RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Operating System**

Subject Code: **IT-501**

Semester: **5th**

Unit 2

Process management: Process management is the process by which operating systems manage processes, threads, enable processes to share information, protect process resources and allocate system resources to processes that request them in a safe manner. This can be a daunting task to the operating system developer and can be very complex in design.

Basic Concepts of CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- In a uniprocessor system, only one process may run at a time; any other processes must wait until the CPU is free and can be rescheduled

CPU–I/O Burst Cycle

- 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 , followed by an I/O burst , then another CPU burst ... etc
- The last CPU burst will end with a system request to terminate execution rather than with another I/O burst.
- The duration of these CPU burst have been measured.
- An I/O - bound program would typically have many short CPU bursts, A CPU- bound program might have a few very long CPU bursts.
- This can help to select an appropriate CPU - scheduling algorithm.

CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc. thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

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:

- 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, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency. Dispatch Latency can be explained using the below figure:

Types of CPU Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for I/O request or invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, completion of I/O).
4. When a process terminates.

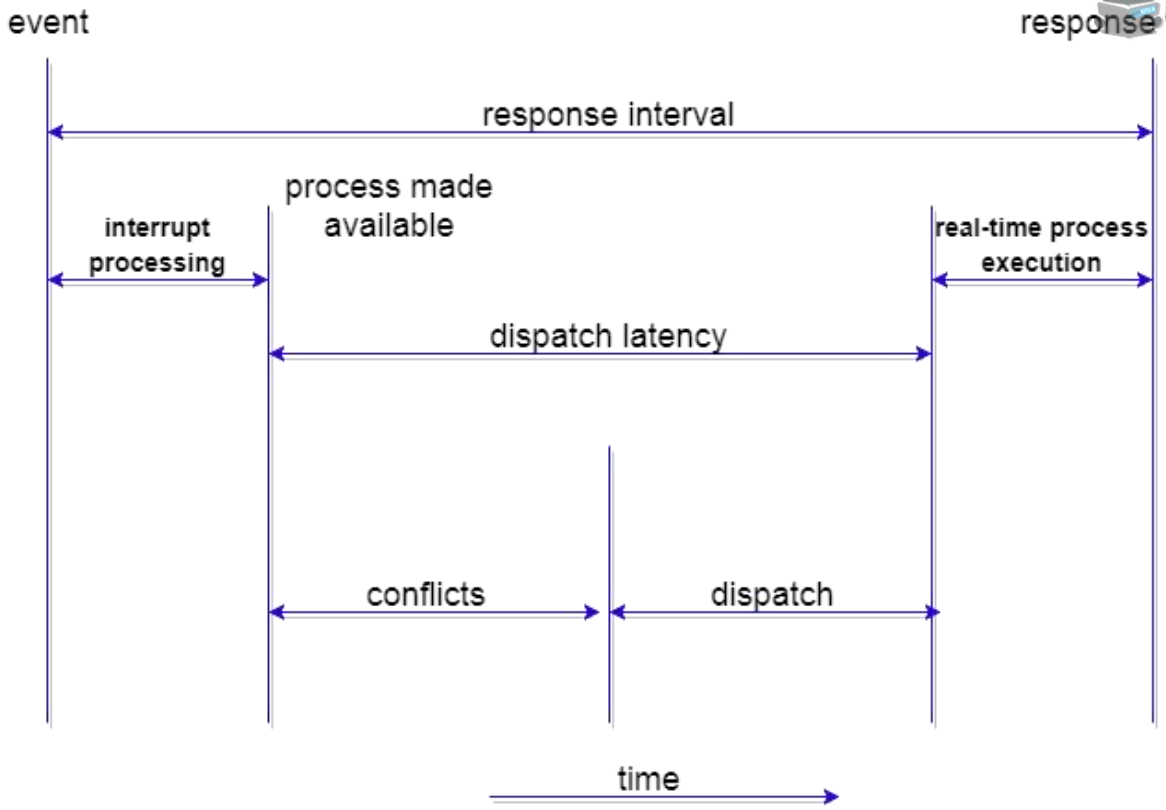


Figure 2.1 CPU Scheduling

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however in circumstances 2 and 3.

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non-preemptive; otherwise the scheduling scheme is preemptive.

Non-Preemptive Scheduling

Under non-preemptive 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 is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Scheduling Criteria

There are many different criteria's to check when considering the "best" scheduling algorithm :

- CPU utilization
  To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time (Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- Throughput
  It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.
- Turnaround time
  It is the amount of time taken to execute a particular process, i.e. the interval from time of submission of the process to the time of completion of the process (Wall clock time).

- Waiting time
  The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- Load average
  It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- Response time
  Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

Scheduling algorithm:

A Process Scheduler schedules different process to be assigned to the CPU based on particular scheduling algorithms. There are six popular process-scheduling algorithms, which we are going to discuss in this chapter-

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithm are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

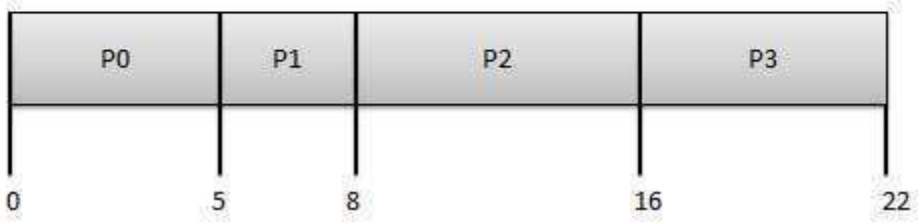


Figure 2.2 FCFS

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job First(SJF)

- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known
- The processer should know in advance how much time process will take

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8

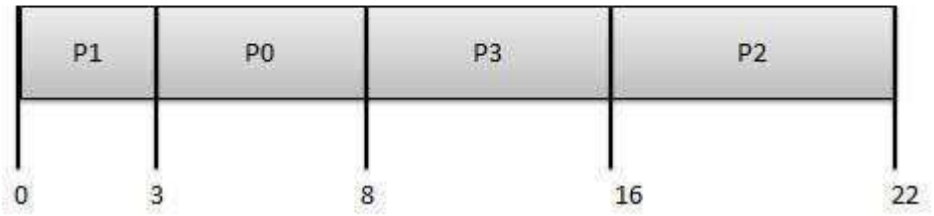


Figure 2.3 SJF

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14
P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority process with highest priority is to be executed first and so on
- Processes with same priority are executed on first come first served basis
- Priority can be decided based on memory requirements, time requirements or any other resource requirement

