Proc. 1985 *Summer Usenix Conference, Portland, OR, June 1985*.

A XINU Virtual Machine

Jonathan Bachrach
John Wallerius
Jehan-François Pâris

Department of Electrical Engineering & Computer Sciences
University of California, San Diego
La Jolla, CA 92093

ABSTRACT

XINU is a simple multi-tasking operating system designed to run on a set of LSI-11's linked by a store-and-forward ring network. It was designed to provide an environment where students could learn operating system concepts by working with a well-documented, complete system. We present here a XINU virtual running on a VAX architecture under Berkeley 4.2 UNIX[†]. Our paper discusses the XINU port and the enhancements that were added to the original XINU system. These enhancements include an interactive shell, a debugger and enhanced networking facilities.

1. INTRODUCTION

Many operating system concepts are better understood if students can have hands-on experience with a complete running operating system. Students should be able to see how functions like process management, memory management, interprocess communication, and so forth are implemented in a real system. They should also be given the opportunity to experiment with the system by modifying some of its functions and evaluating the impact of these modifications on the system performance.

Traditional multi-tasking operating systems are not suited for this purpose. They are too complicated and often contain proprietary information. Besides, any attempt to modify the system kernel would result in having the machine unavailable for other uses for long periods of time, to say nothing of the security risks involved. These considerations have led to the emergence of a new type of operating systems that are specially designed to be teaching tools. These so-called "toy operating systems" are much smaller than regular systems. Since efficiency is not anymore a key consideration, the various functions of the system can be implemented in a straightforward way, so as not to confuse the novice.

The TOY Operating System developed by Fabry et al. at the University of California, Berkeley, is a prime example of such a system [Fabr83]. TOY runs as a user process on a host computer and simulates a fictitious machine with a very simple instruction set. Since the TOY operating system is itself written in a high-level language (C in the current implementation), students can quickly learn to modify it. The impact of these modifications can be later assessed by running user programs on the modified system. Since the TOY virtual machine is simulated at the instruction set level, all TOY machine instructions are interpreted by software. A very large range of memory management schemes, which includes virtual memory, can thus be simulated.

[†] UNIX is a Trademark of Bell Laboratories

The TOY system lacks, on other hand, any compiler or assembler. As a result, all user programs need to be written in the TOY machine language, which might soon prove to be a rather tedious task.

The HOCA system developed by Babaoglu et al. at Cornell University [Baba83b] does not suffer from this limitation. HOCA runs on the CHIP virtual machine, which is a slightly modified PDP-11 [Baba83a]. CHIP and HOCA were written to run on a VAX 11/780. They can thus take advantage of the existence of a PDP-11 compatibility mode on that architecture. Like TOY, HOCA lacks any facility for implementing distributed systems. We felt this to be an important limitation in an environment like ours where students typically take a two-course sequence in operating systems with the first course dedicated to traditional operating systems and the second one covering distributed systems. We thus decided to turn to a third educational system, XINU [Come84]. XINU was designed by Comer at Purdue University to run on a set of LSI-11's linked by a store-and-forward ring network. It is written almost entirely in C and is exceptionally well documented. Because of its simplicity and its clarity, it constitutes an excellent tool for illustrating operating system concepts.

The only drawback of the system lies in the fact that it requires its own dedicated hardware. If we wanted to let students experiment with the XINU networking facilities, we would have to provide each group of students with access to their private set of LSI-11's. We thus decided to build a XINU virtual machine to run on a host computer. This also gave us an opportunity to enhance the existing XINU facilities by improving the XINU user interface and by allowing the realistic simulation of a wide range of network topologies.

An initial implementation of our XINU virtual machine has been running for several months on VAX 780's and 750's under Berkeley 4.2 UNIX. Our virtual machine simulates in software all LSI-11 functions required by the XINU software.

Most of the work in porting standard XINU to the UNIX environment involved XINU kernel modifications and the writing of special I/O simulation code. Although the machine dependent parts of the code differ from the original XINU implementation, the XINU programmer's interface is compatible with the standard version. Since XINU is written in C, all XINU programs run in the native instruction set of the host computer, which results in a very lean and very efficient virtual machine. This approach has the major drawback of limiting our ability to implement in XINU any memory management scheme that would require the trapping of invalid address exceptions.

