CS 431: Introduction to Operating System

Class Note

Introduction to Process Management

V Kumar

Process

Main function of an operating system

a. Accept a set of jobs
b. Provide them their desired resources
c. Execute them and store the results to be collected by users.

User perception: Every user thinks that he/she has the entire machine, even though a number of users are serviced by the O/S simultaneously.

Formally: An operating system creates one **Virtual Machines** for one user.

To manage a virtual machine the O/S must keep track of

a. Memory uses.
b. Data/Files used by jobs
c. Status of every job (failed, running etc.).
d. Association of CPU with a job.

Program: Sequence of instructions representing relevant operations for manipulating data to get the desired result. In other words it is the coding of the solution of a problem in a programming language. It does not contain any command to the O/S.

Job: A sequence of commands to O/S and the program. The sequence of commands tells the OS how to treat the program, what to do with the result (print or save etc.) and how to terminate the program.

Convention: We will use job or program to refer to the same thing. Whenever necessary we will make the distingtion.

Behavior of the CPU: CPU is shared among a number of jobs. An instruction of a program tells the CPU the operation to be performed. Before the execution begins, all necessary information, such as, program size, object code location, files to be used, priority etc., about the job is saved in a special place. During execution this information may change and such changes are recorded. When this happens we say that the status of the jobs is "**being in execution**" but not necessarily being executed at a particular moment. When a job enters this state, i. e., when all information about the job and its intention is know to the system to put it in "**being in execution**" status then the program does not remain a program but termed as a **process**. Operationally, a process is a job specifying (dynamically) a sequence of actions and some information that represents its current state.

We, therefore, realize that inside a system it is meaningless to talk in terms of programs, since a program does not exist there, everything is a process. From now on, we will be using process to indicate a job/program. Sometime we will not differentiate among process, job and program. However, when confusion is likely to arise, we will be specific.

From the above description it is obvious that a program must be converted into a process before it can be executed by the O/S. This implies that one cannot build an O/S without defining process. The O/S must know the intention of a program and it is only possible if the program becomes "alive", i.e., converted into a process. It is the process which communicates with the O/S in behalf of a program.

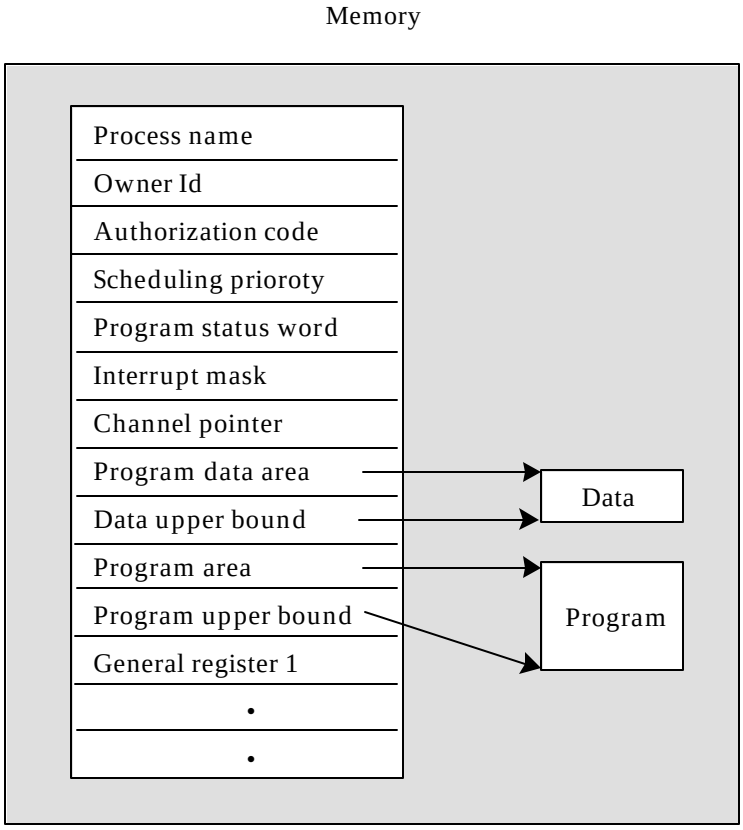
There are a number of additional properties of a process. Strictly speaking a process must finish in a finite time. If a program runs for an infinite amount of time then it is not regarded a process. However, there

are some system routines that run continuously as long as the system is up. Some of the poling routines, routines that checks the status of peripheral devices, etc., must run continuously and the have no termination point. These modules are also called process. But we will use the formal definition where a process must terminates in a finite time.

Since it is impossible to keep track of the speed of the execution of a process, i.e., how fast a process is progressing forward, we assume that the speed of a process is unpredictable.

Process Implementation

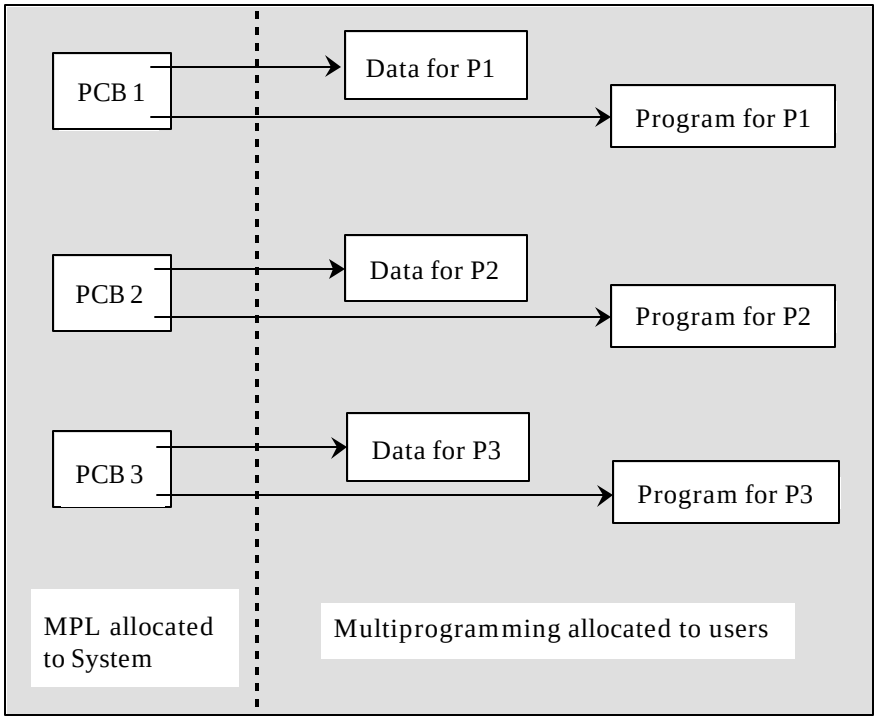
Implementation of a process means making available the requirements of a program to the O/S. A lot of information is associated with a process. This must be stored in a place in the memory and easily accessible to O/S during the lifetime of the process. This information is stored in a table called: **Process Control Block (PCB)**. A typical PCB may look like as follows:



Process Control Block

There is a reserved portion of the memory for a PCB. This area can only be accessed by O/S. A PCB contains pointers to other parts of the memory that holds data and object code of your process as shown in the above diagram. The data used by your process can also be used by other processes, i.e., data can be shared among a number of processes. In this case there are pointers from many processes to the same set of data. This creates the problem of process synchronization and will be studied in detail later. A set of PCBs can be scattered all over the main memory and at one time only one PCB will be referenced by the O/S. In many cases a procedure is also shared by many processes. If a code sharing is allowed then the O/S must guarantee that the code is pure (pure procedure or reentrant procedure), i.e., the code is not changed by a process or it does not modify itself.

So in a multiprogramming system the state of the system at any time may look like:



Memory

Process Creation: Creating (spawning) a new process requires that the program to be executed be identified and that the process be given a name. In addition, data relevant to process startup may be communicated.

Example: EDIT MYFILE System call: CREATE

Steps

1. Locate the load module file for EDIT.
2. Allocate memory for a PCB
3. Allocate a swap file for the new process and copy the program image (object code) into it.
4. Complete the various fields of the PCB, storing name, program size, disk swap file address, null base register value (the process is initially swapped out), save PSW pointing to the first instruction of the program, and so on.
5. Send a message containing the initialization text :MYFILE.
6. Notify the dispatcher that new process is to be initiated and its PCB placed in the ready queue.

PSW: Program Status Word. Special purpose register (hardware).

run / wait	sys / proc	mask	map	condition code	program counter

Run/wait: One bit. Indicates whether the process is running or waiting.
System/process: One bit. Indicates operation in either the system state or process state, with the primary objective being the authorization to execute privileged instructions.
Mask: Bit string specifies which interrupt classes are enabled (e.g., I/O or timer interrupt)

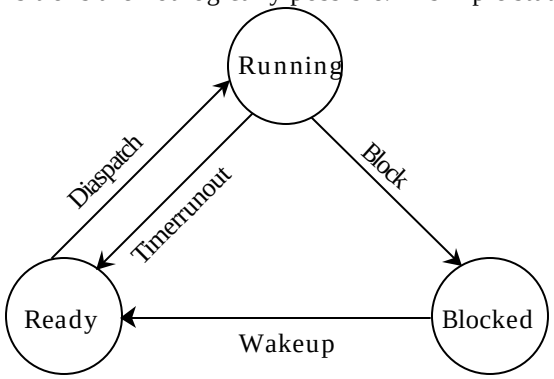
Map:	One bit. Indicates whether or not memory address mapping is enabled
Condition code:	Bit string holds the result of the last operation.
Program counter:	Holds the address of the next instruction to be executed or of the current instruction if an error occurs.

Process State: A process is a dynamic entity, so it can go into several states:

1.	Being executed by the CPU:	Running.
2.	Waiting for a resource:	Waiting.
3.	Ready for execution (waiting only for CPU):	Ready.
4.	Blocked by another process:	Blocked.
5.	Suspended (temporary hold):	Suspended.

Process state transition: A process may move from one state to another many times during its execution life. Such movement of processes is caused by a number of factors. Processes change state mainly to improve resource utilization and to satisfy scheduling policy. If there is an infinite number of resources (CPU, files, I/O processors, peripherals, etc.) then a process's request for a resource will never be denied. In this situation the state transition becomes meaningless. No computer system can have an infinite number of resources. If there are n processes then at time T at least the resource request of process i may not be satisfied and the O/S then must force process i to wait by moving it to the relevant state. Since there are a number of transitions possible a set of rules is defined to manage state transition.

State transition rules: State transition is represented by a State Transition Diagram (a digraph). It indicates the next possible state of a process and the operation that initiates the transition. State transition must follow certain rules. Some transitions are not logically possible. A simple state transition diagram look like:



Process State Transition

A process changes its state when some event occurs. For example, if a process is waiting (blocked) for a line printer that is being used by another process, then the waiting process goes to ready state when the line printer becomes available (event).

State transition modules

Dispatcher: changes the state from ready to running (assignment of CPU to a ready process is called dispatching).

dispatch (process_name): ready -----> running

Timerrunout: processes timeshare CPU resource (typically less than 1 microsecond). If the process does not voluntarily relinquish the CPU before the time interval expires, the clock generates an interrupt causing the operating system to regain control and the following state transition takes place:

timerrunout (process_name): running -----> ready **dispatch (process_name): ready ------> running**

Block: if a running process initiates an I/O operation before its CPU share expires, the running process voluntarily relinquishes the CPU (i.e., the process blocks itself pending the completion of the I/O operation). This state transition is:

block (process_name): running -----> blocked

Wakeup: when the required I/O completes, the blocked process is activated and the transition is:

wakeup (process_name): blocked -----> ready

IMPORTANT: the only state transition initiated by the user process itself is block; the other three transitions are initiated by entities external to the process.

Sometime it becomes necessary to force a state change externally (by human operator). For example, if a process is using a significant amount of CPU resource or a process needs a data that has not yet been created by another process, for some reason a process is slowing down the performance of the system, etc. In these situations the operator can move a process to one of the suspension states.

