AN ADVANCED OPERATING SYSTEMS PROJECT USING CONCURRENCY

Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C. 20059

ABSTRACT

Most courses in operating systems follow one of three patterns : high level discussion with most programming done in a high level language; building operating systems from device driver level up; and courses which use emulator programs which emulate special architectures. In this paper, we describe a course which emphasizes the strong points of these approaches. The course also emphasizes teaching concurrency, introducing networking, and developing a deeper understanding of many of the system calls and system commands of the UNIX operating system. The major course project is described in the paper.

1. INTRODUCTION

The topic of concurrency is extremely difficult for students. It is so fundamental that it must be reinforced in the curriculum and not merely discussed in a single course. This paper discusses a course taught at Howard University which emphasizes the nature of concurrency. Shub [9] has commented that this topic is often covered only at very high levels in a typical operating systems course The course we describe here is a second course in operating systems. Primary goals of the course are the reinforcement of topics learned in the first course in operating systems, knowledge of concurrency at both a theoretical and operational level, use of a network for more sophisticated purposes than electronic mail, evaluation of the performance and trade-offs made in operating systems design, and development of a deeper understanding of the UNIX operating system which provides the development environment. The course project is to develop a simulation of a parallel computer architecture using processes to simulate processing nodes. Each of the simulated processing nodes runs an interpreter that executes both shell commands and commands to send data to other processors. The communication between simulated processing nodes is implemented using both message passing and shared memory facilities available under AT&T System V UNIX. Materials used in this course include the texts [1], [5] and [7].

The course is heavily based on the UNIX operating system. It uses the shell both as a command interpreter and as a means of performing computations. Most operating systems have command interpreters but few have the UNIX feature of allowing significant programming using the command processor. We also make extensive use of UNIX facilities for communication between concurrently running processes.

2. BACKGROUND OF THE STUDENTS

The course described here is open to both undergraduate and graduate students at Howard University. The prerequisite is a first course in operating systems which has included a typical overview of the major concepts in operating systems: device management, cpu scheduling, memory management, virtual memory, concurrency, etc. The students have typically implemented projects in device management, cpu scheduling, and memory management, but not in concurrency. All of the students are proficient in C. Undergraduate students have taken a sur-

vey course in programming languages in which C and Ada (among others) are taught. The structure of an "Ada virtual computer" [6,p493-494] is taught and the students have completed a project which requires tasks and some level of process communication/control [3]. Undergraduate students have used the C language for some of the projects in the first operating systems course. Equivalent exposure is required of all graduate students; this is not a problem since most of these graduate students are employees of AT&T or Bell Communications Research attending Howard on sponsored graduate fellowships. Thus all of the students have seen concurrency at least twice before they take this course.

In addition, the students also have a user's knowledge of the UNIX operating system at least at the level of editing, compiling, and executing programs as well as the use of the electronic mail system on a single computer. The students typically have no experience with use of a network for anything other than electronic mail; many students do not have even that experience.

All students, graduate and undergraduate, do the same projects and take the same examinations. Graduate students are also responsible for written and oral research presentations that are based on assignments given by the instructor and are based on the current literature.

3. THE COMPUTING ENVIRONMENT

Howard students at this level do much of their work on a collection of AT&T 3B2 computers running AT&T System V UNIX. The computers are networked together via Ethernet using TCP/IP which supports electronic mail and a limited amount of running commands on other UNIX based computers on the network using remote file copy and remote login. There are many nodes on the network; the relevant ones for this paper are a machine for program development (labeled scsla in figure 1), a machine for experimentation (scslx in figure 1) and a machine for sending reports via electronic mail (scs2 in figure 1).

The development machine (scsla) contains the UNIX operating system with full compilation, debugging, networking, and editing facilities. It is a moderately well-equipped UNIX machine for editing, compiling, and general computing with a heavy user load. Previous experience with this course has shown that the additional computational load on a general purpose computer caused by the increased number of processes and the need for inter-process communication (IPC) is intolerable. In addition, these new processes make the machine prone to crashes or periods in which little computation can proceed because of the large number of running processes. For these reasons, all projects for this course are run only on the experimental machine scslx.

