SimOS: A Fast Operating System Simulation Environment

Mendel Rosenblum
Mani Varadarajan

Technical Report: CSL-TR-94-631

July 1994

Mendel Rosenblum was supported by an NSF Young Investigator Award. Mani Varadarajan was supported by a grant from the Charles Lee Powell Foundation.

SimOS: A Fast Operating System Simulation Environment

Mendel Rosenblum and Mani Varadarajan

Technical Report: CSL-TR-94-631

July 1994

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-2140
{mendel,mani}@cs.Stanford.EDU

Abstract

This paper describes techniques used in SimOS, a simulation environment developed for building and studying operating system software. SimOS allows an operating system to run at user-level on top of a general-purpose operating system, such as System V R4 Unix. SimOS works by simulating a machine's hardware using services provided by the underlying operating system. In this paper, we describe the techniques used by SimOS to simulate a machine. These techniques encompass using the operating system's process abstraction and signals to simulate a CPU, using file mapping operations to simulate the memory management unit, and using separate processes to simulate the I/O devices. The techniques we present allow the simulator to run with high speed and sufficient detail so that software development and operating system behavior studies can be performed. The speed of the simulation depends on the quantity and the cost of the simulated operations. Real programs can run in the simulated environment at between 50% and 100% of the speed of the underyling machine. The simulation detail we provide allows an operating system running in the simulated environment to be nearly indistinguishable from that of the real machine, as seen from a user perspective.

1.0 Introduction

Significant amounts of time and money are spent on the development, tuning, and maintenance of operating system software. These costs are partly due to the high standards of correctness and performance expected by the users of operating systems. The presence of bugs in the system software can have very serious consequences, including loss of system availability, incorrect results, and corruption of permanent data. Resources consumed by the system software are unavailable for end-user application programs.

Meeting such high expectations for correctness, performance, and robustness is further complicated by some inherent challenges imposed by the nature of the software. The level of functionality expected from a general-purpose operating system necessitates that the system be large and complex, requiring hundreds or even thousands of person-years of work for the development of the software. Code bases reaching into the millions of lines are not uncommon. Furthermore, providing the expected level of performance requires supporting a high amount of concurrency and dealing with interrupts and other asynchronous events. Consequently, it is significantly harder to develop and maintain operating systems than sequential programs.

Operating systems, as opposed to application programs, present an additional difficulty as they run directly on top of the “raw” hardware of the machine. Because of this, the software development, monitoring, and debugging environment is significantly less hospitable than that of application programs. To make matters worse, the availability of the environment is often limited, or in the case of start-of-the-art machines, totally unavailable.

Given these difficulties, it is not surprising that operating systems are relatively poorly understood when compared to application programs. Most user-level tools used to understand the behavior of applications totally ignore the effects of the underlying operating system, even when the underlying OS is a significant source of both correctness and performance problems on the real machines. To address this problem, we created SimOS, a simulated environment that supports the execution of operating systems and applications on top of general-purpose operating systems. Using SimOS, it is possible to study the behavior of operating systems and the applications that run on them with the same ease that one studies application programs today.

One goal for SimOS was to provide a simulation environment significantly faster than previous machine simulators that supported operating systems. There are two reasons for this. First, operating systems execute a large amount of code in a non-repetitive style. This makes it difficult to formulate general statements about the operating system behavior without being able to observe the system’s execution over relatively long periods of simulated time. It is not possible to extract a small inner-most loop that accounts for most of the observed behavior.

Second, as some simulation studies require such a large amount of CPU time, it is not possible to run these studies on all the code executed by a workload. For these simulations it is useful to have a much faster simulator which can be used to position the slower, more detailed simulator for execution. In this way, the fast simulator can be used to position the slow simulation in front of a interesting section of the workload. Or one can repeatedly switch between the fast and slow simulator and use statistical sampling techniques to get better coverage of the workload.

Regardless of the level of detail required by a simulation study, it is important that the base environment support a fast simulation mode.

The approach used in SimOS to achieve high speed is to simulate the hardware of the machine using the hardware and OS services of a general-purpose computing platform, such as an Unix workstation. By using the hardware and services of a general-purpose computer, SimOS can create a simulation environment that is both widely available and fast. This paper focuses on the techniques used to create the “fast mode” of SimOS.

The techniques reported in this paper were developed while building two fully operational operating system simulation systems over a two year period, starting in the spring of 1992. The first system was a port of the Sprite network operating system[18] to run on a Sun SPARCstation under SunOS 4.1. While this system required a rewrite of most of the machine dependent code, the machine independent code remained entirely intact. It first booted multi-user in the fall of 1992, retaining binary compatibility with native Sprite on SPARC platforms. In this paper we refer to this system as the “Sprite port.”

The second system was started in the fall of 1993. This second system is significantly more aggressive in its attempt to minimize changes needed to port the kernel to the simulation environment. Because of requirements from other projects, our initial work for this environment was moved to MIPS R3000 and R4000 based machines from Silicon Graphics running IRIX version 5. The operating system ported was also IRIX version 5, a flavor of System V R4-based Unix. This system first came on-line in early 1994, supporting debugging of all machine independent code as well as most of the R3000 machine dependent code. In this paper we refer to this system as the “IRIX port.”

The remainder of this paper is divided into 8 sections. Section 2.0 discusses some basic needs and approaches for machine simulation environments. Section 3.0 presents the key techniques for simulating hardware using operating system services. Section 4.0 describes some of the interfaces to the simulated environment. Section 5.0 describes our experience with the systems we have built. Section 6.0 describes the limitations of our approach. In the course of building the simulators, we discovered that the environments facilitated a number of uses beyond that of software development. Section 7.0 describes some of these additional uses relevant to operating system developers and computer architects. Section 8.0 discusses related work. We conclude in Section 9.0 with a summary and our thoughts for future directions.

2.0 Previous work in machine simulation environments

Like most simulation environments, an operating system simulator faces a trade-off between the amount of detail simulated and the speed of the simulation. Usually, the less detailed the simulator needs to be, the faster it can be made to run. The criteria for detail and speed depends on what is being studied. A simulator that positions a more detailed simulator or checks if a particular software algorithm produces the correct results could be much faster than one that computes the number of cache misses or processor pipeline stalls of a next-generation dynamically-scheduled superscalar processor with lockup free caches.

The fast mode of SimOS aims for a point in the speed to detail trade-off spectrum that has not been addressed by previous simulation environments. In terms of speed, SimOS tries to support execution speeds fast enough to be used for software development. This requires execution speeds of well less than an order-of-magnitude off the real machine. With this speed SimOS can execute large amounts of code in a reasonable amount time. This is an important feature for software debugging and test cycles. In terms of detail, SimOS should be faithful enough to the real machine that a bug should exist in the simulated environment if and only if it exists in the environment being simulated. When more detailed simulations are necessary, SimOS should be able to position the more detailed simulator in the code and provide it with enough initial state to start execution.

Previously, user-level simulations of OS code have taken one of two approaches. Either the simulation environment simulated the entire machine by running the OS and applications on the machine, or the simulation environment modified the operating systems to run at user-level. The rest of this section describes these two techniques in more detail.

One traditional way of running an operating system at user-level is to build a sufficiently detailed simulation of the hardware that the OS can boot and on which the application can run. This includes simulating the CPU, the memory management unit, and the I/O devices accessed by the kernel. For environments that are building the hardware, these machine simulation systems likely exist as a result of the hardware verification efforts. Unfortunately, most simulations used for hardware verification are more detailed, and hence slower, than needed for OS behavior studies.