The remainder of this paper focuses on the most interesting aspects of our port, namely the XINU kernel, the I/O subsystem, the enhanced networking facilities, and the new XINU user interface. The two last sections review our experience with XINU as an educational tool and contain our conclusions.

2. THE XINU KERNEL

The XINU machine is simulated on the host machine by multiplexing one UNIX process into many XINU processes. A portion of the full UNIX process memory space is allocated for XINU and is used in the same way that standard XINU uses the LSI-11 memory. As in XINU, this memory space contains a text segment, an initialized data area an uninitialized data area, and a free data area, from lowest to highest address respectively. The free data area is further subdivided into a heap space and a stack space, where the heap space provides allocatable storage and the stack space provides one stack per XINU process. The stack space grows from highest memory and the heap space grows from low memory.

In the rest of this section we describe how we ported the kernel to the VAX. This description is rather technical and detailed, and is not necessary for an understanding of the rest of the paper. A little background information on the VAX architecture will be necessary.

The VAX instruction set provides a sophisticated procedure call instruction. When this instruction is executed, the registers that the programmer (or compiler writer) wants saved are placed on the stack. Some registers, like the pc and processor status register, are always saved. A register called the frame pointer (fp) is automatically set to point to this save area, so that when the called procedure returns, the previous machine state can be restored efficiently. One of the saved registers is the frame pointer that points back at the previous context. For example, if A calls B and B calls C, the frame pointer points at the saved registers of B, one of which is the fp that points at the saved registers of A. Two other registers are always saved, the stack pointer (*sp*), and an argument pointer (*ap*).

As a result, saving the frame pointer, the stack pointer, the argument pointer and the status register of a XINU process in some table allows XINU to start executing some other XINU process on some other stack while guaranteeing that the first process can be continued at any time. To go back to the first process, the XINU kernel would have to load the *fp*, *sp*, *ap* and status register with the values from the table, and execute a return instruction. This instruction would cause the values in the stack save area that the fp points at to be loaded into the machine registers, thus continuing the process where it left off.

Here is a slightly simplified example of a context switch:

A UNIX alarm clock signal is received. An interrupt handling routine checks to see if it is time to preempt the currently running XINU process, which we designate process O, for old. If it is time to preempt process O, the handler calls a C function that selects a process N which will be the next to run on the CPU. The scheduler then calls a short assembly language procedure, passing it pointers to the process table entries for O and N.

This assembler procedure, *ctxsw* for "context switch", stores the current *fp*, *ap*, and *sp*, in the process table entry for process O. It also stores another variable, called *_clk_stat*, which plays the role of the status word, in the sense that it records how UNIX signals are to be treated. It then loads these same registers and _clk_stat with values it gets from N's table entry. The last thing *ctxsw* does is to execute a return from procedure call. This has the effect that the saved context area on N's stack gets loaded into the machine registers, and process N picks up where it left off. It is an odd use of the return instruction, since control does not end up going immediately back to the caller, O, but to N.

Let us now see what happens when a process completes. When the process was created, an initial stack frame was constructed in such a way that when the main procedure of a process completes, it "returns" to a system function, that removes the process from system tables, deallocates its stack space, and then calls the scheduler to schedule another process.

The function that creates each process actually sets up two stack frames. The first frame has the arguments the process was passed when created, and pointers to an uninitialized frame below, to be used by the system cleanup functions referred in the previous paragraph. The second frame is constructed to look exactly as if the process had been interrupted and switched away from. When the new process is selected to run, the pc that is "restored" from this second frame points at the first instruction of the new process.

It is interesting to note that the only change required in the kernel was to modify one short assembly language procedure and the process creation function, and to make small changes in the format of the process table entries.

In our XINU virtual machine, we use an interval timer to time a variety of simulated hardware events. This means that we can't completely block out the timer signals, which are analogous to clock interrupts. What we do instead is set *_clk_stat* to DISABLED. This means that when the signal handler is invoked, it can check *_clk_stat* to see whether context switches or other potentially dangerous operations are permissible, and it can increment I/O simulation delay counters.

