Extensibility, Safety and Performance in the *SPIN* Operating System

Brian N. Bershad Stefan Savage Przemysław Pardyak Emin Gün Sirer
Marc Fiuczynski David Becker Susan Eggers Craig Chambers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

DRAFT of March 30, 1995

Abstract

This paper describes the motivation, architecture and performance of *SPIN*, an extensible operating system. *SPIN* provides an extension infrastructure together with a core set of extensible services that allow applications to safely change the operating system's interface and implementation. These changes can be specified with fine-granularity, allowing applications to achieve a desired level of performance and functionality from the system. Extensions are dynamically linked into the operating system kernel at application runtime, enabling them to access system services with low overhead. A capability-based protection model that relies on language and link-time mechanisms enables the system to inexpensively export fine-grained interfaces to system services. *SPIN* and its extensions are written in Modula-3 and run on DEC Alpha workstations.

1 Introduction

SPIN is an operating system that can be dynamically specialized to safely meet the performance and functionality requirements of applications. *SPIN* is motivated by the need to support high performance applications which present service demands which are poorly matched by the interfaces and implementations of contemporary operating systems. For example, disk buffering and paging algorithms found in modern file and virtual memory systems are inappropriate for database applications. Other applications, such as web servers, network routers, interactive multimedia clients and servers, and realtime and parallel programs are similarly mismatched by operating system services. Using *SPIN*, applications can extend the interfaces and implementations provided by the kernel, enabling them to efficiently specialize the system to their needs.

Three critical concerns in an application-oriented operating system such as *SPIN* are extensibility, safety, and efficiency. A system's *extensibility* is determined by the interfaces to services and resources which are exported to applications. *Safety* determines the exposure of applications to the actions of others. Finally, *efficiency* reflects the overhead of specializing an operating system for an application. Each of these concerns is easily addressed in isolation, as demonstrated by most existing operating systems. However, systems have compromised one or more of these concerns in favor of the others.

1.1 Goals

The goal of our research is to build a general purpose operating system that does not compromise extensibility, safety or efficiency. Extensibility requires an infrastructure that allows for fine-grained access to system resources and functions. Safety requires that access be controlled at the same granularity. Efficiency requires that the overhead of both protection and access be low. With these three properties satisfied, it is possible to construct services that implement safe and extensible interfaces to low-level system resources such as scheduling and memory management.

Our insight is that language and runtime services can be used to provide low-cost, fine-grained, protected access to operating system resources. We have applied this insight in the design of *SPIN* through the use of four techniques – co-location, type safety, logical protection domains, and dynamic binding:

- *Co-location.* Extensions are dynamically linked into the kernel virtual address space and protection domain. As a result, minimal runtime overhead (for either control or data transfer) is incurred when communicating between system code and application extensions.

- *Type safety.* Extensions execute in the kernel's address space where they are prevented from accessing memory or issuing privileged instructions that could violate the base integrity of the operating system. We rely on a type safe, modular programming language, Modula-3 [Nel91], to ensure that extensions do not issue privileged instructions, or reference memory to which they have not been given

This research was sponsored by the Advanced Research Projects Agency and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Chambers was partially sponsored by an a National Science Foundation Presidential Young Investigator Award. Sirer was supported by an IBM Graduate Student Fellowship.

explicit access through an interface. A type safe language permits many memory references to be validated at compile-time rather than runtime using either hardware or software mechanisms. Strict typing and modular interfaces enable *SPIN* to inexpensively provide a protection model based on capabilities. A capability in *SPIN* is simply an unforgeable opaque reference to a data structure whose implementation is concealed behind an interface boundary.

- *Logical protection domains.* Extensions are placed within logical protection domains, which are kernel namespaces consisting of interfaces. Domains can be intersecting or disjoint, enabling applications to either share services, or introduce new ones. Logical protection domains are themselves referenced through capabilities, and provide a rich set of linking operations that allow access control to be specified at the granularity of an interface or a collection of interfaces.

- *Dynamic binding.* Extensions are integrated with the running system using an event-based communication mechanism. System and extension code raise events, which are dynamically dispatched to handlers. Events are defined as procedures, which are tightly integrated with the language and runtime system. Additional or alternative handlers can be installed to change system behavior.

Co-location, type safety, logical protection domains, and dynamic binding enable interfaces to be defined and safely accessed with low overhead at a fine-grained level. However, these techniques do not guarantee the system's extensibility. Ultimately, extensibility is achieved through the system service interfaces themselves, which define the exact set of resources and operations that are exported to applications.

SPIN provides a set of interfaces to core system services, such as memory management and scheduling. These interfaces exploit co-location to efficiently export fine-grained operations on system resources, type safety to ensure their own integrity against errant extensions, logical protection domains to provide access points to related interfaces, and dynamic binding to express complex relationships between system components at runtime.

1.2 System overview

The *SPIN* operating system is written in Modula-3, which allows extensions to be easily integrated because they can share interfaces, data structures, and programming models. Although *SPIN* uses language services to provide safe extensibility within the kernel, applications can be written in any language and execute within their own virtual address space. Only code that requires low-latency access to system services must be written in the system's safe extension language. For example, we are using *SPIN* to implement a version of the UNIX operating system in which applications run in their own address space. In this implementation, we are partitioning the system into a user-level server and a set of kernel extensions that provide the server low-latency access to core services such as memory, scheduling and networking. We have found that the system structuring approach made possible by *SPIN*'s architecture can yield extremely good performance. For example, on a 133 MHz DEC Alpha workstation, a user program such as a server can create a new thread that runs in an extension in 5 μsecs, synchronize with another thread in 29 μsecs, and handle a page fault in 17 μsecs. Client programs can execute a protected procedure call into server code in 12 μsecs. We show that comparable functions using a traditional or microkernel-based operating system execute with substantially greater overhead.

Networked video is another application domain that demonstrates the utility of *SPIN*'s architecture. We have implemented a networked video server and client that runs on top of the *SPIN* services. The video server defines an extension that implements a direct stream between the disk and the network. The video server, as it starts up, links this extension into the kernel and subsequently controls it through system calls, which it also defines as system extensions. The new module transfers data directly from disk to the network. On the client side, the viewer application installs an extension into the kernel that awaits incoming network video packets, decompresses them, and then deposits the frames directly into the frame buffer. There are no unnecessary control or data transfers across the user/kernel boundary. With this approach, the client and server operating system kernels are *specialized* to provide low latency, low overhead, networked video service.

1.3 The rest of this paper

The rest of this paper describes the motivation, design, and performance of *SPIN*. In the next section, we motivate the need for extensible operating systems and discuss related work. In Section 3 we describe the system's architecture in terms of its protection and extension models. In Section 4 we describe the core extensible services provided by the system to applications. In Section 5 we demonstrate the use of the system's mechanisms in constructing some application-specific services. In Section 6 we discuss the system's performance and compare it against that of several other operating systems. Finally, in Section 7 we present our conclusions.

2 Motivation

An extensible system is one that can be changed to meet the performance or functionality demands of a specific application. In contrast, most traditional operating systems have been designed with the goal of supporting only general purpose computing. Consequently, these operating systems are often adequate for many functions, but excellent at none. For this reason, many research and commercial systems have been modified to address performance problems caused by a particular application's needs, such as interprocess communication, synchronization, thread management, networking, virtual memory, and cache management [DBRD91, BRE92, SBC93, Ber93, YBMM94, MB93, Fel92, YTR$^+$87, HC91, MA90, ABLL92, FP93, WB92, RLB94, ROKB95]. For example, most improvements in IPC performance have been motivated by database applications or operating system servers. Each change required careful and deliberate modifications of the operating system kernel, making the extension difficult to implement. Moreover, each change intended to improve the performance of one class of applications would often degrade that of others.

Commodity operating systems have been slow to adopt new technologies to support advanced applications [VGA94]. In order to justify the cost of introducing a new service, it must be demonstrated that the service has broad applicability. By its very nature, this demonstration precludes application-specific services for all but the most common applications. Nevertheless, the need for extensibility in operating systems is shown clearly by systems such as MS-DOS, Windows, or the Macintosh Operating System. Although these systems were not designed to be extensible, application builders have been able to exploit weaknesses in their protection mechanisms to directly modify operating system data structures and code [SMP92]. While individual applications have benefited from this level of freedom, the lack of safe interfaces to either operating system services, or operating system extension services, has created system configuration "chaos [Dra93]." *SPIN* directly addresses this deficit.

