# Supporting flexible safety and sharing in multi-threaded environments*

Steven H. Samorodin[1] and Raju Pandey[2]

[1] Marimba, Inc. Mountain View, Ca.
shs@marimba.com
[2] Computer Science Department, University of California at Davis
pandey@cs.ucdavis.edu

**Abstract.** There is increasing interest in extensible systems (such as extensible operating systems, mobile code runtime systems, Internet browsers and servers) that allow external programs to be downloaded and executed directly within the system. While appealing from system design and extensibility points of view, extensible systems are vulnerable to aberrant behaviors of external programs. External programs can interfere with executions of other programs by reading and writing into their memory locations. In this paper, we present an approach for providing safe execution of external programs through a safe threads mechanism. The approach also provides a novel technique for safe sharing among external programs. The paper also describes the design and implementation of the safe threads.

## 1 Introduction

There is increasing interest in extensible systems that allow external programs to be downloaded and executed directly within a local system. Examples of such systems include extensible operating systems [3, 7], the Java runtime system [1], mobile code runtime systems [6], Internet browsers and web servers. While appealing from both system design and extensibility points of view, extensible systems are vulnerable to aberrant behaviors of external programs. External programs can interfere with executions of other programs by accessing their memory. They can corrupt system-dependent data, force a program into an inconsistent state, and crash the system. They can write into another program's memory, thereby corrupting system-dependent data, force a program into an

* This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

inconsistent state, and overwrite other programs. Clearly, system software must provide *safety* against malicious or buggy external programs.

The notion of safety has been studied quite extensively in the operating system research and, recently, in type-safety based approaches [8, 9]. Most operating systems implement the notion of safety through address containment as in UNIX [12]. Address containment schemes provide safety by ensuring that a program cannot address the memory used by another program. The problem with the address containment-based approaches is that, in general, they enforce a rigid notion of safety and do not adequately support flexible sharing of data between processes. Sharing mechanisms, such as inter-process communication (IPC) or shared memory, are either inefficient (due to data copying) or require coordination of addresses among processes. Work in single address operating systems (SASOS) [5, 10] have proposed the notion of address spaces that support safety among threads of execution, while providing sharing through address pointers. SASOS provide a nice solution but requires a specialized operating system.

While the above approaches do provide mechanisms for safety and sharing, the mechanisms are either too inflexible or difficult to use for the kind of application we are building. We are interested in developing a mobile code runtime system that creates a thread of execution for every mobile code. Our focus is on developing an execution environment that protects the runtime system and the mobile programs from each other. Further, since data sharing among mobile programs may be dynamic and flexible, the system software must support sharing mechanisms that can be customized dynamically to reflect these sharing patterns.

We, thus, need a protection mechanism that provides protection as well as flexible and dynamic sharing among threads of execution at the user level. This paper presents such a notion of protection and sharing for threads. We present a threads package, called Safe Threads, that supports the notion of threads whose stacks and data elements are completely protected. The thread package contains a novel mechanism for specifying flexible and dynamic sharing and protection among threads. In this approach, the notion of protection is represented by an abstract entity, called *protected domain*. Sharing is defined by *permission* relationships among protected domains. Applications can bind threads and data elements to different protected domains in order to implement different sharing relationships dynamically. We have implemented the thread package through mprotect system calls which make threads context switches quite expensive. Performance analysis of the thread package shows that protected thread creation is approximately 1.5 times more expensive. Context switch times are more expensive as well, but vary depending upon the number of protected domains involved.

The rest of this paper is organized as follows: In Section 2, we describe the notion of safe threads and sharing among them. We also present an implementation of the threads package in this section. In Section 3, we present the performance characteristics of our system. Section 4 discusses related work and we conclude in Section 5.

## 2 Safe Threads package

In this section, we present the notion of safety and sharing within a thread package that we have developed. The thread package supports creation of multiple threads, provide fundamental safety guarantees, and supports mechanism for safe sharing among threads. We first briefly describe the notion of threads and then discuss how safety and sharing is defined in the thread package

### 2.1 Support for Threads

User-level threads packages provide creation, deletion, and management of multiple threads of execution. Threads are execution contexts and share an address space and other per-process resources. Unlike processes which may require a large amount of state information, threads generally need only a program counter, a set of registers, and a stack of activation records. Context switching costs for threads are, therefore, much lower.