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0

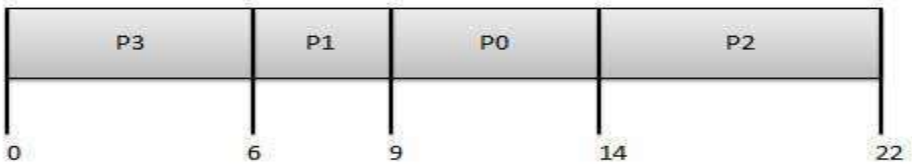


Figure 2.4 Priority Based Scheduling

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion
- Impossible to implement in interactive systems where required CPU time is not known
- It is often used in batch environments where short jobs need to give preference

Round Robin Scheduling

- Round Robin is the preemptive process-scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes

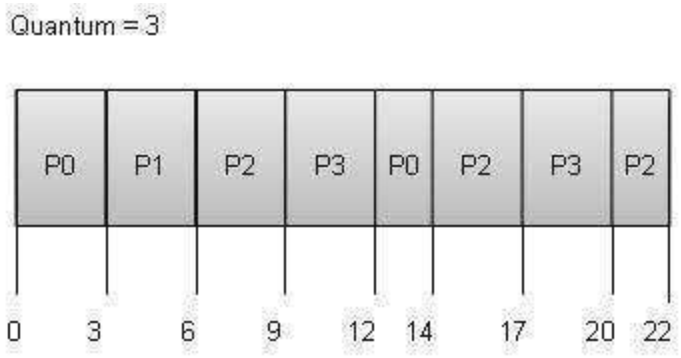


Figure 2.5 Round Robin Scheduling

Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

Multiple-level queues are dependent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics

- Multiple queues are maintained for processes with common characteristics
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue

For example, CPU-bound jobs can schedule in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

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 will be preempted.

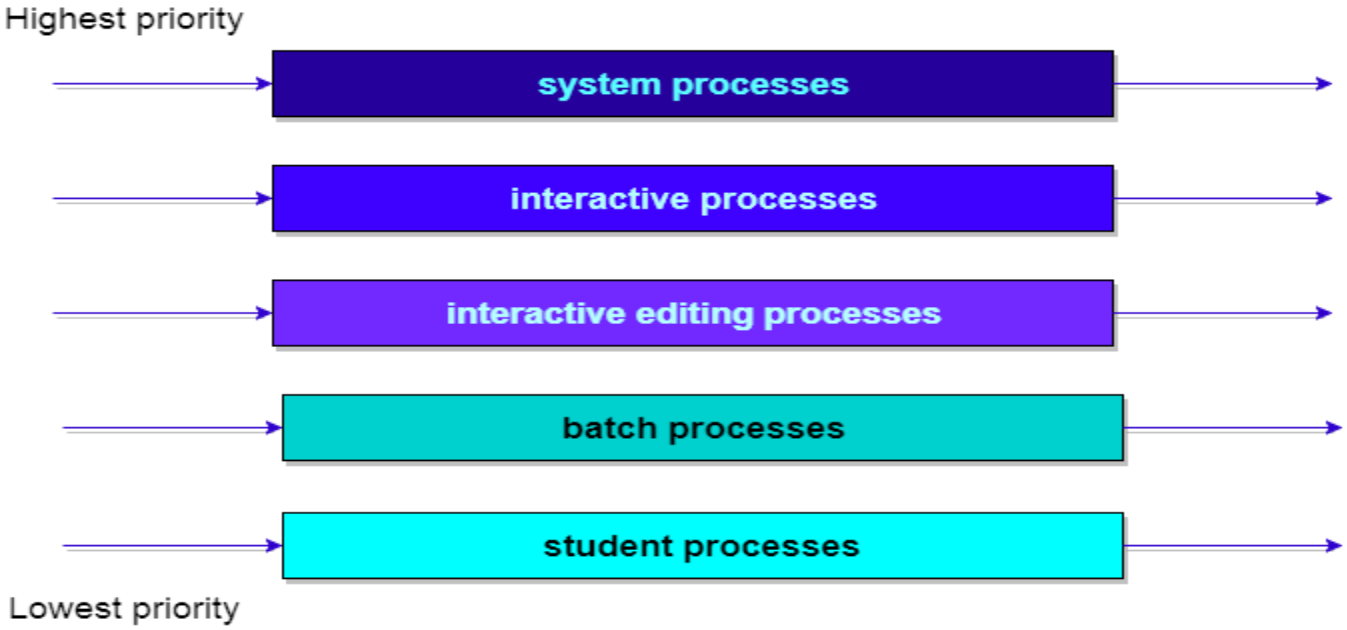


Figure 2.6 Multiple-Level Queues Scheduling

Multilevel Feedback Queue Scheduling

In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, 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.

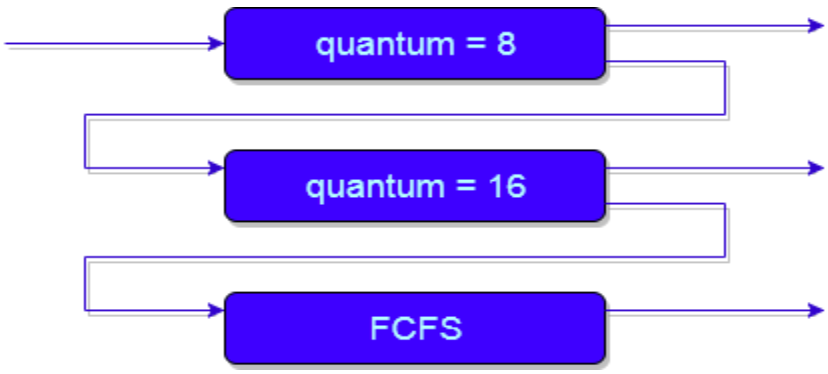


Figure 2.7 Multilevel Feedback Queue Scheduling

In general, 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.

Evaluation of Process Scheduling Algorithms

Deterministic Modeling: This evaluation method takes a predetermined workload and evaluates each algorithm using that workload.

Queuing Models

Another method of evaluating scheduling algorithms is to use queuing theory. Using data from real processes we can arrive at a probability distribution for the length of a burst time and the I/O times for a process. We can now generate these times with a certain distribution.

- We can also generate arrival times for processes (arrival time distribution).
- If we define a queue for the CPU and a queue for each I/O device we can test the various scheduling algorithms using queuing theory.
- Knowing the arrival rates and the service rates we can calculate various figures such as average queue length, average wait time, CPU utilization etc.

Simulations

Rather than using queuing models we simulate a computer. A Variable, representing a clock is incremented. At each increment the state of the simulation is updated.