2.1 Related work

There have been several attempts to build extensible systems. For example, Hydra [WLH81] provided an infrastructure that allowed applications to manage resources through multi-level policies. The kernel provided the mechanism for allocating resources between processes, and the processes themselves implemented the mechanisms for managing those resources. Hydra's approach, although highly influential, was expensive to use. Basic system services had high overhead because the potential for extension incurred a cost, even when not used. Hydra's capability-based architecture also incurred a high cost when used to access an out-of-kernel extension. Lastly, the system was designed with the notion of "large objects" as the basic building blocks, requiring a large programming effort to affect even a small extension.

Recently, researchers have investigated the use of microkernels as a vehicle for building extensible systems [B$^+$92, MRT$^+$90, CZ83, CD94, TSS88]. In a microkernel-based system, operating system services are implemented as user-level programs and accessed through an interprocess communication service provided by the kernel. Unfortunately, microkernels share many of Hydra's performance problems (using an extension is much more expensive than using a built-in kernel service). Moreover, the granularity with which extensions can be specified in microkernel-based systems is quite large.

Some systems rely on "little languages" to extend safely the system call interface through the use of interpreted code that runs in the kernel [LCC94, MRA87, YBMM94]. These systems suffer from two problems. First, the languages, being little, make cumbersome the expression of arbitrary control and data structures, therefore limiting the range of possible extensions. Second, the interface between the language's programming environment and the rest of the system is generally narrow, making system integration difficult. Finally, interpretation overhead may be a limiting factor to performance.

Several projects [Luc94, EKO94, SS94] are exploring the use of *software fault isolation* [WLAG93] to allow application code, written in any language, to be linked into the kernel's virtual address space. Software fault isolation relies on a binary rewriting tool that inserts explicit checks on memory references and branch instructions. These checks allow the system to define protected memory segments that limit the range of accessible memory without relying on virtual memory hardware. Software fault isolation shows promise as a co-location mechanism for relatively large code and data segments, but it is not clear if the mechanism is appropriate for systems that fine-grained sharing.

Researchers at MIT are building a microkernel that exports fine-grained hardware services, such as TLB management, directly to applications [EKO94]. The system uses a combination of techniques, including software fault isolation, code inspection, type safety, and probabilistic capabilities, to provide low-latency access to hardware resources. Unlike the *SPIN* kernel, which provides extension-oriented facilities for logical protection domains and event binding, their system provides no abstractions beyond those minimally provided by the hardware [EK95].

Several systems [CHL91, RDH$^+$80, Mos94, SS94], like *SPIN*, have leveraged type safety to build an extensible system. Pilot, for example, was a single-address space system which ran programs written in Mesa [GMS77], a high-level systems programming language which is an ancestor of Modula-3 [Nel91]. In general, systems such as Pilot have depended on the language for all protection in the system, not just for the protection of operating system extensions. In contrast, *SPIN*'s reliance on type safety applies only to extension code that runs in the kernel. Address spaces are used to otherwise isolate the operating system and programs from one another.

Many systems provide interfaces that enable code to be installed into the kernel at runtime. Examples include dynamically linked device drivers, stackable layers [HP94], or server-co-location [LHfL93, RAA$^+$88]. Reflective operating systems, such as Apertos [YTT89], provide arbitrary access to system internals. In all of these systems, the right to define extensions is restricted because any extension can bring down the entire system. Application-specific extensibility is not possible.

3 The *SPIN* Architecture

The *SPIN* architecture provides a software infrastructure for safely combining system and application code. The *SPIN* protection model supports efficient, fine-grained access control of resources, while the extension model enables extensions to be specialized with the granularity of a procedure call. The system's architecture is heavily biased towards mechanisms that can be implemented with low-cost on conventional processors. Consequently, *SPIN* makes few demands on the hardware, and instead relies on efficient language-level mechanisms, such as static typechecking, dynamic linking, and dynamic binding, for protection and extensibility. Static typechecking provides for inexpensive memory protection and capabilities. Dynamic linking, which maps program symbols to virtual addresses, serves as the basis of the system's co-location and logical protection facilities. Finally, dynamic binding, which determines caller/callee relationships at the time of the call, enables services to be extended with fine-granularity.

3.1 The Protection Model

A protection model controls the set of operations that can be applied to resources. For example, a protection model based on address space boundaries ensures that a process can only access memory within a particular range of virtual addresses. Most systems rely on a combination of hardware and software mechanisms to define their protection model. UNIX, for example, places processes into separate address spaces, and exports a privileged system call interface through which applications request system services. Address spaces, though, are generally insufficient as a basis for building fine-grained protection mechanisms, such as those that can selectively export interfaces or instances of resources to clients. *SPIN*'s protection model addresses this requirement.

Capabilities

All kernel resources in *SPIN* are referenced by capabilities. A capability is a secure reference to a resource, which can be a system object, an interface, or a collection of interfaces. An example of each of these is a physical page, a physical page allocation interface, and the entire virtual memory system. Individual resources can be protected to ensure that extensions only reference the resources to which they have been given access. Interfaces and collections of interfaces can be protected to allow different extensions to have different views on the set of available services.

Unlike other capability-based operating systems, which implement capabilities using special-purpose hardware [CKD94], virtual memory mechanisms [WLH81], probabilistic protection [EKO94], or protected message channels [B$^+$92], *SPIN*'s capabilities are implemented in terms of opaque references. An opaque reference is simply a type safe pointer to an abstract data type. The type system statically ensures that only modules in which the reference can be revealed have access to the underlying data structure. The most important implication of this static analysis is that no runtime translation mechanisms are required to map from a protected capability to the underlying referent.

Capabilities can be passed from the kernel to user-level applications (which are not required to be type safe) as externalized references. An externalized reference is a handle to an entry in an in-kernel data structure that maps between unprotected and protected references. An externalized reference can be converted back into its underlying type safe reference through this data structure. Any code running in the kernel must explicitly perform this conversion as it transfers references across the user/kernel boundary.

The key point about capabilities in *SPIN* is that they refer directly to the underlying resource which they name. The "value" of a capability is the address in kernel virtual memory at which the named resource resides. In a type safe system, it is impossible to express these values directly. Instead, all references must be through symbolic names which are instances of pointers to objects of the referent type. It is the responsibility of the language implementation to convert this symbolic name into its underlying representation, which is ultimately a memory address, using the type of the object to which the name refers. For example, the name of a procedure is a symbolic representation for the address of the first instruction of the procedure, and the name of a variable is the symbolic representation for the address at which the variable can be found. Because extensions and the kernel execute in the same virtual address space, a dynamic linker can automatically map symbol names to values (addresses), allowing capabilities to be exercised at the speed with which the hardware allows direct loads and stores.

Domains

Just as virtual addresses are a resource in virtual memory systems, the names of symbols and types are a resource in our system. Unlike a virtual memory system though, the names of resources in *SPIN* are directly meaningful to the programmer. For example, if t is an instance of the type T, then both t and T occupy a portion of the namespace. Because the namespace is dynamically extended, it must be explicitly managed to control or prevent name conflicts. Traditional virtual memory systems address this problem through the use of separate address spaces; a virtual address (name) in one address space is, by default, independent of that in another. Only through explicit mapping and sharing operations is it possible for names to be meaningful between address spaces. *SPIN* provides an analogous service for the management of symbols that are component to separate entities, or *domains*, in the system.

A domain defines a set of interfaces that contain names which can be referenced by any extension that has access to the domain. A new domain is created using the *Create* operation, which initializes a domain with the contents of a safe object file. Any symbols exported by interfaces that are implemented in the object file are automatically exported from the domain, and any imported symbols are left unresolved. An object file is safe if it has been cryptographically signed by a trusted object file generator. Currently, we generate safe object files using a Modula-3 compiler or a special program with which trusted users can vet code.[1] A Modula-3 program is "safe" if it is pointer safe, uses only the safe subset of Modula-3 operators, and imports only safe interfaces [Nel91].

