GATE

Be sure with Vidyalankar Since 1960

Pearl Centre, S.B. Marg, Dadar (W), Mumbai – 400 028. Tel. 4232 4232

CS : COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Operating Systems

INDEX

Sr. No.	Contents	Topics	Pg. No.
1.	**Processes and IPC**		
	Notes	Processes	1
		Process Control Block	4
		Schedulers	6
		Context Switch	7
		CPU Scheduling	10
		Process Scheduling	11
		Scheduling Algorithms	13
		The Convoy Effect	23
		Interprocess Communication	24
		Race Condition	24
		Critical Sections	25
		Concurrency	27
		Synchronization	28
		Mutual Exclusion	28
		Sleep and Wakeup	33
		Event Counters	33
		Semaphores	34
		Monitors	38
		Producer Consumer Problem	40
		Message Passing	43
		Readers / Writers Problem	47
		Starvation	48
		Deadlock	49
		Dining Philosophers Problem	57
		LMR (Last Minute Revision)	60
	Assignment–1	Questions	63
	Test Paper–1	Questions	67

Sr. No.	Contents	Topics	Pg. No.
2.	**Memory Management**		
	Notes	Introduction	70
		Bare Machine	71
		Resident Monitor	72
		Swapping	75
		Multiple Partitions	76
		Paging	85
		Segmentation	88
		Overlays	90
		Demand Paging	93
		Virtual Memory	95
		Page Replacement Algorithms	99
		Allocation Algorithms	107
		Thrashing	107
		LMR (Last Minute Revision)	110
	Assignment–2	Questions	112
	Test Paper–2	Questions	116
3.	**File System**		
	Notes	Introduction	119
		File Concept	119
		Directories	127
		Access Methods	133
		Allocation Methods	135
		File Protection	143
		LMR (Last Minute Revision)	147
	Assignment–3	Questions	148
	Test Paper–3	Questions	151
Practice Problems		Questions	154
SOLUTIONS			
Assignment		Answer Key	163
		Model Solutions	164
Test Paper		Answer Key	174
		Model Solutions	175
Practice Problems		Answer Key	183
		Model Solutions	184

Topic 1 : Processes and IPC

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.

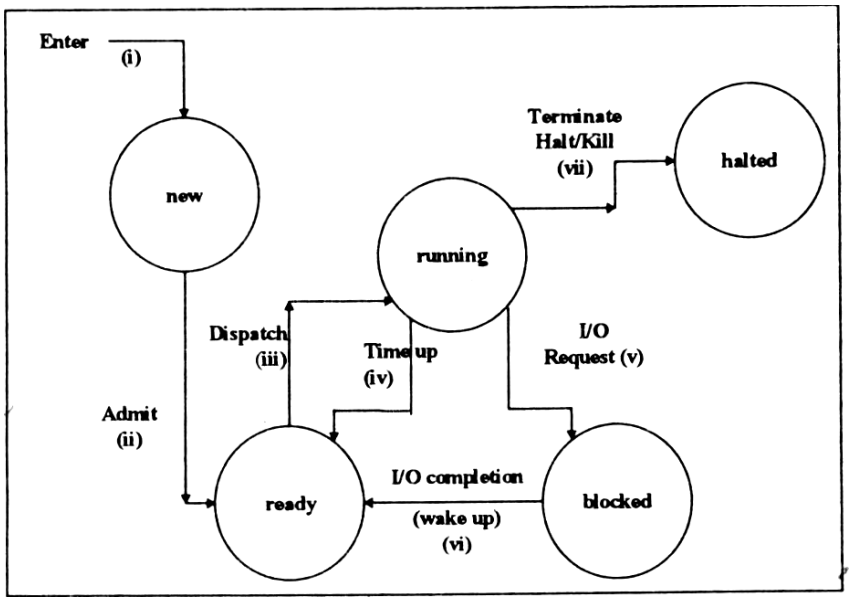


Fig. : The process state transition

(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.

Events Pertaining to a Process

Process state transitions are caused by the occurrence of events in the system. A sample list of events is as follows:

1. *Request event:* Process makes a resource request
2. *Allocation event:* A requested resource is allocated.
3. *I/O initiation event:* Process wishes to start I/O.
4. *I/O termination event:* An I/O operation completes.
5. *Timer interrupt*: The system timer indicates end of a time interval.
6. *Process creation event:* A new process is created.
7. *Process termination event:* A process finishes its execution.
8. *Message arrival event:* An interprocess message is received.

An event may be *internal* to a *running* process, or it may be *external* to it. For example, a request event is internal to the *running* process, whereas the allocation, I/O termination and timer interrupt events are external to it. When an internal event occurs, the change of state, if any, concerns the *running* process. When an external event occurs, OS must determine the process affected by the event and mark an appropriate change of state.

Solved Examples

1. Process context contains
   (a) Process code only
   (b) Process code, register values, local variables, stack pointer
   (c) The entire program
   (d) Process Control Block (PCB)

Ans : (b)
This is since the OS should be able to revert back to the calling program and restore to the same state which the calling program was before switching.

2. Windows 95 is
   (a) Multi tasking, Multiuser
   (b) Multitasking only
   (c) Multiprocessing, Multiuser
   (d) None of the above

Ans : (b)

PROCESS CONTROL BLOCK

The operating system maintains the information about each process in a record or a data structure with fields for recording various aspects of process execution and resource usage called Process Control Block (PCB) as shown in the fig. Each user process has a PCB. It is created when a user creates a process and it is removed from the system when the process is killed. All these PCBs are kept in the memory reserved for the operating system.

Process–id
Process state
Process priority
Register Save Area
for PC, IR, SP, . . . for PC, IR, SP, . . . …. ….
Pointers to process's memory
Pointers to other resources
List of open files
Accounting information
Other info if required (current Dir)
Pointers to other PCBs

Fig. : Process Control Block

Let us now study the fields within a PCB. The fields are as follows:

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.

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

x) 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.

3. Accounting information is used for what purpose ?
   (a) To produce statistical reports
   (b) For keeping track of logged activities
   (c) To ensure restriction are obeyed
   (d) All of the above

Ans : (d)
Used by the OS to find processor efficiency in a given environment.

SCHEDULERS

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select processes from these queues in some fashion for scheduling purposes. The selection process is carried out by the appropriate scheduler.

Types of Schedulers :

1. Long term scheduler 2. Medium term scheduler 3. Short term scheduler

1. Long term Scheduler or job scheduler:

In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass storage device, where they are kept for later execution. The long term scheduler or job scheduler selects processes from this pool and loads them into memory for execution. Once the job scheduler makes a job active, it stays active until it terminates.

2. Medium term Scheduler or swapper :

At any time the main memory is limited and can hold only a certain number of processes. If the availability of the main memory is limited then the best option would be to swap it out and put it in another queue called swapped out and blocked which is different from only blocked processes that remain in memory hence requires a separate PCB chain.

When the I/O request is completed, the process is pushed into yet another queue called swapped out but ready state. This also requires a separate PCB chain.

When some memory is freed the OS looks at the list of swapped out but ready processes and depending on the priority, memory and other resources required links that PCB in the chain of ready processes required. This is done by medium term scheduler.

In the figure shown, a portion of suspended processes are assumed to be swapped out.

3. Short term Scheduler or dispatcher

The short term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them. The long–term scheduler controls the degree of multiprogramming i.e. the number of processes in memory.

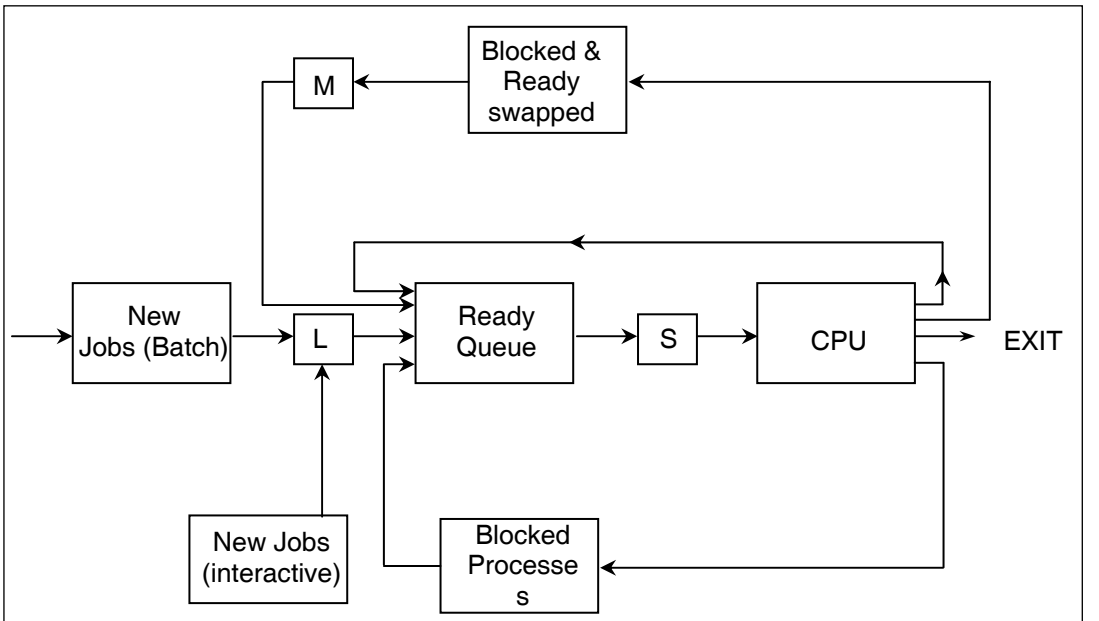


Fig. Showing Function of Schedulers

4. Where does the dispatcher reside ?
   (a) In RAM or main memory
   (b) In Cache Memory
   (c) In HDD
   (d) In external / auxillary memory device

Ans. (a)
The main memory maintains a buffer to hold the kernel and dispatcher. The program execution takes place in PWM.

CONTEXT SWITCH

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch.

- The context of a process is represented in the PCB of the process, it includes the value of the CPU registers the process state and memory management information.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context – switch times are highly dependent on hardware support. Its speed varies form machine to machine, depending on the memory speed, the number of registers that must be copied and the existence of special instructions.
- Context switch time is pure overhead, because the system does no useful work while switching.

OPERATIONS ON PROCESSES

The processes in the system can execute concurrently and they must be created and deleted dynamically. Various operations on processes are:

a) Process Creation :

- Create–process system call, may create several new processes during the course of execution.
- The creating process is called a parent process, whereas the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.
- The parent may have to partition its resources among its children, or it may be able to share some resources among several of its children. A subprocess may be able to obtain its resources directly from the OS.

When a process creates a new process, two possibility exist in terms of execution :

(i) The parent continues to execute concurrently with its children
(ii) The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process :

(i) The child process is a duplicate of the parent process.
(ii) The child process has a program loaded into it.

b) Process Termination :

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the 'exit' system call.
- When process terminates, it returns data (output) to its parent process (via the "wait" system call).

  Resources like physical and virtual memory, open files and I/O buffers are deallocated by the operating system.

  A parent may terminate the execution of one of its children for a variety of reasons, such as these :

  i) The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.

  ii) The task assigned to the child is no longer required.

  iii) The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

Note : If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination is normally initiated by the OS.

Independent Process :
A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share any data (temporary or persistent) with any other process is independent.

Cooperating Process :
A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.

Note : The concurrent processes executing in the OS may be either independent processes or cooperating processes.

Process Cooperation is required for following reasons :

(i) Information sharing :

Since several users may be interested in the same piece of information. One must provide an environment to allow concurrent access to these types of resources.

(ii) Computation speedup :

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements.

(iii) Modularity :

One can construct the system in a modular fashion by dividing the system functions into separate processes or threads.

(iv) Convenience :

Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing and compiling in parallel.

5. Define context switch.

Ans.

Saving state information of process P1 and switching to another process P2.