- Statistics are gathered at each clock tick so that the system performance can be analysed.
- The data to drive the simulation can be generated in the same way as the queuing model, although this leads to similar problems.
- Alternatively, we can use trace data. This is data collected from real processes on real machines and is fed into the simulation. This can often provide good results and good comparisons over a range of scheduling algorithms.
- However, simulations can take a long time to run, can take a long time to implement and the trace data may be difficult to collect and require large amounts of storage.

Implementation

The best way to compare algorithms is to implement them on real machines. This will give the best results but does have a number of disadvantages.

- It is expensive as the algorithm has to be written and then implemented on real hardware.
- If typical workloads are to be monitored, the scheduling algorithm must be used in a live situation. Users may not be happy with an environment that is constantly changing.
- If we find a scheduling algorithm that performs well there is no guarantee that this state will continue if the workload or environment changes.

Multiple-Processor Scheduling

•CPU scheduling more complex when multiple CPUs are available-Most current general purpose processors are multiprocessors (i.e. multicore processors)

-No single 'best' solution to multiple-processor scheduling

• A multicore processor typically has two or more homogeneous processor cores.

-Because the cores are all the same, any available processor can be allocated to any process in the system

Approaches to Multiple-Processor Scheduling

- Asymmetric multiprocessing

-All scheduling decisions, I/O processing, and other system activities handled by a single processor-Only one processor accesses the system data structures, alleviating the need for data sharing

- Symmetric multiprocessing (SMP)

-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-Currently, most common approach to multiple-processor scheduling

Process Concept

Process: A process is a program in execution. The execution of a process must progress in a sequential fashion a process defines as an entity, which represents the basic unit of work implemented in the system.
To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process, which performs all the tasks mentioned in the program.
When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data. The following image shows a simplified layout of a process inside main memory –

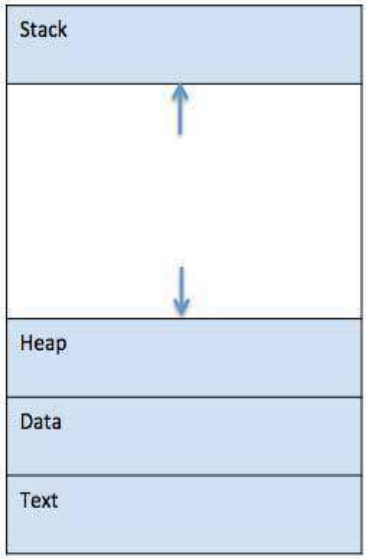


Figure 2.8 layout of a process inside main memory

Process State

S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created
2	**Ready** The process is waiting to assign to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory

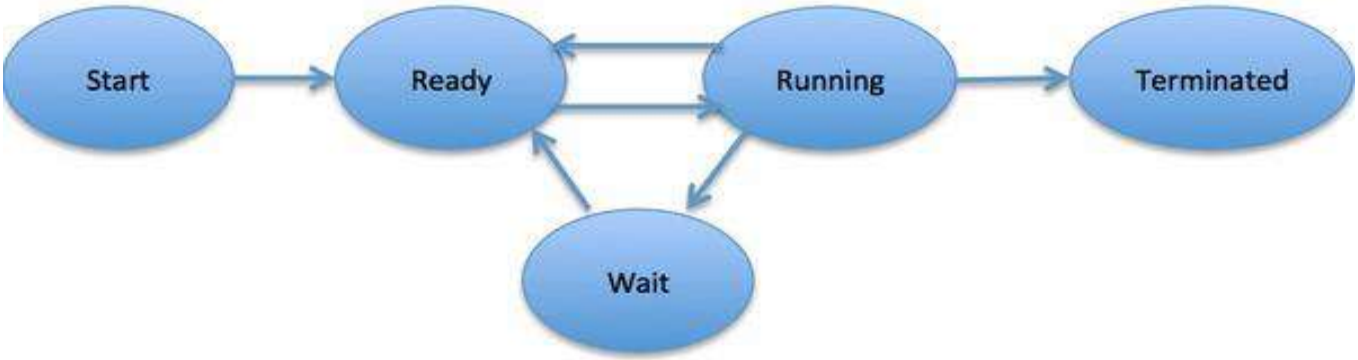


Figure 2.9 Process State Diagram

Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. An integer process ID (PID) identifies the PCB. A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system
4	**Pointer** A pointer to parent process
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process
8	**Memory management information** This includes the information of page table, memory limits, and Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −

Figure 2.10 PCB

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates

Schedulers

Schedulers are special system software, which handles process scheduling in various ways. Their main task is to select jobs submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

A long-term scheduler determines which programs admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. 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.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long-term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers are.

Medium Term Scheduler

Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspend if it makes an I/O request. Suspended processes cannot make any progress towards completion in this condition, to remove the process from memory and make space for

other process; the suspended process moved to the secondary storage. This process called swapping, and the process said swapped out or rolled out. Swapping may be necessary to improve the process mix

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long-term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can resume from the same point later. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

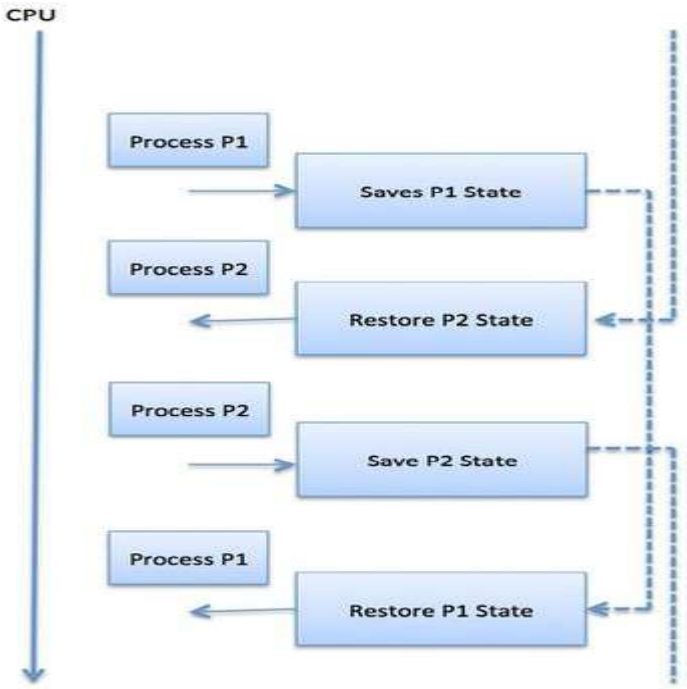


Figure 2.11 Context switches

Context switches are computationally intensive since register and memory state, some hardware systems employ two or more sets of processor registers. When the process switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

Operations on Processes:

There are many operations that can be performed on processes. Some of these are process creation, process preemption, process blocking, and process termination. These are given in detail as follows:

Process Creation

Processes need to be created in the system for different operations. This can be done by the following events:

- User request for process creation
- System initialization
- Execution of a process creation system call by a running process
- Batch job initialization

A process may be created by another process using fork(). The creating process is called the parent process and the created process is the child process. A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows:

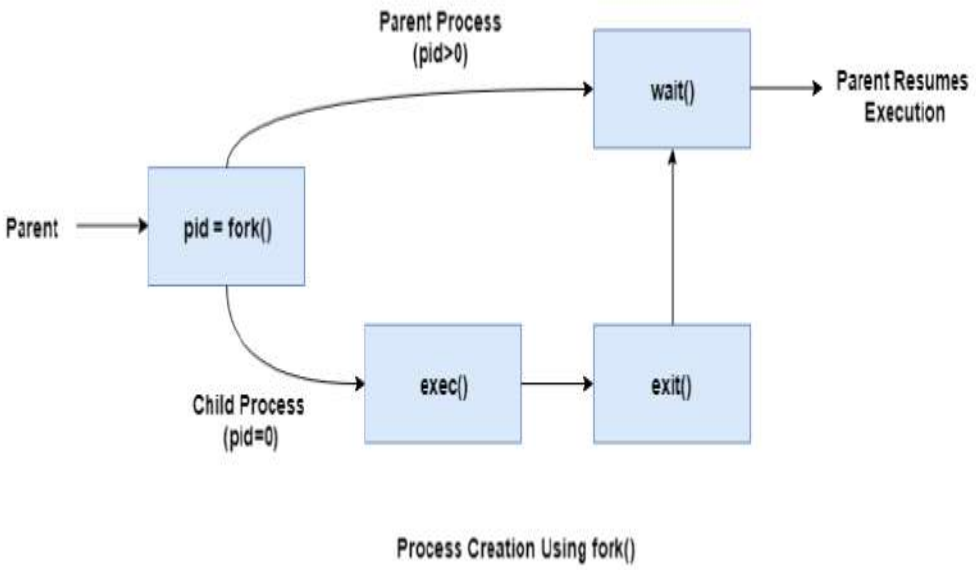


Figure 2.12 Process Creation

Process Preemption

An interrupt mechanism is used in preemption that suspends the process executing currently and the next process to execute is determined by the short-term scheduler. Preemption makes sure that all processes get some CPU time for execution.

A diagram that demonstrates process preemption is as follows:

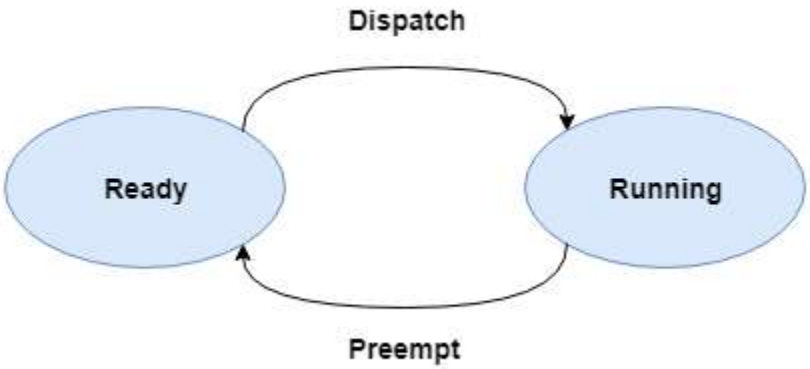


Figure 2.13 Process Preemption

Process Blocking

The process is blocked if it is waiting for some event to occur. This event may be I/O as the I/O events are executed in the main memory and don't require the processor. After the event is complete, the process again goes to the ready state.

A diagram that demonstrates process blocking is as follows:

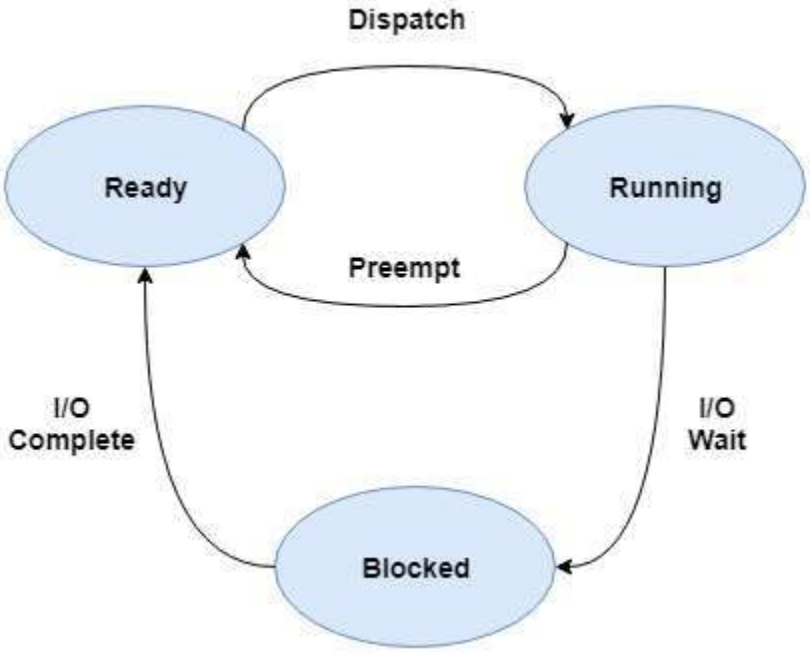


Figure 2.14 Process Blocking

Process Termination

After the process has completed the execution of its last instruction, it is terminated. The resources held by a process are released after it is terminated.

A child process can be terminated by its parent process if its task is no longer relevant. The child process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated.