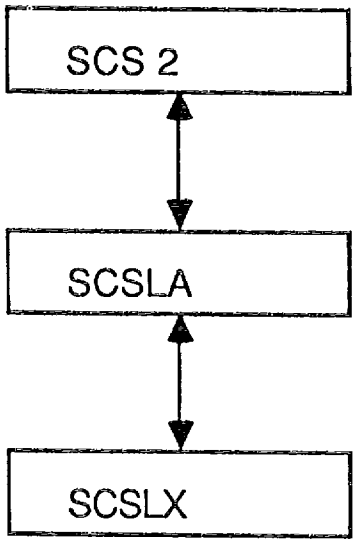

figure 1 The development environment

The machine for experimentation (scslx) is an AT&T 3B2/310 with 2 MB memory, a very small hard disk (31 MB), and limited compilation facilities. This makes the machine unusable for program development and forces use of the network for sending files from the development machine to the experimental machine. It also encourages the use of the network for the execution of programs. It is expected that students will frequently crash the underlying UNIX operating system on this machine because of misuse of the IPC facilities. The scslx computer has facilities to handle the full range of IPC methods available in System V UNIX such as semaphores, FIFO's and shared memory. All these IPC methods are needed for the projects in this course.

4. THE BASIC IDEA

The standard model of multiprocessing is shown in figure 2.

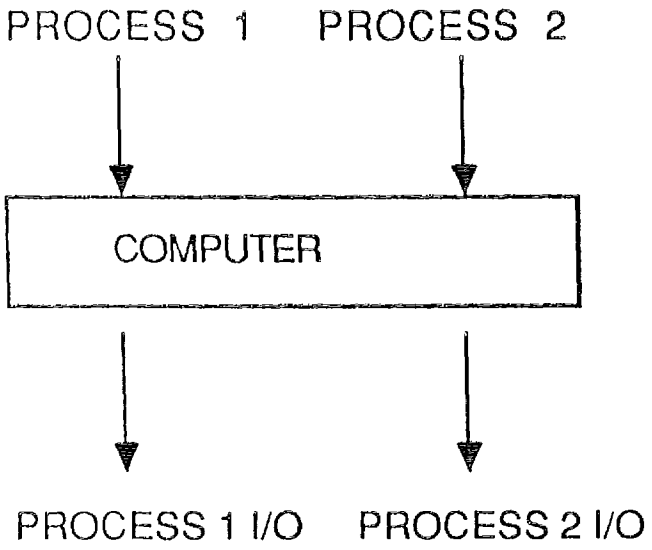

figure 2 The standard multiprocessing model

Here *COMPUTER* refers to a system that may have one or more cpu's. The model in figure 2 is replaced by the logical model in figure 3.

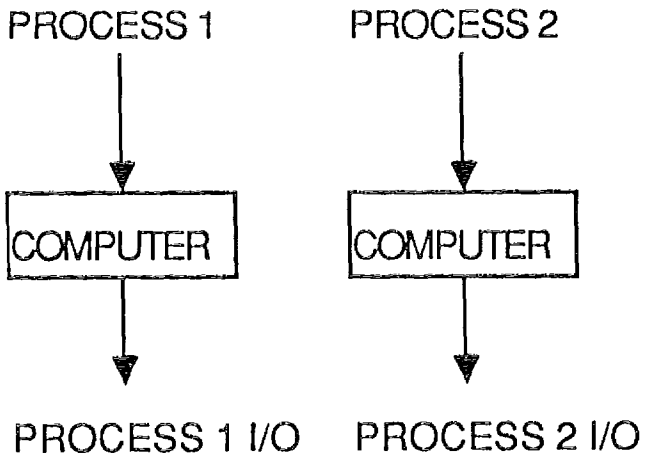

figure 3 The logical multiprocessing model

Note that this logical model is close to reality under AT&T System V UNIX because of *fork()* copying the data segments.

Students are required to write various portions of an operating system for hypothetical computers of the types listed below. They are required to use C and to make use of the accessibility in C of the system clock and various signals. Thus this course has the advantage of having small computers to write operating systems for as well as providing an opportunity to do advanced programming in C.

5. THE PROJECTS

Students are divided into groups of three or four and are required to develop emulators for various multiprocessor architectures. We describe a typical project in detail in order to indicate the type of tasks that students must carry out. Other commonly chosen projects will differ only slightly from the one described in this paper.

Students write a simulation of an eight node hypercube computer. The nodes of the computer are simulated by processes that are tiated by *exec()*. Each of the nodes executes a process that is an emulation of a single node computer. Therefore, each of these nodes is now a full processing element that has the power of the program that is executing within the new process.

The program executes in three phases. Phase 1 is a check to see if sufficient resources are available in order to be able to create and execute the processes needed for the assignment. Phase 2 is the actual creation of the processes and their initialization, including the attachment of shared memory regions to the processes. Phase 3 is the interpretation and execution of the commands in an input file. The three phases are described below in more detail.

