Operating System

MODULE-2: Process Management

Compiled by: Prof. Snehal Andhare

snehal.andhare@vit.edu.in

VIT Vidyalankar Institute of Technology

Vidyalankar Institute of Technology
Wadala (E), Mumbai
www.vit.edu.in

Certificate

This is to certify that the e-book titled "OPERATING SYSTEM" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.

Signature

Date: 12-10-2015

Prof. Snehal Andhare

Assistant Professor

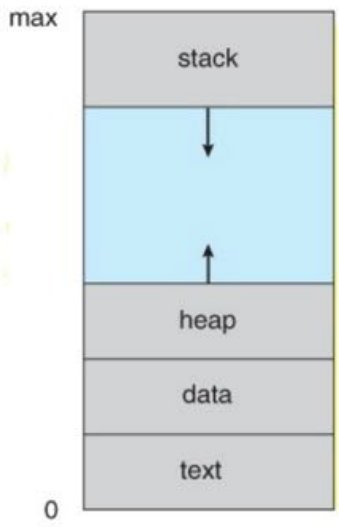
Department of Computer Engineering

Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "jobs". Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

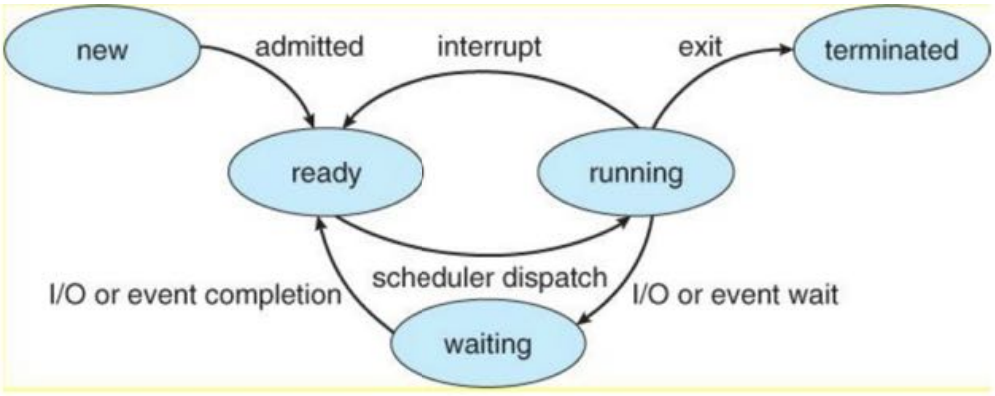
The Process

- Process memory is divided into four sections as shown in Figure 3.1 below:
    - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
    - The data section stores global and static variables, allocated and initialized prior to executing main.
    - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
    - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
    - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.



Process State

- Processes may be in one of 5 states, as shown in Figure 3.2 below.
  - **New** - The process is in the stage of being created.
  - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
  - **Running** - The CPU is working on this process's instructions.
  - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
  - **Terminated -** The process has completed.
- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

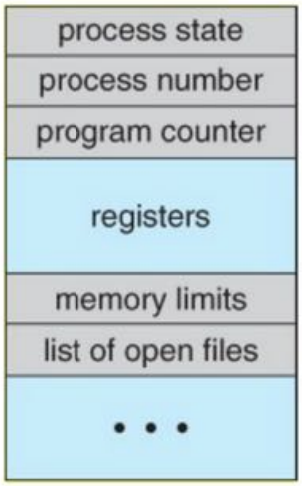


Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information.

Process State - Running, waiting, etc., as discussed above.

- **Process ID,** and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.



PROCESSES

A process is a program in execution. A process is the unit of work in a modern time–sharing system.

Many processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU among various processes, the operating system can make the computer more productive. On a single–user system, such as Microsoft Windows 95, 98 or XP, a user may be able to run several programs at one time. Windows 2000, Windows NT, 2003+ are multiuser, multi–programmer systems.

PROCESS CONTROL

Process Creation

OS creates a user process to represent the execution of a user program. This is done when a user types the name of a program to be executed. Alternatively, a user process may be explicitly created by another user process through a system call. A system process may be created by an OS for its own purposes, e.g. to service a request made by a user.

OS performs the following actions when a new process is created:

1. Creates a *process control block* (PCB) for the process.
2. Assign process id and priority
3. Allocate memory and other resources to the process
4. Set up the process environment
5. Initialize resource accounting information for the process

Setting up of process environment includes loading the process code in the memory, setting up the data space to consist of private and shared data areas of the process, setting up file pointers for standard files, etc. At the end of these actions, the state of the process is set to *ready* and the PCB of the process is entered in the data structures of the process scheduler. From this point onwards, the process will compete for CPU time in accordance with its priority.

The contents of a file stored on disk are a passive entity and thus program is a passive entity whereas process is an active entity.

A process is more than the program code. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. It also contain the process stack, which contains temporary data and a data section, which contains global variables. This entire information is called as process context.

PROCESS STATE

The current activity of a process defines the state of that process. As a process executes it changes states. Each process may be in one of the following states :

1. **Blocked**
   A process that is waiting for some event to occur before it can continue executing. Most frequently, this event is completion of an I/O operation. Blocked processes do not require the services of the CPU since their execution cannot proceed until the blocking event completes.

2. **Ready**
   A process that is not allocated to a CPU but is ready to run. A ready process could execute if allocated to a CPU.

3. **Running**
   A process that is executing on a CPU. If the system has n CPUs, at most n processes may be in the running state.

4. **Terminated or Exit**
   A process that has halted its execution but a record of the process is still maintained by the operating system

5. **New**
   The process is accepted to be entertained or handled.

Note : Only one process can be running on any processor at any instant, although many processes may be ready and blocked.