Machine simulation systems that focus on execution speed have emerged to support machine studies. Published examples of such systems are the works of Bedichek[2] and Magnusson[15]. These simulations, while they have sufficient detail to run arbitrary workloads and can be used for studying memory system behavior, are generally too slow for software development. These CPU and MMU simulations result in slow-downs of over an order of magnitude, often 20 to 70 times slower than real time. This technique is more useful for studying memory system behavior and helping OS bringup than for software development.

The alternative to hardware simulation is to remove the parts of the operating system not running at user level. Unfortunately, removing the machine dependent code from an operating system essentially leaves nothing to invoke the machine independent code that remains. This is because machine dependent code, such as trap handlers, is the conduit through which all user program requests pass to the machine independent code. Without this machine dependent code, it becomes difficult to support the necessary input/output to drive the OS simulation.

An alternative that has been used successfully is to build an environment in which a portion of the operating system runs in isolation. This is commonly done for modules such as file systems[24]. Software is written to support the desired component in an artificial environment—the software generates simulated inputs to drive the module, and the module’s outputs are observed and possibly fed back into the simulation. This is analogous to the testing of a single chip in a hardware development environment.

The chief problem with this technique is the difficulty of driving the testbed with realistic workloads. Without the full operating system environment, the simulation must either be driven by traces from the real system or by a randomly generated workload. Although this permits examination of the algorithm in great detail, it doesn't allow observation of the behavior of the entire operating system. In particular, because the technique does not model the interactions between different OS subsystems and application programs, it lacks the detail necessary for many studies.

3.0 Simulating hardware using OS services

An operating system can be viewed as a program with inputs and outputs (see Figure 3.1). In order to run an operating system in a simulation environment, the environment must provide inputs and outputs essentially the same or similar to those of the real machine. Our approach to fast OS simulation is to simulate the machine's hardware using the services provided by a general-purpose operating system. Although the interface the OS expects from the hardware is different from that of an operating system, modern operating systems provide a level of functionality that can simulate the hardware with acceptable speed. For example, typical OS interfaces do not directly export the trap architecture of the processor to a user process. However, they do provide a way for user processes to get control of certain traps and exceptions. Similarly, while modern operating systems do not provide access to the memory management unit of the hardware, they do provide system calls for establishing and de-establishing mappings to a file's blocks.

One of the challenges of simulating hardware using the services of an operating system is that most hardware-supported operations execute in a few cycles, whereas even the fastest system calls of an operating system take hundreds or thousands of cycles. This requires efficient solutions for performance-critical elements such as the CPU and the MMU. In the remainder of this section, we first describe the implementation of the CPU and MMU, then describe the disk, network, and terminal simulation. Although the techniques presented here will work on other systems, in this discussion we assume an advanced Unix operating system such as System V R4.

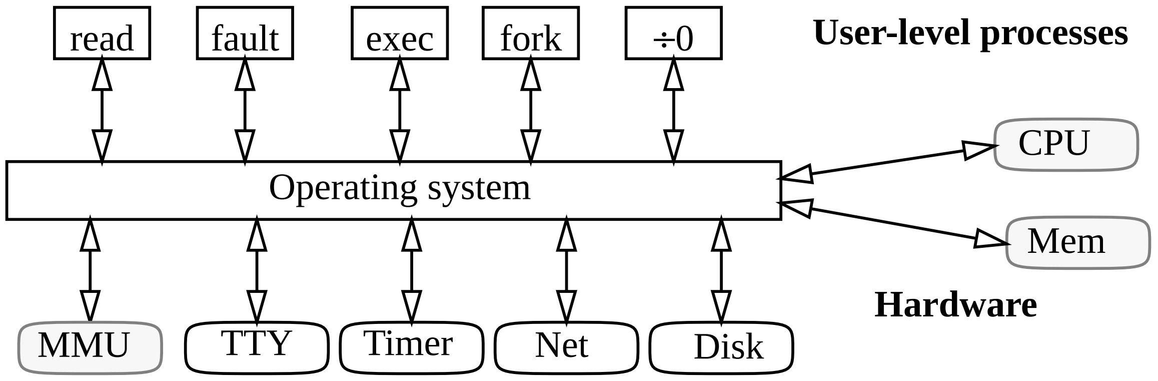

FIGURE 3.1. Input/output interactions of an operating system.
The sources of inputs for an operating system include trap and exceptions that come from user mode (such as system calls, page faults, math errors, etc.) and interrupts, and DMA from devices. The outputs of the operating system control the MMU, the contents of user memory, and I/O devices. Of all the interactions, the CPU, MMU, and memory are the most performance critical for a fast simulation.

3.1 CPU simulation

The obvious operating system service to use for simulating a CPU is the process abstraction. For example, the Unix process abstraction consists of a single thread of control running in a virtual address space. When simulating the operating system, this process abstraction looks enough like a CPU running under its control that most of the simulation runs unmodified in a process. Events that do not normally happen on the raw hardware, such as scheduler preemptions and page faults, are transparent to the running process. As a result, a simulated OS is not aware that these events are happening.

By using the process abstraction for a CPU, it is simple to simulate a multiprocessor using a collection of processes, one per simulated CPU. The number of simulated CPUs is only limited by the number of processes supported by the operating systems. In practice, unless the underlying system has enough real processors available for these processes, some unrealistic execution interleaving between the CPUs will be generated. Section 6.0 describes this problem in more detail.

Although the machine independent code of an OS runs without any problem in a Unix process, a portion of the OS, particularly in the low-level machine dependent parts of the kernel, uses the CPU in a way that is not supported at user-level. This code assumes that all exceptions will use the trap architecture defined for the CPU and that it can access the privileged instructions and registers of the CPU. Neither of these are typically available at user-level.

3.2 CPU trap architecture simulation

The trap architecture of a CPU allows an operating system to get control on any exception or interrupt. In modern operating systems, the CPU hardware, firmware, and the OS trap handlers in combination save the state of execution and call into the machine independent code for processing. To the machine independent code, the trap architecture can be thought of as a source of inputs into the kernel. These inputs include the type of exception that occurred and the state of the processor at the exception.

For a user-level process, the exception notification mechanism provided by the operating system is the closest analogue to a trap architecture. Signals, which are the Unix exception notification mechanism, allow a process to get control of such exception events as illegal instructions, bad memory references, and external notifications (such as interrupts.) Unix signals save the processor state and jump to the user specified routine for that signal. Most of the events that cause traps on the real hardware can be mapped directly to Unix signals.

An OS’s trap handlers can be simulated at user-level by catching signals and feeding the information into the kernel being simulated. For example, in most RISC architectures, attempting to load a 32-bit word from an address that is not word-aligned will generate an unaligned address exception. The Unix operating system converts this exception into a SIGBUS signal and sends to the hardware simulation. The hardware simulation converts the SIGBUS signal into an unaligned address exception for the simulated OS trap handlers. The simulated OS is likely to generate an exception condition for the simulated user process.

Once a signal is caught, operations can be performed and execution resumes. For example, a reference to an unavailable page generates a SIGSEGV signal. This passes a page fault notification to the page fault handling routine of the simulated kernel. When the page fault returns, execution of the faulting process can be restarted from the point of the faulting reference.

3.3 Privileged instructions

Privileged instructions and registers are other features that are used by operating systems but are not available at the user-level. Most CPUs support privileged operations such as disabling interrupts or changing the MMU. Neither of these operations can be done at user-level. The operating system simulation can deal with these instructions in either one of two ways. It can replace the privileged code with non-privileged instructions that emulate the effects the instruction would have had. Or, it can detect the execution of the privileged instruction and emulate the instruction in-place.

3.3.1 Removing privileged instructions

Most portable operating systems have well defined interfaces between the machine independent code and the machine dependent code, which contains all the privileged instructions and the access to privileged registers. One technique for handling the privileged instructions involves rewriting all the machine dependent parts of the operating system to use the simulated CPU state. Logically, this can be viewed as porting the operating system to a new platform, which is the simulated platform. The Sprite simulator uses this technique.

