A User-Level Unix Server for the *SPIN* Operating System

David Dion

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

October 24, 1996

Abstract

An operating system that emulates Unix supports a wide range of popular applications. This paper describes the *SPIN* Unix Server, an implementation of Unix for the *SPIN* operating system. The *SPIN* Unix Server is a user-level application program supported by extensions that are dynamically linked into the *SPIN* kernel. It exports a traditional Unix system interface and provides backwards compatibility to Unix applications. In addition, applications can dynamically customize the Unix interface to optimize the performance of critical services. *SPIN* and the *SPIN* Unix Server are implemented on DEC Alpha workstations.

1 Introduction

An important factor in the acceptance of a new operating system is the range of applications it supports. For instance, a top priority in the design of Windows NT was to run all existing Windows and MS-DOS programs [Custer 93]. Although research operating systems rarely demand backwards compatibility with commercial applications, they still benefit from exporting a well-known, widely available programming interface. This paper describes the *SPIN* Unix Server, an implementation of Unix for the *SPIN* Operating System.

The *SPIN* Unix Server is an architecture for Unix *emulation*. Emulation allows Unix applications to execute on *SPIN* as if they were executing on a native Unix operating system. The *SPIN* Unix Server is a large, user-level program which emulates Unix by exporting the Unix system call interface to client applications. The server architecture is transparent to clients, as it offers binary compatibility and identical services to a traditional monolithic Unix system.

While *SPIN* supports client applications through the Unix server, the server runs on *SPIN* by means of a separate emulation. The *SPIN* Unix Server was actually developed for the Mach 3.0 microkernel [Golub et al. 90]. It has been ported to *SPIN* using a Mach emulation layer dynamically embedded into the kernel. Hence, *SPIN* emulates Unix in the context of an emulation of Mach.

This paper focuses on the implementation of the *SPIN* Unix Server architecture. This involves both the Mach emulation and utilization of the server for the Unix emulation. An evaluation of the architecture is included, as well as a technique for dynamically optimizing the emulation for critical paths in an application.

1.1 The rest of this paper

The next section gives some background on emulating Unix. In Section 3, the *SPIN* operating system is described. Section 4 presents the architecture of the *SPIN* Unix Server. Performance is evaluated in Section 5, and related work is discussed in Section 6. Finally, Section 7 concludes.

2 Background

Unix is an ideal emulation target for two main reasons. First, it is recognized more for its programming interface than its interaction with low-level machine resources. This allows the emulation to occur at a relatively high level of abstraction—the system call interface. Second, the Unix programming interface is simple, popular, and readily available. Unix implementations exist on virtually all architectures and the source code for several versions is publicly available.

Methods for emulating Unix on a foreign operating system target three main goals:

- extensibility: can the emulation be modified without rebooting or rebuilding the system?
- performance: do applications perform on the emulation comparably to on a native system?
- safety: can the emulation damage the system, or can an application corrupt the emulation?

Some earlier systems [Ousterhout et al. 88, Rashid et al. 89] export a Unix programming interface directly from the kernel. Systems which adopt this approach typically achieve the performance and protection of a traditional monolithic system, but sacrifice flexibility. As extensibility has evolved into an important trend in operating system design, Unix functionality has been separated from the kernel and spread among libraries, server applications, and kernel extensions. Library emulations [Khalidi & Nelson 92] generally have excellent performance (provided calls to native system services can be minimized), since system calls can be reduced to local procedure calls. However, libraries are inherently unsafe from applications. Errant or malicious applications can corrupt library data, causing unpredictable failures or even breakdowns in system security. User-level servers in separate address spaces [Golub et al. 90] protect emulators from applications. Unfortunately, the client/server model requires frequent context switching and transfer of data across address space boundaries, resulting in poor performance. Emulations through kernel extensions [Bricker et al. 91] achieve performance comparable to libraries, and the emulation state is protected from applications. However, kernel extensions are historically trusted entities. Although the kernel extension is protected from the application, the kernel is not protected from the kernel extension.

Emulation in the *SPIN* operating system has the advantages of these approaches without the disadvantages. *SPIN* [Bershad et al. 95] is an operating system which can be safely and dynamically specialized to the needs of applications via untrusted kernel extensions. Extensions are linked directly into the kernel address space, where they may access system resources and services with low latency. Protection in *SPIN* is based on restricted dynamic linking and the type-safety of Modula-3 [Nelson 91], the programming language in which *SPIN* and its extensions are written.

An emulation constructed from *SPIN* extensions harnesses the safety, flexibility, and performance of the *SPIN* extension architecture. Emulation extensions execute in the kernel address space, isolating them from corruption from applications. At the same time, the *SPIN* protection model guards the kernel from abusive extensions, allowing untrusted emulations to be linked into the kernel. Performance is optimized by migrating services into the kernel, reducing the need for expensive context switching. Extensibility is fundamental to the extension architecture; applications can customize emulations by dynamically linking additional extensions into the kernel.

Implementing Unix in the *SPIN* extension framework can take several forms. A complete Unix emulation could be developed in Modula-3 and dynamically linked into the kernel. This approach would approximate traditional monolithic systems and fully harness the advantages of *SPIN* extensions; however, it involves reimplementing an entire Unix infrastructure in Modula-3*. An alternate approach is to implement Unix as a server application for *SPIN*. Source code from publicly available versions of Unix could be reused, since user-level *SPIN* applications can be written in any language. Nonetheless, extracting and debugging a Unix server from a monolithic system requires a significant amount of development. The third approach, which is described in the rest of this paper, is to port an existing Unix server to *SPIN*. Rather than implementing Unix, kernel extensions are used to emulate Mach 3.0, which the Unix server was originally developed for. This approach allows a large amount of code to be reused at relatively low cost. Furthermore, the *SPIN* extension architecture can be used to improve the historically poor performance of user-level servers.

