TEACHING OPERATING SYSTEMS WITH MODULA-2

Jeffrey A. Brumfield

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-13 July 1985

Abstract. Programming projects in an operating systems course expose students to issues involved in the design and implementation of operating systems. The Modula-2 language provides features needed in such projects. This paper overviews the capabilities of Modula-2 and describes a programming project in which students implement a process manager for an operating system. The process manager supports concurrent processes and provides operations for their synchronization and communication.

1. INTRODUCTION

A course in operating systems has become an important part of an undergraduate computer science curriculum. Many schools now require that all computer science majors complete an operating systems course as part of their upper division course work.

Ideally, an operating systems course not only familiarizes students with the important concepts but also gives them an appreciation of the issues involved in the design and implementation of an operating system. There seems to be widespread agreement on the concepts that should be covered in an operating systems course. This is evident from the similarities in the tables of contents of recent operating systems texts [e.g., 2 and 4]. There is more diversity in how students are exposed to design and implementation issues. Most texts provide an inadequate treatment of these topics.

Programming projects give students the opportunity to apply ideas discussed in class and to deal with implementation issues that may have been omitted from lectures. Two major problems are faced by the instructor who wants to assign programming projects in an operating systems course.

The first problem is devising challenging yet tractable projects. Certainly, implementing several components of an operating system would be valuable experience for the students. But, the time involved may be prohibitive even if students are allowed to work in teams. Such an ambitious project may be better handled in a separate labora-

tory course or in an advanced course on operating systems. Many interesting projects require access to single-user systems on which a student's program can interact directly with the hardware. Limited computing resources may preclude this type of project at some schools.

The second problem is choosing a programming language with which the students will work. Most students taking an operating systems course will be experienced Pascal programmers. Unfortunately, Pascal is not well suited to operating systems applications. While several languages contain the necessary facilities, it is not reasonable to expect the students to learn an entirely new programming language at the beginning of the course.

This paper will discuss the use of the Modula-2 language in teaching operating systems. We believe that Modula-2 provides the needed features and can be learned easily by students familiar with Pascal. A sample programming project is presented that involves the implementation of a process manager. The assignment does not require single-user systems or any special software environment.

2. THE LIMITATIONS OF PASCAL

When selecting a language for use in an operating systems course, Pascal will likely be considered because of its widespread success in universities. Pascal has proved to be an ideal language for use in introductory programming courses. It encourages dis-

ciplined programming using a limited number of language features. Unfortunately, Pascal does not provide several capabilities that might be needed in an operating systems course.

Pascal does not support data abstraction or separate compilation. This makes the implementation of large and complex software systems difficult. The Pascal language does not include primitives for concurrent (or pseudo-concurrent) programming. Concurrent processes are the building blocks in many operating systems and represent one of the fundamental concepts taught in operating systems courses. When designing operating systems components, it is sometimes necessary to include code that is dependent on hardware characteristics such as the word size or the special use of certain memory locations by the hardware. The ability to write hardware dependent code was not among the design goals of Pascal.

There are several languages that provide the facilities that Pascal lacks. Among them are Ada, C, Modula-2, and various extended versions of Pascal. It is beyond the scope of this paper to compare and contrast these languages. We simply note that our choice of Modula-2 was motivated by its simplicity, elegance, and similarity to Pascal.

3. THE MODULA-2 LANGUAGE

Modula-2 [5] was invented by Niklaus Wirth, the same person who designed the Pascal language. One of the advantages of Modula-2 is that it can be learned quickly and easily by students who already know Pascal. A large subset of Modula-2 is almost identical to Pascal. There have been a few changes in syntax to make Modula-2 more consistent, and therefore more easily learned by beginning programmers. For example, every language structure that begins with a keyword also ends with a keyword. The FOR, WHILE, and IF statements are always terminated by the word END, whether they control a single statement or a compound statement.

A number of the restrictions of Pascal have been relaxed. An identifier may be referenced before it is declared, allowing declarations to appear in any order. An expression involving constants may appear anywhere a constant is allowed.

