Learning Operating Systems Structure and Implementation through the MPS Computer System Simulator

Mauro Morsiani
National Institute for Nuclear Physics, Ferrara Section
Via Paradiso, 12
44100 Ferrara (FE) - ITALY
+39 0532 291900
morsiani@fe.infn.it

Renzo Davoli
University of Bologna, Dept. of Computer Science
Mura Anteo Zamboni 7
40127 Bologna (BO) - ITALY
+39 051 354501
renzo@cs.unibo.it

1. ABSTRACT

Lab activity is fundamental for the real understanding of several computer science topics such as operating systems. We have built our own hardware emulator after using software tools from other Universities for several years. MPS is a general-purpose computer system simulator based on MIPS R3000 processor. Together with the main processor, RAM, ROM, disks, tapes, printer and terminal interfaces are carefully emulated and fully configurable; non-volatile memory units may be retained between simulations.

MPS features a full-fledged graphic user interface running under X Window, complete sources and documentation. Along with it we present TINA, an experimental project on operating system development, together with several other project proposals.

1.1 Keywords

Simulation, lab activity, operating systems, computer architecture, MIPS.

2. INTRODUCTION

The teaching of computer science courses at the undergraduate level often requires the development of suitable lab activities. This is especially true for courses on computer architecture and operating systems, where the complex interactions between the software and the underlying hardware may be understood in a better way by putting theory into practice.

Unfortunately, the sheer speed and complexity of modern computer systems make them really hard to understand by the average student. Thus, a number of computer simulators, together with suitable project proposals, have been developed for lab practice: they allow the main features of a computer system to be shown without meddling with the most intricate details, and let the students have a better control and understanding of the events which happen inside it.

This paper is organized as follows: chapter 3 summarizes the teaching methods used in our operating systems courses, chapter 4 is a comparison with similar works found in literature, chapter 5 and 6 present the MPS emulated architecture and user interface, and chapter 7 is a overview of the TINA project specifications. Some final remarks about other projects, future extensions to MPS, acknowledgments and bibliography conclude the paper.

3. PRACTICUM IN OPERATING SYSTEMS

The course named "Practicum in Operating Systems" (CS415) has been part of the Computer Science undergraduate curriculum at the Cornell University for more than ten years. Computer Science in Bologna decided to create a two-semester course (about 80 hours of classroom work, plus lab activity) named "Laboratorio di Informatica II" (CS Lab2) as an operating system lab course when the undergraduate curriculum was modified in 1994.

One of us (Davoli) has been the responsible of the course since its creation, and he has previous experience in organizing lab activity for standard operating system courses. The other (Morsiani) developed the MPS simulator code as his graduation project under Davoli's supervision.

Here are some relevant issues about the organization of a course on practicum in operating systems.

Supervised or unsupervised lab activity. Each part of the project may be implemented at the same time by each student (or group) under the direct supervision of the teacher or it can be realized as an assignment, with a submit date. We preferred the latter, as it is allows to each student to work at his/her speed, using his/her design skill and implementation technique. The use of widespread operating environments, such as Linux, allows students to use their own PC to work on the project even outside the University

SIGCSE '99 3/99 New Orleans, LA, USA

lab facilities. This way, part-time students may take part to the course, as well as students that need more time to complete the assignment.

Personal or group activity. The lab work could be carried out by each single student by him/herself, or students could be divided into working groups. We took the latter choice, for several reasons: the assignment may be more complex, interesting and complete; the students learn from practice how to deal with large projects, how to share the work load, test incomplete programs and join partial implementation into a complete result.

A single exercise for the whole course vs. different exercises, one for each student/group, which may sum up into a large project. The latter would make each student work only on some details of the whole; so we have chosen the former. This choice also simplifies the teacher's work as all students deal with the same problems at the same time. Some form of cooperation between different groups is also possible and we think it can be useful. The point is that some competition have to be held between groups, and brute plagiarism should be avoided by carefully analyzing the submitted works during the evaluation. We used a local newsgroup as an asynchronous communication channel for consulting both for the teacher and for the colleagues.