3 Overview of *SPIN*

The *SPIN* kernel consists of a set of core system services and an extension mechanism. Core system services include threads, memory management, and a low-level machine interface. The extension mechanism allows foreign code to be linked into the kernel address space at run-time. *SPIN* extensions access system services with procedure calls and system resources with loads and stores. They are not hindered by costly protection mechanisms, such as address space switches, in their interaction with the kernel. Instead, *SPIN* relies on language features for protection. *SPIN* and its extensions are written in the safe subset of Modula-3 [Nelson 91], a type-safe programming language. Using language features for protection allows much of the protection overhead to be incurred at compile-time and link-time.

The *SPIN* extension mechanism has two parts: installation and invocation. Installation involves loading an extension into the kernel and granting it access to system resources and services. Invocation ensures that extension code is executed at appropriate times. Each service is based on flexibility, but includes mechanisms to dynamically protect the kernel.

3.1 Installation

Installation is controlled by the *dynamic linker* [Sirer et al. 96]. The dynamic linker accepts extensions as partially resolved object code generated by a trusted Modula-3 compiler. Extensions must provide capabilities for interfaces they link against. Capabilities are requested in the Modula-3 build environment and, if granted, embedded into the downloaded extension. If an extension fails to present a valid capability for a link request, it is rejected. Dynamically linked extensions execute just as statically linked kernel code. They may access data and call procedures in any visible kernel interface (i.e. an interface successfully linked against).

*A Unix emulation of this type is currently under development.

3.2 Invocation

Invocation is controlled by the *SPIN dispatcher* [Pardyak & Bershad 96]. The dispatcher communicates *events* to *event handlers*. An event is an announcement of or a request for a change in system state. An event handler is a procedure which acts in response to an event. Dynamically linked extensions integrate themselves into a running system by installing new handlers on system events. For example, a system call is announced in the *SPIN* kernel by the `MachineTrap.Syscall` event, which is raised by low-level trap handling facilities. A Unix emulation would respond to the `MachineTrap.Syscall` event by providing a new handler and registering it with the dispatcher.

In general, dynamically installed handlers should not be invoked at every instance of the event. For instance, the `MachineTrap.Syscall` event will be raised for every system call from every user-level application; however, a system call handler for a Unix emulation should not handle a system call from an MS-DOS application. Event instances may be filtered using one or more *guards*. A guard defines a predicate to be evaluated before the handler is invoked. If all of a handler's guards evaluate to `TRUE` when the event is raised, then the handler is executed. Otherwise, the handler is ignored. Thus, a Unix emulation can use a guard to determine which system calls actually come from Unix application.

SPIN events are defined by procedures in interfaces, and in fact all procedures define events. Calling a procedure is synonymous with raising the event. The default handler of an event is the implementation of the procedure, and a dynamically installed handler is an alternate implementation of the procedure. Thus, the `MachineTrap.Syscall` event is actually a procedure named `Syscall` in the `MachineTrap` interface. The default handler of the `MachineTrap.Syscall` event is the implementation of the `MachineTrap.Syscall` procedure. A dynamically installed handler for the `MachineTrap.Syscall` event is a procedure with the same parameters and return type as specified in the `MachineTrap` interface. A guard is a procedure which takes the same parameters as the handler and event it is associated with; however, guards must return either `TRUE` or `FALSE`. Figure 1 shows the definition of the `MachineTrap.Syscall` event and an extension which emulates the Mach `vm_allocate` system call.

Extensions can register handlers for events in any visible interface. However, the default handler of the event can set restrictions on the handling of the event through an *authorizer*. An authorizer is a procedure called by the dispatcher when an extension registers a new handler for an event. It may restrict the invocation of the handler with an *imposed guard*. An imposed guard is a predicate applied for the purpose of dynamically limiting access to an event. For example, the module defining the default handler of the `MachineTrap.Syscall` event has an authorizer which verifies the identity of the extension. If the identity is trusted, the authorizer allows the extension to handle arbitrary system calls. If the identity is not trusted, the authorizer imposes a guard restricting applications for which the extension can handle system calls. This prevents application-specific extensions from errantly or maliciously modifying the execution environment of other applications.

4 Emulating Unix with the *SPIN* Unix Server

The *SPIN* Unix Server architecture comprises two concurrent emulations. First, the server is supported by an emulation of the Mach microkernel. The Mach emulation consists of a kernel extension which distributes requests to preexisting *SPIN* services. Second, DEC OSF/1 Unix is emulated in the context of the *SPIN* Unix Server and the Mach emulation. The *SPIN* Unix Server

```
(* Interface to trap handling *)
INTERFACE MachineTrap;

(* declaration of the MachineTrap.Syscall event *)
PROCEDURE Syscall(s: Strand.T; VAR ms: MachineCPU.SavedState);
  (* a Strand.T is the unit of scheduling in SPIN *)

END MachineTrap.
```

```
(* Implementation of Mach emulator *)
MODULE MachEmul;

(* procedure which handles Mach system calls *)
PROCEDURE SyscallHandler(s: Strand.T; VAR ms: MachineCPU.SavedState) =
  BEGIN
    CASE ms.regs[SyscallNumber] OF
    ...
    | -65 (* vm_allocate *)
      VMHandlers.vm_allocate(s, ms);
    ...
    ELSE
      Error.UnhandledSyscall(s, ms);
    END;
  END SyscallHandler;

(* guard for the system call handler *)
PROCEDURE SyscallGuard(s: Strand.T; VAR ms: MachineCPU.SavedState) : BOOLEAN
  BEGIN
    RETURN IsMachSyscall(ms.regs[SyscallNumber) AND IsMachTask(s);
  END SyscallGuard;

(* module initialization procedure *)
BEGIN
  (* register system call handler with dispatcher *)
  WITH event = MachineTrap.Syscall,
       handler = SyscallHandler,
       guard = SyscallGuard DO
    Dispatcher.InstallHandler(event, guard, handler);
  END;
END MachEmul.
```

