SunOS Minix: a tool for use in Operating System Laboratories

Paul Ashton
Daniel Ayers
Peter Smith

Technical Report COSC 05/93

Department of Computer Science
University of Canterbury, Private Bag 4800
Christchurch, New Zealand

SunOS Minix: a tool for use in Operating System Laboratories

Paul Ashton, Daniel Ayers and Peter Smith
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
email: paul@cosc.canterbury.ac.nz

Abstract

Laboratory work is an essential part of the learning experience in many areas of Computer Science, and this is particularly true in the area of operating systems. To support laboratory work in operating systems, we have created SunOS Minix, a version of the Minix operating system that runs as a process under Sun Unix (SunOS). To date, projects for two advanced classes on operating systems have involved extensive work with the SunOS Minix source code. Also, we are in the process of developing a novel graphical monitoring and control interface that will make SunOS Minix a powerful tool for use in introductory operating system laboratories.

1 Introduction

Laboratories have an important role in Computer Science education [19]. Laboratory projects are used extensively in teaching of operating systems, with many operating system laboratories described in the literature (see, for example, [1]–[18]). In introductory laboratories, investigation, using readily available monitoring tools, of an existing operating system may be sufficient. In more advanced laboratories, students often delve deeper and either create some or all of an operating system, or read and perhaps change the source code of an existing operating system.

A reason for the high level of interest in operating system laboratories is that many difficulties must be overcome in designing and implementing them. Providing a **realistic** operating system environment in which students can read and write source code is a considerable challenge. Difficulties include: obtaining operating system source code; providing suitable real or simulated machines; and helping students come to terms with large and complex systems, particularly with the concurrent execution that occurs within an operating system.

Use of a small operating system created for use in teaching of operating systems has a number of advantages:

1. Source code is much easier to obtain.

2. The relatively small size of the operating system (compared to a production operating system) means that students find the source code much more accessible.

3. A high degree of realism can be obtained because students are dealing with an actual operating system.

We decided to use Minix [18], a widely used "teaching" operating system, as the basis for several operating system laboratories. As Minix was not available for our main computing environment, a network of Suns running SunOS, we created SunOS Minix, a version in which each copy of Minix runs in the environment of a SunOS process. Each of our Suns can support many instances of SunOS Minix, each of which is a multi-user operating system that provides preemptive multitasking for a considerable range of system utilities, as well as user-written programs. SunOS Minix has been in use for two years as the basis for operating system laboratories in an advanced operating system course.

Currently, our introductory operating system laboratories consist of several experiments performed under SunOS. Many of these could be much better supported by SunOS Minix. To this end, we are in the process of designing and implementing a sophisticated SunOS Minix monitoring and control interface that is for the most part implemented outside SunOS Minix. (The fact that SunOS Minix is hosted by a powerful general-purpose operating system raises the possibility of providing such an interface.) This novel interface will provide a student with a wide range of information on the behaviour of a running version of SunOS Minix and will give the student the ability to control the execution of SunOS Minix by (for example) enabling various breakpoints. The initial version of the interface will include (amongst other things): the ability to start and stop SunOS Minix at will and at various predefined breakpoints; a continuously updated view of the process tree; message passing traces and animations; and the ability to browse through the operating system data structures using a hypertext style of interface.

The remainder of the paper is structured as follows. Approaches to operating system laboratories taken by others are summarised in Section 2. Minix is introduced in Section 3, and SunOS Minix is described in Section 4. An overview of two projects that have been based on SunOS Minix is given in Section 5. The design of a novel monitoring and control interface for SunOS Minix is outlined in Section 6. Finally, a summary is presented and some conclusions are drawn in Section 7.

2 Background

A wide range of operating system laboratories has been described in the literature. We classify these laboratories in two ways: by the type of projects undertaken, and by the type of the operating system experimented with.

We classify projects into three categories:

1. Non-source code projects—those that do not involve reading or writing operating system code.

2. Operating system modification projects—those that involve reading and perhaps modification of source code for an existing operating system.

