UNIT - II
Process and CPU Scheduling

1. The Process

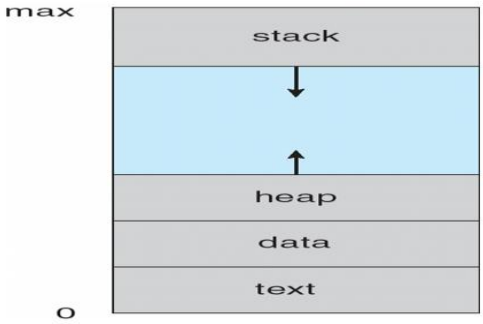
Process Definition:

A process can be thought of as a program in execution. A process is the unit of work in most systems.

A process will need certain resources—such as CPU time, memory, files, and I/O devices to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

Structure of a Process in Memory

- A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables).
- A **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.



When a Program becomes Process?

A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (Often called as **executable file**). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out).

If two processes are associated with the same program, are they same or different? (Or) Explain if you run same program twice, what section would be shared in memory?

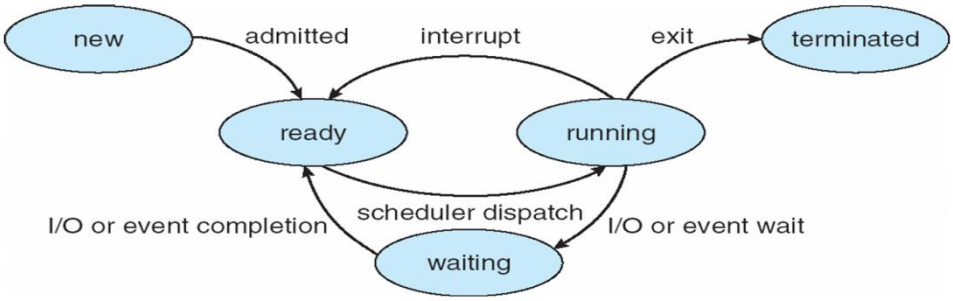
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

2. Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process.

A process may be in one of the following states:

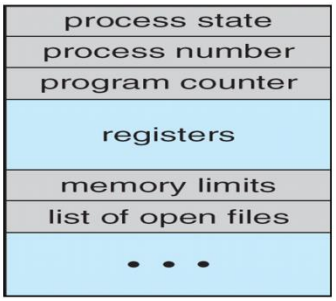
- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.



3. Process Control Block

Each process is represented in the operating system by a **Process Control Block (PCB)** or **Task Control Block.** It contains many pieces of information associated with a specific process, including these:

- **Process state:** The state may be new, ready, running, and waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.



Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

1. Scheduling Queues

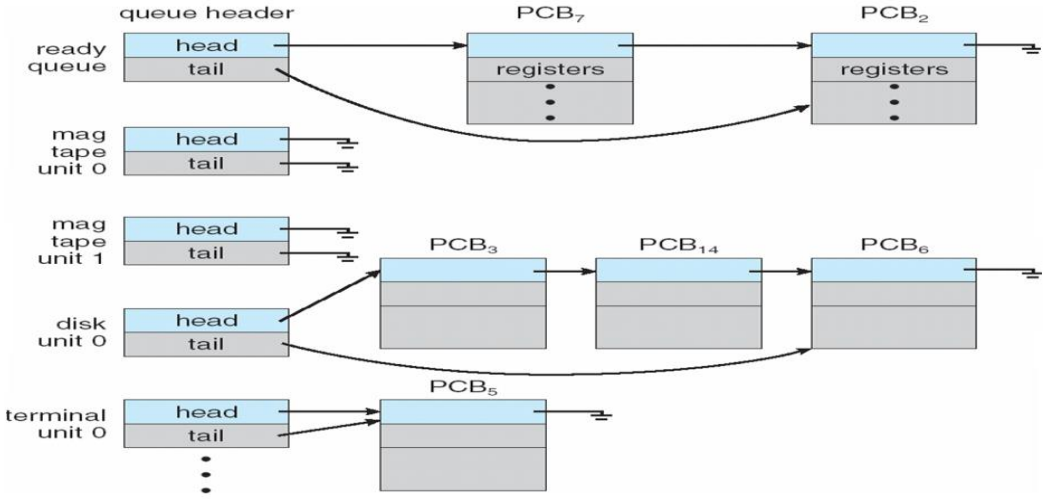
The following are the different queues available,

a. Job Queue

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

b. Ready Queue

- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.
- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.



c. Device Queue

- The list of processes waiting for a particular I/O device is called a **device queue**.
- Each device has its own device queue.

Queuing-diagram representation of process scheduling

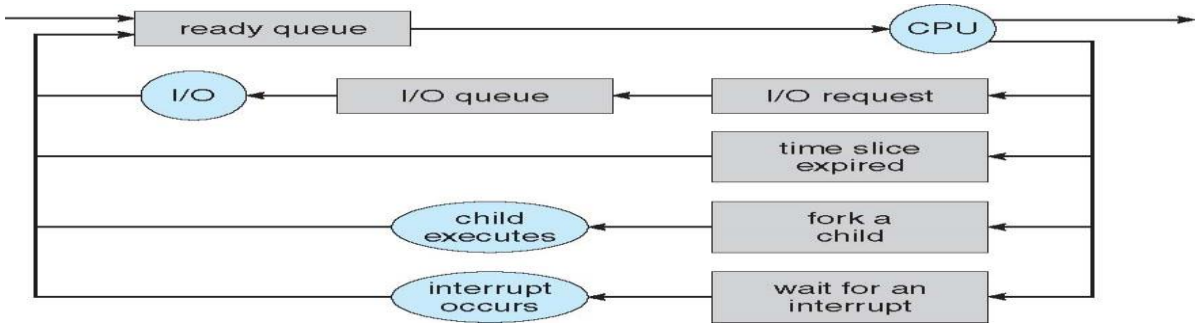
A common representation of process scheduling is a **queuing diagram**. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a

set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.



2. Schedulers

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

Types of Schedulers

a. Long-Term Scheduler or **Job Scheduler**

- Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), 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.
- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

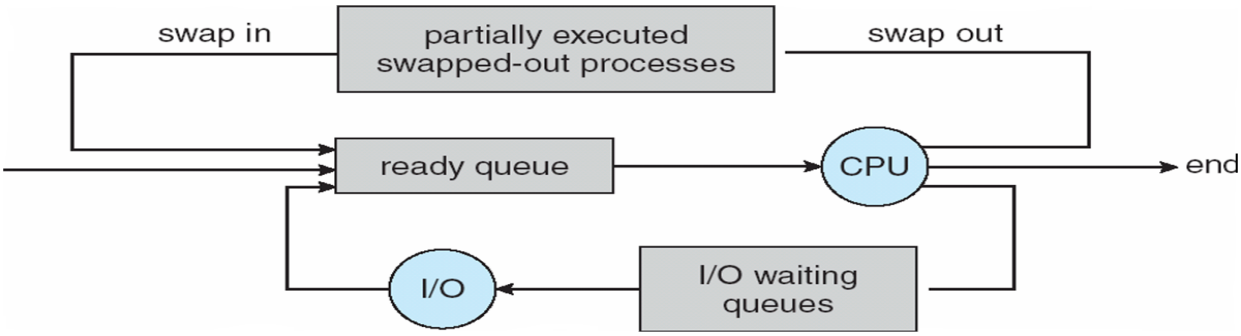
- It is important that the long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes.
- On some systems, the long-term scheduler may be absent or minimal.

b. Short-Term Scheduler, Or **CPU Scheduler**

- 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 short-term scheduler must select a new process for the CPU frequently.
- A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds.
- Because of the short time between executions, the short-term scheduler must be fast.

c. Medium-Term Scheduler

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



3. Context Switch

Definition: Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

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.