The *Resolve* operation serves as the basis for dynamic linking. It takes two domains, the target and the source. Any unresolved symbols in the target domain are resolved by linking against symbols in the source. Cross-linking occurs through a pair of *Resolve* operations. Resolution only resolves a domain's internally undefined symbols; it does not cause additional symbols to be exported. Domains can be aggregated to create linkable namespaces which are the union of existing domains. In this way, a domain can be used to bind together a collection of related interfaces. For example, many of the *SPIN* kernel services are contained within a single domain that is globally accessible.

Domains are themselves named by capabilities, allowing them to act as the root of a capability-based protection space. A domain capability (an opaque object implemented by the *SPIN* kernel) entitles its holder to manipulate the domain according to the rights associated with the reference. A domain with the *modify* right allows it to serve as the target of a *Resolve* operation, as the operation will actually modify the code within the domain as it resolves references. The *import* right allows a domain to serve as the source of a *Resolve* operation, as it implies that the holder of the right may access the exported interfaces in the domain. Clients of the domain interface can either use the low-level manipulation operations described above, or can use a higher-level *Link* function, which takes a list of domains and automatically cross-links them. The domain interface directly supports protected procedure calls between extensions. This is accomplished by creating a domain with an implementation of a protected interface, and then passing out a *import* right to any clients that require the interface.

Figure 1 demonstrates the use of the domain operations to affect a cross-link between three separate object files, each of which initially exists within its own domain. We have implemented an in-kernel file system for *SPIN*, called *SFS*, that provides a UNIX file system interface to a raw disk. Although the file system consists of a fair amount of code, it exports a narrow interface to clients (*open, close, read, etc...*). *SFS* isolates itself from clients by creating two domains. The first is an implementation of the exported *SFS* interface, and the second is an implementation of the non-exported service procedures required by the interface (*ReadInode, etc.*). *SFS* cross-links the two domains during its initialization phase. Potential clients of *SFS* are passed a reference to the first domain with the *import* right enabled. This allows clients to resolve against the *SFS* interface and implementation, but does not give them access to the private procedures.

[1] The vetting process allows us to run code written in any language in the kernel. It serves as a loophole mechanism which should only be used by privileged users.

3.2 The Extension Model

An extension changes the way in which a system provides service. All software is extensible in one way or another, but it is the extension model that determines the ease, transparency, or efficiency with which an extension can be applied.

An extension model is defined by some invocation mechanism, which is responsible for transferring control and data from one point in the system to another in response to some system event. For example, UNIX uses system calls as the invocation mechanism, and microkernel-based systems use messages. In *SPIN*, neither of these mechanisms are appropriate. Because the system's components change over time, the static approach suggested by a system call interface cannot be used. Because performance is critical, the level of data and control indirection suggested by the messaging approach is inappropriate. Instead, *SPIN*'s invocation mechanism is based on *events*, which provide a low-overhead, indirect control and data transfer facility within the kernel.

Events provide "hooks" on which applications can attach their extensions. An event is raised to announce a change in the state of the system, or to request a service. An event is handled by an *event handler*, which is a piece of code that is executed in response to a specific event. Any code running in the kernel can raise or handle an event. The event name provides the common interaction point between the raiser and the handler.

There may be any number of handlers installed on a particular event. The *SPIN dispatcher* is responsible for event delivery and provides for indirect invocation and multicast. The dispatcher matches event signallers to handlers, ensuring that any handler waiting on an event is executed when the event occurs. In this way, neither extensions nor code that interacts with extensions must implement mechanisms for dealing with a frequently changing space of services. An event for which there is no handler can be raised as easily as one for which there are several.

Specifying events and handlers

Events are exported through interfaces and described by procedure signatures. Extensions install a handler on an event by calling the dispatcher with the name of the event and the handler. The handler must be a procedure of the same type as the event itself, as the handler will be invoked with the arguments whose types are specified by the event's definition, and whose values are specified by the event raiser. this way. This approach to event naming is attractive for several reasons. First, access to event names, either for raising or handling, is controlled through the domain system. An event can not be named unless it is named through an interface that has been

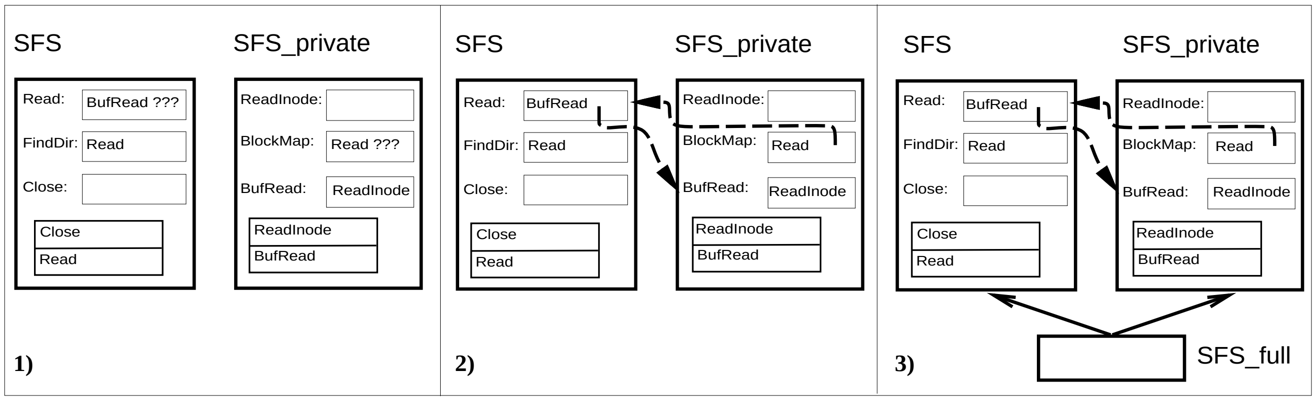

Figure 1: This figure shows domains being used to link together separate interfaces for *SPIN*'s in-kernel file system, SFS. This figure shows the link progressing in stages. In the first panel, the two domains comprising SFS are loaded and their internal symbols are resolved. Both domains export two symbols. Unresolved symbols are indicated by "???". In the second panel, the two domains are shown after cross-linking (the first domain is resolved against the second and the second is resolved against the first). In the final panel, a domain is created that contains both the public and private procedures of SFS, but only the public interface is exported. (Note that the actual implementation of SFS contains more symbols than shown in the figure.)

explicitly imported. More specifically, a thread running in a domain can only name an event, say to raise it, if the event is described in an interface to which the domain has access. Events can also be named indirectly through variables. Rather than defining an event as a constant (procedure signature) in an interface, the interface can instead export a procedure that returns an event through a procedure variable. This allows a service to specify extremely fine-grained access rights to components in an interface. A second advantage that comes from identifying events as procedures is that, in many cases, the raising and handling of an event can be implemented with a direct procedure call from the raiser to the handler.

Event guards

A handler can be associated with a *guard*, which defines an arbitrary predicate that is evaluated by the dispatcher prior to the handler being invoked. If the predicate is true when the event is raised, then the handler is invoked, otherwise the handler is ignored. Guards permit extensions and the base system to separate the specification of what should happen from when it should happen. Extensions define the operations that occur in response to events, whereas guards ensure that those operations happen only at the proper time.

Guards are used to finely restrict access to events. An interface can, instead of exporting an event directly, export a procedure that installs an event handler on behalf of a potential client. When installing the handler, an additional guard can be imposed on a primary guard specified by the client. In this way, an implementation of an interface can specify an additional set of predicates that must be true before a particular handler is fired. For example, consider the event *Ethernet.PacketArrived(pkt: Network.Packet)* defined by the interface to the Ethernet device driver. The driver raises this event each time an Ethernet packet arrives. If the event were globally exported, then any extension could intercept any arriving packet. To prevent this, the network interface exports the event indirectly through an installation procedure that composes the client's guard with an additional one that discriminates according to the packet header.

Other mechanisms could of course be used to implement the functionality of guards. For example, each extension could export an array of event names and raise only the "right" one on each event, in effect doing the predicate analysis itself. In practice, separating the "what" from the "when" simplifies the system because the predicate only needs to be specified once rather than every time the event is raised, and enables centralized optimization in the dispatcher, such as decision-tree pruning, executable data structures [MP89], and dynamic code generation [KEH93].

Event properties

The dispatcher provides a central point of control through which a rich set of event handling semantics can be defined. A handler may be constrained to execute synchronously or asynchronously, in bounded time, or in some arbitrary order with respect to other handlers participating in the same event. By default, the dispatcher executes handlers synchronously, to completion, in undefined order, and returns the result of the final handler executed. The default behavior matches that of procedure call and is appropriate for cases where extensions trust one another to complete, or only one handler is invoked in response to an event.

