its full share, which eliminates the excess CPU time of 1/4 that was previously shared among A, B, and C. At this point, the Hog is limited to a 1/4 share of the CPU. In our script, we assume that some mechanism detects that this CPU consumption by the Hog is undesirable and that the CPU should go entirely to the MD5 servlets doing useful work. To achieve this goal, the script instructs the driver to set the share of the Hog to zero, after which the three MD5 servlets should again split the CPU evenly at 1/3 each. Finally, let us assume that servlet C's throughput should be increased to 1/2 of the server's capacity (as would be the case if it were decided that C provide some kind of premium service for a selected group of users). Servlet C's share is increased to 1/2, and A and B's

Table 5.6. Expected results for CpuHog and Garbagehog scenarios.

Timeline	Assigned Shares				Service Share Obtained			
	A	B	C	Hog	A	B	C	Hog
A, B & C active	1/4	1/4	1/4	1/4	1/3	1/3	1/3	0
Hog starts	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4
Hog's share set to zero	1/3	1/3	1/3	0	1/3	1/3	1/3	0
C's share increased	1/4	1/4	1/2	0	1/4	1/4	1/2	0

The left-hand side shows the assignment of shares, the right-hand side shows the expected share of service.

shares are reduced to 1/4, after which we expect to see an identical ratio in the observed throughput of these three servlets.

Figure 5.7 shows our results for the CPUHog servlet. When activated, the CPUHog servlet simply sits in a loop and consumes CPU cycles. The results displayed in the graph match the expected results from the scenario in Table 5.6. The graph is stacked; i.e., the area between the lines indicates what share each servlet received. The straight line at the top was determined by measuring the aggregate average throughput of the MD5 servlets with no Hog. It provides an approximation of the maximum throughput capacity in our system. The CpuHog's

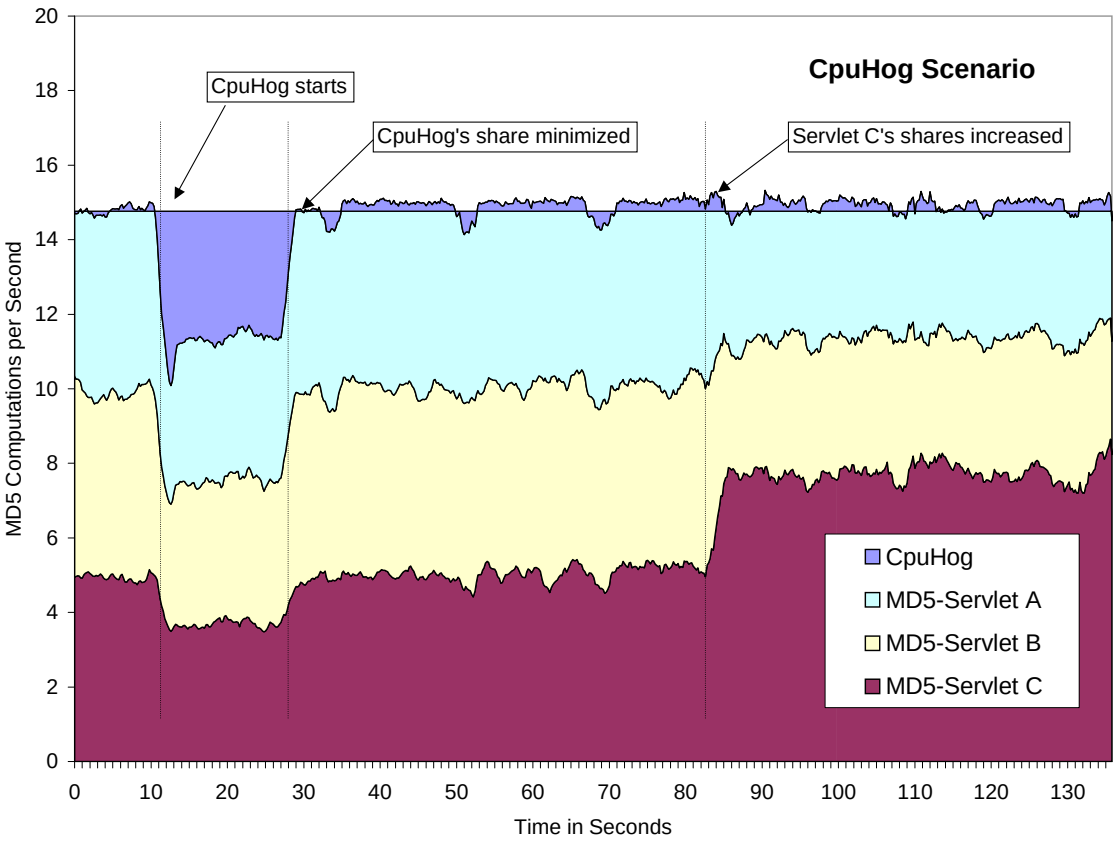


Figure 5.7. Throughput for CpuHog scenario. This stacked graph shows the measured throughput over time for the scenario described in Table 5.6. The measured throughput matches the expected throughput in this scenario.

area is computed as the difference between this expected maximum capacity and the sum of the throughputs of the A, B, and C servlets.

For implementation reasons, we give each task (even those with no share) at least one ticket in the stride scheduler. This implementation artifact accounts for the slight periodic dents at the top, in which the CPUHog gets to run for a brief period of time, despite having been assigned a zero share. We disallow zero tickets to avoid having to provide for CPU time to execute the exit code for applications whose share is reduced to zero—other policies for handling this situation are possible.

5.3.4 GarbageHog Scenario

In the third and last scenario, we replaced the CPUHog servlet with a "GarbageHog" servlet that attempts to hog the garbage collector. The GarbageHog servlet produces some garbage at each invocation; specifically, it allocates 25,000 objects of type `Integer`. Unlike the MemHog servlet, it does not keep those objects alive but drops all references to them. The garbage collector has to collect these objects to reclaim their memory. Unlike the CpuHog servlet, the GarbageHog servlet serves requests: in this respect, it resembles a well-behaved servlet, except for its generation of excessive and unnecessary garbage.

The GarbageHog scenario's results are shown in Figure 5.8. The results are similar to the CpuHog, which indicates that KaffeOS is successful at isolating the garbage collection activity of this servlet. In other words, it demonstrates that we were successful in separating the garbage collection activities of different processes. The garbage collector is no longer a resource that is shared between processes but has become an activity within a process that is subject to the stride scheduler's scheduling policy, along with all other activities in the process.

We must caution that our results are only approximate, in that we do not measure how accurately our prototype's stride scheduler assigns CPU time. Waldspurger [84] discussed stride scheduling as a scheduling mechanism and showed it to be accurate and effective. Our results demonstrate that KaffeOS's mechanisms can successfully defend against certain denial-of-service attacks that are directed at CPU time. The attacks can either be targeted directly at CPU time, as in the

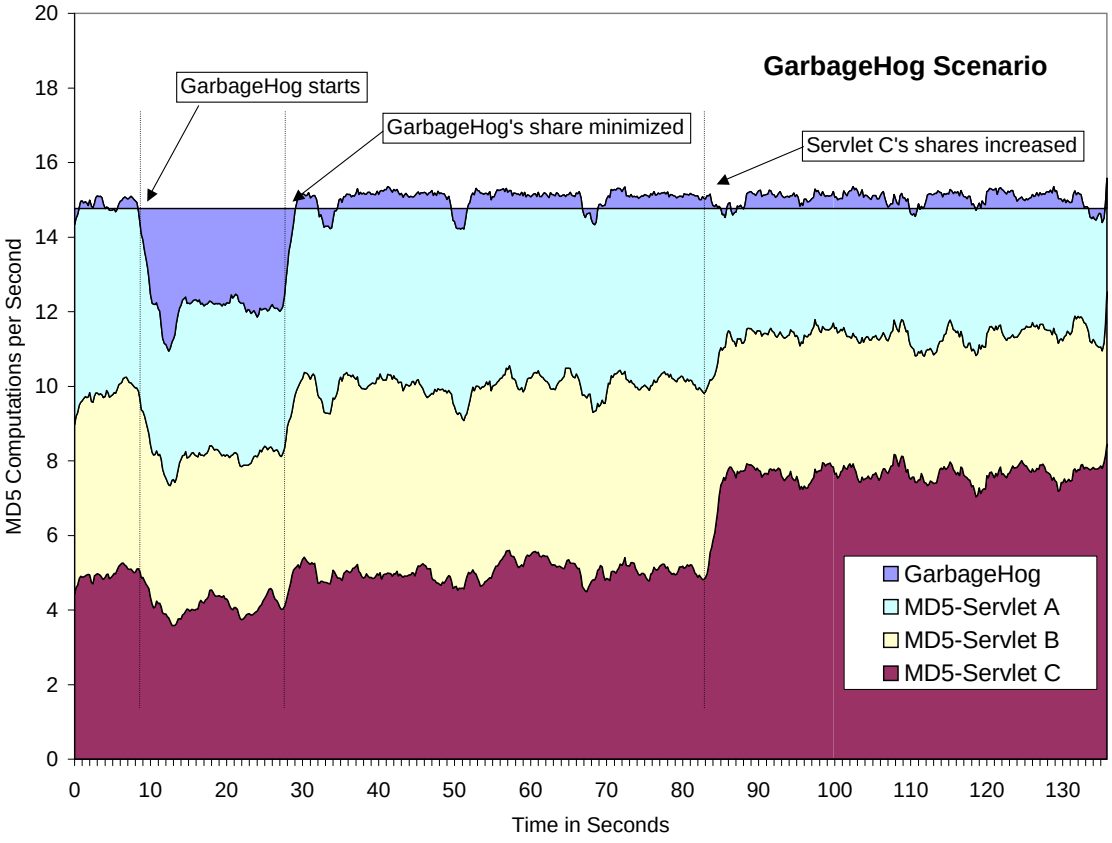


Figure 5.8. Throughput for GarbageHog scenario. This stacked graph shows the measured throughput over time for the scenario described in Table 5.6. The measured throughput matches the expected throughput in this scenario, which proves that the garbage collection activity of the GarbageHog servlet can be controlled independently of the garbage collection activity of the other processes.

case of the CPUHog servlet, or indirectly, as in the case of the GarbageHog servlet. Furthermore, we showed that KaffeOS can provide different processes with different amounts of CPU time, which allows for the implementation of user-defined resource policies.

5.4 Sharing Programming Model

To evaluate the practicality of our model for direct sharing and to investigate its impact on the programming model, we implemented a number of programs that

share different data structures. As proof of feasibility, we implemented a small subset of the Java 1.2 collection API under the shared programming model. We first list some of the restrictions on shared heaps and how they affect programming, and then provide specific examples.

- *Shared heaps remain fixed in size after their creation.* Consequently, all required objects must be allocated while the shared heap is being created. Some collection types, such as `java.util.LinkedList`, use cells that contain a reference to an item in the list and a `next` and `prev` pointer. In an unrestricted implementation, these cells are allocated via `new` when an element is inserted in the list; they become unreachable and eventually subject to garbage collection when an element is deleted. On a KaffeOS shared heap, we cannot allocate new cells after the heap is frozen. Furthermore, because there is no way to reclaim and reuse memory, we cannot afford to discard removed cells. Instead, we must manage cells manually, which can be done by keeping them on a separate freelist.

- *Shared objects cannot allocate data structures lazily.* As discussed in Section 3.3.2, this requirement implies that all link-time references be resolved and all bytecode be translated into native code before freezing a shared heap. This requirement also affects certain programming idioms in the Java code. For instance, the `java.util.HashMap.keySet()` method returns a `Set` object that is created and cached on the first call. Subsequent invocations return the same `Set` object. In our programming model, this object must be created and cached before the shared heap is frozen; for instance, by allocating the `Set` object eagerly in the constructor of `HashMap`.

- *All entry points into the shared heaps must be known.* This requirement follows from the fact that the shared heap's size is fixed once the heap is frozen, and therefore no entry items can be allocated afterwards. An `OutOfMemoryError` is thrown if a user process attempts to create an entry item after a heap is frozen. To circumvent this restriction, we introduced an

API function that allows the reservation of entry items for specified objects while the heap is created. This function preallocates the entry item; when the write barrier code detects that a reference to the object is written to a user heap, it will not attempt to allocate a new entry item, which would fail, and will use the preallocated entry item instead.

Shared classes must be careful to reserve entry items for all objects to which a user heap might acquire references later. Such references are often needed in the implementation of iterator objects for collection classes. To maintain encapsulation, such iterator objects are often implemented as nonstatic inner classes. Nonstatic inner classes are linked to an instance of their encapsulating class through a compiler-generated hidden private field `this$0`. When an iterator object is created on a user heap, a reference to the collection object is written to this field, which creates a cross-heap reference that requires an entry item for the collection object.

One alternative to explicitly reserving entry items would be to provide for the worst case and reserve entry items for all objects that are allocated during the creation of the shared heap. However, these objects also include private objects that are never exported, as well as objects that are part of the shared metadata, which is why we rejected that option. One entry item takes up 12 bytes in our implementation.

- *No references to objects on user heaps can be created.* We must avoid assignments to fields in shared objects, unless the object to which a reference is being assigned is known to also reside on the same shared heap. This requirement precludes the storage of temporary objects, which are allocated on user heaps, in fields of shared objects. Instead, references to such objects must be restricted to local variables on the stack.