6. Process context is stored in
   (a) LIFO data structure
   (b) FIFO data structure

Ans: (a)
LIFO data structure also called stack maintained by the OS, in the PWM.

CPU SCHEDULING

- CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.
- The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request.
  In a simple computer system, the CPU would then just sit idle. All this waiting time is wasted; no useful work is accomplished.
- With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process may take over the use of the CPU.

CPU – I/O BURST CYCLE

The success of CPU scheduling depends on the following observed property of processes :

Process execution consists of a cycle of CPU execution and I/O. Processes alternate back and forth between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.

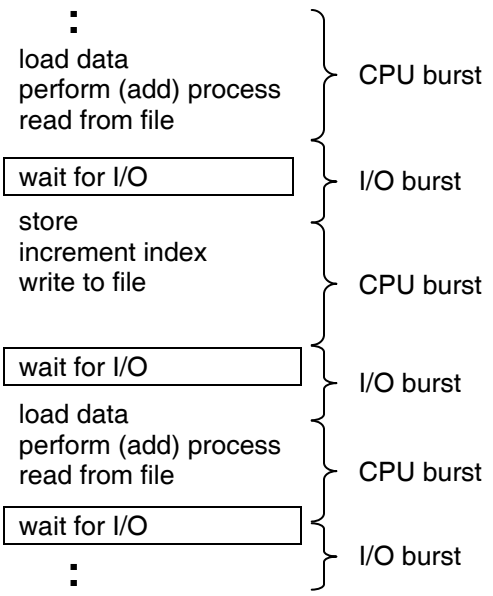


Fig. Alternating sequence of CPU and I/O bursts

7. Increase in main memory would increase the degree of multiprogramming
   (a) True
   (b) False

Ans : (a)

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 for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically. At each clock interrupt, the operating system gets to run and decide whether the currently running process should be allowed to continue, or whether it has had enough CPU time for the moment and should be suspended to give another process the CPU.

8. Goodput is different from throughput because . . . . . .

Ans.

Goodput measures the effective number of processes successfully in a unit of time. Throughput measures the number of processes executed per unit time irrespective whether they produce results or not.

Scheduling Philosophies

There are basically two scheduling philosophies. **Non–preemptive and Preemptive**. Depending upon the need, the Operating system designers have to decide upon one of them.

A **non preemptive** philosophy means that a running process retains the control of the CPU and all the allocated resources, until it surrenders control to the operating system (on its own). This means that, even if a higher priority process enters the system, the running process cannot be forced to give up the control.

However, if the running process becomes blocked due to any I/O request, another process can be scheduled because the waiting time for the I/O completion is too high. This philosophy is better suited for getting a higher throughput due to less overhead incurred in context switching, but it is not suited for real time systems, where higher priority events need an immediate attention and therefore, need to interrupt the currently running process.

A **pre–emptive** philosophy on the other hand allows a higher priority process to replace a currently running process, even if its time slice is not over or it has not requested for any I/O. This requires context switching more frequently, thus reducing the throughput, but then it is better suited for on–line, real time processing, where interactive users and high priority processes require immediate attention.

Preemptive Scheduling

A preemptive server can be switched to the processing of a new request before completing the processing of a request scheduled earlier. Thus, scheduling can be performed asynchronously with the processing of requests on the server. If a scheduling decision is made while some request R_i is being processed, processing of R_i is preempted and R_i is put back into the list of pending requests R_i would be resumed sometime in future when it is scheduled again. Thus, a request may have to be scheduled many times before completing.

Preemptive scheduling is motivated by a desire to improve system performance. Preemption can be used in one of two ways. A process may be preempted when it enters a passive state to free the server for processing another request. For example, in a reservation system, a process may need additional information from a user while processing his request. The process may be preempted at the start of the passive phase, and may be scheduled again when it enters the active phase. Preemption is also useful in another context – it permits the scheduler to respond to developments within the system, e.g. arrival of new requests, changes in process priorities etc.

Thus although nonpreemptive scheduling algorithms are simple and easy to implement, they are usually not suitable for general purpose systems with multiple competing users. On the other hand, for a dedicated system such as a database system, it may well be reasonable for the master process to start a child process working on a request and let it run until it completes. The difference with the general purpose system is that all processes in the database system are under the control of a single master, which knows what each child is going to do and about how long it will take.

9. What type of scheduling is preferable for a auction site like ebay.com?
   (a) Non preemptive (b) Preemptive

Ans : (b)
In any interactive process, to maintain a good effective response time, preemptive scheduling is used. ebay.com is a highly interactive e–commerce site.

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.

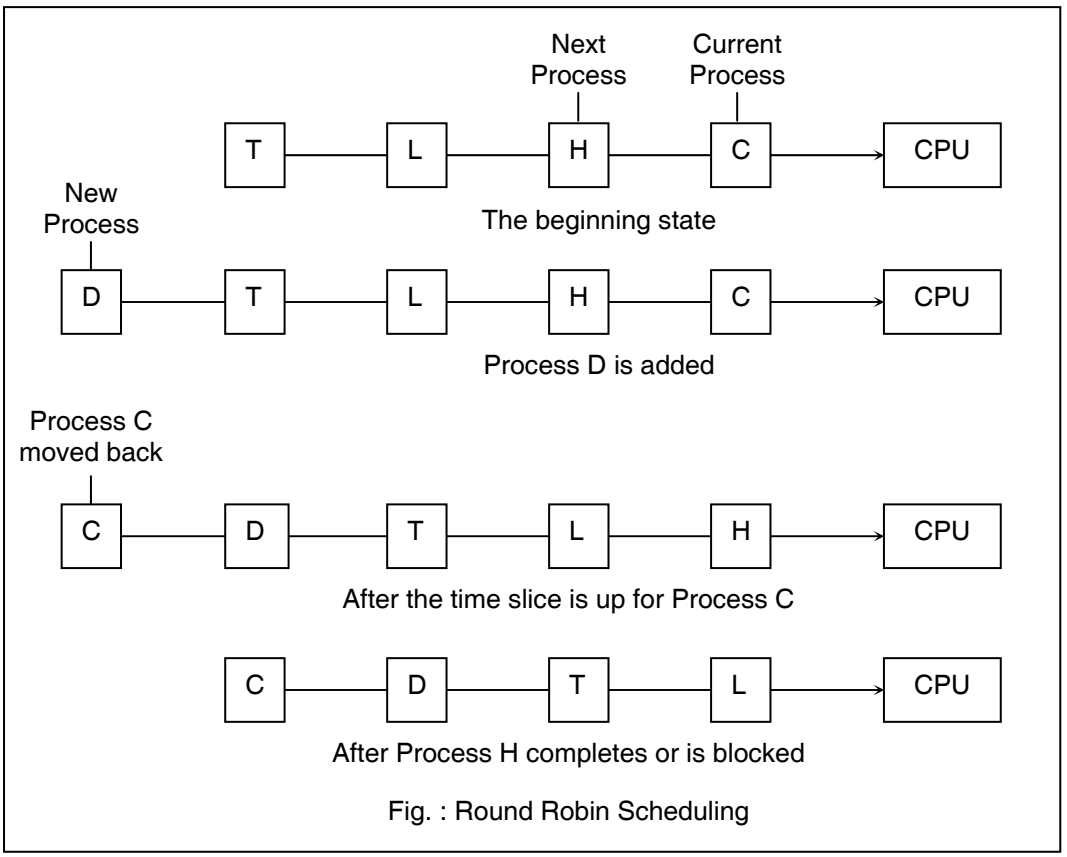


Fig. : Round Robin Scheduling

- 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.

Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.

A quantum around 100 msec is often a reasonable compromise.

10. Appropriately set time quantum usually leads to
    (a) Both CPU efficiency
    (b) Lesser context switches
    (c) Improves response time
    (d) None of the above
    (e) All of the above

Ans : (e)

11. If the quantum time is too long, Round robin degenerates
    (a) SJF
    (b) FCFS
    (c) SRT
    (d) Depends on sequence of jobs

Ans : (b)
In a long time quantum the process execution finishes before the time slot is elapsed and processor remains idle, thereby dropping the efficiency to FCFS.

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 it 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.

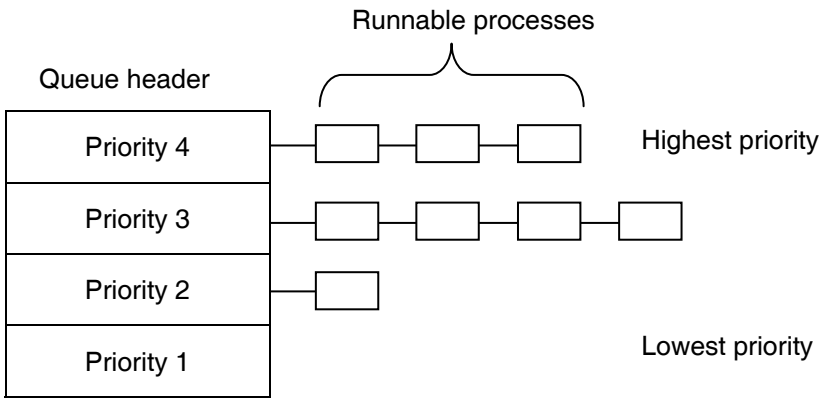


Fig. : A system with Four Priority Classes

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 internally 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.

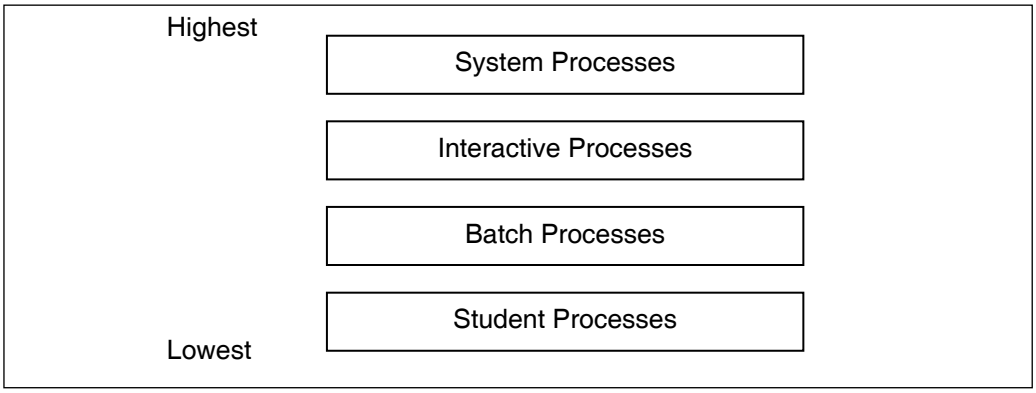
  iii) Recalculate the priorities at the appropriate time. For instance for preemptive philosophy, it could be at every clock tick. For others, it could be at the context switch due to the I/O completion or termination of the current process. After the recalculation, adjust the PCB chains appropriately.

  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.

4. Heuristic Scheduling

Let us take a CPU bound program performing mainly calculations as an example. Let us assume that after each I/O, it does calculations for say 200 milliseconds before requesting the I/O for the next record. In a round robin policy, if the time slice for this process is, say 25 ms, then during the calculation phase, it will have to give up the CPU 8 times due to the 'time up' situation. Eight times, the operating system will have to incur overheads due to context switching, hence, decreasing the throughput. Is increasing the time slice the solution ? If we increase the time slice, it will be good for this process, but then this big a time slice may be quite unnecessary for the other processes. You need a policy whereby the operating system can increase the time slice only for certain types of processes (which are CPU bound). But then, in order to be fair, it should allocate this larger time slice less frequently to it, i.e. it should reduce its priority. For an I/O bound processes, the actions should be just the reverse. Hence, if you want to be fair and also increase the throughput, for CPU bound processes, you need to increase the time slice and reduce the priority.

Therefore, CPU bound process should get more time slice, but less frequently.