In cases where the default behavior is inappropriate, additional properties can be specified. For example, an event may be bounded by a time quantum so that, if a handler runs for longer than its quantum, it will be aborted. An event may be asynchronous, which causes each of its handlers to execute as a separate thread and does not stall the handler. Finally, while multiple handlers may execute in response to an event, a single result must be communicated back to the raiser by associating with each event a procedure which ultimately determines the final result [PB94].

Locating an interface

SPIN's nameservice facilities demonstrate the use of the event machinery. Typically, systems provide some type of a nameserver that allows programs to map from some symbolic name, such as "SFS," to a system-dependent reference, such as a port. In *SPIN*, the reference is a domain capability against which extensions resolve.

Rather than providing a nameserver, *SPIN* provides only a nameserver interface that describes a *publish and subscribe* interface [DEFS88, OPS+93]. Potential users of a domain can raise a *Subscribe* event, specifying as an argument to the event the textual name of the requested interface. An interface implementation registers a handler, which acts as a publisher, for the interface. The handler's guard is a predicate that compares the argument string to the name of the exported interface. When the *Subscribe* event is raised, the dispatcher evaluates all registered guards for that event until it finds one for which the predicate evaluates to true. The dispatcher invokes the handler, which returns a domain capability for the requested interface. Additional predicates, for example to compare the raiser's identity against an access control list, can also be specified in the guard. Figure 2 illustrates the relationship between a subscribing extension, the dispatcher, and several publishers.

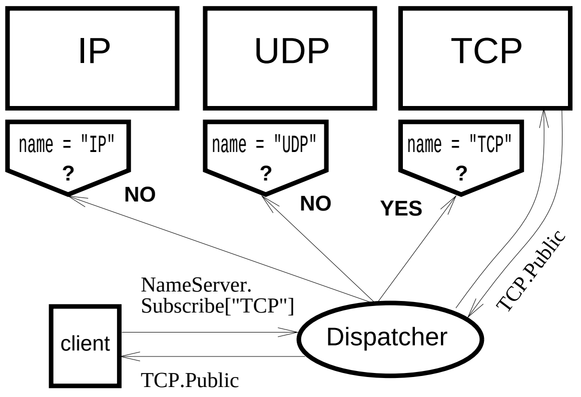

Figure 2: *This figure shows a client extension raising the* NameServer.Subscribe *event for the "TCP" service. Any domain that exports a service registers a handler for the NameServer event with a guard that compares the argument to the raise event against the exported name. The dispatcher evaluates the guards and invokes any handlers for which the guard predicate is true. The handler returns a domain capability which the client can resolve against.*

4 The core extensible services

The *SPIN* protection and extension mechanisms described in the previous section provide a framework for managing storage and control between services in the kernel. Applications, though, are ultimately concerned with manipulating physical resources such as memory and the processor. Consequently, *SPIN* provides a set of extensible services that encapsulate processor and memory resources. These are fundamental system resources that reflect the ability to execute instructions and manipulate state external to the processor.

The processor and memory services are two instances of *SPIN*'s *core* services. The core services are statically linked into the kernel and are available from the time the system boots. The services provide interfaces to basic hardware mechanisms, and, as such, are *trusted.* Trust is required because the services must access underlying hardware facilities and at times must step outside the protection model enforced by the language. Without trust, the protection and extension mechanisms described in the previous section, which themselves rely on the correct functioning of the hardware, could not function safely. Because trusted services mediate all access to physical resources, all applications and extensions must also trust the services that are trusted by the *SPIN* kernel.

The interfaces to *SPIN*'s core trusted services are extensible. Applications can extend the interfaces to a trusted service with additional code that runs in the kernel. An extension that fails to use a trusted service correctly may itself fail, but that failure is isolated to the extension and its clients. Trusted code may raise events that are handled by application-specific code. In these cases, the trusted code must not depend on the handler's actions. For example, the *SPIN* scheduler raises events that are handled by application-specific thread packages in order to start or stop a thread that had previously requested a kernel service. Although it is in the handler's best interests to respect, or at least not interfere with, the semantics implied by the event, this is not enforced. An application-specific thread package may ignore the event that a particular user-level thread is runnable, but only the application which is using the thread package will be affected.

4.1 Extensible memory management

A memory management system is responsible for the allocation of virtual addresses, physical addresses, and the mappings between the two. Other systems have demonstrated significant performance improvements from specialized or "tuned" memory management policies that are accessible through interfaces exposed by the memory management system. Some of these interfaces have made it possible to manipulate large objects, for example entire address spaces [YTR+87, KN93], or to direct expensive operations, for example page-out [HC91, MA90, CFL94], entirely from user level. Others have enabled control over relatively small objects, for example cache pages [RLB94] or TLB entries [BKW94], entirely from the kernel. None have allowed for fast, fine-grained control over the physical and virtual memory resources required by applications.

The *SPIN* core virtual memory management system decomposes memory services into three basic components: physical storage, naming, and protection. These correspond to the basic memory resources exported by processors, namely physical addresses, virtual addresses, and translations between the two. Application specific services interact with these three services to define higher level virtual memory abstractions. Although the services are fine-grained, composition is possible because the cost of accessing them from an extension is low.

Each of the three basic components of the memory system is provided by a separate service interface. The *physical address service* controls the use and allocation of physical pages. Clients raise the *Allocate* event to request physical memory with a certain size and an optional series of attributes which reflect preferences for machine specific parameters such as cache color, contiguity, volatility, I/O specific ranges for devices, or physical address ranges. A physical page represents a unit of high speed storage. It is not, for most purposes, a nameable entity and may not be addressed directly from an extension or a user program. Instead, clients of the physical address service receive a capability for the memory.

The naming of memory is accomplished through the separate abstraction of a virtual address. The *virtual address service* allocates capabilities for virtual addresses in much the same manner as is done for physical memory. *SPIN*'s core virtual address service currently supports multiple address spaces, where the implementation of the capability is composed of the virtual address and an address space identifier that makes the name unique. In a single address space system, this name would be identical to the value of the virtual address.

The relationship between virtual addresses and physical memory is implemented by a trusted *translation service*. This service interprets references to both virtual and physical addresses, constructs a mapping between the two, and installs this mapping into the processor's memory management unit (MMU). The translation service raises a set of events that correspond to various exceptional MMU conditions. For example, if a user program attempts to access an unallocated virtual memory address, then the *Translation.BadAddress* event is raised. If instead, it accesses an allocated, but unmapped virtual page, then the *Translation.PageNotPresent* event is raised. Implementors of higher level memory management abstractions use these events to implement services such as demand paging, copy-on-write [RTY^{+}87], distributed shared memory [CBZ91], or concurrent garbage collection [AL91].

Resource reclamation

The physical page service may at any time reclaim memory. A page is recalled by raising the *PhysicalPage.Recall* event with the specific physical page specified as an argument. The interface allows the handler for this event to volunteer an alternative page, which may be of less importance than the page targeted by the physical page service. The recall event is bounded to ensure that the page service's reclamation is not delayed indefinitely (as mentioned in Section 3.2, bounded events are guaranteed to return in bounded time). The translation service is responsible for maintaining coherence with resource reclamation events. If a physical page is reclaimed from an extension, any mappings to that page are also invalidated.

4.2 Extensible Processor management

An operating system's execution model provides applications with interfaces for scheduling and thread management. The implementation of the model determines its performance and functionality. Application requirements are often mismatched (in terms of performance or semantics) by the processor management interface exported by operating systems. User-level thread management systems have addressed this mismatch[WLH81, CD88, MSLM91, ABLL92], but only partially. For example, Mach's user-level C-Threads implementation [CD88], although flexible, is poorly integrated with kernel services [ABLL92]. In contrast, *scheduler activations*, which were well-integrated with the kernel, had excessively high overhead [DMVL93].

SPIN avoids the problem of defining the wrong execution model for applications by not defining one at all. Instead, *SPIN* defines set of events that coordinate processor allocation between schedulers and thread packages. Application-specific services provide the implementation of the handlers for these events through extensions that execute in the kernel. The advantage of this approach is that applications can define arbitrary thread semantics, and implement those semantics very near the actual hardware processing resources.