Figure 1: *The Mach emulation extension registers a system call routine and a guard with the dispatcher. The handler and the guard are attached to the* `MachineTrap.Syscall` *event, which announces a system call trap from user space. Both the handler and guard have the same signature as the* `Syscall` *procedure defined in the* `MachineTrap` *interface, except that the guard returns a boolean.*

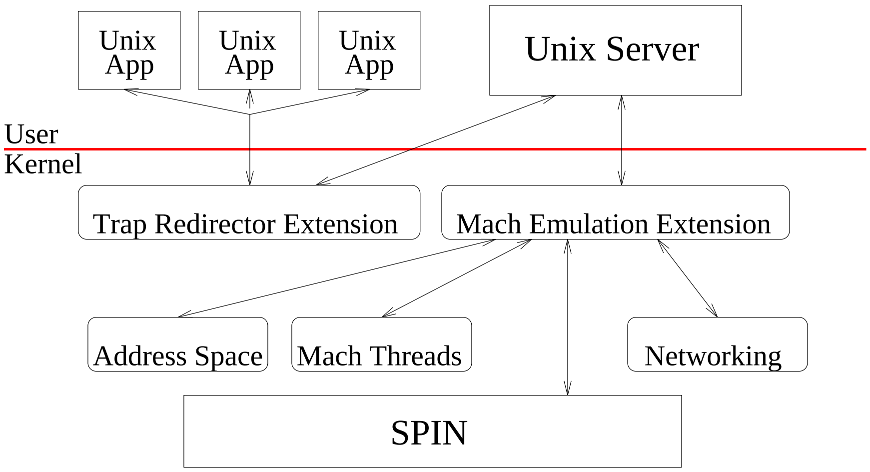

Figure 2: *In the* SPIN *Unix Server implementation, application system calls are intercepted by the Trap Redirector extension and redirected to the user-level Unix Server. The Unix Server executes the system calls, possibly requesting Mach kernel services. Mach kernel requests are intercepted by the Mach Emulation Extension, which satisfies them by relying on the* SPIN *kernel and other extensions.*

architecture is shown in Figure 2.

4.1 Emulating Mach

The *SPIN* Unix Server is derived from the BSD4.3 Single Server, which was implemented for the Mach 3.0 microkernel. Although the *SPIN* Unix Server maintains the same general design as the BSD4.3 Single Server, some significant changes have taken place. For the purposes of this discussion† the most significant modification is the removal of Mach IPC. All Mach messages have been converted to system call traps.

The Mach microkernel system call interface [Loepere 92] is considerably different than the Unix system call interface. In general, Mach system calls request lower-level services than Unix system calls. For instance, whereas the Unix system call `brk()` requests that a process data segment be enlarged, the Mach system call `vm_allocate()` specifies precisely where new virtual memory should be allocated. The problem in running the Unix server on *SPIN* is identifying the subset of Mach system calls which are required and then implementing acceptable semantics in the context of *SPIN* services.

Identifying the set of required Mach system calls was accomplished with a system call tracing extension. The Server was executed in a user-level address space on *SPIN*. In response to each system call trap, the trace extension printed the number of the system call and returned a Mach error code. As system calls were identified, handlers were added to the trace extension and the corresponding system call numbers were no longer printed. Over time, the trace extension evolved into an emulation of precisely the Mach system calls required to support the Unix server. A list of the currently emulated Mach system calls is provided in Table 1. In general, these calls fall into one of four areas: ports, threads, devices, and tasks and virtual memory. The remainder of this section briefly describes how these four areas mapped to *SPIN* services.

†The evolution of the *SPIN* Unix Server is discussed with slightly more detail in Section 6.1.

VM/Tasks	Ports	Devices	Threads	Miscellaneous
vm_write	mach_reply_port	device_read	thread_rendezvous	host_kernel_version
vm_read	mach_task_self	device_read_inband	thread_switch	processor_set_default
vm_inherit	mach_thread_self	device_read_request	thread_create	host_processor_set_priv
vm_region	mach_host_self	device_read_overwrite	thread_set_state	host_info
vm_statistics	mach_port_move_member	device_read_overwrite_request	thread_resume	set_softclock
vm_map	task_set_special_port	device_write		poke_softclock
vm_allocate	mach_port_allocate	device_write_request		get_mach_time
vm_deallocate	mach_port_deallocate	device_write_inband		
vm_protect	mach_port_insert_right	device_write_request_inband		
task_create	mach_port_allocate_name	device_set_status		
task_info	task_get_special_port	device_get_status		
task_terminate	task_get_master_port	device_open		
		device_close		

Table 1: *A list of emulated Mach system calls. Table 3 in Appendix A provides an annotated list with data on the implementation of each call.*

4.1.1 Ports

The Mach port is primarily a handle for message-passing. Since Mach message-passing was removed from the *SPIN* Unix Server, the implementation of port-related system calls is not critical. However, when a Mach application holds a port for a system resource, it holds a capability to interact with that resource. For instance, if a Mach application accesses the machine console with the `device_open()` system call, the kernel will return a port. Then, when the application reads or writes from the console, it will pass the console port as a capability to request the respective service.