Similarly, for I/O bound processes, you need to decrease the time slice and increase the priority.

Hence, an I/O bound process should get less time slice each time, but it should get it more frequently. The reason is that such a process cannot utilize a bigger time slice anyway. Notice the fairness in the policy. For both CPU bound and I/O bound processes, the total time allocated is more or less equitable, if not exactly the same.

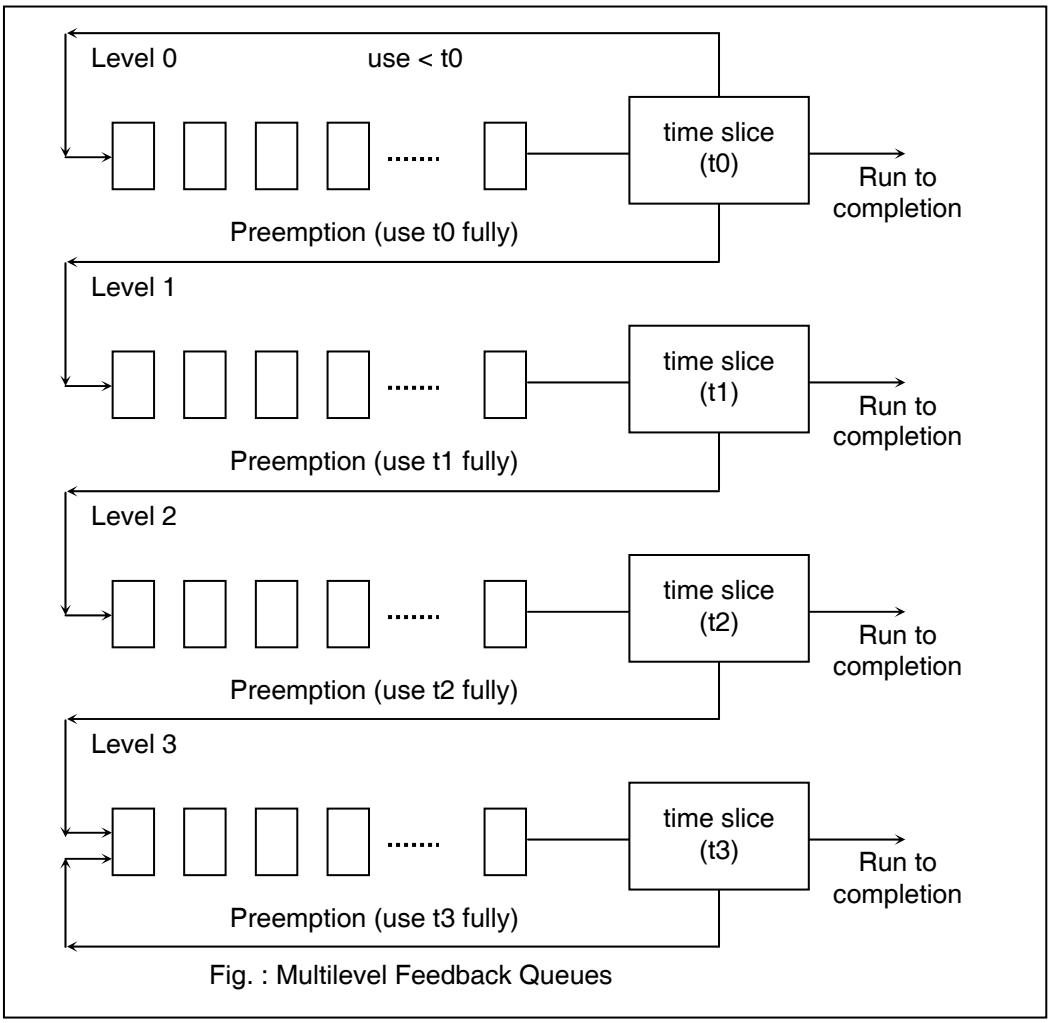


Fig. : Multilevel Feedback Queues

How should this policy be implemented? When the process is initiated, the operating system does not know whether it is I/O bound or CPU bound. What we need is a heuristic approach for the operating system which will monitor the performance of the process in terms of the frequency of I/O calls (I/O boundness) and then change the priority and the time slice of that process accordingly. This is normally implemented using Multilevel Feedback Queues (MIFQ) as shown in the fig.

The scheme works as follows :

(a) The list of ready processes is split up into many queues with levels 0 to n, (in the figure shown, we have assumed n = 3). At each level, the PCBs are chained together as before.

(b) Each level corresponds to a value of time slice. For instance level 0 has time slice = t0, level 1 has time slice = t1 and so on. These time slice values are stored by the operating system. When it wants to dispatch a process belonging to a specific queue, it loads the corresponding value of the time slice into the timer, so that there will be a 'time up' interrupt generated after that much time, as we have studied earlier.

(c) This is organized in such a way that as you go down the level, i.e. from level 0 to level 3, the time slice increases, i.e. t3 > t2 > t1 > t0. In practice if t0 = x milliseconds, t1 could be 2x, t2 could be 4x and t3 could be 8x.

(d) As you go down the level, the priority decreases. This is implemented by having the scheduler search through the PCBs at level 0 first, then level 1, then level 2 and so on for choosing a process for dispatching. Hence, if a PCB is found in level 0, the scheduler will schedule it without going to level 1 implying thereby that level 0 has higher priority than level 1. It will search for the queue at level 1 only if the level 0 queue is empty. The same philosophy applies to all the levels below. Hence, as we traverse from level 0 to level 3, the time slice increases and the priority decreases. After studying the past behaviour at the regular interval, now the kernel needs to somehow keep pushing the I/O bound processes at the upper levels and push the CPU bound processes to the lower levels. Let us see how this is achieved.

(e) A new process always enters at level 0 and it is allocated a time slice t0.

(f) If the process uses the time slice fully (i.e. if it is CPU bound), it is pushed to the lower level, thereby increasing the time slice but decreasing the priority. This is done for all levels, excepting if it is already at the lowest level in which case it is reintroduced at the end of the same (lowest) level only, because, obviously, it cannot be pushed any further.

(g) If the process requests for an I/O before the time slice is over (i.e. if it is I/O bound), the process gets blocked and when the I/O is complete, it is pushed up to the next higher level, excepting if it is already at level 0, it is reintroduced at the end of level 0 only.

Hence, instead of only one queue header for ready processes, the operating system will have to maintain four queue headers for four different queues for the processes in ready state. The CPU bound jobs will keep getting pushed down, whereas the I/O bound jobs will get pushed up.

12. Multilevel feedback queues are used for
    (a) To reduce priority of I/O bound over CPU
    (b) To reduce priority of CPU bound over I/O
    (c) To apply priority initially set
    (d) To create a heuristic / statistical approach
    (e) All of the above

Ans : (d)
In MIFQ policy, the OS increases the time slice for CPU bound processes for increasing the processor efficiency but accordingly reduces the priority or chance of being scheduled, therefore called heuristic approach.

13. Heuristic scheduling is more efficient than multilevel priority scheduling.
(a) only for I/O jobs
(b) only for CPU jobs
(c) None. Above statement is false.
(d) Both (a) & (b), above statement is true.

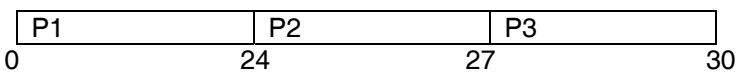
Ans : (d)
Since in multilevel priority scheduling the priority of the processes is fixed.

5. 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 time given in milliseconds

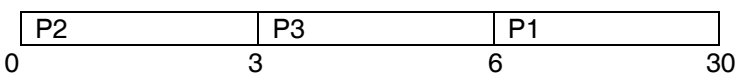
Process	Burst time
P1	24
P2	3
P3	3

- 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.

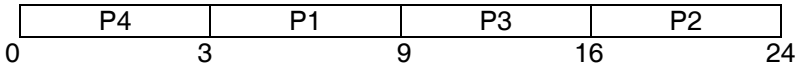
- The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, it releases the CPU, either by terminating or by requesting I/O.

6. 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 generally predicted as an exponential average of the measured length of previous CPU bursts.

Let $t_n \rightarrow$ length of the n^{th} burst

$\tau_{n+1} \rightarrow$ predicted value for the next CPU burst

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

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

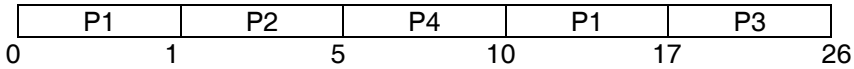
- The SJF algorithm may be either preemptive or non preemptive
A preemptive SJF algorithm will preempt the currently executing process, whereas a unpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called Shortest – remaining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

14. Consider the following four processes, with the length of the CPU – burst time given in milliseconds:

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Ans.

The resulting preemptive SJF schedule is as depicted in the following Gantt Chart:



Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds).

So process P1 is preempted and process P2 is scheduled.

The average waiting time = ((9 – 0) + (1 – 1) + (17 – 2) + (5 – 3)) / 4
= 26/4
= 6.5 milliseconds

(A non preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.)

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)

THE CONVOY EFFECT

Assume that we have one CPU bound process and many I/O bound processes. We illustrate the following sequence for the processes.

- The CPU bound process will get the CPU and hold it. During this time, all the other processes will finish their I/O and move into the ready queue, waiting for the CPU.

- While the processes wait in the queue, the I/O devices are idle. Eventually the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O bound processes, which have very short CPU bursts, execute quickly and move back to the I/O queue. At this time, the CPU sits idle.

- The CPU bound process will move back to the ready queue and CPU will be allocated to it. Again, the entire I/O processes end up waiting in the ready queue until the CPU bound process has finished.

- Such cycle result is the CONVOY effect wherein all the other processes end up waiting for one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

15. How would you reduce the Convoy effect? Name scheduling algorithm.
    (a) Shortest job first
    (b) FCFS
    (c) Priority based scheduling
    (d) Not possible

Ans : (a)

INTERPROCESS COMMUNICATION

Processes Frequently need to communicate with other processes.
For example : In a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well–structured way not using interrupts.

RACE CONDITION

In some operating systems, processes that are working together often have some common storage that each one can read and write. The shared storage may be in main memory or it may be a shared file, the location of the shared memory does not change the nature of the communication or the problems that arise.

Example to understand the IPC :

- When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

- By considering, a spooler directory with large number of slots, numbered 0, 1, 2, … each one capable of holding a file name and also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory.

- These two variables might well be kept on a two–word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing).

- More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in figure :

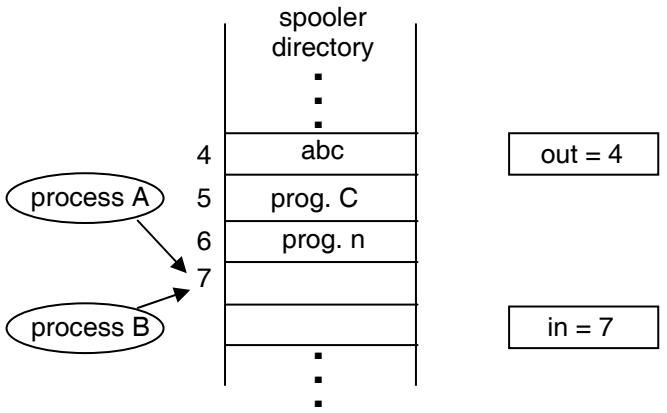


Fig. Two processes want to access shared memory at the same

- Consider a situation where, Process A reads in and stores the value 7, in a local variable called next–free–slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8, then it goes off and does other things.
  - → Eventually process A runs again, starting from the place it left off. It looks at next–free–slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.
  - → Then it computes next–free–slot + 1, which is 8, and set in to 8.
- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never get any output. This example illustrates that unsynchronized concurrent execution of cooperating processes may result in timing errors detrimental to the systems reliability.

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

16. Is Race condition problem a outcome of mutual exclusion among processes?
    (a) Yes
    (b) No

Ans : (a)

CRITICAL SECTIONS

How do we avoid race conditions ?

