THREE SURVEYS ON OPERATING SYSTEM TOPICS*

TR 80-425

May 1980

by

Gregory R. Andrews[1]
Fred B. Schneider[2]

Computer Science Department
Cornell University
Ithaca, New York 14853

---

[1]Department of Computer Science, University of Arizona, Tucson, Arizona.
[2]Department of Computer Science, Cornell University, Ithaca, New York.
*This report is also available as TR80-14 in the Computer Science Department, University of Arizona, Tucson, Arizona.

Preface

Recently, we were asked by the Wiley Publishing Company to write survey articles covering some of the important concepts in operating systems for their forthcoming Handbook of Electrical and Computer Engineering. The three brief articles that comprise this report are the results of those efforts. The first was written by Schneider, the latter two by Andrews. We have collected them here in the hope that some may find them of interest. The first article discusses concurrent programming, which is an underlying concern in most operating systems. The use of a kernel or nucleus in constructing an operating system is described in the second article. Lastly, in the third article the notion of hierarchical organization is developed and some of the advantages of this approach are presented.

Our thanks to Professor A. J. Bernstein of the editorial board for the Handbook for his comments on earlier drafts of these papers.

G. R. Andrews

F. B. Schneider

I. Synchronization and Concurrent Programming

1. Introduction

A sequential program consists of some variables and a sequentially executed list of statements; its execution is called a sequential process [4]. Alternatively, a program that involves execution of more than one statement at a time is called a non sequential process. Of particular interest are those programs that involve execution of more than one sequential process at a time. They are called concurrent programs. A concurrent program is often a useful way to formulate a computation. For example, an airline reservation system that involves processing transactions from many terminals would have a natural specification as a concurrent program--each terminal would be monitored by a sequential process.

Even when simultaneous execution of processes is not intended (or involved), it is often easier to structure a system as a collection of cooperating processes than as a single sequential program. This approach frequently finds application in operating systems and database management systems. For example, a simple operating system could be viewed in terms of three concurrent processes: a cardreader process, a job manager process and an output process. The cardreader process reads cards from the cardreader and places card images in an input buffer. The job manager process reads card images from the input buffer and processes them, generating line images that are stored in an output buffer. The output process obtains line images from the output buffer and writes them to the lineprinter. One advantage of this structure is that the effects of speed variations of each process can be damped, as long as the average rate that images are stored in a buffer is the same as the average rate they are removed. The amount of speed variation that

can be smoothed depends on the size of the buffer used. This sort or producer - consumer relationship appears frequently in concurrent programs.

One way to execute a concurrent program is to use a multiprocessor. Each process is executed on a separate processor, and any shared variables are kept in a store that is accessible to all processors. A more common way to execute a concurrent program is to time multiplex the concurrent processes on one (or more) processors. A process executes for some length of time on a processor. Then, the processor state (i.e., the values of the processor registers) is saved and another process is selected and executed. Whenever execution of a process is resumed, the state information for that process is restored to the appropriate processor registers. This creates the illusion that processes are executing on a "variable speed" processor, as opposed to being periodically suspended and resumed.

Coroutines provide a mechanism that allows the programmer explicit control over this process switching [9]. Recall that the subroutine call instruction always causes control to be transferred to the first instruction of the named routine. The coroutine resume instruction, like the call instruction, causes control to be transferred to the named routine. However, the resume instruction causes execution to commence at the instruction following the last resume instruction in that routine. This is illustrated in Figure 1.

Usually, process switching is not under control of the programmer, but is implemented by the kernel or lowest level of an operating system. Process switching may occur periodically -- in response to timer interrupts -- or randomly, in response to other events that cause the kernel to receive control. It may be assumed that each process will progress at some finite speed;

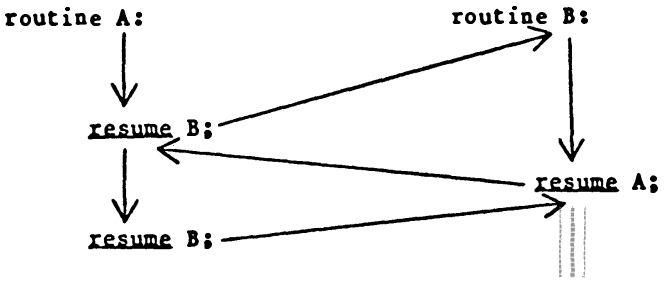


Figure 1 Coroutines

however, no assumption can be made about that speed and its relation to the speeds of other processes. Consequently, processes executing in this manner are referred to as asynchronous. Furthermore, since process switching may in general occur between any two instructions, the programmer may not assume that high level language statements are necessarily executed as indivisible actions.

2. Expressing Concurrency

Numerous language constructs have been proposed to allow specification of concurrent computations. The constructs differ primarily in the level of implementation detail to which the programmer is exposed. Three common constructs are described below.

fork/join [9]

A fork statement is like a call statement -- it specifies that execution of a designated routine should commence. However, in a fork statement execution of the statements following the fork (i.e., the invoking routine) and the invoked routine proceed concurrently. (In a call statement the caller is suspended until the invoked routine terminates.) For example in Figure 2, the fork statement causes concurrent execution of Program2 with the statements

following L0 in Program1.