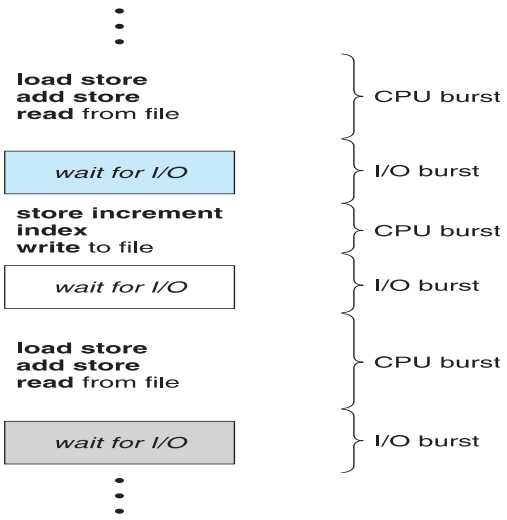
Overhead: Context-switch time is pure overhead, because the system does no useful work while switching.

Switching Speed: Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

Hardware Support: Context-switch times are highly dependent on hardware support. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch

4. CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate 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 final CPU burst ends with a system request to terminate execution.



Definition of Non Preemptive Scheduling

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x.

Definition of Preemptive Scheduling

Under this, a running process may be replaced by higher priority process at any time. Used from Windows 95 to till now. Incurs the cost associated with access to shared data. It also affects the design of OS.

Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode

- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch.

Dispatch Latency: The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Operations on processes (OR) System call interface for process management-fork, exit, wait, waitpid, exec

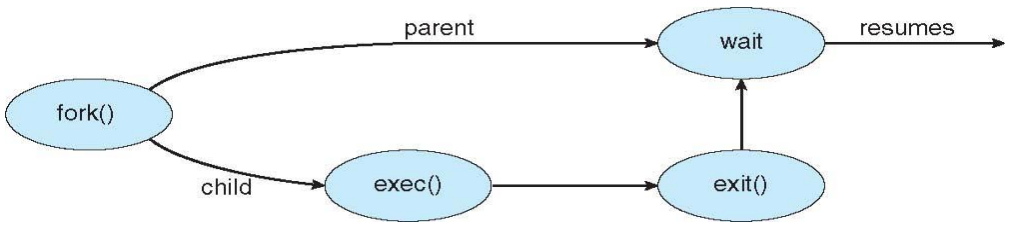
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

a. Process Creation

During the course of execution, a process may create several new processes. The creating process is called a parent process, and 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.

System Calls

- **fork()**
  - Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
  - A new process is created by the fork () system call. The new process consists of a copy of the address space of the original process.
  - This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork (), with one difference: the return code for the fork () is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- **exec()**
  - After a fork () system call, one of the two processes typically uses the exec () system call to replace the process's memory space with a new program.
  - The exec () system call loads a binary file into memory and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.
- **wait()**
  - The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child. Because the call to exec () overlays the process's address space with a new program, the call to exec () does not return control unless an error occurs.



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. At that point, the process may return a status value (typically an integer) to its parent process (via the wait () system call). All the resources of the process—including physical and virtual memory, open files and I/O buffers—are deallocated by the operating system.

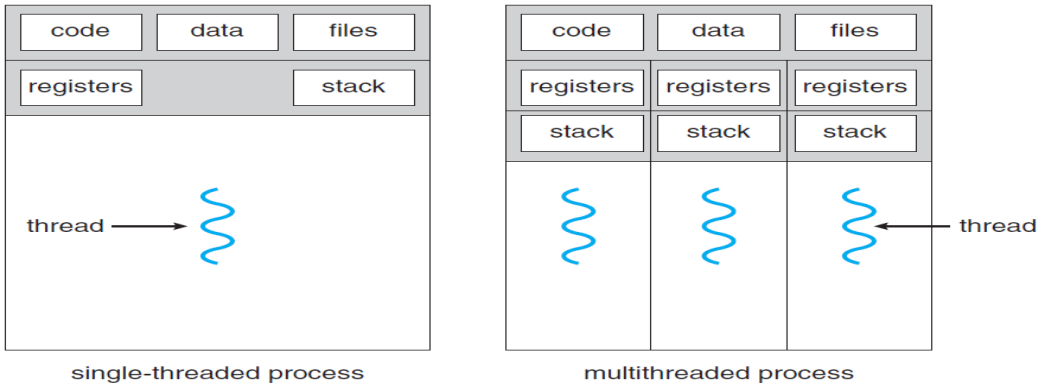
Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

Threads

Defining Thread

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.



Single Thread

- A process is a program that performs a single **thread** of execution.
- For example, when a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

Multi Thread

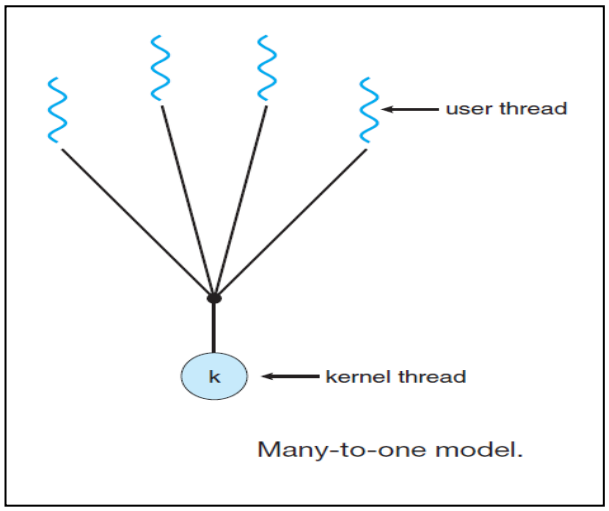
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.
- On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

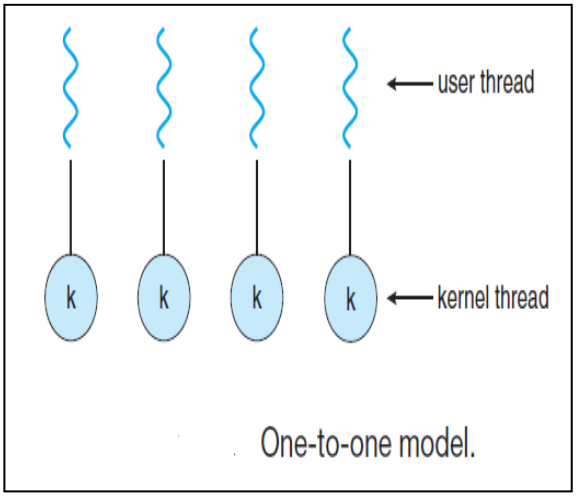
Multithreading Models

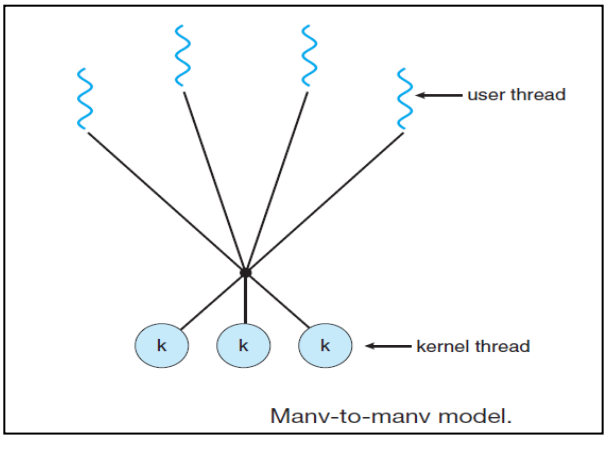
Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. The following are the three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to many models.

1. **Many-to-One Model**
   - The many-to-one model maps many user-level threads to one kernel thread.
2. **One-to-One Model**
   - The one-to-one model maps each user thread to a kernel thread.
3. **Many-to-Many Model**
   - It multiplexes many user-level threads to a smaller or equal number of kernel threads.







Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Scheduling algorithms

First-Come, First-Served Scheduling

- **First-Come, First-Served (FCFS)** scheduling algorithm is the simplest CPU-scheduling algorithm.
- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- There is a **convoy effect** as all the other processes wait for the 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.

- FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Example:

Shortest-Job-First Scheduling

- This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- A more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm knows the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. With short-term scheduling, there is no way to know the length of the next CPU burst.
- One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones.
- The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

Example:

Priority Scheduling

- The SJF algorithm is a special case of the general **priority-scheduling** algorithm.
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

- An SJF algorithm is simply a priority algorithm where the priority (*p*) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority.
- Priorities can be defined either **internally or externally**. **Internally** defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. **External** priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Priority scheduling can be either **preemptive or nonpreemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A **preemptive priority scheduling algorithm** will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A **nonpreemptive priority scheduling algorithm** will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- A solution to the problem of in definite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

Example:

Round-Robin Scheduling

- The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue. To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The

scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

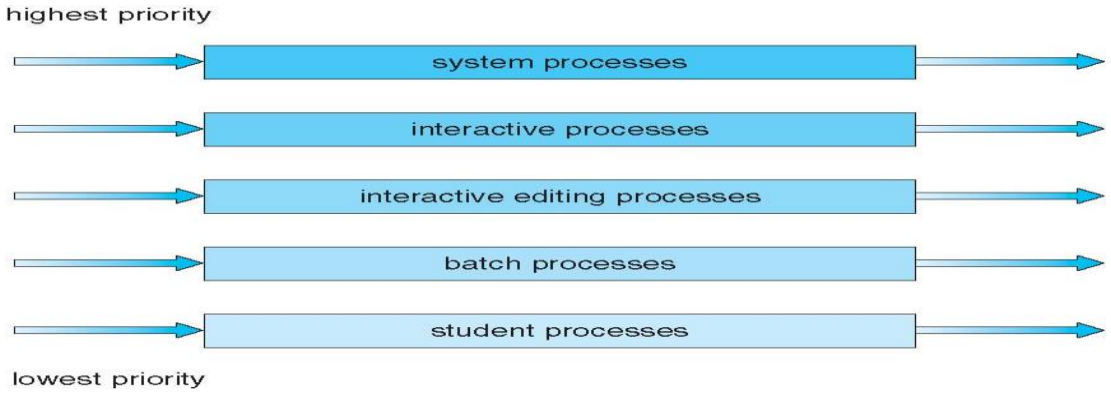
- The average waiting time under the RR policy is often long.
- The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. It creates a processor sharing and creates an appearance that each of n processes has its own processor running at 1/n the speed of the real processor.

Example:

Multilevel Queue Scheduling

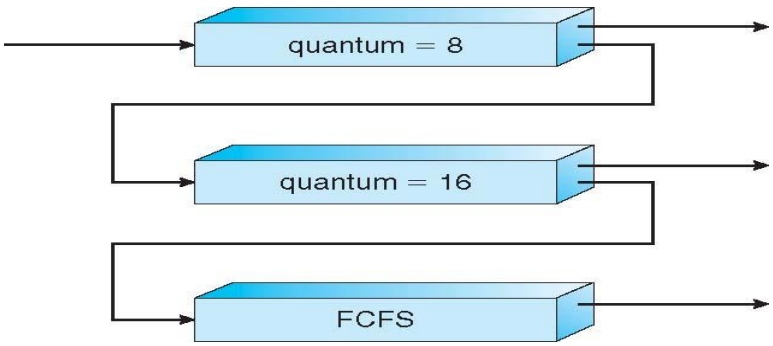
- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.
- Consider the example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:
  1. System processes
  2. Interactive processes
  3. Interactive editing processes
  4. Batch processes
  5. Student processes
- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the

background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.



Multilevel Feedback Queue Scheduling

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
- This setup has the advantage of low scheduling overhead, but it is inflexible.
- The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.



- A multilevel feedback queue scheduler is defined by the following parameters:
  - ❖ The number of queues.
  - ❖ The scheduling algorithm for each queue.

- ❖ The method used to determine when to upgrade a process to a higher priority queue.
- ❖ The method used to determine when to demote a process to a lower priority queue.
- ❖ The method used to determine which queue a process will enter when that process needs service.

Multiple- Processor Scheduling

The following are the several concerns in multiprocessor scheduling,

Approaches to Multiple-Processor Scheduling

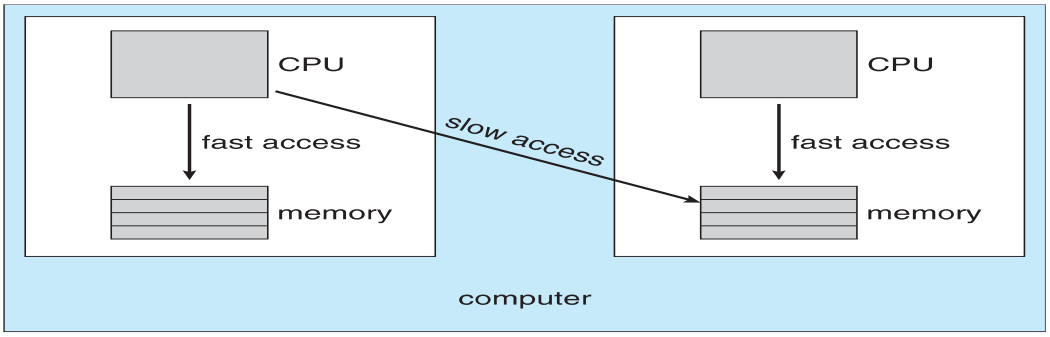
1. **Asymmetric Multiprocessing**
   - All scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code.
   - This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.
2. **Symmetric Multiprocessing (SMP)**,
   - A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
   - Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
   - If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
   - We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.
   - Virtually all modern operating systems support SMP, including Windows, Linux, & Mac OS X.

Processor Affinity

Most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**—that is, a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms,

- **Soft Affinity**: The operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
- **Hard Affinity:** It allows a process to specify a subset of processors on which it may run. The main-memory architecture of a system can affect processor affinity issues. Consider, non-uniform memory access (NUMA). The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system.



Load Balancing

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is necessary only on systems where each processor has its own private queue of eligible processes to execute.

There are two general approaches to load balancing:

1. **Push Migration**
   - With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.

2. **Pull Migration**
   - Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

Introduction to Dead locks

→ A lack of process synchronization can result in two extreme conditions are deadlock or starvation. Deadlock is the problem of multiprogrammed system.

→ Deadlock can be defined as the permanent blocking of a set of processes that either complete for system resources.

→ deadlock can occur on sharable resources such as files, printers, database, disks, tape drives, memory, cpu cycles etc.

→ A process is in deadlock state if it was waiting for particular event that will not occur. In a system deadlock, one or more processes are deadlocked.

Deadlock Example :-

→ System is collection of limited / finite no. of resources. These resources are distributed among a numbers of competing processes. Resources are of two types:

① Reusable resources.

② Consumable resources.

→ Reusable resource is used only by one process at a time. process can release resource after use. processors, I/O channel, I/O device, file, DB, primary & secondary memory, semaphores are example of the reusable resource.

→ Consumable resource is on that can be created & destroyed. There is no limit on the no. of Consumable resource of a particular type.

An interrupt, messages, signals and messages in I/O buffers are Example of consumable resources.

→ Process utilize the resources in the following sequence.

① Request for resource.

② use of resource.

③ Resource release.

Process uses system call for requesting the resource. After allocating resource to the process, it use or operate the resource.

→ The process, if ~~use or operate~~ Release the resource after use.

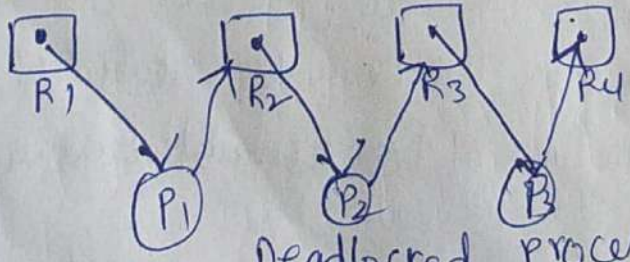
→ The resource is not available when it is requested, the requesting process is forced to wait.

Process 1	Process 2	Process 3	Process 4
Printf (" ")	int a;	Procedure ( )	Sort ( )
=	=	=	=
request (resource 1);	request (Resource 2);	request (resource 3);	request (resource 4)
/* Holding res 1 */	/* Holding res 2 */	/* Holding res 3 */	/* Holding res 4 */
=	=	=	=
Request (Resource 2)	request (Resource 3);	Request (Resource 4);	request (Resource 1)

Deadlock with 4 processes

→ Process 1 is holding resource 1 & requesting 2; Process 2 is holding resource 2 & requesting resource 3.