Single term submission or mid-term partial goals. The latter is certainly better. Students may fix structural defects of their work without compromising their final evaluation. The teacher also gets a better view about common problems. We tried one to three mid-term submissions in different years. Actually we use a single mid-term, to avoid interferences with other courses and because the students need to learn both how to use the emulator and its tools, and how to design concurrent O.S. code. The mid-term allows the students to split this in two phases: the first one is typically an implementation of data structures to be used in the second phase, which is the main exercise. When the students start solving the second phase they already got acquainted with the emulator and its tools.

4. RELATED WORKS

CHIP [1], a PDP/11 simulator developed at the Cornell University, together with its accompanying HOCA project [2], has been employed for many years in the past at the University of Bologna. CHIP provided the students with a realistic (but outdated by now) architecture, with tape and disk units, terminals and printers. The main lack of the CHIP environment was its usability. Its interface was a full-screen text one until 1996. Users interacted through a quite cryptic command language. The cross-compiler was realized from the UNIX V7 C compiler [9], and thus the syntax used was neither object oriented nor even ANSI.

Other examples of hardware emulators used for educational purposes include SPIM [7] and Nachos [4]. Both these emulators refer to the architecture of MIPS processors. SPIM has been developed as a tool for the study of microprocessor architectures. It emulates a system having only RAM, ROM and a single terminal. All the support for virtual memory, as well as pipelining and delay slots, has not been implemented. These characteristics make SPIM unsuitable as a tool to teach operating systems.

Nachos, on the other hand, has been expressly created for teaching operating systems. It implements the virtual memory support for MIPS and features a terminal, a disk and a network interface. The main lack of Nachos is in its software architecture: it is not a self-contained emulator, but a set of C++ objects containing both the MIPS emulator and an experimental operating system. From a teaching point of view it does not give a clean perception of the software/hardware interface. The kernel is linked together with the emulator and thus it is loaded as a software module for the hosting machine (i.e. on non-MIPS machines, the kernel executable will not be MIPS machine code).

We examined these projects carefully; in fact, some of the MPS simulator features were directly inspired by them. Our main goal, in developing MPS, were to obtain a clean, up-to-date, and easy-to-understand computer architecture. It should be controlled by an intuitive interface, which should feature powerful debugging tools, and be flexible enough to make the simulator easy to use and suitable for a broad range of experimental projects.

5. THE MPS ARCHITECTURE

The MPS computer system architecture is a classical one: one microprocessor, RAM, ROM and memory-mapped device controllers straightly connected by one system bus. Many of the simulator characteristics were defined along with the architectural design process, so they are reported here.

5.1 MPS Microprocessor

The MPS main processor is an almost-perfect emulation of the MIPS R3000 RISC integer microprocessor unit. It has been chosen for several reasons:

- the R3000 features many of the most interesting characteristics of modern processors: large address space and word size (32-bit), reduced instruction set, load/store pipelined architecture, load and branch delay slots, kernel/user modes, interrupt and trap handling, etc., and still it is not too complex for students to understand;
- ease of simulation, due to its RISC nature;
- availability of some detailed computer literature, both technical [6] and for teaching purposes [8], and of a cross-compiler kit, based on the popular *gcc* compiler;
- other similar projects ([7], [4]) have made a successful use of it.

By simulating the R3000 to the slightest detail (thanks to the detailed documentation) we may use the cross-compiler

kit Straight **out** Of the box; the availability of a specific textbook is also a great help in organizing teaching courses.

5.2 RAM and ROM

The amount of installed RAM in the simulated system may be changed, up to 2 MB or more if necessary.

Bootstrap and BIOS ROMs may be easily written to suit specific needs using the assembler provided with the cross-compiler kit, and then loaded inside the simulator. **BIOS** may contain support routines for normal operation, thread-safe code for Critical sections useful in multi-processing kernel development, etc.

Three bootstrap methods are available, using or developing suitable bootstrap ROMs: "core boot", tape, and disk boot. This allows the simulation to faithfully mimic the bootstrap of a real machine (reading the start-up code from disk or tape, under the control of bootstrap ROM code), or allowing a quicker boot (the "core boot", which implies that the start-up code is already present in memory).

5.3 I/O System

The R3000 processor specifications do not detail I/O structure; they just make available up to six external interrupt lines, with appropriate interrupt handling mechanisms.

We reserved the highest priority interrupt line to the system timer, and assigned other lines to other devices (from highest priority to lowest one): disk drives, tape drives, network interfaces (still under development), printer interfaces and terminal interfaces.