- The key to preventing trouble here and in many other situations involving shared memory, shared files and shared everything else, is to find some way to prohibit more than one process from reading and writing the shared data at the same time.

Mutual exclusion **:** Some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

- The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions.

- However, sometimes a process may be accessing shared memory or files, or doing other critical things that can lead to races.

The part of the program where the shared memory is accessed is called the **critical section**.

Note : If we could arrange matters such that no two processes were ever in their critical sections at the same time, we could avoid race conditions.

Four conditions to hold to have good solution

1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical section may block other processes.
4. No process should have to wait forever to enter its critical section.

17. Accessing database by two or more users in different modes, say user A in write mode and in user B in read mode, is an example of
    (a) Race condition (b) Mutual exclusion
    (c) IPC (d) All the above

Ans : (b)

In an abstract sense, the behavior that we want is shown in the fig. Here process A enters its critical region at time T_1. A little later, at time T_2 process B attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T_3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

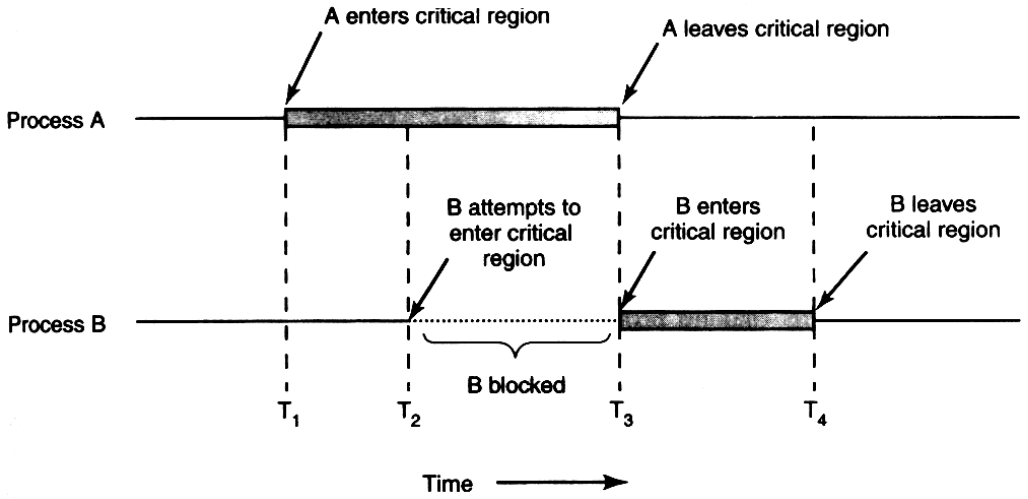


Fig. Mutual exclusion using critical regions

18. For a particular code to be shareable, it should be
(a) serially refusable (b) Re–entrant
(c) Reducible (d) None of these

Ans : (b)
A sharable code is Re–entrant, that the process accessing the shared data has to enter the critical section.

CONCURRENCY

Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes and allocation of processor time to processes.

Concurrency arises in 3 different contexts :

i) Multiple applications

Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.

ii) Structured applications

As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.

iii) Operating system structure

The same structuring advantages apply to the system programmer and operating systems are implemented as a set of processes or threads.

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 terrupts, 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.

Mutual Exclusion : Software Approaches

Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory. Simultaneous accesses to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time.

i) Dekkar's algorithm

Dekkar's algorithm can be explained by the following steps considering P0 and P1 be the two processes trying to enter into the critical section :

(a) When P0 wants to enter its critical section, it sets its flag to true.

(b) It then checks the flag of P1

- **i)** If flag is false, P0 may immediately enter its critical section.
- **ii)** If flag is true, P0 consults turn.

(c) If turn = 0 , then it is P0's turn and it periodically checks P1's flag.

(d) P1 defer and set its flag false; thus allowing P0 to proceed.

(e) After using critical section P0 sets its flag to false to free the critical section and sets turn to 1 to transfer the right to insist to P1.

```
boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        while (flag [1])
                if (turn == 1)
                {
                    flag [0] = false;
                        while (turn == 1)
                                /* do nothing */;
                    flag [0] = true;
                }
        /* critical section  */;
        turn = 1;
        flag [0] = false;
        /* remainder   */;
    }
}

void P1( )
{
    while (true)
    {
        flag [1] = true;
        while (flag [0])
           if (turn == 0)
           {
                flag [1] = false;
                while (turn == 0)
                    /* do nothing */;
                flag [1] = true;
           }
        /* critical section   */;
        turn = 0;
        flag [1] = false;
        /* remainder   */;
    }
}
void main 0
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Dekkar's Algorithm

19. What is the disadvantage of Dekkar's algorithm?
    (a) It can handle only two processes at a time
    (b) It assumes relative speeds of the process
    (c) Deadlock in possible if two processes are running on two different PC's.
    (d) All the above

Ans : (d)

ii) Peterson's Algorithm

Flag. : It is a global array variable which indicates the position of each process with respect to mutual exclusion.

Turn : It is a global variable that resolves simultaneity conflicts.

Consider Process P0. Once it has set flag [0] to true, P1 cannot enter its critical section. If P1 already is in its critical section, then flag [1] = true and P0 is blocked from entering its critical section.

Suppose that P0 is blocked in its while loop. This means that Flag [1] is true and turn = 1.

P0 can enter its critical section when either flag [1] becomes false or turn becomes 0.

```
boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1)
                /* do nothing */;
        /* critical section  */;
        flag [0] = false;
        /* remainder  */;
    }
}
void P1()
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0)
                /* do nothing */;
        /* critical section  */;
        flag [1] = false;
        /* remainder  */;
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Peterson's Algorithm for Two Processes

20. What do you understand by busy waiting?
    (a) A process periodically checking on a variable
    (b) A process continuously checking on a variable
    (c) A process polling on a variable
    (d) A process issuing a interrupt

Ans : (c)
Busy waiting concept means that the process continuously checks for the required value in a variable or an event, thereby wasting CPU cycles; as no useful work is being done.

- The Peterson's solution has a defect of requiring busy waiting. In essence, what these solutions do is this : When a process wants to enter its critical section, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.
- **Drawback of this approach :** Wastage of CPU time
- Consider a computer with two processes, H, with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state.

At the certain moment, with L in its critical region. H becomes ready to run (e.g. an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever.
This situation is sometimes referred to as the priority inversion problem.

SLEEP AND WAKEUP

Some inter-process communication primitives :

→ These primitives block, instead of wasting CPU time when they are not allowed to enter their critical sections.

→ SLEEP : is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
WAKEUP : call has one parameter, the process to be awakened.

Note :
Both SLEEP and WAKEUP each have one parameter, a memory address used to match up SLEEPs with WAKEUPs.

EVENT COUNTERS

- Event counters is a method which gives the solution for the producer consumer problem without requiring mutual exclusion.
- It uses a special kind of variable called an event counter.

Three operations are defined on an Event counter E :

1) Read(E) $\Rightarrow$ Return the current value of E.
2) Advance (E) $\Rightarrow$ Atomically increment E by 1.
3) Await (E, v) $\Rightarrow$ Wait until E has a value of v or more.

Note : The producer–consumer problem does not use Read, but it is needed for other synchronization problems.
- Event counters only increases, never decreases. They always start at 0.

21. Can event counters be binary?

Ans. No. Because they have to be advanced / incremented to indicate the number of events waiting for a critical section.

22. An event counter
    (a) is a integer counter that does not decrease
    (b) keeps track of the number of occurrences of events of a particular class of related events.
    (c) orders events
    (d) Both (a) & (b)
    (e) Both (a) & (c)

Ans : (d)

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

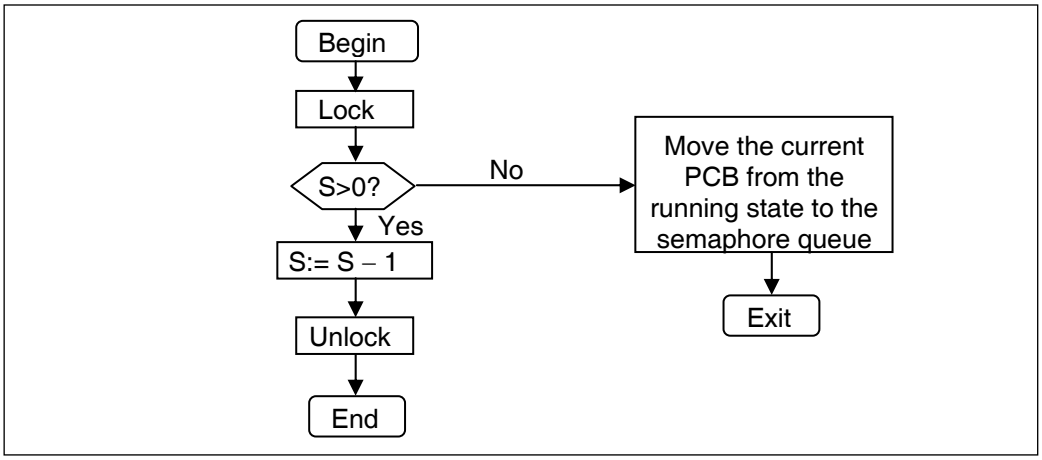
"DOWN and UP" form the mutual exclusion primitives for any process. Thus, if a process has a critical region, it has to be encapsulated between these DOWN and UP instructions, which is shown below :

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

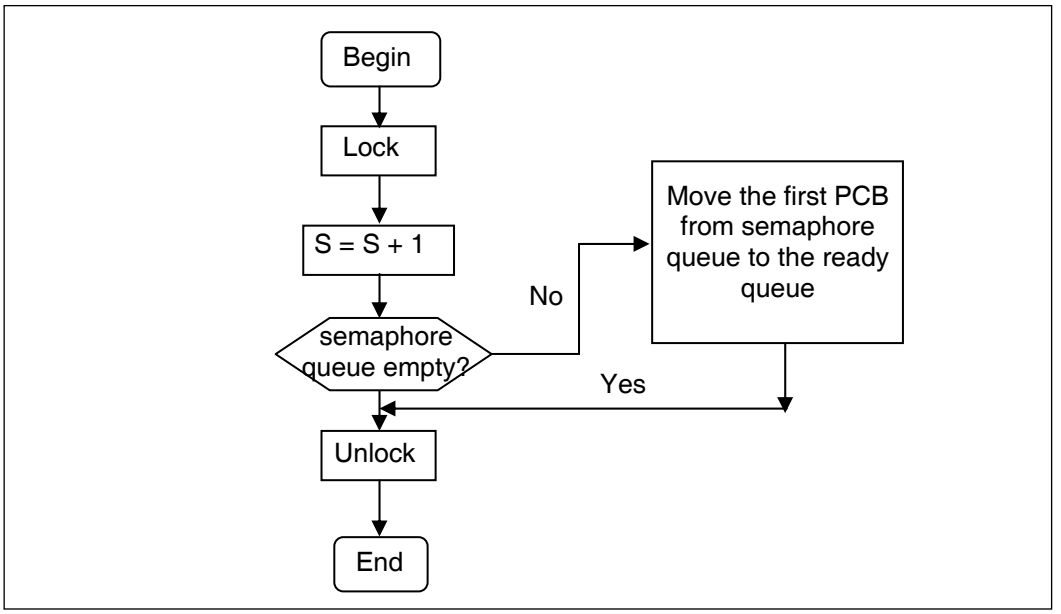
Fig. General Structure of a process with semaphores

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 : = – 1
D.3                       else Wait on S
D.4                 Endif
D.5                 AZEnable 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.

iii) Any new process cannot get into its critical region when S = 0. If it tries to execute DOWN(S) routine, it will be added to the semaphore queue and cannot proceed into its critical region because S = 0. The process is pushed from running state into a semaphore queue in instruction 1 only. Thus cannot enter into its critical region to execute instruction 2.

iv) S can become 1 again only in the UP(S) routine as shown in the flowchart. UP(S) is executed in instruction 3, only after a process has come out of its critical region. Thus when S = 1, no other process is in the critical region at that time and thus new process could be allowed to enter into it.