The join statement causes the invoker to be suspended until the previously "forked" routine terminates. In Figure 2, Program1 can not progress beyond L1 until Program2 terminates.

```
Program1:  ...                          Program2:  ...
           ...                                     ...
           L0 : fork Program 2;                    ...
           ...                                     return;
           L1 : join;
           ...
```

Figure 2 fork/join

cobegin [2]

The cobegin statement is a structured way to denote the concurrent execution of a set of statements. Thus,

$$\underline{\text{cobegin}}\ S_1\ //\ S_2\ //\ \ldots\ //\ S_n\ \underline{\text{coend}}$$

results in an arbitrary interleaving of the execution of S_1, S_2, ..., S_n. Each of the S_i's may be any statement (including cobegin). Execution of cobegin terminates only when each of the S_i's has terminated. Execution of each S_i can be viewed as a process.

process [3]

Large programs are often structured as a collection of sequential routines (modules or subroutines): one routine initially receives control and invokes other routines, as required. A process declaration identifies a routine that executes concurrently with its caller. Activation of a process is accomplished by execution a start instruction. Using this, a concurrent program can be structured as a collection of processes, and a main routine that

initially receives control and starts these processes. For example, Figure 3 shows the program outline for the simple operating system discussed earlier in terms of processes.

```
main:  start cardreader_process;
       start job_manager_process;
       start output_process;

cardreader_process: process;
       var  card : cardimage;
       do  forever
         begin
           read card from cardreader;
           store card in input_buffer;
           end;
output_process: process;
       var  line : lineimage;
       do  forever
         begin
           obtain line from output_buffer;
           print line on lineprinter;
           end;
job_manager_process:  process;
       var  card : cardimage;
           line : lineimage;
       do  forever
         begin
           obtain card from input_buffer;
           process card generating line;
           store line in output_buffer;
           end;
         end;
       end;
```

Figure 3 A system composed of three processes

3. Synchronization

To cooperate in performing a task, processes need some means to communicate. Shared variables (variables accessible to more than one process) are often used for this purpose. However, certain problems may occur when execution of processes that access shared variables is interleaved in an arbitrary manner. To illustrate this, consider the following program.

```
(I)     A := a;
        cobegin
             R:  A  := A + 1;
        //
             Q:  A  := A + 1;
        coend
        Print(A)
```

Note that the statement "A := A + 1" might be implemented in machine language as follows:

```
Load    Reg,A
Add     Reg,"1"
Store   Reg, A
```

One possible interleaving that might result from execution of the cobegin statement in (I) is shown in Figure 4. In that description, "a" represents the initial value of variable A. Note that in this case, the final value of A is a+1, not a+2 as might be expected. Furthermore, this seemingly pathological behavior is purely a consequence of the interleaving of the Load instruction in process Q and the Store instruction in process R.

ProcessR:		ProcessQ:		Description:
Load	Reg1,A			Reg1=a
Add	Reg1,1			Reg1=a+1
		Load	Reg2,A	Reg2=a
		Add	Reg2,1	Reg2=a+1
		Store	Reg2,A	A=a+1
Store	Reg1,A			A=a+1

Figure 4 Interleaved Execution

Often, execution of a sequence of statements must be made indivisible. For example, the manipulation of a complex data structure may be in terms of operations that are implemented by sequences of statements. If processes concurrently perform operations on the same shared data object, then pathological results like those described earlier might be observed due to interlesving of the execution of those operations. A sequence of one or more statements that

should be executed as an indivisible action is called a critical section. The mutual exclusion problem is concerned with preventing the interleaving of processes executing in critical sections. Clearly, if all accesses to a given shared data structure are made from statements within critical sections, and execution of critical sections is mutually exclusive, then the integrity of shared data objects will be preserved.

Another situation where it is necessary to coordinate execution of concurrent processes occurs when a shared data object is not in a state conducive to execution of a particular operation. A process attempting such an operation should be delayed, since the state (i.e., value of the variables that comprise the object) of the data object may subsequently change, as a result of operations performed by other processes. This is called condition synchronization. An example of this appears in the simple operating system discussed earlier. In that system, the cardreader process must be delayed when it attempts to store a card image in the input buffer if all of the buffer frames are full. (This might happen if the cardreader process was reading cards faster than the job manager process was using them.) Thus, a process executing a store operation on a buffer must be delayed if there was no space in that buffer. Similarly, a process attempting to remove a card image from the buffer should be delayed if there is nothing in the buffer to remove.

Synchronization mechanisms are used to restrict the possible interleavings of concurrently executing processes. In the sequel, a number of common mechanisms will be discussed. They provide a convenient way for a programmer to regulate access to shared data. In particular, they facilitate the definition of critical sections and mutual exclusion, as well as provide ways to prescribe other constraints on interleavings in the execution of concurrent

programs. Note that it is possible to coordinate execution of concurrent processes solely by use of shared variables because of the atomic nature of a memory reference. Unfortunately, programs that use only shared variables to implement synchronization tend to be complex, hence difficult to understand and formally verify. Typically, these programs set and repeatedly test the value of a shared variable, waiting until that variable has a certain value. Waiting is accomplished by performing the test operation in a tight (short) loop. Thus, in order to delay itself, a process may cause the processor to busy wait (also known as spinning) -- execute instructions for no purpose other than to pass time. Clearly, this is wasteful of CPU cycles.