The experimental machine (scslx) permits a maximum of 10 semaphores, 10 shared memory regions, and 10 FIFO's to be active in the system at any one time. In order to avoid depletion of these resources, each student program must begin execution with the testing of the existence of a specially named file "/usr/tmp/opsys.lk". Existence of this "lock file" is tested by the *creat()* system call. The call to

creat() returns -1 if the file already exists and 0 if it does not. The code is

```
if (creat("/usr/tmp/opsys.lk",  ) == -1)
        {fputs("Error --insufficient resources");
        exit(0);
        }
REST OF PROGRAM
unlink("usr/tmp/opsys.lk");
```

This acts as a slow binary semaphore. It is unsatisfactory for general IPC use but works fine for a one time check. If no other group is using the machine scslx, then the /usr/tmp/opsys.lk file does not exist, *creat()* returns 0, and the program can continue because resources are available. When the program finishes execution, it releases resources and removes the file /usr/tmp/opsys.lk using the *unlink()* system call to indicate that the resources are now available. If *creat()* returns the value -1, then no resources are available and the program will exit gracefully.

Phase 2 involves the creation of the processes making up the processing elements (PE's) and the scheduler. This involves *fork()*'s, *exec()*'s, and the attachment of shared memory regions to the processes using the system calls *shmget(), shmat(), shmdt()*, etc.

Each of the processing elements gets its input from an individual input file; there are as many individual input files as there are processing elements. The individual input files are created by a separate scheduler process that takes a large input file and separates it into the individual input files according to figure 4.

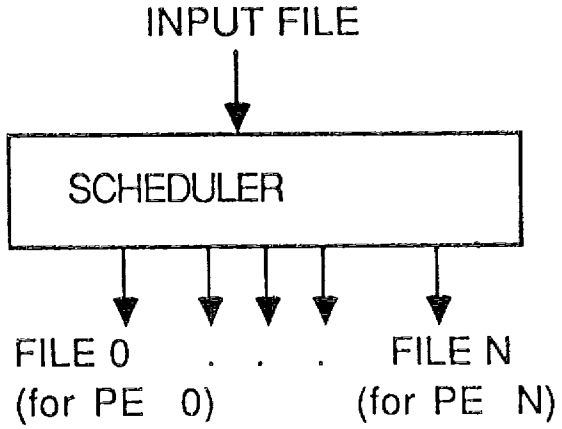

figure 4 File processing diagram

The large input file contains information about commands that are to be executed , how these commands are to be executed, and the processing nodes that is to execute the command. A typical large file looks something like this.

```
line 1      10
line 2      x=3
line 3      y=5
line 4      z='expr $x + $y'
line 5      send z to 3
line 6      11
line 7      x=5
line 8      13
line 9      r2=7
line 10     receive z from 0
line 11     r0='expr $acc +$r2'
```

Lines 2, 3, 4, 7, 9, and 11 are standard UNIX shell commands. Line 1 contains the characters '10' which are a delimiter protocol indicating that everything until the next delimiter protocol is to be run on processor number 0. The *send* command on line 5 indicates that the value of a variable named 'z' is to be sent to processing element number 3. This statement implies that some form of IPC is to be used between the processes. The receive statement on line 10 is similar. However, any later statements to be executed on processor number 3 are blocked until the value of 'z' is actually received.

Each processing element has the following variables used for data: r0, r1, ... r9 (representing registers 0 .. 9), acc (representing an accumulator), x, y, z, a, and b.

The *send* and *receive* statements on lines 5 and 10 require that data be sent from one process to another.

Students are formed into small groups (usually three or four) and are responsible for the development of protocols for message passing and for general communication between the various processing elements. The IPC method used is FIFO's, with messages being sent and received from these FIFO's. The structure of a message is

```
struct message {int r0, r1, r2, r2, r4, r5,
r6, r7, r8, r9, acc, x, y, z, a, b} msg;
```

and the *send* and *receive* functions are similar to those presented in Rochkind [7]. To facilitate programming, the entire message is sent even if only one of the fields is needed by the receiver.

Note that the values that are received may be used in computations by the receiving processing element. Therefore a blocking mechanism must be provided so the computations on the receiver are delayed until the sender's message is received. This is done using signals. Thus students need both FIFO's and signals to be able to implement this phase of the program.