In *SPIN*, capabilities are represented by reference variables and protected by the type-safe properties of Modula-3. However, when capabilities to a system resource are passed to a user-level application, the in-kernel language protection is no longer valid. *SPIN* provides a utility which allows capabilities to be *externalized*, or safely passed out of and back into the kernel address space. In this respect, externalized references are user-level capabilities for *SPIN* kernel services and resources, just as ports are user-level capabilities for Mach kernel services and resources. Hence, externalized references can be used to emulate Mach ports that represent kernel capabilities.

4.1.2 Threads

The *SPIN* Unix Server has multiple threads of execution managed primarily by the Mach C Threads package. The Mach C Threads package is a library of user-friendly thread operations. It is implemented through the Mach microkernel thread interface. A preexisting *SPIN* kernel extension provides a thread interface very similar to the Mach thread interface; consequently, emulating Mach threads was fairly straightforward.

4.1.3 Devices

The Mach device interface was complicated to emulate for a few reasons. First, *SPIN* does not present a unified device interface. *SPIN* kernel devices vary slightly in both the operations they export and the signatures of these operations. For instance, an array of bytes can be written to a disk device, but data written to the ethernet device must be packaged in an `mbuf`. The problem of identifying the proper device operation is solved using the port parameter to the system call. As described in Section 4.1.1, tasks request Mach services from a device through a port. In *SPIN*, ports are analogous to capabilities, which are implemented as typed references to kernel objects. Using Modula-3 run-time type identification, the type of the device can be identified through the type of the device capability. Handlers for device system calls multiplex among different versions of the same operation depending on the type of the device being acted upon.

The second complication in emulating the Mach device interface was the variation in standard services. Mach offers alternatives to its fundamental device system calls. For instance, the `device_write()` system call is supplemented by

- `device_write_inband()`: compact the data into the Mach message which communicates the request to the device[‡].
- `device_write_request()`: write the data asynchronously.
- `device_write_request_inband()`: write the data asynchronously and transfer it to the kernel inband.

To simplify the emulation, these three variations are implemented in the same handler. The loss of semantics is minimal since inband data transfer applies only to message passing and the server already implements an asynchronous device protocol in terms of the kernel device interface. Other Mach device calls offer similar alternatives.

The third complexity in emulating the Mach device interface was mapping low-level details between the server virtual devices and *SPIN* devices. For instance, the *SPIN* console device treats break characters differently than the server `tty` implementation. Also, the Mach `device_get_status` and `device_set_status` calls need to be mapped to the ioctls exported by each device. Currently, many of these mappings are fragile; they will require review once the *SPIN* device interface is more mature.

4.1.4 Tasks and Virtual Memory

Support for Mach tasks[§] and virtual memory is provided by a *SPIN* address space extension. The address space interface includes creation of an address space, allocation and deallocation of virtual memory, protection of a virtual memory region, and transfer of data from the kernel into a user-level address space. Although these services are sufficient for basic operations, they lack some of the features exported by the Mach task and virtual address interfaces. For instance, Mach allows virtual memory regions to be mapped between tasks and inherited during the creation of new tasks. The Unix server makes use of memory mapping to reduce the cost of transferring data between address spaces. The Mach virtual memory model also has a more efficient implementation, with features such as copy-on-write. Copy-on-write is especially effective in reducing the cost of creating new tasks.

Emulation techniques were used to disguise the lack of advanced features in the *SPIN* address space extension. The emulation involves maintaining extra state and implementing the additional services. Boundary data and inheritance attributes must be recorded for each region allocated. Inheritance and mapping can then be replaced by aggressively copying the appropriate regions of memory. Aggressive copying is correct as long as sharing is not involved, but it suffers severely

[‡]With Mach IPC removed, this call has the same functionality as `device_write()` but with slightly different parameters.

[§]A Mach task is an execution environment and the basic unit of resource allocation [Golub et al. 90].

in performance. Accordingly, as the demand on task and virtual memory support grows, it is becoming apparent that side-stepping these Mach optimizations is more costly and complex than implementing them. A better approach to emulating the Mach virtual memory and task services would be to identify in advance the major services required and implement them for *SPIN* in a kernel extension. The initial development cost would be warranted by the more elegant emulation and the performance improvement.

4.2 Emulating Unix

The Unix emulation consists mainly of the *SPIN* Unix Server, which exports the system call interface which DEC OSF/1 applications require. However, system call traps land in the kernel, not in a user-level server application. Hence, some means of transparently forwarding application system call traps to the Unix server is necessary for binary compatibility.

System calls are forwarded to the Unix server by a *Trap Redirector* extension. The Trap Redirector is an extension composed of two cooperating handlers: the *App Handler* and the *Server Handler*. Together the handlers implement a remote procedure call facility to redirect application system calls to the Unix server. The App Handler intercepts system calls from Unix applications and prepares them for reflection to the *SPIN* Unix Server. The Server Handler forwards Unix system calls to the server and transfers return values to applications. The Trap Redirector extension is illustrated in Figure 3.