Semaphores

A semaphore is a data type that assumes non-negative integer values on which two operations are defined: P and V [4]. If s is a semaphore, then V(s) causes s to be incremented by one in an atomic action. P(s) causes s to be decremented by one in an atomic action, provided s will remain non-negative. If the decrement operation would cause s to become negative, then execution of the P operation is delayed. Semaphore implementations are expected to exhibit fairness. This means that no process delayed while executing a P operation on a semaphore s will remain suspended forever if V operations are performed on s by other processes. The notion of fairness is required since a number of processes may be simultaneously delayed, all attempting to execute a P operation on the same semaphore. Clearly, a choice exists as to which one will be allowed to proceed when a V is ultimately performed. A simple way to ensure fairness is to awaken processes in the order they were suspended, as V operations are performed.

Implementation of critical sections that mutually exclude each other in

execution is possible using semaphores. Each sequence of statements that is to be a critical section is preceded by a P operation on a semaphore and followed by a V operation on the same semaphore. All mutually exclusive critical sections use the same semaphore, which is initialized to one. Below, program (I) has been modified to ensure that statements R and Q are each executed as atomic actions.

```
(II)        S := 1;
            A := a;
            cobegin
                 P(S); R:  A  := A + 1; V(S)
            //
                 P(S); Q:  A  := A + 1; V(S)
            coend
            Print(A)
```

The pathological interleaving described earlier can not occur here because of the semantics of P and V. Thus, program (II) always establishes the truth of the assertion: A=a+2.

It is sometimes necessary to coordinate execution of a number of processes so that at most k processes can be executing in critical sections at any time. When k=1 this is the mutual exclusion problem, for which a solution has already been given. A solution for the case when k>1 can be obtained by changing the initialization of the mutual exclusion semaphore from 1 to k.

Semaphores can also be used to solve mutual exclusion problems when the critical sections are partitioned into sets, and execution must be constrained so that no two members of the same set are executed concurrently, although execution of critical sections in different sets can be interleaved. (Presumably, the critical sections in a given set all involve references to the same shared variables.) This is illustrated in Figure 5 where initially S1=S2=1. Execution of <CS2> and <CS3> is mutually exclusive. However, concurrent exe-

cution of <CS3> and <CS4> is permitted.

```
Program1: process;          Program2: process;          Program3: process;
            .                           .                           .
            .                           .                           .
            .                         P(S1);                        .
          P(S1);                       <CS3>                      P(S2);
          <CS1>                       V(S1);                      <CS4>
          V(S1);                        .                         V(S2);
            .                           .                           .
            .                           .                           .
            .                           .                           .
          P(S2);
          <CS2>
          V(S2);
            .
            .
```

Figure 5
A system of processes with multiple critical sections

Semaphores can be used to implement condition synchronization, as well as various types of mutual exclusion. V operations are used to signal occurences of events, and P operations to cause process execution to be delayed if an event has not yet occurred. This use of semaphores can be seen in Figure 6, a somewhat expanded version of the simple operating system illustrated in Figure 3.

```
var       input_mutex, output_mutex, no_cards, no_lines,
             free_input, free_output : semaphore;

main:     input_mutex := 1; output_mutex := 1;
          no_lines := 0; no_cards := 0;
          free_input := input_buffer_size;
          free_output := output_buffer_size;
          start cardreader_process;
          start job_manager_process;
          start output_process;

cardreader_process: process;
          var card : cardimage;
          do forever
             begin
```

```
            read card from cardreader;
            P(free_input);
            P(input_mutex);
               store card in input_buffer;
            V(input_mutex);
            V(no_cards);
            end;
          end;
output_process: process;
       var  line : lineimage;
       do  forever
          begin
            P(no_lines);
            P(output_mutex);
              obtain line from output_buffer;
            V(output_mutex);
            V(free_output);
            write line on lineprinter;
            end;
          end
job_manager_process: process;
       var  card  : cardimage;
            line  : lineimage;
       do  forever
            begin
              P(no_cards);
              P(input_mutex);
                obtain card from input_buffer;
              V(input_mutex);
              V(free_input)
              process card, generating line;
              P(free_output);
              P(output_mutex);
                store line in output_buffer;
              V(output_mutex);
              V(no_lines);
              end;
            end;
          end;
```

Figure 6 Synchronization using semaphores

The input_mutex and output_mutex semaphores are used to implement mutual exclusion of operations on input_buffer and output_buffer, respectively. However, an operation on input_buffer may proceed concurrently with an operation on output_buffer. The no_cards and no_lines semaphores are used to ensure that processes do not attempt to obtain values from empty buffer frames. V operations are performed on these semaphores after each store in a buffer, and

P operations prior to removal of information from the buffer. Lastly, the free_input and free_output semaphores are employed to delay a process attempting to store an item in a buffer, if that buffer is already full. It is assumed that the input_buffer is of size input_buffer_size, and the output_buffer of size output_buffer_size.

Semaphores can be implemented by using a test-and-set instruction, which is found on most modern day computers (for exactly this reason). The operation of the test-and-set instruction is as follows:

```
TS(x,y) :  As an atomic action do;
               x := y;
               y := true;
               end;
```

Usually, y is a shared variable, and x is local to the invoking process.

A binary semaphore is a semaphore that is always either 0 or 1. Using test-and-set, the operations on a binary semaphore P_b and V_B can be implemented as follows:

```
Pb(s): var  slocal : boolean;
       L:  TS(slocal,s);
           if  slocal then go to L;

Vb(s): s :=  false;
```

It is possible to implement a general semaphore sem (i.e., a semaphore with no upper bound on its value) by using an integer variable VALUE, a binary semaphore MUTEX for mutual exclusion, and a binary semaphore DELAY to delay process execution. MUTEX is initialized to 1; DELAY to 0; and VALUE to the initial value of semaphore sem.

```
P(sem): P_b(MUTEX);
        VALUE  :=  VALUE - 1;
        if  VALUE < 0 then  do  begin
              V_b(MUTEX);
              P_b(DELAY);
              end;
        else  V_b(MUTEX);

V(sem): P_b(MUTEX);
        VALUE  :=  VALUE + 1;
        if  VALUE ≤ 0 then  V_b (DELAY);
        V_b(MUTEX);
```

Notice that busy waiting is used to delay process execution.

