ENHANCING OPERATING SYSTEM COURSE USING A COMPREHENSIVE PROJECT: DECADES OF EXPERIENCE OUTLINED*

Onkar P. Sharma, Marist College, School of Computer Science and Mathematics

ABSTRACT

This paper describes the characteristics and implementation of a Multiprogramming Batch Operating System (MOS) project. An earlier paper [1] has described problems encountered by student groups during the first decade of its offering, and also solutions designed to solve those problems. This paper outlines recent changes introduced in the administration of the project to overcome residual problems. While characteristics of the MOS project described in the previous and this paper are basically the same, the methodology of administration presented in the two papers are completely different.

INTRODUCTION

Study of Operating Systems (OS) has been consistently emphasized in all the ACM curricula recommendations. For example, OS is one of the 14 areas listed under the CS body of knowledge in Computing Curricula 2001 [2]. The Overview Report of Computing Curricula 2004 [4] lists 36 computing topics. Two of these 36 topics, namely "Operating Systems Principle and Design" and "Operating System Configuration and Use", again pertain to OS. The weight assigned to "Operating Systems Principle and Design" topic is highest for the CS discipline. Moreover, this overview report states, "…we expect that a student will complete a major programming project, either creating an operating system "from scratch" or creating a significant enhancement to an existing operating system." The author is in total agreement with this statement and the project described here fits the recommendation of creating an OS from scratch.

The MOS project described here was originally included as an appendix in an OS textbook by Shaw [3]. However, this project does not appear in a later book on OS by the

---

same author and a co-author [5] in which they have included five smaller programming projects which are also well-designed but do not provide an integrative experience. The original project is very comprehensive and deals with important OS topics of "input-output, interrupt-handling, process synchronization, scheduling, main and auxiliary storage management, process and resource data structure, and system organization [3]." Thus this project does provide for design and implementation of an OS "from scratch." Since the project specification was developed in as far back as the 70s, it does talk about antiquated devices such as card reader as an input device and drum as an auxiliary storage. However, one of the positive aspects of the project is that replacing these devices by modern I/O and storage units changes only the specification and not the characteristics or implementation of the MOS. Several other OS projects can be found in books written specifically for OS classes [6,7].

PROJECT SPECIFICATION

The project envisages the design, coding and testing of a MOS for a hypothetical machine. The hardware platform (the "real" machine) consists of a CPU, card reader (which could actually be any on-line batch input device), printer, main memory, magnetic drum as secondary storage, 3 channels (I/O processors), and unlimited supervisory memory. Input comes from the card reader and output is sent to the printer. The drum consists of 100 40-byte tracks. Channel 1 performs input from the card reader into supervisor memory, and channel 2 performs the transfer of information from supervisor memory to the printer, both 40 bytes at a time. They both operate at the same speed. Channel 3, which is faster than the other two channels, transfers information between the supervisor memory and the drum, as well as between the drum and main memory of 300 words. Supervisor memory is an unlimited resource not managed by the MOS.

The "real" machine supports a simplified virtual machine which user programs run under. This machine consists of virtual memory, CPU, card reader and printer. The virtual memory consists of 100 four-byte words organized into 10 ten-word pages. The CPU is quite simple, accepting only seven machine instructions including input (GD) and output (PD) instructions. There are three CPU registers: a general purpose four-byte register R, a two-byte instruction counter IC, and a one-byte toggle register C, to hold the result of comparisons. Other instructions are LR, SR and CR for loading, storing, and comparing register R, all with respect to a memory location. The Result of CR is stored in C and used with instruction BT, Branch on True. Finally, the instruction H halts the computer.

The "real" machine switches between master and slave modes of operation. In master mode, it consumes no time (a simplifying assumption). MOS runs in this mode. User programs run in slave mode. Each user instruction consumes one time unit to execute, and occupies one word of storage. The first two bytes contain a mnemonic such as LR which specifies the instruction to be executed and the last two bytes contain the numeric virtual memory address between 00 and 99 of the instruction's operand. The first instruction of a user program is assumed to be loaded at virtual address 00.