Applications of Semaphore

i) Semaphores have wide uses and applications whenever shared variables are used.

ii) OS can use them to implement the scheme of blocking / waking up of processes, when they wait for any event such as the completion of the I/O.

iii) Semaphores can also be used to synchronize block / wakeup processes.

23. Pick out a wrong statement
    (a) Semaphores are used to deal with n process critical section problem
    (b) Semaphores are used to solve synchronization problems
    (c) Semaphores are used to solve asynchronization problems
    (d) All are correct statements.

Ans : (c)
If the processes are synchronized then why do they need a synchronizing variable like a semaphore.

MONITORS

The monitor is a programming language construct that provides equivalent functionality as that of semaphores and that is easier to control. A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package. Process may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Chief characteristics of monitor :

i) The local data variables are accessible only by the monitor's procedures and not by any external procedure.

ii) A process enters the monitor by invoking one of its procedures.

iii) Only one process may be executing in the monitor at a time, any other process that has invoked the monitor is suspended, waiting for the monitor to become available.

The data variables in the monitor can be accessed by only one process at a time. Thus, a shared data structure can be protected by placing it in a monitor. If the data in a monitor represent some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

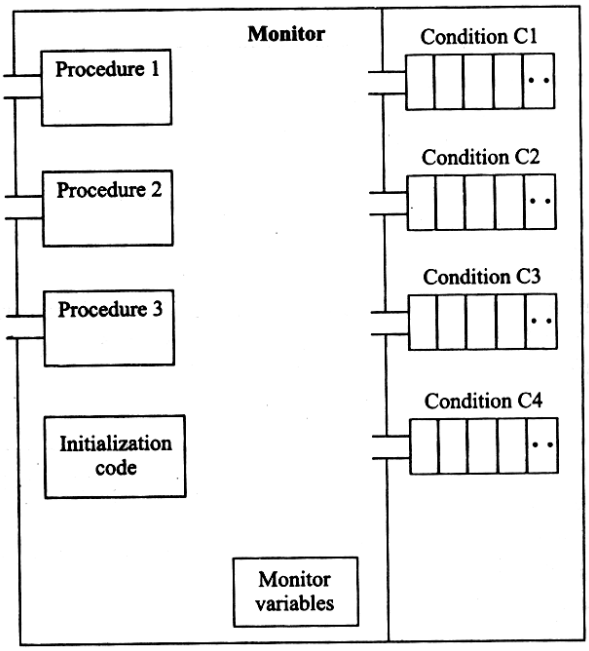
To be useful for concurrent processing, the monitor must include synchronization tools. A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two functions operate on condition variables.

i) cwait(c)

Suspend execution of the calling process on condition c. The monitor is now available for use by another process.

ii) csignal(c)

Resume execution of some process suspended after a cwait on the same condition. If there are several such processes, choose one of them. If there is no such process, do nothing.



Structure of monitor

Monitor wait / signal operations are different from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

A process can enter the monitor by invoking any of its procedures. One can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time. Other processes that attempt to enter the monitor, join a queue of processes suspended waiting for monitor availability. Once a process is in the monitor, it may temporarily suspend itself on condition x by issuing cwait(x), it is then placed in a queue of processes waiting to re–enter the monitor when the condition changes. If a process that is executing in the monitor detects a change in condition variable x, it issues csignal(x), which alerts the corresponding condition queue that the condition has changed.

The major disadvantage of the monitor concept is its lack of implementation in most commonly used programming languages. While semaphores are not included in most languages either, P and V operations are easily added as independent subroutines or as operating system supervisor calls. Monitors cannot easily be added if they are not natively supported by the language.

24. Which of the following is a software solution which requires configuration details?
    (a) Event counters
    (b) Semaphores
    (c) Monitor
    (d) Test and set instruction

Ans : (c)
Since monitor is defined as a collection of procedures, data parameters which provide functionality of a semaphore.

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.

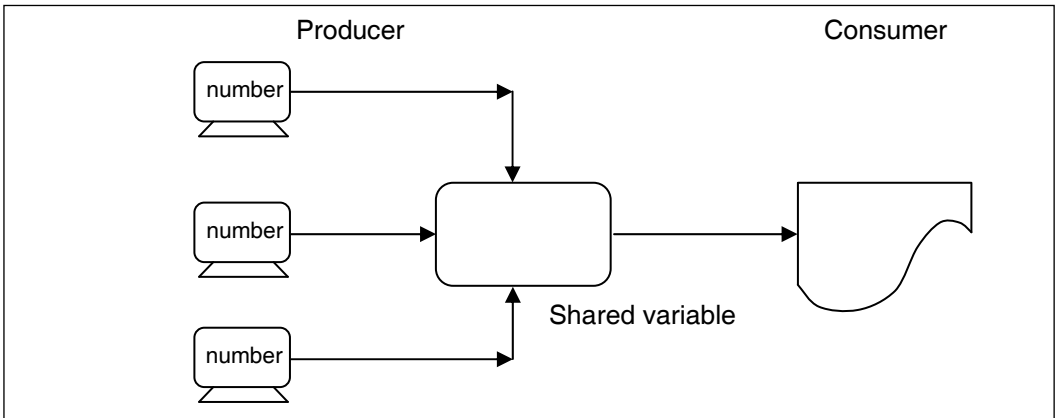


Fig. : Producer consumer processes

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.

25. Is (Monitor) frame buffer a consumer process while keyboard buffer a producer process?

Ans. No. As the application and system software's consume data from the keyboard buffer and with respect to the monitor the application acts as a producer.

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.

26. Can event counter be used to solve multiple producer – simple consumer problem?
    (a) Yes, by using 1 event counter per process.
    (b) Yes, by using only 2 event counters only.
    (c) No, event counters can solve the problem for a pair of producer – consumer process.
    (d) Event counters are not used at all.

Ans : (a)

Solution of Producer Consumer Problem by Monitors

- Only one process can be active in a monitor at any instant.

- Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.

- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor.

- If some process is active within the monitor, and another process calls it then the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.
  The monitors provide an easy way to achieve mutual exclusion but not a way for process to block when they cannot proceed. The solution lies in the introduction of condition variables, along with two operations on them, WAIT and SIGNAL.

- When a monitor procedure discovers that it cannot continue (e.g. the producer finds the buffer full), it does a WAIT on some condition variable say, *full*. Thus causes the calling process to block and allow another process that has been previously prohibited from entering the monitor to enter now.

- The entered process, the consumer, can wake up its sleeping partner by doing a SIGNAL on the condition variable that its partner is waiting on.

- To avoid having two active processes in the monitor at the same time, we require that a process doing a SIGNAL must exit the monitor immediately.

- If a SIGNAL is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

- Condition variables are not counters. Moreover, if a condition variable is signaled with no one waiting on it, the signal is lost. The WAIT must come before the signal.

27. An example of interprocess communication is
    (a) when a process issues a I/O request, it blocks itself to a wait completion of I/O. Some other process awakens the blocked process.
    (b) The producer – consumer relationship.
    (c) Both (a) & (b)
    (d) None of these

Ans : (b)

MESSAGE PASSING

When processes interact with one another, two fundamental requirements must be satisfied. They are :

i) Synchronization.
ii) Communication

Processes need to be synchronized to enforce mutual exclusion, cooperating processes may need to exchange information. One approach to provide both of these functions is Message Passing.

Message Passing is provided in the form of a pair of primitives :

send (destination, message)

receive (source, message)

A process sends information in the form of a message to another process is designated by a destination. A process receives information by executing the receive primitive, indicating the sources of the sending process and the message.

Design Characteristics of Message Systems for Interprocessor Communication and Synchronization

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

i) Synchronization :

The receiver cannot receive a message until it has been sent by another process. Thus communication of a message between two processes should have some level of synchronization.

When a process issues a send primitive, there are two possibilities :

(a) Sending process is blocked until the message is received.

(b) Sending process is not blocked.

When a process issues a receive primitive, there are two possibilities :

(a) If a message has previously been sent, the message is received and execution continues.

(b) If there is no waiting message then either

(1) The process is blocked until a message arrives, or

(2) The process continues to execute, abandoning the attempt to receive.

Three common combinations of any system are :

(a) Blocking send, blocking receive
Both sender and receiver are blocked until the message is delivered. This combination allows for tight synchronization between processes.

(b) Nonblocking send, blocking receive

The sender may continue on, the receiver is blocked until the requested message arrives.

(c) Nonblocking send, nonblocking receive

Neither party is required to wait.

28. In message passing system, how many points of synchronization are possible?
(a) 1 (b) 2 (c) 4 (d) 6

Ans : (c)
The 4 points are
(a) At the sender process
(b) At the communication protocol on the sender end
(c) At the communication protocol on the receiver end
(d) At the receiver process

ii) Addressing :

The various schemes for specifying processes in send and receive primitives fall into two categories. :

(a) Direct addressing (b) Indirect addressing

Direct addressing :

With direct addressing, the send primitive includes a specific identifier of the destination process. The receive primitive can be handled in one of two ways. One possibility is to require that the process explicitly designate a sending process. Thus, the process must know ahead of time from which process a message is expected. In other cases, it is impossible to specify the anticipated source process.

Indirect addressing :

Here messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as mailboxes. Thus, for two processes to communicate, one process sends a message to the appropriate mailbox and the other process picks up the message from the mailbox.

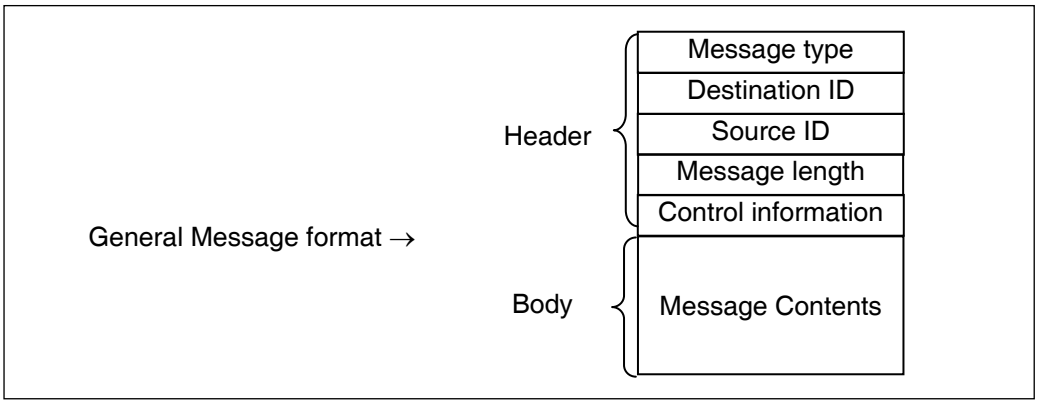
iii) Message Format :

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system.

→ For some OS, designers have preferred short, fixed length messages to minimize processing and storage overhead.

→ If a large amount of data is to be passed, the data can be placed in a file and the message then simply references that file

→ Variable length messages



The message is divided into two parts :

(a) Header
It contains information about the message. It may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages. It may also contain additional control information.

(b) Body
It contains the actual contents of the message.

iv) Queuing Discipline :

The simplest queuing discipline is FIFO, but this may not be sufficient if some messages are more urgent than others. One can also specify message priority, on the basis of message type or by designation by the sender. Another alternative is to allow the receiver to inspect the message queue and select which message to receive next.

v) Mutual Exclusion :

A set of concurrent processes share a mailbox, mutex, which can be used by all processes to send and receive. The mailbox is initialized to contain a single message with null content. A process wishing to enter its critical section first attempts to receive a message. If the mailbox is empty, then the process is blocked. Once a process has acquired the message, it performs its critical section and then places the message back into the mailbox. Thus the message functions as a token that is passed from process to process.