In a single-processor system with a kernel, P and V operations can be implemented as kernel calls that do not involve busy waiting. Recall that the kernel implements the process abstraction. To do this, the kernel maintains a queue of all processes that are eligible to run on the processor. This queue is called the ready list. The kernel switches the processor among these processes. In addition, the kernel maintains a queue of the processes that are suspended on each semaphore. Processes are not executed while they are on these queues. Execution of a V operation moves a waiting process from the corresponding semaphore queue to the ready list, thereby making that process eligible for execution. Execution of a P operation may or may not remove the executing process from the ready list, depending on the value of the semaphore. Crucial to the correctness of this implementation is that the kernel not be interrupted while performing these operations. This is because the ready list and semaphore queues are shared data objects, and therefore arbitrary interleaving of concurrent accesses (by invoking processes and interrupt handlers) might yield pathological results. Thus, execution in the kernel occurs with interrupts masked off (disabled). Consequently, each kernel operation is atomic. Such a technique is not suitable in multiple processor systems, and busy waiting may be required to ensue the integrity of the ready

list and semaphore queues. This is because it may become necessary to delay a processor that is attempting to access one of these queues, if that queue is concurrently being manipulated by another processor. Furthermore, no other process could be executed by that processor because access to the ready list would be required to run such a process.

Monitors [3]

Although semaphores can be used to program any synchronization problem, P and V are rather unstructured primitives. It is therefore easy for a programmer to make coding errors when using semaphores. For example, the programmer might forget to include all statements that access shared variables in critical sections, or might perform operations on the wrong semaphore. Monitors provide a somewhat more structured synchronization mechanism. When using monitors, many of these problems then can be avoided, or detected by a compiler.

The monitor enforces a very disciplined use of shared variables. Consequently, it is easy to write concurrent programs using monitors, although the amount of concurrency in such programs may be somewhat restricted. A monitor is essentially an abstract data type or module extended for use in an asynchronous environment. It consists of a collection of permanent variables and a set of procedures, which are used to manipulate these variables. The values of the permanent variables are retained between activations of monitor procedures, and may be accessed only from within those procedures. The permanent variables are generally related, and they are referred to as a resource. For example, a buffer can be implemented in terms of an array and two pointer variables. These three variables comprise the buffer -- a resource. The syntax of a monitor is shown in Figure 7. Generally, each monitor procedure implements an operation on the resource defined by the permanent variables of

the monitor. A procedure is invoked by coding:

call <monitor_name>.<procedure_name> (...argument list...);

Execution of monitor procedures is guaranteed to be mutually exclusive. Thus, at most one process can be executing in a given monitor at any time. This ensures the integrity of the permanent variables in the monitor. Notice that a monitor is really just a grouping of all the critical sections that reference a set of shared variables.

```
<monitor_name>  : monitor;
      var  ... define permanent variables ...

      procedure entry <operation_1> (... parameters ...);
         var ... define local variables ...
         begin
               ... code to implement <operation_1> ...
         end;
      .
      .
      .
      procedure entry <operation_n> (... parameters ...);
         var  ... define local variables ...
         begin
               ... code to implement <operation_n> ...
         end;
      begin
         ... initialization ...
      end;
```

Figure 7 Monitor syntax

The use of monitors allows a programmer to ignore the implementation details of a resource when using it, as he need only be concerned with the monitor procedure interfaces. Similarly, the circumstances surrounding the use of the resource may be ignored when the monitor is being coded, as long as the implementation satisfies its interface specification.

Within a monitor, condition variables are used to delay a process when the values of the permanent variables of the monitor are not conducive to con-

tinued execution by that process. Two operations are defined on condition variables: send and wait. If cond is a condition variable, then cond.wait causes the invoker to be suspended and to relinquish control of the monitor. Cond.send causes the invoker to be suspended, and a process currently delayed on cond is reactivated. A process suspended due to a send operation is reactivated when control of the monitor has been relinquished. A new process is granted entry into a monitor procedure only if there are no processes delayed as a result of having performed a send operation and there is no process actively executing in the monitor.

Figure 8 is the operating system example programmed using monitors.

```
inbuff: monitor;
   var  no_cards   : integer;
        in_full, in_free   : condition;
        ... other variables to represent input_buffer ...

   procedure entry deposit ( c : cardimage);
        begin
          if  no_cards = input_buffer_size then  in_free.wait;
          ... code to store c in input_buffer ...
          no_cards := no_cards +1;
          in_full.send
          end;

   begin  (*initialization*)
        no_cards := 0;
        ... initialization of other variables ...
        end;

   outbuff: monitor;
        var  no lines : integer;
        out_full, out_free :  condition;
        ... other variables to represent output_buffer ...

   procedure entry deposit (c : lineimage);
        begin
          if  no_lines = output_buffer_size then  out_free.wait;
          ... code to store c in output_buffer ...
          no_lines := no_lines + 1;
          out_full.send;
          end;

procedure entry retrieve (var  c: lineimage);
```

```
  begin
    if  no_lines = 0 then  out_full.wait;
    ... code to remove a lineimage from output_buffer
        and place it in c ...
    no_lines := no_lines - 1;
    out_free.send;
    end

  begin  (*initialization*)
    no_lines := 0;
    ... initialization of other variables ...
  end;

  main:  start cardreader_process;
         start job_manager_process;
         start output_process;

cardreader_process:  process;
         var  card : cardimage;
         do  forever
           begin
             read card from cardreader;
             call inbuff.deposit(card);
             end;
         end;

output_process: process;
         var  line : lineimage;
         do  forever
           begin
             call outbuff.retrieve(line);
             write line on lineprinter;
             end;
           end;

job_manager_process: process;
        var  card : cardimage;
             line : lineimage;
        do  forever
          begin
            call inbuff.retrieve(card);
            process card, generating line;
            call outbuff.deposit(line);
            end;
          end;
end;
```