An additional facility is available in the input file - the means of specifying IPC via shared memory instead of FIFO's. The syntax in the input files is

> send *shared* value to *processor number*
>
> receive *shared* value from *processor number*

This requires the students to use shared memory. Shared memory normally unstructured and therefore students access shared memory regions by overlaying the message structure mentioned above on the shared memory region. To simplify the programming, only one shared memory region is used per program. Blocking of concurrent reads and writes is done by using semaphores.

The three phases of the project are typically grouped into five separate subprojects, with each project buiding on the previous ones.

1. Creation of the individual processes, identifying the process ID using the *getpid()* system call and running a simple shell command such as *date* in each PE. The *creat()* system call is used as a binary semaphore to insure that the system's IPC resources are available. This corresponds directly to phase 1.

2. Implementation of the requirements of phase 2 is separated into two parts to be implemented in this and the next step. The first step is sending a message from each PE to its nearest neighbor via FIFO's.

3. The next step in phase 2 is sending a message from any PE to any other using FIFO's.

4. Phase 3 requires the creation of a parser to use the protocol delimiters discussed previously to break up the original large input file into individual files.

5. Shared memory IPC is implemented.

Thus many of the real world problems in software engineering are also reinforced by the course. Each of the groups and the design committee is responsible for weekly reports indicating design decisions, goals and milestones. The group leaders are also responsible for applying various software metrics during the specification, design, and coding phases of the project.

Some of the groups of students implement emulations of other multiprocessor architectures. Other architectures implemented are based on the topologies of dual bus hypercubes, stars, and rings. Their experiences were similar to the hypercube groups in that mastery of concurrency was essential for success of their programs.

Students were required to evaluate the performance of their programs which designed operating systems for their emulations. Evaluation techniques ranged from simple use of the system clock to more detailed timing analysis so that various algorithms designed by various groups were compared for both student generated and teacher generated test files.

6. EVALUATION OF THE COURSE

The goals of the course were reinforcement of the topics learned in the first course, knowledge of concurrency at both a theoretical and operational level, evaluation of the performance and trade-offs used in the design of operating systems and development of a deeper understanding of the UNIX operating system. All of these goals were met in this course.

Reinforcement of the topics covered in the first course is a by-product of the implementation of memory management and scheduling algorithms. Students learned some of the details of memory management by actually manipulating the amount of space attached to a UNIX

process. The effects of the scheduling algorithm became apparent when the processes were initially created.

The essential topic of concurrency was discussed at a theoretical level using the abstract concepts of *fork* and *join*, precedence graphs, *parbegin - parend*, precedence graphs, as well as standard examples. Operational understanding was obtained by use of the *fork()* and exec() commands and the synchronization of various processes using the devices of pipes, messages, and semaphores. Performance measures were made using the built-in system clock. In some situations, more detailed profiles of program performance were given. In future semesters, some of the students will also use Concurrent C as an implementation language.

Actually deciding on the algorithms to be used and their implementation gave the students a deeper understanding of the trade-offs in operating system design. Students were forced to use the UNIX operating system at a fairly sophisticated level. They had to use many system calls and interpret signals correctly. Major IPC methods were used. Embedding of the hypothetical computer in the UNIX operating system environment allowed access to signals, messages, semaphores, shared memory, the system clock, UNIX file structure, etc. while allowing the students to write in a higher level language than assembly language.

In addition, having one machine for program development, one for reporting, and one for program execution required significant use of the network. The executable files were sent by rcp and were run under remote login. As a bonus, students obtained considerable experience in experimentation and in general principles of software engineering.

Perhaps the most important feature of the course is that both high level and low level concepts were taught and implemented while students obtained a deeper understanding of the UNIX operating system and networking. See [2] for comments on integration of networking into the curriculum.

This course has been taught for several years using the format described in this paper. It has been especially popular with our graduate students who have considerable exposure to UNIX such as in their previous work at AT&T or Bell Communications Research.

7. SOME SUGGESTIONS FOR SIMILAR COURSES

Clearly a major feature of the course is the embedding of the emulator in the UNIX operating system. This can be done using other emulators and other operating systems as long as a language which allows concurrent processing is available. See [4] and [8] for examples. Another feature of this course was the existence of separate machines for development and for execution. We suggest that projects such as this not be done unless there are separate machines and a network available. Indeed, the students often crashed the experimental machine by overrunning the process table and such activities are frowned upon by other users.

ACKNOWLEDGEMENT

This research was partially supported by the Army Research Office under grant number DAAL 03-89-G-0100 and by the Maryland Procurement Office.

