SPIN – An Extensible Microkernel for Application-specific Operating System Services

Brian N. Bershad Craig Chambers Susan Eggers Chris Maeda
Dylan McNamee Przemysław Pardyak Stefan Savage Emin Gün Sirer

Dept. of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195

April 30, 1994

Abstract

Application domains such as multimedia, databases, and parallel computing, require operating system services with high performance and high functionality. Existing operating systems provide fixed interfaces and implementations to system services and resources. This makes them inappropriate for applications whose resource demands and usage patterns are poorly matched by the services provided. The *SPIN* operating system enables system services to be defined in an application-specific fashion through an *extensible* microkernel. It offers applications fine-grained control over a machine's logical and physical resources through run-time *adaptation* of the system to application requirements.

1 Introduction

The next decade will bring a radical change in the resource requirements of common computer applications. High performance applications that were at one time "niche services" such as large distributed databases, interactive multimedia, and programs for massively parallel systems, will become common. Although application demands are changing substantially, the operating systems base has remained relatively static. Consequently, application performance is frequently limited by today's operating systems, which provide an inadequate interface to computer system resources.

This paper describes an operating system called *SPIN* that will address the requirements of this coming generation of resource-intensive applications. In *SPIN*, these requirements are satisfied through kernel support for *application-specific* services. An application-specific service is one that precisely satisfies the functional and performance requirements of an application or class of applications. The key idea in *SPIN* is that application-specific services can be implemented with code sequences that are installed into the kernel at runtime. These code sequences expose alternative interfaces, and enable alternative implementations of existing interfaces to demanding applications. A trusted compiler and safe language runtime environment ensure that the installed sequences do not violate system integrity.

The ideas underlying *SPIN* stem from research over the last several years that has addressed some of the fundamental performance problems that arise in modern operating system services, including interprocess communication, synchronization, thread management, networking, virtual memory, and cache management [Draves et al. 91, Bershad et al. 92, Stodolsky et al. 93, Bershad 93, Maeda & Bershad 93, Yuhara et al. 94, Thekkath et al. 93, Felten 92, Young 89, McNamee & Armstrong 90, Anderson et al. 92, Wheeler & Bershad 92]. In each case, the interfaces exported by a service were poorly matched to the needs of important applications. The solution to the performance problem came from enabling applications to *adapt* the behavior (interface and implementation) of system services to realize maximum performance. Each change, though, required careful and deliberate modifications of the operating system kernel.

Our goal in building *SPIN* is to provide applications with an *adaptable* kernel platform that enables system resources to be efficiently and safely managed by the application. By *efficient*, we mean that capable applications execute more quickly and with less programming complexity than when using a more conventional platform, such as Ultrix or Mach. By *safe*, we mean that multiple applications may run at the same time, yet be protected from one another through hardware and software firewalls.

In the rest of this position paper, we expand on our approach to operating system adaptability and resource management, discuss the language and compiler requirements of the system, and briefly describe the system's current status.

2 Operating System Adapatability

SPIN supports adaptability through an extensible microkernel that can safely execute application-specific code at the kernel level. The application-specific kernel components are called *spindles* (*SPIN* Dynamically Loaded Extensions), and enable applications to define the precise interface and implementation for the kernel services that they require. Specifically, installing code at the kernel level allows for flexible and rapid response to system hardware and software events. For example, an application program can install a code sequence that runs in response to one of its threads being preempted by an interrupt, a time-slice event, or a higher-priority thread. In the first two cases, the program can ensure system-wide or application-wide invariants about preemptability. In the third case, the application can enforce constraints that deny priority inversion.

Spindles enable a service to be partitioned across the user/kernel boundary in the most efficient manner that still satisfies its safety and sharing requirements. A service might be crafted in terms of *application-level components*, which are linked into the application's address space, *kernel-level components*, which provides fast, specialized access to in-kernel services, or *user-level server components*, which manage long-lived service state. By allowing applications to participate in the implementation of services, we permit them to make informed decisions about their resource requirements. By placing the implementation within an application component (*application-level library*), or a kernel-level code sequence, the service can be accessed with low latency.

3 Resource Management

An operating system kernel offers two general functions: it provides abstractions of the system's physical and logical resources, and it implements a set of management policies for those resources. In the *SPIN* kernel, both of these functions are adaptable. Low-level resource controllers provide lightweight abstractions of the physical hardware, such as page frames and activation contexts. Higher level resource abstractions such as threads or address spaces are implemented by collections of communicating spindles, which may each be individually replaced or interposed on.