Figure 8 Use of Monitors

Message Passing [2] [1]

The synchronization mechanisms described thus far involve the explicit use of shared variables to implement interprocess communication and synchronization. A second approach to the coordination of concurrent processes uses message passing for this. The programmer defines logical communications channels between processes that must communicate. Two operations are defined on a channel: send and receive. The send primitive causes a message to be transmitted on the designated channel. The receive primitive removes a message from the designated channel. It may delay the invoker until a message has been sent on that channel.

Although shared memory can be used in the implementation of message passing, it is not required. This makes such a synchronization mechanism suitable for distributed processing and network applications, where there is no shared memory. Furthermore, since there need not be any shared variables, there is no need for the programmer to be concerned with mutual exclusion and critical sections. A consequence of this is that it is easy to understand in isolation each of the processes that comprise the concurrent program. The message passing primitives designate the only places in that code that can be affected by executive of other processes.

Message passing can be organized in a number of different ways. It may be required that a channel connect exactly one sender and receiver. Alternatively, more than one receiver and/or sender may be associated with each channel. Secondly, a channel may or may not have the capacity to buffer messages. If it does, then a send operation will delay the invoker only if the capacity of the buffer has been exhausted by messages that have been sent, but not yet accepted by a receiver. There are good reasons to require that a channel have

no buffering capacity, a finite buffering capacity, or an infinite buffering capacity. Note that if the channel has no buffering capabilities, then a send-receive pair defines a synchronization point in the execution of the invoking processes.

The use of message passing for synchronization is illustrated in Figure 9. There, the operating system example has been coded using typical message passing primitives. The following primitives are used:

```
send(x,m) -- send to process x message m.
receive(x,m) -- receive from process x message m.
```

Note that the buffer capacity of the channels has no effect on the correctness of this particular program.

4. Synchronization Mechanisms and System Structure

A system can be viewed as a collection of objects and tasks. The tasks use the objects to perform a computation. Shared objects may be implemented in one of two ways. In the following these two approaches are discussed, and their implications are explored.

In the passive model of objects, each shared object is represented in a portion of shared memory, perhaps with some procedures to facilitate access and manipulation of the abstraction implemented by the object. In order to perform an operation on an object, a task invokes one of these procedures. Thus, objects are manipulated directly by the process that requires the service. Since objects are subject to concurrent processes, the programmer must be concerned with defining critical sections and arranging for their mutual exclusion.

The active model of objects associates a process with each shared object.

```
main:     start cardreader_process;
          start output_process;
          start job_manager_process;

cardreader_process:  process;
   var  card : cardimage;
   do  forever
        begin
           read card from cardreader;
           send(job_manager_process, card);
           end;
        end;

output_process:  process;
   var  line : lineimage;
   do  forever
      begin
        receive(job_manager_process,line);
        write line on lineprinter;
        end;
      end;

job_manager_process: process;
   var  card : cardimage;
        line : lineimage;
   do  forever
      begin
        receive(cardreader_process,card);
        process card, generating line;
        send(output_process,line);
        end;
      end;
   end;
```

Figure 9 Synchronization using message passing

This "caretaker" process performs all operations on its object. When a process requires that an operation be performed on an object, a message is sent to the caretaker for that object. The caretaker performs the actual operation, and may respond with a completion message when the operation has been performed. Thus, objects are never directly manipulated by tasks, as was the case in the passive model. Furthermore, although concurrent processes interact and in fact may be synchronized through the use of these shared objects, the objects are never actually subject to concurrent access. Rather, operations

are performed on objects by the caretaker process on behalf of other processes.

5. Deadlock [9]

Deadlock or deadly embrace is a phenomenon that can be exhibited by concurrent programs. It occurs (in its simplest form) when one process waits for an action to be performed by another process, while this second process is waiting for an action to be performed by the first process. Clearly, in that case neither process can progress. In more complicated situations, deadlock may result in some interleavings of a concurrent program's execution, but not in others. Clearly, when writing a concurrent program it is necessary to ensure the absence of the possibility of deadlock for all possible interleavings. Techniques to accomplish this are described in [2] [9].

II. Kernel Systems

1. Nature of a Kernel

An operating system contains the software that manages and provides access to the hardware resources of the machine on which it is implemented. In particular, it contains modules that manage memory, schedule the processor(s), control IO devices, and implement a file system. The organization of an operating system can be based upon either of two concepts: a monolithic monitor or a kernel.

In a system based on a monolithic monitor, all the operating system modules are grouped into one large program that provides the means for all interactions between user programs and the hardware. Each module consists of a collection of procedures together with large tables that record the status of each system resource and user program. In order to avoid timing errors that could result from asynchronous interrupts, monitor modules are executed with interrupts inhibited or employ complex interrupt stacking and state saving protocols.