The most significant features of Modula-2 are those that extend Pascal. Modula-2 allows data objects and procedures to be grouped together into *modules*. A module is the fundamental unit for program decomposition and separate compilation. A separately compiled module consists of a *definition module*, which includes declarations of the objects and operations visible to the user, and an *implementation module*, which contains the executable statements for the operations. The only objects declared within a module that can be accessed outside the module are those whose names are specified in an *export list*. Similarly, objects declared outside a module must be named in an *import list* to accessible within the module. These facilities allow modules to be used to

implement abstract data types.

In Pascal, the primary unit of program execution is the procedure. In addition to procedures, Modula-2 also provides *coroutines*[*]. Coroutines differ from procedures in the flow of control and the allocation of variables. When control is transferred to a coroutine, execution begins at the statement following the last one executed by that coroutine. When a coroutine transfers control, it always specifies which coroutine will receive control next. Although a coroutine begins execution in a procedure specified when the coroutine is created, the coroutine may call other procedures. Each coroutine has its own set of local variables. Therefore, two coroutines may execute within the same procedure without interfering.

Modula-2 also provides low-level facilities that allow the programmer to subvert type checking and to write machine dependent programs. Such facilities might be needed when writing memory management routines or accessing memory locations used for specific purposes by the hardware.

Modula-2 is not without some drawbacks. The most noticeable of these is the lack of a standard set of powerful input/output operations. To keep Modula-2 simple, input/output procedures are not included as part of the language. Instead, they must be provided by library modules. Wirth has defined a rather primitive set of input/output operations that are included with many compilers. Other compiler imple-

[*]Coroutines are incorrectly called processes in the Modula-2 language. We will use the correct terminology throughout this paper.

mentors have chosen to provide libraries with input/output operations patterned after those in other languages. Until a powerful set of input/output operations becomes available with all compilers, it will be difficult to write completely portable Modula-2 programs.

4. CONCURRENT PROGRAMMING

The Modula-2 language does not directly support concurrent processes. However, coroutines can be used to implement pseudo-concurrency. The design of a library module to support concurrent processes closely parallels the design of a process manager for an operating system.

A process manager typically provides three types of abstract objects: processes, semaphores, and message links. Processes are the fundamental unit of work in the system. Semaphores are used to synchronize the execution of processes; message links are used for inter-process communication.

Figure 1 shows a definition module specifying one possible interface for a process manager. There exist operations for dynamically creating and destroying each type of object. The Suspend operation prevents a process from being executed; Resume allows the process to again compete for service. Terminate provides a simple way for a process to destroy itself. The semaphore operations Wait and Signal are identical to Dijkstra's P and V operations. ReceiveMsg and SendMsg provide the same synchronization as

```
DEFINITION MODULE ProcessManager ;

EXPORT QUALIFIED
                process, semaphore, link,
                CreateProcess, Suspend, Resume, Terminate, DestroyProcess,
                CreateSem, Wait, Signal, DestroySem,
                CreateLink, SendMsg, ReceiveMsg, DestroyLink;

TYPE
                process; semaphore; link;

PROCEDURE   CreateProcess   (e : PROC; wspsize : CARDINAL) : process ;
PROCEDURE   Suspend         (p : process) ;
PROCEDURE   Resume          (p : process) ;
PROCEDURE   Terminate       ( ) ;
PROCEDURE   DestroyProcess  (p : process) ;

PROCEDURE   CreateSem       (InitCount : CARDINAL) : semaphore ;
PROCEDURE   Wait            (s : semaphore) ;
PROCEDURE   Signal          (s : semaphore) ;
PROCEDURE   DestroySem      (s : semaphore) ;

PROCEDURE   CreateLink      (from, to : process) : link ;
PROCEDURE   SendMsg         (l : link; msg : ARRAY OF CHAR) ;
PROCEDURE   ReceiveMsg      (l : link; VAR msg : ARRAY OF CHAR) ;
PROCEDURE   DestroyLink     (l : link) ;

END ProcessManager .
```

Figure 1. An interface to a process manager.

Wait and Signal, and additionally allow a sequence of characters to be transmitted from one process to another. A complete specification of the workings of these operations depends upon a multitude of design decisions.

