

UNIT-II
VTS

OS II-UNIT

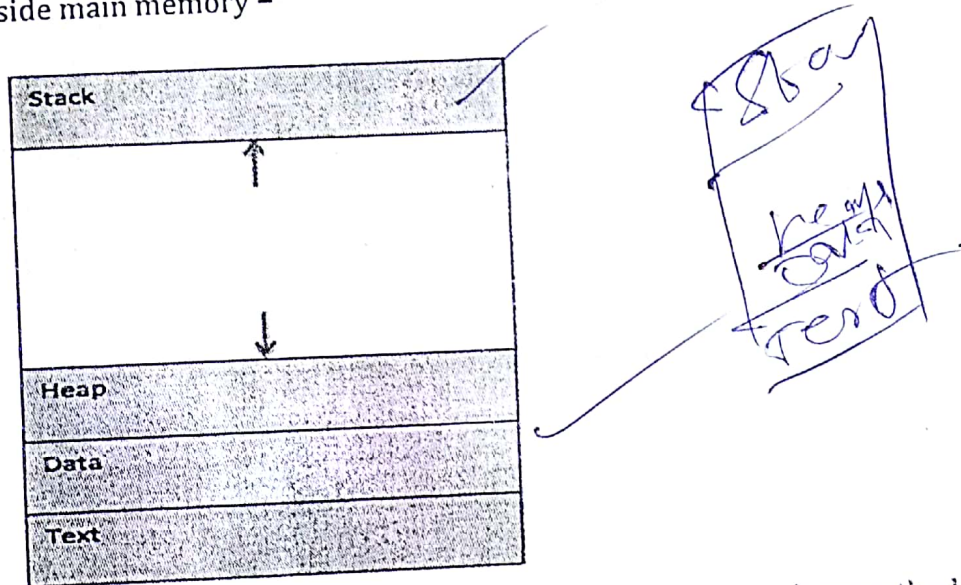
1) Process

"A process is basically a program in execution. The execution of a process must progress in a sequential fashion."

A process is defined as an entity which represents the basic unit of work to be 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 —



Stack: The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap: This is dynamically allocated memory to a process during its run time.

Text: This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

Data: This section contains the global and static variables.