A scheduler multiplexes the underlying processing resources among competing contexts, called *strands*. A strand is similar to a thread in traditional operating systems in that it reflects a processor context. Unlike a thread though, a strand has no minimal or requisite kernel state other than a name. The application-specific thread package that implements the strands interface defines a strand's state. For example, a strand may be associated with a single address spaces, or multiple [BALL90], have thread-private data, or even a different set of virtual memory mappings than other strands in the same address space. These semantics can be determined by the strand package itself, which implements the machinery that saves and restores a strand's state in response to events that are raised by a scheduler. The *Checkpoint* and *Resume* events allow a strand package to save and restore strand state in response to scheduler actions.

While strands only interact with the scheduler, the scheduler interacts with many components of the kernel. For example, the virtual memory system stops a strand if it incurs a page fault, and an interrupt handler will start a strand to signal the completion of an I/O operation. The scheduler interface contains two events that can be raised by code running in the kernel to signal changes in a strand's execution state. *Block* and *Unblock* are raised to signal that a particular strand has become ineligible or eligible for execution.

A trusted scheduler implements the primary processor allocation policy between strands. Figure 3 shows the relationship between the physical processor, schedulers, strand packages, and other kernel services that interact with the scheduler. While the scheduling policy is replaceable, that it is trusted implies that it can not be replaced by an arbitrary application, *and* that its replacement may have global effects. In the current implementation, the trusted scheduler implements a round-robin, preemptive, priority policy. Additional, application-specific schedulers can be implemented on top of the central scheduler using *Checkpoint* and *Resume* events to pass control upwards. That is, a higher level scheduler presents itself to a lower level as a strand package. *Resume* allows the higher level package to schedule its own strands, while *Checkpoint* notifies the scheduler that it is no longer scheduling the processor. The *Block* and *Unblock* events, when raised on strands scheduled by higher-level schedulers will be routed by the dispatcher to the appropriate scheduling implementation. This allows arbitrary scheduling policies to be implemented within and integrated with the kernel, provided that a higher level policy does not fundamentally conflict with a lower level one.

Isolation and trust

A strand package may fail to implement the expected semantics of its interface without compromising the system's overall integrity. For example, in response to a *Checkpoint* operation raised by the virtual memory system, an errant strand package's handler may fail to save the currently running thread's user state in a way that can be later resumed. Only the specified strand will be affected, though. In general, each strand event represents a separate service contract that affects only one strand, so the failure of any handler results only in the failure of strands that depend on its functionality.

Application-specific strand packages, although executing in the kernel, only manipulate the flow of control for threads executing outside of the kernel, or *application threads*. Within the kernel, *SPIN* supplies a trusted strand package that implements the Modula-3 thread semantics [Nel91]. Applications can not change the implementation of this package. The rationale for this restriction is straightforward: the *SPIN* kernel runs code on behalf of applications, but it does not let applications control how the code will be run. Allowing applications to define the execution model for code running in the kernel could lead to a breach of the mechanisms that the kernel's protection mechanisms. For

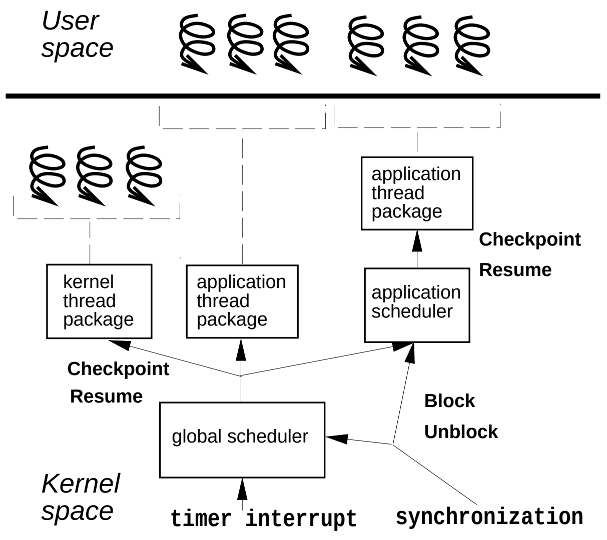

Figure 3: *This figure shows three different thread implementations with* SPIN. *Trusted kernel threads are for use within the kernel. Applications implement threads by providing thread packages that handle strand events. Additionally, application-specific scheduling policies may be implemented by installing a specialized scheduler that handles scheduling events. Synchronization (Block and Unblock) and preemption (timer interrupt) events are propagated through the scheduling hierarchy to thread packages.*

example, if an application could arbitrarily start and stop a thread while running in the kernel, then it would be impossible to make any assumptions about synchronization and progress. More fundamentally, if an application implemented the context switch routines for kernel code, then no assumptions could be made about register contents, and therefore the entire domain system, which is based on type safety, would be rendered useless.

5 Building services with *SPIN*

Application and service writers demand levels of system functionality much higher than those described in the previous section. In this section we show that the kernel and core services can be used to implement more conventional operating system abstractions, such as system calls, address spaces, and networking. These services have all been defined as extensions to the *SPIN* kernel. The performance of these services and others is described in Section 6.

5.1 Address spaces

Most conventional operating systems provide an "address space" abstraction which represents both a hardware protected addressing context and a system-wide unique name for a set of resources. Systems implement this abstraction with different mechanisms and semantics according to the needs of their intended class of applications. For example, the lazy evaluation strategies used in Mach's virtual memory system are intended to support UNIX applications [GDFR90], but are inappropriate for real-time applications.

The *SPIN* core services do not define an address space model directly, but may be used to implement a range of models using a variety of optimization techniques. For example, an extension that defines UNIX address space semantics must provide an interface for copying an existing address space, and allocating additional memory within an address space. For each new address space, the extension must allocate a new map from the translation service, which provides the interface to a hardware protection and naming context. This translation is subsequently filled in with virtual and physical address resources obtained from the core memory allocation services. An address space extension can allocate virtual memory lazily using fault events, deferring the allocation of physical memory and the related entry in the translation map.

5.2 System calls

A system call is the mechanism by which user-level code interacts with kernel code. The important property of a system call is that the hardware mediates the control transfer between the user and the kernel, allowing the kernel to safely dispatch through a large number of protected entry points. In conventional systems, system calls are statically defined when the system is built. Some systems allow new system calls to be defined at runtime through the use of privileged link operation, which makes the system call available to all applications. A few systems offer application-specific trap vectoring, whereby system call vector tables can be installed on a per application basis enabling system calls to be redirected out to user space. This mechanism serves as the foundation for building operating system emulation environments [Jon93], and is often a critical factor in their performance [Pat93].

In *SPIN*, a system call is simply an event that takes place within the context of some strand, and is reflected by the *SystemCall* event, which is raised by the kernel when the system call occurs. The dispatcher finds any handlers that should execute in response to the event (in accordance to their associated guards) and invokes them. When invoked, the handler is passed the number of the system call and the identity of the strand that caused it. The handler can access the user space registers for the strand to obtain any system call arguments. A handler blocks the thread using the scheduler interface, and dispatches to the appropriate routine within the handler's domain. Arguments referring to kernel capabilities must be internalized, as described in Section 3. Lastly, we implement protected cross-address space communication by composing protected procedure calls, which were described in Section 3.1, with system calls.

5.3 Networking protocols

We have implemented network protocol stacks for both Ethernet and ATM. The protocol stacks are structured as an inverted tree, with the network device driver being at the root. Figure 4 illustrates the use of guards and handlers for a set of in-kernel protocols. Each incoming packet is "pushed" up through the protocol graph according to events raised in each preceding layer. The handlers at the top of the graph either process the message entirely within the kernel, or copy it out to an application. In the figure, several application-specific protocol handlers are shown: "UDP count" simply counts the number of incoming UDP packets, "A.M." is an extension that implements active messages [vECGS92], "RPC" implements the network transport for a remote procedure call package, and "Video" provides a direct path for video packets from the network to the framebuffer. *SPIN*'s protocol stacks have structure similar to the *x*-kernel's [HPAO89] except that *SPIN* permits user code to be placed within the stack.