Typical user-level threads packages, such as Pthreads [4],are implemented by constructing a separate stack for each thread, while sharing the code and heap data segments. In these thread packages, any thread can access any memory location, including code, scheduler thread stack, other thread stacks, and heap segments.

We have developed a thread package that provides for safe execution of external programs. The thread package provides two levels of safety guarantees. The first is an absolute safety guarantee for data that must always be protected. A thread's per-thread data (including stack and code) are completely protected from other threads. The second guarantee concerns data whose safety and sharing properties can be defined dynamically by the threads themselves. The thread package supports this through the notion of *protected domains* and *permission relationships.*

### 2.2 Protected domains and Permission relationships

A protected domain aggregates regions of memory that have similar sharing properties. A thread cannot access a protected domain and therefore any of the data contained in that the protected domain unless the thread has been bound to the protected domain. A thread can define a binding relationship with a protected domain explicitly or implicitly. An explicit binding between a thread, $T_1$, and a protected domain, $P_1$, denoted $T_1 \rightarrow P_1$, can occur in two ways: (i) When $T_1$ creates $P_1$, $T_1$ is said to be the owner of $P_1$ and can access all entities bound to $P_1$. (ii) When $T_1$, the owner thread of $P_1$, explicitly binds a thread $T_2$ with $P_1$, denoted $T_1(T_2 \rightarrow P_1)$. This explicit binding allows $T_2$ to access any data entities associated with $P_1$. Note that such bindings allow $T_1$ to share any data contained within $P_1$ with other threads. Only the owner can change bindings to allow other threads access or permit other protected domains access.

Implicit binding, occurs as a result of thread bindings and permission relationships among protected domains. A *permission* relationship $\mapsto$ between two

protected domains captures an asymmetric sharing relationship between threads bound to the protected domains. For instance, the relation $P_1 \mapsto P_2$ (read $P_1$ is permitted by $P_2$) specifies that threads bound to $P_1$ can access data entities bound to $P_2$, but not vice versa.

We represent threads, protected domains and permission relationships in terms of a directed graph called a *sharing relationship graph* in which a node denotes a thread or a protected domain and an edge denotes a permission relationship. Each permission relationship indicates a chaining of access to the contents of the protected domain for threads bound to the permitting protected domain. The access associated with each permission relationship is labeled read, write, or read/write, indicating the kind of permission that is allowed. Each protected domain has an access list of (thread ID, access type) pairs associated with it.

The notions of protected domain and permission relationship allow one to define complex and dynamic sharing relationships between threads and data. An example of such a relationship is the hierarchical notion of trust and safety implemented in many systems. In these systems, a multi-level information sharing specification is created where entities (for instance, workers) at level $L$ can access any information that exists at levels $\leq L$. However, they cannot access any information that exists above level $L$. Such a sharing relationship can be easily represented through protected domain and permission relationships. Thus, protected domain and permission relationships allow one to capture patterns of accesses and restriction among cooperating threads.

### 2.3 Implementation

We have implemented the Safe Threads package on top of the QuickThreads [11] library on the FreeBSD 2.2.6 operating system. The QuickThreads library supports non-preemptive user level threads. Safe Threads implements basic threading functionality on top of protection mechanisms. Our current implementation runs inside a single UNIX process virtual address space. Protection is enforced through use of the mprotect(2) system call. mprotect changes the access restrictions for the calling process on specified regions of memory within that processes' virtual address space. Utilizing this mechanism allows for the flexibility to protect any page-sized region of memory.

One important design decision involved whether a thread's stack should be protected from other threads. In order that a thread truly be safe from other threads, stacks must be inaccessible to other threads. There are two important implications, however, on impact the performance of the threads package. Firstly, the context switching code cannot be executed on either thread's stack since at some point in the algorithm each stack is not accessible. This makes context switches more expensive than if the switching code could be executed directly on the stack of the thread that was previously executing. Secondly, and perhaps more importantly, because the stacks are not visible to all threads, parameters passed between threads must be copied.