Using the objects and operations provided by the process manager, programs can be written to solve the classical problems involving concurrency. Figure 2 shows a solution to the bounded buffer producer-consumer problem that uses semaphores. In this problem, a producer process sends data to a consumer process via a shared buffer. The producer and consumer can theoretically operate at different (and possibly varying) speeds. So, the producer may sometimes have to wait for a free slot in the buffer, and the consumer may sometimes have to wait for a full slot. Additional synchronization problems are described in [1].

5. PROCESS MANAGER IMPLEMENTATION

Implementing a process manager gives students an understanding of concurrent processes, synchronization, and inter-process communication that is difficult to achieve through classroom lectures alone. Students see how each type of abstract object is realized using data structures, and how each operation is implemented as a procedure that manipulates these data structures.

```
MODULE BoundedBuffer;

FROM ProcessManager IMPORT process, semaphore, CreateProcess, Terminate,
                           CreateSem, Wait, Signal;
FROM InOut          IMPORT Read, Write;

CONST
   BufferSize  = 4;                              (* number slots in buffer  *)
TYPE
   BufferIndex = [0..BufferSize-1];              (* index into buffer       *)
VAR
   prod, cons  : process;                        (* producer and consumer   *)
   empty, full : semaphore;                      (* synchronization sems    *)

   buffer : ARRAY BufferIndex OF CHAR;           (* shared circular buffer  *)

(*-------------------------------------------------------------------------*)
(*  producer process - produces values and stores them in buffer           *)
(*-------------------------------------------------------------------------*)

PROCEDURE producer;
VAR
   i  : BufferIndex;                             (* producer buffer pointer *)
   ch : CHAR;                                    (* local storage for value *)
BEGIN
   i := 0;
   LOOP
      Read (ch);                                 (* produce a value         *)
      Wait (empty);                              (* wait for empty slot     *)
      buffer[i] := ch;                           (* store value in buffer   *)
      i := (i+1) MOD BufferSize;                 (* increment buffer ptr    *)
      Signal (full);                             (* signal full slot        *)
   END;
END producer;

(*-------------------------------------------------------------------------*)
(*  consumer process - retrieves values from buffer and consumes them      *)
(*-------------------------------------------------------------------------*)

PROCEDURE consumer;
VAR
   i  : BufferIndex;                             (* consumer buffer pointer *)
   ch : CHAR;                                    (* local storage for value *)
BEGIN
   i := 0;
   LOOP
      Wait (full);                               (* wait for full slot      *)
      ch := buffer[i];                           (* get value from buffer   *)
      i := (i+1) MOD BufferSize;                 (* increment buffer ptr    *)
      Signal (empty);                            (* signal empty slot       *)
      Write (ch);                                (* consume the value       *)
   END;
END consumer;
```

```
(*-------------------------------------------------------------------------*)
(*  main process - creates semaphores and other processes                  *)
(*-------------------------------------------------------------------------*)

BEGIN
   full  := CreateSem (0);
   empty := CreateSem (BufferSize);
   prod  := CreateProcess (producer, 1000);
   cons  := CreateProcess (consumer, 1000);
   Terminate;
END BoundedBuffer.
```

Figure 2. Solution to the bounded buffer problem.

The possible variations in the user interface and in the specifications and implementation of the operations are enormous. For this reason, we do not attempt to give a concise statement of a programming project in this paper. Instead, we discuss in this section some issues relevant to the design of any process manager. In the next section we describe more sophisticated features of process managers.

Before any operations are implemented, a preliminary format for the data structures must be decided. Each process, semaphore, and message link is represented by a record that can be allocated dynamically or from a static array of such records. The fields needed in these records depend upon the features supported. A process record (sometimes called a process control block) must include information about the coroutine associated with the process. It must also contain a link field so that processes can be put into lists. One such list is the ready list, which contains all processes that are eligible to be executed. A semaphore record must include a count and pointers to the head and tail of the list of processes waiting on the semaphore. Reference [3] provides more details about process manager data structures and operations.

Each operation that creates an object returns to the user a unique identifier for the object. This identifier is passed to the process manager each time the user wishes to manipulate the object. If storage for objects is allocated from arrays, an array index can be used to identify an object. If storage is allocated dynamically, a pointer to the internal representation of an object serves the same purpose. To prevent the user from directly modifying an object, the pointer type can be exported opaquely by the process

manager.