Short term suspension: process P1 is waiting for some input from process P2, and P2 has not produced the desired input then P1 will be suspended, not blocked or ready, until P2 sends the data to P1. When P1 gets the data, it goes to ready state.

Long term suspension: a process is not likely to get its desired resources for sometime or the process is affecting other parts of the system (performance etc.) or the operator may wish to suspend a process for the following reasons:

a. System is functioning poorly and may fail.
b. A user suspicious about the partial results of a process may suspend it (rather than aborting it).
c. In response to short-term fluctuations in system load, .

Two new states for managing process suspension,

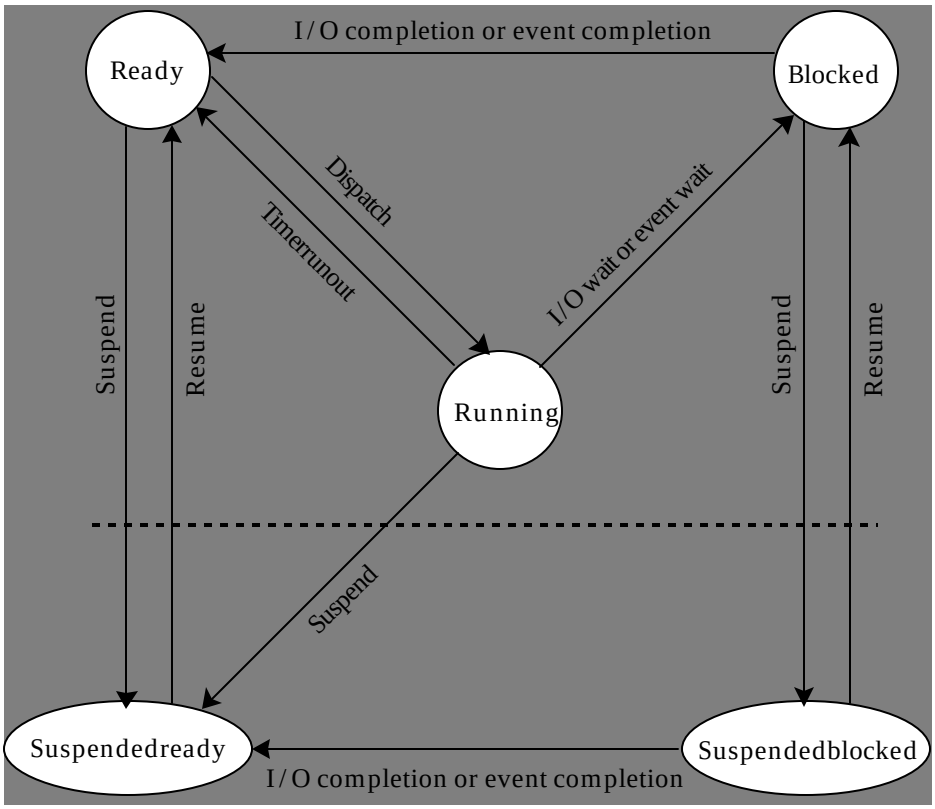
Suspendready: Suspension of ready process: **suspend (process_name): ready ----> suspendready**
A suspendready process may be resumed: **resume (process_name): suspendready -----> ready**

Suspendblocked: Suspension of a blocked process: **suspend (process_name): blocked ---> suspendblocked**

It is possible that a suspendblocked process may be changed to suspendeready on the completion of the desired I/O. In this case the transition is:

completion (process_name): suspendblocked -----> suspendready.

To keep track of all processes, the O/S maintains list of PCBs allowed. In general, there is a ready list, which contains PCBs of all ready processes, a waiting list which contains PCBs of all waiting processes, etc. By means of these lists, the O/S forms pools of processes in similar states which are examined by the O/S resource allocation routine when a resource becomes available.

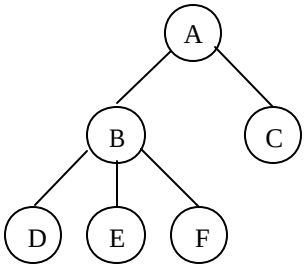


Process State Transition Diagram

State change: Occurrence of some event may lead to a state change. A state change is achieved by moving the relevant PCB from one list to another appropriate list. Since PCBs are always on the move the common data structure for their management is a linked list. To improve system performance a linked list with multiple pointers are used.

Process hierarchy (Spawning) and process creation: A process may spawn a new process. The creating process is called the parent process and the created process is called the child process. In turn the child process may create another process and thus a hierarchical process structure is created.

Processes may be dynamically created or destroyed. In a simple O/S, a set of processes already exist. In such a simple system no more processes are ever required. In more complex systems there are system calls to create a process or child process dynamically. In this way, at any time a tree of processes may exist in the system. For example, during system initialization a special process called boot loads the O/S. It then creates a process for each terminal connected to the system. The following diagram gives a snapshot of an O/S state:



Management of process hierarchy: In many system when a parent process dies then all its children also die since there is only one PCB for the entire process tree. In other systems the elimination of parent process

does not affect children process. This however, is difficult to manage since the O/S must create separate PCBs for each child or grandchild. It must maintain a link between the ancestors and dependents since a dependent any time may request some information from its parent. These requirements make the O/S very complex.

Interrupt

Sometime it is necessary to alter the normal execution of a process. This is done by generating an interrupt. Such break in the execution of a process may be generated by the process itself or by the operating system. An interrupt generated by the operating system (register overflow, division by zero, addressing error, etc.) is usually termed as trap and the interrupt generated by the user process is regarded as a normal interrupt.

Example: pressing control and C key on the keyboard during the execution of a process. Switching off the line printer while printing result, etc.

Management of interrupts: When an interrupt occurs the following steps are taken

1. OS takes control.
2. Save status of the process during which interrupt occurred.
3. Decide which interrupt routine is to be used to manage the interrupt.
4. Load and execute that routine.
5. At the end of interrupt processing resume normal execution of the process saved in step 2.