29. Why mailboxes are used in message passing system?
[Hint : choose most appropriate option]
(a) All processes can communicate independently
(b) One sender can talk to move than one receiver
(c) To store fixed length messages
(d) To interface both sender and receiver

Ans : (a)
Mailbox is a buffer space reserved for a particular registered user at the Mail Server. It enables the sender and receiver processes to communicate independently.

30. Is message header context information is optional part of a control information?
(a) Yes
(b) No

Ans : (a)

READERS / WRITERS PROBLEM

Problem Definition

There is a data area shared among a number of processes. The data area could be a file, a block of main memory or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows :

i) Any number of readers may simultaneously read the file.

ii) Only one writer is writing to the file, no reader may read it.

iii) If a writer is writing to the file, no reader may read it.

Here readers do not write to the data area, nor do writers read the data area. Instead allow any of the processes to read or write the data area. Thus any portion of a process that accesses the data area is declared to be a critical section and impose mutual exclusion solution.

Solutions :

i) Readers have Priority

This solution guarantees that if there is a new reader ready to read data then no writers are allowed. It uses semaphore to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process makes use of semaphore to enforce mutual exclusion. To allow multiple readers, when no reader reading, the first reader that attempts to read should wait on semaphore. When one reader is reading, subsequent readers need not wait before entering. A global variable keep track of the number of readers.

ii) Writers have priority

This solution guarantees that no new readers are allowed to access data if at least one writer has declared a desire to write. For writers, the following semaphores and variables are added :

(a) A semaphore that inhibits all readers while there is at least one writer desiring access to the data area.

(b) A variable that controls the setting of above semaphore.

(c) A semaphore that controls the updating of above defined variable.

iii) Message passing

Here, there is a controller process that has access to the shared data area. Other processes wishing to access the data area send a request message to the controller, are granted access with an "OK" reply message, and indicate completion of access with a "finished" message. The controller is equipped with three mailboxes, one for each type of message that it may receive.

The controller process services write request messages before read request message to give writers priority. Mutual exclusion is enforced by a variable which is initialized to some number greater than the maximum possible number of readers. The action of the controller is

i) If variable > 0, then no writer is waiting and there may or may not be readers active. Service all "finished" messages first to clear active readers. Then service write requests and then read requests.

ii) If variable = 0, then the only request outstanding is a write request. Allow the writer to proceed and wait for a "finished" message.

iii) If variable < 0, then a writer has made a request and is being made to wait to clear all active readers. Only "finished" messages should be serviced.

31. If the reader issues wait( ) on a semaphore for entering the reading state on a database, then what approach is being used?
    (a) Readers have priority
    (b) Writers have priority

Ans : (b)
As readers issued wait ( ) signal that means they are waiting for all write operation queued to finish first.

STARVATION

When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access, which is called starvation of that process for the resource.

For Example : Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the operating system grants access to P3 and that P1 again requires access before completing its critical section. If the OS grants access to P1 after P3 as finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation. Thus P2 is made starved for the resource R.

DEADLOCK

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. Or Deadlock involves conflicting needs for resources by two or more processes. Whether or not Deadlock occurs depends on both the dynamics of the execution and on the details of the application.

32. What is true about starvation and deadlock?
    (a) Starvation only leads to deadlock
    (b) Starvation is an extreme case of deadlock
    (c) Deadlock is an extreme case of starvation
    (d) Deadlock and starvation are two unrelated concepts

Ans : (c)
Starvation leads to allocation of resources to processes after an unacceptable delay; while in deadlock the processes never get the resources.

For Example :

In a multiprogramming system, suppose two processes are there and each wants to print a very large tape file. Process A requests permission to use the printer and is granted. Process B then requests permission to use the tape drive and is also granted. Now A asks for the tape drive, but the request is denied until B releases it. Instead of releasing the tape drive, B asks for the printer. At this point both processes are blocked and will remain so forever. This situation is called a deadlock.

Conditions of Deadlock

- **Mutual exclusion :–** At least one resource is held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released. Each resource is either currently assigned to exactly one process or is available.
- **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by another processes. Process currently holding resources granted earlier can request new resources.
- **No Preemption :** Resources cannot be preempted; that is a resource can only be released voluntarily by the process holding it, after the process has completed its task. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- **Circular Wait :** There exist a set (P0, P1 .... Pn) of waiting processes such that P0 is waiting for a resource which is held by P1, P1 is waiting for a resource which is held by P2. Pn-1 is waiting for a resource which is held by Pn, and Pn, is waiting for a resource which is held by P0. Thus there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

33. Circular wait is a special case of Hold and Wait where process P_i is waiting for a resource held by P_{i+1}.
    (a) Yes
    (b) No
    (c) Under certain circumstances

Ans : (a)

Deadlock Prevention

1. Mutual Exclusion

- The mutual exclusion condition must hold for non-sharable types of resources. **For example:** Several processes cannot simultaneously share a printer.
- Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.
- It is not possible to prevent deadlocks by denying the mutual-exclusion condition.

2. Hold and Wait

- In order to ensure that the hold-and-wait condition never holds in the system, one must guarantee that whenever a process request a resource it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all of its resources before it begins execution.

  For Example : Consider a process which copies from a card reader to a disk file, sorts the disk file, and then prints the results to a line printer and copies them to a magnetic tape. If all resources to be requested at the beginning of the process, then the process must initially request the card reader, disk file, line printer, and tape drive. It will hold the magnetic tape drive for its entire execution, even though it needs only at the end.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated. To illustrate the difference between these two protocols.

  For Example : Process request initially only the card reader and disk file. It copies from the card reader to the disk, and then releases both the card reader and the disk file. The process must then re-request the disk file and the line printer. After copying the disk file to the line printer, it releases both, and then requests the disk, file and tape drive. It copies the disk file to tape, then releases these two resources and terminates.

There are two main disadvantages to these protocols :

i) First, resource utilization may be very low, since many of the resources may be allocated but unused for a long period of time. In the example given, for instance, we can release the card reader and disk file and then re-request the disk file and printer only if we can be sure that our data will remain on the disk file. If this cannot be ensured, then we must, request all resources at the beginning for both protocols.

ii) Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely while at least one of the resources that it needs is always allocated to some other process.

3. No Preemption

- The third necessary condition is that there is no preemption of resources that have already been allocated. Guaranteeing a situation so that the "no preemption" condition is not met is very difficult.
- The preempted resources are added to the list of resources for which the process is waiting. The process will only be restarted when it can regain its old resources, as well as the new ones that it is requesting.

  For Example : If a process request some resources, we first check if they are available. If so, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can only be restarted when it is allocated the new resources it is requesting and recovers any resources that we preempted while it was waiting.

34. With respect to shareable resources like a database in read mode or a storage medium, can enter deadlock condition between two or more processes only if any of these conditions are satisfied ?

(1)	Mutual exclusion	(2)	Hold and wait
(3)	No preemption	(4)	Circular wait

Options :

(a)	1, 2, 3	(b)	2, 3, 4
(c)	1, 3, 4	(d)	1, 2, 4

Ans : (b)
These are necessary conditions for a deadlock to occur.

4. Circular Wait

- If a circular wait condition is prevented, the problem of the deadlock can be prevented too.
- One way in which this can be achieved is to force a process to hold only one resource at a time. If it requires another resource, it must first give up the one that is held by it and then request for another.

- If a process P1 holds R1 and wants R2, it must give up R1 first, because another process P2 should be able to get it (R1). We are faced with a problem of assigning a tape drive to P2 after P1 has processed only half the records therefore, is also an unacceptable solution
- There is a better solution to the problem, in which all resources are numbered as shown in Fig.

Number	Resource Name
0	Tape drive
1	Printer
2	Plotter
3	Card Reader
4	Card Punch

- Any process has to request for all the required resources in a numerically ascending order during its execution. This would prevent a deadlock.
- Let us assume that two processes P1 and P2 are holding a tape drive and a plotter respectively. A deadlock can take place only if P1 holds the tape drive and wants the plotter, whereas P2 holds the plotter and requests for the tape drive, i.e. if the order in which the resources are requested by the two processes is exactly opposite. And this contradicts our assumption. Because 0 < 2, a tape drive has to be requested for before a plotter, by each process, whether it is P1 or P2. Therefore, it is impossible to get a situation that will lead to the deadlock.

Minor Problem

There are two tape drives T1 and T2 and two processes PI and P2 in the system. If PI holds T1 and requests for T2 whereas P2 holds T2 and requests for T1, the deadlock can occur.

Major Problem

It is almost impossible to force all the processes to request resources in a globally predetermined order, because the processes may not actually require them in that order. The waiting periods and the consequent wastage could be enormous. And, this surely is the major problem.

35. Why is deadlock prevention not practically feasible to implement?
    (a) Applicable to only shareable resources
    (b) It is only a theoretical concept
    (c) Resource utilization is very low
    (d) It is only applicable to Unix OS

Ans : (c)
As all the processes have to prestate all the resources that may be required in the future. This will always lead to overstating thereby reduce the degree of multiprogramming.

Deadlock Avoidance

- Deadlock prevention was concerned with imposing certain restrictions on the environment or processes, so that deadlocks can never occur. The operating system aims at avoiding a deadlock rather than preventing it.
- Deadlock avoidance is concerned with starting with an environment, where a deadlock is theoretically possible (it is not prevented), but by some algorithm in the operating system, it is ensured, before allocating any resource that after allocating it, a deadlock can be avoided. If that cannot be guaranteed, the operating system does not grant the request of the process for a resource in the first place.
- Deadlock prevention algorithms, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and hence, that deadlock cannot hold. A side effect of preventing deadlocks by this method, however, is possibly low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to ensure additional information about how resources are to be requested.
- Dijkstra was the first person to propose an algorithm in 1965 for deadlock avoidance. This is known as 'Banker's algorithm.
- This algorithm in the operating system is such that it can know in advance before a resource is allocated to a process, whether it can lead to a deadlock ("unsafe state") or it can certainly manage to avoid it ("safe state").
- Banker's algorithm maintains two 2–dimensional matrices on dynamic basis with resource type as rows and processes as columns. Matrix A consists of the resources allocated to different processes at a given time. Matrix B maintains the resources still needed by different processes at the same time. These resources could be needed one after the other or simultaneously.

  Examples of both these matrices are shown in Fig.

Process	Tape Drives	Printers	Plotters
p0	2	0	0
P1	0	1	0
P2	1	2	1
P3	1	0	1

Matrix A (or matrix allocated)
Resources assigned

Process	Tape Drives	Printers	Plotters
P0	1	0	0
P1	1	1	0
P2	2	1	1
P3	1	1	1

Matrix B (or claims matrix)
Resources still required

Vectors
Total Resources (T) = 543
Held Resources (H) = 432
Free Resources (F) = 111

Banker's Algorithm

- Matrix A shows that process P0 is holding 2 tape drives at a given time. At the same moment, process P1 is holding 1 printer and so on. If we add these figures vertically, we get a vector of Held Resources (H) = 432. This is shown as the second row in the rows for vectors.
- Vector for the Total Resources (T) is 543. This means that in the whole system, there are physically 5 tape drives, 4 printers and 3 plotters. These resources are made known to the operating system at the time of system generation.
- By subtraction of (H) from (T) column wise, we get a vector (F) of free resources which is 111. This means that the resources available to the operating system for further allocation :1 tape drive, 1 printer and 1 plotter at that juncture.
- Matrix B gives processwise additional resources that are expected to be required in due course during the execution of these processes. For instance, process P2 will require 2 tape drives, 1 printer and 1 plotter, in addition to the resources already held by it.
- Process P2 requires in all 1 + 2 = 3 tape drives, 2 + 1 = 3 printers and 1 + 1= 2 plotters. If the vector of all the resources required by all the processes (vector addition of Matrix A and Matrix B) is less then the vector (T) for each of the resources, there will be no contention and therefore, no deadlock. If that is not so, a deadlock has to be avoided.