Each device is controlled by a specific device register; up to 32 devices are available for each interrupt line. All the device registers are arrayed in a memory area, together with other system configuration information (RAM address range, system clock and timer registers, etc.)

We decided to build &vice registers with a standardized structure and operation codes (one word-sized field each for status and command, and two more word-sized fields for parameters). An interrupt-driven full-handshake sequence is required to perform a successful I/O operation. Each device may show transient hardware failures, upon user selection, to test software fault-tolerance.

Each simulated device performance (that is, the time required for an I/O operation, viewed from inside the simulation), is consistent with the real performance of actual devices, and may be easily modified to suit specific needs.

The processor clock itself may be set in a wide range (currently, 1 to 100 MHz): this forces the simulator to scale the I/O operation times accordingly, and thus allows to experiment with different trade-offs between processor and I/O performance with a simple and consistent operation, without worrying about changing all the device performance figures, or moving them out of realistic ranges.

5.3.1 Disk Devices

Simulation of disk devices is the most accurate, since many disk management algorithms may be good candidates for practice. Thus, disk geometry and performance are fully configurable: it is possible to specify the number of cylinders, heads, sectors, the rotational speed, the inter-sector gap and track-to-track seek time; all these figures are used to compute the time required by the I/O operation.

The simulator saves on a separate file the actual contents for each simulated disk, so they may be kept between sessions.

Each disk sector contains 512 bytes, and may be addressed by a ***<cylinder, head, sector>*** triple; typical operations are cylinder seek, and sector read and write with emulated DMA transfer. Any file system implementation is possible.

5.3.2 Tape Devices

Simulation of tape devices has been included to allow the user to load external files (program objects and data) inside the simulator. These files are assembled **together** in a file, which represents a tape cartridge; this tape may be "loaded" and "unloaded" by the user into the tape drive, and read from inside the simulation.

Tapes may be kept between sessions; each tape may be read, one 512-byte block at a time, by the tape drive, with DMA transfer; the entire tape may be fully rewound or scanned, looking for block and file markers.

5.3.3 Printer Devices

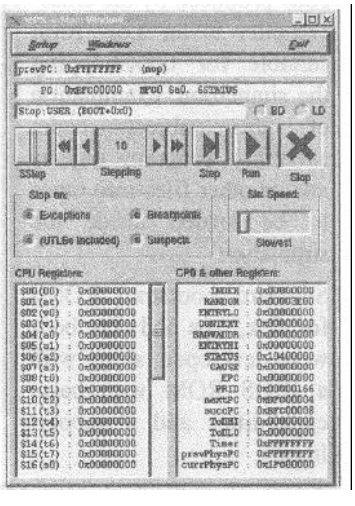
Printer interfaces are emulated as simple character devices: the simulator dumps their output in external files (one for each active printer).

5.3.4 Terminal Devices

Terminal interfaces too are emulated as simple receiver/transmitter character devices: their **output** is put both in external files (like printers), and shown inside a simulator window, while their input may be set interactively by the user, one line at a time, during the simulation. This allows simulating the user-machine interaction of a real computer system

6. THE MPS SIMULATOR

6.1 Graphical User Interface



The simulator GUI runs under X Window System, and comprises many windows: the most prominent is the main one (Figure 1), that shows most of the simulator status (processor registers and instructions), and allows the control of the

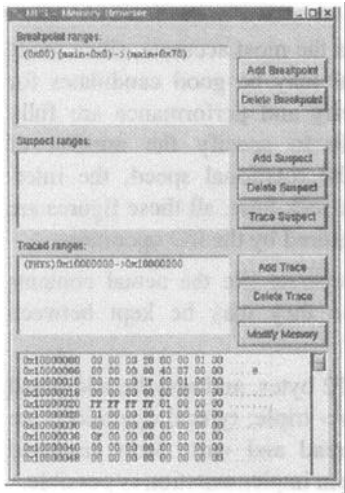


Figure 2

simulation advancement; buttons are provided to advance the simulation for a predefined number of instructions, and/or to make it run at various speeds to reach a particular point quickly or I slowly step through the code.

