CONCURRENT PROGRAMMING PROJECTS

IN OPERATING SYSTEMS I USING PASCAL-S

Carol M. Torsone
Department of Mathematics & Computer Science
St. John Fisher College
Rochester, New York 14618

Introduction

A great deal of the theory covered in an Operating Systems I course has to do with concurrency problems, and deadlock problems introduced as a result of concurrency. We can lecture about features found in concurrent languages, such as semaphores and monitors, and how they are used for mutual exclusion, coordinating the actions of processes, controlling the access of multiple instances of resources, etc. However, lecturing alone often doesn't provide students with any real feeling for concurrency and the issues and problems involved. They think they understand the concepts, but until they actually try to write and debug a concurrent program, they probably haven't really dealt with and understood the subtle issues involved.

However, after writing only two or three relatively simple programs in a concurrent programming language, it becomes very clear to students how processes can coordinate their actions, in a way that could not be done by study alone. Through the sometimes interleaved output of their programs, they can actually "see" their processes taking turns and the results of their interaction. They can "see" the result of not having mutual exclusion to a critical resource.

We have used the Pascal-S programming language for several years in the Operating Systems I course, for the purpose of writing concurrent programs as lab exercises. The programs are kept purposely simple, so that the student does not have to spend a lot of time on a lengthy program and many details not having to do with concurrency. The only real issues in the programs have to do with the concurrency.

Why Use Pascal-S?

Pascal-S is a good vehicle for teaching these concepts for two main reasons. First, Pascal-S is extremely easy for students to learn and use if they already know Pascal or any Pascal-like language. It is on one hand a subset of Pascal, since it is identical to Pascal but doesn't contain many advanced features such as external files, linked lists, records, etc. On the other hand, it's an extension of Pascal since it includes the cobegin-coend construct for creating processes, and semaphores (wait and signal) for synchronization. Since there is no real teaching of the programming language involved, other than for the concurrency features, valuable classroom time does not have to be tied up learning a language. This is important since we usually have a hard enough time cramming in all the topics which must be included in Operating Systems I. Trying to spend time teaching a language would be prohibitive.

Second, Pascal-S is free and available to almost anyone. A program listing for the compiler and interpreter is included in the text, Principles of Concurrent Programming, by M. Ben-Ari [1]. We are using this language, adapted for the VAX 785 running under the UNIX operation system. Since small colleges typically have small budgets, buying an expensive compiler for a concurrent language for the sake of a few programming projects may be out of the question. This is a reasonable solution.

Pascal-S

Pascal-S in the Ben-Ari text is a "simplification and modification" of the original Pascal-S interpreter written by Nicholas Wirth. It works in two steps. The first step compiles source code into P-code (pseudocode); the second interprets the P-code module. Interpretation is accomplished using random numbers to simulate concurrency; i.e. random numbers determine which process will be scheduled to execute next, and each process executes for a random number of P-code statements before control is turned over to another. Thus a form of non-determinism is achieved; however, the program execution is repeatable as long as the interpreter uses the same sequence of random numbers each time. This makes debugging programs much easier.

As mentioned earlier, Pascal-S is a subset of Pascal and does not have many of the advanced features which we have become dependent on. For that reason, it is often necessary to keep programs simple in order to make them workable. Due to the fact that external files are not a part of the language, all input on our system must be done either interactively from the terminal, or by using redirected standard input under UNIX. A combination of the two is not possible. Records and pointers are not allowed; therefore, data structures must kept simple. The only data types are integer, boolean, and arrays; there are no reals, scalar, or subrange types. Random numbers, used in simulating delay times, must be generated as integers.

Other Languages

There is what appears to be a similar version of Pascal-S which was created for the IBM-PC family by Charles Schoening. The language, called Co-Pascal, is in the public domain. It also uses the cobegin-coend construct for concurrency. [5]

Programming Assignments

Four labs are presented below. I always start with lab one because it's the easiest to understand. I then assign anywhere from one to all of the rest as time permits. In addition, there is a wealth of ideas for other projects in the references listed below.

