A SIMULATION PROJECT FOR AN OPERATING SYSTEMS COURSE

John K. Estell
The University of Toledo

1. INTRODUCTION

One of the dilemmas facing instructors of introductory operating systems courses is the development of suitable programming assignments. It is desirable to expose undergraduate students to realistic projects that allow them to apply the theoretical concepts learned in lecture; however, great care must be taken not to overwhelm the student enrolled in this course. Several methodologies have been discussed in the literature over the years, one of which is the use of simulation. Various papers have described the use of generic simulators being provided as a framework, writing a multitasking operating system in assembly language for use on a IBM Virtual Machine[2], developing an operating system for a provided virtual hardware simulator and developing the simulation of the machine along with the operating system[4]. One popular simulation package is the Nachos system[5], where one is provided with the code for a working instructional operating system along with a machine simulator. The use of simulated machines allows for the avoidance of such potential problems as architectural idiosyncrasies and programming in a complex and unfamiliar language.

Another dilemma is dealing with the abstract nature of a process. Students initially look at a process as being the same as a program, for that is how it appears to them as a user when they are learning how to program: same thing, different names. Now that they have to deal with the internals of a multitasking operating system, students are left trying to cope with theoretical concepts that are not well defined. Additionally, the dynamic and concurrent nature of processes on a multitasking system departs from the sequential concepts they have been exposed to in their previous studies. In order to grasp an understanding of the nature of a process, students need to work with processes, and the greater the immersion, the greater will be their comprehension.

This paper presents a group project used as a programming assignment for groups of three to four students in a junior-level introductory operating systems course. The quarter-long assignment, written in C on a UNIX platform, was broken down into four phases: the initial simulation of the specified computer architecture and accompanying machine language with the incorporation of a process management model, the development of process scheduling routines for handling the simultaneous execution of multiple processes, the development of performance evaluation programs to filter data generated by the simulator, and an extension to the simulator initiated and designed by each group. Test programs modelling various types of processes were developed and executed using the simulation and evaluation programs. The results were used to examine the performance of several process scheduling algorithms under a variety of conditions. Through the development and implementation of this project, the students gained experience with processes, obtained insight into the nature of an operating system, learned about simulation methods and furthered their understanding of how computers operate at the machine instruction level.

2. THE LS COMPUTER SYSTEM

In order to work with actual processes a hypothetical computer system, complete with machine language, was developed. As the primary focus was to work with processes and not to explore the nuances of computer architecture, the specifications for the system were kept as simple as possible. The system, called LS, consists of an 8-bit accumulator register, a 16-bit indirect (X) register, a 16-bit program counter (PC), a 16-bit stack pointer (SP), and a status register. The status register contains three 1-bit flags: the zero flag, the carry flag, and the negative flag. All arithmetic and logical operations will place results in the accumulator and modify the appropriate status flags. The system supports three addressing modes: immediate, where the data is contained with the instruction; direct, where a PC-relative offset refers to the desired memory location; and indirect, where the contents of the X register refer to the desired memory location.

A minimal instruction set was specified in order to keep the implementation of the simulator as simple as possible. This information was presented to the class in terms of specific machine codes for each allowed combination of operation and addressing mode. For each machine code, a value is specified for the number of clock cycles it would take to fetch and execute that particular instruction; these values are needed to update the master clock used for keeping track of time in the simulation. The instructions break down into several mostly recognizable categories, such as load and store register contents, arithmetic and logical operations on the accumulator, comparison and branch, and subroutines. Port-based I/O instructions are used for flexibility in that no new instruction is needed to add a new I/O feature. All that is required is to assign a port number to the new operation and write the supporting simulation code. The fork instruction is not commonly found in an instruction set; however, it is suitable here as it makes the action of forking a process atomic, greatly simplifying the design.

In order to allow groups to test their simulators an assembler program for the LS mnemonics was needed; otherwise, object code would have to be hand-compiled. As this course had a honors section associated with it which required extra work to be performed, it was decided that writing a simple assembler would constitute the honors students' project. The specifications called for a minimal number of assembler directives: "byte" specifies data to be stored, ".blkb" reserves memory space, "start" indicates the start of the executable portion of the source file, and "stop" indicates the end of assembly. To simplify matters, a label format consisting of a letter followed by a digit was adopted. This assignment was given to the honors students at the beginning of the quarter. Its completion was scheduled such that a working assembler program would be available to the groups for at least one week prior to the deadline for the first phase of the simulation program.