3. Operating system creation projects—those that involve writing all or some of an operating system from scratch.

The type of operating system experimented with gives some indication of how close the system is to being a "real" operating system. At one extreme are queueing models, in which very high level abstractions of both the operating system and its workload are used. At the other extreme are production operating systems that are processing real workloads. We now describe several categories of operating system used in laboratories, and for each we describe the types of project that can be set:

- Total simulation of operating system operation. Simulators of this type can be used in non-source code projects. Since there is little or no operating system code, such simulators cannot be used in operating system modification or operating system creation projects.

  In [15], Stafford describes a collection of ten simulation programs intended to provide insight into the tradeoffs involved in various design decisions. Information from each simulation is recorded in detail in a text file, and can also be presented in various graphs. The simulations include investigation of several CPU scheduling policies and of various disk request scheduling policies.

  In GRAPHOS [2], animation is used to show in detail the operation of several operating system components. Animations are drawn from the areas of concurrent processes, processor scheduling and memory management.

- Operating systems running directly on the bare hardware. Production operating systems can be used as the basis for operating system projects. A non-source code project for a production operating system could involve using monitoring tools to investigate the internal operation of the system (by observing the process hierarchy and the contents of various tables, for example), or might require the student to write programs that use various operating system facilities by making appropriate system calls. A laboratory described in [12] involves investigation of Unix processes by using standard SunOS monitoring tools.

  While production operating systems often provide a reasonable environment for non-source code projects, they are usually less well suited to operating system modification projects (and operating system creation projects are out of the question!). The major reasons for this are the difficulty of getting the source code for production operating systems (cost and copyright problems) and their complexity. Shub observed that "workstation operating systems seem to be too complex for undergraduates to experiment with in a first course in operating systems" [14].

  Because of these problems, several "teaching" operating systems have been developed, including Minix [18] and Xinu [5]. These operating systems provide all of the important operating system services, but have been kept as small and simple as possible to enable students to gain an understanding of the entire system. These operating systems can be used for non-source code and operating system modification projects, with operating system modification projects reported on most frequently in the literature [1], [3], [7], [8].

  Some operating system creation projects involve implementing a very small operating system directly on the hardware (see, for example, [9] and [13]).

- Partial simulation of operating system and/or hardware operation. The degree of simulation varies considerably. At one extreme are systems in which students are asked to write modules that conform to given interfaces, and that perform in relative isolation functions such as memory allocation and process scheduling. These modules are invoked by a controller based on a simulated workload (see [11], for example).

At the other extreme are fully fledged operating systems designed to run on a simulated machine rather than on an actual machine. Some of these systems come with considerable amounts of application software, although most are driven by simulated workloads. Examples include VXinu [16], and systems such as those described in [6], designed to run under a VM/370 virtual machine. Such systems can also be run directly on 370 series hardware.

Non-source code, operating system modification and operating system creation projects can be done in these "partially simulated" environments.

For several years we have been using non-source code projects based on production operating systems—initially PRIMOS and more recently SunOS. Two years ago, we decided that we would also like to give students at an advanced level the opportunity to work with operating system source code. In choosing an operating system, we had to decide:

1. On the type of projects that would be undertaken. Both operating system modification and operating system creation projects have been described in the literature, with both approaches apparently successful. To date, there is no conclusive evidence as to which is the better approach [8], if indeed one is substantially better than the other.

2. On the type of operating system that would be used. Although running an operating system directly on the hardware provides the most realism, some problems exist with this approach. One is that students must have a dedicated machine while they are running their copy of the operating system. A second is that there is very little scope for the type of support tools (debuggers and monitors for example) that can be constructed for partially simulated environments. Donaldson describes a batch operating system developed for use in a VM/370 environment [6], and comments that a great benefit of using that environment is the availability of powerful debugging tools that can be used to debug operating systems that run on top of VM.

We decided to use operating system modification projects based on Minix. Our main departmental computing system is a network of Sun workstations. Because students use three large servers, and it is infeasible to set operating system laboratories that involve use of the bare hardware, we created SunOS Minix. This gave us a hosted implementation of a complete time-sharing operating system, with full source code and an extensive set of user commands, including shells, editors, and file utilities. We are aware of few similar systems:

- A PC simulator exists that allows Minix for the IBM PC to be run under VAX Unix [18]. We tried a version of this simulator that had been ported to the Sun, but found execution to be extremely slow.