This porting approach is relatively easy and results in a faster simulation. Furthermore, most of the code in the machine independent part of the OS continues to run at full speed. The chief drawbacks of this approach are as follows:

- There can be no debugging of the machine dependent code of the real machine.
- It adds another platform that must be maintained by the OS development team.

3.3.2 Emulation of privileged instructions

As another option, privileged instructions can be emulated in-place. This can be accomplished by detecting the instructions and executing the emulation code instead. The Unix operating system uses a SIGILL signal to notify a user-level process that it is trying to execute a privileged instruction. The OS simulator can catch this SIGILL signal and decode the trapping instruction. It can then emulate the instruction by changing the simulated machine state and restarting the program execution at the instruction following the trapped privileged instruction. This technique is used in the IRIX port.

This in-place emulation technique enables the operating system to be executed on the simulated CPU without changing most of the machine dependent code, such as the trap handlers and privileged register accesses. For example, consider the simulation of the IRIX trap handlers. These trap handler routines perform operations such as reading the trap state from privileged registers and re-enabling interrupts. When the trap architecture simulation detects an exception, it saves the trap state in memory and jumps to the correct trap handler of the simulated OS. The trap handler issues the privileged instructions to collect the trap state and change the interrupt level. These instructions generate SIGILL signals. The signals are caught and the instructions simulated using the saved trap state.

On many architectures, operating system trap handlers contain a high density of privileged instructions. In addition, there is the relatively high cost of catching a SIGILL signal (in the hundreds of microseconds.) Together, these suggest it is better to simulate multiple instructions per SIGILL signal. In the time it takes to catch just one SIGILL signal, a fairly simple CPU simulator can simulate hundreds of instructions.

Because the CPU simulator must handle all the privileged instructions, it is an easy addition to have it simulate the non-privileged instructions that are common to trap handlers. Most expensive traps can be totally avoided by switching to the CPU simulator when an exception is taken. The simulator then emulates both the privileged and non-privileged instructions until the CPU reaches the OS's machine independent code. We observed a 30% to 50% improvement in execution time, on average, with this technique. In addition, the CPU simulator also allows for the modeling of pipeline hazards that are commonly exposed in the use of privileged instructions. Section 7.0 discusses some additional uses of this idea.

The chief drawback of this technique is speed. A SIGILL signal is very slow compared to using direct procedure calls, as in the Sprite port. Emulating multiple instructions on one SIGILL cannot alleviate the slowness. This is particularly true for operations such as system calls, where the trap handler execution can be a substantial part of the total execution time. Section 5.0 contains an additional discussion of this effect.

A final option for emulating privileged instructions is to replace privileged instructions when encountered with an equivalent sequence of non-privileged instructions that effect the same changes to the simulator's state. This dynamic replacement has the best properties of both of the above schemes. As it is performed dynamically, the code containing privileged instructions can be executed without being "ported" to the SimOS environment. The replacement code also rids the simulator of the frequent and expensive SIGILL signals. Unfortunately, this type of replacement is dependent on the hardware architecture. It can be difficult to accomplish in an arbitrary piece of code without performing some analysis of the control flow of the code. This optimization is particularly difficult to use with architectures that utilize delayed branches, such as SPARC and MIPS.

3.4 MMU simulation

The memory management unit (MMU) is used heavily by the operating system for relocation and virtual memory support. Unlike the CPU, the normal Unix process abstraction does not have an obvious analog to the MMU. The basic problem is that the OS assumes that it can control the virtual to physical address mappings for itself and the private address spaces for user processes. The OS incorrectly assumes it is in total control of the physical memory of the machine and that it can establish arbitrary mappings of virtual addresses to physical memory pages.

3.4.1 Kernel virtual address space

The user-level mechanism used to map and unmap files into a process's address space is the closest approximation to an MMU's ability to establish virtual memory mapping. To simulate physical memory and an MMU, we represent memory as a file that is mapped into a process's address space using the Unix *mmap* system call. Establishing a mapping in the simulated MMU is done by mapping a particular page-sized region of the file into the specified virtual memory location. The offset into the file is specified by the physical address. The specified virtual address is used as the actual address to perform the mapping. Figure 3.2 shows how this might look.

The Unix *munmap* system call simulates removal of an address mapping. The *munmap* system call de-establishes a mapping created with a previous *mmap* call. An *munmap* of the entire user address space simulates an MMU flush-mapping operation. Protection bits of the MMU can be simulated by setting the corresponding protection bits of the *mmap* system call. For example, using *mmap* without the PROT_WRITE option will cause any write access to the address space to generate a SIGSEGV. The trap architecture simulation converts the SIGSEGV into input for the kernel.

A shared memory multiprocessor can be simulated by sharing the same physical memory file among multiple CPU/MMU simulations. When a block of the file is mapped into two different CPU simulations, it acts exactly like a memory board in a multiprocessor. Changes made by one processor are immediately visible to the other processors. It is also possible to use different memory files per CPU to simulate a multi-computer without shared memory support.

The MMU simulator for the Sprite port is a result of rewriting the machine dependent portions of the virtual memory system to use the *mmap* and *munmap* calls. Sprite supports an interface to the virtual memory hardware of a machine, similar to the Mach pmap[20] or SunOS hat[6] calls. It was possible to run all of the Sprite VM module's machine independent code in the simulation by using a module that supported this interface, but which used system calls rather than privileged instructions to access an MMU. As each MMU had its own memory file, different Sprite kernels

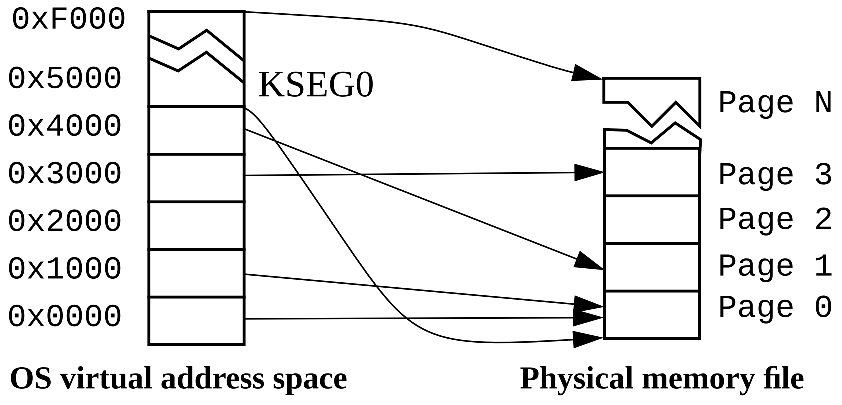

FIGURE 3.2. Kernel address space simulation with mapped file blocks.
This figure shows how an MMU can be simulated by mapping the blocks of a file into the virtual memory of a process. Mapping a physical memory page into a virtual address space is simulated by mapping a page of the physical memory file into the page of the virtual address space. The direct physical memory access of the MIPS KSEG0 segment can be simulated by mapping the entire file into memory.

could not use shared memory for communication. This was natural for Sprite since it was developed as a network operating system that used RPC for all inter-kernel communication.

In the IRIX port, the MMU changes were detected after the privileged instructions which write values into the software had reloaded TLB, and the CPU simulator encountered the reloaded TLB. A TLB simulation was maintained to implement the MIPS TLBWR or TLBWI instructions. Changes written into a TLB entry were translated into *mmap* calls. These calls implemented the mapping specified in the entry. If the overwritten TLB entry had been valid, an *munmap* was used to simulate the replacement of the entry. Flushing the TLB means unmapping all the currently mapped entries. The simulated TLB is also accessed by the trap architecture simulator to invoke the correct TLB miss trap handler (user or kernel) when a SIGSEGV is received.