Program

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language —

OS II-UNIT

```
#include <stdio.h>
```

```
int main() {  
    printf("Hello, World! \n");  
    return 0;  
}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

2) Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

Start : This is the initial state when a process is first started/created.

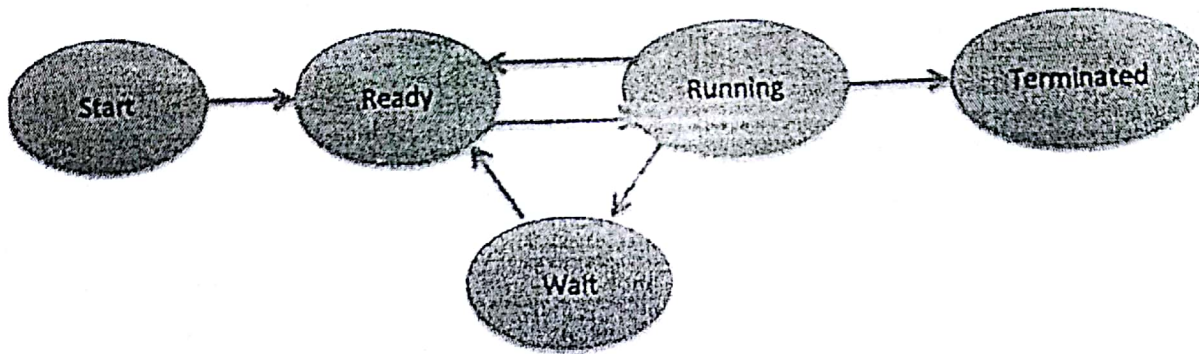
Ready : The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.

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.

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.

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.

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.



3) Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below :

Process State: The current state of the process i.e., whether it is ready, running, waiting, or whatever.

Process privileges : This is required to allow/disallow access to system resources.

Process ID: Unique identification for each of the process in the operating system.

Pointer : A pointer to parent process.

Program Counter : Program Counter is a pointer to the address of the next instruction to be executed for this process.

CPU registers: Various CPU registers where process need to be stored for execution for running state.

CPU Scheduling Information : Process priority and other scheduling information which is required to schedule the process.

Memory management information: This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

Accounting information: This includes the amount of CPU used for process execution, time limits, execution ID etc.

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 –

OS II-UNIT

Process ID
State
Pointer
Priority
Program counter
CPU registers
I/O information
Accounting information
etc...

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

4) OPERATING SYSTEM - PROCESS SCHEDULING

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

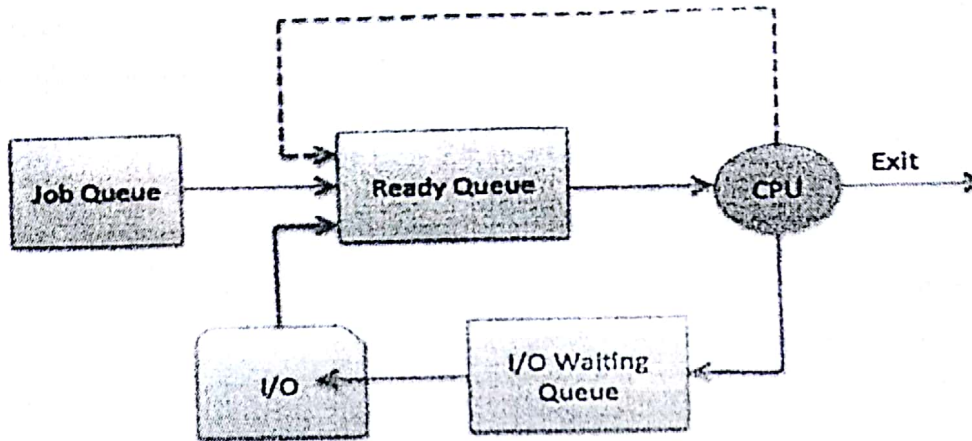
The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues –

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

OS II-UNIT

- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below –

S.N.	State & Description
