Creating Interest in Operating Systems via Active Learning

Susan J Lincke
Asst. Prof. of Computer Science, University of Wisconsin Parkside
Kenosha, WI susan@lincke.org

Abstract - It is generally accepted that multi-threading, concurrency and semaphores (including producer-consumer, readers-writers problems, etc) are a major challenge to undergraduate operating systems students. In addition, while the author became fascinated while working with systems programming in industry, students did not appear to be interested in the theoretical, passive-learning approach of an all-lecture course. The challenge was to add an active-learning component that would help students to better understand concepts and enhance their interest, while still retaining all course content and cutting the lecture time. The OS course was modified to include about one hour of active learning for two hours of lecture. Our active learning approach uses exercises and mini-labs on a Linux system, and successfully raised interest for the students.

Index Terms - Computer Science Education, Operating Systems, Active Learning

INTRODUCTION

Multithreading and concurrency is a difficult but important topic to teach in an Operating Systems (OS) course, and thus has been a popular topic in Computer Science (CS) education papers (e.g., [1-3]). A major problem with teaching concurrency is that it requires experience to learn, but lecture is the normal mode of delivery in CS courses. Active learning enables students to learn by doing, and yet allows the student to ask questions when perplexed. Concepts behind active learning include that students collaborate to work on complex real world problems, that they interpret data, and that they are challenged to form knowledge, not simply to accept it [4]. Once active learning is adopted as a teaching technique, it is hard to revert to a full-lecture style, given the choice between watching students become bored in class or be active working with curiosity on real problems.

Active learning is being adopted at all grade levels because of its increased effectiveness. A popular teaching text by the Wongs for all grade levels [5] emphasizes that the person who does the work is the only one who learns. If a teacher increases the amount of time the student is working, the teacher increases learning. When a teacher lectures, the teacher is the one working and learning! Learning can be further extended via group work, or cooperative learning, where students work together to complete an assignment. Cooperative learning encourages deeper thinking and improves retention regardless of student ability, in addition to improving communication, interdependence, decision making, and higher self-confidence [5].

Papers on active learning in computer science have mostly addressed its use in introductory courses or programming (e.g., [6-7]), although Hill et al. [1] discusses how games can be used as active learning exercises in OS. Null describes different active learning techniques that can be used in general [8]. This paper focuses on using mini-labs with instructor-provided programs and OS tools, and exercises involving program development and analysis, as active learning components.

The goals of the course improvement include:

- Create student interest in working with operating systems (commensurate to what system programmers experience).
- Actively involve students in observing and analyzing actual OS behavior
- Introduce tools and utilities of real operating systems in the classroom environment.
- Improve critical thinking and analysis skills required for working with concurrent programming.
- Retain full course content while adding active learning components.

The major challenge with active learning is to fit all necessary topics into a semester long class of a 3 credit-hour lecture. The goal was to cover the full range of topics while lecturing for 1.5-2.5 hours and introducing active learning for 0.5-1.5 hours per week.

Section two describes the active learning methodology used in the classroom. Section three and four discusses the exercises and mini-labs that were used in class. Section five is a brief section on lessons learned. Section six is the conclusion.

METHODOLOGY

The steps recommended by the Wongs in presenting a lesson include [5]:

- Task Focus: Present an objective and describe why the concept is important.
- Lesson Presentation: Model and teach the new skill or information.
- Guided Practice: Practice the new skill together.
- Independent Practice: Have students practice independently. Provide individualized attention.
MINI-LABS

Mini-labs allow students to observe concurrency issues related to processes, threads, and semaphores. Carefully thought out guided procedures and questions lead students to analyze what they see, when running instructor-provided Java programs. Students use Linux system administrator tools such as ‘ps -al’ and ‘top’ to observe process states, blocking, priorities, processor utilization and memory utilization. Although working with real operating systems is difficult because of their complexity, students do get a real-world view of what to expect when programming, and get familiar with actual OS monitoring tools. Instructors get a lead in to future topics (“Remember when you saw in the lab…? Now let’s talk about that”). Our text [9] usually provides background as to why a specific OS behaves as it does.