The MIPS R3000 and R4000 give the CPU direct access to the physical memory of the machine by using part of the kernel's virtual address space. This can be efficiently implemented by mapping the entire physical memory file at the virtual addresses at which physical memory is to reside. Again, Figure 3.2 shows how this might look. Since IRIX has had a relatively long history of running on shared memory multiprocessors, a single memory file was used so that multiple CPU/MMU simulations were all part of the same shared memory multiprocessor.

3.4.2 MMU private address space simulation

Besides providing the ability to map pages into the kernel address space, MMUs on modern machines also provide an abstraction that allows the operating system to implement private virtual address spaces for user processes. Typically, the MMU supports some number of MMU "contexts" or "PIDs" that maintain independent mappings of virtual to physical addresses. These contexts allow the operating system to quickly switch between processes with private address spaces.

At user-level, there are two ways to simulate MMU contexts—reusing the addresses of a single Unix process or using multiple Unix processes. When using a single Unix process, an MMU context switch simulation requires unmapping all the mappings of the currently simulated context and establishing all the mappings of the new context. In the IRIX port, for example, an instruction that changes the MMU PID field in the TLBHI register causes the simulator to unmap all TLB entries that match the original PID and map all the entries that match the new PID. Entries with the global bit set, meaning the PID match requirement should be ignored, are left unchanged.

Mapping and unmapping at every context switch leads to a simple implementation but one that can suffer large overheads. This is particularly noticeable if the simulated machine has large amounts of mapping hardware, such as a large TLB or many mapping registers. In such systems the time to simulate the MMU context switch can be many thousands of times slower than the real hardware.

To speed the simulation of context switching, the simulator can use multiple Unix processes to implement private address spaces. Each MMU context can be mapped to a different Unix process. An MMU context switch then becomes a switch from the execution of one Unix process to another process. The cost of an MMU context switch becomes that of the Unix process switch, the speed of which is hundreds of times faster than changing the mapping for a process with several *mmap* and *munmap* system calls (see Table 1).

This latter technique has a particularly attractive property for the simulation. In essence, we use system calls to put the hardware MMU of the machine on which we are running into a state where the simulated operating system is directly talking to the MMU. Effectively, we are using the real MMU to simulate itself. This alleviates the need for expensive and slow software simulation of the MMU.

The chief disadvantage of using one process per context is the potential of having a large number of processes for machines with large numbers of MMU contexts. The Unix process abstraction is sufficiently heavyweight that having more that a few hundred processes can cause problems. The number of processes becomes a problem for architectures with thousands of MMU contexts, like the SPARC.

We have found that a combination of the above techniques has worked well. The approach we use is to have multiple processes for implementing MMU contexts, all the while sharing these processes among the simulation MMU contexts by using the remapping technique. In this way a small number of processes can be used to hold the “active” contexts of the simulated OS. We get the benefit of a fast context switch without an inordinately large number of processes.

3.4.3 Control of address spaces

The CPU simulator and the MMU simulator must work together to ensure that the hardware is correctly modeled. For example, the trap architecture of a machine interacts with the MMU in several ways with certain assumptions. Primarily, the trap architecture assumes that the CPU gets control no matter what address space the process is running in. On some architectures, there is even an implicit MMU context switch when the CPU takes a trap. Finally, the trap architecture assumes that it can control and modify the trap state and memory of all the MMU contexts.

As mentioned in Section 3.2, Unix uses signals to get control of a process. There are two possible mechanisms for controlling an MMU context—the process controls itself or a different process is used for control. To control itself, a process establishes signal handlers for all signals used by the simulator. When an exception occurs, the state at the exception is saved and the signal handler is called. This signal handler can then use the saved state as input for the simulated OS trap handlers.

The debugging interface is the mechanism in Unix that allows one process to control another. The debugging interface of an operating system normally allows a process to start and stop execution of another process, to get control on any exception, and to access the memory of another process—including receiving notification of any signals and reading the state of the process. By using this interface, the simulator can control the multiple Unix processes that simulate MMU contexts. As part of controlling multiple processes, the simulator can read from and write to their memory, registers, and trap state.

The Sprite port uses the debugging interface for MMU context control. Both the trap handlers and the routines which access user memory were rewritten to make calls to the SunOS debugging routine *ptrace*. One of the chief advantages of this approach is that a user process can not harm the simulated OS, even though the OS can control the user processes running in the MMU context. Each simulated user process runs in a protected environment. Within this environment, even incorrectly behaving applications can be made to behave the same as on the actual hardware. In fact, it is possible to use the simulator to run programs that are suspected of being malicious; any file changes made by such dangerous programs will only affect files in the simulated environment.

For the IRIX port we used the signal mechanism because it more closely approximates what happens on the real machine. Speed was also a concern. On the SGI Indy machines, the signal interface is 15 times faster than the debugging interface. The chief disadvantage of the signal mechanism is that part of the simulated MMU address space must be used for signal handlers and other hardware simulation code and data (see Figure 3.4.) These addresses are unavailable for the simulated MMU and are vulnerable to corruption by the simulated user processes. A user process can use normal load and store instructions to access or overwrite the hardware simulator.

Another disadvantage is that signals are unable to catch system call traps. To enable the simulator to maintain control of programs, we had to change all system call instructions to an instruction that trapped. This was a fairly minor change with IRIX, since system calls are issued from dynamically linked libraries. Only the C library and the few programs that didn't use dynamically linked libraries needed to be changed.

3.4.4 Address conflicts

The MMU simulation has a serious problem with address conflicts. It assumes that the virtual address space that is available to the Unix process is the same size as the space available when running on the raw hardware. This is not the case with many systems, including the SPARC and MIPS platforms used for this work. On the Sun SPARCserver 670 platform running SunOS 4.1.2, the MMU supports a 32-bit virtual address space. However, the operating system kernel uses the top 256 megabytes, which means this space is unavailable for user processes. The problem is even worse on the MIPS machines. The MIPS R3000 (and the R4000 in 32-bit mode) supports a 32-bit virtual address space, but the top half—2 gigabytes—is not available to user processes.

In both the MIPS and SPARC systems, the kernel is normally loaded at addresses that are unavailable to normal user processes. In order to run in a user process, the kernel had to be relocated into user accessible addresses. Relinking the kernel with a new address accomplished most of this relocation. Unfortunately, because the OS kernel knows about its new location, it was also necessary to change some “#defines” which specified the kernel's location and to recompile the source code.

There are two different places to which the kernel can be moved. In the Sprite port, the kernel was moved into its own Unix process. This required the MMU simulator to use the debugger interface to control the MMU contexts. Figure 3.3 shows the process structure of the Sprite port. Effectively, we are treating the Sprite kernel as if it is a normal user program. The hardware simulation module, based on directions from the kernel, simulates inputs from the trap handlers and controls the MMU context. By running the Sprite kernel in a Unix process, the kernel appears enough like a normal user program that tools such as debuggers and profilers run without change.

This approach has the disadvantage of using a different structure from that used in the real machine. For this reason the IRIX port was unable to use this approach. The SparcStation port of Sprite maps the kernel into the high portion of the address space. The user's address space becomes part of the kernel virtual address space. Sprite can use regular load and store instructions to access memory in the address space of the active process. However, routines that accessed user memory directly had to be modified. Fortunately, much of this work had to be done for the port of Sprite on top of Mach[13] and was reusable for this port.

As the IRIX port assumed direct access to user virtual addresses, it simply moved the kernel down into user accessible addresses. Figure 3.4 shows the process and memory layout of this approach. As on the real machine, the IRIX kernel is mapped into every MMU context. The hardware simulator is also mapped into all MMU contexts. A token passing scheme is used to implement the passing of control from one context to another on an MMU context switch.