The object file generated by the assembler consists of ASCII hexadecimal characters, with each line containing two characters to form one machine code. This format was chosen so that, if the need arose, object files could be easily sight-verified. The LS system operates on a relative addressing scheme; therefore, addressing begins at relative location 0000. The first word in the object file gives the offset, in low byte - high byte order, to the first executable machine code in the file. This information is used by the LS loader to properly adjust the PC for beginning the execution of the process created for this program. The remaining bytes are the data and machine codes that make up the program.

3. PHASE ONE: SIMULATOR AND PROCESS MANAGEMENT

The first phase of the project was to implement the basic system so that a single process could be successfully executed on the simulator. Each student was given the specification for the LS computer system. Time was given in class for the groups to assemble and discuss design criteria under the general supervision of the instructor. This phase was the most difficult for the students to accomplish as an understanding of all aspects of the material was needed in order for the system to be functional.

The five-state model[6] was used for process management. The New state handles just-created processes, It initializes the data structures for a new process, after which it enters the process into the ready queue. The **Ready** state contains the ready queue, which lists those processes that are ready to use the processor. The **Running** state deals with the one currently executing process. A process is placed into this state by a scheduling algorithm that determines which process in the ready queue will be given access to the processor. A process continues operating in this state until either an 1/0 or event wait is encountered or a timer interrupt is received. The **Waiting** state contains those processes blocked by such things as an I/O request. Once the event that is being waited on has occurred then the process associated with the event will be placed into the ready queue. Finally, the **Terminated** state deals with the exiting of a process. The data structures associated with the process are properly disposed of and data regarding the execution of the process is collected. In order to use this model, a process control block must be created for each process on the system. This block will store such data as the register contents, the process identification number (PID), memory location, queue pointers, and statistical data. The information contained within the process control block is used by all of the process management states.

A process is executed by specifying its program object file as a command line parameter when invoking the simulator. A process control block and a PID are created for the process, the program's object codes are stored in the simulator's memory, and memory is allocated for the process stack. The dispatcher begins the execution by performing a context switch, giving the processor to the process. The dispatcher is a system-owned process that executes the scheduling algorithm that controls the use of the processor. The basic simulation uses the first-come, first-served (FCFS) scheduling algorithm. The process has control of the processor until it either terminates or is blocked by an 1/0 request. When a process is blocked, it is assumed that a fixed number of clock cycles will elapse before the request is serviced and the process is unblocked. The output from the simulator is a trace in column form of the activity during the simulation. In order, the output gives on each line the cumulative clock cycle time when the event occurred, the PID of the affected process, and the reason for the context switch. A PID of zero indicates the dispatcher process.

4. PHASE TWO: MULTITASKING AND PROCESS SCHEDULING

The next phase of the project involved the modification and refinement of the LS computer system simulation. The main goals here were to implement multitasking and various process scheduling algorithms. Multitasking was reasonably straightforward, as most of the necessary elements were in place. The object files of all of the processes to be executed during a simulation are specified on the command line; they enter the New state in order of appearance on the command line. The command line is also used to explicitly specify a scheduling algorithm. In the first phase of the project the FCFS algorithm was implemented. Under FCFS, the ready queue is designed as a simple FIFO (first-in, first-out) queue. When the process in the **Running** state either finishes execution or is blocked, the FCFS algorithm will select the process at the head of the ready queue for execution. Because of the way that the FCFS algorithm was implemented in the first phase of the project, it was already available for use in a multitasking environment.

Two scheduling algorithms were added to the simulation: round robin (RR), and shortest remaining time (SRT). The RR algorithm is similar to the FCFS algorithm, but with the inclusion of preemption to allow for switching between processes. Preemption is based upon allowing a process to execute in the **Running** state only up to a specified maximum amount of time referred to as a time slice. When a process is preempted, it is placed at the rear of the ready queue, and the next process to run is taken from the head of the ready queue. For the simulation, the time slice was specified as a passed parameter expressing the time slice in terms of number of clock cycles. By allowing the students to set the time slice value at run time, one sets the stage for experimentation into the effect of varying time slices to be conducted later on in the course. The SRT algorithm gives the students some experience with algorithms that use the history of previous accesses to the processor to determine which process should next be scheduled for execution. For this algorithm, one needs to keep a history of the amount of processor time used by each process for each of the last few previous visits in the **Running** state. Each of these times are multiplied by a weighted value biased towards the more recent visits, then are summed together to form an estimation as to the amount of time the process will spend in its next visit to the **Running** state. The SRT scheduling algorithm will select the process with the shortest estimated time for execution. The SRT algorithm is similar to the RR algorithm in that it also uses a time slice preemption mechanism; however, it differs in that it is possible for a process with a sufficiently high time estimate to encounter starvation, where it does not have the opportunity to access the processor because of other processes having shorter estimated times always being present in the ready queue.

