Drawing the Red Line in Java

Godmar Back and Wilson Hsieh
Department of Computer Science
University of Utah
{gback,wilson}@cs.utah.edu

Abstract

Software-based protection has become a viable alternative to hardware-based protection in systems based on languages such as Java, but the absence of hardware mechanisms for protection has been coupled with an absence of a user/kernel boundary. We show why such a "red line" must be present in order for a Java virtual machine to be as effective and as reliable as an operating system. We discuss how the red line can be implemented using software mechanisms, and explain the ones we use in the Java system that we are building.

1. Introduction

A paper that appeared at a previous HotOS [4] stated that "protection is a software issue." This statement is incomplete; we would reword it as "Protection is a software issue, but it is not the only software issue." In particular, issues such as resource control, communication, and termination need to be dealt with in software if hardware protection mechanisms are not present. To date, systems that replace hardware mechanisms with software mechanisms (such as type safety in Java [13]) have neglected these other issues. As a result, these systems—as exemplified by Java—are not as useful or robust as operating systems. Our goal is to support applications that require the simultaneous execution of multiple programs, such as running mobile code in a browser or untrusted extensions in a database.

Processes use trap instructions to cross the "red line" [6] that separates user processes and the kernel. This red line not only enforces protection, but the separation between user and kernel mode also enables the implementation of process termination, the enforcement of resource controls, and safe inter-process communication. We claim that systems that use software mechanisms for protection (such as Java) must retain the notion of a red line if they replace hardware-based protection mechanisms with software-based ones. We use Java as our motivating example, but our discussion applies to more than just Java. Our arguments are applicable to both operating systems that use type-safe languages for extensibility [5] and language runtime systems that are evolving towards operating systems [12, 18]. We show how the failure of Java to enforce a distinction between user and kernel results in non-robust systems. Finally, we examine how the "red line" can be implemented in Java.

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Dept. of the Army under contract DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

2. The Red Line

Figure 1 illustrates how the traditional red line separates user mode and kernel mode in terms of four important (and interdependent) system functions: protection, resource control, safe inter-process communication, and termination.

We assume a model in which processes executing in user mode enter and leave kernel mode via these trap instructions. Furthermore, kernel code is not just arbitrary code: it is trusted. We do not discuss processes that only execute in kernel mode, although our conclusions also apply to them.

2.1. Protection

The primary function of the red line in a traditional, hardware-based system is enforcing protection. A process executing in user mode may not access any memory or I/O ports outside its address space. In addition, it does not have access to the privileged instructions needed to change

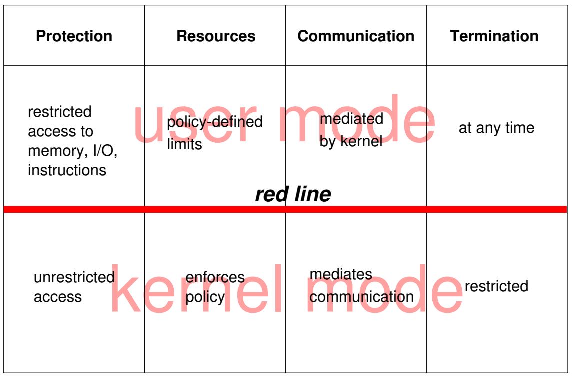

Figure 1. The "red line" in traditional operating systems.

those settings. Therefore, the kernel and other processes are protected from buggy or malicious applications.

Protection in Java does not require the user/kernel distinction because Java is type-safe. We assume that Java's type system is sound and that the bytecode verifier functions properly. Type safety provides protection directly, because processes cannot write to arbitrary locations in memory. Although protection is an important function of the red line, the others that we have mentioned are equally important. Unfortunately, the use of type safety for protection has obscured the need for a red line.

2.2. Resource Control

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, user processes are subject to limits on how many processor cycles they are allowed to use before they are preempted. Similar restrictions apply to the use of memory: a process may only use the memory provided to it by the kernel.

Changes to resource allocations are done by entering the kernel. For example, the kernel can change the amount of memory provided to a process when the process executes an `sbrk` system call. Some operating systems (such as the exokernel [10]) 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 and cause that process to be terminated, then the kernel must be structured so that the resource control violation does not cause an abrupt termination that endangers the integrity of the kernel itself.