**Optimizations:** Since systems calls require crossing the user/kernel protection boundary, system calls are more expensive than normal procedure calls. The method for implementing the thread context switch described above may potentially require many system calls per thread context switch. Therefore we have developed methods to speed up a protected context switch. There are two kinds of optimization possible: the first reduces the number of memory regions that must be protected and the second reduces the number of times the user/kernel boundary is crossed. The first can be achieved by combining protection domains of a thread if they do not export data to different threads, and by placing protected regions in contiguous regions so that such regions can be protected through one system call. The current version of the package does not include these optimizations yet as we are still formulating a general algorithm for using the sharing relationship graph to generate optimal[1] protected memory region layouts. Further, since the context switching code is usually very small, it is not clear if there are large benefits to be derived from implementing complex memory layout algorithms.

The second optimization involves reducing the number of system calls. During a context switch, the threads package determines which protected domains need to be protected and unprotected. In our initial implementation, the package makes one mprotect call for each protected region that needed to be protected or unprotected. This results in $O(n)$ system calls per context switch, where $n$ is the number of protected regions. To reduce this number, we extended the FreeBSD kernel to include a new system call, multiMprotect(), which takes a vector of (address, length, protection type) triples. We, therefore, make one system call per context switch by packing all of the data into an array of triples. multiMprotect is a simple wrapper that takes each argument from the parameter vector and calls mprotect.

## 3 Performance Analysis

In this section, we focus on analyzing the costs associated with providing the safety and sharing model. Two benchmarks were performed: a thread creation benchmark and a context switching benchmark.

### 3.1 Thread Creation

The thread creation benchmark compares the cost of creating protected and unprotected threads. Beyond what is required to create an unprotected thread, protected thread creation requires creating a protected domain, adding its stack as a data item, and protecting the stack. Table 1 shows results which indicate that for large numbers of threads creating protected threads is about 1.5 times as expensive.

[1] It is our belief that the general algorithm is at least NP-hard, but we have not proven it yet.

| Thread creation times on Pentium 120 w/32mb | | | |
|---|---|---|---|
| # Threads | Time (No Protection) | Time (With Protection) | **% Difference** |
| 100 | 6.85 | 11.82 | 173 |
| 500 | 8.89 | 13.40 | 151 |
| 750 | 9.45 | 13.75 | 146 |
| 1000 | 9.66 | 14.01 | 145 |
| **Thread creation times on Pentium II 300 w/128mb** | | | |
| # Threads | Time (No Protection) | Time (With Protection) | **% Difference** |
| 100 | 2.27 | 4.18 | 184 |
| 500 | 2.72 | 4.72 | 174 |
| 750 | 4.16 | 6.65 | 160 |
| 1000 | 3.94 | 6.37 | 162 |

**Table 1.** Data for thread creation times is given for two different machines. All times are in micro seconds and are the average of 20 runs of creating the number of threads specified. All machines run FreeBSD 2.2.6-STABLE. g++ v.2.7.2.1 with -O2 and -m486 optimizations was used to compile all test programs.

### 3.2 Context Switch

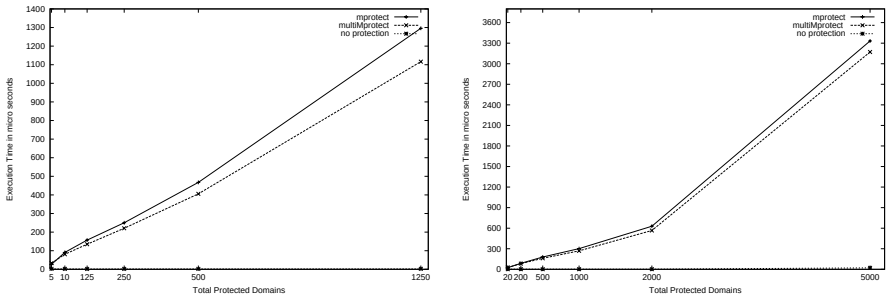
Context switch times for Safe Threads are highly dependent upon the number of protected domains and the number of data elements contained within those protected domains. Figures 1(a) and 1(b) show the cost for context switches with different numbers of protected domains. The cost of an unprotected context switch is, as expected, a constant value. This number was determined by using the Safe Threads package with protection turned off.