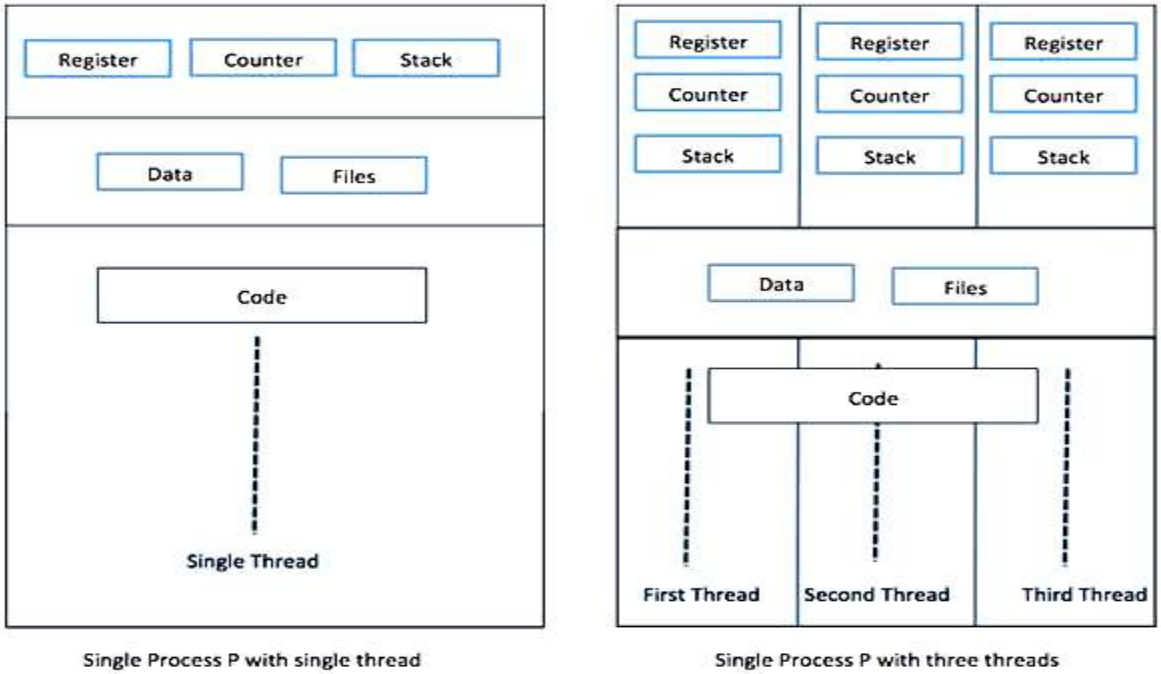
Thread

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack, which contains the execution history.

A thread shares with its peer threads little information like code segment, data segment and open files. When one thread alters a code segment memory items all other threads see that

A thread is also called a lightweight process Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads has been successfully used in implementing network servers and web server they also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread

Single Process P with three threads

Figure 2.15 Threads

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.

3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes, each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads is implemented in following two ways-

User Level Threads – User managed threads.

Kernel Level Threads – Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

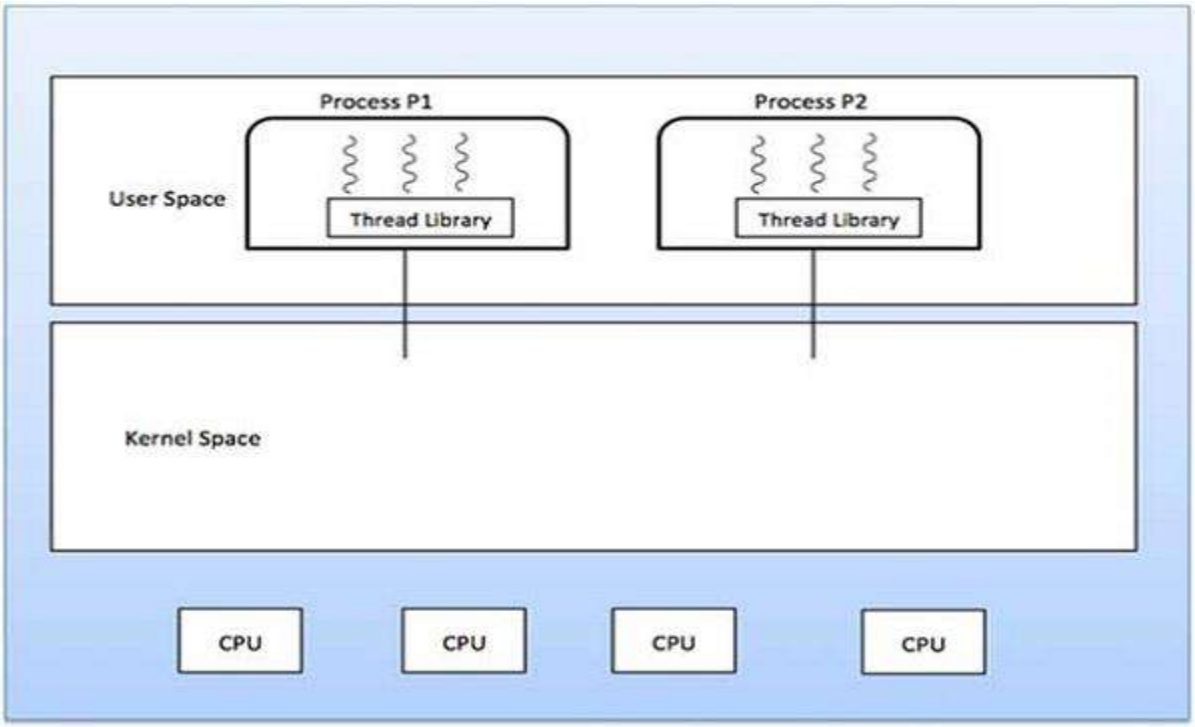


Figure 2.16 User level threads

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.
- Disadvantages
- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, the Kernel does thread management. There is no thread management code in the application area. Kernel threads supported directly by the operating system

Any application can be programmed to be multithreaded All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individual's threads within the process. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many-to-many relationship
- Many to one relationship
- One to one relationship

Many-to- Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

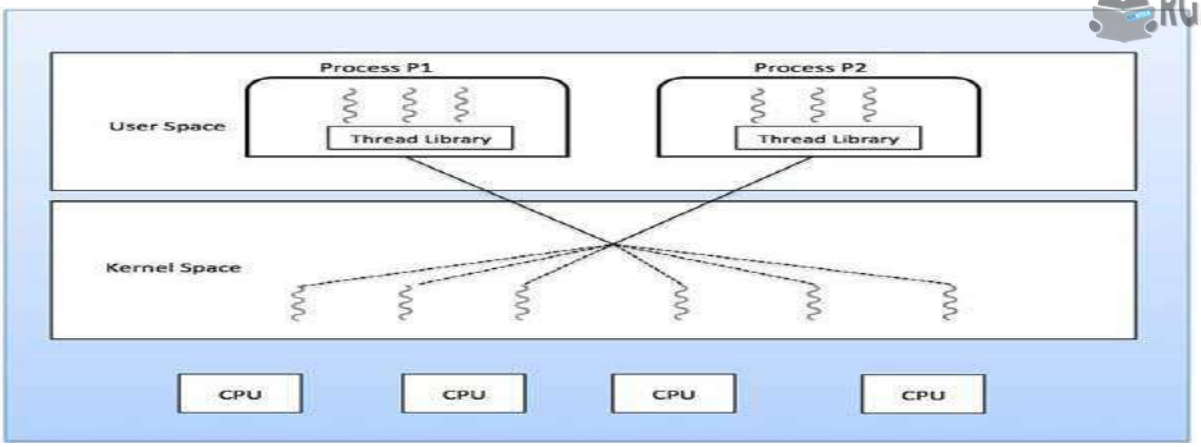


Figure 2.17 Many-to- Many Model

Many-to- One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management done in user space by the thread library. When thread makes a blocking system call, only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes

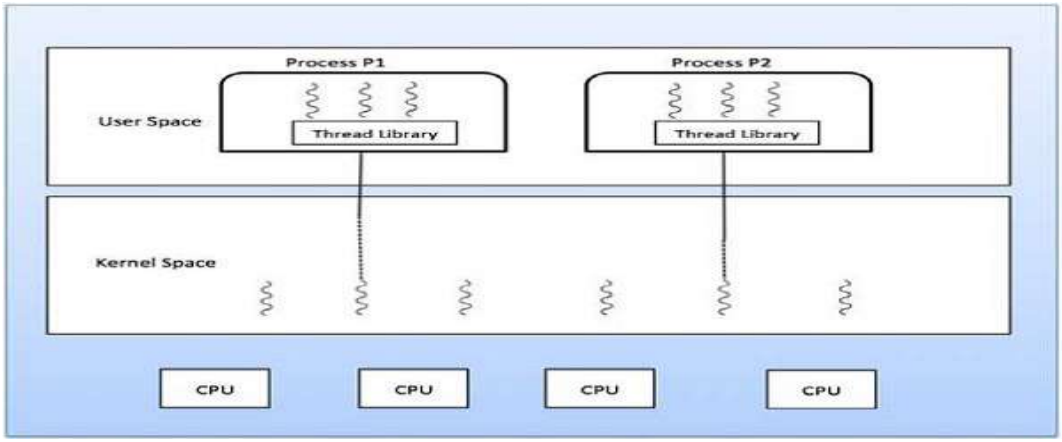


Figure 2.18 Many-to- One Model

One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

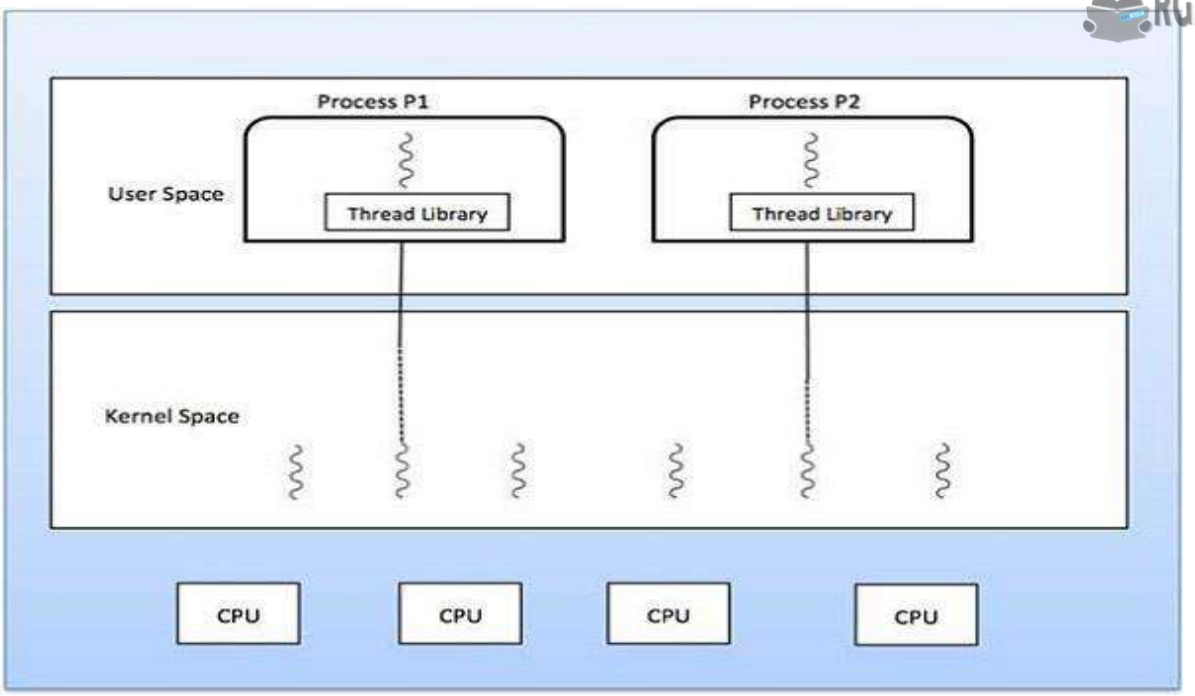


Figure 2.19 One to One Model

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: Execution of one process does not affect the execution of other processes.
- **Cooperative Process**: Execution of one process affects the execution of other processes.

Cooperating Processes

- Once we have multiple processes or threads, it is likely that two or more of them will want to communicate with each other
- Process cooperation (i.e., interprocess communication) deals with three main issues
  - Passing information between processes/threads
  - Making sure that processes/threads do not interfere with each other
  - Ensuring proper sequencing of dependent operations
- These issues apply to both processes and threads
  - Initially we concentrate on shared memory mechanisms

Cooperating Process Definition

- An independent process cannot affect or be affected by the execution of another process.
- A cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation

- Information sharing
- Computation speed-up
- Modularity
- Convenience

Issues for Cooperating Processes

- Race conditions
  - A race condition is a situation where the semantics of an operation on shared memory are affected by the arbitrary timing sequence of collaborating processes
- Critical regions
  - A critical region is a portion of a process that accesses shared memory
- Mutual exclusion
  - Mutual exclusion is a mechanism to enforce that only one process at a time is allowed into a critical region

Cooperating Processes Approach

- Any approach to process cooperation requires that
  - No two processes may be simultaneously inside their critical regions
  - No assumptions may be made about speeds or the number of CPUs
  - No process running outside of its critical region may block other processes
  - No process should have to wait forever to enter its critical region