By contrast, in a system based on a kernel there is a small monitor, called the kernel or nucleus, that implements processes, and provides an interface to the host machine. All other operating system modules are moved from the monitor to distinct, uninterruptable processes. The role of the kernel is to support multiprogramming, namely the concurrent execution of more than one system or user program. It provides a virtual (abstract) machine that is more attractive than the bare machine since it implements processes, handles interrupts, and provides mechanisms for process communication. The advantages of a kernel based system relative to a monolithic monitor are that

system programs can execute concurrently with each other as well as with user programs, system modules are more cleanly separated, interrupt processing is more efficient, and fewer large tables are required.

2. Components of a Kernel

To support multiprogramming on any contemporary machine, all kernels contain the following components:

(1) Interrupt handlers, which receive hardware interrupts from IO devices and relay status information to device handling processes.

(2) Primitive operations, which implement the mechanisms used to create and destroy processes and to synchronize process execution, for example to allow processes to send and receive messages.

(3) A process scheduler or dispatcher, which allocates the processor to executable processes.

To support the other hardware management functions of an operating system, a kernel may also contain:

(4) Input/output routines, which initiate IO on peripheral devices on command from device handling processes.

(5) Memory management routines, which allocate and manage the main memory used by processes.

(6) Protection mechanisms, which control access to memory and files

Input/output routines are sometimes included since IO requires the use of special machine instructions executable only by the kernel. Memory management routines are included if memory mapping information, such as page tables, are

maintained by the kernel. Protection mechanisms are included for systems where the security of stored data is of concern. (Such a kernel is often called a security kernel). It should be noted that the kernel merely provides mechanisms that support I/O, memory management, or security; the modules that implement these functions usually reside within processes outside of the kernel. For example, in a kernel based system there is typically one IO driver process for each device; each driver process employs the kernel implemented mechanisms to communicate with other processes, initiate IO, and wait for hardware interrupts.

3. Implementation of a Kernel

The relation in a typical system between a kernel, the host hardware, and the system and user processes is shown in Figure 1. Also shown is the flow of control into and out of the kernel as well as the kernel's internal organization.

The kernel is always entered through the interrupt handlers. There is one handler for each kind of interrupt that can occur. Each handler saves the state of the executing process in the process' descriptor then handles the interrupt. To illustrate the functions of the interrupt handlers, the IBM 370 machine will be considered. (Other machines are similar at this level of detail.)

The 370 has five kinds of interrupts: supervisor call, IO, clock, memory, and program check. A supervisor call is used to invoke a kernel primitive such as "create a process"; the appropriate kernel primitive is therefore executed. An IO interrupt signals the completion of an IO operation; the appropriate IO process is notified, usually by sending it a message. A clock

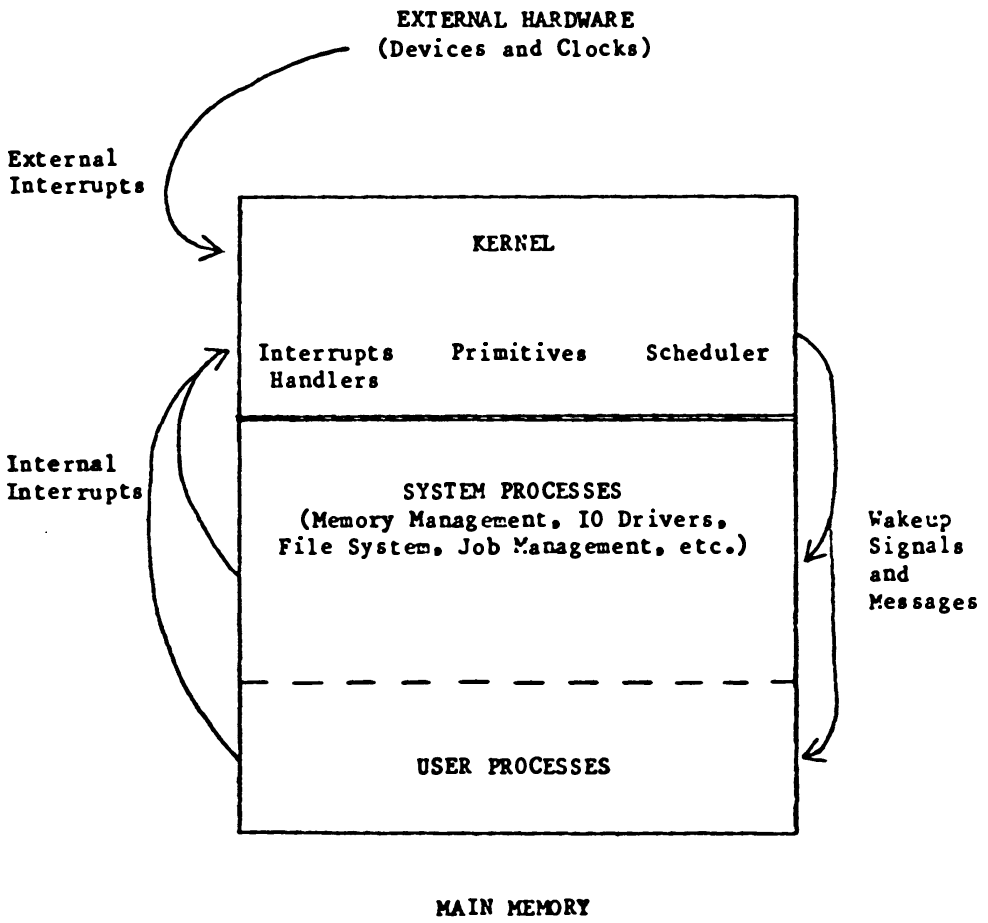


Figure 1
Kernel Organization

interrupt signals that a clock has reached a specified value (usually zero); the clock handler notifies the kernel scheduler or an appropriate system or user process. A memory fault interrupt signals an addressing exception; the appropriate memory management function of the kernel is notified either by sending it a message (if it is implemented as a process) or by branching to it (if it is implemented as a procedure). Program check interrupts signal pro-