Lab One

The first lab assigned is the classic reporter/observer problem. The observer counts some occurrence as it happens; in this case, lines being entered from a terminal. The reporter periodically reports (prints) the results of the observer's counting; i.e., the number of lines entered since the last time of reporting. The count is then set back to zero. The count is a global variable which both processes are accessing, and which is the critical resource of the program. The assignment is done in two parts. Part one uses one reporter and one observer, and mutual exclusion is provided using Peterson's algorithm [6]. Part two of the assignment involves four observers and one reporter; i.e. four terminals are counting lines being entered using one global variable. The count reported is then the total lines entered from all terminals.

Part Two has been done in two different ways. The first is to continue using Peterson's algorithm for mutual exclusion, which brings home the point that this algorithm is difficult to generalize to more than two processes. More recently, the use of Lamport's bakery algorithm has been specified for this part [6] [1]. It involves real thinking on the part of the student to correctly interpret and translate the algorithm into code.

In doing this assignment, students quickly find out that the count is not the only critical resource involved in the program. The "printer" can also be considered a critical resource, and if a process does not have mutual exclusion when writing, the output of the two processes becomes interleaved, as shown below.

Output should be:
Number of lines counted = 6
Counted a line

Interleaved output:
Number of lines counted = Counted a line
6

In this program, we can avoid that by putting all write statements inside the critical section. In later programs, it is necessary to define a separate semaphore to control printing.

Code for Part Two of this assignment is shown in Appendix A. The procedure clock, used in conjunction with procedure sleep, is a way of simulating a system clock. It starts a timing mechanism only when timing is requested through a call to procedure sleep. This is an approximation to the "busy" statement included in the concurrent language CSP/k [4]. The coding for procedures clock and sleep are given to the student and discussed in class.

As you can see, the coding for this lab is very minimal, but it succeeds in acquainting the students with the very new concept of writing a program which has more than one process executing at once, and with the concept of global variables and the protection that must be given to them.

Students quickly find out after completing this assignment that just because the output of a concurrent program looks good, it doesn't mean that the program is written correctly! The mutual exclusion procedure is often incorrectly written and scenarios can be constructed and worked out by hand which would point out incorrect execution, yet the program appears to execute correctly, at least if it is run for a relatively short amount of time. Even after a lengthy run showing correct output, we still can't guarantee that the program is correct, based only on the output. Often the "right" combination of events hasn't happened as yet, which would make the error obvious. This observation is fascinating to the students, and heightens their interest in analyzing their code. (This is a good time to point out the desirability of proving the correctness of algorithms.)

Lab Two

The second lab is a variation on a producer/consumer problem originally posed by Per Brinch Hansen [2]. It refers to a system which has a fixed number of buffers being shared by three processes: a Reader, a Translator, and a Printer. The Reader process has a never-ending supply of data to read and store into an available buffer. The Translator removes the data from a waiting buffer, does some transformation on the data and writes it back into a buffer to be printed. The Printer takes any buffer directed to it and prints the contents. The flow of data is as follows:

READER-->INPUT BUFFER-->TRANSLATOR-->OUTPUT BUFFER--> PRINTER

We can make the following assumptions:

1. There will always be data for the Reader process to read, and the Reader will eventually input all waiting data.

2. As long as there is data being supplied to it, the Translator will eventually consume the data and produce output for the Printer.

3. As long as there is data being supplied to it, the Printer will eventually consume it.

To simplify programming of this problem, we do not actually create and use buffers, although it could be done if you wanted to expand the problem. We simply start with five buffers, and keep track of how many are in use and how many are free. Coordination between the processes is done with semaphores.

This lab is also done in two parts. Part one is done with no restraints put on any of the processes as far as the number of buffers they may use (other than a maximum of

five), so that deadlock will eventually result, and the students must analyze why the deadlock occurred (the Reader takes all the buffers, the Translator has data ready to send to the Printer but can't get a buffer, and the Printer is waiting for a buffer). In part two, the students have to come up with some restraint to put on the processes which will avoid deadlock and incorporate that into their program (the Reader cannot have more than four buffers in use at one time). The code for the three processes for Part two is shown in Appendix B.