1	Running When a new process is created, it enters into the system as in the running state.
2	Not Running Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

5)Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be 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

It is also called a **job scheduler**. A long-term scheduler determines which programs are 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

It is also called as **CPU 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.

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.

OS II-UNIT

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be 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 and execution can be continued.

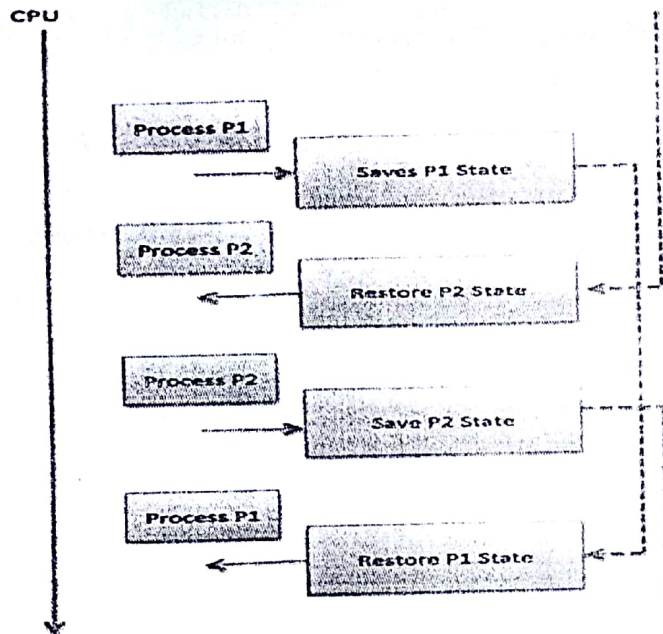
6)Context Switch

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 be resumed from the same point at a later time. 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

OS II-UNIT

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.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

Operating System - Multi-Threading

What is 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.

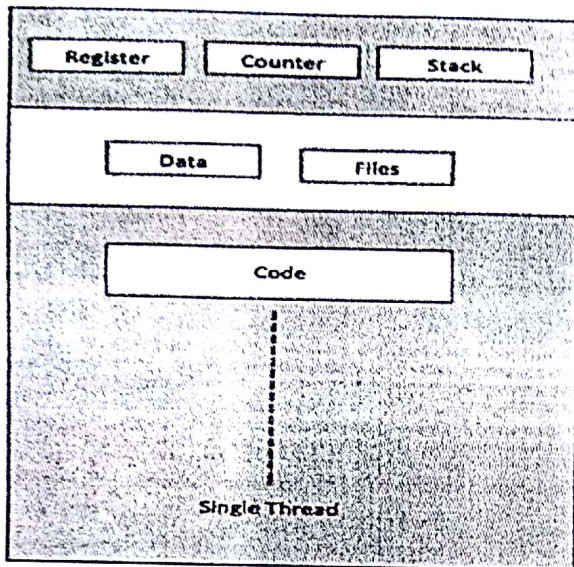
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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.

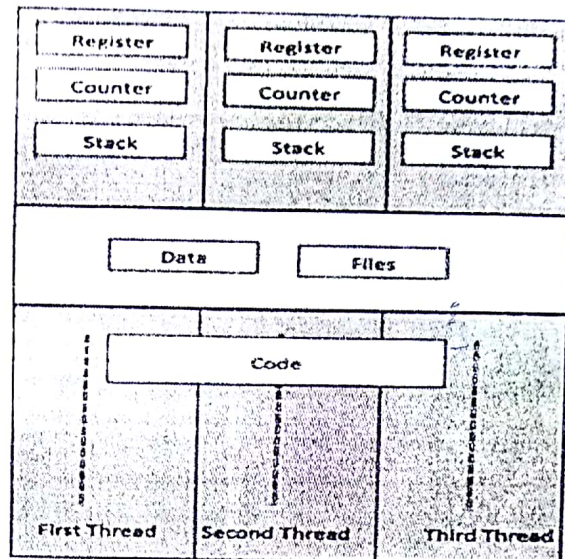
Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing

OS II-UNIT

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

6) 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	Multiple threaded processes use fewer

OS II-UNIT

more resources.

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.

7) User Level thread Vs Kernel Level thread

USER LEVEL THREAD

User thread are implemented by users.

OS doesn't recognized user level threads.

Implementation of User threads is easy.

Context switch time is less.

Context switch requires no hardware support.

If one user level thread perform blocking operation then entire process will be blocked.

Example : Java thread, POSIX threads.

KERNEL LEVEL THREAD

kernel threads are implemented by OS.

Kernel threads are recognized by OS.

Implementation of Kernel thread is complicated.

Context switch time is more.

Hardware support is needed.