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. The latter restriction is analogous to not allowing page fault handlers to fault themselves. For example, the exception handling code in the Java runtime must not depend on the ability to acquire heap memory, since the inability to acquire memory is signaled via an exception. The developers of Java failed to separate those parts of the system that should be subjected to policy-defined resource limits, and those that should always have access to all of the physical resources. In other words, the lack of a red line makes it difficult to build a robust system.

In addition to limiting the resources required by the kernel, it must be understood how the garbage collector interacts with the red line. In standard Java, the garbage collector is considered to be below the red line, since it is a shared system service. Unfortunately, if the entire garbage collector is in the kernel, it is difficult to properly account for the GC pressure caused by any particular process. As we discuss in Section 3, appropriate parts of the garbage collector should exist above the red line (in user space), so that time spent in the kernel is minimized.

2.3. Communication

The kernel must ensure that all communication mechanisms are subject to the constraints imposed by protection, resource control, and termination. For example, the kernel must bound IPC port queues to ensure that one process does not deny service to another process's communications. Similarly, the kernel must do bookkeeping on capabilities to ensure that a process does not acquire communication rights that it should not have.

We focus on data sharing as the primary communication mechanism of concern, because of its pervasiveness in systems such as Java. Many operating systems provide data sharing in the form of shared memory or memory-mapped files. In the case of `mmap'ed` files, the kernel provides memory that is shared between processes. The red line must be crossed to establish the memory mappings into the process's address space, so as to ensure that processes cannot violate any protection guarantees.

In Java, the lack of a red line causes problems with data sharing. In particular, uncontrolled sharing causes problems in two areas: resource control (memory accounting) and termination. Memory accounting is difficult with uncontrolled sharing because it is not clear how to account for an object that is shared by multiple user processes. Termination is complicated because if other processes keep alive pointers to objects that belong to a process, those objects may not be reclaimed when that process ends, which undermines resource control.

It is important (and difficult) in Java to ensure that the kernel be designed to not expose critical objects to user processes. For example, Java allows any object to be used as a monitor. If the kernel handed out references to an internal thread control block, then a user process could obtain a lock on it and possibly prevent other user processes from making progress.

2.4. Termination

When 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 shared data structures are modified atomically. Termination is related to resource control because the kernel must reclaim the resources used by a process after it terminates.

Atomicity can be provided in two ways: by supplying cleanup handlers that are invoked in case of termination, or by deferring termination requests. While assigning cleanup handlers has a slightly lower cost in the common case, programming them is tedious and error-prone. Therefore, most operating systems provide a mechanism to defer termination requests. Usually, this is a software flag that is set when a non-terminatable region is entered and cleared when it is left. In traditional operating systems such as Unix, the non-terminatable region encompasses the complete kernel. Hence, crossing the red line is tantamount to entering code in which termination requests are deferred.

In order to understand why the red line is necessary in Java for safe termination, it is useful to look at the history of Java threads [17]. The original Java specification provided a method to stop threads that caused an exception to be thrown in the 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 run-time system could be left in inconsistent states. As a result, JavaSoft deprecated `Thread.stop()`.

A later proposal for termination was also flawed due to the missing red line in Java. A different mechanism for termination, `Thread.destroy()`, was proposed which would have terminated a thread without releasing any locks it held. This proposal had the potential for deadlock, and JavaSoft never implemented it.

These problems of corrupting vital parts of the run-time system can be avoided by defining a red line between user and kernel. First, parts of the Java run-time libraries must be considered "kernel code" in order to ensure correctness. Second, mechanisms must be present to protect kernel code from termination.

Defining the red line to protect the integrity of the system does not make asynchronous exceptions a usable programming tool for multi-threaded applications. Such applications must still rely on cooperation to stop their own threads. Making termination safe only allows us to kill uncooperative code, such as a malicious applets.

3. How To Draw The Java Red Line

While designing and building KaffeOS, a multi-process Java virtual machine [1], we encountered the issues described in the previous sections. In this section, we describe possible software mechanisms for implementing the various aspects of the red line and explain the solution we have chosen for our system.