For all scheduling algorithms, the estimated amount of clock cycles it would take for the dispatcher to perform the appropriate operations is calculated. While the dispatcher could be implemented as an actual process executed by the simulator, it was decided that it would be better to allow the students to perform all of the necessary dispatcher functions in the high-level language used in writing the simulation. In this way the students still get the necessary experience of working with such items as process control blocks and ready queues without having to deal with the tedious nature of working with these structures in a primitive assembly language. The estimated value for the dispatcher operation is used to update the system time during simulation for each invocation of the dispatcher.

The more complex LS machine codes were implemented during this stage; this includes FRK, for forking a process, and the specification of port numbers for handling both terminal I/O and interprocess communication (IPC). The fork instruction is used to spawn a child process. The child process is created by making a new process control block based on the current state of the parent process and by allocating memory into which will be copied an image of the parent process. Upon return from the fork instruction, both the parent and the child processes will have the PID of the related process located in the X register. The carry flag is used to differentiate between parent and child; the parent has the carry flag set to one, and the child has the carry flag set to zero. This design facilitates the use of interprocess communication in that, upon creation of a child process, the child knows the PID of the parent and the parent knows the PID of the child. Message passing between two processes is performed by having a receiving buffer for each process. Semaphores are used to control the reading and writing of data to this buffer; this is essentially an implementation of the bounded-buffer producer/consumer problem[6]. When an IPC write operation is performed, the transmitted data is place into the message queue of the target process. If the message buffer is full, then the transmitting process is blocked until an entry is read by the target process. When an IPC read operation is performed, the process accesses its message queue to search for data. If the message buffer is empty then the process is blocked until data is received from the transmitting process. As multiple processes are now sharing memory, a bounds checking mechanism was incorporated into the simulation in this development phase to provide process integrity security. This was done to prevent a process from performing a scan of the address space outside of its allocated area in an attempt to obtain data from, or to otherwise interfere with, another process.

5. PHASE THREE: PROCESS SCHEDULING ALGORITHM EVALUATION

The third phase of the project dealt with the testing and performance of the simulation. Each group had to write LS assembly language programs that would serve as characteristic examples of various types of processes. Several CPU-bound and I/O-bound processes with specified execution times were developed, as well as processes that would implement interprocess communication and attempt to examine the entire contents of memory. A filter program was also written to take the output from the simulator, analyze it, then display the results in a tabular format. The filter program provides information regarding the various scheduling criteria used when analyzing algorithm performance. One of the key criteria is CPU utilization. As the CPU is a scarce resource, it needs to be kept as busy as possible executing user programs. Because of the nature of the simulation, the CPU utilization is measured not over time but only for the life-span of the processes used in the simulation. This still provides informative data for comparing the difference in system performance between algorithms. As an example, one of the evaluations required the students to run the simulation using the RR algorithm with various time slices for the same set of processes. As smaller time slices will increase the number of context switches, the students can observe from their data that the CPU utilization suffers when the time slice is too small and how other scheduling criteria are adversely affected when the time slice is too large.

The filter output also gives the values for arrival time, amount of service time, the start and finish times, the regular and normalized turnaround times, the waiting time, and the throughput, which is the number of processes completed per time unit. The ideal scheduling algorithm would maximize the CPU utilization and throughput to make the system as efficient as possible. Additionally, this algorithm would minimize the waiting, turnaround, and response times for the convenience of the user. From their experiments using various scheduling algorithms and process mixes, students learn that when one criteria is optimized, it is at the expense of another. A report was submitted by each group along with their test data for the various processes and scheduling algorithms investigated.