→ Here deadlock occurs because none of the processes can Proceed because all are waiting for a resource held by another blocked process



Deadlocked processes.

Resource Allocation Graphs :-

→ Resource allocation graph is introduced by Holt. It is a directed graph that depicts a state of the sim of resources & processes.

→ process & resource are represented by node in directed graph. Graph consist of a set of vertices (V) & set of edges (E)

fig : process.

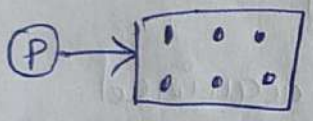
fig : Resource with 2 instances.

→ A process node is graphically represented by circle.

→ A resource node is graphically represented by rectangle

→ The number of bullet symbols in a resource node indicates how many units of that resource class exist in the system.

→ Claim edge $P \longrightarrow R$ indicated.

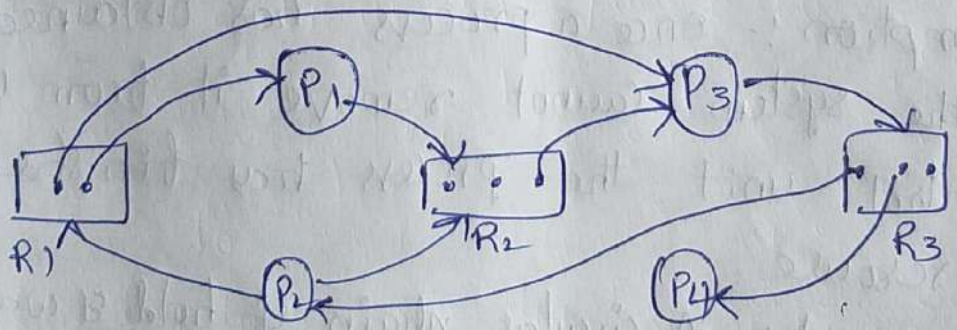


a) Request edge.

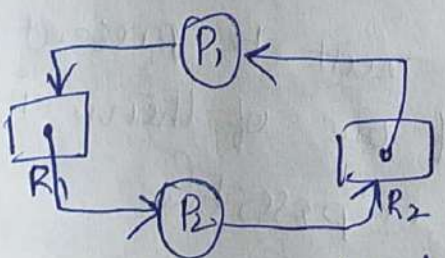
b) Claim edge.

Shows a resource allocation graph. System consist of process & resources.

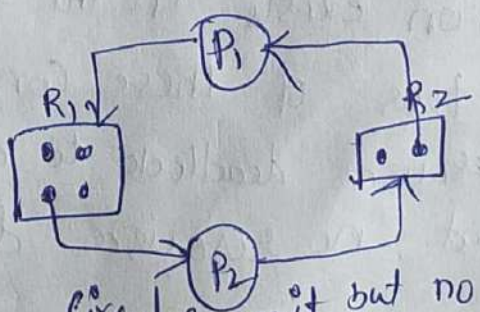
Process : P_1, P_2, P_3, P_4. Resource : R_1, R_2



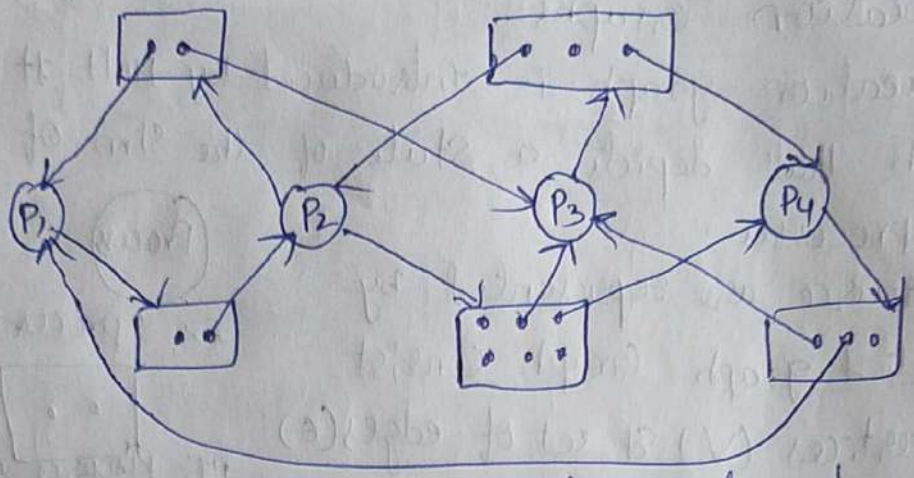
Resource allocation Graph.



Circular wait with deadlock.



Circular wait but no deadlock.



Resource allocation Graph.

Deadlock Characteristics :-

→ Necessary codition for Deadlock.

→ Following four conditions are necessary for deadlock to exist.

① Mutual exclusion ② Hold and wait

③ No preemption ④ circular wait.

① Mutual Exclusion :- A resource may be acquired exclusively by only one process at a time.

② Hold and wait :- processes currently holding resources that were granted earlier can request new resources.

③ No preemption :- Once a process has obtained a resource, the system cannot remove it from the process control unit the process has finished using the resource.

④ Circular wait :- A circular chain of hold & wait condition exists in the Sln.

→ All fours of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.

Deadlock Solution

There are four approaches for deadlock solution

① Deadlock prevention
② Deadlock avoidance
③ Deadlock detection
④ Deadlock recovery

→ In deadlock prevention, aim is to condition a system to remove any possibility of deadlock occurring. poor resource utilization may be possible.

→ Avoidance: Deadlocks can be avoided by clearly identifying safe states and unsafe states.

→ Detection: Deadlock detection methods are used in systems in which deadlocks can occur.

→ Recovery: used to resolve the deadlock from a system.

→ deadlock prevention and detection algorithm is used for ignoring the deadlock.

① Deadlock prevention:-

→ To prevent a deadlock, the OS must eliminate one of the four necessary conditions.

① mutual exclusion ② Hold and wait.
③ No preemption ④ Circular wait.

① mutual exclusion: It is necessary in any computer system because some resources (memory, CPU) must be exclusively allocated to one user at a time. No other process can use a resource while it is allocated to a process.

② Hold and wait :- If a process holding certain resources is denied a further request, it must release its original resources & if required request them again.

③ No preemption :- It could by bypassed by allowing the operating system to deallocate resources from process.

④ Circular wait : Circular wait can be bypassed if the operating system prevents the formation of a circle.

→ A deadlock is possible only if all four of these conditions simultaneously hold in the system.

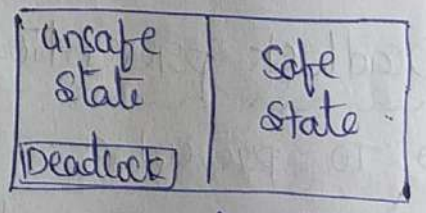
→ prevention strategies ensure that atleast on the conditions is always false.

② Deadlock Avoidance :-

→ Deadlock avoidance depends on additional information about the long term resource needs of each process.

→ The system must be able to decide whether granting a resource is safe or not & only make the allocation when it is safe.

→ when a process is created, it must declare its maximum claim, ie. the maximum no. of unit Resource.



Safe and unsafe state.

→ The resource manager can grant the request if the resources are available.

Bankers Alg is the deadlock avoidance algorithm.

→ Algorithm is checked to see if granting the request leads to an unsafe state. If it does, the request is denied.

Deadlock Avoidance

→ Simplest & most useful model requires that each process declare max no. of resource that it may need.

→ Deadlock avoidance alg dynamically examines the resource allocation can never be a circular condition.

→ If a system is in safe state there is no deadlock.

Avoidance: Ensure that a system will never enter a unsafe state.

Eg: Deadlock Avoidance by using Banker's Algorithm.

→ when a process gets all its resources then the job is over of that process it must be return in a finite amount of time.