As mentioned in Section 2.3, our optimization goal with multiMprotect was to reduce the number of system calls from $O(n)$ to 1 per context switch. In this we were successful, but we found that additional overhead introduced minimizes the performance advantage gained by reducing the number of system calls. For all but the smallest numbers of protected domains, our new system call multiMprotect outperforms mprotect. However, the performance benefit from using multimprotect is not as great as we expected. We feel that this is largely due to inefficient implementation. With different data structures and other optimizations these numbers could be significantly reduced.

While the times for individual context switches can be very high for large numbers of protected domains, the tests were constructed to show worst case behavior where no protected domains are shared between threads. We believe that many applications will share protected domains and thereby incur lower context switch costs, even for large number of context switches.

## 4 Existing Safety Solutions

Existing solutions to the safety problem function at three levels of abstraction: hardware/OS, software and language. Hardware-based solutions address the problem at the lowest level. These solutions rely upon hardware to enforce




(a) Context switch times for various Safe Threads protection options for 5 threads.

(b) Context switch times for various Safe Threads protection options for 20 threads.

**Fig. 1.** Overhead Cost of context switching for safe threads

safety [12]. Hardware protection has the advantage that it physically guarantees protection.

The problem of safely executing untrusted code can also be addressed at the software level. Software safety solutions work at the user-level modifying the compiler, runtime system, and sometimes the untrusted code itself to ensure that software modules do not misbehave. Software Fault Isolation (SFI) [13] and Protected Shared Libraries (PSL) [2] are examples of software safety solutions.

Finally, type-safe languages, such as Java, use language semantics to provide safety. Name space encapsulation ensures that private variables and methods cannot be accessed by other classes. Language-based protection schemes have the advantage that often a cross protection domain call can be as inexpensive as a procedure call. Several systems have been built using these languages including the SPIN extensible operating system [3] and the J-Kernel system [8]. The J-Kernel [8] protection system provides a general framework for supporting multiple protection domains within a single process address space. This work is similar to Safe Threads in that both develop a mechanism for allowing multiple protection domains to exist within a single address space. However, since J-Kernel relies upon Java to enforce its protection, it is limited to creating safety solutions for Java programs.

Opal [5], Mungi [10], and other single address space operating systems (SASOS) address many of the same problems as Safe Threads on an operating system level. Specifically, providing protection and sharing within a single address space.

## 5 Conclusion

We have presented the design and implementation of a threads package that supports provides safety among threads. The package supports creation of threads, provides isolation among them, and includes mechanisms for protected sharing among threads. We have implemented the thread package and initial performance analysis suggests that thread creation is approximately 1.5 times more expensive for creating protected threads. Context switching times depend upon the number of protected domains involved. We are currently looking at different techniques for optimizing the cost of thread creation and context switching.

## References


1. K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.
2. A. Banerji, J. M. Tracey, and D. L. Cohn. Protected shared libraries-a new approach to modularity and sharing. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 59–75, Anaheim, CA, January 1997.
3. B. Bershad et al. Extensibility, safety and performance in the SPIN operating system. *15th Symposium on Operating Systems Principles*, pages 267–283, December 1995.
4. D. R. Butenhof. *Programming with POSIX Threads.* Addison Wesley Longman, Inc., 1997.
5. J. Chase, H. Levy, M. Feeley, and E. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions On Computer Systems*, 12(4):271–307, May 1994.
6. D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are they a good idea? In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48. Springer-Verlag, April 1997.
7. D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *15th Symposium on Operating Systems Principles*, pages 251–266, December 1995.
8. C. Hawblitzel, C. Chang, G. Gzajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 259–272, New Orleans, La., June 1998.
9. C. Hawblitzel and T. von Eicken. A case for language-based protection. Technical Report 98-1670, Cornell University, Ithaca, NY, 1998.
10. G. Heiser, K. Elphinstone, J. Vochteloo, and S. Russell. Implementation and performance of the Mungi single-address-space operating system. Technical Report UNSW-CSE-TR-9704, The University of New South Wales, Sydney, Australia, June 1997.
11. D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.
12. U. Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1996.
13. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *14th Symposium on Operating Systems Principles*, pages 203–216, 1993.