The first mini-lab investigates C++ processes and introduces students to context switches, concurrency timing, process control blocks, process states, and parent-child relationships. An instructor-provided fork program spawns 3 child processes, which each prints ten laughs per row (either ‘He’, ‘Ha’, or ‘Ho’). If students enter a non-zero sleep time at a command prompt, a ‘sleep()’ function forces periodic context switches. The lab worksheet directs students to use ‘ps –al’ to observe process priority, states, and address space size per parent/child as well as to experiment with using Ctrl-C to terminate the parent process and see the effect on child processes. The students are asked what caused the process switches. The ‘ps –Al’ command allows students to see the full spectrum of processes on the system, as well as their priorities, states, and (often) reasons for blocking (e.g., sleep, print, timeout).

The next mini-lab introduces Java threads, memory sharing, and signal sharing. In this program, child threads share an object or are each associated with a private object. When threads share an object, errors appear in the output (due to non-synchronized shared memory) and students are asked why. The difference between sharing an object, but using separate stacks (for functions’ local variables) is observed. Ctrl-C signals have a different effect on child threads than processes. Using ‘ps –al’, students can observe how the OS manages threads, their states, their children, etc. The two labs demonstrate that different languages can result in very different behavior relating to scheduling and output buffering, and demonstrates the differences between processes and threads related to memory- and signal-sharing.

A mini-lab on semaphores introduces students to the mechanics of critical sections and semaphores. A voting booth program (Fig. 1) randomly generates voter threads which may enter a single voting booth if the P() operation permits. Print functions allow the students to observe voters entering and leaving the voting booth. The worksheet (shown as Fig. 2) has students initialize the semaphore to 0, 1 or two. Students are directed to change the P() to a V() and vice versa, to observe and explain the effects. This lab prepares students...
to understand producer-consumer and other more complex semaphore problems.

```java
for (int i=0; i<100; i++) {
    try {
        sem.Pacquire();
    } catch(Exception e) {…}
    System.out.println("Enter vote box "+ me.getName());
    vote = rand.nextInt(2);
    System.out.println("Depart vote box "+ me.getName());
    sem.Vrelease();
    total += vote;
    voteText = voteText + me.getName() + " ";
}
System.out.println("Completed Poll Station: "+ me.getName());
if (++done == 10)
    System.out.println(voteText);
// thread terminates after return from run()
```

### Figure 1
**INTRO TO SEMAPHORES LAB: VOTE.JAVA**

### Lab Exercise
Get the Vote.java file from my web page.

**Test one:** Comment out the Pacquire() and Vrelease() code from the run() function. How many voters can be in the voting booth simultaneously? What output do you get?

**Test two:** Reapply the Pacquire() and Vrelease() code within the run() function. How many voters are ever in the voting booth simultaneously? What output do you get?

**Test three:** Change the Semaphore constructor call to initialize the value to two (instead of one). How many voters can enter the voting booth simultaneously? What is happening and why?

**Test four:** Comment out the Pacquire() code from the run() function. Set the Semaphore constructor call to initialize the semaphore value to one. How many voters are in the voting booth simultaneously? What is happening and why?

**Test five:** Comment out the Vrelease() code from the run() function. Reinstall the Pacquire() code from the run() function. How many voters are in the voting booth simultaneously? What is happening and why?

### Figure 2
**LAB WORKSHEET FOR INTRODUCTION TO SEMAPHORES**

---

### Exercises

Whereas labs allow students to observe and analyze, exercises allow students to predict and construct when students think through problems. Exercises also allow students to think through problems at their own speed.