Inter Process Communication

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

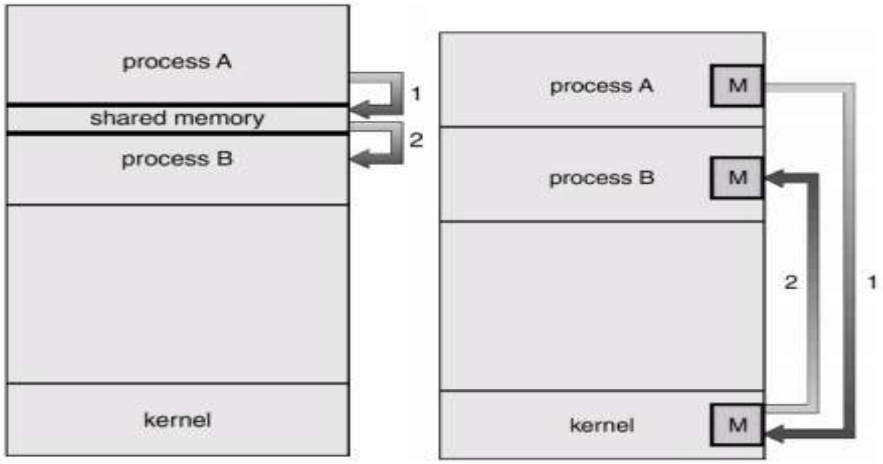


Figure 2.20 Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, and then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first checks for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

ii) Messaging Passing Method

Now, we will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
- We need at least two primitives:
  Send (message, destination) or **send** (message)
  Receive (message, host) or **receive** (message)

Figure 2.21 Message Passing method

The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.** The **header part** is used for storing Message type, destination id, source id, and message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so Message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that, for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes But the sender expect acknowledgement from receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution but at the same time, if the message send keep on failing, receiver will have to wait for indefinitely. That is why we also consider the other possibility of message passing. There are basically three most preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

Process Synchronization

Process Synchronization means sharing system resources by processes in a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Precedence Graph

A precedence graph is a directed graph acyclic graph where edge represents execution order and node represents individual statements of the program code.

For example: consider the following statements;
(S1): a = x + y;
(S2): b = a + z;
(S3): c = x - y;
(S4): w = b + c;

Precedence graph:

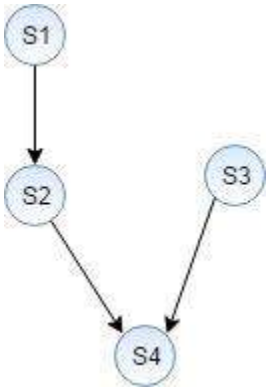
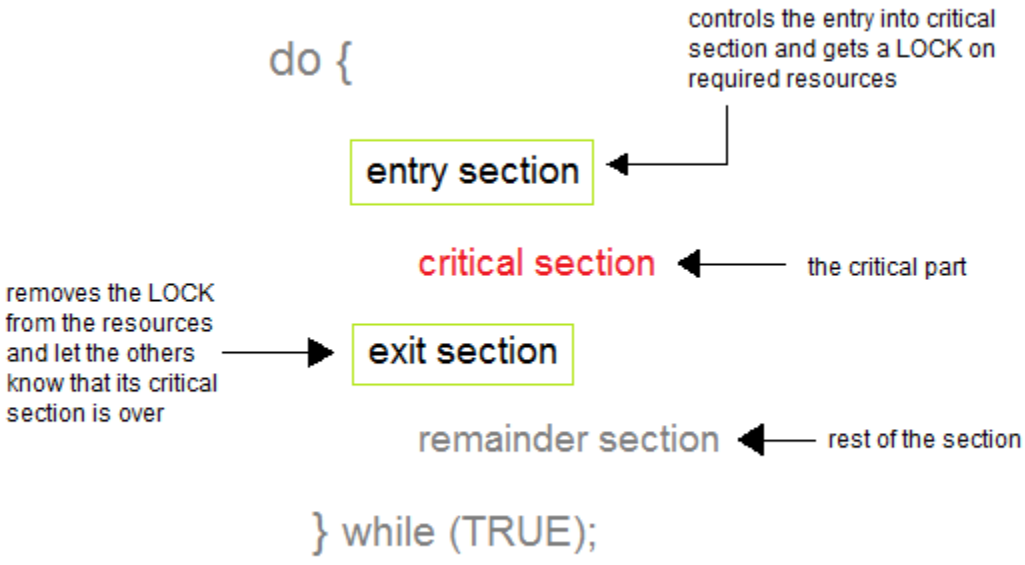


Figure 2.22 Example of Precedence graph:

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.



Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress
   If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
3. Bounded Waiting
   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.

The classical definition of wait and signal are :

- Wait: decrement the value of its argument S as soon as it would become non-negative.
- Signal: increment the value of its argument, S as an individual operation.

Properties of Semaphores

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types:

1. Binary Semaphore
   It is a special form of semaphore used for implementing mutual exclusion; hence it is often called Mutex. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.
2. Counting Semaphores
   These are used to implement bounded concurrency.

Limitations of Semaphores

1. Priority Inversion is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

Classical Problem of Synchronization

Following are some of the classical problem faced while process synchronization in systems where cooperating processes are present.

Bounded Buffer (or Producer and Consumer) Problem

We have two processes named as Producer and Consumer. There is finite size of buffer or circular queue with two pointers 'in' and 'out'.

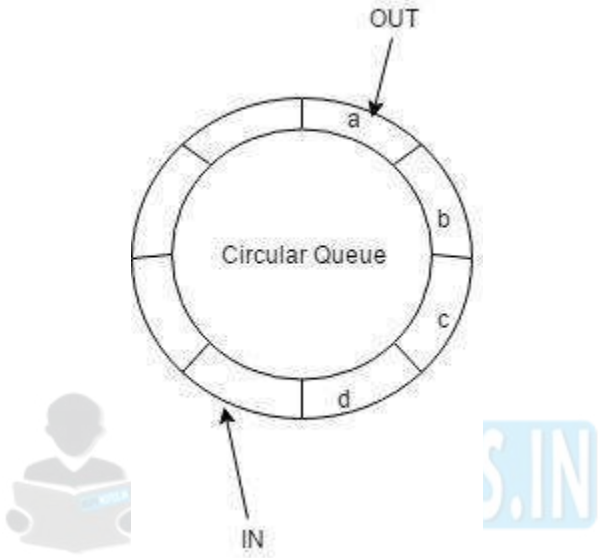


Figure 2.13 Bounded Buffers (or Producer and Consumer) Problem

Producer process produces items and fills into buffer using 'in' pointer and consumer process consumes process from the buffer using 'out' pointer. We must maintain count for empty and full buffers. Also, there should be synchronization between producing items and consuming items otherwise processes go to sleep an item is available to consume or a slot is available to put item.