Operations that destroy objects are surprisingly difficult to implement. Decisions must be made regarding in which states an object can be destroyed. For example, destroying a semaphore on which processes are waiting may not be practical. When an object is destroyed, the state of other objects may need to be updated. For example, if a process waiting on a semaphore is destroyed, the internal representation of the semaphore must be adjusted.

Error detection and recovery is an important part of the process manager. Invalid operations, such as attempting to create a semaphore with a negative initial count, must be detected to ensure the consistency of the process manager data structures. The process manager can refuse to perform an invalid operation and return an error code to the requesting process. Alternatively, an error message can be written to an output file and the process can be terminated.

Debugging a process manager can be tedious. To facilitate this task, the process manager should include a trace facility that outputs descriptive messages when significant events occur. These messages should clearly identify the processes, semaphores, and links that are involved. Figure 3 lists some possible trace messages. Production of these messages could be enabled or disaled at compile time by selecting the appropriate value for a Boolean constant.

```
Process p creates process q
Process p made ready
Process p selected to run
Process p terminated
Process p creates semaphore s
Process p waits on semaphore s
Process p delayed on semaphore s
Process p signals semaphore s
Process p destroys semaphore s
```

Figure 3. Sample messages for tracing program execution.

The Appendix contains an implementation of a very simple process manager that supports only a subset of the operations shown in Figure 1. In this implementation, a process executes until it waits on a semaphore or terminates. The ready list and the semaphore lists are managed as FIFO queues. The procedures include no error checking and no output statements to trace execution. Although none of the operations require a process as input, the CreateProcess operation returns an identifier of the newly created process. This process manager is sufficiently complete to be used with the bounded buffer program in Figure 2. The code can be extended to include any of the features described in this paper.

6. ADVANCED FEATURES

Several features can be added to the process manager to make it more closely resemble the process manager in an actual operating system. These enhancements can be used in various combinations to create programming projects illustrating the desired concepts and being of appropriate difficulty.

Instead of all processes receiving equal service, a priority can be specified for each process at the time of its creation. The process manager should ensure that at all times the highest priority ready process is running. This will require that the ready list be maintained as a priority queue. Additionally, a process switch may be necessary in any of the routines in which a process is made ready.

Restrictions may be placed on the use of certain operations. For example, it may be unreasonable for any process to be able to destroy any other process in the system. One way of limiting access is to associate an owner with each object and to allow only the owner to perform certain operations. The owner may be the creator of the object, or ownership may be transferable.

A new object, called a private semaphore, can be added to the process manager. A private semaphore is a semaphore on which only one process can wait. Most implementations also specify that signals cannot accumulate on a private semaphore. Because of this, the internal representation of a private semaphore is much simpler than that of a general semaphore. Instead of dynamically allocating and deallocating private

semaphores, each process can have one or more private semaphores automatically allocated in its process control block when the process is created. Operations Pwait and Psignal can function in a way analogous to Wait and Signal.

Usually, if there are no processes eligible to execute, then all processes have terminated or a deadlock has occurred. In either case, execution of the entire program should halt. An exception to this rule occurs if a process can wait for an external event, such as the expiration of a timer or the occurrence of an I/O interrupt. In this case, a null process must be built into the process manager. The null process is selected only if no other process is ready. If process priorities are available, the null process can simply be an infinite loop process having the lowest priority. Otherwise, the null process must constantly check the ready list for the appearance of another process.

7. RANDOMIZING PROCESS EXECUTION

In a concurrent program, execution of the statements in the individual processes can be interleaved in any way, except as restricted by synchronization operations. In the process manager we have described, a process executes until it terminates or is delayed on a semaphore or message link. This severely limits the way in which processes interact. For example, in the bounded buffer program presented earlier, the produces always fills the entire buffer before the consumer begins execution. The consumer the empties the entire buffer before the producer receives control again. These restrictions may prevent flaws in the design of a concurrent program from being detected.

Variations in process execution can be introduced into the process manager in two ways. The first way is to time-slice the execution of processes. When a process begins execution, a timer is set to generate an interrupt at the end of the time slice. The interrupt handler forces a process switch. Because setting timers and handling interrupts are operating system dependent, we will not further elaborate on this idea.