Example

Event: During printing line printer ran out of paper. **User process**: P1.
An I/O interrupt was generated by the I/O program handling the printing process.

1. Save status of the printing process (how far executed, what is the next instruction when stopped and many other things)
2. OS takes control.
3. Looks at the interrupt code and decides which interrupt routine is responsible for detecting the problem.
4. Loads this routine from the library.
5. Creates process (system process) to run this routine.
6. This interrupt routine prints message "LP out of paper".
7. Operator loads paper.
8. Interrupt routine detects this and passes control to OS.
9. OS removes the interrupt process and loads P1.
10. End of interrupt processing.
11. Resumes execution of P1.

Interrupt facility improves CPU efficiency. Interrupts can be generated by hardware or software. All interrupts can be categorized into six different classes:

SVC (supervisor call): A user process, if requires more memory or any other resource asks the OS by SVC instruction. An interrupt is generated and the request is either denied or granted.

I/O interrupt: Generated by I/O system (hardware and software combination) during I/O processing. They can be generated when an I/O completes, error during file transfer, printing etc.

External interrupt: External to the system. By operator or by another CPU in a multiprocessor system.

Restart interrupt: Bootstrap. System reloading.

Program check interrupt: Interrupt during normal execution of a process. For example, register overflow, division by zero etc.

Machine check interrupt: By hardware.

Context Switching: CPU is taken away from one process and assigned to another process. The PCB of the process that has to give up the CPU is saved some place in the memory or on the disk and the new process uses the CPU.

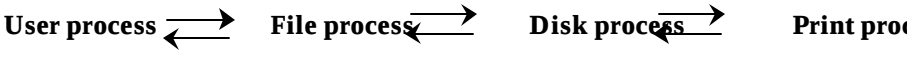
Inter-process Communication

Processes during execution may interact with each other. Such interaction can be categorized as follows:

Interprocess Synchronization: Concurrent processes need to synchronize their execution over the use of common resources to preserve system integrity. For example, the execution of read and write operation on a common file, use of a common line printer, etc.

Interprocess Signalling: Concurrent processes may need to exchange some timing signal among each other to coordinate their collective execution. For example, in real-time systems a process may need to signal another process to begin processing the data produced by the first process.

Interprocess Communication: Concurrent processes may need to communicate to each other for exchanging data, reporting progress, etc. For example, communication among different O/S modules can be as follows:



Problems in process communication

Race Condition: A race condition exists when two or more processes give different results for same set of data (if they need data) when executed at different times. We give some examples. We observe the following:

a. We know nothing about the relative speed of processes.
b. Reading and Writing are independent indivisible operations.

Example 1: Suppose an installation has one card reader and one line printer which are shared between process P1 and P2.

```
          P1
again:  read (a, b, c);
          if (a = b) then
             write ('I am first')


          else
            ---------
          goto again;
```

```
                    P2
------------
------------
again:  read (x, y, z);
              if (x = y) then
                 write ('I am second')

               else
                   -----------
               goto again;
```

If these processes are allowed to use these resources in an uncontrolled manner then the following incorrect output may be generated:

I am first	**OR**	**I am second**
I am second		**I am first**
I am first		**I am second**

Example 2

Consider a system with many time sharing terminals. Suppose it is desired to monitor continuously the total number of lines that user(s) have entered through these terminals since the day began. Assume each terminal is monitored by different process. Each time one of these processes receive a line from a user terminal it increments a global shared variable (accessible to all processes): LINECOUNT which acts as a counter. We can visualize our system as follows. N and M are local variables for P1 and P2 respectively.

user 1 (process P1)	**user 2 (Process P2)**
LINECOUNT = 9 (initial value)	
load N LINECOUNT	
add 1 N	load M LINECOUNT
store N LINECOUNT	add 1 M
	store M LINECOUNT

Execution

N = 9	------
N = N+1 (= 10)	M = 9
End of timeslice	M = M+1 (= 10)
	store M (in LINECOUNT)
	End of P2
-- Resumes execution --	
goes back to the ready queue	
store N (in LINECOUNT)	
End of P1	

Final result: 10 (incorrect). **Correct value**: 11.

Reason: process P2 was allowed to change the value of LINECOUNT when process P1 was working with LINECOUNT.

Process management

To eliminate race condition and some other problems (we will discuss very soon) processes must be executed in a way that will guarantee the correctness of the final results and the integrity of the entire system. We study such schemes under the heading of Process management.

Critical Section: A critical section of a process is a set of instructions which operates on global variables.

In LINECOUNT program the critical sections of P1 and P2 are:

load	**N**	**LINECOUNT**	**load**	**M**	**LINECOUNT**
store	**N**	**LINECOUNT**	**store**	**M**	**LINECOUNT**

So

a. LINECOUNT should not be accessed for writing by more than one process.
b. Reading of LINECOUNT should be allowed by concurrent processes since read operation does not change the value of the variable.

Mutual Exclusion: Mutual exclusion of processes with respect to a given critical section means that no more than one process can be in the critical section at a given time.

Formally: No more than one process should be allowed to enter into the critical section at one time, i.e., processes should mutually exclude each other over the use of common (global) variables.

While a process is in its critical section, other processes may certainly continue executing outside their critical sections. When a process leaves the critical section, then one of the waiting processes should be allowed to proceed (if there is a waiting process).

Inside a critical section a process has exclusive access to shared data, and all other processes currently requiring access to that data are kept waiting. Therefore critical section must execute as quickly as possible, a process must not block within its critical section, and critical section must be carefully coded (to avoid the possibility of infinite loop, for example). Therefore, to manage processes correctly an algorithm must satisfy the following requirements:

i. Concurrent processes with critical section (with respect to each other) must be mutually excluded from simultaneous execution of their critical sections.
ii. A process stopped (crashing or terminating) outside the critical section should not affect the ability of other contending processes to access the shared resource.
iii. When more than one process wishes to enter the critical section then only one process should be granted the permission to enter in a finite time.