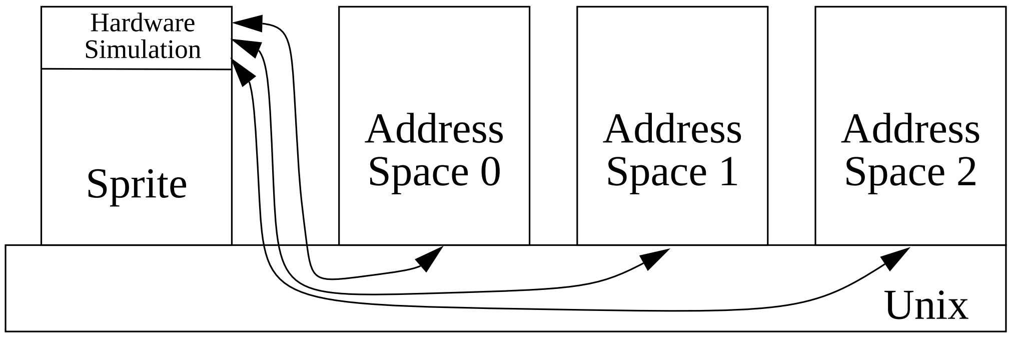

FIGURE 3.3. MMU simulation using the debugger interface.
This figure shows the Unix process structure of the Sprite port of the simulation environment. In this system the Sprite kernel resides in a different Unix process than the simulated MMU contexts. The hardware simulation package uses the Unix debugger interface to control and access the memory of the MMU context, as directed by the kernel.

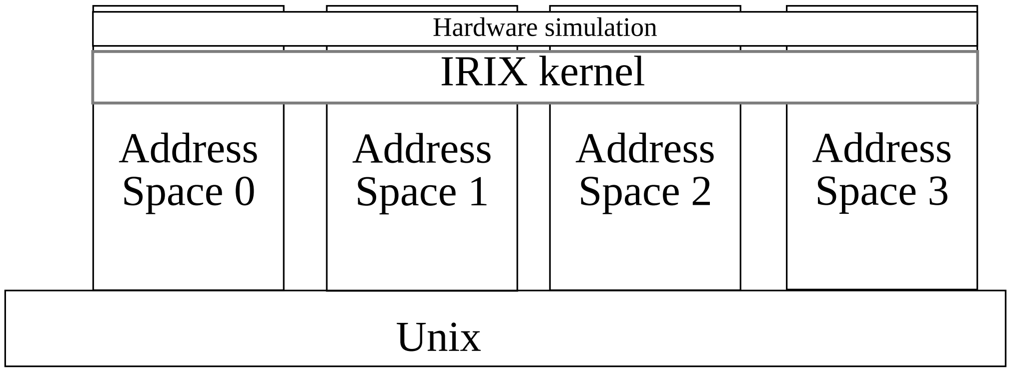

FIGURE 3.4. MMU simulation using signals to control address spaces.
This figure shows the Unix process structure of the IRIX port of the simulator. The hardware simulation package sits at the top of the Unix process' address space of all MMU context. Below the hardware simulator is the IRIX kernel and the user portion of the address space. The hardware simulation code contains the signal handlers that invoke IRIX's trap handlers.

The technique used in the IRIX port has its own set of advantages and disadvantages. When the kernel is run in the same address space, it requires a minimum of modifications because the environment is so similar to the real machine. Machine dependent code that accesses user addresses can run without modification. However, this benefit comes with a price. First, there is the loss of space for the user process and the kernel. Putting the kernel, hardware simulator, and user process all in the same address space required squeezing the user process to operate in less space than on the real machine. Fortunately, most Unix programs are insensitive to the topmost address of their address space. In fact, different platforms within binary compatible lines frequently have different top addresses. The IRIX port further complicated the move because the kernel occupies two gigabytes of virtual address space on the real machine; consequently, it was also necessary to move and compress the kernel. Because the two gigabytes were used in a sparse manner, it only required very minor changes to compress the kernel to less than 128 megabytes.

Besides stealing address locations from the user, this technique has a second disadvantage of providing no protection of the OS text and data from the user program. Although a correctly executing program will not access the addresses where the kernel is located, incorrect programs can unpredictably corrupt the kernel.

3.5 Input/output device simulation

In order to have a complete kernel simulation, it is necessary to provide communication with simulated input/output devices. There are three possible ways that input/output devices interface to an operating system:

- The operating system, using programmed input/output (PIO) techniques, uses memory load and store instructions to transmit and receive commands and data from devices.
- The devices use direct memory access (DMA) to read and write the memory of the command
- The devices use interrupts to notify the CPU of interesting conditions.

We briefly describe how each of these techniques can be implemented.

PIO is normally done by using load and store instructions to memory addresses marked as uncacheable. It is possible to implement these load and store requests by setting the memory address to be invalid. Then, a signal catches the operations and simulates the correct information flow with the I/O device simulator. With this approach we could have maintained binary compatibility with existing device drivers. However, we chose to ignore binary compatibility and installed new device drivers in the kernel. We also replaced all the PIO operations with subroutine calls into the device simulator. This second approach led to a faster and simpler operation, and we used it for the Sprite and the IRIX ports.

DMA is implemented by giving the simulated device access to the physical memory file of the simulator. The device can implement DMA read and write operations by reading and writing to this file. For architectures that support DMA to virtual addresses, the simulated MMU can be used to read and write the data via its mappings.

Finally, interrupts are implemented by having the device send a signal to the process that contains the running hardware simulator. The hardware simulator catches this signal and converts it to an interrupt, which is then input to the trap handlers of the OS. To disable interrupts, the hardware simulator uses the *sigblock* system call, as it blocks the reception of this interrupt signal.

In the case of the IRIX port, where the simulation can be running in one of multiple Unix processes, it is useful to vector all interrupts though an interrupt dispatcher process. This process can synchronize with the MMU context switches to ensure that the correct Unix process is given the signal. Furthermore, a simple boolean flag kept in shared memory between the CPU simulator and the interrupt dispatcher can be used to disable interrupts. This eliminates using the *sigblock* system call.

3.5.1 Device simulation examples

The basic model for simulating devices is to use a separate Unix process to implement the device simulator. The simulated kernel calls stub routines that communicate with the simulated device using shared memory and signals. Simulated devices have access to the physical memory file to simulate DMA, and they can use signals to simulate CPU interrupts. Currently, we have simulators for four devices: the console, disk, ethernet interface, and timer chip. We briefly describe the implementation of these devices below.

- ***Console.*** The operating system console is simulated by doing read and write requests to the standard input and output devices of the process from which the simulator was run. This means that the user of the simulator can type commands at the console and receive the simulator output at the console. Like the real hardware, the console simulator uses an interrupt driven approach. That is, the user, as characters are typed, generates an interrupt per keystroke and the simulator gets an interrupt after each character is typed.
- ***Disk***. A disk is implemented by using a large file to represent the contents of the disk. The file is accessed using *lseek*, *read*, and *write* system calls. The disk simulator understands enough of the SCSI command set[1] to interpret the commands sent by the operating system SCSI device drivers. The disk simulator interfaces with the CPU using DMA to transfer data and interrupts for completion notification. Because the simulator uses the same file system format as the real disks on the system, it is possible to copy a disk into a file and mount it under the simulated OS. This provides a convenient facility for moving large quantities of data between the real and simulated environment.
- ***Ethernet.*** The ethernet simulator provides a simulated LAN that allows the simulated machines to communicate with each other and the outside world. The network interfaces of the simulated machine communicate with the ethernet simulator by sending ethernet packets encapsulated in UDP/IP messages. Using UDP/IP allows physically distributed machines to each run simulated machines that appear on the same simulated ethernet. Communication with the outside world uses the ethernet simulator process as a gateway onto the local ethernet, which in turn uses the SunOS NIT (network interface tap) device. With network connectivity, simulated environment users can remote login to the simulated machines and transfer files using services such as *ftp* or a *NFS*. For ease of use, an internet subnet was established and a set of host names were entered into the local name server.
- ***Timer.*** Both IRIX and Sprite assume they have a period interrupt timer that interrupts at a specified rate. The timer can be simulated by using the Unix *setitimer* system call, which allows a process to be sent a SIGALRM signal at regular intervals. This SIGALRM can be mapped into a timer interrupt for the system.
- ***Inter-processor interrupt.*** Symmetric multiprocessing kernels such as IRIX need the ability for one processor to interrupt another. This is implemented in SimOS by having one CPU simulator do a *kill()* system call to send a signal that interrupts another CPU simulator.
- ***PROM.*** In addition to the above devices, the simulated machine provides a simulated boot PROM that the OS can query to determine information normally provided by the boot PROM. This includes the amount of simulated memory, the simulated ethernet address, and a debugging interface.