Remaining

Process	Allocation A	B	C	maximum A	B	C	Available Current work	(Max − Allocation) Remaining need A	B	C
							3 3 2			
P_0	0	1	0	7	5	3	(10 → 7)	7	4	3
							5 3 2	(7−0)	(5−1)	(3−0)
P_1	2	0	0	3	2	2		1	2	2
							7 4 3			
P_2	3	0	2	9	0	2		6	0	0
							7 4 5			
P_3	2	1	1	4	2	2		2	1	1
							7 5 5			
P_4	0	0	2	5	3	3		5	3	1

105-7 ~~current work~~ Available

Add 7 2 5

Total Allocation A=10, B=5, C=7.

< P_1, P_3, P_4, P_0, P_2 >

↳ Safe sequence.

→ In this way the process will exites.

Remaining need = maximum − Allocation

Current work = Total − Allocation) we will get current work

→ we have to calculate the sequence.
→ we need to find out the safe state process.
→ so, that safe state process is assigned first.
→ we need to calculate the next safe state process.
→ now here we are ~~to~~ using Bankers algorithm.

need i ≤ work (current work.
(remaining need)

work = work + allocation (we need to update the work).

→ If the condition is satisfys we have to arrange in that sequence.
→ The resources should be allocated to remaining process.
→ For Example .P_0 - process uses 3 3 2 resources after completion of P_0 those resources are allocated to P_1 process. that P_1 process may use the that resources.

P_0 - need is 7 4 3. ($i \leq$ work)

= 743 ≤ 332

we need to check each individual value. not total value. ⇒ 743 ≤ 332 — x (unsatisfy) not in safe state

→ If

means P_0 resource are not suffient to execute or complete the P_0 work.

P_1 — 122 ≤ 332 — ✓ (satisfy)

w ⇒ 332 + 200

= 532

now update with 332

now P_2 600 ≤ 532 × (unsatisfy).

P_3 → 211 ≤ 532 ✓ (satisfy

w = 532 + 211

⇒ 743

P_4 → 531 ≤ 743 ✓ (satisfy)

work = 743 + 002

= 745

now update the work with current work

→ Now again come to P_0, when the work is over of all the resources of that particular process are allocated to another processes.

→ The resources order is in increasing order, so there may be availability to execute previous processes.

→ now P_0 → 743 ≤ 745 ✓ (satisfy)

⇒ w = 745 + 010

⇒ 755

→ update the current work.

→ now P_2 → 600 ≤ 755 (satisfy)

w = 755 + 302

= 1057

Dead Lock Detection :- Deadlock detection is the process of determining that a deadlock exists and identifying the processes & resources involved in the deadlock.

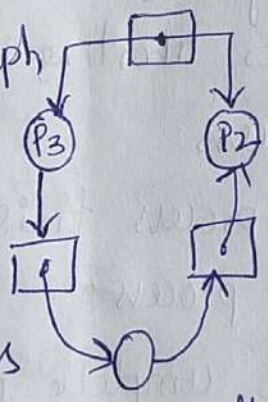
→ If processes are blocked on resources for an inordinately long time, the detection algorithm is executed to determine whether the current state is a deadlock.

wait for Graph :-

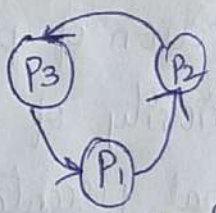
→ Any resources allocation graph with a single copy of resources can be transferred to a wait for Graph.

→ Fig shows resource allocation graph with corresponding wait for graph. The state of the system can be modeled by directed graph, called a wait for graph.

→ wait for Graph is a graph where each node represents a process. An edge $P_i \rightarrow P_j$ means that process P_i is blocked & waiting for process P_j to release a resource.



(a) Resource allocation Graph



(b) wait for Graph.

→ The wait for graph of a s/m is always smaller than the resource allocation graph of that same system.

→ There is a deadlock in a system if and only if there is a loop in the wait for graph of that system.

→ Deadlock detection involves two issues :

① maintenance of the wait for graph

② searching of the wait for graph for the presence of cycles.

→ For multiple instances of a resource type use an alg similar to Banker's Alg

→ Deadlock detection requires overhead for run time cost of maintaining necessary information & executing the detection Alg.

Deadlock Recovery

→ Once deadlock has been detected in the sys, the deadlock must be broken by removing one or more of the four necessary conditions.

→ Here one or more processes will have to be preempted, thus releasing their resources so that the other deadlocked processes can become unblocked.

① process termination :-

→ Deadlock is removed by aborting a process. But Aborting process is not easy. All deadlocked processes are aborted.

→ Circular wait is ~~aborted~~ eliminated by aborting one by one process. There will be lot of overhead.

→ Deadlock detection alg must rerun after each process kill.

→ Selection of process for aborting is difficult. Parameters are -

① Priority of the process

② what percentage the process finished its execution?

③ Resource is used by process.

④ Need of resources to complete process remaining operation.

⑤ How many processes will need to be terminated.

⑥ process type : Batch or interactive

⑦ Resource preemption :-

→ Some times, resource temporarily take away from its current process & allocate it to another process.

→ for selecting victim, following factors are considered.

① priority of the process, Higher priority process are usally not selected

② cpu time is used by process. The process which is close to completion are usually not selected.

(2) The numbers of other process that would be affected if this process were selected as the victim.

(3) Recovery through rollback :-

(1) when a process in a sim terminates, the sim performs a rollback by undoing every operation related to the terminated process.

(2) checkpointing a process means that its state is written to a file so that it can be restarted later.

(3) risk in this method is that the original deadlock may recover by the non determinancy of concurrent processing may ensure that this does not happen.

(4) Starvation :

→ Starvation is one type situation in which a process waits for an event that might never occur in the sm.

→ select the victim only for finite number of time. use rollback method for selecting victim process.

comparision blw Detection, prevention & avoidance methods

Parameters	Avoidance	Detection	prevention.
(1) Resource Allocation policy	midway blw that of detection & prevention	very liberal	Conservative undes commit resource.
(2) Different schemes	manipulate to find at least one safe path	invoke periodically to test for deadlock	preemption, resource ordering, requesting all resources at once.
(3) Advantages	No preemption necessary	Never delays process initiation.	No preemption necessary.
(4) Dis-advantages	Process can be blocked for long period	Inherent preemption losses	delays process initiation.

Process synchronization :-

⇒ logical control flows are concurrent if they overlap in time. this is known as concurrency.

→ Concurrency refers to any form of interaction among processes or threads.

Ex:- Hlw exception handlers, processes & unix signal handlers.

→ Concurrency is good for users because working on the same problem, simultaneous execution of programs.

→ Concurrency means that two or more calculations happen with in the same time & there is usually some sort of dependency blw them. parallelism means popular solution is interleaved processing.

→ But Concurrency describe a problem, while parallelism describe a solution.

Principle of Concurrency :-

* concurrent access to share data may result in data inconsistency. maintaining data consistency requires mechanisms to ensure the ordarly execution of cooperating processes.

Following are the example of concurrency in different types of os.

① Concurrency in multiprograming :- An interaction between multiple processes running on cpu.

② Concurrency in multithreading :- An interaction blw multiple threads running in one process.

③ Concurrency in multiprocessors: An interaction blw multiple cpu's running multiple processes or threads.

④ Concurrency in multi-computers: An interaction between multiple computers running distributed processes or threads.

→ Java is concurrent programing language.

Process synchronization means sharing s/m resources by processes in a such a way that, concurrent access to shared data is handled there by minimizing the chance of in-consistent data.

→ It is required in uni-processor s/m, multiprocessor s/m, Network.

Race Condition:-

→ Race condition occurs when two or more operations occur in an undefined manner.

when two or more processes are reading or writing some shared data & the final result depends on who runs precisely. when, are called Race condition.

→ there is a "race condition" if the cpu outcome depends on the order of the execution. The out come depends on the cpu scheduling or "interleaving" of the threads.

Ex:- Bank Balane → Deposite

Bank Balame → Trancefer.

→ The concurrent execution of two processes is not guaranteed to be determinate, since different executions of the same program on the same data may not produce the same result.

Operating Sln concerns:-

Design & management issue for concurrency are as follows

① Track of various processes is kept by OS.

② OS allocates & deallocate slw & hlw resources to active process

③ Process execution speed do not depends upon other process execution speed.

④ OS protect user data & physical resources from un-authorized process.