SPIN addresses the management of these resources with a two-level resource allocation architecture. The primary, or *system allocator* manages a global pool of resources, such as pages, CPUs, and network bandwidth. The secondary, or *user allocator*, manages private pools of resources that have been acquired from the system allocator. A user allocator may be implemented as a spindle to allow application specific knowledge to be directly applied to the management of its resources. Each application may potentially have its own user allocator, although applications without special purpose resource mangement requirements can use default policies.

The system allocator is responsible for reclaiming resources when a shortage occurs. The user allocators in turn may select individual resource instances as more or less important, and consequently influence their eligibility for reclamation. The resource management policy used by the system allocator may vary depending on the types of guarantees the system needs to provide to application programs.

4 Language and Compiler Requirements

An operating system interface is much like a programming language in that it defines a primitive set of operations available to the programmer [Lampson 84]. In *SPIN*, the operating system interface is defined by an actual programming language through which applications can define and install new interfaces that match their requirements. Because we anticipate aggressive use of spindles for system decomposition, we require this language and the associated compiler technology to provide specialized support for safety and performance. Our target language is a safe subset of C that we have defined.

We depend on a combination of type safety, object based methodology, and explicit guards to limit the access of untrusted spindles. The combination of type safety and well-defined interfaces to kernel services can ensure that only legal operations may be invoked on data structures shared between spindles and native kernel code. Synchronization between trusted kernel code and untrusted spindles is accomplished through the use of time bounded closures defined by spindles running in the context of critical sections defined and implemented by the kernel.

Spindles are compiled into an executing system at run-time. We rely on aggressive compiler technology to ensure that the *SPIN* microkernel extended with user-defined spindles performs as well as a non-extensible monolithic operating system with services built-in to the kernel. Good performance can be achieved with well-understood optimizing compiler technology, such as intraprocedural data flow analysis, symbolic evaluation, and inline expansion. These techniques can eliminate much of the overhead of the spindle language: the compiler can inline-expand calls in spindles to kernel operations, replacing them with direct data structure accesses or even constants, and the compiler can evaluate predicate expressions guarding kernel operations in the context of the spindle code preceding the call. With this technology, spindles can be installed and executed quickly.

Advanced compilation technology, such as *partial evaluation* [Jones et al. 89, Consel 90, Weise et al. 91, Jones et al. 93] can blend together multiple spindle routines and the surrounding kernel code to reduce the overheads of maintaining large numbers of spindles. It can also reduce the cost of crossing from the kernel's execution environment to the spindle's. Partial evaluation is a program transformation technique that specializes program code with respect to some of its argument values. In our context, for example, if several spindles are associated with the same kernel event, the compiler can specialize the event dispatcher to produce a single code sequence tuned just for the spindles installed at that time.

5 Status

We are developing *SPIN* in the context of the Mach 3.0 microkernel and an OSF/1 Unix server running on DEC Alpha workstations. We are partitioning the system statically into a *SPIN* component and a native (Mach 3.0 and OSF/1) component. Existing OSF/1 binaries will continue to run by accessing the OSF/1 services that manage the native-component. *SPIN* will manage the *SPIN* component across applications that have been explicitly marked to run within *SPIN*. This approach will allow us to migrate away from a mixed-mode system to one that runs *SPIN* natively.

References

[Anderson et al. 92] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Bershad 93] Bershad, B. N. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, May 1993.

[Bershad et al. 92] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 223–233, October 1992.

[Consel 90] Consel, C. Binding Time Analysis for Higher Order Untyped Functional Languages. In *Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[Felten 92] Felten, E. The Case for Application-Specific Communication Protocols. In *Proceedings of Intel Supercomputer Systems Division Technology Focus Conference*, pages 171–181, 1992.

[Jones et al. 89] Jones, N., Sestoft, P., and Sondergaard, H. MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp & Symbolic Computing*, 2(1):9–50, February 1989.

[Jones et al. 93] Jones, N., Gomard, C., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[Lampson 84] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[McNamee & Armstrong 90] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the Usenix Mach Symposium*, pages 17–29, 1990.

[Stodolsky et al. 93] Stodolsky, D., Bershad, B. N., and Chen, B. Fast Interrupt Priority Management for Operating System Kernels. In *Proceedings of the Second Usenix Workshop on Microkernels and Other Kernel Architectures*, September 1993.

[Thekkath et al. 93] Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[Weise et al. 91] Weise, D., Conybeare, R., Ruf, E., and Seligman, S. Automatic Online Partial Evaluation. In *Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag, August 1991. LNCS 202.

[Wheeler & Bershad 92] Wheeler, B. and Bershad, B. N. Consistency Management for Virtually Indexed Caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992.

[Young 89] Young, M. W. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Technical Report CMU-CS-89-202, Carnegie Mellon University, November 1989.

[Yuhara et al. 94] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.