PROCESS STATE TRANSITIONS

Process state transitions can be depicted by a diagram as shown in Figure. This shows the way a process typically changes its states during the course of its execution. We can summarize these steps as follows:

(a) When you start executing a program, i.e. create a process, the operating system puts it in the list of new processes as shown by (i) in the figure. The operating system at any time wants only a certain number of processes to be in the ready list to reduce competition. Therefore, the operating system introduces a process in a new list first, and depending upon the length of the ready queue, upgrades processes from new to the ready list. This is shown by the admit (ii) arrow in the figure. Some systems bypass this step and directly admit a created process to the ready list.

(b) When its turn comes, the operating system dispatches it to the running state by loading the CPU registers with values stored in the register save area. This is shown by the dispatch (iii) arrow in the figure.

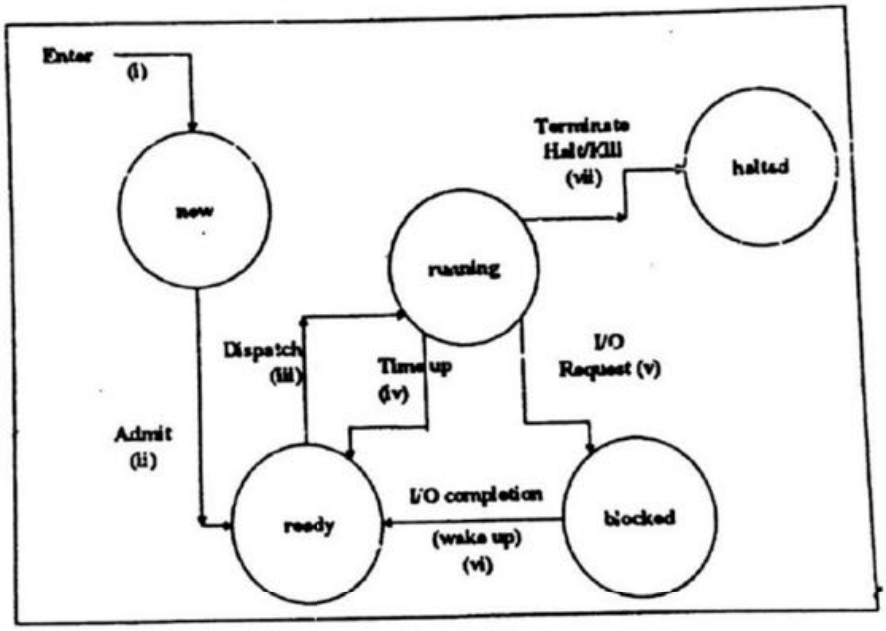


Fig. : The process state transition

(c) Each process is normally given certain time to run. This is known as time slice. This is done so that a process does not use the CPU indefinitely. When the time slice for a process is over, it is put in the ready state again, as it is not waiting for any external event. This is shown by (iv) arrow in the figure.

(d) Before the time slice is over, if the process wants to perform some I/O operation denoted by the I/O request (v) in the diagram, a software interrupts results because of the I/O system call. At this juncture, the operating system makes this process blocked, and takes up the next ready process for dispatching.

(e) When the I/O for the original process is over, denoted by I/O completion (vi), the hardware generates an interrupt whereupon the operating system changes this process into a ready process. This is called a wake up operation denoted by (vi) in the figure. Now the process can again be dispatched when its turn arrives.

(f) The whole cycle is repeated until the process is terminated.

i) **Process–id (PID) :** This is number or token allocated by the operating system to the process on creation. This is the number which is used subsequently for carrying out any operation on the process as is clear in fig. The operating system normally sets a limit on the maximum number of processes that it can handle and schedule. Let us assume that this number is n. This means that the PID can take on values between 0 and n.
The operating system starts allocating Pids from number 0. The next process is given Pid as 1, and so on. This continues till (n – 1) processes are created. At this juncture, if a new process is created, the operating system wraps around and starts again with 0.
When a process is created, a free PCB slot is selected and its PCB number itself is chosen as the Pid number.

ii) **Process state :** We have studied different process states such as running, ready etc. This information is kept in a codified fashion in the PCB.

iii) **Process priority :** Some processes are urgently required (higher priority) than others (lower priority). This priority can be set externally by the user/system manager, or it can be decided by the operating system internally depending on various parameters.

iv) **Register save area :** This is needed to save all the final registers at the context switch.

v) **Pointers to the process's memory :** This gives direct or indirect addresses or pointers to the locations where the process image resides in the memory. For instance, in paging systems, it could point towards the page map tables which in turn point towards the physical memory (indirect). In the same way, in contiguous memory systems, it could point to the starting physical memory address (direct).

vi) **Pointers to other resources :** This gives pointers to other data structures maintained for that process.

vii) **List of open files :** This can be used by the operating system to close all open files not closed by a process explicitly on termination.

viii) **Accounting information :** This gives the account of the usage of resources such as CPU time, connect time, disk I/O used, etc. by the process. This information is used especially in a data centre environment or cost centre environment where different users are to be charged for their system usage. This obviously means an extra overhead for the operating system, as it has to collect all this information and update the PCBs with it for different processes.

Other information : As an example, with regard to the directory, this contains the pathname or the BFD number of the current directory.

Pointers to other PCBs : This essentially gives the address of the next PCB (e.g. PCB number) within a specific category. This category could mean the process state. For instance, the operating system maintains a list of ready processes. In this case, this pointer field could mean " the address of the next PCB with state = ready". Similarly, the operating system maintains a hierarchy of all processes so that a parent process could traverse to the PCBs of all the child processes that it has created.

➢ PROCESS SCHEDULING

When more than one process is runnable, the operating system must decide which one to run first. The part of the operating system concerned with this is called the scheduler, and the algorithm it uses is called the scheduling algorithm.