gram exceptions such as overflow or division by zero; the appropriate trap handler, which is generally a special routine in the offending process, then receives control.

The primitives consist of the routines that implement process management and synchronization, and, if included in the kernel, the routines that implement IO, memory management, or protection mechanisms. These primitives are invoked by the interrupt handler, execute the appropriate program, and then call the scheduler.

The scheduler (sometimes called a dispatcher) selects a process to run and then loads the machine registers and program counter with the appropriate information, which was stored in the process' descriptor when the process was created or last interrupted. The selected process may be the one that was interrupted, or it may be another process. The scheduler may employ any number of CPU scheduling algorithms. (An alternative kernel organization that is sometimes employed is to have the interrupt handlers call the primitives as procedures, then call the scheduler if necessary, and then load the state of the interrupted or newly scheduled process.)

Most machines distinguish between (at least) two modes of execution: supervisor mode, in which any machine instruction may be executed, and user or problem mode, in which only the non-privileged instructions may be executed. The kernel executes in privileged mode; all processes, except possibly for IO drivers, execute in user mode. In this way the kernel is the only software module that may affect hardware status registers. Another distinction between the kernel and the processes is that the kernel executes with interrupts inhibited whereas processes may be interrupted. The kernel must be non-interruptable (unless great care is taken) since it maintains critical status

tables, such as process descriptors. Processes may be interrupted, however, because the kernel ensures that asynchronous interrupts are correctly synchronized with the recipient process.

4. Examples and References

One of the first operating systems to be based on a kernel was the T.H.E. system developed at the Technological University of Eindhoven under the leadership of Dijkstra [5]. The T.H.E. system kernel implements processes and semaphores. All other system modules are implemented as processes that synchronize by means of semaphores.

A brief, classic description of a nucleus (kernel) is given in [1], which describes the RC4000 multiprogramming system developed for a Danish machine by Brinch Hansen. The RC4000 nucleus provides primitives for process management and message passing. Using the message passing primitives, user processes can communicate with IO and file processes.

The components of a kernel, their implementation, and their relation to the other parts of the system are described in the book by Holt, et.al. [6]. A complete description of a comprehensive nucleus (kernel), which includes protection and memory management mechanisms, is included in the book by Shaw [9].

III. Hierarchical Organization

1. Basic Concepts

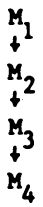
A software system consists of a number of modules, each of which defines some abstraction and provides operations to manipulate that abstraction. For example, a file system typically contains several modules. A file management module defines file directories and provides operations to create or open a file. A file access module defines the structure of files and provides read and write operations for accessing them. A buffer management module implements buffer storage and provides operations for requesting and releasing buffers. Finally, device drivers define IO devices and provide routines for reading and writing those devices.

When implementing one module, it is often convenient to make use of the operations provided by another module. Thus a file access module might use a device driver module. If one module, M1, uses the operations of another, M2, then we say M1 depends on M2. The dependency graph of a system is the graph that is obtained from all depends on relations between modules. (It has one node for each module and a directed arc from node M_i to node M_j if module M_i depends on module M_j.) A system is hierarchically organized if its dependency graph is acyclic. By contrast, a system has circular dependencies if some module depends directly or indirectly on itself.

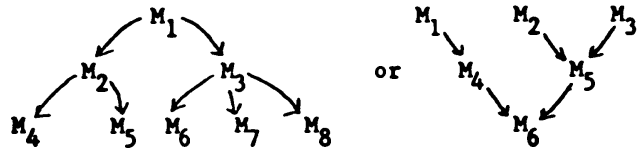
The simplest hierarchical organization is a linear one in which the modules are connected in a chain. More common are a tree-structured organization, in which the dependency graph forms a tree, or a level-structured organization, in which the dependency graph consists of two or more levels of nodes (such a graph is called a series-parallel graph). These three types of

hierarchical organizations are illustrated in Figure 1. In a level-structured

(a) Chain or Linear Structure



(b) Tree Structure



(c) Level-Structure

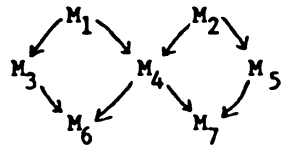


Figure 1
Examples of Hierarchical Organizations

organization, each level can be thought of as a virtual (abstract) machine whose modules provide operations used by higher levels and use operations provided by lower levels. (The lowest level interfaces to the physical machine.) Note that trees and chains are special cases of level structures.

2. Implementation of Hierarchies

Software systems are typically organized in a hierarchical fashion because a hierarchically organized system can be implemented one level at a time, starting with the lowest level (the one closest to the machine). Once a level is correctly implemented, the next higher level can be implemented. This is repeated until the entire system has been constructed.