5.4.1 Example: java.util.Vector

The `java.util.Vector` class provides a simple container for objects of different types. Vectors are like arrays of arbitrary objects, except that they can automati-

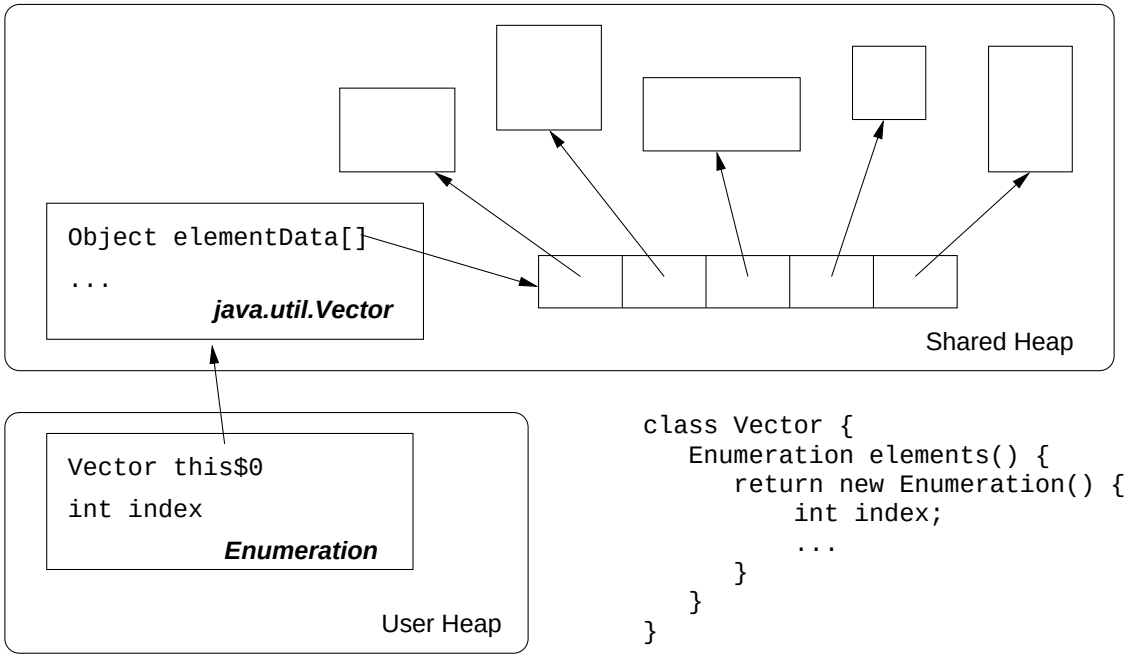
cally grow and shrink and that they can be used as collections and lists. They implement the `Collection` and `List` interfaces and the iterators that these interfaces provide. The `Vector` class also supports the old-style `java.util.Enumeration` interface from JDK 1.1. The full API can be found in [17].

The `Vector` class can be used on a shared heap without any code changes. However, if enumerations or iterators are to be used on a `Vector` object, an entry item must be reserved for the vector instance. Figure 5.9 shows an example of a shared object of type `shared.ShVector` that uses a shared vector. In the constructor, the vector is populated, and an entry item is reserved. The `dump()` method, which is invoked after the heap is frozen, calls `elements()` in the usual fashion. The `Enumeration` object shown in the figure is returned by the `elements()` method.

Table 5.7 lists those methods of `Vector` whose behavior is affected when the `Vector` instance is on the shared heap. No other methods are affected.

Table 5.7. Affected methods of `java.util.Vector` if instance is located on a shared heap.

Method	Impact
`add`, `addAll`, `addElement`, `insertElementAt`, `ensureCapacity`, `set`, `setSize`, `setElementAt`	These methods can fail with `OutOfMemoryError` if the `elementData[]` array needs to be reallocated. If elements are to be added, `ensureCapacity()` should be invoked before the heap is frozen. In addition, a `SegmentationViolationError` can be thrown if an attempt is made to insert an object that is not on the shared heap.
`Vector(Collection)`	This constructor throws a `SegmentationViolationError` if an element of the collection is not on the shared heap.
`copyInto`, `clone`, `toArray`	Because these methods allocate `Object[]` arrays on the user heap into which the elements of the vector are copied, they require that entry items have been reserved for the elements, or else `OutOfMemoryError` is thrown.




```
package shared;
public class ShVector {
    private Vector vec;

    // constructor
    public ShVector() {
        ...
        vec = new Vector();
        while (...) {
            vec.addElement(...);
        }
        // reserve entry item for this$0 field in Enumeration
        Heap.getCurrentHeap().reserveEntryItem(vec);
    }

    // invoked after heap is frozen
    public void dump(PrintStream stream) {
        Enumeration e = vec.elements();
        while (e.hasMoreElements())
            stream.println(e.nextElement());
    }
}
```

Figure 5.9. Use of `java.util.Vector` on a shared heap. Except for the reservation of an entry item for the vector object in the constructor, no programmatic changes are necessary. The entry item is needed to allow for the `Enumeration` object that is returned by `Vector.elements()`.

5.4.2 Example: shared.util.HashMap

The `java.util.HashMap` class provides a generic hashtable implementation. Kaffe's implementation uses an array of buckets, each of which is an anchor to a singly linked list of `Entry` objects. Each `Entry` object consists of a `next` field and two fields that hold references to a $(key, value)$ pair of objects. A hashmap provides methods that return handles to its key and value sets in the form of `Set` objects, which in turn provide iterators. The key and value set objects are implemented in an abstract base class `AbstractMap`, which allows their implementations to be reused for other map types besides `HashMap`.

In KaffeOS's implementation, the use of iterators on a heap requires direct pointers to the list entries in the hashtable. Consequently, we must reserve entry items for all `Entry` objects. However, for encapsulation reasons, these objects are private; i.e., they are not visible to a user of `java.util.HashMap`. Hence, we were unable to use the hashmap implementation unchanged; instead, we created a sharable variant of `HashMap` in the `shared.util.*` package.

Our implementation is shown in Figure 5.10. It differs from the original implementation only in three aspects.

1. It is in the `shared.util` package instead of the `java.util` package. As a side effect, we had to create `shared.util.AbstractMap` as well, albeit without changing `java.util.AbstractMap`, because `HashMap.clone()` relies on package access to fields in `AbstractMap`. Alternatively, we could have added the shared hashmap implementation under a different name to the `java.util` package, but we did not want to needlessly modify the content of the `java.util.*` package, which is part of the Java standard API. We did not duplicate the `Map` interface in the `shared.util` package: hence, shared hashmaps can be used wherever `java.util.Map` instances can be used.

2. It allocates its `keyset` set (and `values` collection, which is treated analogously) eagerly in the constructor.

3. It reserves entry items for all entries and the key and value collections.

```
package shared.util;
import java.util.Map;

public abstract class AbstractMap
  implements Map
{
  Set keyset;
  ...
  public Set keySet() {
    // create on demand and cache
    if (keyset != null) {
      return keyset;
    }

    keyset = new AbstractSet() {
      ...
      // map key iterator to
      // entry set iterator
      public Iterator iterator() {
        return new Iterator() {
          private Iterator i
            = entrySet().iterator();
          public Object next() { ... }
          ...
        };
      }
    }
    return keyset;
  }
  ...
  public abstract Set entrySet();
}
```

```
public class HashMap
  extends AbstractMap
  implements Map, Cloneable,
             Serializable
{
  public HashMap(...) {
    ...
    Heap h = Heap.getCurrentHeap();
    h.reserveEntryItem(keySet());
  }
  ...
  public Object put(Object key,
                    Object val) {
    Entry e;
    ...
    // Create and add new entry
    e = new Entry(key, val);
    Heap h = Heap.getCurrentHeap();
    h.reserveEntryItem(e);
    e.next = table[bucket];
    table[bucket] = e;
    ...
  }

  public Set entrySet() {
    return
      new AbstractMapEntrySet(this) {
        public Iterator iterator() {
          return new EntryIterator();
        }
        ...
      };
  }

  private class EntryIterator
    implements Iterator
  {
    private Entry next, prev;
    private int bucket;
    public Object next() { ... }
  }
}
```

Figure 5.10. Implementation of `shared.util.Hashmap`. By copying the code of `java.util.HashMap` and making slight modifications, we were able to create an implementation of a hashmap that is suitable for use on KaffeOS's shared heaps.

Figure 5.11 discusses the interheap connections that are created if an iterator for the set of keys in the hashmap is created. Table 5.8 lists those methods of `shared.util.HashMap` whose behavior differs from `java.util.HashMap`. All other methods behave in the same way. In particular, the `entrySet`, `values`, and `keySet` behave as defined in the API specification; they return sets and collections with the same types that a `java.util.HashMap` instance would return.

Both the shared vector and the shared hashmap implementations provide examples of immutable data structures. An instance in which the hashmap could be used might be a dictionary that is built once but accessed frequently. If the dictionary changes infrequently, discarding the shared heap that contains the hashmap and copying and updating the hashmap on a newly created shared heap should be viable.

5.4.3 Example: shared.util.LinkedList

A service queue is a common communication idiom that is used in client and server applications. The `java.util.LinkedList` class provides a doubly linked list implementation that can be used for this purpose. Unlike a hashmap, there are few

Table 5.8. Affected methods of `shared.util.Hashmap` if instance is located on a shared heap.

Method	Impact
`put`, `putAll`	These methods will fail with `OutOfMemoryError` because they would require the allocation of a new `HashMap.Entry` object.
`HashMap(Map)`	This constructor throws a `SegmentationViolationError` if a key or value of the map is not on the shared heap.
`clone`	Because these methods create objects on the user heap that refer to the keys and values in the hashmap, they require that entry items have been reserved for the keys and values, or else `OutOfMemoryError` is thrown.

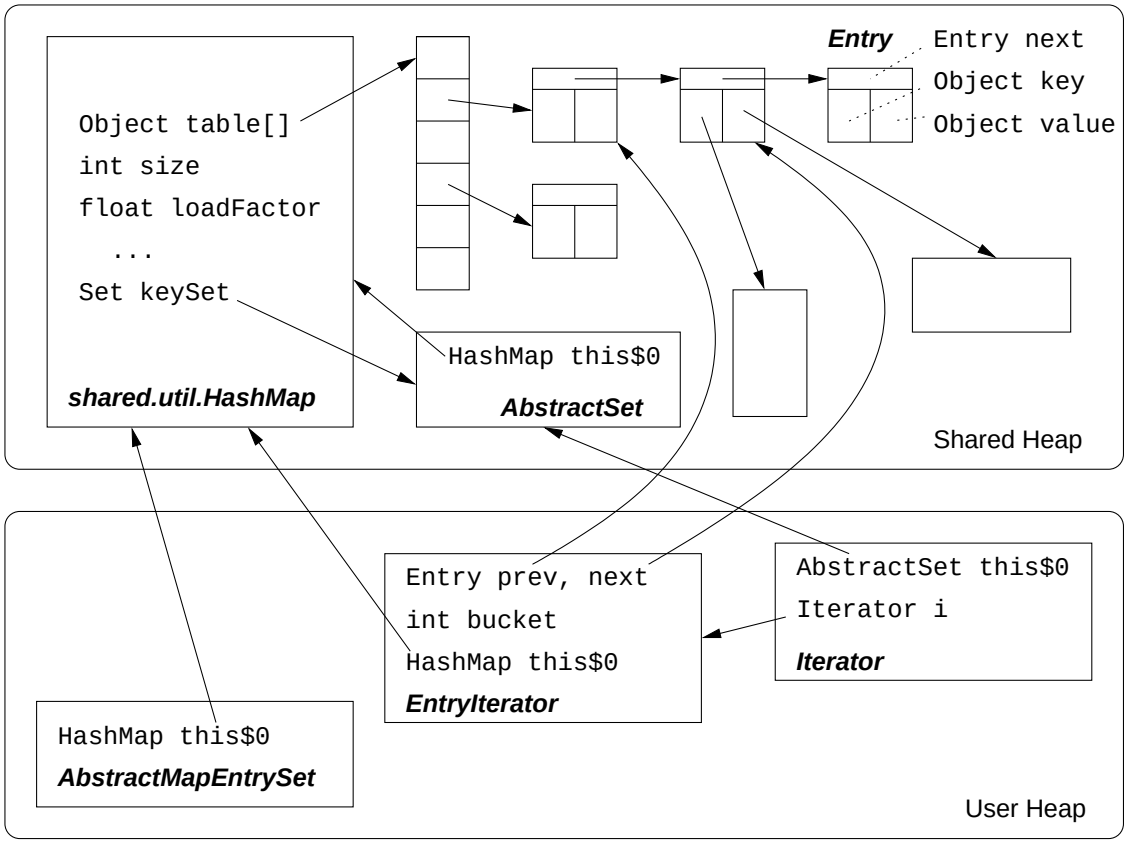


Figure 5.11. Use of `shared.util.HashMap` on a shared heap. This figure shows the objects involved in iterating over the set of keys in the hashmap. The `keyset` field in the hashmap object refers to an `AbstractSet` allocated eagerly on the shared heap. Its `iterator()` method returns an anonymous `Iterator` object, which in turn refers to a `HashMap.EntryIterator` object upon which its implementation is based. This entry iterator object is obtained from a call to `HashMap.entrySet()`, which allocates an `AbstractMapEntrySet` object and a subsequent invocation of the entry set's `iterator` method. In KaffeOS, the use of iterators for hashmaps on a shared heap is unconstrained, because our implementations allocate the key set eagerly and reserve entry items for all `Entry` objects (not shown in the figure). See Figure 5.10 for accompanying code.

uses for a queue that do not involve changing the queue's elements. We therefore adapted the implementation to allow for the addition and removal of elements to the queue. Because shared heaps are a fixed size, addition is only possible as long as the total number of added elements does not exceed maximum number specified when constructing the `shared.util.LinkedList` object.

The modifications that were applied to `LinkedList` are discussed in Figure 5.12. Figure 5.13 sketches the object instances involved. As with shared hashmaps, each list cell requires the reservation of an entry item.

Table 5.9 lists those methods of `shared.util.LinkedList` whose behavior differs from `java.util.LinkedList`. All other methods behave in the same way; in particular, iterator() returns an object of type `java.util.Iterator`, and the `shared.util.LinkedList` itself can be used as a `java.util.List` instance.

Putting It All Together: To provide an example of how KaffeOS's shared heaps can be used for interprocess communication, we used the shared hashmap and

Table 5.9. Affected methods of `shared.util.LinkedList` if instance is located on a shared heap.