An alternative is to maintain the collection of ready processes as an unordered set instead of a FIFO queue. Every time a process manager operation is invoked, a random number is generated to select a process to execute. Because the currently running process is among those processes eligible to execute, a process switch may not always be necessary. When the process manager is modified in this way, the producer (or consumer) in the bounded buffer example will fill (or empty) a random number of slots in the buffer each time it is executed.

8. DISCUSSION

Programming assignments involving process management can be incorporated into an operating systems course in several ways. Initially, a compiled version of a process manager can be provided to the students so they can learn to design concurrent programs. Students can be required to develop solutions to several of the classical problems using shared variables protected by semaphores and using message links. When the students thoroughly understand the use of processes, semaphores, and message links, they can be required to implement their own process manager.

When formulating a description of this project, the instructor must decide how rigid the specifications will be. Certainly, it is easier to evaluate a student's solution to an assignment that has very specific requirements. The definition module for the process manager can serve as a partial specification of the assignment. This allows the instructor to prepare one set of programs that can be used to test all students' process managers.

We believe that students learn more when they are given the opportunity to experiment with design alternatives. This may involve modifying the process manager interface and, consequently, the programs that use the process manager. To ensure that students do not miss any of the basic objectives of the project, we require all students to develop process managers that satisfy a set of minimal requirements before enhancing their programs with features they find interesting. Students are expected to completely document and thoroughly test all extensions to the basic process manager.

Acknowledgments

The design of the process manager presented in this paper was inspired by a design used by Peter Denning in operating systems courses at Purdue University. Modula-2 code appearing in this paper was tested using the DEC Western Research Laboratory compiler developed by Michael Powell.

References

1. M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, Englewood Cliffs, N.J., 1982.

2. H. M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Reading, Mass., 1984.

3. P. J. Denning, T. D. Dennis, and J. A. Brumfield, "Low Contention Semaphores and Ready Lists," *Communications of the ACM*, Vol. 24, No. 10, October 1981, pp. 687-699.