This assignment is full of pitfalls! Students find that their programs for Part one deadlock and they assume that everything is as it should be. However, often the program has deadlocked because of some programming error which they don't see. On the other hand, some programs don't deadlock due to a programming error, so the student assumes he has magically found the correct solution to part two without really knowing what it is. It is extremely helpful to the students to have some discussion on this problem before the assignment is due to help them get on the right track. This assignment tends to be somewhat difficult for the instructor to grade, since the program has to be studied in detail to determine its correctness, as opposed to simply looking at the output. (This is generally true for concurrent programs.) In spite of the pitfalls, however, the students find this assignment very interesting and helpful in understanding deadlock and the interactions involved in a producer/consumer relationship.

Lab Three

Lab three is another producer/consumer problem: The Sleeping Barber [3]. This is the simulation of a barbershop which has two rooms: a waiting room and a room containing the barber chair (Figure 1). There is a sliding door which allows a customer to enter either the waiting room or the barber's room, and customers must enter one at a time through the narrow doorway, therefore the order of entry is defined. When the barber finishes a haircut, he looks in the waiting room. If there are no customers, he goes to sleep in the waiting room, and the next customer to come in wakes him up. If there are customers waiting, they are taken in turn. After their haircut, the customers leave through the back door.

The first solution to this problem (barber and customer processes) is shown in Appendix C, and uses Dijkstra's method of keeping a count of the number of customers waiting (i.e. number of buffers filled by the producer). The barber (the consumer) always decrements the number of customers as his first action. If the count becomes -1, that means there were no customers waiting, and he can go to sleep. An entering customer also uses this information in that if the count becomes zero after it is incremented, it means that he is the first customer to enter the shop in a while and the barber must be awakened. Otherwise, the barber is working and will eventually find the customer without a wakeup call. This method accomplishes the producer/consumer without having a wait/signal for each transaction in the buffer.

There is a problem, however, with this simulation, in that by definition of the semaphore, we have no control over which waiting process is awakened by an incoming signal. In particular, in Pascal-S, if there is more than one process waiting on a semaphore, the awakened process is chosen randomly from among them. This leads to some strange results in this program, and we find customers pushing ahead of others to get their haircuts first. This in itself is an interesting result for students to observe, and the lab can be done in this way and left as such.

However, it led me to redefine the problem as follows (Figure 2). We still have the same barber shop, but now the waiting room is very small and can only hold two customers. Any customers in excess of this must wait outside the door and fight for first place in line. There are only two chairs in the waiting room. If a customer enters and finds the barber sleeping, he wakes him up and moves directly into the barber's room. If the customer finds that he is alone in the waiting room but there is another customer in the barber chair, he sits down in chair two and waits to be called in by the barber. If a customer enters and finds another customer already sitting in chair two, he sits down in chair one. When the first customer goes in for his haircut, the customer from chair one

moves up to chair two, and in this way they keep track of who entered first. The barber and customer processes are shown as version two of the sleeping barber in Appendix C.

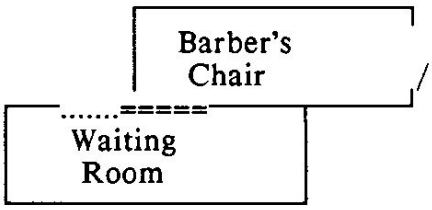

Figure 1

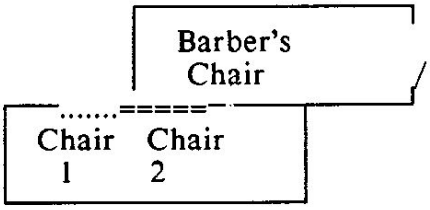

Figure 2

Lab Four

Monitors are an interesting concept but very difficult for students to master unless they can work with them. Pascal-S does not implement monitors, but they can be simulated by using semaphores [1], the exception being that the waiting queues of the monitors will not actually be queues. The waiting queues of the monitor are implemented by using a semaphore wait, which as discussed above does not give us any control over the order in which the processes are removed from the wait state.

The monitor simulated in Lab four controls the acquisition and release of the tape drive resources on a system. The program assumes that no process requests more than one drive, and it must be returned in a finite amount of time. The order that processes come out of the wait state is not a problem in this program since we do not keep track of the order in which the processes request a drive.

This again is a relatively simple program once the implementation is understood. It has possibilities, also, of being expanded by allowing a process to request more than one resource, and adding a deadlock avoidance algorithm. The coding for this problem in shown in Appendix D.