Method	Impact
`LinkedList()`	This constructor's signature was changed to take an integer argument (`LinkedList(int)`); the argument specifies the maximum number of elements in the list.
`LinkedList (Collection)`	This constructor throws a `SegmentationViolationError` if an object in the collection is not on the shared heap.
`add`, `addAll`, `addFirst`, `addLast`, `set`	If the number of elements exceeds the available capacity, an `OutOfMemoryError` is thrown. If the element is located on the user heap, a `SegmentationViolationError` is thrown.
`clone`, `toArray`	Because these methods create list elements on the user heap that refer to the original list elements, they require that entry items have been reserved for these elements, or else `OutOfMemoryError` is thrown.

```
package shared.util;

public class LinkedList
  extends AbstractSequentialList
  implements List, Cloneable, Serializable
{
  Elem head = null;
  Elem tail = null;
  private Elem freelist;

  // get a fresh Elem cell
  private Elem getElem(Object o) {
    Elem next = freelist;
    if (next == null) {
      throw new OutOfMemoryError();
    }
    freelist = freelist.next;
    next.o = o;
    next.next = next.prev = null;
    return next;
  }

  // recycle used Elem cells
  private Elem putElem(Elem e) {
    Elem oldnext = e.next;
    e.next = freelist;
    freelist = e;
    return oldnext;
  }

  // preallocate cells
  public LinkedList(int n) {
    Heap h = Heap.getCurrentHeap();
    for (int i = 0; i < n; i++) {
      Elem e = new Elem(null);
      h.reserveEntryItem(e);
      putElem(e);
    }
    h.reserveEntryItem(this);
  }

  public void addFirst(Object o) {
    ...
    // was: e = new Elem(o);
    Elem e = getElem(o);
    ...
    if (length == 0) {
      head = tail = e;
    } else {
      e.next = head;
      head.prev = e;
      head = e;
    }
    ...
  }

  public Object removeFirst() {
    ...
    Object rtn = head.o;
    // was: head = head.next
    head = putElem(head);
    if (head == null) {
      tail = null;
    } else {
      head.prev = null;
    }
    ...
    return rtn;
  }

  class LinkedListIterator
    implements ListIterator
  {
    LinkedList.Elem elem;
    ...
    public Object next() { ... }
  }
}
```

Figure 5.12. Implementation of `shared.util.LinkedList`. To make the linked list implementation suitable for inclusion on a shared heap, we manually manage a number of preallocated `Elem` cells. Compared to the original code in `java.util.LinkedList`, we added a `freelist` field and methods `getElem` and `putElem` to get and put elements on the freelist. In the rest of the code, we replaced all calls to `new Elem(...)` with calls to `getElem`, as shown in the `addFirst` method. In addition, we carefully inserted calls `putElem` in those places where a list element is discarded, as, for example, in `removeFirst`.

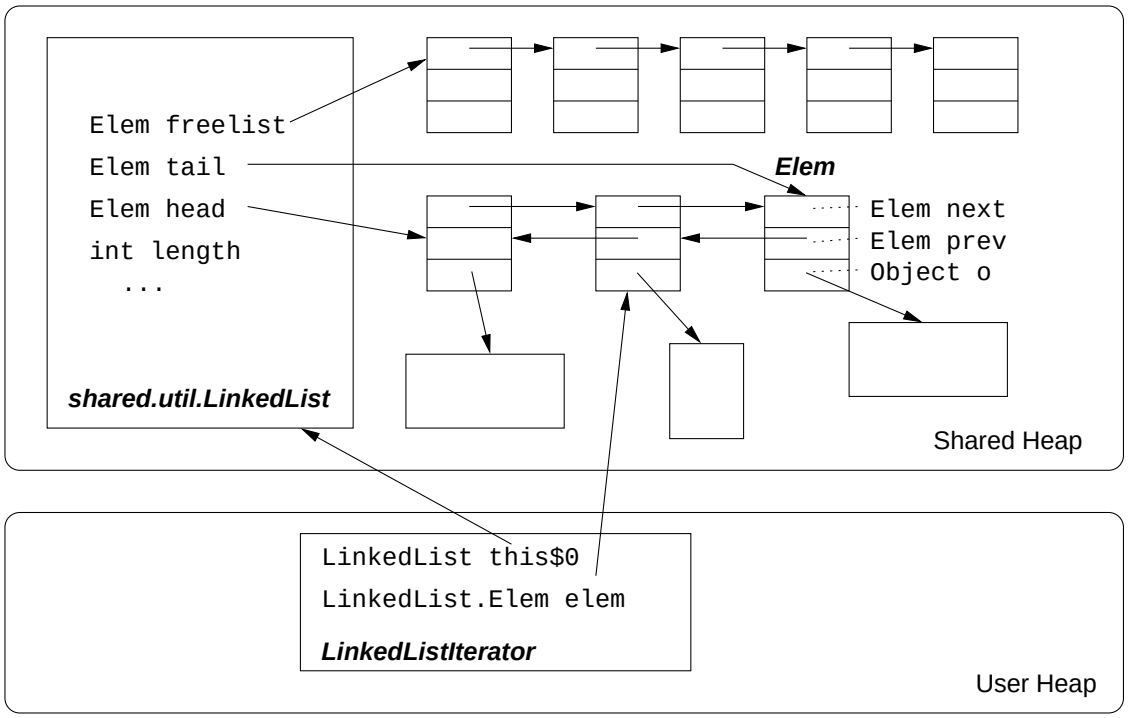


Figure 5.13. Use of `shared.util.LinkedList` on a shared heap. Our implementation of `shared.util.LinkedList`, which can be used for queues, adds a freelist of `Elem` objects to store elements not currently in use. The queue is bounded by the number of `Elem` cells that are preallocated. When an object is added to the list, a cell is moved from the freelist onto the doubly linked list headed by `head`; the cell is moved back onto the freelist when the object is removed. The use of `LinkedListIterator` instances is unconstrained since we reserve entry items (not shown in figure) for all elements and the `LinkedList` instance itself. See Figure 5.12 for accompanying code.

linked list classes to implement a servlet microengine, whose code is shown in Appendix B. Unlike the Apache/JServ engine, this engine does not implement the full Java servlet API; however, the example is sufficiently complete to illustrate KaffeOS interprocess communication by way of shared heaps.

The example includes three processes: a http server process and two servlet engine processes. The server and servlet processes share a queue for http requests on a shared heap. The http server receives http GET requests, examines the uniform resource locator (URL) contained in the request, and chooses the servlet to which

to dispatch the request. In our example, the servlets perform identical functions, although there is of course no such restriction in the general case. Each servlet waits for requests on its queue and processes incoming requests. The servlets perform a lookup in a German-English dictionary and display the result to the user. The dictionary is stored in an object of type `shared.util.HashMap` located on a second heap that is shared between the servlet processes.

5.4.4 Extensions

The restrictions on our shared programming model directly result from two independent assumptions. The first assumption is the desire to avoid sharing attacks and guarantee full reclamation, which provides isolation between processes. This goal requires the use of write barriers to prevent illegal cross-heap references. The second assumption is our desire to prevent programs from asynchronously running out of memory. For this reason, all sharers are charged, which in turn requires that a shared heap is frozen after creation so as to accurately charge all sharers. In some situations, we might be able to relax this model somewhat.

For instance, sharing attacks cannot occur through a shared heap when it is known that all sharers terminate at around the same time. Although this is not typically true for client-server systems, it often holds when different processes co-operate in a pipe-like fashion. In this situation, we could relax the first assumption and would not need to prevent cross-references from the shared heap to the user heaps of the participating processes, because we would know that all user heaps and the shared heap (or heaps) are merged with the kernel heap once the processes terminate.

The second assumption could be relaxed when all communicating processes share a soft ancestor memlimit; we do not need to double charge in this case. This situation could occur if the processes trusted each other. As discussed in Section 3.4.1, if multiple processes share a soft parent memlimit, they are subject to a summary limit. They then have to coordinate not to exceed that limit. In this scenario, the shared heap could be associated with a memlimit that is a sibling of all the memlimits of the sharing processes. The sharing processes must avoid situations

in which a process cannot allocate memory on its heap because another process allocated too much memory on a heap that is shared by both. This extension would require the provision of an API that would allow untrusted user threads to decide from which heap allocations should be performed.

Another possible extension would be to garbage collect shared heaps after they are frozen. However, unlike for user heaps, the memory reclaimed from unreachable objects would not be credited back to a shared heap but would be used to satisfy future allocations from the shared heap instead. This scenario would also require an API function that allowed untrusted threads to allocate new objects from a shared heap. Adding such memory management facilities would eliminate the need for manual memory management, such as the freelist scheme used to manage elements in `shared.util.LinkedList` objects, which we described earlier.

However, there are possible drawbacks with such a scheme: First, if a conservative collector is used, unreachable objects may be kept alive as floating garbage, which is outside the programmer's control. Therefore, it is not guaranteed that unreachable objects can be reclaimed in time to satisfy a future allocation. This uncertainty would imply that a process could not rely on being able to allocate new objects, which is undesirable. Even if a precise collector were used, meticulous assignments of `null` values to locations that hold references to dropped objects are needed to avoid memory leaks; such a programming model is error prone. Second, buggy code might inadvertently allocate objects on a shared heap. Such bugs could cause a shared heap to run out of memory at some unspecified point in the future, which would make them hard to debug. For these reasons, we did not implement this alternative.

5.5 Summary

We performed a series of experiments to evaluate how our KaffeOS prototype achieves KaffeOS's design goals. The first group of experiments was designed to measure the overhead introduced by KaffeOS, when compared to a JVM that does not provide isolation and resource management for processes. The second group

of experiments was designed to show that our prototype achieves KaffeOS's design goals and that achieving these design goals provides significant practical benefit.

Our first set of experiments examined the overhead introduced by KaffeOS. This overhead stems from both its use of write barriers and changes we made to the underlying Kaffe JVM to support multiple processes. Our main result is that this overhead is not prohibitive. However, we were unable to pinpoint the exact contribution of the write barriers to the total overhead observed. Our measurements showed a total run-time overhead of less than 8%. A conservative estimate of the expected overhead if we integrated KaffeOS's write barrier code in a state-of-the-art JVM puts the maximum run-time overhead at 14%. Because the total overhead remains reasonable, we conclude that the use of write barriers for memory isolation and resource control is practical.

KaffeOS's separate, per-heap garbage collection requires that each process has accurate information about references on remote thread stacks that are pointing to its heap. This requirement imposes an additional burden on the garbage collector. Examining the stacks of threads in other processes has the potential to become a source of priority inversion should one process need to stop the threads in other processes to perform those scans. Our implementation of thread stack scanning is simple in that it scans every remote thread every time without sharing any results between collectors. We found that this implementation does not become a source of priority inversion, because remote threads do not have to be stopped for long periods of time. The total execution time of our algorithm depends on the total number of threads in the system, which must therefore be limited. We conclude from these experiments that our design, which requires taking possible references from remote threads into account, can be implemented in a way that only slightly affects the isolation provided between processes.

We examined the behavior of our prototype under denial-of-service scenarios, both those directed at memory and at CPU time. For memory, the principal line of defense is to limit an offending process's memory use and to kill it if it attempts to use more than its limit. KaffeOS isolates different processes, which is why it

is able to terminate the offending process safely and to reclaim virtually all of its memory. Hence, the impact of a denial-of-service attack can be contained to the offending process; unrelated processes continue to be able to deliver service to clients. Our prototype's ability to defend against such attacks is a direct consequence of KaffeOS's design.

Our strategy for preventing denial-of-service attacks against CPU time or the garbage collector is to guarantee that well-behaved tasks obtain enough CPU time to complete their tasks. This strategy requires that there are no significant sources of contention between processes and that all activity that is done on behalf of a process is accounted for in the context of that process. KaffeOS's garbage collection scheme allows a process's garbage collection activity to be accounted for in the context of that process. This separation of garbage collection eliminates an important source of contention, which enables KaffeOS to successfully defend against denial-of-service attacks directed at CPU time.

Our final concern in this chapter was the practicality of the model for direct sharing and its impact on programs. A program could behave differently than when run in a standard JVM in two ways: it could run out of memory because shared heaps are fixed after creation, and it could throw segmentation violation errors if it tries to write pointers to its heap into a shared heap. We showed that despite these restrictions, processes are able to easily and directly share complex data structures such as vectors, hashmaps, and lists. In addition, such shared collection classes can implement standard collection interfaces, which facilitates seamless integration into existing applications. We gave examples of how existing classes can be adapted for this programming model and how applications can make use of shared heaps in a servlet microengine application. We conclude that our model of direct sharing provides a viable way for processes to communicate while preserving the constraints imposed by isolation and resource control.

CHAPTER 6

RELATED WORK

The discussion of related work is divided into three sections. In the first section, we discuss work that is based on or directly related to Java. In the second section, we discuss KaffeOS's relationship to other work in the area of garbage collection and resource management. Finally, we conclude by comparing KaffeOS to operating systems, including those that—like KaffeOS—use a single address space and those that exploit language mechanisms for extensibility.

6.1 Java-based Systems

Several other researchers and developers have built systems for Java applications that address some of the problems KaffeOS is designed to address. Early systems [4, 8, 68] provided simple multiprocessing facilities for Java without addressing isolation and resource management. Other systems focused on the increased scalability that can be obtained by running multiple applications in one JVM [21, 29, 55], but they do not address resource control. We compare KaffeOS to systems that also provide process models for Java but that use different approaches, make different assumptions about the environment in which they run, or use different sharing models [42, 80, 81]. Finally, we discuss Java-based systems that are related with respect to either goals or mechanisms but that address only a single issue, such as resource control [25], memory management [14], or termination [67].

6.1.1 Early Java-based Systems

One of the earliest systems that aimed at providing the ability to run multiple applications within the same Java virtual machine was described by Balfanz [4]. His modified JVM provides different applications with different namespaces and

the illusion of executing in a JVM by themselves. However, there was no isolation beyond namespace separation: none of the computational resources was accounted for, and applications could not be safely terminated.