Concurrent Processes: Concurrency means two or more operations can proceed in parallel if each operation has a CPU available to it. If there is only one CPU then such concurrency is implemented by time-slicing. Let us first look at some example of **parallel operations** which would produce the same result as sequential operations. Consider the following sequence of instructions:

1. a := x + y; 3. b := z + 1;
2. c := a - b; 4. w := c + 1;

No. of CPU = 1.
Sequential execution result = P. (Correct result)

No. of CPUs = 4.
Parallel execution, (one instruction on one CPU) result = Q.
P ? Q (Q is not correct)

Reason for incorrect result: Dependency among statements.

Example: last three instructions are dependent on each other since they use common variables **b** and **c**, and they cannot be executed on several CPUs. Similarly the first and the second also cannot be executed on several CPUs because of the common variable **a**.

Now consider the following example:

1. a := m+1; 3. b := x+1;
2. c := r +1; 4. d := c +2;

The first three instructions can run on three CPUs or on one CPU in any order and they will produce the same result every time since there is no dependency. The last two instructions have a common variable and so the c := r + 1 must be run before d := c + 2. We are going to formalize these ideas and discuss several new concepts which evolve from concurrency.

While a process is in its critical section, other processes may certainly continue executing outside their critical sections. When a process leaves the critical section, then one of the waiting processes should be allowed to proceed (if there is a waiting process).

Inside a critical section a process has exclusive access to shared data, and all other processes currently requiring access to that data are kept waiting. Therefore critical section must execute as quickly as possible, a process must not block within its critical section, and critical section must be carefully coded (to avoid the possibility of infinite loop, for example).

Synchronization Mechanisms

A number of simple solutions is presented. We use the results of these solutions to identify the depth of process management problems and then present improved solutions that are used by almost all operating systems. We use terms "parabegin" and "paraend" to indicate that enclosing statements can be executed in parallel in multi-processor systems or concurrently in uniprocessor systems.

Solution 1:

```
begin   integer turn;
            turn := 1;
            parabegin
                P1: begin  L1:   if turn = 2 then goto L1;
                                 critical section 1;
                                 turn := 2;
                                 more statements;
                                 goto L1
                    end
                P2: begin  L2:   if turn = 1 then goto L2;
                                 critical section 2;
                                 turn := 1;
                                 more statements;
                                 goto L2
                    end
            paraend
        end;
```

Problems:

a. Busy wait. Waste of CPU time.
b. If turn is not set to 1 by the process which set it to 2 then other waiting processes will go into infinite loop.
c. Not very suitable for large number of processes.
d. Too restrictive; the critical sections can be entered only in the order P1, P2, P1, P2, ... This violates the requirement ii (above) since a process stopped outside its critical section can hold up the progress of the other, supposedly independent process.

Taking this into account, consider a second solution which uses two flags, C1 and C2, to indicate when a process is in its critical section. C1 = 0 means P1 is in its critical section; C1 = 1 means P1 is outside its critical section. Likewise for C2 and P2.

Solution 2:

```
begin   integer C1, C2;
    C1:= 1; C2 := 1;
    parabegin
        P1: begin  L1:  if C2 = 0 then goto L1;
                        C1 := 0;        (*P1 is in CS *)
                                critical section 1;
                                C1 := 1;
                                more statements;
                                goto L1
            end
        P2: begin  L2:  if C1 = 0 then goto L2;
                        C2 := 0;        (* P2 in CS *)
                                critical section 2;
                                C2 := 1;
                                more statements;
                                goto L2
            end
    paraend
end;
```

Problems:

a. Both processes can pass the test on C variables at the same time, thereby, determining that it is safe to proceed, entering their CS at the same time, violating requirement 1.
b. Busy wait may happen.

The problem is that setting the inside/outside flags (the C's) is not protected from simultaneous execution. Solution 3 remedies this problem by having a process set its own flag before testing the other process' flag.

Solution 3:

```
begin   integer C1, C2;
    C1:= 1; C2 := 1;
    parabegin
        P1: begin  A1: C1 := 0;             (* P1 in CS *)
                L1:    if C2 = 0 then goto L1;
                                critical section 1;
                                C1 := 1;
                                more statements;
                                goto A1
            end
        P2: begin  A2: C2 := 0;             (* P2 in CS *)
                L2:    if C1 = 0 then goto L2;
                                critical section 2;
                                C2 := 1;
                                more statements;
                                goto A2
            end
    paraend
end;
```

Mutual exclusion: Yes.

Problem: P1 and P2 may deadlock. The problem is that each process waits for entry into critical section with its inside/outside flag set wrong. That is, it waits with the flag indicating it is in its critical section when in fact it is not. Solution 4 tackles this problem by removing the inside flag while waiting:

Solution 4:

```
begin   integer C1, C2;
    C1:= 1; C2 := 1;
    parabegin
        P1: begin  L1:  C1 := 0;
                   if C2 = 0 then begin  C1 := 1;  goto L1;  end
                            critical section 1;
                            C1 := 1;
                            more statements;
                            goto L1;
            end
        P2: begin  L2:  C2 := 0;
                   if C1 = 0 then  begin  C2 := 1;  goto L2;  end;
                            critical section 2;
                            C2 := 1;
                            more statements;
                            goto L2
            end
    paraend
end;
```

Deadlock: No.

Problem: what if P1 and P2 happen to proceed together? They can get into an infinite loop. This violates requirements **c**.

The first correct solution of this control problem is by Dekker. The problem with his solution is that it works only for two processes and cannot be easily extended beyond that number. Thus its theoretical significance cannot be overlooked but its applicability in practice is non-existent.