The scheduler is concerned with deciding on policy, not providing a mechanism some of the more important criteria includes :

- **CPU Utilization**
  The percentage of time the CPU is executing a process. The load on the system affects the level of utilization that can be achieved; high utilization is more easily achieved on more heavily loaded systems. The importance of this criterion typically varies depending on the degree the system is shared. On a single–user system, CPU utilization is relatively unimportant. On a large, expensive, time–shared system, it may be the primary consideration.
- **Balanced Utilization**
  The percentage of time all resources are utilized. Instead of just evaluating CPU utilization, utilization of memory, I/O devices, and other system resources are also considered.
- **Throughput**
  The number of processes the system can execute in a period of time. Evaluation of throughput must consider the average length of a process. On systems with long processes, throughput will be less than on systems with short processes.
- **Turnaround Time**
  The average period of time it takes a process to execute. A process's turnaround time includes all the time it spends in the system, and can be computed by subtracting the time the process was created from the time it terminated. Turnaround time is inversely proportional to throughput.
- **Wait Time**
  The average period of time a process spends waiting. One deficiency of turnaround time as a measure of performance is the time a process spends productively computing increases the turnaround time, lowering the performance evaluation. Wait time presents a more accurate measure of performance because it does not include the time a process is executing on the CPU or performing I/O; it includes only the time a process spends waiting.
- **Response Time**
  On interactive systems, the average time it takes the system to start responding to user inputs. On a system where there is a dialog between process and user, turnaround time may be mostly dependent on the speed of the user's responses and relatively unimportant. Of more concern is the speed with which the system responds to each user input.
- **Predictability**
  Lack of variability in other measures of performance. Users prefer consistency. For example, an interactive system that routinely responds within a second, but on occasion takes 10s to respond, may be viewed more negatively than a system that consistently responds in 2s. Although average response time in the latter system is greater, users may prefer the system with greater predictability.
- **Fairness**
  The degree to which all processes are given equal opportunity to execute. In particular, do not allow a process to suffer form ***starvation***. A process is a victim of starvation if it becomes struck in a scheduling queue indefinitely.
- **Priorities**
  Give preferential treatment to processes with higher priorities.

A little thought will show that some of these goals are contradictory. A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either

SCHEDULING ALGORITHMS

In interactive environment e.g. as time sharing systems the primary requirement is to provide reasonable good response time share system resources equitably among all users. For this the scheduling algorithms are :

1. Round Robin Scheduling :

- One of the oldest, simplest, fairest and most widely used algorithms is **Round Robin**. Each process is assigned a time interval, called its quantum, which is allowed to run.
- If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks.
- Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes as shown below.

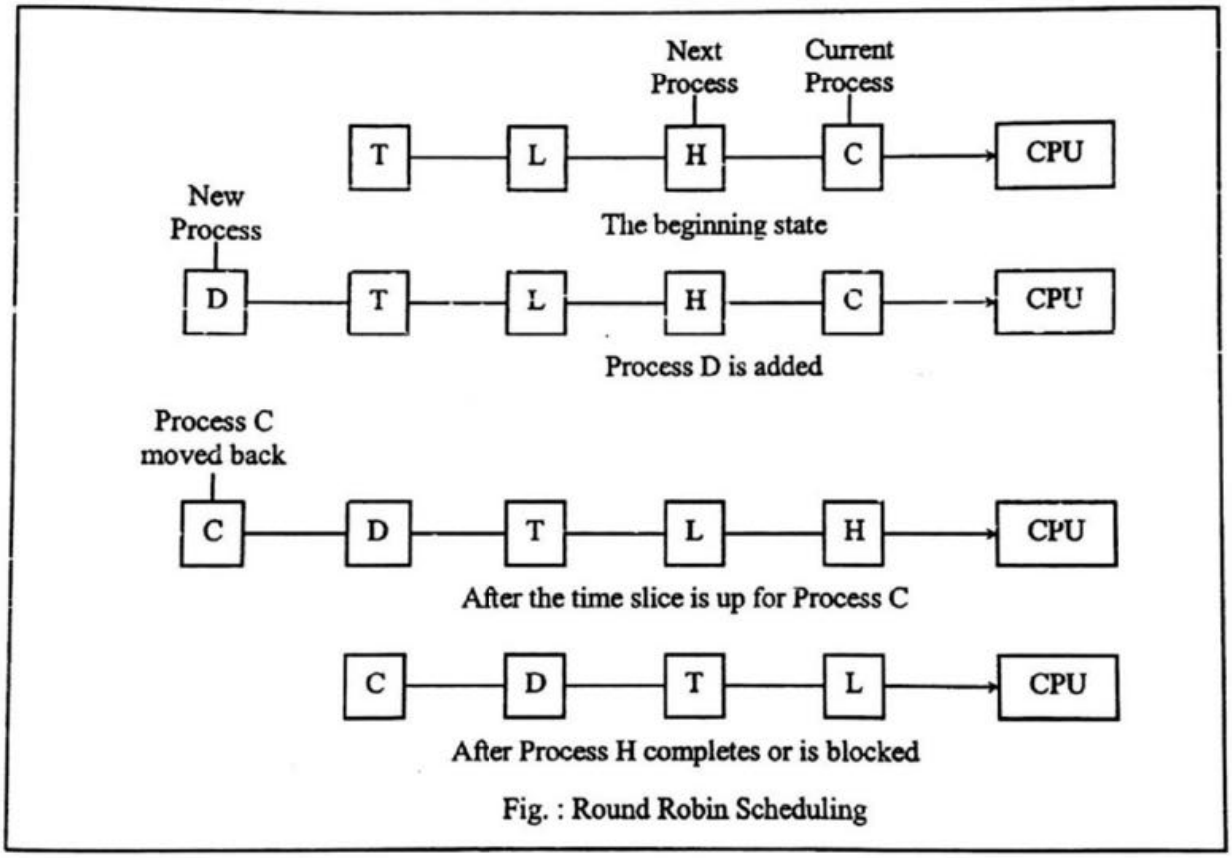


Fig. : Round Robin Scheduling

- Switching from one process to another requires a certain amount of time for doing the administration – saving, loading registers and memory maps, updating various tables and lists, etc.
- Suppose this process switch or context switch, takes 5 msec. Also suppose that the quantum is set at 20 msec. With these parameters, after doing 20 msec of useful work, the CPU will have to spend 5 msec on process switching. 20% of the CPU time will be wasted on administrative overhead.
- To improve the CPU efficiency, we would set the quantum to say 500 msec. Now the wasted time is less than 1%. But consider what happens if ten interactive users hit the carriage return key at roughly the same time. Ten processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start after time quantum and soon and so forth.