4.2.1 The Server Handler

Each client process controlled by the *SPIN* Unix Server is assigned a server thread. Server threads are responsible for executing all system calls from their respective processes. Server threads handle client system calls in three steps: transferring the system call trap state to the server, executing the system call, and transferring the return state back to the application.

A server thread drops into the kernel to wait for a client system call via the `get_next_exception()` system call. In the kernel, the `get_next_exception()` system call is intercepted by the Server Handler. The Server Handler blocks the server thread until its process makes a system call. When a system call occurs, the server thread is woken up. It acquires the system call trap state from the application thread and transfers it to the server address space.

The server thread returns from the `get_next_exception()` system call and executes the client system call in the server address space. While executing the system call the server thread may invoke Mach system calls to request kernel services. These system calls are handled by the Mach emulation, as described in Section 4.1. When the system call is complete, the server thread calls the `get_next_exception()` system call with the system call return state as an argument.

The `get_next_exception()` system call is again intercepted by the Server Handler. It copies the system call return state from the server address space and transfers it to the application thread. The server thread is then blocked until the next system call from its application.

4.2.2 The App Handler

While the Mach emulation intercepts Mach system calls from the *SPIN* Unix Server, the App Handler intercepts Unix system calls from Unix applications. When a Unix system call is identified and intercepted, it must be redirected to the Unix server. The App Handler locates the server thread

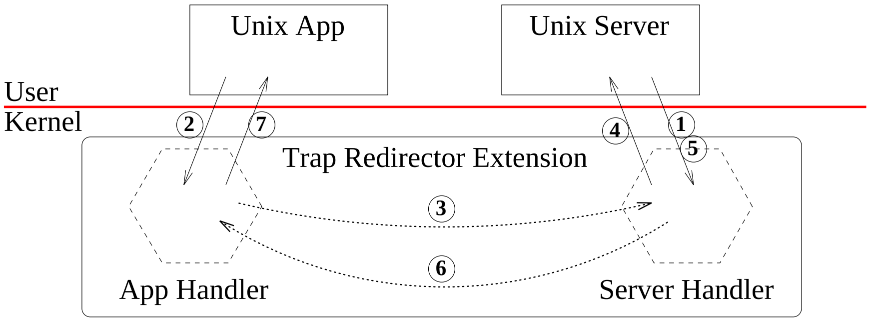

Figure 3: *The Trap Redirector extension forwards Unix system calls to the user-level server in the following steps: (1) A server thread drops into the kernel to await an application system call. It blocks in the Server Handler. (2) The application makes a system call. It is intercepted by the App Handler. (3) The trap state is transfered from the App Handler to the Server Handler and the application thread is blocked. (4) The server thread returns to the server with the trap state. It executes the system call. (5) The server thread returns to the kernel with the return state. It is intercepted by the Server Handler. (6) The return state is transfered to the App Handler and the server thread is blocked until the next system call. (7) The application thread returns to the application with the return state.*

.

for the current application thread and grants it a reference for the system call trap state. It then wakes up the server thread and blocks the application thread until the system call is complete. When the application thread wakes up, it acquires the return state of the system call and loads the appropriate registers¶ The application thread then returns to the application address space to continue running the client program.

5 Performance

This section evaluates the performance of the *SPIN* Unix Server. It is necessary to bear in mind that the *SPIN* Unix Server was not implemented for optimal performance. Other approaches, such as an emulation implemented entirely in *SPIN* extensions, would reap better performance than a user-level server. Nonetheless, a different server-based emulation, such as the Microsoft Win32 API, may be an important *SPIN* project in the future. For this reason, it is valuable to quantify the performance of the *SPIN* Unix Server and identify some techniques for improving it.

The *SPIN* Unix Server is compared against Digital Unix V3.2. Because of the redirection inherent in server emulations, Digital Unix is expected to perform somewhat better. The *SPIN* Unix Server implementation is not measured against the Mach Unix Server implementation in this paper. The Server has been modified substantially since it was developed for Mach 3.0; the overall design is roughly the same, but many implementation details have changed. Consequently, measuring it as the BSD4.3 Single Server for Mach 3.0 would be invalid‖. When measured, the modified server on Mach was found to perform comparably to the *SPIN* Unix Server—occasionally a bit faster, more often a bit slower. This implies that the poor performance is the result of the

¶On the Alpha, register `v0` holds the system call return value, register `a3` holds the Unix errno value, and register `a4` indicates either parent or child after a `fork()` system call.

‖The evolution of the server is described in Section 6.1. In addition, measurements for the current Unix server running on Mach 3.0 are provided in Appendix B.

server model, not this emulation.

Measurements were collected on DEC Alpha 133MHz AXP 3000/400 workstations, which are rated at 74 SPECint 92. Each machine has 64 MBs of memory, a 512KB unified external cache, an HP C2247-300 1GB disk-drive, and 10Mb/sec Lance Ethernet interface. Measurements are listed in Table 2.