The MOS is interrupt-driven. There are four types of interrupts. Program interrupts (PI) and Supervisor interrupts (SI) are generated by a users program. The former indicates an error condition such as memory protection exception or a page fault, while the latter indicates a supervisor call to perform I/O or terminate the user program. The third

interrupt type is an I/O interrupt (IOI), which is generated by a channel when its operation has completed. The fourth interrupt type is a Timer interrupt (TI), which is generated when a user program exceeds its time quantum (for time-sharing purposes) or has exceeded its total allowed run time. The latter two interrupts are hardware-related and are generated by the "real" machine.

MOS provides for input and output spooling. Input spooling is employed to transfer the entire job to the drum from the card reader before loading and execution (channels 1 and 3). Real I/O at execution time takes place from the drum to shorten the residency of a user program in main memory. After job termination, output spooling is used to send the program output to the printer from the drum (channels 3 and 2). Buffering is employed by I/O spooling to provide maximum concurrency and synchronization. Paging is done to manage virtual memory implementation with 30 frames and 10 pages.

This OS project incorporates several nice features, such as:

- I/O programming including the notions of interrupts, channels, buffering, and spooling
- Virtual memory including paging and address mapping
- Master and slave mode operations to distinguish between user and system processes, and privileged (GD, PD, H) versus non-privileged instructions
- Synchronization and communication between interacting MOS components
- Simulation of concurrent operation of CPU, channels and timers

IMPLEMENTATION: THEN

On the very first day of the classes, the project specification was distributed to students and teams of three or less students were formed. Class lectures on OS concepts such as memory management, processor management, device and file management, and presentation of the project proceeded concurrently. The project specification was followed by a detailed discussion of a possible design of the MOS. Since it was quite hard for students to understand how various features (processes) of the OS should be integrated into a properly executing operating system, this particular aspect of the project was repeatedly emphasized. The discussion on design always highlighted areas where different design choices were possible and students needed to make a decision.

The detailed presentation was completed in about six weeks of a 15-week semester. Students were then given one to two weeks to write a complete top-down hierarchical design in English using an algorithmic type of language. The design was required to be comprehensive and accurate, include all the data structures and simulations, and be in such a state of completion that coding could directly proceed from and totally reflect the design. The design was presented by each team around $7^{th}/8^{th}$ week. Where major revisions were needed, teams were asked to make changes and present it again in the following week. Coding was not permitted until the design was approved.

Each group was also required to submit two assigned user programs, which could run on MOS, for testing purposes. The assignments were carefully chosen to ensure that all different types of run-time errors did occur during the execution of the test programs. These user programs collectively constituted the test file for the entire class. With the

nature of output known in advance from each user program, this test file provided an excellent debugging tool for the teams, and also as an evaluative tool for the faculty.

In the week before the final examination week, teams made the final project presentation along with the following items:

- The OUTPUT from running of the test file
- The TRACE of the system's operation
- The SYSTEM and USER documentations

The OUTPUT indicated whether the OS executes correctly or not. The TRACE was a cycle-by-cycle portrayal of the run-time behavior of the MOS. It served as an excellent debugging tool for the student team, while at the same time it displayed for the faculty whether the coding followed the design specifications correctly or not. The SYSTEM and USER documentations need no description except that students were encouraged to develop them as the project progressed and not write them at the end.

Although the above approach had met with a reasonable level of success, there always were few teams which were not able to complete the project in time. Briefly, the two reasons determined as cause for noncompletion in time were late start in implementing the project (tenth week or so), and non-adherence to the software engineering principles in the development of an appropriate software architectural design. The solution employed had then consisted of presenting a possible object-oriented software architectural design to students early in the semester. This design consisted of 16 separately compiled modules which are listed in the paper [1] mentioned above. This changed approach, though very helpful, seemed to create another major problem. Students would now complete the project without a full comprehension of the intricacies involved. Students blindly followed the specified architectural design without analyzing and therefore, comprehending the logic behind the algorithms of the component modules. Their lack of understanding of the project became obvious during the final presentation. It became apparent that what processes are needed and how they are going to interact with each other to form the required OS remained elusive for many teams. It was hard for some of them to visualize and conceptualize the complete project design at one time.