Critical section problem:-

→ A critical section is a block of code that only one process at a time can execute so, when one process is in its critical section, no other process may be in its critical section.

→ The critical section problem is to ensure that only one process at a time is allowed to be operating in its critical section.

* Critical section means process may change some common variable, writing files, updating memory location, updating a process table etc.

→ When process is accessing shared modifiable data, it is said to be in a C.S.

→ Each process take permission from OS to enter into the critical section. structure as follows.

① Entry section

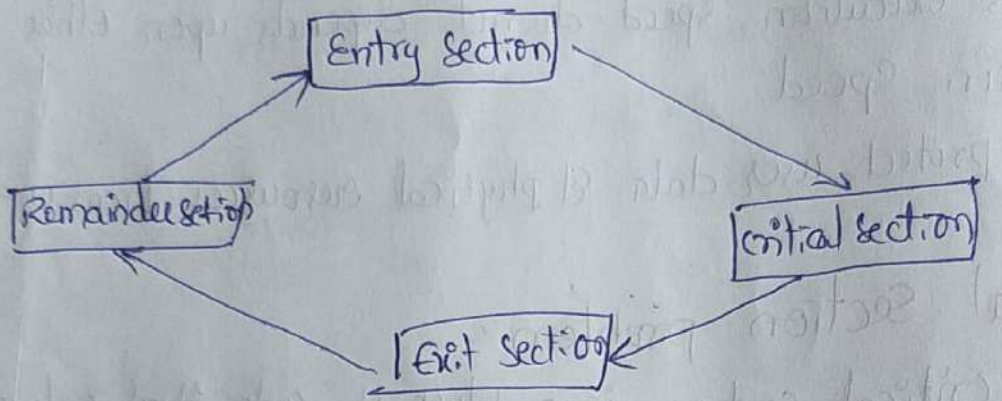
② Remainder section

③ Exit section.

① Entry section: It is a block of code executed in preparation for entering critical section

② Exit section: The code executed upon leaving the CS

③ Remainder section: Rest of the code is remainder section.



critical section.

* Each process cycles through remainder, entry, critical, exit sections. In this order.

Ex:- Process 1 int count = 5 Process 2

Count = count + 1 Count = count - 1

O/P: Count = ? (6 or) 5.

General framework :- Every solution will have the following layout

```
Entry section
critical section
Exit Section
Remainder section
```

→ To prevent critical section problem, the system should ensure that only one process at a time can execute the instruction in its critical section for a particular resource.

(15) Solution of critical section:-

→ solution to critical section to problem must satisfy the mutual exclusion, progress, & bounded waiting parameters.

→ Hlw & slw solution to critical section problem is available.

→ process using kernal is active in the Sln at any given time. Race condition is with implementing a kernal code.

→ kernal data structure maintains a list of all open files in the Sln.

→ OS handles critical section problem by using kernal. kernal classified into ① preemptive kernal.
② Non-preemptive kernal.

Requirements of mutual exclusion:-

① At any time, only one process is allowed to enter in its critical section.

② Solution is implemented purely in slw on a Machine.

③ A process remains inside its CS for a bounded time only.

④ A process must not be indefinitely postponed from entering its critical section.

⑤ A process cannot prevent any other process for entering into critical section.

⑯ Critical Region

→ Critical Region is an area of a process which is sensitive to inter-process communications. To guarantee mutual exclusion only one process is allowed to enter its critical Region at a time.

Mutual Exclusion :-

→ The need for mutual exclusion comes with concurrency. There are several kinds of concurrent exclusion.

1. Interrupt handlers
2. Interleaved preemptively scheduled threads.
3. multiprocessor clusters.
4. Distributed Systems.

mutual exclusion methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

Approaches to implementing mutual exclusion :-

1. Software Method
2. Hardware Method
3. Programing language method.

Requirements of mutual Exclusion :-

1. At any time, only one process is allowed to enter in its CS.
2. sln is implemented purely in Slw on a machine
3. A process remains inside it CS for a bounded time only
4. A process cant prevent any other process for entering into Critical section.

Lock :-

→ Lock is shared variable. Software mutual exclusion required that a process read a variable to determine that no other process is executing a CS, then set a variable know as lockvar

→ It indicates that the process is executing its CS.

→ Initially lock variable is initialized with zero. The process set it to 1 & enters its critical section.

Lock = 0 No process in the CS.

Lock = 1 Process in the CS.

→ Using simple lock variable, process synchronization problem is not solved. To avoid this, spinlock is used.

→ A lock that uses busy waiting is called a spin lock.

Mutex :-

→ mutex is used to ensure only one thread at a time can access the resource protected by the mutex.

→ The process that locks the mutex must be the one that unlocks it.

→ Mutex is good only for managing mutual exclusion to some shared resource.

→ Mutex is easy & efficient for implement.

→ Mutex can be in one of two states ① locked
② unlocked.

Peterson's solution :- Peterson's solution is sw based solution for critical section problem.

→ This algorithm will not work correctly on the modern computer architecture

→ It is simpler algorithm for two process mutual exclusion with busy waiting.

→ This algorithm less complicated than dekker's Alg.

→ peterson's algorithm solves critical section problem of two processes only. It does not require any special hardware.

→ Two processes alternates execution between their critical section & remainder sections.

→ deadlock & indefinite postponement are impossible in peterson's solution as long as no process or thread terminates unexpectedly.

Synchronization Hardware :-

Test & Set Operations.

→ Test & set is a single individual machine instruction. It is simply known as TS. It is introduced by IBM.

→ In a one machine cycle, it tests to see if the key is available & , if it is , sets it to unavailable. TS are a a hardware solution for critical section problem.

→ TS instruction eliminates the possibility of preemption occuring during the interval. The instruction : TestAndSet(p,q).

→ The instruction reads the value of q , which may be either true or false. Then the value is copied into 'p' & the instruction sets the value of 'q' to true.

* Process P_i would test the condition code using TS instruction before entering a critical section. If no other process was in this CS, then process P_i would be allowed to proceed & condition code would be changed from zero to one.

→ when process P_i exist the CS, the condition code is reset to 'zero' so another process can enter into CS.

→ If proces P_i finds a busy condition code then it is placed in a waiting loop. where it continues to test the condition code & waits until it is free.

Advantages:- Simple to implement, It can be used to support multiple CS. It works well for a small no. of processes.

Drawbacks:- It suffers from starvation, there may be deadlock. There is possibility of busy waiting.

Semaphores :-

* Semaphore is described by dijkstra. Semaphore is a non-negative integer variable that is used as a flag.

→ Semaphore is an OS abstract data type. It takes only integer value. Its used to solve critical section problem.

→ Dijkistra introduced two operations (P & V) to operate on semaphore to solve process synchronization problem.

→ A process calls the 'P' operation when it wants to enter its CS & calls 'V' operation when it wants to exit its CS.

→ P operation is also called as 'wait' operation.

→ V " " " " "signal" "

→ A wait operation on a semaphore decease its value by one.

wait: while S<0
do loops;
S:=S-1

→ A signal operation increments its value.

signal: S:=S+1;

→ A proper semaphore implementation requires that P & V be invisible operations. A semaphore operation is atomic.

→ There is no guarantee that no two processes can execute wait & signal operations on the same semaphore at the same time.

Binary semephore:-

* Binary semaphore is also known as mutex locks. It deals with the critical section for multiple processes.

+ Binary semaphore value is only blw 0 & 1.

Counting semaphore:-

→ used with that resource which has a finite no. of instances.

→ It works over unrestricted domain.

→ nonbinary semaphore often referred to as either a counting semaphore or general semaphore.

→ Process waiting for semaphore is stored in the Queue for binary & counting semaphore. Queue uses FIFO policy.

Busy waiting:-

* Busy waiting is a situation in which a process is blocked on a resource but does not yield the processor. A Busy wait keeps the CPU busy in executing a process even as the process does nothing.

→ Busy waiting is also called as spin waiting.

(18) Properties of semaphores:

→ Semaphores are machine independent
→ Semaphores are simple to implement.
→ Correctness is easy to determine.
→ Can have many different critical sections with different semaphores.
→ Semaphore acquire many resources simultaneously.