- VXinu is a version of Xinu that runs as a Unix process [16]. VXinu was developed independently during the period in which SunOS Minix was written. A detailed comparison with SunOS Minix is not possible at this point because all documentation on VXinu is written in Japanese.

- Nachos is a very small instructional operating system that runs as a Unix process [4]. Its functionality is limited in many areas because of a desire to keep the amount of code to a minimum and because a set of projects used with Nachos requires students to implement (or reimplement) substantial components of Nachos. Nachos does, however, include some features (basic network support for instance) that are seldom found in other instructional operating systems.

  Nachos does not come with any user-mode programs, although students can write their own. All user-mode executables are interpreted, although the operating system runs in native mode.

3 Minix

Because SunOS Minix is a port of Minix, we will give an overview of Minix before going on to describe SunOS Minix. Minix was designed specifically for use in teaching of operating systems [18]. To this end:

- Minix has been kept (relatively) small and simple.
- Extensive documentation of the internals of the original version of Minix is available in [17]. Although the current version of Minix differs in many ways from the original version, [17] still gives a very good overview.
- The source code of Minix is inexpensive and, once a copy has been purchased by a teaching institution, can be freely copied within that institution.

Although Minix is small, it is nevertheless a preemptive, multitasking operating system, capable of supporting several users simultaneously. Minix comes with a substantial number of system utilities, including shells, editors, compilers and libraries, and file utilities. Indeed, the Minix operating system and utilities are maintained under Minix.

Externally, Minix is very similar to Unix Version 7. Nearly all of the Unix Version 7 system calls are supported, most of the system utilities are provided, and a C compiler and a version of `make` are included. Version 7 was chosen as a base because it is possible to construct an operating system that, while it supports the Version 7 system calls, is small enough to be understood by students. A benefit of basing Minix on Version 7 is that most students are already familiar with Unix.

Internally, Minix is highly modular, consisting of several system processes organised into a number of independent programs. System processes communicate with each other and with user processes through message passing. The Version 7 kernel, on the other hand, is a single program. There are no system processes in Version 7; instead user processes enter and leave the kernel through system calls and returns.

The internal structure of Minix is shown in Figure 1. Layers 1 to 3 comprise the Minix operating system, with applications running in Layer 4. The operating system code is linked into three totally separate programs—`mm` (the memory manager), `fs` (the file system), and `kernel` (layers 1 and 2). Processes in layer 2 and interrupt handlers in layer 1 can communicate by message passing or shared memory; all other communication is through message passing.

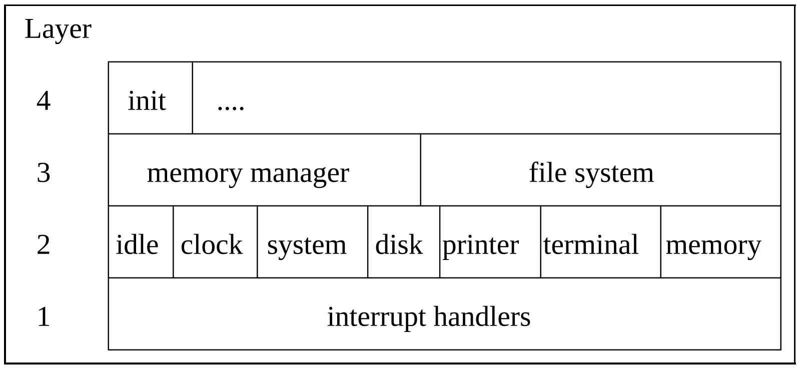

Figure 1: Internal structure of Minix

Layer 1 implements communicating sequential processes, thus allowing the remainder of Minix to be implemented as a set of communicating sequential processes. Layer 1 is a group of interrupt handlers, with its own stack space within `kernel`. Device interrupts are converted into messages to the appropriate device driver process in layer 2.

Layer 2 contains device driver processes. Some processes (`disk`, `clock`, `terminal`, `printer`) interface with actual devices. `system` provides an interface between `kernel` and the layer 3 processes. `memory` is a pseudo-device that implements ram disks and provides access to `kernel` address space and to physical memory.