If one kernel thread perform blocking operation then another thread can continue execution.

Example : Window Solaris.

8) Multithreading Models

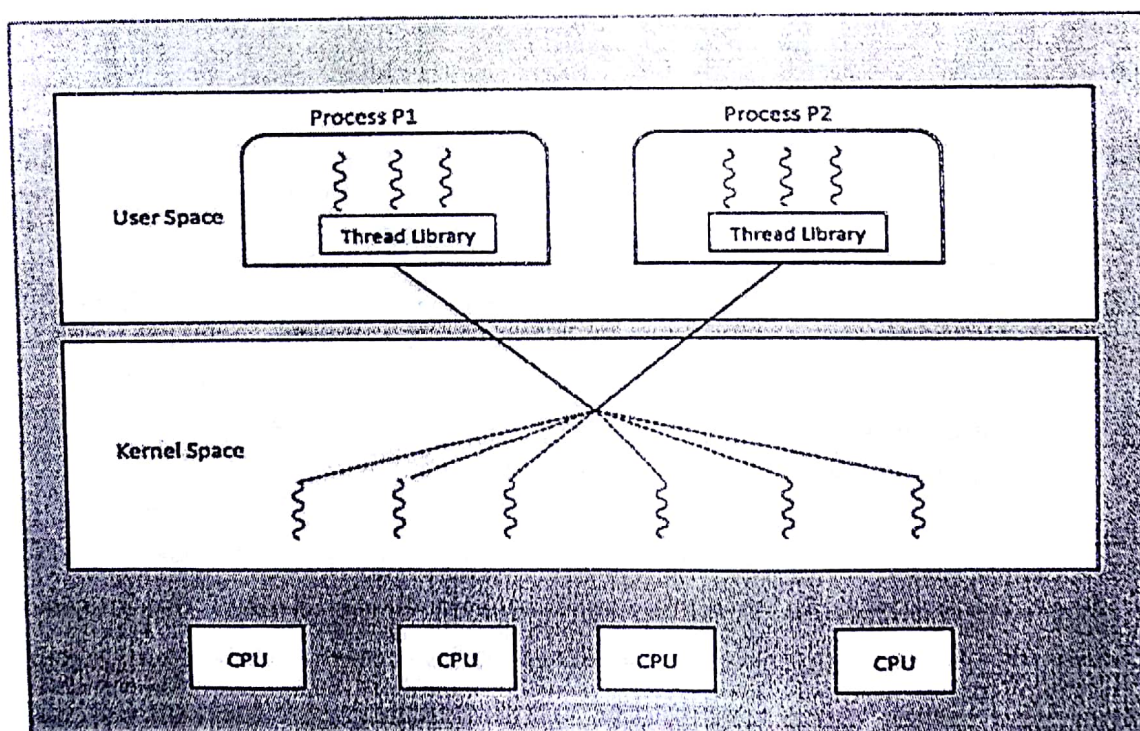
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

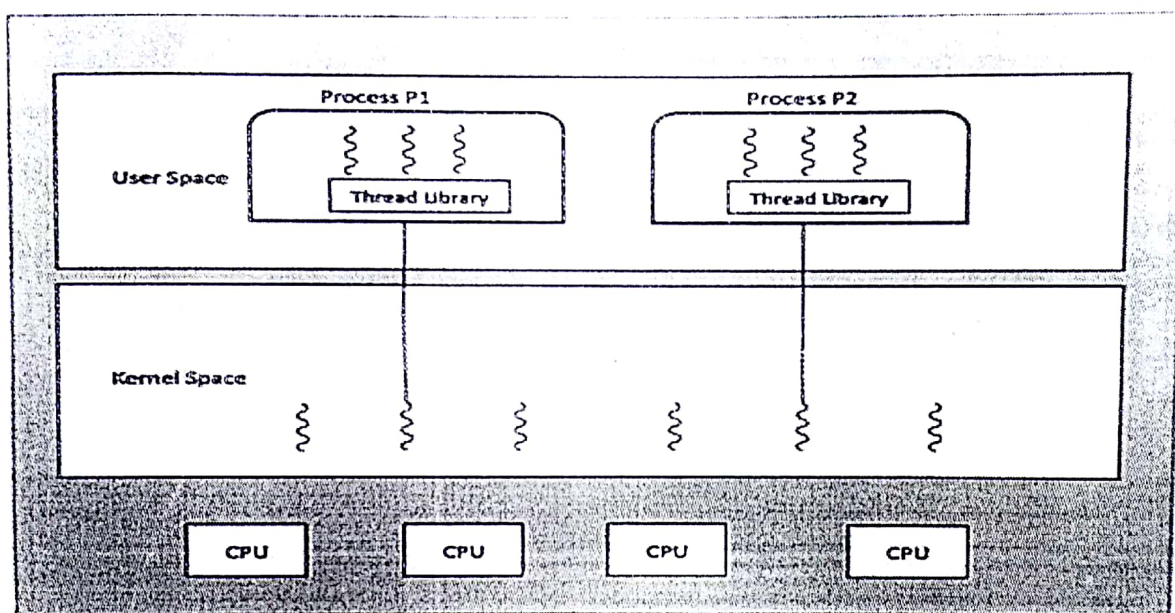


OS II-UNIT

Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. 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 are 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.

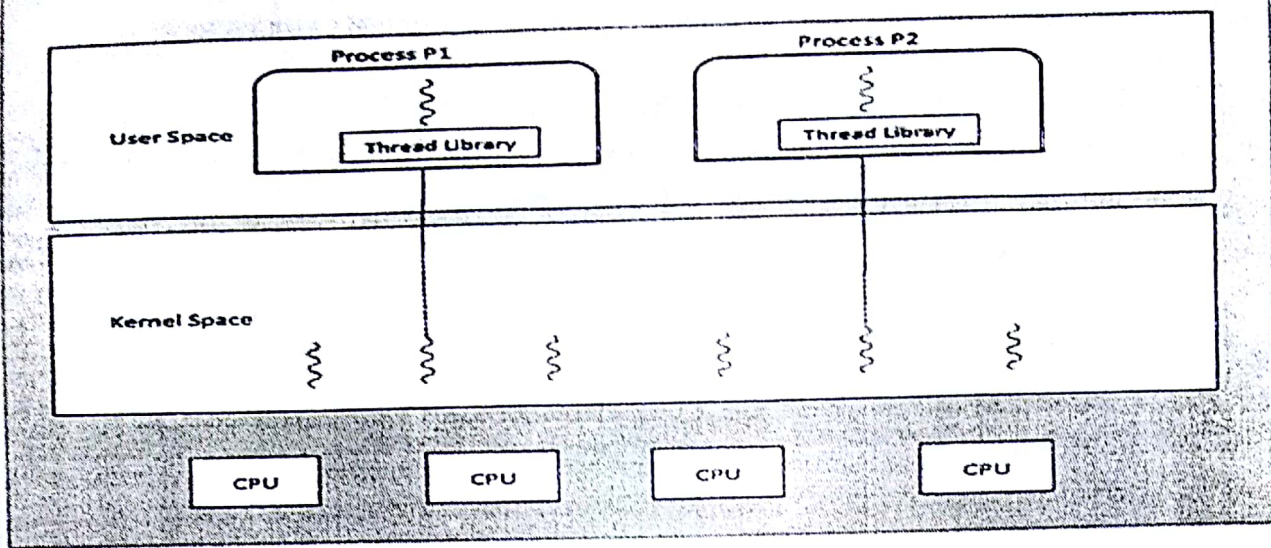


One to One Model

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.

OS II-UNIT



9) THE ADVANTAGES OF MULTITHREADED PROGRAMMING:

1. Responsiveness:

Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

2. Resource sharing:

By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. Economy:

Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

4. Utilization of multiprocessor architectures:

The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only

OS II-UNIT

run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency. ✓.

10) OPERATING SYSTEM SCHEDULING ALGORITHMS

A Process Scheduler schedules different processes 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

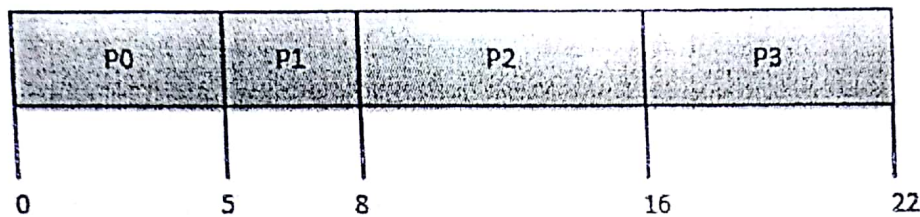
These algorithms are either **non-preemptive** or **preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a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

OS II-UNIT

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



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 Next (SJN)

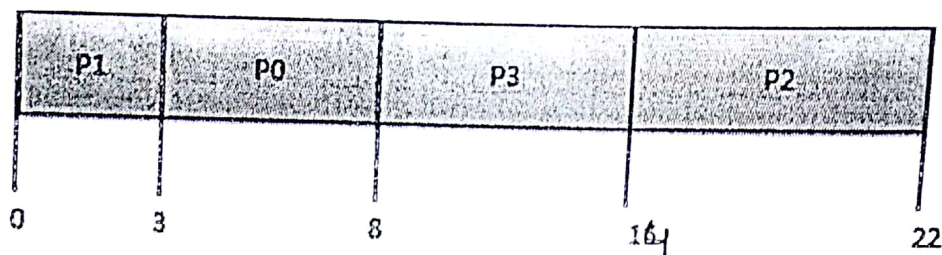
- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required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

$$W.T = T_{at} - b \cdot 1 - n$$

OS II-UNIT

- The processor 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



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

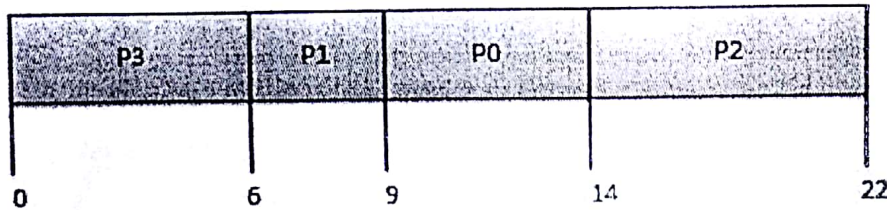
- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.

OS II-UNIT

- 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

at a point of time
comes into running
state



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.

17

Process	A.T	B.T
P0	0	5
P1	1	3
P2	2	8
P3	3	6

P_0	P_1	P_0	P_3	P_2
0	1 3 4	8	14	22

$P_0 \text{ w.t} = 8 - 5 - 0 = 3$
 $P_1 \text{ w.t} = 4 - 3 - 1 = 0$