After an introduction to producer-consumer and possibly 1-2 examples of semaphore problems, students can begin to evaluate semaphore problems on their own. Students evaluate a simple reader-writer problem before proceeding to a more complex one. The worksheet also asks questions in order of complexity, to lead students to understand the programs. For example, students evaluate the purpose of a mutex semaphore before evaluating semaphores used to prevent readers and writers from operating simultaneously. Once the purpose of each semaphore is understood, the general program can be understood.

Java monitors can also be introduced in lecture with an example. Then an exercise (Fig. 3) asks students to evaluate an easy, then more complex version of the Dining Philosopher problem (which would be difficult to explain in class because the overhead example spans multiple pages! [9]).

When teaching process/thread scheduling, it is customary to teach FIFO, Round Robin, Priority, and other algorithms using 4-5 transactions. An active learning exercise includes more transactions to allow students to envision queues filling and emptying in a real time way, and allowing them to observe a system instead of just an algorithm. A homework assignment then asks them to program and evaluate different scheduling algorithms using semaphores.

A worksheet on deadlock has students draw resource allocation graphs for the Dining Philosopher problem. Students are asked to draw and solve a graph for a described scenario, and for a deadlocked scenario. Students are also asked to demonstrate that deadlock cannot occur when three threads each require at most two resources in a system holding four resources of the same type. (Taken from page 311 [9]).
THE DINING PHILOSOPHERS EXERCISE

Consider Figure 7.24 [8]
1) Describe what is happening in your own words in Fig. 7.24.
2) Deadlock can occur under a certain condition. How?
3) What should each chopstick semaphore be initialized to? Why?
4) It is possible to ensure that deadlock does not occur by only allowing a certain number of philosophers into the dining room. Add lines of code to implement this solution. (Hint: This requires one more semaphore.) What should the semaphore be initialized to?

Consider Figure 7.26 [8]
1) When does a philosopher self[].wait?
2) When does a philosopher self[].signal?
3) What conditions must be true for a philosopher to start eating?
4) How does this solution ensure deadlock does not occur?
5) What should each semaphore self[].be initialized to?

FIGURE 3
EXERCISE FOR DINING PHILOSOPHERS

Our Operating Systems course covers an introduction to computer networks (instead of file systems). As part of that topic, students are introduced to the Network File System (NFS) concepts of stateful/stateless servers and local/remote caching. After concepts are explained, students are asked to provide high-level pseudo-code for a simple stateless NFS client and server, using local caching. Through this exercise, students also learn how remote client-server programs work interactively. This is an excellent follow-up lab for a lab on Java RMI, but this year we used an introductory lab on .NET instead.

LESSONS LEARNED

What worked was creating student interest. Students were very interested in why they saw what they saw and felt more confident in working with concurrent systems. Teams learned at their own pace. They also appreciated working with actual UNIX tools and observing actual OS behavior. As a result, the course evaluations improved by 18% from the previous year across all questions on the evaluations.

What could be improved next time is to provide enough time for all students to think through critically what they saw in the mini-labs. Often, in order to complete the lab in the time allocated, the author was too quick to answer questions to teams instead of letting them think problems through. (At least they were interested in hearing the answer!) Possibly, as a result, critical thinking on the homework assignments did not improve to where it should. Possibly this can be solved by asking more direct questions for the homework labs, and/or by allocating more time to the early concurrency labs to encourage students to think through problems by themselves. While there was no reduction in coverage of OS topics by incorporating active learning, more streamlining of the course lectures or lab components may provide the extra in-lab time that would lead to improved critical thinking.

CONCLUSION

Experiential learning works well when teaching operating systems. Allowing students to observe the guts and behavior of actual operating systems makes a theoretical class more practical and interesting. It also allows the instructor to refer back: “remember when you saw…?” The students found the active learning interesting, had improved comprehension, and felt more confident about the subject material. Incorporating active learning into a course is possible by replacing note-taking with exercises, and by streamlining lectures and exercises.

REFERENCES


Session S3C

October 19 – 22, 2005, Indianapolis, IN

35th ASEE/IEEE Frontiers in Education Conference

S3C-10