Sun's first JavaOS [68] provided an environment in which to run Java programs in a JVM without a separate underlying operating system. Its primary goal was to implement as many OS subsystems as possible in Java, such as device drivers and graphics drivers. Applications were assumed to be cooperative, and there were no mechanisms to deal with uncooperative applications. For instance, applications were notified of a low memory situation and asked to reduce their resource usage voluntarily.

The Conversant project prototyped a system [8] whose goals were to provide quality of service guarantees for mobile Java code. Written for the Realtime Mach OS, it mapped Java threads onto real-time native threads. The Conversant system did provide separate heaps for multiple applications in a JVM. However, it did not provide a way for applications to share objects and resources nor did it allow for safe termination.

6.1.2 Industrial Scalable Java Virtual Machines

IBM designed and built a scalable JVM for its OS/390 operating system that is targeted at server applications [29]. Its main goal is to reduce the overhead associated with starting up multiple JVMs by allowing multiple JVM instances to share static data and classes. The OS/390 JVM can be run in two modes: in a resource-owning mode or in a worker mode. Each JVM instance runs in its own operating system process, but a resource-owning JVM can share system data with multiple JVMs running in worker mode. These data, which primarily consist of resolved classes, are allocated on a shared heap that is accessible to all worker JVMs. This shared heap is comparable to KaffeOS's kernel heap. Like in KaffeOS, each worker JVM also has a local heap from which it allocates local objects. Worker JVMs can be reused between applications if the applications that run in them are well-behaved. An application is considered well-behaved if it leaves no residual resources such as threads or open file descriptors behind. If an application has been

well-behaved, the heap of the worker JVM is reinitialized and another application is run. If it was not, the worker JVM process exits and relies on the operating system to free associated resources.

Unlike KaffeOS, IBM's JVM is not concerned with protecting applications from untrusted code. It assumes that the resource-owning and the worker JVMs cooperate. For instance, every JVM can place classes in the shared heap area; access to the shared heap is not controlled. Local heaps are garbage collected separately; however, the question of how to handle cross-references does not arise because IBM's shared heap is not garbage collected. These design decisions suggest that IBM's system would be inadequate when dealing with malicious applications that could engage in sharing attacks. The system also does not appear to address the issue of safely terminating uncooperative applications.

Oracle's JServer environment [55] provides a Java virtual machine to support such server-side technologies as enterprise beans, CORBA servers, and database stored procedures. It provides virtual virtual machines or VVMs for such applications. VVMs use very little memory (as little as 35 KB for a "Hello, World" application.) All system data are shared between VVMs. A VVM's local storage area can be garbage-collected separately. Like IBM's OS/390 JVM, Oracle JServer primarily focuses on increasing scalability by reducing per-session memory usage and is unable to cope with untrusted or failing code in a robust manner.

Czajkowski [21] describes a way to avoid the use of classloaders to achieve the same effect as reloading classes for multiple applications. His scheme logically duplicates the static fields of a class for all applications using that class. Additionally, static synchronized methods—which according to Java's semantics acquire a lock on the Java class object—are changed to acquire per-application locks.

Two implementations of this scheme are presented: one implementation in pure Java that relies on rewriting bytecode and an implementation integrated into KVM, which is a small Java interpreter for the PalmPilot portable device. The bytecode approach replaces all accesses to static fields with calls to accessor functions. Despite the high run-time overhead of this approach, it outperforms

a class loader-based approach in the number of applications that can be run in one JVM, because only a single copy of each class is loaded for all applications. Because class loaders are not used, running different versions of a given class is more complicated in this approach.

KaffeOS uses class loaders to provide multiple copies of static fields for some system classes, but it could benefit from a faster and more memory-efficient mechanism such as the one used in KVM. KaffeOS's design does not hinge on which mechanism for locally replicating static fields is chosen. In addition, much of the memory savings could be obtained by optimizing Java's class loading mechanism. In particular, a VM could share internal data structures if it recognized when different class loaders reload the same class.

In a follow-up project, called the Multitasking Virtual Machine (MVM) [23], Czajkowski and Daynès modified the HotSpot virtual machine to support safe, secure, and scalable multitasking. MVM's design extends the approach to class sharing developed in their earlier work. Unlike KaffeOS, it deliberately does not support a way for applications to directly share user data.

In MVM, a runtime component is either fully replicated on a per-task basis or it is shared. Read-only state can be trivially shared. Writable globally shared state, which KaffeOS manipulates within its kernel, is manipulated in separate server threads instead. The services provided by these threads include class loading, just-in-time compilation and garbage collection. These servers service requests from application threads one at a time. This approach does not require user threads to directly manipulate shared state and thus simplifies task termination, but it can incur context-switching overhead; in addition, because requests are serialized, access to such shared components can become contended.

6.1.3 J-Kernel and JRes

The J-Kernel system by Hawblitzel et al. [42] is a layer on top of a standard JVM that adds some operating system functionality. Its microkernel supports multiple protection domains called tasks.

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 that contains a direct pointer to the shared object. Because of the level of indirection through capabilities to shared objects, access to shared objects can be revoked. As with shared classes in KaffeOS, a capability can be passed only if two tasks share the same class through a common class loader.

All arguments to intertask 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. In contrast, KaffeOS provides its direct sharing model as a means of interprocess communication. Although not optimized for indirect sharing, KaffeOS does not preclude the use of object serialization if needed. Unlike the J-Kernel's microkernel, indirect sharing in KaffeOS is not a task's primary means to obtain access to system services such as filesystem or network access—those are provided by the KaffeOS kernel.

The J-Kernel's use of indirect sharing, combined with revocation of capabilities, allows for full reclamation of a task's objects when that task terminates. However, because the J-Kernel is merely a layer on top of a standard JVM, it cannot isolate the garbage collection activities of different tasks, nor can it account for other shared functionality.

The J-Kernel supports thread migration between tasks if a thread invokes a method on an indirectly shared object. The thread logically changes protection domains during the call; a full context switch is not required. To prevent malicious callers from damaging a callee's data structures, each task is allowed to stop a thread only 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. In contrast, KaffeOS can always terminate uncooperative threads safely with respect to the system, but interprocess commu-

nication will involve at least one context switch.

A companion project to the J-Kernel, the JRes system [22, 25], provides a way to control and manage computational resources of applications. Like the J-Kernel, JRes can run on top of a standard JVM. It controls CPU time, memory, and network bandwidth. CPU time is accounted for by using hooks into the underlying OS, which is responsible for scheduling them. JRes lowers the priority of threads that have exceeded their resource limit. Memory is accounted for by rewriting Java bytecode in a way that adds bookkeeping code to instructions that allocate memory. JRes adds finalization code to classes so that tasks can be reimbursed for allocated objects.

Bytecode rewriting is a viable approach where changes to the underlying JVM cannot be done. However, it suffers from high overhead. In addition, it makes resource accounting incomplete, because allocations within internal VM code cannot be accounted for.

6.1.4 Alta

The Alta [79, 80] system provides an environment that runs multiple applications in a single JVM that is modeled after the Fluke microkernel [36]. The Fluke microkernel provides a nested process model, in which a parent process controls all aspects of its child process. It does so by interposing on the child's communication, which is done via IPC. As in many microkernels, system services are provided through servers, which complicates per-task accounting.

Capabilities provide a way to indirectly share objects between processes, but there are no write barriers to prevent the leaking of references to objects in one process to another. It is the parent's responsibility to restrict a child's communication so object references are not leaked. As a result, Alta cannot always guarantee full reclamation.

Objects can be directly shared between processes if their types, and the closure of the types of their constituent fields, are structurally identical. This approach constitutes an extension of Java's type system. By contrast, KaffeOS guarantees

the type safety of its directly shared objects through the use of a common class loader.

Alta provides the ability to control a process's memory usage, which is also done by the process's parent. Unlike KaffeOS, Alta does not separate the memory resource hierarchy from the process hierarchy. Because Alta does not provide separate heaps, it cannot account for garbage collection activity on a per-process basis.

6.1.5 Secure JVM on Paramecium

Van Doorn describes the implementation of a secure JVM [81] that runs on top of the Paramecium extensible operating system [82]. Van Doorn argues that language-based protection is insufficient and prone to bugs; hence, the multiple protection domains their JVM supports are separated using hardware protection, which eliminates the need to trust the bytecode verifier.

In the Paramecium JVM, each class or instance belongs to a given domain. The language protection attributes of a class or instance, such as private or protected, are mapped to page protection attributes. As a result, an object's fields may have to be split over multiple pages (*instance splitting*).

Classes and objects are shared between domains if they are placed in the same page. False sharing could occur if a page contained both a shared object and another, unrelated object that should not be shared. Because such false sharing would violate the confinement of domains, the garbage collector moves objects or parts of objects between pages to prevent this situation. Conversely, the garbage collector attempts to move objects that can be safely shared to pages with the same set of access permissions.

Because one garbage collector is responsible for all domains, garbage collection costs are not separately accounted for. In addition, in order to not undermine the gain in safety added by using hardware protection, the collector must be written in a way that can handle possible corruption of the objects it examines. This necessity leads to increased overhead. Finally, denial-of-service attacks against the collector

are possible. For example, an adversary could manipulate an object's color in a way that causes the collector to consume large amounts of CPU.

By allowing direct access to shared pages, the Paramecium JVM avoids marshaling costs during interprocess communication. The Paramecium JVM must also pay the overhead involved in context switching between different hardware protection domains. The authors provide no quantitative data as to the efficiency of their sharing model, which prevents us from comparing it to KaffeOS.

The Paramecium JVM can control a thread's CPU usage by manipulating its scheduler priority; cross-domain calls are done via migratory threads. As in the J-Kernel, safe termination of uncooperative threads is not possible.

6.1.6 Real-Time Java (RTJ)

In 1998, Sun Microsystems started an effort to bring real-time capabilities to the Java platform. Using its Java Community Process, Sun, IBM, and others developed a specification for Real-Time Java (RTJ) [14], an implementation of which has been announced in May 2001. Unlike KaffeOS, RTJ is not concerned with untrusted or buggy applications but instead focuses on providing bounds on the execution time of applications. In particular, RTJ provides facilities to manage memory in a way that allows for real-time execution bounds in the presence of automatically managed memory. Despite dissimilar goals, RTJ and KaffeOS use similar mechanisms.

Similar to KaffeOS processes that allocate memory from their own heaps, threads in RTJ can allocate their objects from *scoped memory*. Such memory areas are not garbage collected and can be freed once all threads have finished using them. In this respect, a scope is similar to explicit regions [39], which are a memory management scheme in which a programmer can allocate objects in different regions but free regions only as a whole.

As in KaffeOS, write barriers are used to ensure this property at run time, and a run-time error is triggered when an attempt to establish a reference from an outer, longer-lived scope into an inner, short-lived scope is being made. Unlike KaffeOS heaps, scoped memory is not garbage collected and does not allow any cross-scope references.

RTJ also provides a mechanism to asynchronously terminate threads. To address the problem of corrupted data structures, RTJ allows the programmer to declare which sections of code are safe from termination requests and in which sections of code such requests must be deferred. This use of deferred termination is similar to the KaffeOS user/kernel boundary. Unlike in KaffeOS, entering a region in which termination is deferred is not a privileged operation. The goal of terminating threads asynchronously in RTJ is not to be able to kill uncooperative applications but to provide the programmer with a means to react to outside events.

RTJ's designers acknowledge that writing code that is safe in the face of asynchronous termination is hard. For this reason, asynchronous events are deferred by default in legacy code, as well as during synchronized blocks and constructors. Only code that explicitly enables asynchronous events by using a throws clause can be asynchronously terminated. KaffeOS's user/kernel boundary does not provide termination as a means of programming multithreaded applications, but solely as a means to terminate threads as a process exits.

We strongly believe that the increased robustness provided by the ability to deal with buggy—if not untrusted—applications would also benefit those embedded systems at which RTJ is targeted. KaffeOS's implementation base could be easily extended to provide such facilities as scoped memory.

6.1.7 Soft Termination

Rudys et al. [67] propose a scheme for safe termination of "codelets" in language-based systems. Their work focuses only on termination and does not include other aspects of a fully developed process model, such as memory control or sharing.

A codelets's class file is rewritten such that a "check-for-termination" flag is added to each class. The bytecode is rewritten to check this flag whenever a non-leaf method is invoked and on every backward branch. System code is not rewritten in this way. Hence, as in KaffeOS, termination is deferred while a thread is executing system code. The authors provide a formal semantics for the rewrite rules and are able to prove that a thread terminates in a bounded amount of time when a termination request is issued.

They argue that a user/kernel boundary is too inflexible for a language-based system because a thread can cross in and out of system code. Although user and system code can indeed both call each other, it does not follow that calls in both directions can be treated alike. Like in KaffeOS, every call from system code to user code must still be treated as an upcall.

Contrary to their claims, the semantics and expressiveness of the soft termination scheme are mostly identical to KaffeOS's user/kernel boundary. Their implementation has the important advantage that it functions on a Java virtual machine that follows Sun's specification, but the price they pay is a 3–25% overhead for checking the flag even if no termination request is pending (the common case). By comparison, KaffeOS does not need to check when executing in user mode; checks are only necessary when leaving kernel mode.

6.2 Garbage Collection and Resource Management

Some garbage collection techniques address issues related to KaffeOS, such as how to reduce the impact of having to perform garbage collection and how to use memory revocation to implement isolation.

6.2.1 GC Issues

Incremental collection: Incremental collection algorithms can limit the time spent on garbage collection. An incremental collector allows garbage collection to proceed concurrently with the execution of the program. Such concurrency can be achieved by performing a small bit of garbage collection every time the program performs an allocation. Doing so reduces garbage collection pauses because well-behaved programs have to stop only at the end of a GC cycle to sweep reclaimable objects. Baker's treadmill [3] algorithm goes one step further by using linked lists to make sweeping immediate.