 $P_2 \text{ w.t} = 22 - 8 - 2 = 12$
 $P_3 \text{ w.t} = 8 - 6 - 3 = -1$

$\text{avg w.t} = \frac{3 + 0 + 12 + 5}{4}$
 $= \frac{20}{4} = 5$

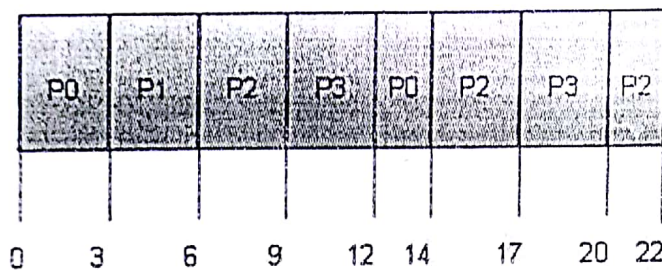
OS II-UNIT

- 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.

Quantum = 3



Wait time of each process is as follows -

$$W.T = T_{at} - b.T -$$

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 not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

OS II-UNIT

- 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 be scheduled 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.

11) about Inter Process Communication

A process can be of two type:

- 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:

12) Inter Process Communication models:

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.

OS UNIT

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. Let's discuss an example of communication between processes using shared memory method.

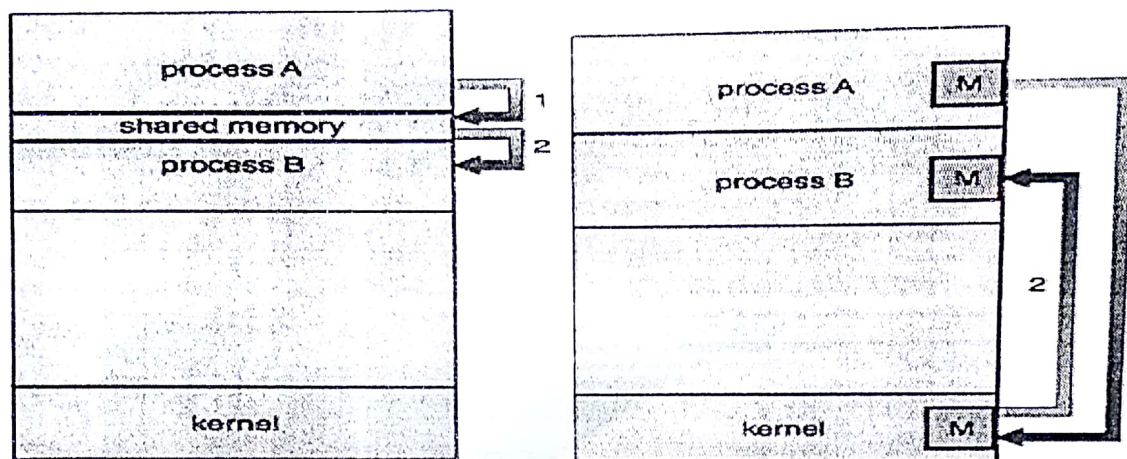


Figure 1: Shared Memory and Message Passing (Image Taken from "Operating System Concepts" by Galvin et al.)

Message-Passing Systems:

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment. A message-passing facility provides at least two operations: `send(message)` and `receive(message)`. Messages sent by a process can be of either fixed or variable size.

➤ Direct or indirect communication

Direct message passing, The process which want to communicate must explicitly name the recipient or sender of communication. e.g. `send(p1, message)` means send the message to p1. similarly, `receive(p2, message)` means receive the message from p2. In this method of communication, the communication link get established automatically, Symmetry and asymmetry between the sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages

OS II-UNIT

or only sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message.

Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. The standard primitives used are : **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**.

Synchronous and Asynchronous Message Passing:
Synchronization Communication between processes takes place through calls to **send()** and **receive ()** primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