IMPLEMENTATION: NOW

Three changes have now been introduced to improve the situation:

1. The main approach now followed is incremental implementation and step-wise refinement. The project is broken down into phases and though students are still presented with a complete specification of the MOS in the first couple of weeks, implementation now proceeds in stages. In each stage, students see a smaller version of the machine – a version which they then design, code and successfully test before proceeding to the next stage where they now see an enlarged version of the machine, and then revise and upgrade the previous design and coding. They also revise the user test programs to test the added features. In this approach, at each stage, they deal with a reduced level of complexity which they are able to comprehend and hence handle. In short, components of MOS project are now assigned in a far more even manner over the course of the semester, eliminating the back-end weighting.

2. It is, of course, a major task to determine the number of stages and then properly define them. On one hand, the more the number of stages, the less complex each stage is, but on the other, there is necessarily some redundancy built into each stage so that it lends itself to testing. Though this redundancy gets removed in the next stage, new redundancies appropriate to testing this new stage now get introduced. Insertion of redundancies increases the overall work needed to implement the project and calls for smaller number of stages. This approach also increases the workload of the faculty because now for each team, there are as many presentations as the number of stages.
3. Earlier, while the architectural design was described using the appropriate tools (chalkboard, transparencies, computer displays), no handouts were intentionally provided. It was assumed that working out and putting different parts of the project together by students will improve their comprehension of the project. However, whereas several teams performed this task successfully, there still were others which had difficulty in working with this jigsaw puzzle type of exercise. Hence appropriate handouts are now given at each stage. Enough details are still left for students to figure out. Careful analysis is needed to decide what to include and what to leave out.

The sequence of OS topics presented in class is now tailored to meet the requirement of the project at each stage. For example, when students are to implement the stage with paging and virtual memory, the topic of memory management is presented in the class.

This 3-prong approach is now being followed for the last four to five years and significant improvement has been noticed. A great majority of students in the class approaching 90% or so are completing the project in time when the timely completion rate hovered around 60% or so earlier.

STAGE DEFINITIONS

As pointed out earlier, stage definition is a major task. Their definition is described in this section. The MOS incorporates the following salient OS features:

- Simulation of hardware including CPU, memory, auxiliary storage, timers, channels, and i/o devices
- Input-output including channel operations
- I/O spooling using buffering
- Memory management including paging and virtual memory
- CPU management including scheduling and dispatching
- Interrupt handling including raising and servicing of interrupts
- Multiprogramming including time sharing
- Run-time error handling
- Managing input (user test programs) and output (user program output and vital run-time statistics) files
- Auxiliary storage (drum) management
- Master and slave mode transitions

The MOS also consists of two interconnected software layers. The inner layer provides the simulation of the hardware platform, and the outer layer is the OS code. The

OS design at each stage highlights this distinction and ensures that no unnecessary coupling exists between these two layers, and simultaneously provides for the features described above.

STAGE ONE

Users minimally need input and output devices, main memory and a CPU. Hence, these four pieces of hardware are simulated at this stage. To keep the I/O interface simple, direct I/O is implemented. Further, no error checking is done and students are instructed to write error-free user programs for MOS testing. Master mode and slave mode are clearly separated. The MOS runs in master mode whereas user programs run in slave mode. Supervisor interrupt (SI) is implemented to transfer control from slave mode to master mode on input (GD), output (PD) and halt (H) instructions which are all privileged instructions. The OS services the SI interrupt resulting in either data input to the program (GD), data output from the program (PD) or termination of the program (H). User programs are loaded by MOS one at a time (no multiprogramming). Loading of a user program from input file is initiated when the previously loaded user program terminates, and on the completion of loading, the mode is switched to slave mode and control of the CPU is passed on to the user program. In brief, OS in the master mode handles READing, WRITEing, LOADing and TERMINATION while the simulated hardware in the slave mode handles the user program.

STAGE TWO

At this stage, paging is introduced without the multiprogramming which is done at the next stage because experience has shown that most often students are able to run user programs individually with correct output but when they resort to multiprogramming, project runs into trouble. An artificial constraint is imposed to implement paging such as loading in only odd or even numbered pages or using a random number to select a page to load into.