Solution 5:

```
 begin   integer C1, C2, turn;
     C1:= 1; C2 := 1; turn := 1;
     parabegin
         P1: begin     A1: C1 := 0;
                   L1:  if C2 = 0 then
                   begin  if turn = 1 then goto L1;
                         C1 := 1;
                         B1:       if turn = 2 then goto B1;
                         goto A1;
                   end
                   critical section 1;
                   turn := 2;
                   C1 := 1;
                   more statements;
                   goto A1
                end
         P2: begin     A2: C2 := 0;
                   L2:  if C1 = 0 then
                   begin  if turn = 2 then goto L2;
```

```
                    C2 := 1;
                    B2:     if turn = 1 then goto B2;
                    goto A2;
                end
                critical section 2;
                turn := 1;
                C1 := 1;
                more statements;
                goto A2
            end
        paraend
    end;
```

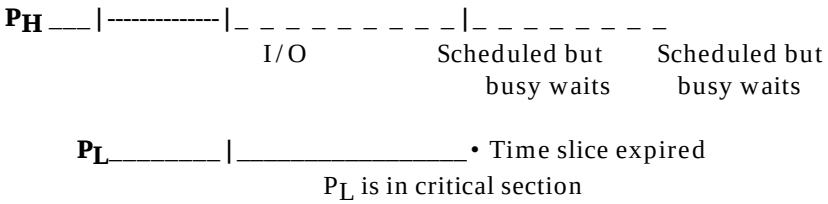
Peterson's Algorithm

turn is an int variable. **interested** is an int array. All elements of **interested** is initialized to 0 (false)

Time	**Process 0**	**Process 1**
↓	other is a local int variable	other is a local int variable
	enter_region (process); other := 1 - process	enter_region (process); other := 1 - process
	interested [process] := TRUE; turn := process	interested [process] := TRUE; turn := process
	while (turn = process AND interested [other] =True)	while (turn = process AND interested [other] =True)
	Enters Critical Section	
	leave_region (process)	**Enters Critical Section**
	interested [process] := false	leave_region (process)
		interested [process] := false

Problem with Peterson's solution

$\mathbf{P_H}$ = high priority process. $\mathbf{P_L}$ = low priority process. The process scheduler works on priority. Execution pattern:



So probably $\mathbf{P_H}$ will never get a chance to run.

Semaphore and Busy-Wait Implementation

Originally Dijkstra proposed a solution which is based on Semaphore and Primitives. His solution is very commonly used in most of the systems today.

Semaphore: A protected variable. A semaphore can be manipulated only by special operations (may be primitives). There are two types of semaphore:

Binary Semaphore : It can take only two values; = 0 resource is free and = 1 resource is busy.
Counting semaphore (also called **General semaphore**): it may take n (n > 2) number of values.
Primitive: A special operation with only two states: done or not done.
Not done: If a primitive fails to complete then ignore all its operations done so far.

Example: Read A. Fails after reading only 10 bits of the data. The variable A would not contain 10 bits. The effect of reading 10 bits will be removed from the system. This is equivalent to Read operation was never started.

Done: If a primitive has started then it cannot be interrupted. It will stop when it has completed its entire operation.

We define two primitives: WAIT and SIGNAL (originally called P and V by Dijkstra). These primitives operate on semaphore S (binary or general). We define the working of these primitives on S:

WAIT (S): Decrement the value of its argument, S, as soon as it is non-negative.

```
if  S > 0  then S := S - 1
else  wait on S
```

SIGNAL (S): Increment the value of its argument, S and wake up a process waiting on semaphore S:

```
if (one or more processes are waiting on S) then
    wakeup one process;
S := S + 1
```

Example: We present an example to show the use of WAIT and SIGNAL.

```
var mutex: semaphore; {binary}
    mutex := 1;

    parabegin
        process p1;
        begin
            while true do
            begin
                wait (mutex);
                Critical section;
                signal (mutex);
                other statements;
            end {while}
        end;

    process p2;
    begin
        wait (mutex);
        Critical section;
        signal (mutex);
        other statements;
        end {while}
    end;

    process p2;
```

```
        begin
            wait (mutex);
            Critical section;
            signal (mutex);
            other statements;
            end {while}
        end;
    paraend;
```

The following table is a log of the execution of p1, p2 and p3. Mutex = 0 (free) = 1 (busy). Before activating the three processes, mutex is initialized to 1. This initialization is indicated in time T1. The table represents the execution history for a multiprocessor system where all processes (p1, p2 and p3) can go in parallel. Under a uniprocessor system, there is always a small time lag between the execution of any two processes and the table should be read accordingly. This means p1 executes wait (mutex) and then immediately after this either p2 or p3 executes wait (mutex).

Time	p1	p2	p3	mutex	processes in CS	wishes to enter
T1	-	-	-	1	-	-
T2	wait (mutex)	wait (mutex)	wait (mutex)	0	-	p1, p2, p3
T3	in CS	waiting	waiting	0	p1	p2, p3
T4	signal (mutex)	waiting	waiting	1	-	p2, p3
T5	other statements	in CS	waiting	0	p2	p3
T6	wait (mutex)	in CS	waiting	0	p2	p3, p1
T7	waiting	signal (mutex)	waiting	1	-	p3, p1
T8	in CS	other statements	waiting	0	p1	p3

Semaphores may be provided in a programming language, as shown in the example, or as the O/S service invoked via system calls. When provided by the O/S, semaphore variables are not declared and manipulated in the language, but are manipulated through system calls such as CREATE_SEMAPHORE, ATTACH_TO_SEMAPHORE, WAIT, SIGNAL, AND CLOSE_SEMAPHORE.

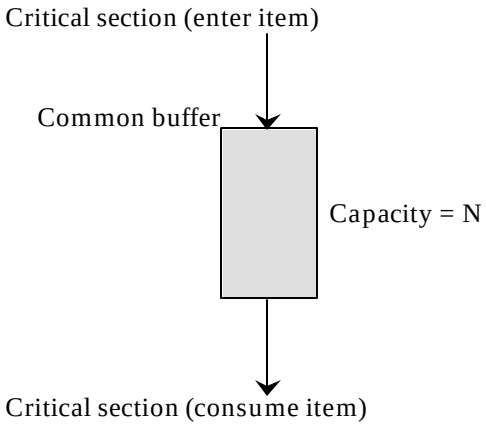
Semaphore mechanism does not define any scheduling policy. For example, if p1 and p3 are waiting then the synchronization program does not decide who should enter the CS. It is taken by the scheduler. A faulty scheduling policy may force some process to wait for ever (starve). To avoid starvation, a FIFO scheduling policy can also be used and this implementation of servicing is sometimes referred to as a "strong implementation of semaphores".