2. **Priority Scheduling :**

- The basic idea behind this is : each process is assigned a priority and the runnable process with the highest priority is allowed to run.
- This prevent high–priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e. at each interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs.
- Priorities can also be assigned dynamically by the system to achieve certain system goals.
  For example : Some processes are highly I/O bound and spend most of their time waiting for I/O to complete.
  Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing.
- Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time.

A simple algorithm for giving good service to I/O bound processes is to set the priority to 1/f, where f is the fraction of the last quantum that a process used. A process that used only 2 msec of its 100 msec quantum would get priority 50, while a process that ran 50 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

Priority Class

According to this philosophy, the operating system allows you only a limited priority classes instead of very large possibilities of priority numbers. It then essentially splits the chain of ready processes into as many different PCB chains as there are priority classes.

Within each priority class, you could have different scheduling policies. You could run all of them round robin for instance. In this case, after a process consumes its time slice, the PCB is linked at the end of the chain for that priority class instead of linking at the end of all PCBs in ready chain. If a process gets blocked, it is put into the blocked queue and after the I/O completion, it is reintroduced at the end of the ready queue for the same priority class that i

originally belonged to. If the currently running process consumes the full time slice, it is introduced at the end of the ready queue for the same priority class.

- It is often convenient to group processes into priority classes and use priority scheduling among the classes but round robin scheduling within each class.
- The following figure shows a system with four priority classes.

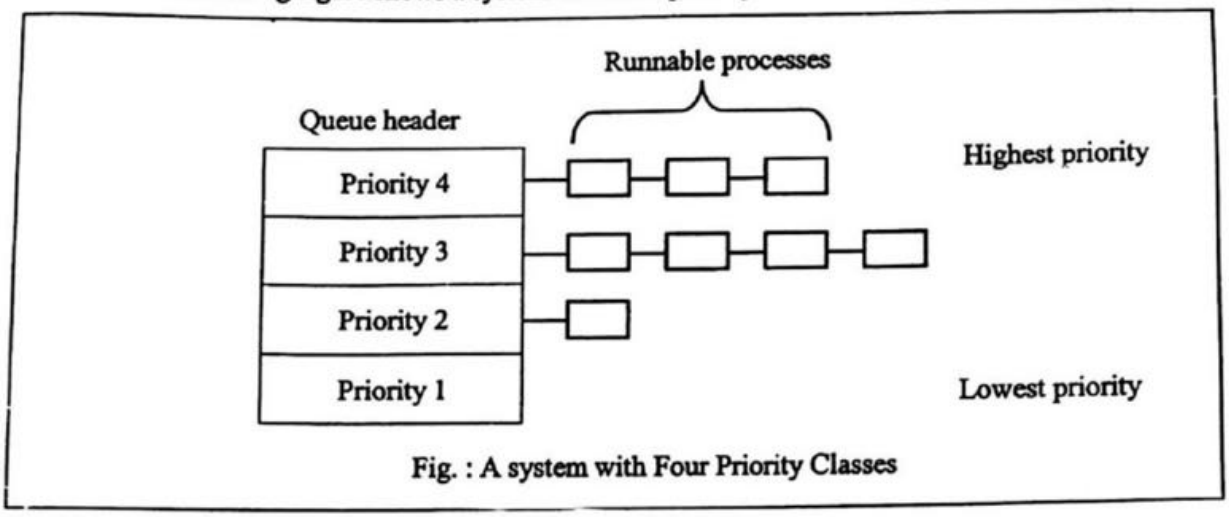


Fig. : A system with Four Priority Classes

As long as there are runnable processes in priority class 4, just run each one for one quantum, round robin fashion and never bother with lower priority classes until priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted from time to time, lower priority classes may all starve to death.

- **Priorities can be defined either internaliy or externally :**

  Internal priorities : The priority of a process is defined using some measurable internal factors like time limits, memory requirements, number of open files.

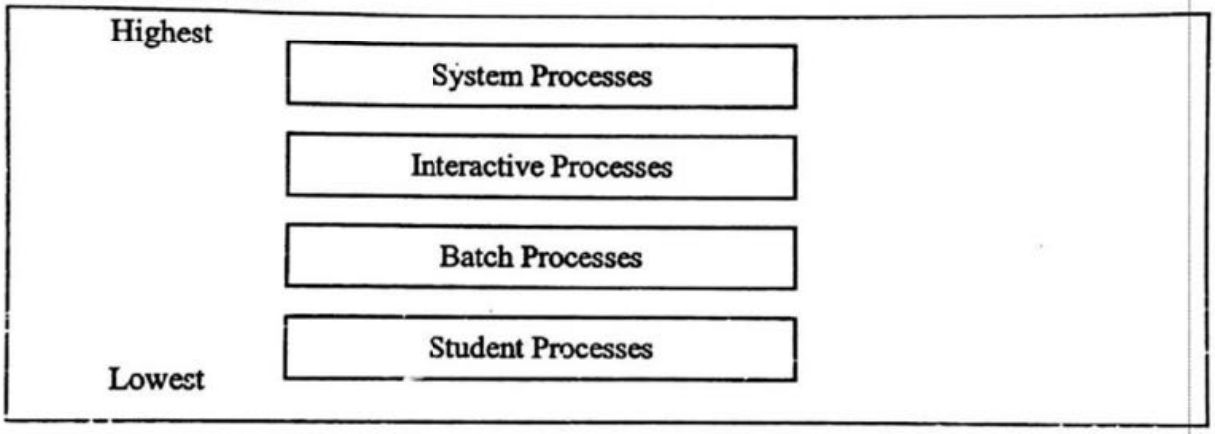
  External priorities : They are set up by criteria external to the operating system such as importance of the process, type and amount of funds being paid for computer use, department sponsoring the work and other organization factors.