Students are now asked to include errors in the test programs and hence PI interrupt is introduced. An operation code error occurs if the first two bytes of an instruction does not match one of the seven specified mnemonics. An operand error is caused if the last two bytes of the instruction is not numeric. The cause for page fault error should be obvious. Students are also asked to include in the test programs “Time Limit” and “Line Limit” errors. The former is caused when the program attempts to execute beyond a time limit specified by the user. This error is signaled by the hardware Timer Interrupt, TI. A line limit error is caused when a user program attempts to print a line beyond a limit which is again specified by the user, and detected by the OS during I/O. These two limits are specified on the first job control statement. The MOS also detects an “out-of-data” error caused by having no more data on an input (GD) instruction.

The design and coding of both slave and master mode require enhancement at this stage. In addition to SI interrupt, the slave mode now detects and raises PI interrupt. Hardware simulation is also changed to incorporate generation of TI interrupt. It is pointed out to students that while PI and SI can’t both be generated in the same cycle, the TI interrupt may be raised independently. Now the OS is also revised to detect, and take

necessary action for servicing "out-of-data" and "line limit" errors as well as PI, SI, and TI interrupts.

STAGE THREE

The remaining features of virtual memory, multiprogramming, channel operations, input-output spooling, buffering, auxiliary storage and multiprogramming are all added at this stage. Time slice concept and hence "Time Slice up interrupt" are introduced to take care of multiprogramming. Students are asked to implement process scheduling using preemptive round robin scheduling algorithm.

Channel operations, concurrent with CPU execution, are introduced. While Channel 1 and Channel 3 together perform input spooling, Channel 3 and Channel 2 cooperate to perform output spooling. Buffers are used as synchronizing agents. In addition, Channel 3 performs three additional tasks of loading the program from the drum into user memory, reading data from the drum into user memory on GD instruction and writing output lines from user memory to the drum on PD instruction. Channels get initiated by the OS, and on completion of the assigned task, they raise I/O interrupts. Writing of Channel 3 interrupt servicing routine becomes quite challenging because it takes care of five possible tasks. This routine becomes a major component of the OS. Virtual memory implementation and page faults are major issues that are dealt with at this stage.

The slave mode requires only minor changes, but the OS requires a major overhaul to include servicing of I/O interrupts, multiprogramming, and spooling.

STAGE OPTIONS

The third stage of the project implementation is still quite complex and in order to simplify it, either it could be split in two stages or some features from this stage may be moved to earlier stages. The choice is not obvious. Multiprogramming can't be separated from channel operations, spooling can't be implemented without channels, buffers are primarily needed to synchronize channel operations, and virtual memory implementation makes no sense without multiprogramming. It is thus not obvious as to what features could be moved to earlier stages. Similarly, there aren't obvious ways to split this stage. It remains a major task to determine the number of stages and their specification.

CONCLUSION

There are two major conclusions. First the recommendation of the ACM curriculum committee to design an OS from scratch in an OS course is completely vindicated. Students admit that they wouldn't have learned the concept of OS without implementing the project and strongly recommend that it should stay as an integral part of the course. Second amongst the two approaches used where in one, students were required to complete the design of the complete project before they could start coding and testing, and in the other, where a stepwise refinement approach was used, the second approach was found to be more effective.

REFERENCES

1. Onkar Sharma et al, "An Operating System Project, its accompanying problems, and their object-oriented design solution", The Journal of Computing for Small Colleges, Vol. 8, No. 2, Nov. 1992.
2. http://www.computer.org/educatio n/cc2001/final/index.htm
3. Shaw, Alan C. The Logical Design of Operating Systems, Prentice Hall, 1974.
4. http://www.acm.org/education/curricula.html#cc204
5. Lubomir Bic & Alan C. Shaw, Operating System Principles, Prentice Hall, 2003.
6. Nutt, Gary Kernel Projects for Linux, Addison Wesley Longman, Inc. 2001
7. Comer, D. & Fossum T. Operating System Design, The Xinu Approach , Prentice Hall, Inc 1988

ACKNOWLEDGEMENT

I acknowledge the help of Evan Jones for technical proof reading of this paper.