Semaphore Granularity

Small: One semaphore for each sharable resource (software and hardware). This provides high concurrency but makes it difficult to manage such large number of semaphores. Maintenance of semaphores are expensive.

Coarse: A set of resources is associated with a semaphore. This setting decreases the run-time overhead but increases the conflicts among processes thus increasing wait time and may have some scheduling problems.

A common scenario in O/S: A producer produces data and stores them into a common buffer for a consumer to consume. For example, a user program (producer) generates data for the line printer and the printer routine (consumer) empties the buffer and prints the data on the line printer. The sharing of the common buffer must be synchronized so that the producer should not try to put data items in a full buffer and the consumer should not try to read data from an empty buffer



Program: S1, S2 : semaphore; S1 := 1; S2 := 0;

Time

Producer	**Consumer**
begin	begin
while true do	while true do
begin	begin
produce item;	Wait (S2);
Wait (S1);	get item;
enter item;	Signal (S1);
Signal (S2);	print item;
end	end
end	end

Problems with semaphores

Most criticisms revolve around the two main themes:

1. Semaphores are unstructured: they make synchronization, and ultimately the system integrity, dependent on strict adherence of all concerned systems programmers to the specific synchronization protocols devised for the problem at hand. Reversing P and V primitives, forgetting either of them, or simply jumping around them may easily corrupt or block the entire system.

Example

A. Suppose that a process interchanges the operations on the semaphore S. That is, it executes:

V(S);

.....
Critical section
.....
P(S);

Result: Several processes may be executing in their critical section simultaneously, violating the mutual exclusion requirement. This time-dependent error may be discovered only if several processes are simultaneously active in their CS. Note that this situation may not always be reproducible.

B. Suppose that a process exchanges V(S) with P(S). That is, it executes:

P(S);
.....
critical section;
.....
P(S);

Result: A deadlock.

C. Suppose that a process omits the P(S) or the V(S) or both.
Result: Mutual exclusion is violated or a deadlock will occur.

2. Semaphores do not support data abstraction.

Another example

Some scheme define Sleep and Wakeup primitives. They work as follows:

Sleep: causes the procedure which called sleep to block. The process is suspended until another process wakes it up.
Wakeup: causes a suspended process to become ready.

Producer and Consumer problem solution

N = 100 (bounded buffer size). count = 0 (no. of items in the buffer)

Producer (P)	**Consumer (C)**
produce_item ()	
	if (count = 0); sleep (); {sleeps}
if (count = N); {Room in the buffer, does not sleep}	
enter_item ()	
count := count + 1	
if (count = 1) then wakeup (C); {P wakes up C*}	
	remove_item
	count := count - 1
	if (count = N-1) then wakeup (P)
	consume item

Problem with Sleep and Wakeup

Consider the following execution:

Buffer is empty. Consumer starts first

if (count == 0)
C reads the value of count (= 0) but before
executing the test, time slice is over and
C is suspended (not asleep).

Producer begins, produces item and enters it in the buffer. Increments count to 1. It then tries to wake up consumer but consumer is not asleep so the signal has no effect and therefore is lost.

Consumer begins execution. It completes
the test (count == 0) and finds it
to be true so it goes to sleep.

Producer begins and executes count = count + 1, making it 2. The next test if (count == 1) wakeup (C) will fail and Producer will keep on filling the buffer. When the buffer is full the Producer goes to sleep. Now both are sleeping.

Hardware support for Mutual Exclusion

Semaphore and primitives are very useful and effective tools for managing concurrent processes. However, they are no good if the implementation of mutual exclusion via these tools is very difficult and not reliable. If their implementation is feasible at the hardware level then we have to see how efficient they are.

Disable / Enable Interrupts

Almost all O / S provide a facility to disable and enable interrupts generated by any system module or processes. This mechanism can be used to obtain a resource exclusively and then release it after use. This may be accomplished as follows:

```
DI                              ;disable interrupts
    Enter critical section ; use the reserved resource
EI                              ; enable interrupts
```

The intent of disabling interrupts is to prevent any interference during execution of the critical section. This process generally defers recognition of external events that may cause another process to run and access the same resource, and also temporarily disables the scheduler in order to prevent preemption due to rescheduling. In effect, whenever a process is in its critical section, interrupt disabling forces the whole system into a state of hibernation.

The innermost kernels of many commercial O / S employ this mechanism as a quick and easy way to allow a process to enter a critical section.

This works fine but has a number of serious problems. If this facility is available to application programmers, they can use it for disrupting the scheduler and the normal running of the entire system. Users can apply DI and indirectly raise the priority of their program thus affecting the scheduling of real higher priority process. Furthermore, it may bring the entire system to a halt. For example:

```
DI          ;disable interrupts
Halt        ;halt the processor
```

swiftly brings down many a multiprogramming O/S. WAIT and SIGNAL may be implemented by the system by the use of DI and EI. But this mechanism works only on one CPU. Thus an operating system for a uniprocessor could not be easily ported to a multiprocessor system.

Test-and-Set (TS) Instruction

A direct hardware support of mutual exclusion. Basic idea is to test a variable associated with a resource to see if the resource is free. If it is free then first make it busy by resetting the value of this variable and use the resource. When finished reset the value of the variable. This path is followed by every process desiring to use a resource. Every resource has such variable associated with it. TS instruction takes only one operand and the entire instruction is executed indivisibly.

```
TS operand          ;test and set operand
```

a. Compare the value of the operand (BUSY or FREE).
b. If free then set it to busy otherwise wait.

The WAIT primitive on the semaphore S may be implemented as follows, if TS is available in the hardware of a system.

```
WAIT:  TS      S              ;request exclusive access
       BNF     WAIT           ;(branch if not free) repeat until granted
       RETURN                 ;proceed to critical section
```

Every process execute this set of instructions indivisibly before it can use the resource. If S = 1 (resource busy) then the process loops otherwise sets S to 1 and enters critical section. IBM/360 were the first systems to use the TS instruction in hardware.