3. THE I/O SUBSYSTEM

We ported the XINU I/O subsystem to the VAX, to allow students to write device drivers for simulated interrupt-driven hardware. The changes we introduced are minimal enough that the XINU textbook [Come84] provides sufficient guidance for programming projects, with only a small amount of additional information needed to describe interface conventions for specific pieces of simulated hardware. Some of the software described in the next two sections was still undergoing changes at the time of this writing.

3.1 Device-Independent I/O

One of the nice things about XINU is that it is fairly successful at encapsulating hardware-dependent parts of the code into a small number of routines and data structures. The same apparatus that shields users from having to know hardware details also makes the job of porting the system easier, even to an environment where all hardware is simulated. In those sections of the code where it was necessary to diverge from the standard, we use code that closely parallels the original approach, so as to be able to use the higher-level software without change.

3.2 Interrupt Handling

A good example of this is the interrupt dispatch code. In the original, all interrupt vectors point at the same dispatcher code. That dispatcher calls the appropriate handler, written in C. The dispatcher knows which routine to call because one of the fields of the vector process status word points into a table containing pointers to all the lower-half device drivers that do the actual work of responding to each interrupt. We imitate this arrangement by having the actual signal handlers set a variable with a number indicating the source of the signal, and then calling a dispatcher that uses this number in the same way that LSI-11 XINU uses the vector *psw*. This way we didn't have to change any data structures or the device driver initialization mechanism.

The actual signal generators and handlers are black boxes from the point of view of students, with only some fixed input and output parameters known. Writing code to cope with a set of these that might change presents many of the same design problems as does writing device drivers for real hardware.

3.3 The Virtual Disk and File System

The initial motivation behind putting in a disk simulator was to let students write scheduling algorithms that would treat I/O-bound and compute-bound processes differently. Other interesting projects might involve modifying the system to manage concurrent access to the same file by two different processes, implementing an alternative to unbuffered disk writes, or integrating the file system into a network.

From the point of view of XINU software, a disk drive is an array of 512-byte blocks, which must be accessed by using a particular protocol, and the accesses have certain delays associated with them. Our virtual disk is a large UNIX file. The format of this file is a structure containing: an initialized directory, a linked list of free index blocks (similar to i-nodes),a linked list of free data blocks, and a couple of text files and their associated index nodes, all conforming to XINU conventions. 4.2 UNIX has a system utility that allows a user to establish an interval timer for a given process. This interval timer sends signals to the process at a fixed rate specified by the user. The handler bound to this signal performs such functions as counting off time slices and access delays.

Access delays are achieved as follows:

1) The XINU process requesting file access is suspended.

2) The clock signal handler counts ticks for as long as a real disk would take to respond.

3) The requested data is moved (by UNIX) to or from the specified buffer.

4) The requesting process is put back on the ready queue, so it becomes eligible to regain the CPU.

Of course, while the requesting process is suspended, other XINU processes are running.

4. THE NETWORK FACILITIES

In standard XINU, software is provided with each PDP-11/02 so that they can be linked together in a ring network, through standard RS232 ports. We wanted to provide a software simulation that would allow users to set up their own network topologies connecting the UNIX processes that contain virtual machines. We also wanted to be able to be able to simulate such undesirable realities of networking as line noise and dead machines, in a controlled manner.

To maximize flexibility, the XINU network is simulated by a socket structure with a central server and clients, one client mapped to each XINU machine. The central server design localizes the control of hazardous conditions such as noisy links and dead machines. All communication is routed via the central server, with a source client sending data first to the server and then the server sending the data to the destination client. The communication lines are statically created at the boot time of the network, thus emulating physical hardware links.

The topology of the XINU network is set up through connection with the central server socket and subsequent initialization codes sent over the newly created socket. Each XINU communication line requires two sockets between client and server. The information used in the communication line initialization sequence includes the process id of the requesting machine (for signals), the id of the requesting machine (for routing), and the id of the destination machine (for routing). After the destination machine has connected, communication can commence.