Processor status, registers, system clock and- timer may be changed just clicking on the appropriate item.Other buttons allow the user to globally activate or deactivate the monitoring of program traps, suspects (variables under examination) and breakpoints in the code; if active, the simulation stops upon reaching such a point. Code position is shown both with address and function name plus offset, when available. The symbol table may be loaded into the simulator together with the program itself, symbols may be used to define breakpoints and suspect variables, using the memory browser window (Figure 2).

Other windows (like the one in Figure 4) allow monitoring the device status, and simulating hardware failures; finally, a setup window (Figure 3) controls device installation and the configuration of system parameters and ROM files.

6.2 Software Development Kit

Companion of the graphical interface, is the software development kit for writing C and assembly programs for the simulator.

It includes the already-mentioned *gcc* cross-compiler kit, together with some utilities written specifically for MPS: they allow the conversion of the ELF format object code produced by the cross-compiler to an a.out form suitable for bootstrapping the simulation; examination of this object code; assembly program objects and/or other files into tapes to be loaded inside the simulation.

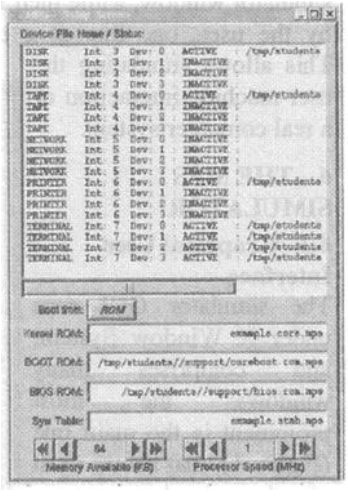


Figure 3

Standard bootstrap ROMs for disk, tape and core boot were developed, along with a BIOS ROM with basic capabilities and a support library.

A user manual, describing MPS architecture and features, completes the development kit.

6.3 Development Notes

The entire MPS simulator software has been developed using C++ language in a Linux environment on Intel platforms. A preliminary project hypothesis considered Java use, but it was discarded, mainly for performance reasons. Special care has been taken to ensure that the simulator could run at fast speeds, to shorten the debug cycle of applications written for it.

The entire project has been ported without effort under Solaris on SUN SPARC platforms; this mainly because the entire project was developed keeping well in mind the little/big endianness issue. The MIPS processor, in fact, may operate as a big-endian processor or a little-endian one; the cross-compiler kit reflects this, offering a version for both. But Intel processors are little endian, and SPARC ones am big-endian; so, the code was carefully written to adopt the endianness of the underlying processor.

This way, the well-known endianness conversion issue ([4], [10]) was avoided altogether, and it ensures the portability of this project to any environment. The price for this is just the use of the correct cross-compiler kit version when compiling.

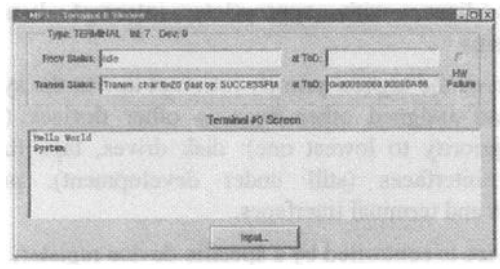


Figure 4

7. THE TINA PROJECT

Companion to the MPS simulator is the TINA project on experimental kernel development It has been used as lab project assignment for students of CS Lab2 course on operating systems at University of Bologna in 1998.

TINA project specifications were taken from those for HOCA operating system, the companion to CHIP, just adapting it to the new simulator and shortening it to two phases from the original three.

The first phase involves writing two modules that ate later used in the nucleus to implement the process queue and the semaphore abstractions. The first module is a process queues management module, while the second is an active semaphore list module; both require management of linked lists and pointers, but no knowledge about concurrent programming. The students test their ability to code in C language (which is introduced during the course) and get accustomed with the simulator.

The second phase involves building the routines that implement the notion of asynchronous sequential threads and processes, a pseudo-clock and the synchronization primitives.

TINA has a monolithic kernel, where system calls execute as reentrant privileged code in the time slices of the requesting processes. Very few system calls ought to be implemented: process creation and termination, P and V on general semaphores, I/O wait, pseudo-clock tick wait, and pass-up of traps to handlers defined by each process; this allows the creation of further layers of system management, as in THE system [5].