The algorithm for the deadlock avoidance works as follows:

i) Each process declares the total required resources to the operating system at the beginning. The operating system puts these figures in Matrix B (resources required for completion) against each process. For a newly created process, the row in Matrix A is fully zeros to begin with, because no resources are yet assigned for that process. For instance, at the beginning of process P2, the figures for the row for P2 in Matrix A will be all 0 s; and those in Matrix B will be 3, 3 and 2 respectively.

ii) When a process requests the operating system for a resource, the operating system finds out whether the resource is free and whether it can be allocated by using the vector (F). If it can be allocated, the operating system does so, and updates Matrix A by adding 1 (allocated figure) to the appropriate slot. It simultaneously subtracts 1 (the same number) from the corresponding slot of Matrix B. For instance, starting from the beginning, if the operating system allocates a tape drive to P2, the row for P2 in Matrix A will become 1, 0 and 0. The row for P2 in Matrix B will correspondingly become 2, 3 and 2. At any time, the total vector of these two rows, i.e. addition of the corresponding numbers in the two rows, is always constant and is equivalent to the total resources needed by P2, which in this case will be 3, 3 and 2.

iii) However, before making the actual allocation, whenever, a process makes a request to the operating system for any resource, the operating system goes through the Banker's algorithm to ensure that after the imaginary allocation, there need not be a deadlock, i.e. after the allocation, the system will still be in a 'safe state'. The operating system actually allocates the resource only after ensuring this. If it finds that there can be a deadlock after the imaginary allocation at some point in time, it postpones the decision to allocate that resource. It calls this state of the system that would result after the possible allocation as 'unsafe'. Remember: the unsafe state is not actually a deadlock. It is a situation of a potential deadlock.

A safe state is not a deadlock state, and a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs: the behavior of the processes controls unsafe states.

36. Which piece of system software is used to implement the deadlock avoidance or Banker's algorithm?
    (a) Loader
    (b) Compiler
    (c) Medium term scheduler
    (d) Kernel

Ans : (c)

37. Each resource request demands the system to consider
    (a) The resource currently available
    (b) The resources currently allocated to each process
    (c) The future requests and release of each process
    (d) All of the above

Ans : (a)

Deadlock Detection:

With deadlock detection, requested resources are granted to processes whenever possible.

OS performs an algorithm that allows it to detect the circular wait condition.

Deadlock Detection Algorithm:

It makes use of allocation matrix, available vector and a request matrix Q such that q_{ij} represents the amount of resources of type j requested by process i. The algorithm proceeds by marking processes that are not deadlocked. Initially all processes are unmarked. Then the following steps are performed :

i) Mark each process that has a row in the allocation matrix of all zeros.

ii) Initialize a temporary vector W to equal the available vector.

iii) Find an index i such that process i is currently unmarked and the i^{th} row of Q is less than or equal to W. If no such row is found, terminate the algorithm.

iv) If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. Return to step 3.

A deadlock exists if and only if there are unmarked processes at the end of the algorithm.

Each unmarked process is deadlocked. It does not guarantee to prevent deadlock as that will depend on the order in which requests are granted.

Recovery

Once deadlock has been detected, some strategy is needed for recovery. Possible approaches are:

i) Abort all deadlocked processes.

ii) Back up each deadlocked process to some previously defined checkpoint and restart all processes. This requires that rollback and restart mechanisms are built into the system.

iii) Successively abort deadlocked processes until deadlock no longer exists. After each abortion, the detection algorithm must be reinvoked to see whether deadlock still exists.

iv) Successively preempt resources until deadlock no longer exists. After preemption, the detection algorithm must be reinvoked to see whether deadlock still exists. A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

An Integrated Deadlock Strategy

Principle	Resource Allocation policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative under commits resources	Requesting all resources at once	* Works well for processes that perform a single burst of activity * No Preemption necessary	* Inefficient * Delays process initiation * Future resource requirement must be known
		Preemption	* Convenient when applied to resources whose state can be saved and restored easily	* Preempts more often than necessary * Subject to cyclic start
		Resource ordering	* Feasible to enforce via compile time checks * Needs no run–time computation since problem is solved in system design	* Preempts without much use * Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	* No preemption necessary	* Future resource requirements must be known * Processes can be blocked for long periods.
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	* Never delays process Initiation * Facilitates online handling	* Inherent preemption losses

38. Preemption of resources is commonly noticed in
(a) Deadlock prevention
(b) Avoidance
(c) Detection
(d) Only with respect to shareable resources irrespective of the strategy

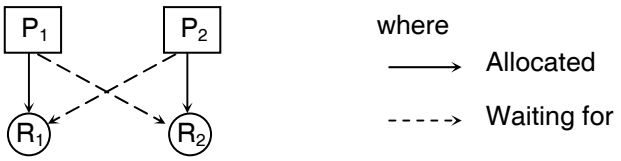
Ans : (c)
∵ If there is a cycle in a Resource Allocation Graph (RAG) formed the process would have to preempt the resources to break the deadlock condition to occur.

39. The characteristic of a deadlock are
(a) Processes never finish executing
(b) System resources are tied up
(c) All of the above
(d) None of these

Ans : (c)

40. In a resource allocation graph, the graph contains a cycle. This implies that
(a) The system is not in a deadlock state
(b) There is a impending deadlocked state
(c) The system may possibly be in a deadlock state
(d) None of these

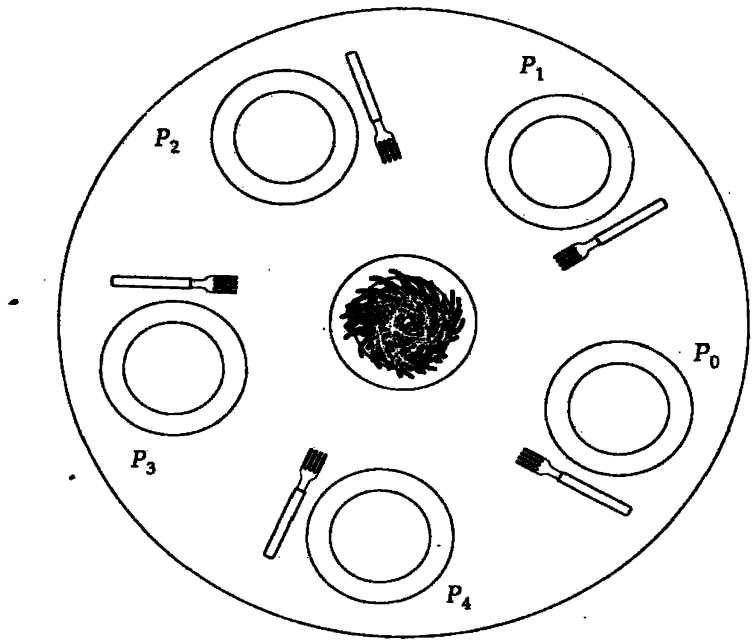
Ans : (b)
A cycle in a resource allocation graph looks like this



DINING PHILOSOPHERS PROBLEM

Problem :

There are five philosophers whose eating arrangements is shown in figure. A round table on which was set a large serving bowl of Spaghetti, five plates, one for each philosopher and five forks. A philosopher wishing to eat would go to his or her assigned place at the table and using the two forks on either side of the plate, take and eat some spaghetti. One should make sure that no two philosophers can use the same fork at the same time and thus avoiding deadlock and starvation.



Dining Arrangement for Philosophers

Solution :

First Solution to the Dining Philosophers Problem

```
/* program  diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
   while (true)
   {
      think();
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal(fork [(i+1) mod 5]);
      signal(fork[i]);
   }
}
void main()
{
   parbegin (philosopher (0), philosopher (1), philosopher (2),
      philosopher (3), philosopher (4));
}
```

A First Solution to the Dining Philosophers Problem

The solution uses semaphore. Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table.

Drawback :

i) Deadlock
If all of the philosophers are hungry at the same, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there, and thus lead to deadlock.

ii) Starvation
Due to this undignified position, all philosophers starve.

Second Solution to the Dining Philosophers Problem

```
program  diningphilosophers;
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i);
{
   while (true)
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
}

void main()
{
   parbegin (philosopher (0), philosopher (1), philosopher (2),
      philosopher (3), philosopher (4));
}
```

Second Solution to the Dining Philosophers Problem

To overcome the risk of deadlock, either we have five additional forks or teach the philosophers to eat spaghetti with just one fork. Consider adding an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers at least one philosopher will have access to two forks. Thus this solution is free of deadlock and starvation.

41. Pick out the false statement :
The side effect of using deadlock presentation algorithm is
(a) Possibly low utilization
(b) Reduced system throughput
(c) Increased waiting time
(d) All are true

Ans : (c)

42. Which method of eliminating a deadlock increases over–head ?
    (a) Kill and deadlocked processes and discard results of partial computations, which will be re–computed later.
    (b) Kill one process at a time until deadlock cycle is eliminated. A deadlock detection algorithm is invoked to determine which process is deadlocked.
    (c) Total rollback at process that creates deadlock
    (d) None of these

Ans : (b)
In deadlock detection overhead is high.
∴ RAG &WFG have to be created and checked for cycle.

43. If m = number of resources,
n = number of processes in the system then, Banker's algorithm, although a general algorithm, may require _____ operations.
    (a) m * n
    (b) (m * n) * (n * n)
    (c) (m * m) * n
    (d) m * (n * n)

Ans : (d)
Since there are two processes, the first process checks the requirement of resource and the second process checks that if the resource is allocated there should not be a unsafe state. m is multiplied since it is done for every resource.

44. Competition among processes for resources creates problems. They are
    (a) Mutual exclusion
    (b) Deadlock
    (c) Starvation
    (d) All of these

Ans : (d)

LMR (LAST MINUTE REVISION)

- A process is a program in execution. As a process executes, it changes state. Each process may be in one of the following states: new, ready, running, waiting or terminated. Each process is represented in the OS by its own PCB.
- A ready process is one that is not currently executing but that is ready to be executed as soon as the OS dispatches it.
- The running process is that process that is currently being executed by the processor.
- A blocked process is waiting for the completion of some event, such as an I/O operation.
- A running process is interrupted either by an interrupt, which is an event that occurs outside the process and that is recognized by the processor, or by executing a supervisor call to the OS.
- The principal function of the OS is to create, manage and terminate processes. While processes are active, the OS must see that each is allocated time for execution by the processor, coordinate their activities, manage conflicting demands and allocate system resources to processes.

- To perform its process management functions, the OS maintains a description of each process or process image, which include the address space within which the process executes and a process control block.

- A process, when it is not executing, is placed in some waiting queue. The two major classes or queues in an OS are I/O request queues and the ready queue.

- Long–term (or job) scheduling is the selection of processes to be allowed to content for the CPU.

- Short term (or CPU) scheduling is the selection of one process from the ready queue.

- Reasons for concurrent execution : information sharing, computation speedup, modularity and convenience. Concurrent execution requires mechanisms for process creation and deletion.

- The processes executing in the OS may be either independent processes or cooperating processes.

- The job of a scheduling algorithm is to determine which process to run next, taking into consideration factors such as response time, efficiency and fairness.

- Each process has its own state, and can be thought of as running on its own virtual processor. Sometimes processes have to interact. For example, sharing a common buffer area.

- A process is a program in execution. Processes frequently need to communicate with other processes. Thus the term interprocess communication comes.

- When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, is called as race conditions.

- The part of the program where the shared memory is accessed is called the critical section.

  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.

- With multiple active processes having potential access to shared spaces or shared I/O resources, case must be taken to provide effective synchronization.

- Processes need to be synchronized to enforce mutual exclusion, cooperating processes may need to exchange information.

- Hardware support for mutual exclusive having potential access to shared spaces or shared I/O resources, care must be taken to provide effective synchronization.

- Software approaches of mutual exclusion include Dekkar's algorithm and Peterson's algorithm.