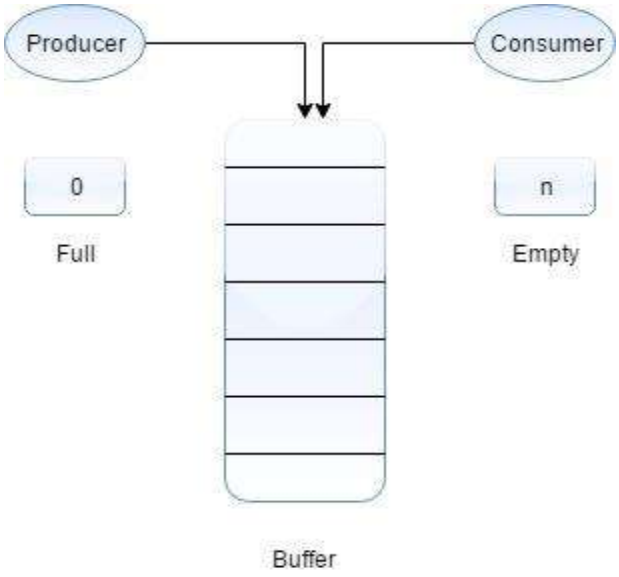


Figure 2.14 Bounded Buffers (or Producer and Consumer) Problem

Producer consumer problem can be solved using semaphores or monitors. Here we see solution using semaphores. There are three semaphores: full, empty, and mutex. 'Full' counts number of full slots, 'empty' counts number of empty slots, and 'mutex' enforces mutual exclusion condition.

Shared Data:
Semaphore mutex = 1 ; /* for mutual exclusion */
Semaphore full = 0 ; /* counts number of full slots, initially none */
Semaphore empty = n ; /* counts number of empty slots, initially all */

Producer Process:

```
Producer ()
{
While (true)
{
Produce_item (item) ; /* produce new item */
P (empty) ; /* decrease number of empty slots */
P (mutex) ; /* enter into critical section */
Add_item (item) ; /* item added in the buffer */
V (mutex) ; /* end the critical section */
V (full) ; /* increase number of full slots */
}
}
```

Consumer Process:

```
Consumer ()
{
while()
{
P (full) ; /* decrease number of full slots */
P (mutex) ; /* enter into critical section */
Remove_item () ; /* item removed from the buffer */
V (mutex) ; /* end the critical section */
V (empty) ; /* increase the number of empty slots */
}
}
```

Mutual exclusion is enforced by 'mutex' semaphore. Synchronization is enforced using 'full' and 'empty' semaphores that avoid race around condition and deadlock in the system.
If we do not use these three semaphores in the producer and consumer problem then there will be race around, deadlock situation and loss of data.

Readers Writers Problem
If we have shared memory or resources, then readers - writers can create conflict due to these combinations: writer - writer and writer- reader access critical section simultaneously that create synchronization problem, loss of data etc. We use semaphores to avoid these problems in reader writer problem.

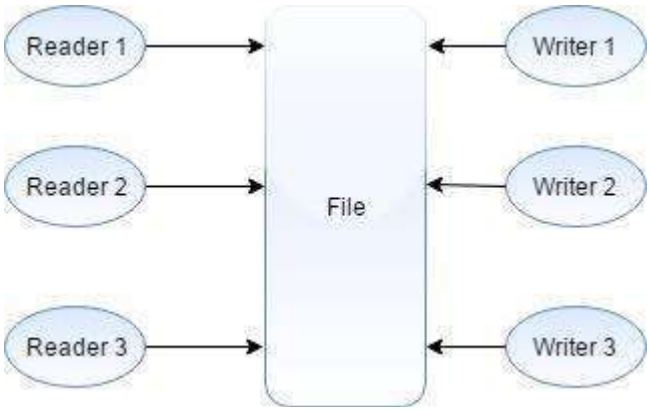


Figure 2.15 Readers Writers Problem

We follow some concepts using semaphore to solve readers - writer's problem: reader and writer cannot enter into critical section at same time. Multiple readers can access critical section simultaneously but multiple writer cannot access.

Shared Data:
Semaphore mutex = 1 ;
Semaphore wsem = 1 ;
intreadcount = 0 ;

Reader Process:

```
Reader ()
{
While (true)
{
Wait (mutex) ;
Readcount ++ ;
If (readcount = = 1)
Wait (wsem) ;
Signal (mutex) ;
// Critical Section () ;
Wait (mutex) ;
Readcount - - ;
If (readcount = = 0)
Signal (wsem) ;
Signal (mutex) ;
}
}
```

Writer Process:

```
Writer ()
{
While (true)
{
Wait (wsem) ;
// Critical Section () ;
Signal (wsem) ;
}
}
```

Semaphore 'mutex' ensures mutual exclusion property. Readcount and wsem ensures that there is no conflict in the critical section problem. These avoids race around, loss of data, and synchronization problem.

Dining Philosophers Problem

There are n numbers of philosophers meeting around a table, eating spaghetti and talking about philosophy. There are only n forks are available and each philosopher needs 2 forks to eat. Only one fork is available between each philosopher. Now we have design algorithm that ensures maximum number of philosophers can eat at once and none starves as long as each philosopher eventually stop eating.

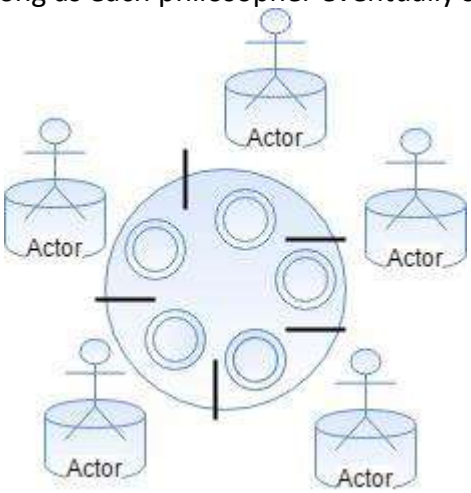


Figure 2.16 Dining Philosophers Problem

```
N = 5 ;  /* total number of philosophers */
Right (i) = (i + 1) mod n ;
Left (i) = ((i = = n) ? 0 : (i + 1))
Philosopher_state[] = {thinking, hungry, eating}
```

```
Semaphore mutex = 1 ;
Semaphore S[n] ;  /*one per philosopher, all 0 initially*/
Philosopher (int process)
{
while(true)
{
Think () ;
Get_forks (process) ;
Eat () ;
Put_forks (process) ;
}
}
Test (int i)
{
If (state[i] = hungry)&&(state[left(i)] != eating)&&(state[right(i) != eating])
{
State[i] = eating ;
V(S[i]) ;
}
}
Get_forks(int i)
{
P(mutex) ;
State[i] = hungry ;
Test [i] ;
V (mutex) ;
P(S[i])
}
Put_forks(int i)
{
P(mutex) ;
State [i] = thinking ;
Test (left(i)) ;
Test (right(i)) ;
V (mutex) ;
}
```

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in