- **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send.** The sending process sends the message and resumes operation.
- **Blocking receive.** The receiver blocks until a message is available.
- **Nonblocking receive.** The receiver retrieves either a valid message or a null.

Different combinations of **send()** and **receive ()** are possible. When both **send()** and **receive()** are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking **send()** and **receive ()** statements. The producer merely invokes the blocking **send()** call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes **receive ()**, it blocks until a message is available.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length n ; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender

OS II-UNIT

can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

13) STARVATION;

It is a situation where a process does not get the resources it needs for a long time because the resources are being allocated to other processes. It generally occurs in a Priority based scheduling System. Where High Priority requests get processed first. Thus a request with least priority may never be processed. Priority scheduling leads to starvation.

14) DIFFERENCE BETWEEN PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING IN OS

Preemptive Scheduling

Processor can be preempted to execute a different process in the middle of execution of any current process.

CPU utilization is more compared to Non-Preemptive Scheduling.

Waiting time and Response time is less.

The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

If a high priority process frequently arrives in the ready queue, low priority process may starve.

Preemptive scheduling is flexible.

Ex:- SRTF, Priority, Round Robin, etc.

Non-Preemptive Scheduling

Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.

CPU utilization is less compared to Preemptive Scheduling.

Waiting time and Response time is more.

When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

If a process with long burst time is running CPU, then another process with less CPU burst time may starve.

Non-preemptive scheduling is rigid.

Ex:- FCFS, SJF, Priority, etc.