4.0 Interfaces to the simulation environment

In addition to the standard ways of accessing a machine kernel, such as via the console or a network connection, SimOS supports other interfaces of interest to developers and researchers. Most of these interfaces involve making the state of the simulated machine available to other programs or routines. In this section we describe the interfaces used for software debugging, inserting more detailed machine simulators, and checkpointing the execution state.

4.1 Debugger interface

SimOS supports an interface that allows the entire simulated machine state to be accessed over a network connection. This interface, when combined with a slightly modified gdb [23] debugger, allows full source-level debugging of both the OS and application running in the simulated environment. This includes the ability to set breakpoints and single step anywhere in the kernel or application code (a feature not possible on most hardware platforms.) The PROM monitor also supports the debugger interface. This ensures that problems detected by the kernel software (i.e., kernel panics) drop into the debugger interface, enabling the investigation of the cause of the panic.

4.2 Detailed simulator interface

The detailed simulator interface is a second interface in SimOS. It provides the ability to encapsulate the entire state of the simulated machine execution. By doing this, SimOs can be used to initialize a more detailed simulator. For the fast mode of SimOS, this provides the ability to construct data structures that contain the values of all the machine's registers and memory. These data structures are passed as arguments to detailed simulation systems. With this mechanism, SimOS can be used to position the execution of these more detail simulators.

The detailed simulator interface also allows the detailed simulator to initialize the state of SimOS fast mode. It does this by filling in the same data structures that are used to pass information into the simulator. This permits the support of a sampling style of execution, where execution switches between the SimOS fast mode and a detailed simulator. The simulator runs after SimOS fast mode and modifies the encapsulated state. SimOS then picks up where the detailed simulator leaves off.

Switches between SimOS and the detailed simulators are triggered either by an event external to the simulator or a special memory reference internal to the simulation environment. By using the signal mechanism, a program can cause the simulator to switch between the detailed simulator and fast mode. Within a program executing in the simulated environment, the simulator can trap a special memory reference and then issue an interrupt. The interrupt serves as a command that controls the simulator's execution. One available command is to switch from fast mode or back to fast mode.

Using this detailed simulator interface we have written several CPU simulators with varying degrees of detail and speed. We have built interfaces between these CPU simulators and several memory system simulators, including a cycle accurate simulator of the Stanford FLASH multiprocessors[14][10].

4.3 Checkpoint/restore interface

The ability to encapsulate and de-encapsulate the execution of the fast mode simulator makes it fairly easy to checkpoint the machine state. This allows the simulation to be restarted without a large amount of simulator execution time. This is a very useful feature when running repeated experiments with different parameters, such as cache size or memory system latencies. The alternative—rebooting the kernel and restarting the workload for each experiment—not only is tiresome. It also causes difficulties in having exactly the same initial state for each run without using the checkpoint mechanism.

In addition to writing the encapsulated register and memory state to a file, the checkpoint mechanism must be able to encapsulate the current state of all I/O devices so they can be restored correctly. The only complexity for this mechanism is the disk I/O device. Because of the large size (over 100 megabytes) of a complete execution environment for a modern operating system, it is not desirable to save the entire contents of a disk in a checkpoint. Instead, the checkpoint mechanism records only the blocks that have changed since boot. It can reapply these changes to the disk file to restore the execution state to that of the checkpoint. This mechanism is also used for normal fast mode operations since all the users of the simulation environment can share the same system disk in a copy-on-write mode. Individual simulators only get private copies of the blocks they modify. It is these private blocks, along with the name of the system disk, that are saved in a checkpoint.

5.0 Experience and performance

Altogether, we have done four ports to the SimoS simulation environment: Sprite, IRIX5, and two different versions of OSF/1, one monolithic and one microkernel. Only the Sprite and IRIX5 ports have gone beyond a demonstration of the techniques. Our observations are mainly based on these two systems. Nevertheless, we have a high degree of confidence that our techniques would apply to practically any modern operating system that runs a standard operating system such as Unix SVR4.

The Sprite port was accomplished by modifying the SPARCstation2 port of Sprite. Its trap handlers, machine dependent virtual memory module, and device drivers were modified to run on top of SunOS version 4.1. The Sprite port uses the SunOS debugging interface (ptrace) to simulate the MMU of the SPARCstation2. Except for memory mapped devices like the frame buffer, the port was binary compatible with the SPARCstation2 port of Sprite. In fact, the root disk we boot from was copied from the main Sprite server at Berkeley. Because of the user-level compatibility, the porting effort required was significantly less than porting to a new machine. The effort was similar to porting to different machines within the same family, such as from an R3000 to an R4000 machine.

The port is complete in terms of functionality. The simulated ethernet allows clusters of simulated Sprite machines to migrate processes and to run the Sprite parallel make facility. The overall feel of the system can best be described as "sluggish" when compared to Sprite running on the real hardware. For real applications, it is usually 10% to a factor of 2 times slower. Nevertheless, in terms of absolute performance, the simulation is faster than the Unix machines that were available until just a few years ago. Table 1 quantifies this performance difference.

The port of IRIX version 5 used the technique described in Figure 3.4 to minimize the number of porting changes. Not including changes to device drivers and makefiles, less than 100 lines of the kernel needed to be changed. Like the Sprite port, the IRIX port was fully functional, including network facilities such as remote login and the network file system. The performance of this port can be best described as “slow.” For real applications, slowdowns range from 20% to over a factor of 50. Table 1 details these numbers.

	Sprite port			**IRIX port**		
Benchmark	**Sim**	**Real**	**Slowdown**	**Sim**	**Real**	**Slowdown**
008.espresso	130 sec	118 sec	1.1x	224 sec	52 sec	4.3x
022.li	1441 sec	507 sec	2.8x	173 sec	150 sec	1.2x
023.eqntott	Not Available			922 sec	22 sec	42.0x
052.alvinn	337 sec	244 sec	1.4x	7710 sec	151 sec	51.0x
ear	40 sec	39 sec	1.0x	19 sec	8 sec	2.4x
mabcompile	253 sec	129 sec	2.0x	1301 sec	36 sec	36.1x
bison	11 sec	7 sec	1.6x	16 sec	3 sec	5.3x
getpid	1900 µs	12 µs	158.3x	3200 µs	8 µs	400.0x
cswitch	6600 µs	600 µs	11.0x	24500 µs	150 µs	163.3x
fork	76 ms	6 ms	12.7x	313 ms	4 ms	78.2x

Table 1 Performance comparison: Simulation versus real hardware

This table compares the performance of the Sprite and IRIX simulation systems to the real machines they are simulating. The benchmarks above the double line are intended to represent real application program performance. The benchmarks with name starting with numbers are taken from the SPEC92 benchmarks[5]. Ear is the short form of the ear benchmark from the SPEC92 suite. Mabcompile is the compile phase of the modified Andrew benchmark described in [19]. It is a particularly OS intensive application. Bison is the execution of the bison parser generation program. The benchmarks below the double line are microbenchmarks that time a particular feature of the operating system. Getpid times a simple *getpid* system call. Cswitch times a round trip context switch between two processes. Fork times process creation and destruction. A full description of these benchmarks can also be found in [19]. The machine used for the Sprite tests was a 40MHz SPARC Sun SparcStation2 (~20 SPECint92). The IRIX test used a 100MHz R4000SC SGI Indy (~60 SPECint92). Because of the different generation CPUs and the different operating systems, it is not possible to compare the performance of the simulation techniques.

