TEACHING OS DESIGN THROUGH IMPLEMENTATION OF A SIMULATED OPERATING SYSTEM*

Todd Peterson and Sean Crosby
petersto@uvsc.edu crosbysean@msn.com

INTRODUCTION

Designing a senior level OS course is a difficult process. The main difficulty is designing appropriate and challenging projects that will reinforce the major concepts for such a course. There are a few main options for designing these projects. The first is to implement an operating system from scratch using native assembly language coupled with a high level language such as C or C++. The second is to assemble a collection of projects that do not interconnect, but still have some effectiveness in teaching the major OS concepts.

A third approach is to design an operating system for a simplified simulator. This approach has the advantage of being easier to implement than a native OS and allows more interconnectivity than the assembly of projects approach. The main difficulty with this approach is that the OS needs to be coded entirely in assembly language, or a separate compiler needs to be built. This is usually an error prone process, and provides a level of frustration for the students.

The approach we take is to implement the OS as a separate part of the simulator. Calls to the operating system trap to the OS part of the simulator. There are several advantages to this approach. First, the operating system can be written in a high level language such as Java or C#. Second, all of the projects are interconnected. Third, the operating system is managing "real" programs running in a simulated environment. Finally, the coding process is not as difficult as writing native code.

In this paper we describe the simulator that the OS is designed for, the OS that the students implement, and the learning experiences of the student that did the implementation.

PROJECT NOTES

Due to the large scale of this project, we recommend that it is completed in two semesters. The simulator is to be built in an Advanced Computer Architecture class. The operating system is to be built in an Advanced Operating Systems Theory class. The greatest benefit can be achieved when the different parts of these projects are completed on the same timeline as the topics are taught in class.

THE SIMULATOR

The simulator is designed to help students understand basic computer architecture and low level programming. The assembly language should be simple to allow most of the focus to be placed on the theories of computer architecture. The students have an opportunity to become familiar with assembly code without learning proprietary details of other assembly languages and architectures (Intel's for example). A specification for the assembly language we used has been appended to the end of this paper.

To gain the full benefit of learning and productivity, it is recommended that an object oriented language be used. This will allow students to encapsulate functionality into the different components of the simulator. Later on, when the operating system is implemented, the separation of the existing objects will allow for quick integration.

Our simulator implements a basic register-register architecture. The following addressing modes were used: Register, Direct, Register Indirect, and Immediate. The Direct addressing mode is for loads and stores. Loads and stores are for both bytes and integers. The Immediate addressing mode is used for the add-immediate and trap instructions. Register-indirect addressing was implemented to allow for the addressing of arrays.

One of the first projects the students will need to complete is designing their memory. They can specify their own word size, but memory should be byte addressable. The memory can be implemented with an array of bytes or integers wrapped in its own class. A library of functions can be placed in the memory class to allow for easy access to items in memory. The memory should be designed to allow for the storage of the data and code segments. The remaining space in the memory will be used for the stack.

The registers, like the memory, are an array encapsulated into their own object. General purpose and special purpose registers are used. The array of general purpose registers should be large enough to comfortably run the sample programs. The special purpose registers can be separate data members inside of the Registers class. There are five special purpose registers: Program Counter, Stack Base, Stack Pointer, Stack Limit, and Frame Pointer.

Four of the special purpose registers are for maintaining the stack. Push and Pop instructions were not implemented in our assembly language. Instead, these special purpose registers are used to create activation records. If the students are preparing to take a class on compiler theory, this part of the project can be invaluable.

A built-in assembler is used to generate byte code. We implemented a two pass assembler. Pass one builds a symbol table for the different labels. Pass two converts the assembly to byte code using the symbol table to generate addresses for the labels. The

resulting byte code can be written to a file, or can be placed right into the simulator's memory.

A few basic traps are implemented during this first semester. Traps are used for: screen output, keyboard input, conversions between integers and characters, and stopping the program. Initially these traps are implemented in the simulator. Later, after the operating system is built into the simulator, the operating system should handle these calls.

Our simulator was completed in five stages. Designing the memory and registers came first. The assembler and simulator were constructed side by side during the final four stages. This allows the student to get something working early on. The instructions are added in three stages: 1) Arithmetic, load, and trap to screen instructions, 2) Jump, compare, store, trap from keyboard, and move instructions, 3) All of the remaining instructions. Between stages two and three the students added function calls.

The students might have a better learning experience by implementing a few extra features. A debugger (either console or with a GUI) can be very beneficial in debugging the students' assembly code. Additional instructions can be implemented, including floating point operations and instructions that the students design themselves.

THE OPERATING SYSTEM

The purpose of the operating system projects is to help the students become familiar with basic operating system concepts and to work out some of the issues associated with implementation. The operating system is implemented as a separate part of the simulator. OS system calls are handled as traps to the OS part of the simulator. This allows the students to write the OS in a high-level language and run real processes using the simulator they have already constructed.