Operation	Digital Unix		*SPIN* + Server	
getpid()	2	μsec	597	μsec
trap and return			13	μsec
read()	24	μsec	643	μsec
ls -l (cold)	246	msec	2043	msec
ls -l (warm)	159	msec	530	msec

Table 2: *Performance measurements for the* SPIN *Unix Server, compared to Digital Unix V3.2.*

The first experiment measures the time for a null system call (`getpid()` is actually used). In Digital Unix, this measures a trap into the kernel and a return to user space. For the *SPIN* Unix Server, two measurements are listed. Trap and return is analogous to Digital Unix `getpid()`. It measures the latency to drop into the kernel with a system call trap and return. The system call trap in *SPIN* has richer semantics than in Digital Unix. For instance, in this experiment, a guard and a handler were invoked through the dispatcher in response to the system call event. The second measurement indicates the latency of `getpid()` executed in the server. It measures the time to trap into the kernel, wake up a server thread, copy the system call state to the server, execute the `getpid()` system call, copy the return state to the kernel, wake up the application thread, and return to user space. There are two major causes of inefficiency. The first is the copy of system call state between the server and the kernel, which involves six separate cross-address space copy operations. The second is the thread handoff between the application thread and the server thread. Currently there is no sophisticated thread handoff implemented, meaning that the receiving thread is placed on the scheduling queue behind any other active threads in the system. Both the state copy and the thread handoff occur twice in each system call to the server.

The second experiment involves a more complicated system call. `read()` measures the time required to transfer a 1KB buffer of data from the file system to an application. Both Digital Unix and the *SPIN* Unix Server use a `copyout` primitive to execute the data transfer. In Digital Unix, `copyout` is only slightly more expensive than `bcopy`. The server, however, must first copy the data to the kernel and then copy it again to the application. The penalty for this indirect data transfer is apparent in the measurements. The *SPIN* Unix Server `read()` measurement, after adjusting for the trap redirection latency, is still somewhat slower than the Digital Unix measurement. The `read()` system call is measured on both systems with a primed cache.

The third experiment measures the performance of a real program. `ls` is run on a directory with 300 files. The `-l` option is used to cause all files to be statted, requiring significantly more work. The experiment is run with both a warm and cold cache. The difference between the warm and cold cache is more striking in the server than in Digital Unix, since the server caches data in user space and must copy it across the user-kernel boundary.

5.1 Improving Performance through Protocol Decomposition

The performance measurements of the *SPIN* Unix Server indicate that in its current state it cannot compete with a monolithic system such as Digital Unix. A primary source of inefficiency in the *SPIN* Unix Server architecture is the indirect data path between applications and machine resources. By removing some of the indirection, the performance of the server can be improved.

Indirection is especially costly in the implementation of the BSD socket interface. When an application sends data via UDP through a socket, control first switches from the application to the kernel to the server. The server then copies the data from the application address space to the server address space. The data is wrapped into an `mbuf` and then passed through the server network protocol stack, where it is packaged into a network packet. At the bottom of the server network protocol stack, the packet is written to a virtual network device driver in the kernel. Control is switched into the kernel, where the virtual driver packages the data into another `mbuf` and passes it to a kernel physical device driver. The physical driver removes the packet from the `mbuf` and sends it across the network. Control is then returned to the server and to the application. The receive path is symmetrical.

The *SPIN* Unix Server provides these layers of abstraction as part of a generalized networking interface. However, applications that focus on a particular subset of the interface, such as UDP send and receive, incur a stiff performance penalty for generality they do not use. The flexibility of *SPIN* allows application-specific optimizations to be integrated with an existing emulation. Hence, developers can identify critical execution paths and streamline them with kernel extensions.

This approach is similar to protocol decomposition work by Maeda [Maeda & Bershad 93], in which critical send and receive services of a protocol migrate into the application once a communication endpoint has been established. However, moving services from the kernel into the application address space provoke some of the same problems encountered with the library-based emulations. Specifically, portions of the protocol stack in the same address space as the application are not protected from the application. An errant application could fail unpredictably, and there is no easy way of regulating the outgoing packets of malicious applications.

SPIN extensions allow the same protocol decomposition without the protection breakdown. To optimize UDP send and receive, an application installs an extension which implements the necessary BSD socket system calls in terms of in-kernel *SPIN* networking services.[**] The "shortcircuit" extension intercepts system calls on the critical path and allows all others to be reflected to the Server. Consequently, the entire protocol stack in the Server is removed from the data path. Figure 4 illustrates the difference between the *SPIN* Unix Server send path and the shortcircuit send path.

The shortcircuit mechanism was evaluated by measuring the latency for a round trip UDP packet sent from and received by a user-level application. The time between a machine running the Server on *SPIN* and another machine running native Digital Unix is about 3.8 milliseconds (this is approximately the same latency measured running the Server on Mach). The time to send the same packet between two machines running Digital Unix is about 620 microseconds. Using the protocol shortcircuit, the round trip latency between the *SPIN* Unix Server machine and the Digital Unix machine dropped from 3.8 milliseconds to about 660 microseconds—nearly the same as measured between two machines running Digital Unix.

[**] In-kernel networking services can refer to native services or other dynamically linked extensions.

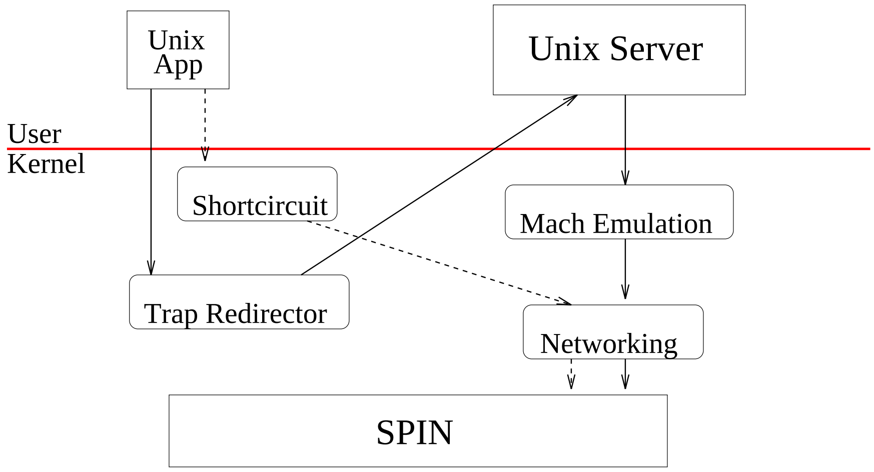

Figure 4: *This path to send a UDP packet through the BSD socket interface without a shortcircuit (solid line) and with a shortcircuit (dashed line).*