The slowdowns in the simulation come from places where a frequent hardware operation must be simulated by making system calls or catching signals. For the Sprite port this slowdown occurs when we switch between user and kernel mode and when we change the MMU mappings. These operations cause the simulator to make *mmap* system calls and to use the debugging interface to control an MMU context. Fortunately, as these operations are relatively infrequent for most workloads, the slowdown is modest. The true magnitude of the slowdown can be seen in the microbenchmarks. The *getpid* benchmark, which switches to the kernel and back to the user in a tight loop, runs over 150 times slower. Operations that have a larger kernel execution component, such as context switching and process creation, have slowdowns in the range of 11 to 12 times slower.

The slowdown of the IRIX port is significantly greater than that of the Sprite port. This is partially due to the relative immaturity of the IRIX5 system. The rest of the slowdown is due to the more detailed simulation. For example, a simple *getpid* system call results in several signals for the simple call itself and for the emulation of privileged instructions. In addition, on entry into the kernel from user mode, the IRIX trap handlers establish mappings in the TLB for a kernel stack and for per process data structures. As this results in several *mmap* system calls per system call, the *getpid* system call is 400 times slower than on the real hardware.

Programs with poor TLB locality also suffer slowdowns. This is a direct consequence of the simulation of the software-reloaded TLB of the MIPS processor. A TLB miss which takes tens of cycles on the real hardware translates into a signal, an *mmap* system call to establish the mappings, and possibly a *munmap* system call to simulate the TLB entry being replaced. For programs with many TLB misses, like the SPEC92 alvinn benchmark, this could result in a slowdown of over 50. For programs with excellent TLB locality, like the SPEC92 022.li benchmark, virtually no slowdown is seen.

6.0 Limitations of the simulation environment

Although the simulation techniques described so far allow for a fairly realistic and detailed simulation, there are several limitations that are worth mentioning. First of all, while the simulator runs very quickly compared to other simulation techniques, there are still slowdowns. Since the different slowdowns depend on the type of kernel operations that are performed, timings generated by the simulated environment are not necessarily similar in absolute or relative terms to the real hardware. The difference in slowdowns makes it difficult to do performance tuning using the simulation system as is.

The timing of multiprocessor SimOS executions can be even more troublesome. If there are more simulated CPUs than available CPUs in the machine on which the simulation is running, the simulated CPU appears to run in an intermittent fashion. When the process that simulates a CPU is scheduled on a real CPU, the simulated CPU will appear to run. However, when it gets descheduled, the CPU will freeze until it is scheduled again. This interleaving of CPU execution, while good for detecting bugs and race conditions in the system software, does not represent the probable execution on the real multiprocessor.

Performance timings using the simulation environment can still be accomplished in two ways. The first is to use event counts to compare the performance of two techniques. If you only care about a count of events, such as page faults, TLB misses, or disk accesses, then the timing mismatch is less of a concern. If more precise timing is needed, it is always possible to switch to a more detailed simulator, as describe in Section 4.2. By simulating the CPU, caches, memory system, etc. in software, exact timings can be computed for the real hardware. The slower execution speeds of the detailed simulations is only used for those sections of execution where more precise timing is needed.

The simulation environment has another limitation—it is not readily apparent how to efficiently simulate certain features of modern architectures and operating systems. These features can be roughly divided into two classes: features that require changes in the OS environment for the simulation to run efficiently, and features that do not require OS modification but which simply can not be simulated at acceptable speeds. In this section we discuss some of the problematic features we encountered during our experience with each of the OS ports.

One class of features that cannot be efficiently simulated are those architectural elements that the operating system does not allow the user to control. Examples of these features are the SPARC register windows and the MIPS k0 and k1 registers. In the SPARC architecture, user-level programs can use register windows. However, these programs cannot get control when register window underflow and overflow traps occur. Because the underlying kernel does all the register window operations, the register window trap handlers of the simulated kernel are never invoked. In the simulation, code that explicitly manipulates the register windows has to be removed. This means that bugs related to the behavior of register windows might not be reproducible in the simulated system.

In the MIPS architecture, the two general-purpose registers k0 and k1 are reserved for use by the kernel trap handlers. These registers are used in trap handlers without being saved. The problem occurs when we try to run the simulated kernel at user-level. Because these registers are normal general-purpose registers, the code will execute. However, there is the possibility that the registers may be changed at any time by the underlying operating system trap handlers. These registers are usable only in trap handlers, but, as accesses to them do not generate illegal instructions, we cannot simulate them.

Our solution for handling the k0 and k1 registers was to just change the trap handler in the simulation environment to not use these registers. Fortunately, k0 and k1 can only be used in assembly language and only occur in two files in IRIX. This change required a modification of less than 30 lines of the source code.

An example of a feature that was difficult to simulate, but didn't require any code changes, was the virtual coherency exceptions (VCE) of the MIPS R4000 architecture. These exceptions are used to maintain coherency in the CPU's virtually indexed primary caches. Since VCEs are hidden from the user-level by the IRIX operating system, the simulated trap handlers never receive a VCE exception. To have the simulator detect when VCEs occur would require running a cache simulator against every CPU access. This would greatly slow the simulation system. The only disadvantage of the current simulator is that the code for handling these VCEs is not tested.

Several other architectural and operating system features cause problems for our simulation techniques. The real CPU cannot be used to execute those instructions which vary in their semantics, depending on whether they are issued user or system mode. The MIPS k0 and k1 are examples of this. Another example is in SPARC version 9[22]—the semantics of accessing global registers can change in the trap handlers to use an alternative set of registers that are not visible to user processes.

The use of user virtual addresses to communicate with the kernel or hardware is an example of an operating system feature that can cause problems. It would be far too expensive to access memory-mapped devices, such as frame buffers, by catching the traps caused by references to these addresses. Indeed, most of these devices are memory-mapped into user space because they need frequent, low overhead communication.

IRIX5 uses a shared memory segment for communication between user processes and the kernel. This memory segment does not use the normal virtual memory system and is unaffected by the *munmap* system call. Fortunately, it is possible to override the memory segment by mapping a file block on top of it. By doing this, the simulator insured that the simulated user process communicated with the simulated operating system and not the real one.

7.0 Uses of the simulation technology

Besides being a platform for OS debugging and for launching more detailed simulators, we plan to use SimOS for several other proposes. We list a few below.

- ***OS counts and tracing.*** The simulation platform provides an easy environment to get counts of different operating system events. It is trivial to run a workload on the simulator and determine the number of events, such as traps, TLB misses, system calls, etc. It is also very easy to modify the kernel being simulated to count higher level events under different workloads. Because the simulator can be paused at any time without ill effects, it is natural to use it as a trace generator for other studies. For example, we added tracing to the Sprite file system that was fed into a stand-alone file system simulator. By running workloads on the Sprite system, we could generate realistic traces for the other simulator.
- ***Fault injection.*** Another use of the simulation platform is to test the error and fault recovery code of an operating system. It is trivial to simulate events such as memory word failure or a processor failure and observe that the correct recovery action was taken by the OS. This type of code is difficult to test on the real hardware. In fact, we found a bug in the file cache writeback code. This occurred when we simulated processor failure by killing the simulated CPU with a Unix SIGKILL signal.
- ***Teaching and research.*** An additional use of operating system simulators is for teaching operating system techniques and for research into new techniques. One can view the simulated OS as an advanced Nachos[3], an instructional OS project. Combining these simulation techniques with some publicly available operating system could make an interesting platform for both teaching and research.