Conclusion

I have found that use of some programming in the Operating Systems course greatly enhances the student's understanding of how concurrent processes interact. They feel a great accomplishment in learning how to do something a little different, while using their known Pascal skills, and many students have actually thanked me for the opportunity. In fact, one student came back and told me how a prospective employer was quite impressed to find that he had some experience in this area, however small.

References

[1] Ben-Ari, M. Principles of Concurrent Programming, Prentice-Hall, 1981.

[2] Brinch Hansen, P. Operating System Principles, Prentice-Hall, Inc., 1973, p. 297.

[3] Dijkstra, E. W. "Cooperating Sequential Processes" in Programming Languages (F. Genuys, Editor), Academic Press, 1968.

[4] Holt, R.C., Graham, G. S., Lazowska, E. D., and Scott, M. A. Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company, 1978.

[5] Schoening, C. B. "Concurrent Programming in Co-Pascal," Computer Language, September 1986, 32-37.

[6] Silberschatz, A., and Peterson, J. L. Operating System Concepts, Alternate Edition, Addison-Wesley Publishing Company, 1988.

Appendix A - Reporter/Observer

```
program countlines;
const
   REP = 0;
   T1 = 1;
   T2 = 2;
   T3 = 3;
   T4 = 4;
   REPTIME = 350;
   N1 = 4;          {processes 0 to 4}
type
   nbrarray= array [0..N1] of integer;
   choosarr = array [0..N1] of boolean;
var
   time : array[0..N1] of integer;  {amt of time for sleep delay}
   i : integer;
   count : integer;          {nbr of lines counted--CRITICAL RESOURCE}
   choosing : choosarr;      {is process i choosing a nbr? }
   number : nbrarray;        {process's ticket number}
   seed : integer;       {for random number generator}

function rand (var seed : integer) : integer;
begin
   seed := (25173 * seed + 13849) mod 65536;
   rand := seed div 655  {get rn's between 1 and 100}
end;

function max (n : nbrarray): integer; {find max of ticket numbers}
var j , largest : integer;
begin
   largest := 0;
   for j := 0 to N1 do
        if number[j] > largest then
           largest := number[j];
   max := largest;
end; {max}

procedure mutexbegin (i : integer);  {Lamport's Bakery algorithm}
var j : integer;
begin
   choosing[i] := true;
   number[i] := max(number) + 1;
   choosing [i] := false;
   for j := 0 to N1 do
        begin
        while choosing[j] do;
        while ((number[j] <> 0) and
              ((number[j]<number[i]) or
              ((number[j]=number[i]) and (j<i)))) do;
        end
end; {mutexbegin}

procedure mutexend (i : integer);
begin
   number[i] := 0;
end; {mutexend}

procedure clock;
var i : integer;
begin
   while true do
```

```
      for i := 0 to N1 do
         if time[i] > 0 then
            time[i] := time[i] - 1;
end; {clock}

procedure sleep (who : integer);   {simulated delay}
var
   i, terminal1time : integer;
begin
   if who = REP then
        begin
        time[who] := REPTIME;
        while time[who] > 0 do;
        end
   else
        begin
        time[who] := rand(seed);
        while time[who] > 0 do;
        end;
end; {sleep}

procedure reporter;
begin
   while true do
        begin
        sleep (REP);                {delay for a while}
        mutexbegin (REP);
        writeln ('Number of lines counted =', count);
        count := 0;
        mutexend (REP);
        end;
end;  {reporter}

procedure terminal(who:integer);
begin
   while true do
        begin
        sleep (who);
        mutexbegin (who);
        count := count + 1;
        writeln ('Terminal ',who,' entered a line');
        mutexend (who);
        end;
end; {terminal}

begin {main}
   writeln;
   writeln ('Enter seed for program random number generator');
   readln (seed);
   count := 0;              {the critical resource - count of lines}
   for i := 0 to N1 do
        begin
        number[i] := 0;   {all numbers are initially 0}
        choosing[i] := false;       {initially no one is choosing}
        end;
   writeln;
   writeln ('Start');
   cobegin
        reporter;
        terminal(T1);
        terminal(T2);
```

```
        terminal(T3);
        terminal(T4);
        clock;
    coend
end.
```