6 Related Work

The Unix emulation described in this paper is derived largely from the implementation of Unix on Mach 3.0 [Golub et al. 90]. The Mach Unix emulation centers around the BSD4.3 Single Server, a user-level application which exports Unix semantics. Client applications communicate with the BSD4.3 Single Server via an *emulation library* loaded into the address space of each Unix program. System calls trap into the kernel and are reflected to the task's emulation library. The emulation library delivers a message to the BSD4.3 Single Server, which executes the actual system call and returns the result to the application via its emulation library. The Mach 3.0 and Single Server lacks the protection and flexibility of the *SPIN* Unix Server. The emulation library is loaded directly into an application's address space, which means that an errant application can corrupt emulation data and cause unpredictable behavior. Also, Mach 3.0 offers no mechanism for applications to define a shortcircuit for calls to the server.

The Spring [Khalidi & Nelson 92] Unix emulation increases the functionality of the dynamically linked library. The Spring Unix emulation library replaces the dynamically linked *libc*. Besides providing the traditional services of *libc*, it contains specialized system call stubs. Rather than generate a system call trap, these stubs use Spring IPC to request services from native Spring modules. The small set of services not provided by existing Spring modules are implemented in a small Unix server. By minimizing the reliance on the Unix server, Spring reduces the message-passing and trap reflection overhead. However, the Spring Unix emulation lacks the protection offered by the *SPIN* Unix Server. The Unix emulation library resides in the application's address space where it can be corrupted just as the Mach emulation library. In addition, since all Unix applications must be dynamically linked against the Spring Unix emulation library, Spring does not offer binary compatibility.

In the CHORUS system [Bricker et al. 91], Unix is emulated by a collection of independent servers, each providing a different aspect of the Unix semantics. One central server is responsible for intercepting system call traps and delegating responsibility for executing system calls via CHORUS IPC. To reduce communication overhead, CHORUS servers may be linked directly into the kernel address space. Since CHORUS servers are linked into the kernel address space, they are not vulnerable to corruption by applications. However, they must be trusted by the CHORUS kernel. This implies that applications may not dynamically link customized servers into the kernel, as applications may do in *SPIN*.

6.1 Evolution of the *SPIN* Unix Server

The *SPIN* Unix Server is a derivative of the BSD4.3 Single Server for the Mach 3.0 microkernel [Golub et al. 90]. The Single Server communicated with applications through trap redirection, Mach IPC, and a *task emulation library*. Each Unix application had a library loaded directly into its address space. When applications made system calls, the system call traps were redirected into this emulation library. The system call was handled in the library, if possible. Otherwise, a Mach message was sent to the Single Server. The Single Server executed the system call in its user-level address space. If the Server required kernel services, it made a system call or sent a message requesting the appropriate service. Upon completion of the system call, the Single Server returned the result to the application emulation library. The emulation library then returned directly to the application.

The server arrived at the University of Washington after a five year evolution which included development work at Carnegie Mellon University, Digital Equipment Corporation, the Open Software Foundation, and the Open Software Foundation Research Institute. In this time, many significant changes were committed to the system. These included networking and file system improvements, better support for multi-threading, and character device enhancements. The most significant modification with respect to the *SPIN* Unix Server was the removal of the emulation library. System call traps are now reflected directly into the server task [Patience 93].

Modifications to the server continued at the University of Washington. Changes were geared toward improving stability and facilitating the port to the *SPIN* kernel. All Mach IPC was removed from the server. Kernel calls that were formerly implemented with Mach messages were transformed into system calls. The removal of message-passing both simplified the exception mechanism and reduced the overhead of communication between the server and the kernel. Other Mach-based optimizations, such as memory-mapped system call parameters, were switched off. The resulting Unix server, although based on the BSD4.3 Single Server for Mach 3.0, differs substantially in its implementation and performance. In some cases it is faster, in others slower.

7 Conclusion

The *SPIN* Unix Server emulates Unix in the context of an emulation of a subset of Mach 3.0. The Mach emulation consists of *SPIN* kernel extensions which intercept and execute Mach system calls by calling upon *SPIN* services. The Unix emulation is constructed with an extension which redirects Unix system calls to the user-level *SPIN* Unix Server.

The performance evaluation of the *SPIN* Unix Server indicates that it is not a viable platform for general purpose computing. Even small applications, such as `ls`, execute significantly more slowly than on Digital Unix. There are many opportunities for optimization in the current implementation of the *SPIN* Unix Server. Some of these, including copy-on-write and memory mapping across address spaces, have already been mentioned. In addition, a technique was described in this paper for migrating services from the server into kernel extensions. This method of improving the performance should be generalized and implemented for real applications.