Incremental collection is a useful concept for real-time systems or other systems desiring or requiring promptness, but it does not address the issue of proper accounting. In particular, processes may incrementally collect garbage they did not cause, which violates the separation between processes. In contrast, we ensure

that one process spends resources only on collecting its own garbage. Our design supports incremental collection within one heap.

Moving collectors and fully persistent systems: A moving collector is able to move objects in memory to compact the used portion of the heap. It has the ability to recognize and adjust references to the objects being moved. This ability can be exploited to fight sharing attacks in a different way than KaffeOS does: namely, by directly revoking references to objects whose owners have gone away. Such revocation can be done by storing an invalid reference value in a forwarding pointer. Some objects, however, may be pinned down by an application, for instance, because they are being accessed by native code. Consequently, this approach would not work for all objects.

Fully persistent systems go one step further by providing the means to save and restore all objects to backing storage. A Modula-3 runtime system [30] used compiler support to maintain precise information on all locations that store pointers or pointer-derived values. Such complete information would be needed to make object revocation an effective tool against sharing attacks. This approach may become more feasible in the future as more complex just-in-time compilers provide more of the information needed.

Distributed GC: Plainfossé and Shapiro present a review of distributed garbage collection techniques in [61]. These techniques assume a model in which objects are located in disjoint *spaces* separated by protection mechanisms. Objects cannot be uniformly accessed by their virtual memory address. Spaces communicate with each other by exchanging messages. When a space exports a public object, an entry item is created in the owner space, and exit items are created in all client spaces that use the object. Mechanisms to collect objects across spaces include reference counting, reference listing, and tracing.

Garbage collection across spaces is complicated by several factors. Simple reference counting requires that messages arrive in causal order [51], to prevent the premature reclamation of an object if a decrement message arrives earlier than an increment message that preceded it. In addition, increment and decrement are

not idempotent operations, so that lost messages cannot simply be resent. This problem is avoided by reference listing, which keeps multiple entry items for each client. KaffeOS does not use message passing to update its reference counts, so no message-related problems can occur. However, the KaffeOS kernel must be equally careful to execute operations that change reference counts in the proper order, because certain operations on references can trigger garbage collections.

Neither reference counting nor reference listing can reclaim cyclic garbage. Typically, systems relying on these mechanisms assume that cyclic garbage is infrequent and that any accumulation thereof can be tolerated. Hybrid technologies, such as object migration [13], address this shortcoming. Objects involved in cycles are migrated to other spaces until the cycle is locally contained in one space and can be collected. KaffeOS's merging of user heaps into the kernel heap can be viewed as migration of a user heap's objects to the kernel heap.

6.2.2 Scheduling and Resource Management

CPU scheduling: CPU inheritance scheduling [37] is a scheduling mechanism that 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. The thread receiving the donation is said to inherit the CPU. Since the thread that inherits the CPU may in turn function as a scheduler thread, scheduler hierarchies can be built. Each nonroot 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.

CPU inheritance scheduling is a mechanism, not a scheduling algorithm. Its strength lies in its ability to accommodate different algorithms that can implement different policies. We could have used CPU inheritance scheduling for KaffeOS. However, the universality it provides exacts a substantial increase in implementation complexity. This added complexity may be justified only for applications that need the added flexibility inheritance scheduling provides, for instance, to implement application-specific scheduling policies. Instead, we used a fixed policy, stride-scheduling, which is a scheduling policy derived from lottery scheduling.

Lottery scheduling, developed by Waldspurger [84], is a proportional-share scheduling algorithm in which each thread is assigned a certain number of lottery tickets. When a scheduling decision is made, a lottery randomly chooses a thread to be scheduled. Since a thread's chances to be picked are proportional to the number of lottery tickets it holds, threads obtain amounts of CPU time that are proportional to their ticket allocations.

Stride scheduling improves on lottery scheduling by avoiding its indeterminism. In stride scheduling, every thread is assigned a fixed stride whose length is indirectly proportional to the number of tickets the thread holds. In addition, a virtual clock is kept for each thread. When a thread is scheduled, its virtual clock is advanced by the number of time quanta multiplied by its stride. The scheduler always picks the thread with the lowest virtual time, which creates a regular scheduling pattern.

Managing multiple resources: Sullivan and Seltzer developed a resource management framework that takes multiple resources into account [74]: CPU time, memory, and disk bandwidth. Like KaffeOS, this framework uses proportional-share algorithms to schedule these resources but goes further in that it allows resource principals to exchange tickets. For instance, a CPU-bound application with little memory consumption can exchange tickets with an application that needs less CPU time but has a larger working set to keep in memory. Sullivan demonstrated that such ticket exchanges can improve overall application performance. KaffeOS could also benefit from the flexibility provided by such exchanges. For instance, applications with high allocation rates could trade CPU time for more memory, which could reduce the frequency of garbage collection. Applications that produce little or no garbage could benefit by trading some of their memory allocations for CPU shares.

Resource containers: Resource containers [5] decouple resource consumption and protection domains in a hardware-based operating system. Traditionally, such systems merged both abstractions into their process abstraction. However, if a programmer uses two processes to implement an activity that is executed on behalf of the same resource principal, then these systems are unable to accurately account

for the resources that are consumed. Resource containers are a means to account for the consumption of resources across traditional process boundaries. They are bound to threads, but they can switch the thread to which they are bound. Resource containers serve a second function in that they simultaneously denote a resource capability. They can be passed between processes, and they can be arranged in a hierarchy to enable hierarchical resource management.

KaffeOS's resource objects provide a similar decoupling of resource consumption and processes as resource containers. The resource object hierarchy can be used to ensure that multiple processes are subject to a common memory limit or CPU share. However, resource objects cannot be passed between processes and threads the way resource containers can. This limitation is partly because memory handed out to a process is not revocable in KaffeOS, unless the whole process is terminated. Sullivan [74] reports that even in systems that support paging, scheduling of physical memory in a way similar to CPU time is problematic, because it can lead to inefficient memory use and is prone to cause thrashing.

6.3 Operating Systems

Language-based operating systems have existed for many years. Most of them were not designed to protect against malicious applications, although a number of them support strong security features. None of them, however, provide strong resource controls. More recent systems use language-based mechanisms for kernel extensions; in these systems, the kernel becomes an instance of the type of runtime environment for which KaffeOS was designed.

6.3.1 Early Single-Language Systems

Pilot: The Pilot Operating System [64] and the Cedar [76] Programming Environment were two of the earliest language-based systems. Their development at Xerox PARC in the 1980s predates a later flurry of research in the 1990s on such systems.

The Pilot OS and all of its applications were written in Mesa. Mesa's type-checking system guaranteed protection without needing MMU support. However,

Pilot's protection mechanisms were targeted at a single-user system, where programming mistakes are a more serious problem than maliciousness or attacks. Pilot did not provide resource management for its applications: resource shortages had to be either managed by the user, or more resources had to be added.

Oberon/Juice: Juice [38] provides an execution environment for downloaded Oberon code (just as a JVM provides an execution environment for Java). Oberon [89] has many of Java's features, such as garbage collection, object-orientation, strong type-checking, and dynamic binding. Unlike Java, Oberon is a nonpreemptive, single-threaded system. Its resource management policies are oriented towards a single-user workstation with interactive and background tasks. Background tasks such as the garbage collector are implemented as calls to procedures, where "interruption" can occur only between top-level procedure calls. Juice is a virtual machine that executes mobile code 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.

6.3.2 Single-Address-Space Systems

Some operating systems [18, 44, 66, 86] decouple the notions of virtual address space and protection domain. Instead, they exploit a single address space for all processes. Placing different processes in different protection domains allows the system to control which pages are accessible to a given process. This control is enforced using traditional hardware protection. This design allows virtual addresses to be used as unique descriptors for pages shared among processes. Such sharing is possible by allowing access to shared pages from multiple protection domains.

Since access to shared memory in Single-Address-Space (SAS) systems can be revoked, SAS systems do not suffer from Java's problem that handing out references to a shared object prevents its reclamation. They are therefore not susceptible to sharing attacks. However, because a type-safe language is not used, memory protection must be used when untrusted threads access a shared segment, which is not required when different KaffeOS processes access shared kernel objects.

Systems such as Angel [86], Mungi [44], or Opal [18] implement access to secondary storage by allowing shared objects to be made persistent. As a consequence, such systems have to manage the secondary storage resources used. This problem is very similar to the problem of managing shared objects in KaffeOS. Angel used a form of garbage collection to reclaim the address space and backing storage for objects, but it ignored the problem of charging for objects whose owners had died. Mungi introduced an economy-based model of period rent collection and bank accounts that would forcefully evict those objects for which no rent was paid. KaffeOS's shared heaps are not persistent: they can be collected after all sharers terminate.

6.3.3 Inferno

Inferno [31] is an operating system for building distributed services. Inferno applications are written in Limbo, which is a type-safe language that runs on a virtual machine called Dis. Inferno supports different tasks that can execute user code contained in modules. Different tasks communicate through channels that are supported by the Inferno kernel, which provide a facility for exchanging typed data.

Inferno's security model protects the security of communication and provides control over logical resources accessible to a task. As in PLAN 9 [60], all resources in Inferno are named and accessed as files in multiple hierarchical file systems. These file systems are mapped into a hierarchical namespace that is private to each task. A communication protocol called Styx is used to access these resources in a uniform fashion. By restricting which resources appear in a task's private namespace, Inferno can implement access restrictions to logical resources.

Inferno uses reference counting to reclaim most of its objects immediately once they become unreachable. Reference counting is used to minimize garbage collection pauses and because it reduces an application's memory footprint. An algorithm known as Very Concurrent Garbage Collection, or VCGC [45], is used to collect the cyclic garbage reference counting cannot reclaim. Unlike KaffeOS, Inferno does monitor the references a task acquires and may consequently be subject to sharing attacks. It does not implement memory controls. Inferno does not distinguish

between user and kernel mode; instead, user and system data objects are treated equally by the collector. This lack of a clear distinction between user and system objects caused problems: aside from making it impossible to forcefully terminate applications in all circumstances, it led to a system that was harder to debug [59].

6.3.4 Extensible Operating Systems

SPIN: SPIN [10] is an operating system kernel into which applications can download application-specific extensions. These extensions provide application-specific services; for instance, an in-kernel video server can take video packets directly from the network interface. Downloading extensions speeds up applications because they are able to directly access the kernel's interfaces without having to cross the red line to enter the kernel.

The SPIN kernel and its extensions are written in Modula-3, a type-safe language similar to Java. Like Java in KaffeOS, Modula-3's type safety provides memory safety for kernel extensions in SPIN. SPIN also supports dynamic interposition on names, so that extensions can have different name spaces.

To prevent denial-of-service attacks, certain procedures in an extension can be declared ephemeral, which implies that they cannot call procedures not declared as such. This declaration allows these procedures to be killed if they execute for too long but also restricts what they can do. In particular, an ephemeral procedure cannot call a procedure that is not itself declared ephemeral, which is why ephemeral procedures provide only a partial solution to the problem of termination.

SPIN does not control the amount of heap memory or other resources used by its extensions, and it cannot always safely unlink and unload an extension. KaffeOS's mechanisms could be applied to provide this functionality by defining a kernel within the SPIN kernel.

VINO: VINO [69] is an extensible operating system that allows for kernel extensions, which are called grafts, to be inserted in the kernel. Instead of employing a type safe language, VINO's grafts are written in C++ and are rewritten using MiSFIT [71], a static software-fault isolation tool, before being inserted into the kernel.

Grafts are subject to resource limits and can be safely aborted when they exceed them. Instead of using language-based mechanisms, VINO wraps grafts in transactions. For instance, all accesses to system state must be done through accessor methods that record state changes on an undo call stack. If the graft is aborted, these changes are undone. Transactions are a very effective but also very heavyweight mechanism. In addition, the authors report that SFI overhead associated with data-intensive grafts can be "irritating" [69].

CHAPTER 7

CONCLUSION

Before we summarize KaffeOS's contributions and conclude, we discuss some of the directions in which our work could be continued.

7.1 Future Work

Implementing KaffeOS's design in a higher-performing JVM would allow us to more accurately gauge its costs and benefits. Specifically, we would expect a more accurate picture of the write barrier overhead, as well as possible gains in scalability and performance for misbehaving applications.

More efficient text sharing: KaffeOS's efficiency could be improved by forgoing the use of class loaders for the reloading of system classes. Our implementation uses class loaders to provide each process with its own set of static variables for roughly one third of the runtime classes. It should be possible to build a mechanism into the JVM that would allow multiple logical instances of a class to be created, without duplicating that class's text or internal class representation. Each logical instance would have to have its own private store for all static variables. The text emitted by the just-in-time compiler would have to use indirection when compiling accesses to these variables, in the same way that a C compiler generates accesses to the data segment in position-independent code (PIC) for shared libraries in some versions of Unix.

More efficient write barrier implementation: Instead of outlining write barriers into separate functions, which requires the generation of a function call before the actual write, we could use inlined write barriers. Such inlining would be enabled by the use of a more sophisticated just-in-time compiler. The write barrier code could be expressed as statements in the intermediate representation language used by such

a compiler, before additional optimization steps are applied. These optimization passes would then automatically take the write barrier and its surrounding code in its entirety into account. The compiler could apply heuristics and other techniques to determine when it is more beneficial to outline the write barrier code, because inlining increases code size and therefore might not always increase performance.

Per-heap garbage collection strategies: Knowledge about the likely lifetime distributions of objects on different heaps could help improve the garbage collection mechanism used by a JVM based on KaffeOS's design. In particular, objects on the kernel heap are likely to have a different lifetime distribution than objects on user heaps. For instance, shared classes on the kernel heap and their associated static member data are likely to be long-lived. Such data include timezone descriptions and character set conversion tables, which are mostly read-only data that are easily shared by all processes. Consequently, a generational collector could immediately tenure such objects into an older generation.