Drawbacks of semaphore:

→ They are essentially shared global variables.
→ Access to semaphores can come from anywhere in a program.
→ There is no control or guarantee of proper usage.
→ There is no linguistic connection b/w the semaphore & the data to which the semaphore controls access.
→ They serve two purposes, mutual exclusion & scheduling constraints.

Semaphore programming.

① semget() :- To create a semaphore or gain access to one that exists, the semget system call is used.
→ The segment system call takes three arguments.
① The first argument, key, is used by the system to generate a unique semaphore identifier.
② The second argument nsems, is the no. of semaphores in the set.
③ The third argument semflag, is used to specify access permission and/or special creation condition.
→ If the segment system call fail, it returns -1 & sets the value stored in errno.
→ when a semaphore is first created, the kernal assigns to it an associated

a) Semaphore control Block (SCB)
b) A unique ID
c) A value (binary or a count)
d) A task-waiting list.

→ A kernal can support many different types of semaphores, including binary, counting & mutex semaphore.

Semctl() function :-

→ The Semctl system call allows the user to perform a variety of generalized control operations on the SIm Semaphore structure, on the semaphores as a set, & on individual semaphores.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int Semctl (int semid, int semnum, int cmd, union semnumarg)
```

→ The semctl system call takes four arguments.

① The first argument semid is a valid semaphore identifier that was returned by a previous semget SIm call.
② The second argument to semnum, is the no. of semaphores in the semaphore set
③ The third argument to semctl, cmd, is an integer command value. The cmd value directs semctl to take one of several control actions. Each action requires specific access permissions to the semaphore control structure.
④ The fourth argument to semctl arg, is a union of type semun. Given the action specified by the preceding cmd argument, the data in arg can be one of any of the following four values.

(19) a. An integer already was set in the val member of sem. union that used with SETVAL to indicate a change.
b. A reference to a semid-ds structure where information is returned when IPC_STAT.
c. A reference to an array of type unsigned short integers: the array is used either to initialize the semaphore set specifying GETALL
d. seminfo structure when IPC_INFO is required
→ If semctl fails, it returns -1 & sets errno to indicate the specific error.

Semop() Function :-

* Additional operations on individual semaphores are implemented by using the semop system call. Its syntax is

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys.sem.h>
int semop (int semid, struct sembuf *sops, unsigned nsops);
```

* The semop system call takes 3 arguments.

① The first argument, semid is a valid semaphore identifier that was returned by a previous successful semget system call.

② The second argument, sops, is a reference to the base address of an array of semaphore operations that will be performed on the semaphore set associated with by semid value.

③ The third argument, nsops, is the no. of elements in the array of semaphore operations.

If semop fails, it returns a value of -1 & sets errno to indicate the specific semaphore operations.

→ If a semop call must block after completing some of its component operations, the kernal rewinds the operation to the beginning to ensure atomicity of the entire call.

→ when semop value is negative, the process specifying the operation is attempting to decrement the semaphore.

→ when semop value is positive, the process is adding to the ~~college~~ semaphore value.

→ Again, when semaphore value is to be modified, the accessing process must have alter permission for the semaphore set.

→ when it zero, the process is testing the semaphore to determine if it is at 0.

Sem-post():-

Syntax:- #include <semaphore.h>

int sem_post (sem_t *sem);

→ The sem-post() function unlocks the specified semaphore by performing a semaphore unlock operation on that semaphore. when this operation results in positive semaphore value, no threads were blocked waiting for the semaphore to be unlocked: the value is simply incremented.

→ When this operation results in a semaphore value of zero, one of the threads waiting for the semaphore is allowed to return successfully from its invocation of the sem-wait function.

Sem_init():

Syntax:- #include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value);

* The sem_init() function is used to initialize the semaphore's value. The pshared argument must be '0' for semaphores local to a process. The value argument specifies the initial value for the semaphore.

→ The pshared argument indicated whether this semaphore is to be shared blw the threads of a process. or blw process

* If pshared has the value 0, then the semaphore is shard blw the threads of a process, & should be located at some address that is visible to all threads.

* If pshared is non zero then the semaphore is shared blw processes. & should be located in a region of shaled memory.

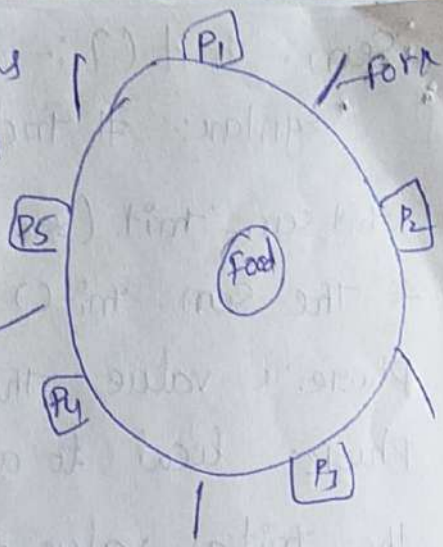
Classical problem of synchronization.

① Dining philosopher problem :-

* Dining philosphers problem is one of classical process synchronization problem. Here five philosphers are seated around a circular table. They spend their lives in thinking & eating. Five plates with five forks are kept on the circular table.

* while philosphers think, they ignore the eating & do not require fork. when a philospher decides to eat, then he/she must obtain a two fork, one from left side & another from right side fork.

* After consuming a food, the philosphers replace the fork & resumes thinking. A philosphers to the left or right of a dinning philosphers cannot eat while the dining philosphers is eating, since forks are a shared resource.



Code:

```
void dinning – philosphers ()
{
while (true)
{
Thinking ();
eating ();
}}
void eating ()
{
  take left fork ();
      Take right fork (i);
      eat food (delay);
      put right fork ();
       put left fork ();
}
```

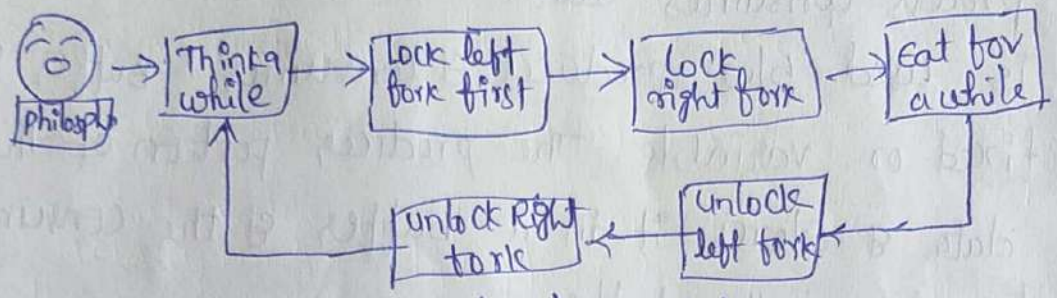
→ Here philosphers operates asynchronously & concurrently. It is possible for each philosphers to enter into eating mode.

Problem Analysis :-

Shared resource :- Here each fork is shared by two philosphers so we called it is a shared resource.

Race condition :- we do not want a philosphers to pick up a fork that has already been picked up by his neighbor.

21. Solution:- we consider each fork as a shared item & protected by a mutex lock. Before eating, each philospher first lock to left fork & then right fork. then philosphers have two locks so he can eat a food.



philosopher states.

Reader-writer problem:

→ when two types of processes need to access a shared resource such as a file or database. They called these Procedures reader & writers.

→ For example in railway reservation s/m, readers are those who want train information. because readers do not change the content of db. many readers may access the db at once. no need to enforce mutual exclusion.