- Strong semaphores guarantee freedom, from starvation but weak semaphores do not.

- The monitor is a programming language construct that provide equivalent functionality as that of semaphores and that is easier to control.

- Only one process can be active in a monitor at any instant.

- Processes which use or consume the data produced by the producer process are called as “Consumer process”.

- All the producer processes communicate with the consumer process through a shared variable where the shared data is deposited. When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access, which is called starvation of that process for the resource. Starvation can be avoided by a first come, first served allocation policy.

- Deadlock can be defined as the permanent blocking of a set of processes that either complete for system resources or communicate with each other.

- A safe state is one in which there exists a sequence of events that guarantee that all processes can finish. An unsafe state has no such guarantee.

- The banker’s algorithm avoids deadlock by not granting a request if that request will put the system in an unsafe state.

- Deadlock can be prevented by numbering all the resources and making processes request them in strictly increasing order.

❑ ❑ ❑ ❑ ❑ ❑

ASSIGNMENT – 1

Duration : 45 Min. **Max. Marks : 30**

Q 1 to Q 6 carry one mark each

1. CPU scheduling may take place under _____
   (A) When a process switches from the running state to the waiting state, I/O request o
   r invocation of wait for the termination of one the child processes
   (B) When a process switches from the running state to the ready state, when an interrupt occurs.
   (C) When a process switches form the waiting state to the ready state.
   (D) All of the above

2. _____ is the module that gives control of the CPU to the process selected by the short term scheduler.
   (A) Dispatcher (B) Loader
   (C) Linker (D) Scheduler

3. Every time scheduling is done, all jobs present at that time are considered for scheduling. A job may get scheduled again and again. This is nothing but _____.
   (A) Static scheduling
   (B) Dynamic scheduling
   (C) Preemptive static scheduling
   (D) Preemptive dynamic scheduling

4. On Kernel when process P_i requests an I/O operation on some device d, and the I/O operations completed successfully, then process P_i changes its state _____.
   (A) first to blocked and remain as it is.
   (B) first to blocked and finally change to ready.
   (C) first to ready and remain as it is.
   (D) first to ready and finally change to blocked.

5. The solution for the producer–consumer problem without requiring mutual exclusion is given by _____ method.
   (A) Monitor (B) Event counter
   (C) Semaphore (D) Message–passing

6. The target thread can periodically check, if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion. This method is called as _____.
   (A) Asynchronous thread cancellation
   (B) Synchronous thread cancellation
   (C) Deferred thread cancellation
   (D) Periodic thread cancellation

Q7 to Q18 carry two marks each

7. In real time system, CPU utilization may range from _____ (for a lightly loaded system) and _____ (for heavily loaded system)

(A) 0% to 100% (B) 20% to 100%
(C) 40% to 90% (D) 20% to 90%

8.

PROCESS	Burst time (millisecond)
P1	22
P2	5
P3	6

Above processes arrive in the order P1, P2, P3 and are served in FCFS order. Then average waiting time is _____

(A) 16 milliseconds (B) 17 milliseconds
(C) 0.16 milliseconds (D) 0.17 milliseconds

9. Consider the following four processes, with the length of the CPU burst time in milliseconds.

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Using the Shortest–Remaining–Time–First (SRTF) scheduling, the average waiting time is ___

(A) 6.5 milliseconds (B) 7.5 milliseconds
(C) 7.75 milliseconds (D) 6.75 milliseconds

10. A process is executed in 100 msec. It takes 10 msec for the CPU to decide which process to execute. Then how much percentage of CPU time is used for scheduling.

(A) 10% (B) 9%
(C) 90% (D) 100%

11. "Degree of multiprogramming is stable" means

(A) Number of programs in the system is same at any time.
(B) Average rate of process creation is equal to the average departure rate of processes leaving the system.
(C) Number of programs executing in the system are using same number of resources.
(D) Average rate of execution is equal the average waiting time of process.

12. Consider the following performance table for FCFS scheduling,

Position in Batch	Job Arrival Time (Ai)	Job Completion Time (Ci)	Weighted turn around (Wi)
1	2	7	1.00
2	7	10	1.33
3	10	15	2.67
4	12	17	2.28
5	13	19	12.40

For this batch of processes, the throughput is _____

(A) 29.4 (B) 3.4

(C) 0.34 (D) 0.294

13. Which of the following statement is false in case of process scheduling in time sharing ?

(A) Process priorities depend on the nature of the processes.
(B) Processes are scheduled in a Round Robin manner.
(C) A running process is preempted when its time slice elapses.
(D) Processes may be swapped out of the memory.

14. Which of the following statement(s) is/are true regarding the Process Management ?

(1) The OS allocates the CPU time to various users based on certain policy.
(2) The PCB block is maintained by the CPU.
(3) The list of blocked processes is maintained in the priority order by the OS.
(4) The process switch occurs only if a process requests an I/O before the time slice is over or it consumes the full time slice.

(A) 1, 2, 4 are true (B) 4 is false

(C) 1, 4 are true (D) All the above are true

15. Which of the following statement is not correct w.r.t. Deadlock ?

(A) In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting

(B) A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, … P_{n-1} is waiting for a resource that is held by P_n, and P_n does not have to wait for any process.

(C) A resource can be released only voluntarily by the process holding it, after that process has completed its task.

(D) A mutual exclusion, hold and wait, no preemption and circular wait, all these conditions are not completely independent.

16. A sequence of processes $<P_1, P_2, \ldots P_n>$ is a safe sequence for the current allocation state if

(A) for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j > i$.

(B) for each P_j, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_i with $j < i$.

(C) for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$.

(D) for each P_j, the resources that P_i can still request can be satisfied by the currently available resources, with $j > i$

17. Consider the following algorithm which is designed to achieve safety state.

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize Work : = Available and Finish [i] : = false for i = 1, 2, 3, … n.
2. Find an i such that both
   (a) Finish [i] = false
   (b) $Need_i \leq Work$
   If no such i exists, go to step 4.
3. Work = Work + Allocation;
   Finish [i] = true
   go to step 2.
4. If Finish [i] = true for all i, then the system is in a safe state.

This algorithm may require an order of operations to decide whether a state is safe.

(A) $m \times n^2$ (B) $m^2 \times n$

(C) $\log (m \times n^2)$ (D) $\log (m^2 \times n)$

18. Consider the following situation :

Consider a process which copies from a card reader to a disk file, sorts the disk file, and then prints the result to a line printer and copies them to a magnetic tape. Then solution for this situation is :

(i) All resources must be requested at the beginning of the process, then the process must initially request the card reader, disk file, line printer and tape drive. It will hold the magnetic tape drive for its entire execution, even though it needs it only at the end.

(ii) The process requests initially only the card reader and disk file. It copies from the card reader to the disk and then releases both the card reader and the disk file. The process must then re–request the disk file and the line printer. After copying the disk file to the line printer, it releases both, and then requests the disk file and tape drive. It copies the disk file to tape, then releases these two resources and terminates.

(A) Only (i) (B) Only (ii)

(C) Both (i) and (ii) (D) Neither (i) nor (ii)

❑ ❑ ❑ ❑ ❑ ❑

TEST PAPER – 1

Duration : 30 Min. **Max. Marks : 25**

Q1 to Q7 carry one mark each

1. In a batch environment, a process is created _____
   (A) in response of Dispatcher
   (B) when a new user attempts to log on.
   (C) in response to the submission of a job
   (D) None of these

2. In time sharing system, the process for a particular user is to be terminated ____
   (A) when new user attempts to log on
   (B) when memory is unavailable
   (C) when user uses invalid instructions
   (D) when the user logs off or turns off his or her terminal.

3. Context – switch time highly depends on _____.
   (A) Software support
   (B) Hardware support
   (C) Memory
   (D) Available swapping space.

4. In UNIX,
   (A) a CPU–bound process is given higher priority than an I/O bound process.
   (B) an I/O bound process is given higher priority than a CPU– bound process.
   (C) both CPU – bound and I/O – bound processes are of equal priority.
   (D) it depends on the scheduling algorithm.

5. If one solves the problem of Producer – Consumer using semaphores, then empty semaphore will have initial value as
   (A) 0 (B) 1
   (C) number of slots in the buffer (D) Any non–zero value

6. The shortest-remaining-time first is same as _____
   (A) non preemptive SJF scheduling
   (B) preemptive SJF scheduling
   (C) non-preemptive FCFS scheduling
   (D) preemptive FCFS scheduling

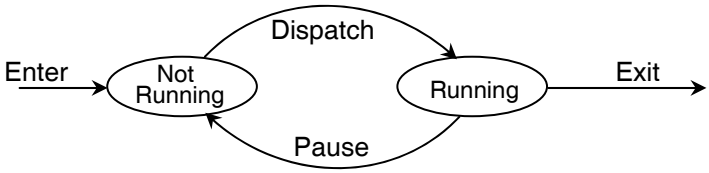
7. In case of priority scheduling, equal priority processes are scheduled in _____
   (A) Shortest remaining time first (B) First come last served
   (C) First come first served (D) Shortest job first

Q8 to Q16 carry two marks each

8. Which one of the following is reason for process termination?
   (i) Bounds violation (ii) Time overrun
   (iii) Operator or OS intervention (iv) protection error

   (A) (i), (ii), (iii), (iv)
   (B) (ii), (iii) (iv)
   (C) (i), (ii), (iii)
   (D) (i), (iv)

9. Consider the following process model and situation.



All the processes are present in the queue which operates in FIFO and processor operates in Round–robin fashion on the available processes (each process in the queue is given a certain amount of time, in turn, to execute and then return to the queue, unless blocked).

Then, which statement is not correct ?
(A) Some processes in the Not running state are ready to execute while others are blocked, waiting for an I/O operation to complete.
(B) Using a single queue, the dispatcher could not just select the process at the oldest end of the queue.
(C) The dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.
(D) The situation created by (A) (B) (C) does not have any feasible solution.

10. Which one is important measure, while selecting any scheduling algorithm?
   (i) Maximize CPU utilization under the constraint that the maximum response time is 1 second.
   (ii) Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.
   (A) (i)
   (B) (ii)
   (C) both (i) & (ii)
   (D) neither (i) nor (ii)

11. Consider a processor which supports with average length (excluding the process being serviced) = 20, and let 2 millisecond is the average waiting time in the queue. The arrival rate which is considered as average process arrival rate for a new process in the queue is 3 processes / second. Then at steady state number of processes leaving the queue = _____.
   (A) 6 (B) 20
   (C) 14 (D) 26

12. Which of the following is / are the reasons for Process Creation?
    (A) (i) New batch job (ii) New user attempts to log on
    (B) (i) Privileged instruction
    (ii) If a user request the particular file to be printed.
    (C) (i) For purpose of modularity or to exploit parallelism.
    (ii) The process attempts to use a resource or file that is not allowed to use.
    (D) None of the above.

13. Consider a computer with two processes, H with high priority and L with low priority. The scheduling algorithm is designed such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run. H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the _____.

    (A) prioritize producer consumer problem
    (B) priority inversion problem
    (C) priority Enabling – disabling problem
    (D) None of these

14. Which one is not advantage of multithreading programming?

    (A) Multithreading, an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.
    (B) Threads share the memory and the resources of the process to which they belong.
    (C) Multithreading on a multi–CPU machine is really not possible but thread may be running in parallel on a different processor.
    (D) It is economical to create and context switch threads and it is much more time consuming to create and manage processes than threads.

15. 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 algorithm, average waiting time is _____
    (A) 7 ms (B) 6 ms
    (C) 6.5 ms (D) 8 ms

16. If process P requires 8 units of CPU time such that it requires I/O at every 2 units of time then it is
    (i) I/O bound (ii) CPU bound
    (A) only (i) (B) only (ii)
    (C) both (i) and (ii) (D) neither (i) nor (ii)

❑ ❑ ❑ ❑ ❑ ❑