Problems

Suppose p1 is executes TS instruction and enters critical section. While it is in the critical section another higher priority process p2 preempts p1 (p1 did not reset S). p2 will loop, p1 could not resume since p2 has not completed its execution. This problem usually occurs on priority-based systems. One way to avoid this is if the O/S monitors the progress of every user process. This simple proposition is rarely implemented since O/S usually cannot afford the overhead imposed by having to keep track of the nature of instructions being executed by user processes.

A high level solution to some of the process synchronization problems

Monitor: All mechanisms discussed so far concern themselves only with making sure that at most one process is allowed in CS at any time. These algorithms have problems. Monitors are operating system structuring mechanism that addresses these issues in a systematic and rigorous manner.

Basic idea behind monitor: Provides data abstraction in addition to concurrency control, that is, to control not only the timing but also the nature of operations performed on global data (critical section) so as to prevent meaningless or potentially harmful updates.

In the solutions presented earlier, a procedure operates directly on CS. That is, it enters into CS and manipulates the CS variables. This way of manipulating CS creates all the problems listed before.

Solution to these problems: Do not let procedures work directly on CS via semaphores and primitives, but ask a procedure to do the job in behalf of the procedure. This is just like invoking a library routine

(procedure/function) to compute mathematical expressions rather than writing your own procedure/function to do the computation. In the first case an extra level of indirection is added.

Advantages

1. The programmer is relieved from worrying about the correctness of the procedure.
2. Many programs (independent) can invoke the same library routine at the same time. This cannot be done under the second option.
3. Malfunction probability extremely low.

Monitor works on this idea of data abstraction and an extra level of indirection. It uses semaphores and also conditional variables. Monitors make the CS accessible indirectly and exclusively via a set of publicly available procedures. In terms of procedure/consumer problem, the shared global buffer may be declared as belonging to a monitor, and neither producers nor consumers would be permitted direct access to it. Instead, procedures may be allowed to call a monitor-provided public procedure and to supply the produces item as its argument. A monitor procedure would then actually append the item to the buffer. Likewise consumers may call on a monitor procedure to obtain a produced item from the buffer. A collection of monitor procedures may thus handle buffer management and synchronization of concurrent requests internally by means of code and variables hidden from users.

Monitor structure

1. Collection of data 2. A set of procedures to manipulate them 3. A set of private variables

Example of a monitor declaration

```
monitorname: monitor;
begin
    declaration of private data;   {local monitor variable}
    ----
    procedure public (formal parameters);      {public procedures}
        begin
            procedure body;
        end;
    ----
    procedure private    ;     {private procedure}
        ----
        initialization of monitor data;
        ----
end monitorname;
```

Properties

1. Static structure
2. Becomes active when one of its procedures is invoked by a running process (user process)
3. Processes executing monitor procedures are allowed to wait on a particular condition without affecting other monitor users significantly.

A sample monitor implementing WAIT and SIGNAL operations

```
wait_signal: monitor;
begin
    busy: boolean; free : condition;
    procedure mwait;
        begin
```

```
            if busy  then free.wait; busy := true
        end;
    procedure msignal;
        begin
            busy := false; free.signal
        end;
    {monitor body - initialization}
    busy := false
end wait_signal
```

Explanation:

free.wait: the second name (wait) invokes the monitor primitive WAIT, the first name (free) indicates on which condition WAIT should be executed, because there may be several conditions in a monitor and the desired one must be identified.

Execution of free.wait: after execution the calling procedure is suspended on the queue associated with FREE. If more callers invoke MWAIT while the first one is still in the critical section, they also join the queue of suspended process associated with the condition FREE.

Execution of msignal: when the first caller finally executes MSIGNAL, BUSY is set to false and condition FREE receives a signal. As a result one of the waiting process is awakened. If a message is sent and there is no process waiting on FREE then the signal is ignored.

Synchronization of Producer/Consumer using Monitor

```
module m_procedure_Consumer
-----
pc : monitor;
    begin
        buffer :  array [1..capacity] of item;
        in, out: (1..capacity);  count: (0..capacity);
        may produce, mayconsume : condition;
        procedure mput(pitem : item);
            begin
                if count = capacity  then mayproduce.wait;
                buffer [in] := pitem;
                in := (in mod capacity) + 1;  count := count + 1;
                mayconsume.signal
            end;      {end of mput}
        procedure  mtake (var citem : item);
            begin
                if count = 0  then mayconsume.wait;
                citem := buffer[out];  out := (out  mod capacity) + 1;  count := count - 1;
                mayproduce.signal;
            end;      {end of mtake}
        {monitor body - initialization}
        in := 1; out := 1; count := 0
    end pc;
end       {m_procedure_consumer}
```

User process that uses the above monitor code.

```
module u_procedure_consumer;
```

```
    -----
    process producerX;
        var pitem : item;
        begin
            while true do
                begin
                    pitem := produce; pc.mput (pitem);
                    other_X_processing
                end; {while}
        end;    {producerX}
        -----
        process consumerZ;
        var citem : item;
        begin
            while true do
                begin
                    pc.mtake (citem); consume (citem);
                    other_Z_processing
                end;    {while}
    {parent process}
    begin
        initiate procedures, consumers
    end
end {procedures_consumers}
```

Explanation

1. MPUT: serves producer process.
2. PITEM: item generated by producer
3. (COUNT = CAPACITY) meaning buffer is full caller waits by the condition MAYPRODUCE.WAIT
4. MAYPRODUCE.SIGNAL is sent by the consumer.
5. (COUNT = 0) meaning consumers are prevented from executing by being suspended on the MAYCONSUME condition.
6. When an item is produced, a consumer is freed and the monitor completes the execution of the MTAKE procedure on the consumer's behalf. This ultimately provides the consumer with an item from the buffer,after signaling MAYPRODUCE to activate a waiting producer, if any.
7. The common buffer is protected by the monitor, and it cannot be accesses o manipulated in any way other than those provided by MPUT and MTAKE procedures.

Interprocess Communication: Text book pages 175 - 183.