Appendix B - Producer/Consumer

```
program buffers;
const
    RCODE = 1;
    TCODE = 2;
    PCODE = 3;
var
    msgtot : semaphore[0];
    msgtop : semaphore[0];
    mutex : semaphore[1];
    free : integer;
...
```

{Procedures clock, sleep and printtable go here. Clock and sleep are similar to that shown in Appendix A. Printtable prints a tally of the number of free and taken buffers, and tells which process just took a buffer or freed a buffer.}

```
procedure takefreebuffer (who : integer);
var
    tryagain : boolean;
begin
    tryagain := true;
    while tryagain do
        begin
        wait (mutex);
        if ((free = 0) or ((free = 1) and (who=1))) then
            tryagain := true
        else
            begin
            free := free -1;
            tryagain := false;
            printtable(who,free);
            end;
        signal (mutex);
        end;
end;

procedure freebuffer (who : integer);
begin
    wait(mutex);
    free:= free + 1;
    printfree(who,free);
    signal (mutex);
end;

procedure Reader;
begin
    while true do
        begin
        takefreebuffer (RCODE);
        writeln ('Reader getting data');
        sleep;                          {delay while data is read}
        writeln ('Reader giving buffer to Translator');
        signal (msgtot);
```

```
        end;
end;

procedure Translator;
begin
    while true do
        begin
        wait (msgtot);
        writeln ('Translator takes buffer');
        freebuffer (TCODE);
        sleep;                  {delay while data is transformed}
        takefreebuffer (TCODE);
        writeln ('Translator giving buffer to printer');
        signal (msgtop);
        end;
end;

procedure Printer;
begin
    while true do
        begin
        wait (msgtop);
        writeln ('Print a buffer');
        sleep;                  {delay while data is printed}
        writeln ('Finished printing');
        freebuffer (PCODE);
        end
end;

begin {main}
    free := 5;          {all buffers are initially free}
    cobegin
        Reader;
        Translator;
        Printer;
       clock
    coend;
end.
```

Appendix C
The Sleeping Barber Version One

```
program barbershop;
var
    mutex, mutexprint : semaphore[1];
    barbersleeping, haircutfinished : semaphore[0];
...
```

{Note: Procedure delay in this program is similar to procedure sleep as shown above. The name was changed in this program so as not to confuse the delay with the barber's sleep.}

```
procedure customer (i : integer);
begin
    while true do
        begin
        delay(i);           {delay as customer walks to the shop}
        wait (mutexprint);
        writeln ('Customer',i,' about to enter shop.');
        signal (mutexprint);
        wait(mutex);
        nbrcustomers := nbrcustomers + 1;
        if nbrcustomers = 0 then     {barber is asleep so ..}
```

```
            signal(barbersleeping);        { wake the barber}
        signal(mutex);
          wait(haircutfinished);
        wait (mutexprint);
          writeln ('Customer',i,' leaving barbershop.');
        signal (mutexprint)
          end;
end;

procedure barber;
begin
   while true do
        begin
        wait(mutex);
        nbrcustomers := nbrcustomers - 1;
        if nbrcustomers = -1 then        {shop empty - go to sleep}
          begin
          signal (mutex);
            wait (barbersleeping);
            end
       else
          signal (mutex);
        delay(BARB);                     {give the haircut}
        writeln('Barber finished a haircut');
        signal(haircutfinished);
        end;
end;

begin{main}
   nbrcustomers := 0;
   cobegin
        clock;
        barber;
        customer(1);
        customer(2);
        customer(3);
        customer(4);
   coend
end.
```