- **Priority scheduling can be either pre–emptive or non pre–emptive :**
  When scheduling is preemptive, a newly arrived processes priority is compared with an already existing priority, if it is higher, the current process is replaced, else the process continues.
  When nonpreemptive, the newly arrived process is put at the head of the priority queue.

- **Requirements for a priority based scheduling :**
  i) Maintain a chain of PCBs in the ready state in the priority order
  ii) Modify the chain after any process is added or deleted or if the priority is changed externally or internally for any progress.
  iii) Recalculate the priorities at the appropriate time. For instance for preemptive philosophy, it could be at every clock tick. For others, it could be at the context switch due to the I/O completion or termination of the current process. After the recalculation, adjust the PCB chains appropriately.
  iv) Invoke the "Dispatch a process" system call (to dispatch the highest priority process) after the recalculation of priorities at the appropriate time depending upon the philosophy as discussed above.

3. **Multilevel Queue Scheduling :**
   - This type of scheduling separates the available processes into different groups. Classification is done on the basis of the fact that different types of processes have different response requirement and hence may require different scheduling.

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to the one queue, generally based on some property of the process, such as memory size, process priority etc.
- Each queue has its own scheduling algorithm.
  e.g. Foreground queue may be scheduled by Round Robin algorithm while background queue may use FCFS. In addition, there must be scheduling between the queues which is commonly implemented as fixed priority preemptive scheduling.
- **Example:** Consider the following queues:
  1. System Processes
  2. Interactive Processes (eg : Editor – m – word 97 & beyond)
  3. Batch Processes
  4. Student Processes



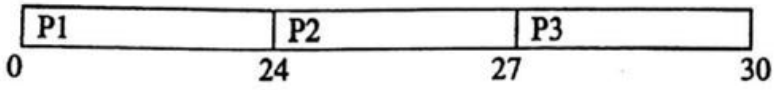
- Each queue has absolute priority over lower priority queues. Hence, no process from the batch queue can run unless the preceeding three queues are empty.
- If an interactive process entered the ready queue while a batch process was executing, the batch process would be terminated.
- Also we can use time slicing between the queues where in each queue would get a certain portion of the CPU time, which can then schedule among its various processes.

First Come – First Served Scheduling :

- The average waiting time under the FCFS policy, however is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU – burst tim given in milliseconds

Process	Burst time
P1	24
P2	3
P3	3

- If the processes arrive in the order P1, P2, P3 and are served in FCFS order, we get the result shown in the following Gantt Chart.



The average waiting time = $\frac{0+24+27}{3}$ = 17 milliseconds

- Now, the waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 and 27 milliseconds for process P3. If the processes arrive in the order P2, P3, P1, the result will be as shown in the following Gnatt Chart:

Now, the average waiting time = $\frac{6+0+3}{3}$ = 3 milliseconds.

This reduction is substantial. Thus, the average waiting time under a FCFS is generally not minimal.

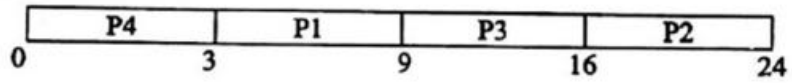
- The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, it releases the CPU, either by terminating or by requesting I/O.

Shortest Job first Scheduling (SJF)

- Consider the following set of processes with the length of the CPU burst time given in milliseconds :

Process	Burst time
P1	6
P2	8
P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt Chart:



- The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.

Thus, the average waiting time = $\frac{3+16+9+0}{4}$ = 7 milliseconds

- The SJF scheduling algorithm is provably optimal. In that it gives the minimum average waiting time for a given set of processes. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request. One approach is to try to approximate SJF scheduling.

The next CPU burst is generaliy predicted as an exponential average of the measured length of previous CPU bursts.

Let t_n → length of the n^{th} burst

τ_{n+1} → predicted value for the next CPU burst

then for, α, $0 \le \alpha \le 1$, define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- The SJF algorithm may be either preemptive or non preemptive

A preemptive SJF algorithm will preempt the currently executing process, whereas a unpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called Shortest – remaining – time – first Scheduling.

Difference between Pre–emptive and Non–Pre–emptive Scheduling :

	Pre–emptive Scheduling		Non–Pre–emptive Scheduling
1.	A pre–emptive philosophy allows a higher priority process to replace an already existing process even if it's time slice is not over.	1.	A non–pre–emptive process means that a running process retains the control of the CPU and all allocated resources until it surrenders control to the OS on its own.
2.	This requires context switching more frequently.	2.	This requires context switching less frequently.
3.	It reduces throughput for the system, but utilizes in interactive processes.	3.	Higher throughputs due to less overheads incurred in context switching.
4.	Suited for use in real time systems.	4.	Suited for use in process which require a large amount of CPU time.
5.	Used in front office application (banks, hospitals, railway reservation etc.)	5.	Used in back office (business application)

➢ SYNCHRONIZATION

With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. A common synchronization mechanism used in multiprocessor OS is locks.