The central server dispatches source client send requests, at which time the server reads the byte, interrupts the source client, sends the byte, and interrupts the destination client. These interrupts correspond to interrupt on output buffer empty and interrupt on input buffer full respectively. With this facility, the simulated hardware and the interrupt handler can be plugged in without affecting the higher network layers. The interface to the higher levels was still undergoing some changes at the time of this writing.

Through the central server, all sorts of test conditions can be simulated, including noisy lines and dead machines. With a simple random number generator, noise can be probabilistically entered in the data stream and machines can brought down.

Because of the complexity of a network system, we feel that an interactive central server with a debugging system would be desirable. The interactive server would be run in the foreground allowing user control over the network through interactive commands. For example, this facility would provide the capability of the adding noise to lines and bringing down machines deterministically instead of probabilistically and tracing the activity on the connections.

5. THE USER INTERFACE

Since XINU is written in C, the XINU virtual machine can be easily modified by changing the source code and recompiling it using the standard UNIX C compiler. Our implementation is strictly compatible with all the existing XINU system calls with the single exception of the command creating semaphores, *screate*(), which now accepts a second argument specifying a name for the semaphore being created. This naming allows easy semaphore tracing and process table printing.

The original version of XINU did not allow the loading of user code without recompiling the whole system. We enhanced XINU by adding a shell and an interactive tracing facility. Our new XINU shell allows the user to:

- directly load executable code without recompiling the whole system,
- display the status of all XINU processes,
- issue interactive system commands such as changing the priority of a process or killing it, and
- issue keyboard interrupts to kill or suspend the current process.

The tracing facility allows XINU users to trace preemptions, context switches and operations on semaphores. It can be called from the XINU shell or from within the program being traced.

6. XINU AS AN EDUCATIONAL TOOL

We wanted to give students homework assignments that would require them to understand something about the working of an operating system kernel. The first such assignment we gave consisted of having students modify the process scheduler to distinguish between I/O-bound and CPU-bound processes. We found that almost all of the facilities necessary to do this are already available in the XINU kernel code, and that the changes necessary were localized to a few procedures.

To teach network communications concepts, we provide each user with a number of XINU virtual machines, each of which is a UNIX process. These processes communicate with each other through sockets that simulate RS-232 ports. Hazards such as noisy lines and dead machines along the data path are simulated, so students get a realistic view of the requirements of communication software at the data link layer and at the network layer.

We have found that students tend to develop a much better grasp of crucial ideas when they have had some hands-on experience with system code.

7. CONCLUSION

We have presented a XINU virtual machine running on VAX architectures under Berkeley 4.2 UNIX. The key features of our implementation are

- a C language interface with user programs, allowing direct execution of all XINU system and user code,
- an improved user interface with an interactive shell and a debugging facility, and
- considerably enhanced networking facilities providing for the realistic simulation of a wide range of point-to-point network topologies.

An initial implementation of our XINU virtual machine has been running since January 1985. It has been used to teach operating systems concepts to more than one hundred and fifty students, without significant problems.

The major limitation of our design is its inability to implement in XINU any memory management scheme that would require the trapping of invalid address exceptions. Future work should include:

- better monitoring functions facilitating the evaluation of the impact on system performance of any modification of the existing XINU policies,
- the port of our virtual machine to different host architectures and different UNIX environments, and

- a redesigned kernel that would allow the simulation of more
- sophisticated memory management schemes, including virtual memory.

A more radical departure to the existing XINU philosophy would be to use XINU to support a simple concurrent programming language like Occam.

REFERENCES

[Baba83a] Babaoglu, O. and F. Schneider, "Documentation for the CHIP Computer System," Technical Report TR 83-584, Department of Computer Science, Cornell University, Ithaca, NY, 1983.

[Baba83b] Babaoglu, O. and F. Schneider, "The HOCA Operating System Specifications," Technical Report TR 83-585, Department of Computer Science, Cornell University, Ithaca, NY, 1983.

[Come84] Comer, D. *Operating System Design: The XINU Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Fabr83] Fabry, R. S. et al. "The TOY Operating System," CS 162 Notes Vol. I, Department of EECS—Computer Sciences Division, University of California, Berkeley, CA.