Layer 3 contains the memory manager and the file server. Every "system call" made by a user process in layer 4 is converted into a message to one or other of these processes. Layer 4 contains a Unix-like process hierarchy rooted at process 1 (`init`).

This layered structure provides students with a highly modular system to study, and isolates hardware-dependent code in layers 1 and 2.

One limitation of Minix needs comment. Memory management is not particularly sophisticated. When a program is executed, it is allocated a fixed amount of memory (as specified in the header of the executable file) for its entire execution. Also, neither swapping or paging is provided.

The original version of Minix was available from Prentice-Hall for the IBM PC, XT, and AT. The version of Minix currently available from Prentice-Hall (1.5) is available for the same machines plus the Amiga, the Atari ST, the Sun SparcStation and the Macintosh. Minix on the Macintosh is hosted under MacOS; all other versions run directly on the hardware.

4 SunOS Minix

In SunOS Minix, each instance of the Minix operating system runs as a SunOS process—that is, the virtual machine on which SunOS Minix runs is the SunOS process abstraction. The processor time available for SunOS Minix is that fraction of the CPU time given to the SunOS process. Each SunOS Minix disk partition is stored as a SunOS file. Clock interrupts are SunOS signals generated by SunOS at regular intervals (the default is every 50ms). The SunOS Minix console is the terminal connected to the SunOS process. Thus when SunOS Minix is started from a terminal session, the interface it presents to the user is exactly the same as the user would experience had they just booted (say) PC Minix.

Our starting point was Macintosh Minix and our initial port was to Suns with the SPARC architecture running SunOS 4.1.1. An overview of the changes made in creating SunOS Minix is given below.

- Bootstrapping. A small SunOS program (`minix`) is responsible for processing a configuration file, allocating memory for SunOS Minix, loading and relocating the SunOS Minix "image" and then jumping to the startup code within the SunOS Minix `kernel` proper. The configuration file specifies one or more SunOS Minix disk device $\rightarrow$ SunOS file mappings, as well as the amount of memory to allocate for SunOS Minix. `minix` allocates the requested amount of memory in the data segment of the SunOS process, and loads the image, consisting of `kernel`, `mm`, `fs` and `init`, at the beginning of the area allocated.

- Layer 1. Layer 1 required several modifications. Context switching code had to be written in SPARC assembler. Handlers for the SunOS Minix "interrupts" (SunOS signals) had to be written. `SIGIO` indicates that terminal input is available, `SIGALRM` indicates that a clock tick has occurred, and `SIGUSR1` is used to force execution into `kernel` when a SunOS Minix process makes one of the three Minix system calls (send a message, receive a message, send a message and wait for a reply).

- Layer 2. The disk and terminal device drivers were changed to perform reads and writes by reading from and writing to SunOS file descriptors. The `idle` process was changed to cause the SunOS process to wait for a signal instead of busy waiting.

- Layer 3. The memory manager had to be modified so as to be able to relocate SunOS Minix executables. Relocation during loading is required because the virtual address of the beginning of a SunOS Minix program is not known until run-time (relocation is unnecessary in PC Minix because of the availability of base registers).

- Compilation. All compilation (of the operating system and utilities) is done under SunOS. Appropriate compilation flags are used to ensure that relocation information remains in executable files. Image files are created under SunOS by concatenating `kernel`, `mm`, `fs` and `init`. For all other SunOS Minix programs, the SunOS executable is converted to SunOS Minix format, then can be read into SunOS Minix using the SunOS Minix `sunread` command.

- Multiple logins. A single instance of SunOS Minix can have multiple users logged into it. The SunOS `mlogin` command opens a connection (through a Unix domain socket) to a currently running instance of SunOS Minix. Upon connection, the user can log into SunOS Minix in the usual way.

The most difficult parts of the port to SunOS were implementing context switching between SunOS Minix processes and implementing relocation.