➢ MUTUAL EXCLUSION

Suppose two or more processes require access to a single sharable resource. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data and receiving data. Such a resource is called critical resource and the portion of the program that uses it is called as critical section of the program. **Mutual Exclusion** does not allow any other process to get executed in their critical section, if a process is already executing in its critical section.

Mutual Exclusion : Hardware support

i) Interrupt Disabling

In a uniprocessor machine, concurrent processes cannot be overlapped, instead they can be interleaved. A process will continue to run until it invokes an OS service or until it is interrupted. Thus to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. Each process should disable all interrupts just after entering its critical region and re–enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene. A process can then enforce mutual exclusion in the following way.

```
while (true)
{
/ * disable interrupts     * / ;
/ * critical section       * / ;
/ * enable interrupts      * / ;
/ * remainder              * / ;
}
```

Because the critical section cannot be interrupted mutual exclusion is guaranteed.

Disadvantages :

1. The price of this approach is high.
2. The efficiency of execution is degraded because the processor is limited in its ability to interleave programs.
3. In a multiprocessor architecture, more than one process is executing at a time. Thus disabled interrupts do not guarantee mutual exclusion.
4. It only works in a single–processor environment.
5. Interrupts can be lost if not serviced promptly. A process remaining in its critical section for any longer than a brief time would potentially disrupt the proper execution of I/O operations.
6. Exclusive access to the CPU while in the critical section could inhibit achievement of the scheduling goals.
7. A process waiting to enter its critical section could suffer from starvation.

ii) Special Machine Instructions

At a hardware level, access to a memory location excludes any other access to the same location. Thus processor designers have proposed several machine instructions that carry out two actions automatically, such as reading and writing or reading and testing, of a single memory location

with one instruction fetch cycle. Because these actions are performed in a single instruction cycle, they are not subject to interference from other instruction

(a) Test and set Instruction :

The test and set instruction can be defined as follows :

```
Boolean test set (int i)
    {   if (i = = 0)
        {   i = 1 ;
            return true ;
        }
        else
        {
            return false ;
        }
    }
```

The instruction tests the value of its argument i. If the value is 0, then it replaces it by 1 and returns true. Otherwise, the value is not changed and false is returned. The entire test set function is carried out automatically. When it returns true all the other processes upon executing test–and–set instruction find i = 1 and continue looping. The process using the resource resets the control variable i = 0 when finished. One of the processes looping on i gains access by observing i = 0.

(b) Exchange Instruction :

The exchange instruction can be defined as follows :

```
Void exchange (int register, int memory)
{
    int temp;
    temp        =   memory ;
    memory      =   register ;
    register    =   temp ;

}
```

The instruction exchanges the contents of a register with that of a memory location. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location.

Advantages

1. It is applicable to any number of processes in either a single processor or multiple processors sharing main memory.
2. It is simple and easy to verify.
3. It can be used to support multiple critical section, each of which can be defined by its own variable

Disadvantages

1. Busy waiting continues to consume processor time.
2. When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary which may result into starvation.
3. Deadlock is possible.

SEMAPHORES

A semaphore is a protected variable which can be accessed and changed only by operations such as wait "DOWN" (or P) and signal "UP" (or V). Semaphore can be counting or general, where it can take on any positive value. Binary semaphore can take values of only 0 or 1. Semaphores can be implemented in software as well as hardware.

One can view the semaphore as a variable that has an integer value upon which three operations are defined :

i) A semaphore may be initialized to a non–negative value.

ii) The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.

iii) The signal operation increments the semaphore value. If the value is not negative, then a process blocked by a wait operation is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores.
Semaphore is to count the number of wakeups saved for future use. A semaphore could have the value 0, indicating that no wakeups were saved or some positive value if one or more wakeups were pending.

```
struct semaphore
{
        int count ;
        queue type  queue ;
}
void wait (semaphore s)
{
    s. count – – ;
    if (s. count < 0)
    {
        place this process in s. queue ;
        block this process
    }
}
void signal (semaphore s)
{
    s. count + + ;
    if (s. count < = 0)
    {
        remove a process P from s. queue ;
        place process P on ready list ;
    }
}
```

Fig. : Mutual exclusion with semaphores

The wait and signal primitives are assumed to be atomic, i.e., they cannot be interrupted and each routine can be treated as an indivisible step.