The strategy for implementing the *SPIN* Unix Server could also be refined. When emulating a large system, it is beneficial to identify fundamental services in the system and implement these in advance. For instance, a fundamental service in Mach is the virtual memory system. For the Mach emulation, individual services were implemented as they were demanded by the server. This methodology was eventually successful; however, designing and implementing a Mach virtual memory extension in advance would have simplified development and probably improved performance in the long run. An extension emulating some features of the Mach virtually memory system is now under development. A future experiment will involve converting the *SPIN* Unix Server virtual memory support to this extension and evaluating the improvement in elegance and performance.

References

[Bershad et al. 95] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[Bricker et al. 91] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M. A New Look at Microkernel-based UNIX Operating Systems: Lessons in Performance and Compatibility. In *Proceedings of the EurOpen Spring'91 Conference*, Tromsoe, Norway, May 1991.

[Custer 93] Custer, H. *Inside Windows NT*. Microsoft Press, 1993.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[Khalidi & Nelson 92] Khalidi, Y. A. and Nelson, M. N. An Implementation of Unix on an Object-oriented Operating System. USENIX 1992. Reprinted with permission.

[Loepere 92] Loepere, K., Editor Mach 3 Kernel Interfaces Open Software Foundation and Carnegie Mellon University, 1992.

[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, Asheville, NC, December 1993.

[Nelson 91] Nelson, G, editor. *Systems Programming in Modula-3*. Prentice Hall, 1991.

[Ousterhout et al. 88] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The Sprite Network Operating System. In *IEEE Computer*, February 1988, pp.23–35.

[Pardyak & Bershad 96] Pardyak, P. and Bershad, B. Dynamic Binding for Extensible Systems. To appear in *Proceedings of the Second Symposium on Operating Systems Design and Implementations*, Seattle, WA, October 1996.

[Patience 93] Patience, S. Redirecting Systems Calls in Mach 3.0, An Alternative to the Emulator. In *Proceedings of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.

[Rashid et al. 89] Rashid, R., Baron, R., Forin, R., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R. Mach: A Foundation for Open Systems In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

[Sirer et al. 96] Sirer, E., Fiuczynski, M., Pardyak, P., and Bershad, B. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the First Workshop on Compiler Support for Systems Software*, February 1996.

A Emulated Mach system calls

System Call	Relative Difficulty	Location	Completion
vm_write	=	S	3
vm_read	=	S	3
vm_inherit	+	E	2
vm_region	+	E	2
vm_statistics	=	E	1
vm_map	+	E	1
vm_allocate	+	S	3
vm_deallocate	+	S	3
vm_protect	+	S	3
task_create	+	S	3
task_info	=	E	1
task_terminate	+	S	2
mach_reply_port	-	E	1
mach_task_self	-	E	3
mach_thread_self	-	E	3
mach_host_self	-	E	1
mach_port_move_member	-	E	1
task_set_special_port	-	E	1
mach_port_allocate	-	E	1
mach_port_deallocate	-	E	1
mach_port_insert_right	-	E	1
mach_port_allocate_name	-	E	1
task_get_special_port	-	E	1
task_get_master_port	-	E	1
device_read	+	S	3
device_read_inband	=	S	3
device_read_request	=	S	2
device_read_overwrite	=	S	3
device_read_overwrite_request	=	S	2
device_write	+	S	3
device_write_request	=	S	2
device_write_inband	=	S	3
device_write_request_inband	=	S	2
device_set_status	+	S	1
device_get_status	+	S	1
device_open	=	S	3
device_close	=	S	3
thread_rendezvous	=	S	3
thread_switch	-	S	2
thread_create	=	S	3
thread_set_state	=	S	3
thread_resume	-	S	3
poke_softclock	+	S	3
set_softclock	+	S	3
host_kernel_version	-	E	1
processor_set_default	-	E	1
host_processor_set_priv	-	E	1
host_info	-	E	1
get_mach_time	=	S	3

Table 3: *An annotated list of emulated Mach system calls. The relative level of difficulty is indicated by (+) for difficult, (=) for medium, and (-) for easy. The location of the emulation refers to whether the call was emulated entirely in the Mach emulation extension (E) or whether some* SPIN *services, native or dynamically linked, were required (S). The degree of completion of the emulation indicates whether the call is fully supported. The values range from supported but completely spoofed (1) to fully supported (3).*

B Measurements of Mach and the Unix Server

This section displays the same measurements as in Section 5, along with measurements for Mach 3.0 running the Unix server. As mentioned in Section 5 and described in Section 6.1, both the Mach microkernel and the Unix server have been modified substantially.

Measurements are listed in Table 4. All benchmarks are as described in Section 5. Unfortunately, not all Mach measurements are available for the platform described in Section 5. Hence, the `getpid()` and `read()` benchmarks below were measured on a DEC Alpha 100MHz AXP 3000/300L workstation. This machine has 64 MBs of memory, a 256KB unified external cache, and an HP C2247-300 1GB disk drive. It runs Digital Unix V2.1 rather than Digital Unix V3.2. The `ls` benchmark was run on DEC Alpha 133MHz AXP 3000/400 workstations, as described in Section 5. The trap and return benchmark is not available for Mach.

Operation	Digital Unix	Mach + Server	*SPIN* + Server
getpid()	5 μsec	1383 μsec	1108 μsec
trap and return			23 μsec
read()	44 μsec	1533 μsec	1207 μsec
ls -l (cold)	246 msec	8360 msec	2043 msec
ls -l (warm)	159 msec	7672 msec	530 msec

Table 4: *Performance measurements for the* SPIN *Unix Server, compared to Digital Unix and the Unix server on Mach 3.0.*