8.0 Related work

In addition to the methods outlined in Section 2.0, our simulation techniques are closely related to several other research projects. We discuss several of the similar techniques in this section.

8.1 Virtual machines

The work presented here can be viewed as building a virtual machine like CP/67[17] or VM[21], except that here we have done it on top of Unix rather than over the raw hardware. Many of the issues explored in virtual machine research, such as architectural features needed to virtualize the CPU, pertain to OS simulation as well. The chief difference is that the virtual machine provides a better debugging environment as it makes the debugging platform more available, rather than improving the level of debugging possible. By running on top of a general-purpose OS, our simulator technology has the benefit of increased availability and an improved debugging environment.

8.2 Microkernels

Microkernels such as Mach have moved much of the operating system into user-level processes[7]. In fact, the original Mach prototype had Unix running at user level. Since the goal of this work was an improved kernel structuring, the resulting system is very different from our approach, which focused on running the OS with as few changes as possible. The approach taken in Mach and other systems was to define new abstractions and interfaces and make wholesale modifications of the existing operating system to enable the move to user-level. Clearly, techniques such as the Mach “system call redirection” mechanism and the Mach and V++[8]

external pagers provide a cleaner and potentially faster implementation than the standard System V R4 interfaces. However, our approach has the advantage of running on systems that still lack these advanced features. At any rate, our environment is meant more as a design tool rather than as an application seen by the end user. Finally, our debugging environment would still be useful for running the microkernel itself on top of Unix, sd the code remaining in a microkernel provides enough functionality in terms of virtual memory, IPC, scheduling, and device drivers.

We have several reasons for not defining new interfaces for this work. Defining new interfaces would work against increasing the availability of the debugging platforms. We would only be able to run this work on systems that used our interface. Any new interfaces we came up with would be of dubious value for most applications. This would leave the operating system developer in the position of maintaining an interface that only a few people use. Finally, by basing the system on interfaces used by many other applications, improvements to the interface for the simulation also benefit other programs. For example, speeding the debugging interface, signals, or memory map would benefit debuggers and programs that use dynamic linking and libraries.

8.3 Instruction-level simulators

Recently, several relatively fast instruction-level simulators have been implemented. The latest version of the g88 simulator[2] can boot and run an operating system, but its performance is still at best an order of magnitude slower than real-time[16]. Shade, a fast processor emulator described in [4], is actually complementary to our work. It only models the CPU from the user's perspective, omitting the MMU, privileged instructions, and devices.

Another type of operating system simulation technology comes from the companies that provide software emulations of popular personal computers. An example of this is Insignia Solutions' SoftPC[11], which can fully simulate a PC hardware and operating system to run popular PC applications. In some ways SoftPC attacks a much more difficult problem because they are trying to run on a different CPU type. In other ways, however, they are solving a different problem. They simulate a system that doesn't really support multitasking, so emulation of private address spaces is not needed. The portable common runtime approach of Xerox PARC[25] is another example of an operating system environment that can run at user-level because of the limited dependencies on the hardware being simulated.

9.0 Conclusion

In this paper we have argued that a good development and study platform is important for operating system development. We present methods that describe how an operating system can be run at user-level by simulating the machine's hardware using services provided by the underlying operating system. Our approach yields high performance that is often only 1 to 5 times slower than the system being simulated. Given the detail and speed of the simulation, these techniques represent a powerful tool for system software development. They also show promise for a number of other uses, including operating systems analysis and research.

Our immediate plans for this research is to attempt to use the tool for the different purposes described in this paper. We believe that our simulation environments will assist in software development and will also enable us to learn more about the behavior of operating systems. After all, this paper describes simulation techniques, and the true measure of simulation techniques are the results that are discovered with it.

References

[1] American National Standards Institute, *American National Standard for Information Systems: Small Computer System Interface (SCSI),* ANSI, New York, NY, 1986.

[2] Robert Bedichek, “Some Efficient Architecture Simulation Techniques,” *Winter 1990 Usenix Technical Conference*, Jan. 1990.

[3] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson, “The Nachos Instructional Operating System,” *Winter 1993 Usenix Conference,* Jan. 1993.

[4] Robert F. Cmelik and David Keppel, “Shade: A Fast Instruction Set Simulator for Execution Profiling,” SMLI TR-93-12, Sun Microsystems Laboratories, Inc., July 1993.

[5] Kaivalya M. Dixit, “New CPU Benchmark Suites from SPEC,” *37th Annual IEEE International Computer Conference — COMPCON Spring ‘92,* San Francisco, CA, Feb. 1992, pages 305–310.

[6] R. A. Gingell, J. P. Moran, and W. A. Shannon, “Virtual Memory Architecture in SunOS,” *Proceedings of the Usenix Technical Conference,* Summer 1987.

[7] David Golub, et al, “Unix as an Application Program,” *Proceedings of the Usenix Technical Conference,* Summer 1990, pages 87–95.

[8] Jim Gray, “Why Do Computers Stop?”, Tandem Technical Report.

[9] Kieran Harty and David R. Cheriton, “Application-controlled Physical Memory Using External Page-Cache Management,” *ASPLOS V — Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, Oct. 1992, pages 187–192.

[10] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, “The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor,” To appear in *ASPLOS VI — Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct 1994.

[11] Insignia Solutions, “SoftPC/Macintosh Questions and Answers,” Promotional Literature, March 1991.

[12] Gerry Kane, *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, NJ, 1989

[13] Michael D. Kupfer, “Sprite on Mach,” *Proceedings of the Third Usenix Mach Symposium,* Santa Fe, New Mexico, April 1993, pages 307–322.

[14] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, “The Stanford FLASH Multiprocessor,” In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, Apr 1994.

[15] Peter S. Magnusson, “A Design For Efficient Simulation of a Multiprocessor,” *MASCOTS ‘93 –Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, La Jolla, California, January 1993.

[16] Peter S. Magnusson, personal communication.

[17] R. A. Meyer and L. H. Seawright, “A Virtual Machine Time-Sharing System,” *IBM Systems Journal*, vol. 9, no. 3, 1970, pages 199 –218.

[18] John K. Ousterhout, et al, “Sprite Network Operating System,” *COMPUTER*, vol. 21, no. 2, Feb. 1988, pages 23–36.

[19] John K. Ousterhout, “Why Aren’t Operating Systems Getting Faster As Fast As Hardware?”, *Proceedings of the Usenix Technical Conference,* Summer 1990, pages 247–256.

[20] Richard Rashid, et al, “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.” *IEEE Transactions on Computers,* vol. 37, no. 8, Aug. 1988, pages 896–908.

[21] L. H. Seawright and R. A. MacKinnon, “VM/370 — A Study of Multiplicity and Usefulness,” *IBM Systems Journal*, vol. 18, no. 1, 1979, pages 4–17.

[22] The SPARC Architecture Manual, Version 9, SPARC International, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[23] Richard Stallman and Roland H. Pesch, *Using GDB: A Guide to the GNU Source-Level Debugger,* Free Software Foundation, Cambridge, MA, 1992.

[24] Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska, “Techniques for File System Simulation,” Technical Report 92-09-08, Department of Computer Science and Engineering, University of Washington, Seattle.

[25] Mark Weiser, et al, “Portable Common Runtime Approach to Interoperability,” *Operating Systems Review,* vol. 23, no. 5, Dec. 1989, pages 114–122.