Concept of Semaphores

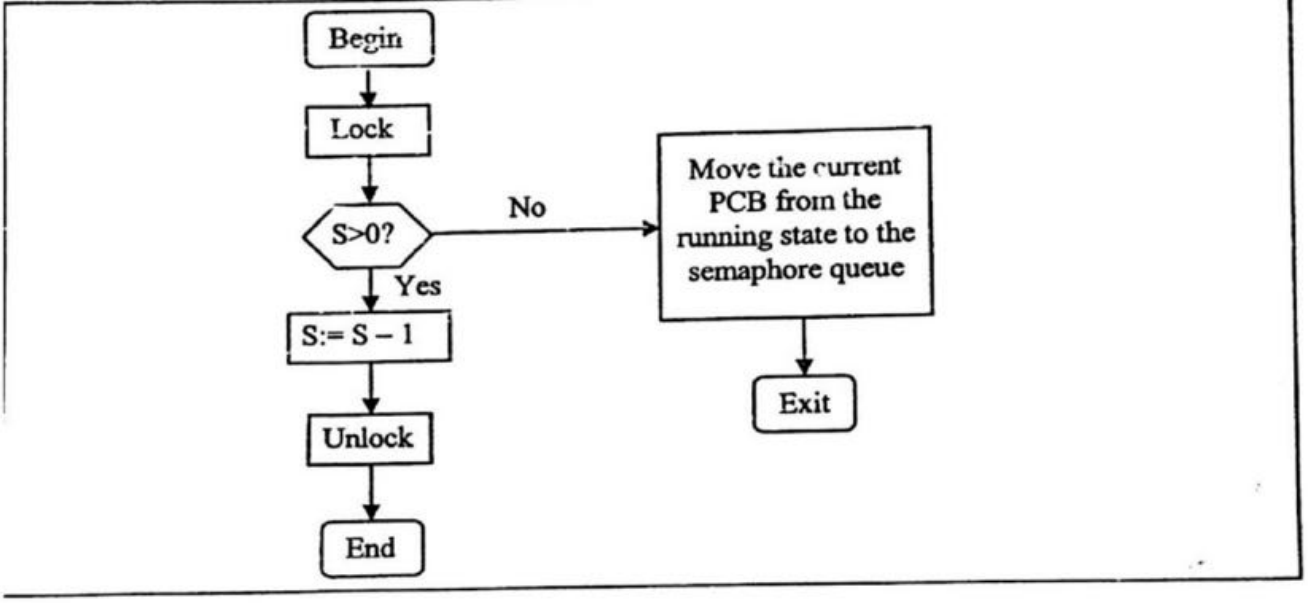
"DOWN and UP" form the mutual exclusion primitives for any process. Thus, if a process has a critical region, it has to be encapsulated between these DOWN and UP instructions, which is shown below :

```
Begin
0.  Initial routine;
1.  DOWN(S);
2.  Critical–Region;
3.  UP(S);
4.  Remaining portion
End
```

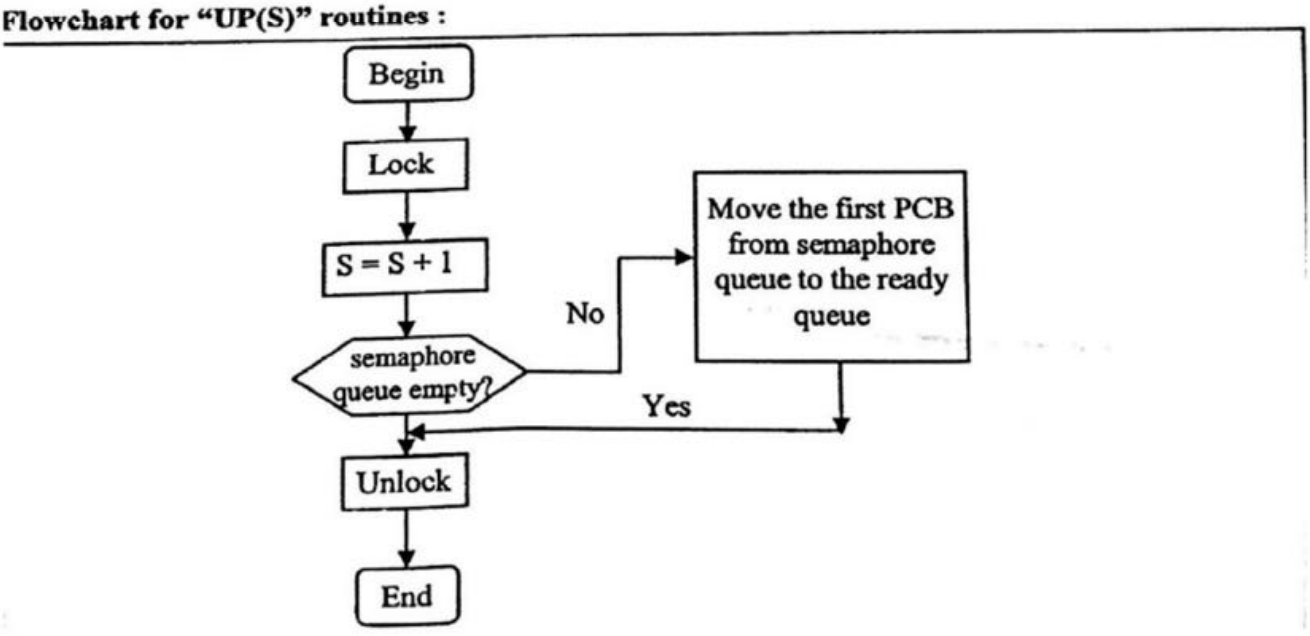
Fig. General Structure of a process with semaphores

All other processes wanting to enter their respective critical regions are kept waiting in a queue called a **"Semaphore queue"**. Only when a process which is in its critical region comes out of it, should the OS allow a new process to be released from the semaphore queue.

Flowchart for "DOWN(S)" routines :



Flowchart for "UP(S)" routines :



Algorithm for DOWN(S) :

```
        DOWN(S)
        Begin
D.0         Disable interrupts;
D.1         If  S > 0
D.2                 then S : = S – 1
D.3                 else Wait on S
D.4         Endif
D.5         Enable interrupts
        End
```

Algorithm for UP(S)

```
        UP(S)
        Begin
U.0         Disable interrupts;
U.1         S : = S + 1;
U.2         If Semaphore queue NOT empty
U.3                 then release a process
U.4         Endif
U.5         Enable interrupts
        End
```

We enable and disable interrupts, so that no process switch can take place.

Types of Semaphores

Binary Semaphore :
More restricted version of semaphore is known as the binary semaphore. A binary semaphore may only take on the values 0 and 1. Thus its implementation is easier and has the same expressive power as the general semaphore. For both semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.

Strong Semaphore :
For any semaphore, a queue is used to hold processes waiting on the semaphore. There are many orders in which processes are removed from such a queue. The fairest policy is first–in first–out (FIFO). The process that has been blocked for a longer time is released from the queue first. A semaphore whose definition includes this policy is called a strong semaphore.