Our kernel consists of both Java and C code. Kernel entry and exit are implemented as function calls which may be inlined. We are considering adding Java language support to mark methods as kernel entry points. Such support would allow a compiler to automatically generate kernel entry and exit code.

A thread entering the kernel defers termination requests, which ensures that kernel code may not terminate abruptly. We enter the kernel whenever we need to ensure non-terminatability. For instance, when resolving the symbols of a class, we must make sure that the state of that class and the global class table are kept consistent.

Kernel code must not throw exceptions. This rule is necessary to avoid abrupt termination. We avoid exceptions due to linkage or verification errors by ensuring that the Java parts of the kernel are sound. We trust kernel code not to cause stack overflow exceptions after checking that sufficient space is available on kernel entry. Similar to kernel stack overflows in many operating systems, this assumption is only empirically ensured.

3.1. Resource Control

One possible solution to controlling memory without a kernel is through bytecode rewriting, which is used in JRes [8]. Instructions are inserted before every bytecode that causes a Java object to be allocated. These instructions check the attempted allocation against the process's memory limit. If a resource violation is detected, a callback is invoked, and appropriate action is taken. The advantage of such an approach is that it is portable across standard JVMs.

This solution, however, is incomplete since rewriting system code creates problems: for example, if a callback function decides to abort from a call that originates from the Java runtime. If system code is not rewritten, however, memory that is allocated on behalf of a user process from within system code may not be accounted for. Memory usage is only controlled while a process executes rewritten code, yet only the application portions can be rewritten in this way.

We structured the KaffeOS kernel such that it is able to back out of out-of-memory conditions—or any error condition, for that matter, without disrupting its state. Where the Java specification requires that an error condition be mapped to a runtime exception, this exception is created and thrown after the return to user code.

In order to support fair scheduling, our kernel is multithreaded and preemptive. Priority inheritance is used on all locks internal to the kernel, which prevents starvation.

Each process receives its own heap, which allows us to collect the heaps independently. Collecting a user process's heap does not require stopping other processes or entering the kernel, because we disallow cross-heap references in general—as a result, per-process GC is above the red line. However, we do have one shared kernel heap (which we describe in the next section), and cross-heap references to that heap must be handled differently. We treat such references as external references in distributed garbage collection schemes [15].

3.2. Communication

One approach to communication is to disallow direct sharing. This approach was taken in the JKernel [12]. This restriction, which obviates many of the problems of direct object sharing, can be accomplished by controlling the process's name space. If two processes want to exchange structured objects, they have to marshal and unmarshal these objects through a serial communication channel. In this way, unwanted cross-references are prevented.

KaffeOS uses a restricted form of direct sharing as its primary means of communication. Shared objects are allocated in a shared area, which is subject to per-process allocation limits. Communicating processes are given direct access to these objects; they do not have to serialize data to communicate. In order to prevent these processes from using shared objects to create illegal cross-heap references, we control writes into the heap. The actual data is exchanged through reading and writing to the primitive fields of shared objects.

We use write barriers [19] to prevent cross-heap references. Write barriers are special code that is executed before bytecode instructions that write references to the heap. If the compiler can prove that an object that is the destination of the write lies in the writing process's heap, the write barrier check may be omitted.

3.3. Termination

One possible way to ensure that termination is safe is to restrict access to the primitives that cause termination. For instance, a thread's execution path can be divided into segments [12]. A termination request is only effective if the thread is executing in a segment where it is safe to do so, and deferred otherwise. This can be used to protect a "server" segment from termination requests posted by a client.

Without a kernel, however, using thread segments is only viable for protecting servers. In order to have an effective means of terminating uncooperative applications, a trusted kernel is required. The kernel must be structured to respond to termination requests in a short, bounded amount of time.

We reclaim a process's memory upon termination in two steps. First, the threads executing in a process are destroyed and their stack space is reclaimed. This step cleans up references from the stack into the heap. Since we disallow cross-heap references, we can then simply merge the process's heap with the kernel heap and garbage collect the kernel heap in turn, which will resolve cycles.

4. Related Work

Other researchers have proposed solutions to some of these shortcomings of Java. For instance, Nilsen et al. [14] introduce Java extensions for real-time computation. Their extensions support atomic sections in which asynchronous exceptions are masked. By subjecting the `catch` and `finally` clauses of real-time threads to a set of restrictions, they attempt to make asynchronous exceptions a useful tool for real-time applications. By contrast, we protect only the kernel to make termination safe: our system is not designed for real-time user processes.