4. J. L. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison-Wesley Publishing Company, Reading, Mass., 1983.

5. N. Wirth, *Programming in Modula-2, Second Edition*, Springer-Verlag, Berlin, 1983.

APPENDIX

```
DEFINITION MODULE ProcessManager;

EXPORT QUALIFIED  process, semaphore,
                  CreateProcess, Terminate,
                  CreateSem, Wait, Signal, DestroySem;

TYPE
   process;
   semaphore;

PROCEDURE CreateProcess  (e : PROC; wspsize : CARDINAL) : process;
PROCEDURE Terminate      ( );
PROCEDURE CreateSem      (InitCount : CARDINAL) : semaphore;
PROCEDURE Wait           (s : semaphore);
PROCEDURE Signal         (s : semaphore);
PROCEDURE DestroySem     (s : semaphore);

END ProcessManager.



IMPLEMENTATION MODULE ProcessManager;

(*-------------------------------------------------------------------------*)
(*   In this minimal function process manager, a process executes until it  *)
(*   waits on a semaphore or terminates.  The ready list and semaphore      *)
(*   lists are managed as fifo queues.  Note that coroutine, newcoroutine,  *)
(*   and switch are aliases for PROCESS, NEWPROCESS, and TRANSFER.          *)
(*-------------------------------------------------------------------------*)

FROM coroutines IMPORT coroutine, newcoroutine, switch;
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM     IMPORT ADDRESS;

TYPE

   process   = POINTER TO RECORD
                  cortn   : coroutine;    (* coroutine variable for process *)
                  wsp     : ADDRESS;      (* workspace for coroutine        *)
                  wspsize : CARDINAL;     (* size of workspace in stg units *)
                  link    : process;      (* next process on list           *)
               END;

   semaphore = POINTER TO RECORD
                  count   : INTEGER;      (* neg value gives queue length   *)
                  head    : process;      (* first process on waiting queue *)
                  tail    : process;      (* last process on waiting queue  *)
               END;
```

```
VAR

   RL         : RECORD                          (* ready list                    *)
                   head : process;               (* first process on list        *)
                   tail : process;               (* last process on list         *)
                END;

   dummy      : coroutine;          (* for switching from a terminated process  *)
   self       : process;            (*  the currently running process           *)

(*-------------------------------------------------------------------------*)
(*   add process p to the tail of the ready list                           *)
(*-------------------------------------------------------------------------*)

PROCEDURE ready (p : process);
BEGIN
   IF RL.head = NIL THEN RL.head := p
   ELSE RL.tail↑.link := p
   END;
   p↑.link := NIL;
   RL.tail := p;
END ready;

(*-------------------------------------------------------------------------*)
(*   remove a process from the head of the ready list                      *)
(*-------------------------------------------------------------------------*)

PROCEDURE nextproc () : process;
VAR
   p : process;
BEGIN
   IF RL.head = NIL THEN HALT END;
   p := RL.head;
   RL.head := RL.head↑.link;
   p↑.link := NIL;
   RETURN p;
END nextproc;

(*-------------------------------------------------------------------------*)
(*   create a ready process having starting point at procedure e           *)
(*-------------------------------------------------------------------------*)

PROCEDURE CreateProcess (e : PROC; wspsize : CARDINAL) : process;
VAR
   p : process;
BEGIN
   NEW (p);                                  (* allocate storage for process *)
   ALLOCATE (p↑.wsp, wspsize);               (* allocate coroutine workspace *)
   p↑.wspsize := wspsize;
   newcoroutine (e, p↑.wsp, p↑.wspsize, p↑.cortn);      (* create coroutine *)
   ready (p);                                          (* make process ready *)
   RETURN p;
END CreateProcess;
```

```
(*-------------------------------------------------------------------------*)
(*   destroy the currently running process                                 *)
(*-------------------------------------------------------------------------*)

PROCEDURE Terminate ();
BEGIN
   IF self↑.wspsize <> 0 THEN DEALLOCATE (self↑.wsp, self↑.wspsize) END;
   DISPOSE (self);
   self := nextproc();                              (* get a ready process *)
   switch (dummy, self↑.cortn);                     (* and switch to it    *)
END Terminate;

(*-------------------------------------------------------------------------*)
(*   create a semaphore having the specified initial count                 *)
(*-------------------------------------------------------------------------*)

PROCEDURE CreateSem (InitCount : CARDINAL) : semaphore;
VAR
   s : semaphore;
BEGIN
   NEW (s);
   s↑.count := InitCount;
   s↑.head  := NIL;
   s↑.tail  := NIL;
   RETURN s;
END CreateSem;

(*-------------------------------------------------------------------------*)
(*   wait on semaphore s                                                   *)
(*-------------------------------------------------------------------------*)

PROCEDURE Wait (s : semaphore);
VAR
   previous : process;
BEGIN
   s↑.count := s↑.count - 1;
   IF s↑.count < 0 THEN
      IF s↑.head = NIL THEN s↑.head := self         (* attach running process *)
      ELSE s↑.tail↑.link := self                    (* to tail of sem queue   *)
      END;
      s↑.tail := self;
      self↑.link := NIL;
      previous := self;
      self := nextproc();                           (* get a ready process *)
      switch (previous↑.cortn, self↑.cortn);        (* and switch to it    *)
      END;
END Wait;

(*-------------------------------------------------------------------------*)
(*   signal semaphore s                                                    *)
(*-------------------------------------------------------------------------*)

PROCEDURE Signal (s : semaphore);
VAR
```

```
   p : process;
BEGIN
   s↑.count := s↑.count + 1;
   IF s↑.count <= 0 THEN
      p := s↑.head;                          (* remove a process from the *)
      s↑.head := s↑.head↑.link;              (* head of the sem queue     *)
      ready (p);                             (* and make it ready         *)
      END;
END Signal;

(*-------------------------------------------------------------------------*)
(*   destroy semaphore s if no processes are waiting on it                 *)
(*-------------------------------------------------------------------------*)

PROCEDURE DestroySem (s : semaphore);
BEGIN
   IF s↑.head = NIL THEN DISPOSE (s) END;
END DestroySem;

(*-------------------------------------------------------------------------*)
(*   initialization of process manager                                     *)
(*-------------------------------------------------------------------------*)

BEGIN  (* initialization *)
   RL.head := NIL;
   RL.tail := NIL;
   NEW (self);                           (* fake entry for main process *)
   self↑.wspsize := 0;
   self↑.link := NIL;
END ProcessManager.
```