There are three important benefits that result from this approach. First, it enables a system to be constructed one module at a time since each level can use the operations of the lower levels without concern for how they were actually implemented. Secondly, it makes it possible to change the implementation of one module without affecting any other module as long as the operation interfaces remain unchanged. Thirdly, it aids debugging, since once a level has been correctly implemented no errors in the implementation of higher levels can affect its correctness.

The highest level of a hierarchy is generally implemented by having one process for each module, for example one process for each user of a file system. The other levels can be implemented using procedures, processes, or monitors. The choice depends on the amount of concurrency desired and the need for protecting shared data from simultaneous access. A module is implemented as a process if one wants to ensure that its local data can only be accessed by one process and also wants to allow the module to execute concurrently with the processes that use it. A module is implemented as a monitor if one wants to ensure that its data is accessed by at most one process at a time and wants to delay a user of an operation from proceeding until the operation has been completed. A module is implemented as a procedure if one wants to allow it to be concurrently used by more than one other module.

3. Examples

Two examples of hierarchically organized systems will illustrate the above concepts. The T.H.E. system is an operating system developed at the Technological University of Eindhoven in the mid 1960's by a group led by E. W. Dijkstra [5]. The system consists of the following levels:

Level 4 - User Programs
Level 3 - Input/Output Controllers
Level 2 - Message Interpreter
Level 1 - Segment Controller
Level 0 - Processor Allocation

Level 0 is a monitor that defines processes, and provides semaphore operations which are used for process synchronization. (Level 0 is a kernel.) Level 1 manages a drum and implements virtual segments. Level 2 manages the console keyboard and provides facilities for user processes to communicate with the operator. Level 3 contains controllers for each peripheral device that implement IO access and buffering operations. Level 4 consists of the processes that manage and execute user programs. Each level, except Level 4, defines and implements an abstraction that is used by the higher levels. For example, above Level 0 each module is implemented as a process and synchronizes with other modules by using semaphores. In [5], Dijkstra describes the levels in detail and discusses the advantages of organizing the system hierarchically.

As mentioned earlier, a file system typically contains several modules and is usually organized hierarchically. An excellent description of the hierarchical approach to file system design is contained in [7]. A specific example is the file system in Unix, a timesharing system developed in the early 1970's by D. Ritchie and K. Thompson at Bell Laboratories [8]. In Unix, there is at least one process for each active user at a terminal. Each user process executes commands or programs, which may use the file system. At the top level of Unix, therefore, there are several processes that use the file system. At the top level of the file system itself is the file manager module, which defines directories and implements operations such as create and open. Once a file is opened, a user process can use the file access module's operations such as read and write. The file access module is logically below the file manager in the file system since it is also used by the file manager,

for example to search directories. To the caller of a file-access operation such as read, input appears to be unbuffered and synchronous (immediately after the completion of read, the data is available). Actually, the file access module depends on another level that implements a fairly complex buffering mechanism. The buffering mechanism in turn employs the lowest level file modules, which schedule and actually perform IO transfers. The resulting structure is summarized in Figure 2. Most of the Unix file modules are implemented as procedures to allow the greatest possible amount of concurrent execution. Only the modules that perform IO, buffer management and other critical operations such as directory manipulation are implemented as monitors.

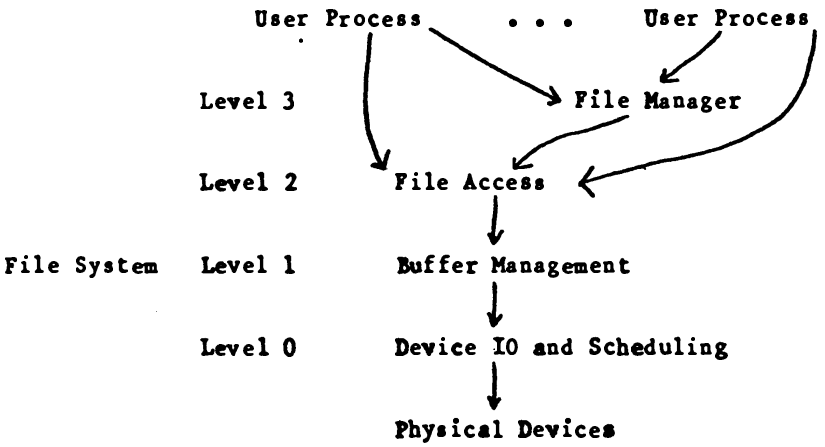


Figure 2

Unix File System Structure

Bibliography

1. Brinch Hansen, P. The nucleus of a multiprogramming system, Comm. ACM 13, 4 (April 1970), 238-241, 250.

2. Brinch Hansen, P., Operating System Principles, Prentice-Hall Inc., Englewood Cliffs, N.J. 1977.

3. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, N.J. 1977.

4. Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," Acta Informatica 1,2.

5. Dijkstra, E. W. The structure of the T.H.E. multiprogramming system. Comm. ACM 11, 5 (May 1968).

6. Holt, R.C. et al. Structured Concurrent Programming with Operating System Applications. Addison-Wesley, Reading, MA, 1978.

7. Madnick, S. E. and J. W. Alsop II. A modular approach to file system design. Proc. 1969 Spring Joint Computer Conference 1-13.

8. Ritchie, D. W. and K. Thompson. The Unix time-sharing system. Comm. ACM 17, 7 (July 1974), 365-375.

9. Shaw, A.C. The Logical Design of Operating Systems, Prentice Hall, Englewood Cliffs, NJ, 1974, Chapter 7.