SunOS Minix has proved to be a good environment for doing operating system projects. A very high degree of realism has been achieved, despite the fact that SunOS Minix is running on top of SunOS. SunOS Minix is a preemptive multitasking system that can support multiple users running a wide variety of SunOS Minix applications. No dedicated hardware is required, and any number of instances of SunOS Minix can be run on a single Sun at the same time as each other and any other workload. SunOS Minix filesystems can be shared in a read-only fashion so, for example, only one copy is needed of the filesystem containing the binaries of the system utilities.

A drawback of SunOS Minix is that some of the device drivers, the disk device driver in particular, are much simpler than their counterparts in systems running directly on the hardware. Emulation of these devices could, however, be made more realistic. Also, SunOS Minix could be enhanced to allow swapping and demand paging.

SunOS Minix has been available for over a year, and has been installed in several sites. The current version (2.0) supports Sun 3s as well as Sun 4s (under SunOS 4.1.x and Solaris 2.x), and is available (as a set of patches to Minix 1.5 for the Macintosh or PC) via anonymous ftp in the file `smx_2_00.tar_z` in the `unix` directory of `csc.canterbury.ac.nz`.

5 Projects based on SunOS Minix

SunOS Minix has been used in two operating system modification projects in an advanced operating system course. In 1992, students were asked to implement memory protection. In version 1 of SunOS Minix, all of the memory in which SunOS Minix ran was readable, writable and executable at all times, so any process could corrupt its own executable code as well as the address spaces of all other processes. Ideally, when a SunOS Minix process is executing it should not have write access to its code, and should have no access outside its own address space. The students were asked to implement this type of memory protection in SunOS Minix by making use of the SunOS `mprotect` system call, which changes the protection of one or more pages in the address space of a SunOS process. The project entailed making the following changes:

1. The context switching code had to be changed to disable access to the address space of the process being switched from, and to enable access to the address space of the process being switched to. Because `kernel` is responsible for changing protection, access to `kernel` could not, however, be totally disabled. The stack used by the interrupt handlers had to be writable, code executed prior to enabling access to `kernel` and after disabling access to `kernel` had to be readable and executable, and data containing details of the address space of the executing process had to be readable. Nevertheless, `kernel` cannot be corrupted by a process executing outside it because the only area of `kernel` that remains writable is the interrupt stack, and nothing is stored there when execution is outside `kernel`.

2. In several places in SunOS Minix, it had been assumed that `kernel` could access directly the address space of any process. With memory protection enabled this was no longer true, so calls to change protection had to be placed around such accesses. These calls were necessary in the `kernel` library functions for copying, in handling the message passing system calls, in performing disk reads and writes, and in the memory manager where relocation is performed.

Feedback from students indicated that they found this project to be challenging but very rewarding. The ability to work with the source code helped to dispel for them the mystique surrounding the source code of an operating system. Also, students discovered the difficulty of debugging operating system code. For this project, there were places in `kernel` from which they could not call `printf` because they had disabled access to it!

In 1993, the project set was rather different. The students were required to audit SunOS Minix and document all potential race conditions, and how they are avoided (if at all). The only multi-threaded program in SunOS Minix is `kernel`, in which the interrupt handlers and device drivers execute, so the student's attention was focussed on `kernel`. Because one device driver cannot preempt another, race conditions can only occur when an interrupt handler interrupts a device driver or another interrupt handler. A major goal of the project was to improve understanding of concurrency.

SunOS Minix has proved to be a good environment for operating system modification projects. Access to the environment has not been a problem because students don't need any dedicated hardware. The ability to share the larger SunOS Minix file systems in a read-only fashion has helped to keep disk space requirements down (1Mb of SunOS disk per student allows students to run SunOS Minix). Finally, SunOS Minix has proved a very robust environment.

6 The SunOS Minix control and monitoring interface

Students in a first course on operating systems do not have the background to deal with projects involving modification or creation of an operating system. In our current introductory laboratories for operating systems, students perform a number of experiments that involve investigation of the operation of SunOS. Process management, process synchronisation, memory management and file management are among the areas covered. Software used in the laboratories includes standard SunOS monitoring tools (including `ps`, `top`, `trace`, and `pstat`), a locally written monitor (`asdump`, which prints details of the address space of a process), and small programs that use various operating system services by making system calls.