Cooperative thread stack scanning and multiprocessor support: Our current prototype scans each thread stack in the system for each collector, which does not scale. In addition, our GC algorithm assumes a uniprocessor architecture, in which no other than the current thread runs during a given time quantum. The first problem could be solved by investigating ways for different collectors to cooperate when scanning thread stacks, while keeping the memory overhead associated with storing intermediate GC results small. The second problem, which is related, could be solved by applying multiprocessor garbage collection techniques.

7.1.1 Applying static checking.

One of KaffeOS's design properties is that state that is shared among different processes in the system must only be modified in kernel mode. However, we have no means of verifying that property in the general case. We can—and have—added checks to critical points in our code where we know we must execute in kernel mode, such as when allocating explicitly managed memory from the kernel heap. Unfortunately, this approach cannot be generalized, because it is not always clear whether a given piece of code must be run in kernel mode or not. For instance,

whether access to a static variable must be in kernel mode depends on whether a class is shared or reloaded. Similarly, the safety of acquiring a lock on a shared class in user mode depends on whether that lock is also used by kernel code elsewhere.

We believe that most of these properties could be checked statically by a tool, which we describe briefly. First, this tool would have to determine which classes could be safely shared and which must be reloaded. If there are linker constraints that influence this decision—as is the case in our implementation described in Section 4.2—then the tool needs to take those into account.

Second, once the classes are classified as shared or reloaded, the tools would need to classify all statements as to whether they must be run in kernel mode or not. If a statement must be run in kernel mode, we need to ensure that there is no user-mode execution path leading to it. We expect to be able to detect violations of the following properties:

- System code acquires shared locks only in kernel mode.
- Kernel code does not contend for locks that are visible to user code.
- Kernel mode is always exited before returning from a system call.
- Kernel code does not hold shared locks when performing an upcall to user-mode code.
- Native methods that allocate resources from an underlying operating system run in kernel mode.

In addition, knowledge about on which heap a thread executes could be included. This knowledge could be used to detect situations in which kernel code might accidentally return references to foreign heaps to user code, or other situations that could subsequently trigger a write barrier violation or lead to memory leaks.

Meta-level compilation (MC) [33] is a recently developed technique that allows for the detection of violations of system-specific properties such as those described. MC provides a language in which a user can write system-specific compiler extensions, which are invoked when the compiler encounters certain syntactic patterns.

These extensions can include knowledge about the semantics of the program that are not normally accessible to a compiler, such as whether a class is shared or reloaded or that `Kernel.enter()` enters kernel mode.

7.2 JanosVM Application

An ongoing project, JanosVM [78], already uses KaffeOS as its design and implementation base. JanosVM is a virtual machine targeted at supporting mobile Java code in active networks. JanosVM code is run in *teams*, which correspond to KaffeOS processes. Unlike KaffeOS, JanosVM is not intended as an environment in which to run applications directly. Instead, it is intended as a foundation on which to build execution environments for active code. Consequently, JanosVM both restricts KaffeOS's flexibility by specialization and extends its flexibility by providing trusted access to some internal kernel functionality.

JanosVM supports direct cross-heap references but does not use entry and exit items the way KaffeOS does. Instead, all cross-heap references are represented by proxies, which are referred to as *importables* and *exportables*. Importables and exportables are like entry and exit items, except that they are explicitly managed. The responsibility of managing falls on the designer who uses JanosVM to build an execution environment. The designer must guarantee that importables and exportables are used in a way that does not create illegal cross-heap references. Guaranteeing this requirement is hard and requires that the kernel be free of bugs. An advantage of explicitly managed cross-heap references is that heaps can be reclaimed immediately after a team terminates, as is the case in the J-Kernel [42]. This property may prove beneficial if the frequency with which teams are created and terminated is high.

KaffeOS's kernel boundary is static, but it also exports primitives for entering and leaving kernel mode to trusted parties. JanosVM makes use of this facility in a systematic way for interprocess communication between trusted parties. Threads can visit other teams, execute the code of those teams, and return to their home teams. A thread runs in kernel mode while visiting another team. System calls

can be viewed as visits to the kernel team. Consequently, a piece of code could run either in kernel mode or in user mode at different times and for different threads. It would run in kernel mode for a visiting team if it is used by a service that is deemed critical to the whole system; it would run in user mode if invoked as part of some other activity by the home team. This extension protects trusted servers without making them statically part of the kernel.

Because JanosVM is not intended to support general-purpose applications, shared heaps are not supported. Instead, JanosVM supports customized sharing for its application domain. For instance, teams can share packet buffers, which are objects with a well-known set of access primitives that are treated specially by the system. Consequently, some implementation complexity was removed, because the kernel garbage collector does not need to check for orphaned shared heaps.

JanosVM improves on KaffeOS's resource framework by including other resources in a uniform way. For instance, the namespace of a process is treated as a resource that can be managed just like the memory and CPU time used by a process. In KaffeOS, such policies would require a new process loader implementation.

7.3 Summary

In this dissertation, we have presented the design and an implementation of a Java runtime system that supports multiple, untrusted applications in a robust and efficient manner. The motivation for this work arose from the increasing use of type-safe languages, such as Java, in systems that execute untrusted code on behalf of different users. Although these systems already provide security frameworks that protect access to logical resources such as files, they do not support proper isolation between applications, safe termination of uncooperative or buggy applications, or the management and control of primary computational resources: memory and CPU time.

When these problems became apparent, researchers were divided: one camp considered the memory protection enabled by type safety, combined with the existing security model, to be sufficient to guarantee the safe execution of mobile

code, and believed that protection against resource-based denial-of-service attacks could be provided as an afterthought. The other camp did not believe that any new problems had arisen, because any and all of these problems could be attacked with existing operating system mechanisms by encapsulating the Java runtime as a whole into an operating system process.

Our approach uses arguments from both camps: instead of running applications in multiple operating system processes, we have used operating system ideas to introduce a process concept into the Java virtual machine. KaffeOS's processes provide the same properties as operating system processes: they protect and isolate applications from each other, and they manage and control the resources that they use. One motivation for using a single JVM for multiple applications is scalability, which results from efficient sharing of data between applications; another motivation is the potential to run on low-end hardware.

We have demonstrated that the red line abstraction, which separates kernel from user code, is necessary to protect critical parts of the system from corruption when applications are killed. Code that executes in kernel mode delays termination requests until it returns to user mode; in addition, it is written such that it can safely back out of exceptional situations. The red line, however, does not erect "thick walls" around applications: it does not prevent sharing, because it does not change the way in which objects access each other's fields and methods; type safety remains the means to enforce memory safety.

KaffeOS manages memory and CPU time consumed by its processes. For memory, each process has its own heap on which to allocate its objects and other data structures. A process's memory usage can be controlled by limiting its heap size. Separating CPU time requires that the time spent in garbage collection by each process can be accurately accounted for. For this reason, KaffeOS's heaps can be independently garbage collected, even in the presence of cross-heap references.

KaffeOS separates the resources used by different processes, but its resource management framework supports policies more flexible than simple partitioning. Soft memory limits allow multiple processes to share a common memory limit, so

that one process can use surplus memory that is not being used by other processes. Similarly, KaffeOS's use of a work-conserving proportional-share scheduler allows other processes to use a process's CPU time when that process is not ready to run.

To ensure full reclamation of a process's memory, KaffeOS prevents writes that could create references that could keep a process's objects alive after the process terminates. For that purpose, KaffeOS uses write barriers, which are a garbage collection technique. This use of write barriers prevents sharing attacks, in which a process could prevent objects from being reclaimed by collaborating with or duping another process into holding onto references to these objects.

KaffeOS adopts distributed garbage collection techniques to allow for the independent collection of each process's heap, even in the face of legal cross-references between the kernel heap and user heaps. The key idea is to keep track of the references that point into and out of each user heap. Incoming references are kept track of in entry items, which are treated as garbage collection roots. Outgoing references are kept track of in exit items, which allow for the reclamation of foreign entry items when their reference counts reach zero. These ideas were originally developed for situations in which objects cannot access other objects directly. KaffeOS does not prevent direct access but does use entry and exit items for resource control.

In addition to kernel-level sharing for increased efficiency, KaffeOS also supports user-level sharing by way of shared heaps. Shared heaps allow processes to share objects directly, which allows for efficient interprocess communication in KaffeOS. However, not to compromise full reclamation and accurate accounting, we had to restrict the programming model for user-level shared objects somewhat: a shared heap's size is frozen after creation, and write barriers prevent the creation references to objects in user heaps. We have shown that despite these restrictions, applications can conveniently share complex data types, such as hashmaps and linked lists.

This dissertation discusses both the design of KaffeOS (in Chapter 3) and an implementation of this design (in Chapter 4). In our implementation, we modified an existing single-processor, user-mode Java virtual machine to support KaffeOS's

functionality. Some implementation decisions, such as the use of class loaders for multiple namespaces or the specifics of how our garbage collector represents its per-heap data structures, were influenced by the infrastructure with which we worked. However, KaffeOS's design principles apply to other JVMs as well, which is why we discussed them without assuming a concrete underlying JVM.

The core ideas of KaffeOS's design are the use of a red line for safe termination, the logical separation of memory in heaps, the use of write barriers for resource control, the use of entry and exit items for separate garbage collection, and restricted direct sharing to support interprocess communication. We emphasize that these ideas do not have to be applied in their totality but are useful individually as well. For instance, a red line can provide safe termination even in JVMs that use a single heap, and write barriers can provide scoped memory as in Realtime Java even when those areas are not subject entry and exit items are not used for separate garbage collection.

Finally, although KaffeOS was designed for a Java runtime system, its techniques should also be applicable to other languages that exploit type safety to run multiple, untrusted applications in a single runtime system. Such environments could also profit from KaffeOS's demonstrated ability to manage primary resources efficiently and the defenses against denial-of-service attacks it provides.

APPENDIX A

ON SAFE TERMINATION

When a thread is terminated asynchronously—either because its process has exceeded a resource limit or because an external request to terminate its process has been issued—critical data structures must be left in a consistent state. There are two basic approaches to achieve this consistency: either a cleanup handler is invoked that detects and repairs inconsistencies (*roll back*) or termination is deferred during the manipulation of the data structures (*roll forward*). In Java, cleanup handlers would be provided as `catch` clauses.

An example demonstrates why using deferred termination should be preferred to exception handling facilities for guaranteeing consistency. Consider the simplified code fragment in Figure A.1; this code might be encountered in an application-specific packet gateway. A packet dispatcher thread sits in a loop, reads packets from an input queue, and dispatches them to an output queue. The critical invariant in this case shall be that every packet that is dequeued will be enqueued in the output queue, i.e., that no packets are lost if the dispatching thread is terminated.

```
class Packet;
class Queue {
    synchronized void enqueue(Packet p);
    synchronized Packet dequeue() throws QueueClosedException;
} inputQueue, outputQueue;

while (!finished) {
    outputQueue.enqueue(inputQueue.dequeue());
}
```

Figure A.1. Packet dispatcher example. A thread dispatches packets from an input queue onto an output queue. This example is properly synchronized and correct in the absence of asynchronous termination.

Figure A.2 shows a naive attempt to write this example using exception handling facilities. The calls to `outputQueue.enqueue()` and `inputQueue.dequeue()` are enclosed in a `try` block. If the thread is terminated while executing that `try` block, a `TerminationException` is thrown that is caught by the provided `catch` clause, which contains the cleanup handler. If we make the simplifying assumption that `enqueue` and `dequeue` do not need cleanup, the cleanup handler only has to determine whether a dequeued packet is pending, and enqueue the packet if so.

Unfortunately, this approach could yield incorrect results. According to the Java language specification [49], the evaluation of the expression `inputQueue.dequeue()` may complete abruptly, and leave the variable `p` unassigned, even though a packet was dequeued. Similarly, `p`, which is reset to `null` at the beginning of every iteration, could still refer to a packet that has already been enqueued and should not be enqueued again.

Changing the code such that the assignment to `p` is visible if and only if a packet was dequeued could be possible by making `p` a public field of `Queue` that is set atomically by the `enqueue` and `dequeue` methods. Even then, the non-null check in the cleanup handler does not work because of Java's flawed memory model [62]. If current proposals to fix the memory model are adopted, additional synchronization,

```
while (!finished) {
    Packet p = null;
    try {
        outputQueue.enqueue(p = inputQueue.dequeue());
    } catch (TerminationException _) {
        if (p != null) {
            outputQueue.enqueue(p);
        }
    }
}
```

Figure A.2. Naive approach to guaranteeing queue consistency using cleanup handlers. If a termination request is signaled via a `TerminationException`, the `catch` clause attempts to dispatch packets that have been dequeued, but not enqueued, so that no packets are lost. This naive approach does not work because it does not reliably detect when a dequeued packet must be enqueued.

combined with annotations such as `volatile`, might be required to ensure that the value of `p` that reflects the correct state of the input and output queues is visible to the cleanup handler.

If we reconsider the earlier assumption that `enqueue` and `dequeue` do not require cleanup, we recognize that the programmer of these methods is likely to face similar problems as the programmer of the packet dispatcher routine. In particular, if the delivery of a `TerminationException` interrupted the execution of `outputQueue.enqueue()`, the queue must not be left in an inconsistent state that would prevent the subsequent invocation of the `enqueue` method in the cleanup handler. Guaranteeing this property involves similar considerations and the addition of cleanup handlers inside `enqueue` and `dequeue`. Consequently, as the amount of state manipulated by a thread grows, so does the complexity of the code needed to repair that state if an asynchronous exception can terminate a thread.

Figure A.3 shows how this example would be rewritten using deferred termination. Termination requests are deferred when the kernel is entered. The dispatcher thread is informed of a pending termination request through a flag that it polls. If the flag is set, the dispatcher thread can terminate when the queues are in a consistent state. This version does not suffer from the potential for errors that the exception-based version does, which is why we chose that approach in KaffeOS.

```
Kernel.enter();
try {
    while (!finished && !Kernel.terminationPending) {
        outputQueue.enqueue(inputQueue.dequeue());
    }
} finally {
    Kernel.leave();
}
```

Figure A.3. Packet dispatch using deferred termination. Because termination requests are not acted upon while code is executing in a `Kernel.enter()`/`Kernel.leave()` section, the invariant that all dequeued packets from the input queue are enqueued in the output queue is guaranteed. The `finally` clause ensures that kernel mode is left even if a `QueueClosedException` is thrown.