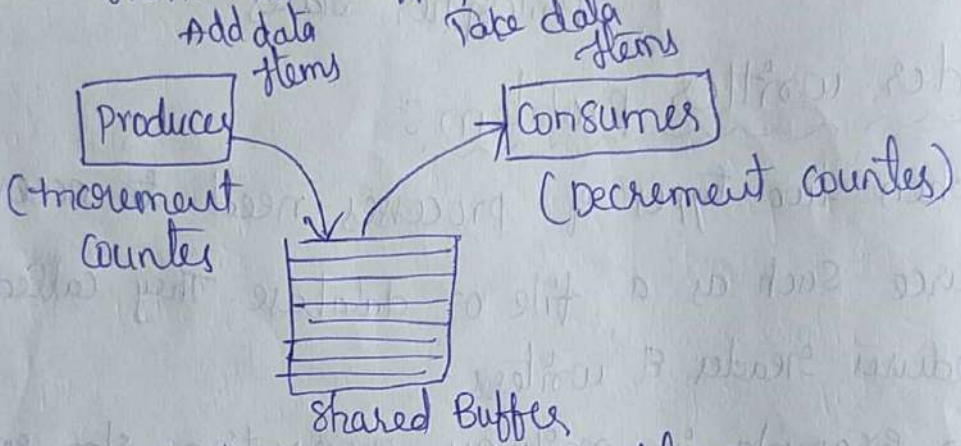
→ The writers are those who making reservations on a particular train. can modify the database, so it must have exclusive access. when writer is active, no other readers or writers may be active. it must enforce mutual exclusion.

→ If Reader having higher priority than the writes, then there will be starvation with writers. for writer having higher priority than reader then starvation with readers.

The producer consumer problem :-

→ producer-consumer problem is example classic problems of synchronization producer process produce data item that consumer process consumes data.

* Buffer is used b/w producer & consumer. Buffer size may be fixed or variable. The producer portion of the apln generates data & stores it in a buffer. & the consumer reads data from the buffer.



Producer-consumer problem

→ producer consumer is also called bounded buffer problem

⇒ The producer consumer problem shows the need for synchronization in Slm where many processes share a resource.

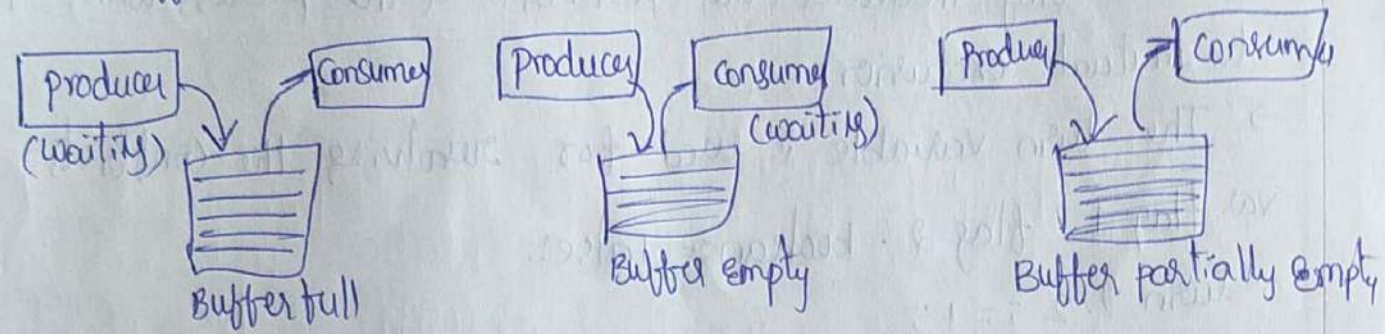
→ Problem arises when the buffer is full & producer wants to add a new data item. Solution to this is that producer goes to sleep until consumer removes data item from the buffer. Similar also to exist with consumer.

→ In order to synchronize these processes, both producer & consumer are blocked on some condition.

Example for producer consumer problem :-

printing word file.

22. Producer - consumes problem is solved by using semaphore, mutex and monitor Mutual exclusion must be enforced on Buffer itself.



Code for Producer

```
Producer (void)
{
  int item;
  while (TRUE)
  {
  Produce item (& item);
  Produce - if (counter == N)
  sleep();
  enter - item (item);
  counter = counter + 1;
  if (Counter == 1)
  wakeup (consumer);
  }
}
```

Code for Consumer

```
Consumer (void)
{
  int item;
  while (TRUE)
  {
  if (count == 0)
  sleep()
  remove - item (& item);
  Counter = counter - 1;
  if (count == N-1)
  wakeup (Producer);
  Consume - item (item);
  }.
```

Dekkar's Solution :-

→ Dekkar's solution was the first correct solution to the critical section problem for two processes / Threads. It uses an array of Boolean values & and an integer variable.

→ Dekkars algorithm is purely software solution with no special purpose h/w instruction. It uses flag to indicate a process desire to enter its critical section.

* processes are loops indefinitely, repeating entering & reentering its critical section. Here flag & turn are two global variables. variable flag indicates the position of the process with respect to mutual exclusion.

→ The turn variable is used for resolving the conflicts.

```
var flag 1, flag 2 : bookean := false;
turn 1.2 := 1;
```

```
P1 : begin
...
flag 1 := True
 turn := 2
 while flag 2
and turn = 2 do;
<critical section>
flag 1 := false;
end P1.
```

```
P2 : begin
...
flag 2 := true;
 turn := 1;
 while flag 1
 and turn = 1 do;
 <critical section>
  flag 2 := false;
  end P2.
```

→ Dekkars algorithm is correct & its satisfies mutual exclusion. & it is free from deadlock & starvation.

→ The process P_1 indicates its desire to enter its critical section by setting its flag to true. The process then proceeds to the while test & determine whether P_2 also wants to enter its critical section.

Limitations

→ It does not provide strict alternation.

→ It will not work with many modern CPU's

→ It won't work on Symmetric multi-processors (SMP) CPU's.

Drawbacks of slw solutions

* There is possibility of busy waiting
* Programming part is complicated.
* Makes difficult assumptions about the memory s/m.

Monitors :-

* Monitors is an object that contains both data & procedures needed to perform allocation of a shared resource.

→ Monitor is implemented in programming languages like pascal, Java & C++.

→ Monitor is an abstract data type for which only one process may be executing a procedure at any given time.

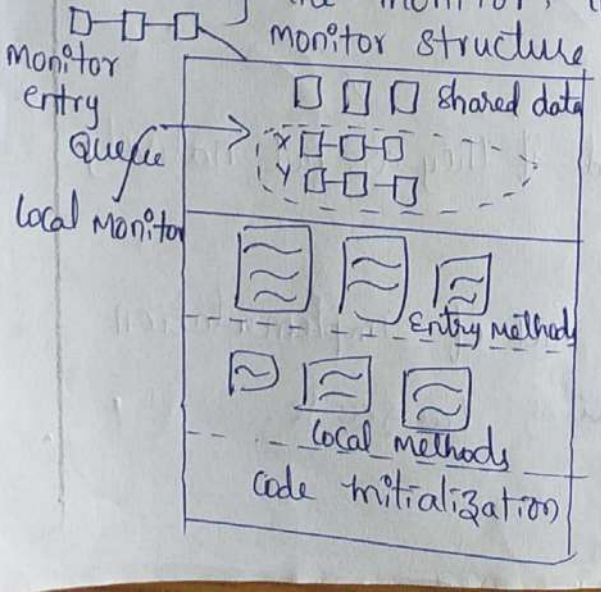
→ Monitor is a collection of procedure, variables and data structure.

→ Data inside the monitor may be either global to all routines within the monitor or local to a specific routine.

→ If the data in monitor represents some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

→ when a process calls a monitor procedure, the first few instruction of the procedure will check to see if any other process is currently active within the monitor.

→ If process is active then 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



→ Monitor supports synchronization by the use of condition variables that are contained within the monitor & accessible only within the monitor. Every conditional variables has associated queue.

cwait (condition variable)

csignal (condition variable.

cwait :- It suspended execution of the calling process on condition.

csignal: Resume execution of some process blocked after a cwait on the same condition.

The cwait must come before the c signal.

→ CPU is a resource that must be shared by all processes. The part of the kernal that apportions CPU time b/w processes is called the Scheduler.

→ A condition variables is like a semaphore, with two differences.

① A semaphore counts the no. of excess up operations, but a signal operation on a condition variable has no effect until some P is waiting

② A wait on a condition variable automatically does an up on the monitor mutex & blocks the caller.

Drawbacks of Monitors:-

* There is possibility of deadlocks in the case of nested monitor calls.
* monitor cannot easily be added if they are not natively supported by the language.
* monitor access concept if its lack of implementation most commonly used programming language.