Weak Semaphore :
For any semaphore, a queue is used to hold processes waiting on the semaphore. A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore.
Strong semaphores guarantee freedom from starvation but weak semaphores do not. Strong semaphores are convenient and it is a form of semaphore typically provided by OS.

Basic Principles of Semaphore Working

i) Unless a process executes a DOWN(S) routine successfully without getting added to the semaphore queue at instruction 1, it cannot get into its critical region at instruction 2.
ii) S is a binary semaphore which can take value only as 0 or 1. Let us decide that a process can enter its critical region only if S = 1. As shown in the flowchart, if S is greater than 0, it is reduced by 1 and it becomes 0 in the DOWN routine (instruction 1) itself. Then only, the process is allowed to enter its critical region at instruction 2. Thus if any process is in its critical region, then S must be 0.

PRODUCER CONSUMER PROBLEM

Introduction

Process is a program in execution. Several processes need to communicate with one another simultaneously, which requires proper synchronization and use of shared data residing in shared memory locations.

Producer Process

Assume that there are multiple users at different terminals running different processes, but each one running the same program. This program prompts for a number from the user and on receiving it, deposits it in a shared variable at some common memory location. As these processes produce some data, they are called "Producer Processes".

Consumer Process

Again imagine that there is another process which picks up this number as soon as any produces process outputs it and prints it. This process which uses or consumes the data produced by the producer process is called "Consumer Process".

Note :

All the producer processes communicate with the consumer process through a shared variable where the shared data is deposited.

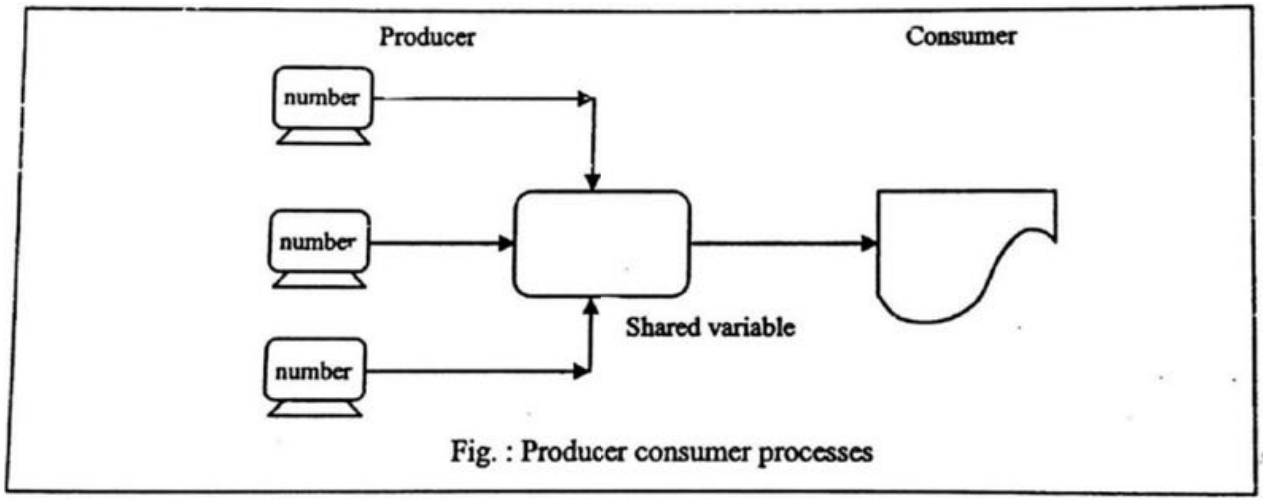


Fig. : Producer consumer processes

The Problem

Two processes share a common, fixed–size buffer. One of them, the producers, puts information into the buffer and the other one, the consumer, takes it out.

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

To keep track of the number of items in the buffer, we need a variable, count. If the maximum number of items the buffer can hold is N, the producer's code finds count as N, then producer will go to sleep. If it is not so, the produces will add an item and increment count. Same is true with consumer's code.

Race Condition

Suppose the buffer is empty and the consumer has just read count to see if it is 0. Thus the scheduler decides to stop running the consumer temporarily and start running the producer. The producer enters an item in the buffer, increment count and notices that it is now 1. As count was just 0, and thus the consumer must be sleeping, the produce calls wakeup to wake the consumer up.

The consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

Note :

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost.

Solution :

Algorithm for the producer consumer problem using Event–counters

1. Two event counters are used.
   i) ***in*** **:** counts the cumulative number of items that the producer has put into the buffer since the program started running.
   ii) ***out*** **:** counts the cumulative number of items that the consumer has removed from the buffer so far.

   Note :

   in must be greater than or equal to *out*, but not by more than the size of the buffer.
2. When the producer has computed a new item, it checks to see if there is room in the buffer, using the AWAIT system call.
   - Initially, *out* = 0 and sequence – N will be negative so the producer does not block.
     (where : sequence → number of items in the buffer
     N → number of slots in the buffer)
   - If the producer generates N + 1 items before the consumer has begun, the AWAIT statement will wait until *out* becomes 1.

   Note : This will only happen after the consumer has removed one item.

Solving the Producer – Consumer Problem using semaphores

This solution uses three semaphores :

i) '*full*' for counting the number of slots that are full. *full* is initially 0.
ii) '*empty*' for counting the number of slots that are empty. *empty* is initially equal to the number of slots in the buffer.
iii) '*mutex*' to make sure the producer and consumer do not access the buffer at the same time *mutex* is initially 1.

Note : Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores.

- If each process does a DOWN just before entering its critical region and an UP just after leaving it, mutual exclusion is guaranteed.
- The mutex semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent confusion.
- The other use of semaphores is for synchronization. The full and empty semaphores are needed to guarantee that certain event sequence do or do not occur. Thus they ensure that the producer stop running when the buffer is full and the consumer stops running when it is empty.