APPENDIX B

SERVLET MICROENGINE

This appendix provides an in-depth discussion of the servlet microengine example presented in Section 5.4.3. The example consists of four source code files, `HttpServerQueue.java`, `DictionaryServer.java`, `GermanEnglishDictionary.java`, and `DictionaryServlet.java`, each of which is discussed in a separate section.

B.1 HttpServerQueue.java

The `shared.HttpServerQueue` class provides a queue for http requests that is shared between server and servlet processes. Each request is stored as an object of type `HttpServerQueue.Request` (lines 11–30). A client request is read into a fixed-size buffer of 2048 bytes, which is preallocated in the constructor (lines 23–29). The constructor also preallocates entry items to enable the subsequent storage of references to the buffer in user-heap objects.

The request object includes an object of type `kaffeos.sys.FD`, which contains a file descriptor for the connection to the client. The `Request.setConnection()` (lines 18–20) and the `Request.getConnection()` methods (line 17) are used to store the file descriptor into the request object and retrieve it from there. `FD.dupInto` is a kernel interface that we added to transfer file descriptors between processes.

The `HttpServerQueue()` constructor (line 39–47) allocates n queues for requests that are dispatched to servlets and one queue for "empty" requests that are not currently in use. It uses the `shared.util.LinkedList` class discussed in Section 5.4.3 for that purpose. The methods `getEmptyRequest()` (lines 53–61), `putRequest()` (lines 67–72), `getNextRequest()` (lines 78–86), and `putEmptyRequest()` (lines 92–97) are wrappers that provide mutual exclusion and unilateral synchronization

for the underlying request queues. These methods are invoked in the following order:

1. The server calls `getEmptyRequest()` to dequeue an empty request. It then accepts a new client and reads the first bytes into the request buffer, which contain the client's request.

2. The server identifies the request and calls `putRequest()` to dispatch the request to a waiting servlet, which is woken up after being notified of the new request.

3. The dispatched servlet calls `getNextRequest()` to obtain the next request from the queue and processes the request.

4. The servlet recycles the request by calling `putEmptyRequest()`.

```
1  package shared;
2  import shared.util.LinkedList;
3  import kaffeos.sys.FD;
4
5  /**
6   * HttpServerQueue allows http servers to share
7   * request queues with servlets
8   */
9  public class HttpServerQueue {
10
11     public static class Request {
12         private byte    []buf;     // first 2048 of data read from client
13         private FD      fd;        // preallocated file descriptor
14
15         public int getCapacity()  { return (buf.length); }
16         public byte[] getBuffer() { return (buf); }
17         public FD getConnection() { return (fd); }
18         public void setConnection(FD fd) {
19             this.fd.dupInto(fd);
20         }
21
22         /* constructor preallocates objects while shared heap is populated */
23         Request() {
24             this.buf = new byte[2048];
25             this.fd = new FD();
26             kaffeos.sys.Heap h = kaffeos.sys.Heap.getCurrentHeap();
27             h.reserveEntryItem(buf);
28             h.reserveEntryItem(fd);
```

```
29          }
30      }
31
32      public final int NMAX = 20;    // max number of entries per queue
33      public final int NQUEUES = 2;  // number of queues
34      private shared.util.LinkedList freeQueue, requestQueues[];
35
36      /**
37       * preallocate sufficient number of requests for queues
38       */
39      HttpServerQueue() {
40          requestQueues = new LinkedList[NQUEUES];
41          for (int i = 0; i < NQUEUES; i++)
42              requestQueues[i] = new LinkedList(NMAX);
43
44          freeQueue = new LinkedList(NMAX * NQUEUES);
45          for (int i = 0; i < NMAX * NQUEUES; i++)
46              freeQueue.add(new Request());
47      }
48
49      /**
50       * get empty Request object to use for server
51       * (called by server)
52       */
53      public Request getEmptyRequest() throws InterruptedException {
54          Request thisreq;
55          synchronized (freeQueue) {
56              while (freeQueue.isEmpty())
57                  freeQueue.wait();
58              thisreq = (Request)freeQueue.removeFirst();
59          }
60          return (thisreq);
61      }
62
63      /**
64       * put Request object in queue for servlet
65       * (called by server)
66       */
67      public void putRequest(Request thisreq, int nr) {
68          synchronized(requestQueues[nr]) {
69              requestQueues[nr].addLast(thisreq);
70              requestQueues[nr].notify();
71          }
72      }
73
74      /**
75       * get next Request object from queue
76       * (called by servlet)
77       */
78      public Request getNextRequest(int nr) throws InterruptedException {
79          Request req;
80          synchronized (requestQueues[nr]) {
```

```
81              while (requestQueues[nr].isEmpty())
82                  requestQueues[nr].wait();
83              req = (Request)requestQueues[nr].removeFirst();
84          }
85          return (req);
86      }
87
88      /**
89       * put now empty Request object back into free queue
90       * (called by servlet)
91       */
92      public void putEmptyRequest(Request req) {
93          synchronized (freeQueue) {
94              freeQueue.add(req);
95              freeQueue.notify();
96          }
97      }
98  }
```

B.2 DictionaryServer.java

The `shared.DictionaryServer` class implements the server process in our microengine example. Its `main()` method marks the point at which execution starts. It first creates a shared request queue on a new shared heap (line 17). After registering the `shared.HttpServerQueue` class, the class is mapped into the server process's namespace. The server can then directly refer to the shared queue instance after casting it into an instance of type `HttpServerQueue` (line 22), which demonstrates the seamlessness with which a KaffeOS process can access shared objects.

The server process starts the servlet processes (lines 23–43). The servlet processes use the same classes and provide the same functionality; a number passed as a command line argument to each servlet is used for identification (lines 35–36). The server creates a socket to listen on port 8080 (line 45) and loops while handling requests (lines 47–96). The server obtains an empty request object (line 51) and reads the data received from the client into it (lines 68–74). An anonymous inner class derived from `java.io.BufferedInputStream` (lines 54–62) provides a buffered input stream based on a user-provided buffer. Because objects allocated on a shared heap can be used seamlessly within a KaffeOS process, we are able to direct the buffered input stream to directly use the shared `byte[]` array stored in

the `HttpServerQueue.Request.buf` field (line 62).

The request is parsed (lines 75–77) and passed onto the servlet (lines 83–87). The `java.net.Socket.getFD()` method, which we added to the `java.net.Socket` class, returns a reference to a socket's underlying `kaffeos.sys.FD` object, which represents a file descriptor in the KaffeOS kernel. This file descriptor is duplicated via `HttpServerQueue.Request.setConnection()`, which is discussed in Section B.1, before the request is dispatched to the servlet (line 80).

```
1  import kaffeos.sys.*;
2  import kaffeos.resource.*;
3  import kaffeos.sys.Process;
4  import shared.HttpServerQueue;
5  import shared.HttpServerQueue.Request;
6  import java.lang.reflect.*;
7  import java.io.*;
8  import java.util.*;
9  import java.net.*;
10
11 /**
12  * micro http server for dictionary servlets
13  */
14 public class DictionaryServer {
15
16     public static void main(String []av) throws Throwable {
17         Shared sh = Shared.registerClass("shared.HttpServerQueue", 1);
18         serveRequests(sh);
19     }
20
21     public static void serveRequests(Shared sh) throws Throwable {
22         HttpServerQueue ds = (HttpServerQueue)sh.getObject(0);
23         ProcessHandle h1, h2;
24         Hashtable r1, r2;
25         MemResource mr = Process.getHeap().getMemResource();
26         CpuResource mc = Process.getCpuResource();
27
28         r1 = new Hashtable();
29         r2 = new Hashtable();
30         r1.put(MemResource.MEMRES, mr.createSoftReserve(mr.getLimit()));
31         r2.put(MemResource.MEMRES, mr.createSoftReserve(mr.getLimit()));
32         r1.put(CpuResource.CPURES, mc.createChildReserve("Servlet #0", 0.2f));
33         r2.put(CpuResource.CPURES, mc.createChildReserve("Servlet #1", 0.5f));
34
35         Object [] firstchild = new Object[] { new String[] { "0" } };
36         Object [] secondchild = new Object[] { new String[] { "1" } };
37         Properties prop = System.getProperties();
38         h1 = Process.newProcess("DictionaryServlet", firstchild, prop, r1);
39         h2 = Process.newProcess("DictionaryServlet", secondchild, prop, r2);
```

```
40
41          h1.start();
42          Thread.sleep(3000);
43          h2.start();
44
45          ServerSocket s = new ServerSocket(8080);
46          try {
47              while (true) {
48                  Socket conn = s.accept();
49                  try {
50                      // get a new fresh request buffer
51                      Request thisreq = ds.getEmptyRequest();
52
53                      // create buffered input stream on top of request buffer
54                      BufferedInputStream ps =
55                          new BufferedInputStream(conn.getInputStream(),
56                                                  thisreq.getCapacity())
57                          {
58                              BufferedInputStream setBuf(byte []buf) {
59                                  this.buf = buf;
60                                  return (this);
61                              }
62                          }.setBuf(thisreq.getBuffer());
63
64                      // make sure buf is not skipped
65                      ps.mark(thisreq.getCapacity());
66
67                      // read request
68                      InputStreamReader in = new InputStreamReader(ps);
69                      StringBuffer sb = new StringBuffer(80);
70                      for (;;) {
71                          int r = in.read();
72                          if (r == '\r' || r == '\n' || r == -1) break;
73                          sb.append((char)r);
74                      }
75                      StringTokenizer st = new StringTokenizer(sb.toString());
76                      String method = st.nextToken();
77                      String request = st.nextToken();
78
79                      if (method.equalsIgnoreCase("GET")) {
80                          thisreq.setConnection(conn.getFD());
81
82                          // dispatch to servlet based on "s=" argument
83                          if (request.indexOf("s=0&") > -1) {
84                              ds.putRequest(thisreq, 0);
85                          } else {
86                              ds.putRequest(thisreq, 1);
87                          }
88                      }
89                  } catch (Exception e) {
90                      e.printStackTrace(System.err);
91                  } finally {
```

```
92                      if (conn != null) {
93                          conn.close();
94                      }
95                  }
96              }
97          } catch (Exception e) {
98              e.printStackTrace(System.err);
99          }
100     }
101 }
```

B.3 GermanEnglishDictionary.java

The `shared.GermanEnglishDictionary` class implements a simple hashmap-based dictionary. The class provides a wrapper around an object of type `shared.util.HashMap`, which is discussed in Section 5.4.2. The constructor, which is executed while the shared heap is populated, reads pairs of words from a file and adds entries to the hashmap. The `lookup()` method (lines 38–40) and the `size()` method (lines 45–47) provide wrappers to access the `get()` and `size()` methods of the underlying hashmap. The use of a default constructor (lines 31–33) in this example is not only because it is convenient, but also because our current KaffeOS implementation does not support the passing of arguments from a user heap to constructors of objects allocated on a shared heap. Argument passing could be implemented in the same way certain arguments to system calls are deep copied, which is described in Section 4.1.

```
1  package shared;
2
3  import java.io.*;
4  import shared.util.*;
5  import java.util.Iterator;
6  import java.util.NoSuchElementException;
7  import java.util.StringTokenizer;
8  import java.util.Map;
9
10 /**
11  * Shared class that provides a German/English dictionary
12  */
13 public class GermanEnglishDictionary {
14     private shared.util.HashMap         hmap;
15
16     /**
17      * Construct dictionary from input file
```

```
18       * This constructor is executed while the shared heap is created.
19       */
20      GermanEnglishDictionary(Reader r) throws IOException {
21          LineNumberReader ln = new LineNumberReader(r);
22          String l = null;
23          hmap = new HashMap();
24          while ((l = ln.readLine()) != null) {
25              if (l.startsWith("#")) continue;
26              StringTokenizer s = new StringTokenizer(l, "\t");
27              hmap.put(s.nextToken(), s.nextToken());
28          }
29      }
30
31      GermanEnglishDictionary() throws IOException {
32          this(new FileReader("German.txt"));
33      }
34
35      /**
36       * provide lookup operation based on shared hashmap
37       */
38      public String lookup(String s) {
39          return ((String)hmap.get(s));
40      }
41
42      /**
43       * report number of entries
44       */
45      public int size() {
46          return (hmap.size());
47      }
48  }
```

B.4 DictionaryServlet.java

The `shared.DictionaryServlet` class implements the servlet process in our microengine example. The `DictionaryServer` server class starts two instances of this class. Both instances look up the shared http server queue created by the server (line 18). They also share an instance of type `shared.GermanEnglishDictionary` (line 19), which is discussed in Section B.3. The serve() method contains the servlet loop (lines 31–82). A servlet waits for the next request on the queue to which it was assigned (line 33). Because server and servlet share only the bare byte buffer in our example, the servlet needs to repeat some of the work done by the server to parse the request (lines 35–41). (To avoid this replication of work, we could have shared the parsed components of the request in `char []` arrays, which would have

complicated the example.)