Both phases were provided with alpha test modules to allow code testing before submission; support libraries, BIOS and documentation were handed down to students. The results we obtained were highly encouraging: the starting group number was 51, 35 submitted the first phase, and 32 the second one. 8 completed kernels were working perfectly, and another 8 had only minor inconsistencies. Over 300 messages were exchanged in the newsgroup.

8. CONCLUSIONS AND FUTURE DEVELOPMENTS

MPS will be used as the main development tool of our course for the next years. Here is a set of possible exercises:

Support layer for TINA. This can be a third phase of the TINA project or a self-contained exercise, maybe combined with some modifications of the kernel to force the students to investigate its structure. A support level may provide the system with some device drivers, virtual memory and protection from user processes misbehavior.

Microkernel O.S. TINA specifications may be modified with a microkernel design [3]. Instead of having several system calls, only two are defined (send and receive): the kernel itself may be designed to translate each hardware signal (interrupt or trap) into a message to a specific management thread.

Shell, file system and specialized drivers (may be seen as a fourth phase in TINA development). The file system could be FAT-based with primitives to create and delete files, and read/write their fixed-size blocks sequentially. Seek minimization algorithms, like the elevator algorithm, could be added to the basic disk driver. Spooling facilities could also be implemented to build either a print spooler or a spool for batch executions. This latter exercise may also include modifications to kernel scheduler to support priority management.

We feel that MPS will allow each one of them, and much more; its features make it suitable for many lab projects, from computer architecture and assembly language programming, to operating systems development, resource management schemes, and real-time algorithms testing.

The development of a network interface card device, which will allow building entire networks of MPS machines, is at an advanced stage, and a multi-processor version of MPS is going to be released, too. New and more specialized projects about operating systems will then be possible; we speculate that these new features will broad the range of experimental projects to inter-machine communication, parallel and distributed computing issues.

All software and documentation related to the MPS project is freely available on the World Wide Web at URL: *http://www.cs.unibo.it/~morsiani*. We will be glad to share our experience with other teachers and instructors that would like to use MPS.

9. ACKNOWLEDGMENTS

Our thanks go to Ozalp Babaoglu, which brought here at Bologna the original CHIP and HOCA projects from Cornell; to James T. Larus for SPIM and to Nachos development team for their projects, which gave so many good ideas to us; to Michael Riepe for its freeware ELF manipulation library, and to T.C. Zhao and Mark Overmars, for their wonderful freeware X FORMS library.

Finally, our greatest thanks go to the over 200 students which served as testers for our simulator, finding the bugs, and giving good suggestions.

10. REFERENCES

[1] Babaoglu, O., *et al.* Documentation for the CHIP Computer System. Dept. of Computer Science, Cornell University, Ithaca NY, 1988. *http://www.cs.utexas.edu/users/lorenzo/corsi/cs372/97F/project.html*

[2] Babaoglu, O., Schneider, F.B. The HOCA Operating System Specifications, Dept. of Computer Science, Cornell University, Ithaca NY, 1988. *http://www.cs.utexas.edu/users/lorenzo/corsi/cs372/97F/project.html*

[3] Baron, R.V. *et al.* MACH Kernel Interface Manual, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh PA, 1989.

[4] Christopher, W.A., Procter, S.J., Anderson, T.E. The Nachos Instructional Operating System. Computer Science Division, University of California, Berkeley CA, 1992. *http://http.cs.berkeley.edu/~tea/nachos*

[5] Dijkstra, E.W. The Structure of THE Multiprogramming System, Commun. ACM 11, 5 (pp. 341-346).

[6] Kane G., Heinrich J. MIPS RISC Architecture. Prentice-Hall, Englewood Cliffs NJ, 1992.

[7] Larus, J.T. SPIM S20: A MIPS R2000 Simulator. Computer Sciences Dept. University of Wisconsin, Madison WI, 1990. *ftp://ftp.cs.wisc.edu/tech-reports/reports*

[8] Patterson, D.A., Hennessy, J.L. Computer Organization & Design: The Hardware/Software Interface. Morgan Kaufmann Publ., San Mateo CA, 1994.

[9] Ritchie, D.M. A Tour Through The UNIX C Compiler, AT&T Bell Laboratories, Murray Hill NJ, 1979.

[10] Tanenbaum, A.S. Structured Computer Organization (III ed.). Prentice-Hall, Englewood Cliffs NJ, 1990.