REFERENCES

1. Bach, M.,*The Design of the UNIX Operating System*,Prentice Hall, Englewood Cliffs, 1987.
2. Cassel, L.N., *Networking Elements in a Files Course*, SIGCSE Bulletin, vol 19, No 1, Feb. 1987, 343-345.
3. Leach, R.J., *Experiences Teaching Concurrency in Ada*,, Ada Letters, vol. 7, no. 2 (1987) 40-41.
4. Olagunju, A., and E. Borders, *Using Emulators as a Vehicle for Instruction in Systems Programming*, SIGCSE Bulletin, vol 19, No 1, Feb, 1987, 132-135.
5. Peterson, J.L., and A. Silberschatz, *Operating Systems Concepts, Alternate. ed.*,, Addison Wesley, Reading, MA, 1987.

****
CONCURRENCY-- continued on page 62

decides that a given device is to have service, he will jerk his foot, thus sending an interrupt to Apple-server. Apple-slave also is responsible for informing Apple-server which interrupt service routine is to be executed. This situation is analogous to the use of a support chip such as the Intel 8259 programmable interrupt controller in an 8086-based system.

Summary

We have tried the analogy described above several times in our introductory computer architecture/assembly language programming classes. Each time it has been extended and refined a bit. Student reaction has varied from guarded amusement to an enthusiastic "Gee whiz--that's the way it works!" We think the analogy is a useful tool in teaching this subject matter.

****
CONCURRENCY-- continued from page 44

6. Pratt, T.W., *Programming Languages, Design and Implementation, 2nd. ed.*, Prentice Hall, Englewood Cliffs, N.J.,1984.

7. Rochkind, M., *Advanced Unix Programming*, Prentice Hall, Englewood Cliffs, N.J., 1985.

8. Shay, W.A.,*A Project for Operating Systems Simulation*, SIGCSE Bulletin, vol 18, No 1, Feb. 1986, 289-295.

9. Shub, C., *The Decline and Fall of Operating Systems* , SIGCSE Bulletin, vol 19, No 1, Feb. 1987, 217-220.

10. Wolfe, J., *Operating System Projects on two Simulated Machines*, SIGCSE Bulletin, vol 19, No 1, Feb. 1987, 212-216.

****
COATROOM-- continued from page 46

Note that the "tag" will be a pointer to a node in the list. Unless the list is bidirectionally connected (or a ring) there will be problems with the implementation of claim. For pedagogical reasons, I hand out a buggy implementation to students of this simple case and have them find the problem and correct it. This is a difficult problem for them and the usual correction of having a tag point physically to the predecessor of the node to which it logically refers will not work correctly as that node may be a candidate for early removal. In fact a simple sweeping garbage collection scheme that marks nodes for removal at claim and then sweeps once per check or once per claim is simple to implement and instructive. Note that a node may be physically removed only when its successor has been claimed.

Another instructive feature of this implementation relates to the problem of copying tags and presenting them for a claim. If such is possible then the implementation of claim is further complicated. Discussion of this problem, and the disastrous effect of ignoring it, is a natural way to introduce the ideas of "capabilities" and verifying authorized use of a resource. The idea of a capability is introduced by creating a tag which is a pair consisting of a pointer to a predecessor node and a unique reference number assigned by the system. The reference numbers are also stored with each coat. Now the claim operation checks the reference number in the tag against the reference number stored with the coat which the pointer actually references. If they do not match it is because the coat has been claimed and physically removed from the coatroom list. Problems with copying of tags may also be used to introduce the notions of reference counting garbage collection schemes.

e) Hash table. Here the coat itself could be passed through a hash function and the returned value, or hash bucket number, could be returned as the tag. Again, care is needed to avoid a buggy implementation. If the buckets cannot be guaranteed to be size one, there is not sufficient information in such a tag to uniquely retrieve the coat. Note that this may not always be a disadvantage. There are situations in statistical work where there is a requirement of anonymity in retrievals from the data. It is difficult to guarantee that a series of retrievals cannot be used to compromise the anonymity of the subjects in the database.

Conclusion: The coatroom is an example of an ADT which has a simple and intuitive description, many possible implementations, several subtle difficulties and a number of deep extensions. It can be valuable to introduce it early in a course in data structures, refer to it often, and use it to introduce additional topics such as garbage collection.

References:

(1) Goldberg and Robson, *Smalltalk-80 The Language and Its Implementation*, Addison-Wesley, 1983.

(2) Levy, *Capability-Based Computer Systems*, Digital Press, 1984.

(3) Sedgewick, *Algorithms, 2nd Edition*, Addison-Wesley, 1988.