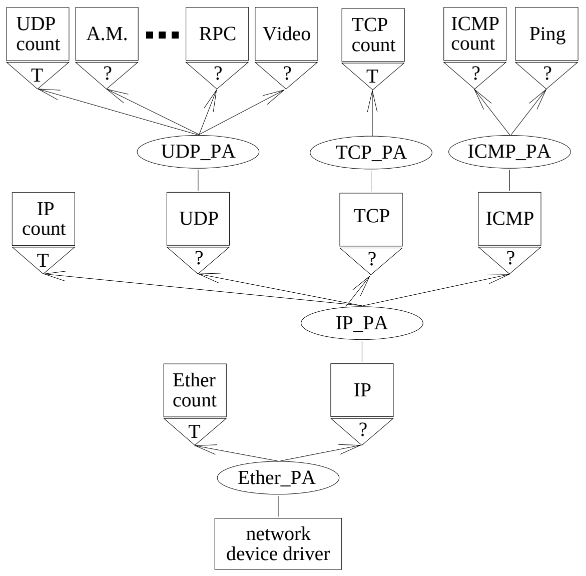

Figure 4: *This figure shows a UDP/IP stack that routes incoming network packets to application-specific endpoints within the kernel. Each guard is responsible for demultiplexing through one layer of the stack, and each handler is responsible for pushing the packet up to the next layer. "PA" indicates a PacketArrived event. Handlers are denoted by rectangles. Events raised are indicated by ovals. Guards are indicated by triangles. A guard may always return true (indicated by a "T" in the guard), or may dispatch based on a field in the packet (indicated by a "?").*

6 Performance

The previous sections have described the extensible and safe attributes of the system. In this section, we demonstrate that the basic system mechanisms perform well. Moreover, we show that these mechanisms, when composed to define application-specific services, admit a variety of functions that are important to applications which are ill-served by the types of general services found in existing operating systems.

We have conducted our experiments on Alpha 133MHz DEC AXP 3000/400 workstations with 64 MB of memory and an HP C2247-300 1GB disk-drive. For our networking experiments, we use 10Mb/sec Ethernet and ATM. Our ATM network consists of FORE TCA-100 155Mb/sec adapter cards. The FORE cards use programmed I/O and deliver substantially less bandwidth than 155 Mb/sec [BB93]. We use the DEC SRC Modula-3 compiler, release 3.3, to compile the *SPIN* kernel and extensions.

We compare the performance of operations on three operating systems running on our hardware: *SPIN*, DEC OSF/1 V2.1, and Mach 3.0+DEC OSF/1 server. We use a version of the Mach 3.0 microkernel that was ported to the Alpha, as well as a user-level UNIX server that is binary compatible with DEC OSF/1 V2.1.[2] We report performance times for operations using the Mach 3.0 kernel as well the server running on top of the kernel. All measurements were collected with the operating systems running in single-user mode.

6.1 Microbenchmarks

Microbenchmarks reveal the overhead of basic system functions, such as system call, thread management, virtual memory, and protected procedure call. They define the bounds of system performance and provide a framework for understanding larger operations.

Page faults

Applications frequently exploit the virtual memory fault path as the basis for extending system services [AL91]. For example, concurrent and generational garbage collectors rely on write faults to maintain invariants or collect reference information. In the most common use of this strategy, application code updates a data structure when the fault occurs, promotes the page to allow read-write access, and resumes the faulting thread.

A longstanding problem with fault-based strategies has been the overhead of safely reflecting the page fault to application-code, so that it can be recovered from [TL94, ALBL91]. Overheads are high because each fault requires five crossings of the user/kernel boundary (fault, notify, request-page-unprotect, reply-page-protect, restart-faulting-thread), with each saving and restoring processor state. Each transfer across the user-kernel boundary also requires transferring through a set of machine-dependent facilities that move between high-level calling conventions (eg, procedure call) and system-defined ones (eg, trap arguments). Finally, the low-level virtual memory systems in non-extensible operating systems do not directly expose interfaces that allow faults to be handled quickly.

We measured the time to write-fault a page from user space, and route the fault to some application-specific code that requests that the page be marked read-write. For DEC OSF/1, we implemented the application-specific code using signals and the *mprotect* system call. For Mach 3.0, we used the external pager interface [YTR+87]. For *SPIN*, we installed an extension that handled *VM.WriteProtect* events for the application's virtual address space, and then performed a call to the translation service requesting that the faulted page be made writeable. Table 1 shows that both Mach and DEC OSF/1 have a handling time that is one to two order of magnitude slower than *SPIN*'s, although all systems provide the same functionality.

Mach 3.0 kernel	DEC OSF/1 kernel	*SPIN* extension
1100	300	17

Table 1: *Write-fault and reprotect times for an application running at user-level. For Mach and DEC OSF/1, the fault is reflected to application code running at user-level. For* SPIN, *the fault handler executes in the kernel as an application-specific extension. The time to handle faults under the DEC OSF/1 server is not relevant, as it is most efficient to implement fault handling using the kernel primitives. All times are in microseconds.*

Thread management

Thread management packages implement concurrency control operations using underlying kernel resources. In *SPIN*, threads execute either in the kernel at user-level. Threads that execute in the kernel are implemented in terms of a trusted strands package that exports the Modula-3 thread interface. Every system call and asynchronous event is executed in its own thread, so the overhead of the kernel's thread management services is critical. Moreover, application-specific extensions also rely on threads executing in the kernel to implement their own concurrent operations.

Table 2 shows the overhead of basic in-kernel thread management operations for the three different kernel platforms. The table shows that *SPIN*'s extensible threads implementation does not incur a performance penalty for kernel execution when compared to non-extensible ones.

Kernel Thread Operation	Mach 3.0 kernel	DEC OSF/1 kernel	*SPIN* extension
Create	41	332	5
Ping-Pong	71	21	29
Terminate	18	260	7

Table 2: *Kernel thread management overhead. This table shows the time to manage kernel threads within the kernel for Mach 3.0, DEC OSF/1 2.1 and* SPIN. *Create is the time to construct and start a new thread. Ping-Pong is the time for two threads to execute a pair of block and unblock operations using the kernel's thread management operations. Terminate is the time to destroy a thread and reclaim its data structures. All times are in microseconds.*

[2] We are converting this server over to run on top of the native *SPIN* kernel. We hope to report on times using this configuration in the final paper.

Thread management overhead for application threads is also important as it determines the granularity with which threads can be used to control concurrent operations. As previously mentioned, an application thread using *SPIN* executes at user-level, although its management interface can be implemented in the kernel. Table 3 compares the performance of thread management operations using the C-Threads interface originally implemented in Mach. The times for two implementations are shown for *SPIN*. The first one, labeled "layered," is for a straight port of the Mach C-threads package to *SPIN* and relies on an extension that exports thread management primitives similar to those of the Mach kernel. That extension in turn implements the strands interface within the kernel to communicate with the scheduler. The second version, labeled "native," is for a version of C-Threads implemented directly as an extension. The extension responds to the global scheduler's *Resume* and *Checkpoint* events, but exports a conventional thread management interface to applications.

Table 3 demonstrates that *SPIN*'s direct approach to thread management (the native case) is substantially faster than the other systems. In Mach 3.0, for example, each thread creation involves four system calls: one to create the thread, one to allocate stack space, one to set its initial state, and one to start it running. In *SPIN*, these calls are folded into one extension that is accessed by a single system call.

User Thread Operation	Mach 3.0	DEC OSF/1	*SPIN* layered	*SPIN* native
Fork	50	1131	103	20
Fork,Run	233	1164	157	64
Ping-Pong	115	233	85	85
ForkJoin	338	1026	223	110

Table 3: *Overhead to use an implementation of the C-Threads interface for Mach 3.0, DEC OSF/1 2.1 and* SPIN *from a user-level application. The Mach 3.0 and DEC OSF/1 numbers are for a user-level library implementation built on top of a kernel thread management interface. (DEC-OSF/1 2.1 provides the P-Threads interface, which is a superset of the C-Threads interface used in Mach and* SPIN*.) "Fork" is the latency paid by the forking thread when spawning a new thread. "Fork,Run" is the amount of time between the call to Fork and the running of the new thread. The* SPIN *layered numbers reflect an in-kernel implementation of C-Threads using an extension that imports the Mach kernel thread interface exported by another extension. The* SPIN *native numbers are for an implementation of C-Threads that directly uses the kernel strand interface. All times are in microseconds.*

System call overhead

System call overhead is the time to cross the user-kernel boundary and then execute and return from an operating system function. In *SPIN*, a system call is reflected by a *SystemCall* event raised by the kernel's trap handler, followed by a dispatch through to an arbitrary Modula-3 procedure which has been installed as the handler for the the specific system call requested. In the Mach and DEC OSF/1 kernels, a system call flows from the trap handler through to a generic but fixed system call dispatcher, and from there to the requested system call. Our Mach-based DEC OSF/1 server handles system calls using an exception vectoring mechanism provided by the Mach kernel. On a system call trap, a thread in the server's address space is signalled by the kernel, and the system call executes within the server's address space. This implementation does not signal using the Mach kernel's IPC facilities, and as such has slightly lower overhead than the standard Mach kernel [Pat93].