Bernadat and others [3] use the idea of separating heaps for controlling the use of memory. They use write barriers to detect cross-heap references as well. Their work is primarily targeted towards thwarting denial-of-service attacks in Java, and does not provide the functionality of our system. For example, they do not allow applications to share objects directly.

The earliest work on using Java to support multiple processes came from Balfanz and Gong [2]. Alta [18] and the JKernel [12] are systems that provide process abstractions in Java. These other researchers have not noted the necessity of providing a clear distinction between user and kernel modes.

In JDK 1.2, JavaSoft [11] introduced the abstraction of protection domains, which represent separate namespaces. These domains do not correspond to user and kernel code, because the clean termination of protection domains is not guaranteed. They are used solely for security purposes: code with the same access privileges is put in a single protection domain. We do not discuss security issues in this paper because the mechanisms to implement security policies do not necessarily require red line crossings.

VINO [16], an extensible OS which uses software fault isolation to protect its extensions (“grafts”), uses transactions to provide cleanup handlers. In addition, VINO establishes a set of rules in order to prevent resource hoarding by single grafts. For instance, a graft cannot access memory to which it has not been granted permission. VINO’s transaction semantics are relatively heavyweight, but they allow any kernel operation to be aborted.

Inferno [9] uses a type-safe language called Limbo to enforce memory protection. The Inferno kernel provides an environment for running applications under the Dis virtual machine. It manages processes, memory and namespaces and provides data streaming and network protocols. Inferno controls an application’s access to resources by controlling the namespace that is visible to that application. Inferno does not control resources such as memory.

5. Conclusions

The use of the red line for protection, resource control, and safe termination has an impact on how system services are structured. For instance, a system service such as `gettimeofday` does not require resource control or protection from termination. The only reason to place this service below the red line would be protection: the system time is read-only to user processes. This example shows that services that do not require all functions of the red line may be implemented with fewer red-line crossings when language-based protection is used.

In some circumstances, the red line must be crossed from kernel to user mode in an upcall [7]. Upcalls are frequent in Java; for example, they are required to implement Java’s user-controlled loading of classes. In such instances, the kernel must assume that the upcall may not return. Therefore, the kernel must clean up its state before an upcall. For example, it must release any locks it holds.

The red line, whose cost has been lamented by OS researchers in traditional operating systems, serves multiple functions beyond its primary protection function. We contend that retaining the notion of a red line is a very useful and effective structuring tool for language-based systems such as Java, in which the protection mechanism has been implemented by type safety.

Acknowledgments

We thank Jay Lepreau for his support of this research. We are grateful to Patrick Tullmann for enlightening discussion and collaboration on this research. We thank Patrick Tullmann, Jay Lepreau, John Carter, and Eric Eide for their feedback on this paper, which helped us to improve it greatly.

References

[1] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Aug. 1998.

[2] D. Balfanz and L. Gong. Experience with secure multiprocessing in Java. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 398–405, Amsterdam, Netherlands, May 1998.

[3] P. Bernadat, D. Lambright, and F. Travostino. 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, Dec. 1998.

[4] B. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. Sirer. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 62–65, Orcas Island, WA, May 4–5, 1995.

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

[6] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193, Monterey, CA, Nov. 1994.

[7] D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 171–180, Orcas Island, WA, Dec. 1–4, 1985.

[8] G. Czajkowski and T. von Eicken. JRes: a resource accounting interface for Java. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, Oct. 1998. ACM.

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

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Dec. 1995.

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

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

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

[14] K. Nilsen, S. Mitra, S. Sankaranarayanan, and V. Thanuvan. Asynchronous Java exception handling in a real-time context. In *Proceedings of the IEEE Workshop on Prog. Lang. for Real-Time Industrial Applications*, Madrid, Spain, Dec. 1998.

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

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

[17] Why JavaSoft is deprecating Thread.stop, Thread.suspend and Thread.resume. www.javasoft.com/products/jdk/1.2/-docs/guide/misc/threadPrimitiveDeprecation.html.

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

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