The servlet extracts the word that is to be looked up (line 43–44) and sends a reply to the client (lines 47–78). Note that the shared file descriptor in the request object is directly used to construct a `java.io.FileOutputStream` object (lines 47–49) that is subsequently passed to a regular `java.io.PrintWriter(OutputStream)` constructor, which again demonstrates the ease with which KaffeOS processes make use of shared objects. Once the servlet has processed the request and sent the reply, it recycles the request object (line 81).

```
1  import kaffeos.sys.*;
2  import kaffeos.sys.Process;
3  import kaffeos.resource.CpuResource;
4  import shared.HttpServerQueue;
5  import shared.HttpServerQueue.Request;
6  import shared.GermanEnglishDictionary;
7  import java.io.*;
8  import java.util.*;
9  import java.lang.reflect.*;
10
11 /**
12  * micro-servlet class for dictionary example
13  */
14 public class DictionaryServlet {
15
16     /* initialize shared data structures */
17     public static void main(String []av) throws Throwable {
18         Shared sh = Shared.registerClass("shared.HttpServerQueue", 1);
19         Shared she = Shared.registerClass("shared.GermanEnglishDictionary", 1);
20         serve(sh, she, Integer.parseInt(av[0]));
21     }
22
23     /**
24      * main servlet loop
25      */
26     public static void serve(Shared shqueue, Shared shdict, int queuenr) {
27         HttpServerQueue ds = (HttpServerQueue)shqueue.getObject(0);
28         GermanEnglishDictionary dict = (GermanEnglishDictionary)shdict.getObject(0);
29
30         try {
31             while (true) {
32                 // get next request from queue
33                 Request req = ds.getNextRequest(queuenr);
34
35                 BufferedReader br =
36                     new BufferedReader(
37                         new InputStreamReader(
```

```
38                          new ByteArrayInputStream(req.getBuffer())));
39                  StringTokenizer st = new StringTokenizer(br.readLine());
40                  st.nextToken();                 // skip GET command.
41                  String request = st.nextToken();
42
43                  int wi = request.indexOf("w=");
44                  String w = wi > 0 ? request.substring(wi + 2) : null;
45
46                  // construct regular print writer from shared fd
47                  PrintWriter p = new PrintWriter(
48                      new FileOutputStream(
49                          new FileDescriptor(req.getConnection())));
50
51                  p.print("HTTP/1.0 200\r\n"
52                        + "Content-Type: text/html\r\n\r\n"
53                        + "<html><body bgcolor=white><h3>");
54                  if (w != null) {
55                      p.println("English: " + w);
56                      String g = dict.lookup(w);
57                      if (g != null) {
58                          p.println("<br>German: " + g);
59                      } else {
60                          p.println("<br><font color=red>"
61                                  + "is not in dictionary</font>");
62                      }
63                  }
64                  p.println("</h3>There are " + dict.size()
65                          + " entries in the dictionary.<br><form>"
66                          + "Please choose a servlet: <select name=s>");
67                  for (int i = 0; i < 2; i++) {
68                      p.println("<option value=" + i + " "
69                          + (queuenr == i ? "selected" : "") + ">");
70                      p.println("Servlet #" + i + "</option>");
71                  }
72                  p.println("</select>"
73                          + "<br>Please enter a word: "
74                          + "<input name=w type=text></form>"
75                          + "<p>You have been served by <i>"
76                          + Kernel.currentProcess().getCpuResource()
77                          + "</i></body></html>");
78                  p.close();
79
80                  // done with request, return to shared queue
81                  ds.putEmptyRequest(req);
82              }
83          } catch (Exception _) {
84              _.printStackTrace(System.err);
85          }
86      }
87  }
```

REFERENCES

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference* (Atlanta, GA, June 1986), pp. 93–112.

[2] Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada, May 1998), ACM, pp. 258–268.

[3] Baker, H. G. The treadmill, real-time garbage collection without motion sickness. *ACM Sigplan Notices 27*, 3 (March 1992), 66–70.

[4] Balfanz, D., and Gong, L. Experience with secure multi-processing in Java. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (Amsterdam, Netherlands, May 1998), pp. 398–405.

[5] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 45–58.

[6] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), ACM, pp. 117–128.

[7] Bergsten, H. *JavaServer Pages*, 1st ed. O'Reilly & Associates, Inc., Sebastopol, CA, December 2000.

[8] Bernadat, P., Lambright, D., and Travostino, F. Towards a resource-safe Java for service guarantees in uncooperative environments. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, December 1998), pp. 101–111.

[9] Bershad, B., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., and Sirer, E. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), pp. 62–65.

[10] Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 267–284.

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

[12] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39–59.

[13] Bishop, P. B. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, May 1977.

[14] Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., and Turnbull, M. *The Real-Time Specification for Java*, 1st ed. The Java Series. Addison-Wesley, Reading, MA, January 2000.

[15] Bothner, P. Kawa—compiling dynamic languages to the Java VM. In *Proceedings of the USENIX 1998 Technical Conference, FREENIX Track* (New Orleans, LA, June 1998), USENIX Association, pp. 261–272.

[16] Chan, P., and Lee, R. *The Java Class Libraries: Volume 2*, 2nd ed. The Java Series. Addison-Wesley, October 1997.

[17] Chan, P., Lee, R., and Kramer, D. *The Java Class Libraries: Volume 1*, 2nd ed. The Java Series. Addison-Wesley, March 1998.

[18] Chase, J. S. *An Operating System Structure for Wide-Address Architectures.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, August 1995.

[19] Cheng, P., Harper, R., and Lee, P. Generational stack collection and profile-driven pretenuring. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 162–173.

[20] Cheriton, D. R., and Duda, K. J. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (Monterey, CA, November 1994), pp. 179–193.

[21] Czajkowski, G. Application isolation in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000), pp. 354–366.

[22] Czajkowski, G., Chang, C.-C., Hawblitzel, C., Hu, D., and von Eicken, T. Resource management for extensible Internet servers. In *Proceedings of the Eighth ACM SIGOPS European Workshop* (Sintra, Portugal, September 1998), pp. 33–39.

[23] Czajkowski, G., and Daynès, L. Multitasking without compromise: A virtual machine evolution. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)* (Tampa Bay, FL, October 2001), pp. 125–138.

[24] Czajkowski, G., Daynès, L., and Wolczko, M. Automated and portable native code isolation. Tech. Rep. 01-96, Sun Microsystems Laboratories, April 2001.

[25] Czajkowski, G., and von Eicken, T. JRes: a resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)* (Vancouver, BC, October 1998), pp. 21–35.

[26] Dahm, M. Byte code engineering. In *JIT '99 - Java-Informations-Tage (Tagungsband)* (Düsseldorf, Germany, September 1999), C. H. Cap, Ed., Springer-Verlag, pp. 267–277.

[27] Dieckmann, S., and Hölzle, U. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)* (Lisbon, Portugal, June 1999), Lecture Notes in Computer Science 1628, Springer-Verlag, pp. 92–115.

[28] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21*, 11 (November 1978), 966–975.

[29] Dillenberger, D., Bordawekar, R., Clark, C. W., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M. F., Samuel, F., and John, R. W. S. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal 39*, 1 (2000), 194–210.

[30] Diwan, A., Moss, E., and Hudson, R. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation* (Las Vegas, NV, June 1992), ACM, pp. 273–282.

[31] Dorward, S., Pike, R., Presotto, D. L., Ritchie, D. M., Trickey, H., and Winterbottom, P. The Inferno operating system. *Bell Labs Technical Journal 2*, 1 (1997), 5–18.

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

[33] Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000), USENIX Association, pp. 1–16.

[34] Engler, D. R., Kaashoek, M. F., and O'Toole Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 251–266.

[35] Farkas, K., Flinn, J., Back, G., Grunwald, D., and Anderson, J. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the 2000 Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, June 2000), pp. 252–263.

[36] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 137–151.

[37] Ford, B., and Susarla, S. CPU inheritance scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 91–105.

[38] Franz, M. Beyond Java: An infrastructure for high-performance mobile code on the World Wide Web. In *Proceedings of WebNet '97, World Conference of the WWW, Internet, and Intranet* (Toronto, Canada, October 1997), S. Lobodzinski and I. Tomek, Eds., Association for the Advancement of Computing in Education, pp. 33–38.

[39] Gay, D., and Aiken, A. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 313–323.

[40] Gorrie, L. Echidna — A free multiprocess system in Java. `http://www.javagroup.org/echidna/`.

[41] Hansen, P. B. Java's insecure parallelism. *SIGPLAN Notices 34*, 4 (April 1999), 38–45.

[42] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, LA, June 1998), pp. 259–270.

[43] Hawblitzel, C. K. *Adding Operating System Structure to Language-Based Protection*. PhD thesis, Department of Computer Science, Cornell University, June 2000.

[44] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. The Mungi single-address-space operating system. *Software Practice and Experience 28*, 9 (1998), 901–928.

[45] Huelsbergen, L., and Winterbottom, P. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the International Symposium on Memory Management* (Vancouver, BC, October 1998), ACM, pp. 166–175.

[46] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1996. Std 1003.1, 1996 edition.

[47] Jaeger, T., Liedtke, J., and Islam, N. Operating system protection for fine-grained programs. In *Proceedings of the Seventh USENIX Security Symposium* (San Antonio, TX, January 1998), pp. 143–157.

[48] Java Apache Project. The Apache JServ project. `http://java.apache.org/jserv`, April 2000.

[49] Joy, B., Steele, G., Gosling, J., and Bracha, G. *The Java Language Specification*, 2nd ed. The Java Series. Addison-Wesley, June 2000.

[50] Jul, E., Levy, H., Hutchison, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[51] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[52] Lepreau, J., Hibler, M., Ford, B., and Law, J. In-kernel servers on Mach 3.0: Implementation and performance. In *Proceedings of the Third USENIX Mach Symposium* (Santa Fe, NM, April 1993), pp. 39–55.

[53] Liang, S. *The Java Native Interface: Programmer's Guide and Specification*, 1st ed. The Java Series. Addison-Wesley, June 1999.

[54] Liang, S., and Bracha, G. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)* (Vancouver, BC, October 1998), pp. 36–44.

[55] Lizt, J. Oracle JServer Scalability and Performance. `http://www.oracle.com/java/scalability/index.html?testresults_twp.html`, July 1999. Java Products Team, Oracle Server Technologies.

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

[57] Marlow, S., Jones, S. P., Moran, A., and Reppy, J. Asynchronous exceptions in haskell. In *Conference on Programming Language Design and Implementation* (Snowbird, UT, June 2001), ACM, pp. 274–285.

[58] McGraw, G., and Felten, E. *Java Security: Hostile Applets, Holes, and Antidotes*. Wiley Computer Publishing. John Wiley and Sons, New York, NY, January 1997.

[59] Pike, R. Personal Communication, April 2000.

[60] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan 9 from Bell Labs. *Computing Systems 8*, 3 (Summer 1995), 221–254.

[61] Plainfossé, D., and Shapiro, M. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM95)* (Kinross, Scotland, September 1995), Lecture Notes in Computer Science 986, Springer-Verlag, pp. 211–249.

[62] Pugh, W. The Java memory model is fatally flawed. *Concurrency: Practice and Experience 12*, 6 (May 2000), 445–455.

[63] Radhakrishnan, R., Rubio, J., and John, L. K. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. In *Proceedings of IEEE International Conference on Computer Design* (Austin, TX, October 1999), pp. 281–284.

[64] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An operating system for a personal computer. *Communications of the ACM 23*, 2 (1980), 81–92.

[65] Rivest, R. The MD5 message-digest algorithm. Internet Request For Comments RFC 1321, Internet Network Working Group, April 1992.

[66] Roscoe, T. *The Structure of a Multi-Service Operating System*. PhD thesis, Queen's College, University of Cambridge, UK, April 1995.

[67] Rudys, A., Clements, J., and Wallach, D. S. Termination in language-based systems. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS '01)* (San Diego, CA, February 2001), pp. 175–187.

[68] Saulpaugh, T., and Mirho, C. A. *Inside the JavaOS Operating System*. The Java Series. Addison-Wesley, January 1999.

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

[70] Sirer, E., Fiuczynski, M., Pardyak, P., and Bershad, B. Safe dynamic linking in an extensible operating system. In *First Workshop on Compiler Support for System Software* (Tucson, AZ, February 1996), pp. 141–148.

[71] Small, C. MiSFIT: a minimal i386 software fault isolation tool. Tech. Rep. TR-07-96, Harvard University Computer Science, 1996.

[72] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.

[73] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal 39*, 1 (2000), 175–193.

[74] Sullivan, D. G., and Seltzer, M. I. Isolation with flexibility: a resource management framework for central servers. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 337–350.

[75] Sun Microsystems, Inc. Why Are `Thread.stop`, `Thread.suspend`, `Thread.resume` and `Runtime.runFinalizersOnExit` Deprecated? `http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html`.

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

[77] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., and Minden, G. J. A survey of active network research. *IEEE Communications Magazine 35*, 1 (January 1997), 80–86.

[78] Tullmann, P., Hibler, M., and Lepreau, J. Janos: A Java-oriented OS for active network nodes. *IEEE Journal on Selected Areas in Communications 19*, 3 (March 2001), 501–510.

[79] Tullmann, P., and Lepreau, J. Nested Java processes: OS structure for mobile code. In *Proceedings of the Eighth ACM SIGOPS European Workshop* (Sintra, Portugal, September 1998), pp. 111–117.

[80] Tullmann, P. A. The Alta operating system. Master's thesis, Department of Computer Science, University of Utah, 1999.

[81] van Doorn, L. A secure Java virtual machine. In *Proceedings of the Ninth USENIX Security Symposium* (Denver, CO, August 2000), pp. 19–34.

[82] van Doorn, L., Homburg, P., and Tanenbaum, A. S. Paramecium: an extensible object-based kernel. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), IEEE Computer Society, pp. 86–89.

[83] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 203–216.

[84] Waldspurger, C. A. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1995.

[85] Wilkinson, T. Kaffe – a Java virtual machine. `http://www.kaffe.org/`.

[86] Wilkinson, T., Stiemerling, T., Gull, A., Whitcroft, A., Osmon, P., Saulsbury, A., and Kelly, P. Angel: a proposed multiprocessor operating system kernel. In *Proceedings of the European Workshop on Parallel Computing* (Barcelona, Spain, 1992), pp. 316–319.

[87] Wilson, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM92)* (St. Malo, France, September 1992), Y. Bekkers and J. Cohen, Eds., Lecture Notes in Computer Science 637, Springer-Verlag, pp. 1–42.

[88] Wind River Systems, Inc. *VxWorks Programmer's Guide.* Alameda, CA, 1995.

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