Table 4 shows the system call overhead using the different operating systems. The table shows that *SPIN* is slightly slower at handling system calls than either Mach or the DEC OSF/1 kernel. The difference between *SPIN* and these other two systems is that *SPIN* forks a new kernel thread for every system call event. This is necessary because the kernel and application do not use the same strands package for their thread implementations.

All three system substantially outperform the Mach-based OSF/1 server, where the system call is implemented in another address space. This case, though, is the one that provides the same functionality as *SPIN*, namely the ability to execute arbitrary code in response to an application's system call. The table shows that system call emulation using protected procedure call is 25 times slower with the microkernel-based system than with *SPIN*.

Xfer Bytes	Mach 3.0 kernel	DEC OSF/1 server	DEC OSF/1 kernel	*SPIN* extension
0	7	309	6	12
128	9	453	8	15

Table 4: *System call overhead for Mach 3.0, Mach 3.0+OSF/1 server, DEC OSF/1 2.1 and* SPIN*. Xfer Bytes refers to the number of bytes returned from the system call. OSF/1 server numbers reflect the time for a syscall to be vectored to the user level OSF/1 server through the Mach kernel and back. All times are in microseconds.*

Cross-address space procedure call

SPIN itself does not provide a cross-address space communication mechanism. As mentioned earlier though, one can be implemented in terms of system calls (for transferring control) and domains (for protecting procedures within the kernel). A server publishes domains that serve as stubs for entering the kernel from the client address spaces, exiting into the server's, and setting up a return path. The other three systems also provide facilities for implementing cross-address space communication. Mach 3.0 uses a highly optimized IPC path, while the DEC OSF/1 kernel implements cross-address space communication using generic sockets and SUN RPC.

Table 5 shows the times to perform a protected, cross-address space procedure call on the different systems. The table shows that *SPIN*'s composite, extensible IPC service is no more costly to use than Mach's built-in service, and both are substantially faster than the SUN RPC implementation. The main difference between the

Mach 3.0 IPC path and *SPIN*'s is that Mach represents the relationship between client and server through indirect references (client port to queue to thread to address space to server stub to server procedure). In contrast, the kernel's system call handler can be specialized to provide exactly the service required (system call to stub to server stub to server procedure), and the relationship between the components can be fixed statically at link-time.

Xfer Bytes	Mach 3.0 kernel	DEC OSF/1 kernel	*SPIN* extension
0	104	845	102
128	124	873	117

Table 5: *Cross address space procedure call times for Mach 3.0, Mach 3.0+DEC OSF/1 server, DEC OSF/1 2.1 kernel and* SPIN. *Xfer Bytes refers to number of bytes returned from the system call. Mach 3.0 uses Mach IPC, and the DEC OSF/1 kernel uses SUN RPC and sockets. The time to perform cross-address space procedure calls with the DEC OSF/1 server is not relevant, as it is more efficient to use the kernel's native IPC primitives. All times are in microseconds.*

Address space management

Table 6 shows the time to create and destroy an address space using the primitives provide by the various operating systems. The time includes that to create the address space and a thread, initialize the address space with code, and start the thread. The *SPIN* address space extension performs all of these operations with a single system call that interacts with the system's trusted services. For Mach 3.0, six system calls are required to create and terminate an address space (construct address space, create thread, allocate stack, allocate space for code and data, write the code, terminate address space). For the DEC OSF/1 server running on Mach, these calls and others (eg, clear data segment, initialize file descriptors, set up signal state, etc.) occur during address space creation. The DEC OSF/1 kernel is faster than the server because it can manipulate the virtual memory maps directly.

Mach 3.0 kernel	DEC OSF/1 server	DEC OSF/1 kernel	*SPIN* extension
2596	16069	5320	1799

Table 6: *The time to create and destroy an address space. For Mach 3.0, we use the kernel's native task and thread management operations. For the DEC OSF/1 systems, we use the fork() and exit() system calls. For* SPIN, *we use the space and thread operations described earlier. All times are in microseconds.*

Networking

A common problem with networking protocols is that the operating system introduces too many layers between the application and the physical device [MB93]. This often results in networking performance that is substantially worse than the raw hardware is capable of delivering. *SPIN* allows code to execute close to the network interface, eliminating much of the overhead that occurs in traditional network architectures. We have implemented a suite of networking protocols for *SPIN* that interface to both a 10Mb/sec Ethernet, and a 155 Mb/sec ATM network. As mentioned, our ATM network interface cards use programmed I/O, so the maximum bandwidth is generally limited by the rate with which the CPU can read data from the network adapter. With our hardware configuration, we have been unable to achieve greater than 53Mb/sec when transferring data reliably between two device drivers.

	DEC OSF/1 kernel	*SPIN* extension	*SPIN* bounded
Ethernet	840	579	510
ATM	631	332 (raw 162)	241 (raw 100)

Table 7: *Round trip network RPC time. This table shows the latency to perform a simple remote procedure call using UDP/IP between a pair of applications on machines connected by a network. For* SPIN, *we report the times for both UDP/IP and a "raw" protocol implemented directly on top of ATM's segmentation and reassembly layer (AAL 4/5). DEC OSF/1 does not allow applications to use raw ATM packets. The column labeled "*SPIN *bounded" reflects the overhead to send between a pair of extensions that have bounded execution time. In these cases, the system runs the extension synchronously, rather than asynchronously. The improvement in latency is due to not having to create and schedule new threads of control on the sender and the receiver. We do not report networking times for Mach 3.0 or the DEC OSF/1 server as our version of Mach on the Alpha does not support the FORE card. All times are in microseconds.*

Table 7 shows the latency of a cross-machine remote procedure call using UDP/IP on the different systems. All measurements reflect the time to transfer a small packet between two remote applications. For DEC OSF/1, the application code executes at user-level, and each packet sent involves a trap and several copies as the data moves across the user/kernel boundary. On the receive side, a process scheduling operation and context-switch occurs. For *SPIN*, the application code executes as an extension in the kernel, where it has low latency access to both the device and data. Each incoming packet causes a series of events to be generated for each layer in the UDP/IP protocol stack (Ethernet, IP, UDP). We also report the round-trip latencies for a protocol based directly on top of AAL 4/5.

The table shows that cross-machine communication has substantially lower latency when the target and source processes are able to send directly from the kernel. The minimum round trip time (device driver to device driver) using our hardware is roughly 400 μsecs on Ethernet and 100 μsecs on ATM. While *SPIN*'s event structure incurs some cost in order to safely dispatch an incoming packet to an extension, the cost is much smaller than that in a traditionally structured system. The final column in the table illustrates part of the runtime cost involved in running untrusted code from within the kernel. When the execution time of an extension can be bounded, the dispatcher can avoid spawning a new thread of control for each event.

Table 8 shows the cross-machine, user-to-user bandwidth that we achieve using the systems. *SPIN* provides user-to-user access through in-kernel extensions, whereas DEC OSF/1 requires a pair of user-level processes. *SPIN* and the DEC OSF/1 kernel are both able to saturate a 10Mb/sec Ethernet. On ATM, *SPIN*'s UDP/IP protocol stack achieves 41 Mb/sec. Extensions that bypass UDP/IP, and instead use a protocol based on AAL 4/5 segmentation and reassembly, achieve 53 Mb/sec over ATM.

We measured processor utilization on the sender and found that an application sending data at the maximum Ethernet transfer rate consumed 35% of the processor using DEC OSF/1, and 20% using *SPIN*. For ATM, sender utilization was 82% with DEC OSF/1 and 55% with *SPIN*. The difference is due to the fact that each packet sent requires less CPU time using *SPIN*, with the difference attributable to the overhead of transferring control and data across the user/kernel boundary.