This operating system project should be designed to resemble a real computer as much as possible. Separate classes are used to divide the simulator into hardware and operating system components. Each project in the second semester requires that changes be made to the simulator. A well designed simulator makes it easy to integrate an operating system. Low coupling is essential when moving these components around.

Project 1: A Simple Shell

Creating a shell to run on top of the operating system is a simple way to start forming the operating system. A few initial changes to the simulator are imminent. Both Operating System and Shell classes are created. The concept of a process is formed.

At this point in the project a number of changes had to be made in the simulator. The simulator went from being the center of the project to a component used for running processes. A Process Control Block (PCB) class is used to keep track of the process' memory, registers, program counter, process id, and state. The simulator is now decoupled from our 'virtual hardware' and can run a process based on the members of the PCB class. The operating system and shell are assigned PCBs. This is a crucial teaching point in the class. It is important that the students understand the different process state. Even though the Operating System and shell aren't true processes, they can still have the state in the PCB blocks set to the appropriate states.

Instructions in the shell include those to browse the local file system, load assembly files, and run processes. Loading a process is completed by running the assembler (which loads the resulting byte code directly into memory), instantiating a PCB object, assigning a Process ID (PID), and setting the process' status to ready. The PCB is added to the PCB table.

Several shell commands are implemented to allow the user to see the current processes and to view a help page. A process is run by the shell command 'run' with the PID as a parameter. The traps implemented in the simulator should be changed to make calls to the operating system. Context switching is implemented to copy the registers and program counter into the PCB. Before control returns back to the process the register values should be copied back into the hardware. When a process is complete, its state is changed to 'finished' and it is removed from the PCB table.

Project 2: Memory Management

The purpose of the memory management project is to give students experience with the implementation of allocating and deallocating memory. Implementing a memory management system in a high level language makes it possible to complete an advanced system in a reasonable amount of time. Full implementation into the simulator allows live processes to allocate and deallocate memory.

The students implemented a system heap that is managed by the operating system. Only minor changes were required in the simulator. Two traps used for allocating and deallocating memory were added.

A Heap Manager class is created to manage the system heap. It keeps a list of all the blocks of memory. A first fit algorithm is used to find the first available block that is large enough for the requested size. Blocks that are larger than needed are split. One block is marked as occupied. On deallocation, a check is performed to see if the preceding or following blocks are unoccupied. If an unoccupied block is found, the blocks are merged.

Memory protection is also implemented. When the Heap Manager marks a block as occupied, it assigns the PID of the requesting process to the block. Every time a request is made to access that memory, the process identifier is compared with the PID of the block. If they don't match, an exception is thrown. A similar method was used to protect main memory. The main memory assigned to each process was marked with a PID for verification.

A special shell instruction ('mem') was implemented to print out the list of blocks with their bounds and PID whenever a call to new was made. This made it possible for the instructor to verify the memory manager.

Project 3: A Scheduler

A scheduling algorithm is probably one of the most challenging concepts to truly implement in a stand alone project. If done on a Windows system, it isn't very reasonable to manage other Windows processes at a high level. An interpreter can be designed to run scripts, but they are not true processes. Using scheduling in connection with the simulated operating system leads to a true implementation of a scheduling algorithm.

This project builds on many of the changes made in project one. If project one was implemented correctly, the mechanism for context switching should already be available. Project one ran every process from start to finish. Scheduling requires that the simulator can start and end processing from points other than the beginning or end of the instructions. When a process is switched out, its registers are stored in the PCB. When a process is switched in, the simulator resumes execution at the point of the program counter. In order to simulate timer interrupts, we changed the 'run' method on the simulator to take a parameter of how many instructions to execute.

We implemented the Round Robin Scheduling algorithm. It simply looped through the processes in the PCB table giving each a specified number of instructions to run. This is a great time in class to teach about interrupts, blocking for I/O, and specifying the amount of time the operating system gets.

Project 4: A File System

Creating a file system in a simulated operating system has two major benefits. First is the opportunity to implement a disk management system. Second is the challenge of integrating the file system into the simulator.

The basic disk is initially a fixed sized file (called 'DISK') full of zeros on the local file system. The File System class creates its own structure inside of the file. We used blocks similar to Unix INodes but with only direct indexes in the node. Each block is the same size. A directory has a list of block ids of its children. The 'free block list' block kept a list of the addresses of free blocks. The Master Block keeps a reference to the root directory and a list of 'free block list' blocks.

The operating system keeps track of the open files. An open file is stored in a FCB object in the FCB table in the operating system. Changes to a file can be simply written through, or can be saved until the file is closed depending on the learning experience that is desired.

The file system is initially set up to be controlled by the shell. The shell implements the following commands: 'ls', 'cat', 'cp', 'mv', 'touch', 'head', 'tail', 'pwd', 'cwd', 'df', 'du', and 'rm'. The rm command deletes files, directories, and sub items. We also implemented a function to copy files to and from the local file system.