While we have been reasonably happy with the laboratories, there are a number of difficulties in basing them on a production operating system:

- Not having the operating system source code makes it difficult for the instructor to find out what is happening within the operating system and to create new monitoring tools designed to illustrate aspects of operating system operation that are poorly illustrated by the standard tools.
- Students are not able to stop and then restart the operating system, making it impossible to get a consistent snapshot of the state of the operating system.
- Students have no ability to influence operating system policies. It is impossible, for example, for a student to select the process to run next whenever a context switch occurs.
- Because students are using the main departmental computers, security risks exist for some types of monitoring.

To substantially improve support for introductory operating system laboratories, we have designed and are in the process of implementing a monitoring and control interface for SunOS Minix. The interface is an X-windows application running under SunOS. When using the interface, a student has on their screen many windows that are related to the execution of an instance of SunOS Minix. The student can enter SunOS Minix commands into any of the

SunOS Minix terminal sessions and sees instantly how the operating system is dealing with them. Windows that may be on the student's screen include:

- The terminal session for the SunOS Minix console.
- Additional SunOS Minix terminal sessions.
- The control panel. The control panel allows the user to specify the points at which SunOS Minix should be suspended. Possible points include: the sending or receiving of some message, the occurrence of an interrupt, and the creation of a new process. The control panel also allows the user to resume execution of SunOS Minix, and to start up monitors.
- Displays from monitors. A wide variety of displays can be envisaged that help to reinforce basic concepts. Some included in the current design are:
  - A tree representation of all or part of the process hierarchy that is updated in real-time as the process hierarchy changes.
  - A hypertext browser for data structures within `kernel`, `mm` and `fs` and on SunOS Minix disk partitions. The browser allows access to (amongst other things) the process tables, open file tables, system queues (including the ready list), various buffers, and the file buffer cache (enabling comparisons between the cache and what is on disk).
  - Traces and animations of the message passing that occurs between the various SunOS Minix processes.
  - A graphical presentation of how the state of one or more processes varies over time. A Gantt chart is one way in which this information can be reported.
  - The ability to select the process to run whenever a new process is dispatched (thereby overriding the scheduler).

  Displays can be driven from data flowing directly from an executing instance of SunOS Minix or from a playback of what occurred in a previous session. During playback both displays from monitors and Minix terminal sessions can be played back. Also, many instances of the same monitor can be connected to a single instance of SunOS Minix (or a playback), allowing (for example) an instructor to broadcast to many students displays of what is happening in an instance of SunOS Minix that the instructor is controlling.

  The goal of this environment is to provide a system that can be used in introductory operating system laboratories. Development of monitors will be driven by this goal.

This novel monitoring and control interface appears to have tremendous potential for use in introductory operating system laboratories. This type of interface is possible because SunOS Minix is hosted under an operating system like SunOS, allowing the execution of SunOS Minix to be controlled and extensively monitored by the interface.

Considerable scope exists for extending the interface so as to better support operating system modification projects. Two such extensions are a hypertext browser for the operating system source (such as that described in [10]) and a source level debugger able to deal with the multiple system processes present in SunOS Minix. Even without such enhancements, students who have done introductory laboratories using the monitoring and control interface to SunOS Minix will be much better prepared to deal with the source code in subsequent operating system modification projects.

7 Summary

We have created SunOS Minix (Minix hosted under SunOS) so that our students can read and change the source code of an operating system. The environment provided is realistic, as Minix is a multi-user operating system that provides preemptive multitasking and comes with a wide range of application software.

Several benefits are gained by hosting Minix under SunOS. In particular, students do not need dedicated hardware, with one Sun capable of supporting many instances of SunOS Minix. Also, extensive monitoring and control software can be implemented under the host environment (SunOS), as is discussed further below. Finally, it is instructive for students to grapple with the concept of one operating system running on top of another.

A disadvantage in using a hosted operating system is that the device drivers are not very realistic. Nevertheless, interrupts for the clock and terminal devices do occur in SunOS Minix, so the important issues of race conditions and interrupt handling still arise. Modification of SunOS Minix to make the device drivers more realistic is one possible future project.