6. **PHASE FOUR: STUDENT EXPLORATION**

The final phase of the project allowed groups to further develop their simulator. Most groups used the curses text windowing package to provide for a visual interface if desired by the user; some included a disassembler that allows for single stepping of the instructions in memory. One group even implemented a simple virtual memory system. Some exploration was performed, but not implemented, in the area of files. This involved the discussion of establishing a directory and file format, and of setting up a disk simulation program that would run as a separate process. Through use of the UNIX socket mechanism, the LS simulator would make requests of the disk simulator. If this course was taught under the semester system instead of the quarter system, there would be sufficient time to extend this project further. Areas such as virtual memory, files and disk I/0, and multiple users could be fully explored. It would also be possible to set up sockets to allow connections between simulators, forming a network that could be used for exploring remote procedure calls.

7. **RESULTS**

Student comments obtained from both discussions and anonymous evaluations indicate that this project was a positive experience. By developing the simulation students gained a greater understanding of both the amount of work and the complexity involved in developing an operating system. The LS computer simulation contained, on average, 3000 lines of code distributed over several modules. While this is minuscule compared to the size of a modern operating system, students still experienced the frustration of yet another bug cropping up during testing, and thereby appreciated the comment made at the beginning of the quarter that operating systems are never fully debugged. While only about a third of the groups in the class had everything working correctly, all of the remaining groups had most of their simulation working according to specifications. The portions not working were minor flaws that did not properly execute one or two test programs.

The project, by itself, allowed the students to learn first hand about such things as what information is necessary when performing a context switch, what things have to be done in order to fork a process, and how one can implement interprocess communication. The students also experienced having to develop test programs for the purpose of analyzing system performance. However, other lessons not directly related to the project were learned as well. For many students this was the first “real” program that they have worked on -

"real" in the sense that they were working in a group environment on a program that actually "did something". Students learned the importance of planning out of necessity, as they had to agree on how they as a group were going to implement the project before coding. The size of the project forced them to develop modular programming skills so that work could be performed by all team members simultaneously. As one student aptly put it, "If this would have been one large program, it would have been impossible - thank goodness for the makefile !" The complexity of the program and the need for working on code designed by others also helped to reinforce the importance of good documentation and a clean, readable programming style.

One of the keys to the success of this project was the development of clear and concise specifications that were distributed to the students at the beginning of the project. The foundation formed by the detailed description of the LS computer system along with assignment handouts spelling out what was to be implemented and what was to be accomplished for each phase of the development process provided a necessary framework for the students. However, they were not led by the hand and told how to implement. There are many ways to implement an operating system, and as each member of a group had his or her own ideas as to how to accomplish this, the design discussions that ensued among the group members enabled each student to obtain a greater understanding of the subject than what would have resulted by working alone.

In conclusion, this was an enlightening experience for both the students and the instructor. While it was not the original intent of the assignment, this project touched upon many separate threads discussed in previous courses. These threads were woven together into a cohesive whole -- a cloth that tied together many concepts, allowing them to see and appreciate why we previously spent all that time over "boring, useless" material. And as the threads that enter a loom are woven into patterns, so too were the concepts that the students used in designing their project woven into a fabric that gave them the opportunity to learn about processes and to further their knowledge of the architecture, organization, and operation of a computer system.

REFERENCES

1. M. Cartereau, "A Tool for Operating System Teaching," *SIGCSE Bulletin,* Vol. 26, No. 3, pp. 51-57, September 1994.
2. J. L. Donaldson, "Teaching Operating Systems in a Virtual Machine Environment," *SIGCSE Bulletin,* Vol. 19, No. 1, pp. 206-211, February 1987.
3. T. Hayashi, "An Operating Systems Programming Laboratory Course," *SIGCSE Bulletin,* Vol. 15, No. 1, pp. 31-35, February 1983.
4. C. Shub, "A Project for a Course in Operating Systems," *SIGCSE Bulletin*, Vol. 15, No. 1, pp. 25-30, February 1983.
5. A. Silberschatz and P. B. Galvin, *Operating System Concepts,* 4th edition, Addison-Wesley, Reading, MA, 1994.
6. W. Stallings, *Operating Systems,* 2nd edition, Prentice Hall, Englewood Cliffs, NJ, 1995.

JOHN K. ESTELL

Dr. Estell received his BS degree (summa cum laude) from The University of Toledo in 1984. Awarded NSF Graduate and Tau Beta Pi Fellowships, he received his MS and PhD degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1987 and 1991. Dr. Estell is a member of ACM, ASEE, IEEE, Eta Kappa Nu, Phi Kappa Phi, and Tau Beta Pi.