The Sleeping Barber Version Two

```
program barbershop;
var
   print, mutex, chair1, chair2 : semaphore[1];
   barberready, sleep, haircutcomplete: semaphore[0];
   waitingroom : semaphore[2];
...
procedure customer (i : integer);
begin
   while true do
     begin
     delay(i);              {let hair grow}
     wait (waitingroom);{wait outside door until you can get in}
     wait (mutex);
     nbrcustomers := nbrcustomers + 1;
     if nbrcustomers = 0 then   {there are no other customers}
        begin
        signal (mutex);
        wait (print);
```

```
      writeln ('Wake up barber!!');
      signal (print);
      signal(sleep);     {wake up the barber}
      end
   else if nbrcustomers = 1 then       {only customer in waiting room}
      begin
      signal (mutex);
      wait (chair2);    {sit in chair nearest barber's door}
      wait (print);
      writeln ('Customer',i,' in chair 2');
      signal(print);
      wait (barberready); {wait until barber finishes haircut}
      signal (chair2);  {let the next guy move up}
      end
   else if nbrcustomers = 2 then       {there is a cust in chair 2}
      begin
      signal (mutex);
      wait (chair1);              {sit down in chair 1}
      wait(print);
      writeln ('Customer',i,' sits in chair 1');
      signal(print);
      wait (chair2); {move to chair 2 when it's occupant leaves}
      signal (chair1);  {let the next guy sit down}
      wait(print);
      writeln ('Customer',i,' moved up to chair 2');
      signal(print);
      wait(barberready);       {wait until barber finishes haircut}
      signal (chair2);  {let the next guy move up}
      end;
   signal (waitingroom);       {leave the waiting room }
   wait(print);
   writeln ('Customer',i,' getting haircut.');
   signal(print);
   wait (haircutcomplete);
   wait(print);
   writeln ('Customer',i,' leaving shop.');
   signal(print);
   end;
d;

ocedure barber;
gin
 while true do
   begin
   wait(mutex);
   nbrcustomers := nbrcustomers - 1;
   if nbrcustomers = -1 then  {shop empty - go to sleep}
      begin
     signal (mutex);
      wait (print);
      writeln ('Barber going to sleep.');
      signal (print);
      wait(sleep);              {go to sleep}
      end
   else
     begin
     signal (mutex);
      signal (barberready);     {invite next customer in}
     end;
   wait(print);
```

```
      writeln ('Starting haircut');
      signal(print);
      delay(BARB);                {give the haircut}
      wait(print);
      writeln('Haircut complete.');
      signal(print);
      signal(haircutcomplete);
      end;
end;

begin {main}
   nbrcustomers := 0;
   cobegin
        clock;
        barber;
        customer(1);
        customer(2);
        customer(3);
        customer(4)
   coend
end.
```

Appendix D

```
program tapedrives;
var   {only the monitor variables are shown}
   mutex : semaphore[1];{for mutual excl. in the monitor}
   noneavailable : semaphore[0]; {queue of waiting processes}
   nbrdrives : integer;                    {nbr of drives available now}
   nbrwaiting : integer;  {nbr of processes waiting for a drive}
...
                  {   THE MONITOR    }
function acquire : integer;
begin
      wait(mutex);        {make sure no other ps is in the monitor}
      if nbrdrives = 0 then       {if there are no drives available..}
            begin
            nbrwaiting:= nbrwaiting + 1; {incr count of ps's waiting}
            signal(mutex);                {let another ps in monitor}
            wait(noneavail);              {join the "queue"}
                                 {a signal is received - drive is avail}
            nbrwaiting := nbrwaiting - 1;
            end;
      nbrdrives := nbrdrives-1;  {decrement nbr of available drives}
      acquire :=getdrivenbr; {assign a drive number to this process}
      signal(mutex);                {let another ps into monitor}
end;

procedure release(drivenbr: integer);
begin
      wait(mutex);                  {make sure no other ps is in monitor}
      nbrdrives := nbrdrives + 1; {release the drive}
      releasedrive (drivenbr);
      if nbrwaiting > 0 then    {if a process is waiting on the Q}
            signal(noneavail)     {signal the ps to get it off the Q}
      else                          {else just exit monitor}
            signal(mutex);
end;
                  {   END OF MONITOR    }
```

```
procedure use(i:integer); {i is the process number}
var
     driveassigned : integer;
begin
     while true do
          begin
          delay1;          {wait for a drive to be needed}
          driveassigned := acquire;
          wait (mutexprint);
          writeln ('Process',i,' has drive',driveassigned);
          signal (mutexprint);
          delay2;                    {use the drive}
          release(driveassigned);
          wait (mutexprint);
          writeln ('Process',i,' released drive',driveassigned);
          signal(mutexprint);
          end;
end;

begin {main}
...
     cobegin
          use(1);
          use(2);
          use(3);
          use(4);
          use(5);
         clock
     coend;
end.
```