Over the last two years, SunOS Minix has been successfully used in an advanced operating systems course to give students hands-on experience with the source code of a real operating system. SunOS Minix has been robust and easy to administer. SunOS Minix has been released to the outside world and is in use in several other institutions.

Work is proceeding on an innovative graphical monitoring and control interface for SunOS Minix. The initial version will concentrate on providing support for introductory laboratories. Many monitors can be envisaged, with process tree display, data structure browsing, message monitoring and animation, and process state display all included in the current design. Scope exists for more advanced versions of the interface that provide high levels of support for projects where the SunOS Minix source is read and modified. Such versions would likely include a source code browser and a source level debugger capable of handling the multi-process nature of SunOS Minix.

The current version of SunOS Minix is a good environment for laboratories that require students to become familiar with and change the source code of an operating system. SunOS Minix is unusual in that it is a multi-user time sharing operating system (complete with applications) that is hosted by another multi-user time sharing operating system (SunOS). This allows multiple instances of SunOS Minix to run independently and simultaneously on a single Sun. The monitoring and control interface under development has the potential to result in an excellent tool for use in introductory laboratories. We have found no references to similar interfaces in the literature. Further development of the interface will provide much improved support for advanced laboratories.

References

[1] G. Aguirre, M. Errecalde, R. Guerrero, C. Kavka, G. Leguizamon, M. Printista, and

R. Gallard. Experiencing Minix as a didactical aid for operating systems courses. *Operating Systems Review*, 25(3):32–39, July 1991.

[2] Daniel A. Cañas. GRAPHOS: A graphic operating system. *SIGCSE Bulletin*, 19(1):201–205, February 1987.

[3] Stephen W. Chappelow, Steven F. Ackerman, and Stephen J. Hartley. Design and implementation of a swapper for the MINIX operating system. *SIGCSE Bulletin*, 22(4):55–59, December 1990.

[4] Wayne A. Christopher, Steven J. Proctor, and Thomas E. Anderson. The Nachos instructional operating system. In *Proceedings of the 1993 Winter USENIX Conference*, pages 479–488, January 1993.

[5] Douglas Comer. *Operating System Design: The XINU Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[6] John L. Donaldson. Teaching operating systems in a virtual machine environment. *SIGCSE Bulletin*, 19(1):206–211, February 1987.

[7] R. Guerrero, L. Leguizamon, and R. Gallard. Implementation and evaluation of alternative process schedulers in MINIX. *Operating Systems Review*, 27(1):79–100, January 1993.

[8] Stephen J. Hartley. More experience with MINIX in an operating systems lab. *Computer Science Education*, to appear.

[9] Larry Hughes. Teaching operating systems using Turbo C. *SIGCSE Bulletin*, 24(1):181–186, February 1992.

[10] William R. Hutchison, Jeffry H. Tischer, Charles A. Johnson, Dan A. Dargel, and Brian A. Rudolph. A source code navigation tool for the XINU operating system. *SIGCSE Bulletin*, 23(1):304–308, March 1991.

[11] Michael Kifer and Scott A. Smolka. OSP: An environment for operating systems projects. *Operating Systems Review*, 26(4):90–100, October 1992.

[12] John Penny and Paul Ashton. Laboratory-style teaching of computer science. *SIGCSE Bulletin*, 22(1):192–196, February 1990.

[13] Margaret M. Reek. An undergraduate operating systems laboratory course. *SIGCSE Bulletin*, 22(1):171–175, February 1990.

[14] Charles M. Shub. Should undergraduates explore internals of workstation operating systems. *SIGCSE Bulletin*, 22(1):111–115, February 1990.

[15] Gary J. Stafford. An operating system simulator for the Apple Macintosh. Technical Report 91/06, University of Wollongong, Department of Computer Science, December 1991.

[16] Yoshikatsu Tada. A virtual operating system 'VXinu'—its implementation and problems. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J75-D-I(1):10–18, January 1992.

[17] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[18] Andrew S. Tanenbaum. A UNIX clone with source code for operating systems courses. *Operating Systems Review*, 21(1):20–29, January 1987.

[19] A. B. Tucker *et al.* A summary of computing curricula 1991. *Communications of the ACM*, 34(6):68–84, June 1991.