	DEC OSF/1 kernel		*SPIN* extension	
	Bwidth	CPU %	Bwidth	CPU %
Ethernet	8.9 Mb/s	35	8.9 Mb/s	20
ATM	25 Mb/s	82	41 Mb/s	55

Table 8: *The user-to-user networking bandwidth and sender CPU utilization for Ethernet and ATM.*

6.2 A networked video client and server

As mentioned in the introduction, we have used *SPIN* to implement a networked video system consisting of a server and client viewers. The server consists of three extensions, one that uses the *SFS* file system to read video frames from the disk, another that sends the video out over the network, and a third that registers itself as a handler on the *SendPacket* event, transforming the single send into a multicast to a list of clients. The server transmits 30 frames per second to each client. On the client, an extension awaits incoming video packets using the structure shown in Figure 4. The client extension decompresses the image and displays it directly to the screen buffer.

The advantage of this structure is that video frames on the server require minimal processing as they travel from disk to network. In addition, the server supports multiple concurrent streams with only the additional overhead of having to send an extra packet per stream. On the client, incoming network packets are checksummed, decompressed, and written to the screen in a single pass over the data. The result is a system structure in which each stream consumes few processing resources on the client and server.

Table 9 shows the processor utilization on the server as a function of the number of client streams for our video system running over Ethernet. As a point of comparison, we have implemented an equivalent client and server on the DEC OSF/1 kernel with user-level code and sockets. Server utilization is much lower for *SPIN* then for the conventional kernel, which incurs much greater overhead for each packet sent. The figures in the table demonstrate that *SPIN*'s basic kernel architecture can be extended to provide scalable video service.

# streams	DEC OSF/1 kernel CPU %	*SPIN* CPU %
1	28	5
5	64	19
10	75	37
15	78	55
20	NA	72

Table 9: *Server utilization as a function of the number of client video streams for the* SPIN *video service. DEC OSF/1 was unable to support 20 streams or more of video, and returned an error to the application.*

7 Conclusions

The *SPIN* operating system demonstrates that it is possible to achieve good performance in an extensible operating system without compromising safety. The system provides a set of efficient mechanisms for extending services, as well as a core set of extensible services. Co-location, type safety, logical protection domains and dynamic binding allow extensions to be dynamically defined and accessed at the granularity of a procedure call. Safety is ensured through a capability-based protection system. Extensibility is provided through an event-based invocation mechanism that provides a level of indirection between system components.

In the past, system builders have relied on the programming language simply to translate operating system policies and mechanisms into machine code. With the choice of a programming language having the appropriate features, we believe that operating system implementors can begin to more heavily rely on the language, the compiler, and runtime mechanisms to help provide system services having excellent performance.

References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[AL91] W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 108–120, April 1991.

[B$^+$92] D. L. Black et al. Microkernel Operating System Architecture and Mach. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[BB93] Jose C. Brustoloni and Brian N. Bershad. Simple Protocol Processing for High-Bandwidth Low-Latency Networking. Technical Report CMU-CS-93-132, Carnegie Mellon University, March 1993.

[Ber93] Brian N. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, May 1993.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, November 1994. USENIX Assoc.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. 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.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, October 1991.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-54, Carnegie Mellon University, February 1988.

[CD94] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, pages 179–194, November 1994.

[CFL94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, November 1994. USENIX Assoc.

[CHL91] Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced Developement of Systems Software. Technical Report CMU-CS-91-187, Carnegie Mellon University, 1991.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 319–327, October 1994.

[CZ83] David R. Cheriton and Willy Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[DEFS88] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, and William E. Summersfeld. The Zephyr Notification Service. In *Proceedings of the 1988 USENIX Conference*, February 1988.

[DMVL93] Paul-Barton Davis, Dylan McNamee, Raj Vaswani, and Ed Lazowska. Adding Scheduler Activations to Mach 3.0. In *Proceedings of the USENIX Mach III Symposium*, pages 119–136, 1993.

[Dra93] Richard Draves. The Case for Tun-Time Replaceable Kernel Modules. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993.

[EK95] Dawson Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[EKO94] Dawson Engler, M. Frans Kaashoek, and James O'Toole. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the 1994 European SIGOPS Workshop*, September 1994.

[Fel92] Edward W. Felten. The case for application-specific communication protocols. In *Intel Supercomputer Systems Technology Focus Conference*, pages 171–181, April 1992.

[FP93] Kevin Fall and Joseph Pasquale. An Implementation of UNIX on an Object-Oriented Operating System. In *Exploiting in-kernel data paths to improve I/O throughput and CPU availabilit*, pages 327–333, January 1993.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[GMS77] C.M. Geschke, J.H. Morris, and E.H. Satterthwaite. Early Experiences with Mesa. *cacm*, 20(8):540–553, August 1977.

[HC91] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 187–197, April 1991.

[HP94] J.S. Heidemann and G.J. Popek. File-System Development with Stackable Layers. *cacm*, 12(1):58–89, February 1994.

[HPAO89] Norman C. Hutchinson, Larry Peterson, Mark B. Abbott, and Sean O'Malley. RPC in x-kernel: Evaluating New Design Techniques. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[Jon93] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Call. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[KEH93] D. Keppel, S.J. Eggers, and R.R. Henry. Evaluating Runtime-Compiled, Value-Specific Optimizations. Technical Report UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[KN93] Yousef A. Khalidi and MichaelN. Nelson. An Implementation of UNIX on an Object-Oriented Operating System. In *Proceedings of the 1993 Winter USENIX Conference*, pages 469–480, January 1993.

[LCC94] Chao Hsien Lee, Meng Chang Chen, and Ruei Chuan Chang. HiPEC: High performance external virtual memory caching. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 153–164, Monterey, CA, November 1994. USENIX Assoc.

[LHfL93] Jay Lepreau, Mike Hibler, Bryan ford, and Jeffrey Law. In-Kernel Servers on Mach 3.0: Implementation and Performance. In *Proceedings of the Usenix Mach III Symposium*, pages 39–56, 1993.

[Luc94] Steven Lucco. High-Performance Microkernel Systems. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, page 199, November 1994.

[MA90] Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.

[MB93] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[Mos94] H. Mossenbock. Extensibility in the Oberon System. *Nordic Journal of Computing*, 1(1):77–93, February 1994.

[MP89] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.

[MRT+90] Sape J. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse, and H. van Staveren. Amoeba – A Distributed Operating System for the 1990's. *IEEE Computer Magazine*, May 1990.

[MSLM91] Brian Marsh, Michael Scott, Thomas LeBlanc, and Evangelos Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[Nel91] Greg Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[OPS+93] Brian Oki, Manfred Pfleugl, Alex Siegel, , and Dale Skeen. The Information Bus – an Architecture for Extensible Distributed Systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 58–68, December 1993.

[Pat93] Simon Patience. Redirecting Systems Calls in Mach 3.0, An Alternative to the Emulat or. In *Proceedings of 3rd USENIX Mach Symposium*, April 1993.

[PB94] Przemyslaw Pardyak and Brian Bershad. A Group Structuring Mechanism for a Distributed Object Oriented Language Objects. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Giend, M. Guillemont, F. Herrmann, P. Leonard, S. Langlois, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[RDH+80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[RLB94] Theodore H. Romer, Dennis Lee, and Brian N. Bershad. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 255–266, Monterey, CA, November 1994. USENIX Assoc.

[ROKB95] Theodore Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 1995.

[RTY+87] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[SBC93] Daniel Stodolsky, Brian N. Bershad, and Brad Chen. Fast Interrupt Priority Management for Operating System Kernels. In *Proceedings of the 2nd Usenix Workshop on Microkernels and Other Kernels*, September 1993.

[SMP92] Andrew Schulman, David Maxey, and Matt Pietrek. *Undocumented Windows*. Addison-Wesley, 1992.

[SS94] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[TL94] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 145–156, October 1994.

[TSS88] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr. Firefly: a multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[VGA94] A. Vahdat, P. Ghormley, and T.E. Anderson. Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report UCB CS-94-842, University of California, Berkeley, December 1994.

[WB92] Bob Wheeler and Brian N. Bershad. 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.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw–Hill, 1981.

[YBMM94] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[YTT89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In S. Cook, editor, *Proceedings ECOOP '89*, pages 89–106, Nottingham, July 10-14 1989. Cambridge University Press.