Six new traps were introduced into the simulator and the assembly language to allow for disk access. This is invaluable for the students to envision how real processes communicate with a disk. The following traps were created: open a file, close a file, read a character, write a character, create a new file and delete a file. One the issues we had to resolve was communicating the file name to the operating system through a trap. This was solved by putting the address of the beginning of the string in a secondary trap register.

This was by far the most time consuming project with many different challenges. Some of the challenges for the students to work out include: using absolute paths, keeping track of open files, deleting sub items, putting files into the file system, which traps to implement in the simulator, and having the shell keep track of where it is at in the directory system.

LEARNING EXPERIENCES

In the Advanced Operating Systems class, there were a few options, or tracks, given for the way in which we would implement the basic concepts that were taught in the course. Track one was to build an operating system for a real architecture, typically Intel. This was thought to be the track that held the potential for being the greatest learning experience. Track two was to write an operating system in assembly code on an existing simulator. Track three was to implement the different concepts in a high level language. The problem with this track was that the different projects weren't interconnected. Most students choose this track to avoid writing massive amounts of low level code.

In the end, the students who had worked to write an operating system for an Intel architecture were only able to implement a few different concepts. The students who chose track three were a little dissatisfied with their learning experience. The student who chose track two wrote a lot of assembly code, but was not able to complete all of the projects. One of the authors enjoyed his learning experience and the range of concepts he was able to implement into my operating system.

THE HORIZON

A number of other projects can be performed in conjunction with the simulated operating system. A project could be done to implement virtual memory. Processes could be moved to and from disk. The file system commands can be threaded allowing for blocking of processes. Students can also implement inter-process communication and threads. Students who take a compiler course could write a compiler that generates target code for their simulator. Shell commands could be added to run the compiler.

APPENDIX

Jump Instructions

Op Code	***Description***	***Operands***
JMP	Branch to Label	Label
JMR	Branch to address in source register	RS
BNZ	Branch to Label if source register is not zero	RS, Label
BGT	Branch to Label if source register is greater than zero	RS, Label
BLT	Branch to Label if source register is less than zero	RS, Label
BRZ	Branch to Label if source register is zero	RS, Label

Move Instructions

Op Code	*Description*	*Operands*
MOV	Move data from source register to destination register	RD, RS
LDA	Load Address	RS, Mem
STR	Store data into Mem from source register	RS, Mem
LDR	Load destination register with data from Mem	RD, Mem
STB	Store byte into Mem from source register	RS, Mem
LDB	Load destination register with byte from Mem	RS, Mem

Arithmetic Instructions

Op Code	*Description*	*Operands*
ADD	Add source register to destination register, result in destination register	RD, RS
ADI	Add immediate data to destination register.	RD, IMM
SUB	Subtract source register from destination register, result in destination register	RD, RS
MUL	Multiply source register by destination register, result in destination register	RD, RS
DIV	Divide destination register by source register, result in destination register	RD, RS

Logical Instructions

Op Code	*Description*	*Operands*
AND	Perform a Boolean AND operation, result in destination register	RD, RS
OR	Perform a Boolean OR operation, result in destination register	RD, RS

Compare Instructions

Op Code	*Description*	*Operands*
CMP	Set destination register to zero if destination is equal to source; Set destination register to greater than zero if destination is greater than source; Set destination register to less than zero if destination is less than source.	RD, RS

Traps

Op Code	*Description*	*Operands*
TRP	Execute an I/O trap routine (a type of operating system or library routine). IMM Values 1, write integer to standard out 2, read an integer from standard in 3, write character to standard out 4, read a character from standard in	IMM

TRP	Execute STOP trap routine. 0, stop program	IMM
TRP	Execute a conversion trap routine. 10, char to int '0' → 0 . . . '9' → 9 otherwise -1 11, int to char 0 → '0' . . . 9 → '9' otherwise -1	IMM

Traps Implemented for the Simulated Operating System

Op Code	***Description***	***Operands***
TRP	Execute an memory management trap routine. IMM Values 20, allocate memory on the system heap (new) 21, deallocate memory on the system heap (delete)	IMM
TRP	Execute a File System trap routine. 30, open file who's name starts at the address in r2. Return FCB ID to r2 if succeeded 31, close the file who's ID is in r2 32, read one char to r2, r3 has the FCB ID 33, write to file, r2 has the char, r3 has the FCB id 34, create a new file who's name starts at the address in r2. Return FCB ID to r2 35, delete file who's name starts at the address in r2. return succeeded to r2	IMM

Directives

Directive	***Description***

.INT value	Allocate space for an integer.
	Example:
	`MONTH .INT 12`
	`DAY .INT 9`
	`YEAR .INT 2005`

.ALN	Align the next byte on a word boundary.

.BYT value	Allocate space for one byte.
	Example:
	`NAME .BYT 74`
	`.BYT 73`
	`.BYT 77`
	`.BYT 0`