



**AKSHAYA INSTITUTE OF TECHNOLOGY**  
Approved by AICTE, New Delhi, Affiliated to VTU, Belagaavi, Recognised by GOK, NBA Accredited(CSE)  
Obalapura Post, Lingapura, Koratagere Road, Tumkur- 572 106, Karnataka



## **Department of Computer Science & Engineering**

Module 2 Notes

**Subject: OPERATING SYSTEMS(BCS303)**

Prepared by:

**Mr. Karthik H D**

**Associate Professor,**

**Dept. of CSE**



[www.ait-tumkur.ac.in](http://www.ait-tumkur.ac.in)

# AKSHAYA INSTITUTE OF TECHNOLOGY

Lingapura, Obalapura Post, Koratagere Road, Tumakuru - 572106

## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING



### VISION

To empower the students to be technically competent, innovative and self-motivated with human values and contribute significantly towards betterment of society and to respond swiftly to the challenges of the changing world.



### MISSION

**M1:** To achieve academic excellence by imparting in-depth and competitive knowledge to the students through effective teaching pedagogies and hands on experience on cutting edge technologies.

**M2:** To collaborate with industry and academia for achieving quality technical education and knowledge transfer through active participation of all the stakeholders.

**M3:** To prepare students to be life-long learners and to upgrade their skills through Centre of Excellence in the thrust areas of Computer Science and Engineering.



### Program Specific Outcomes (PSOs)

*After Successful Completion of Computer Science and Engineering Program Students will be able to*

- \* Apply fundamental knowledge for professional software development as well as to acquire new skills.
- \* Implement disciplinary knowledge in problem solving, analyzing and decision-making abilities through different domains like database management, networking, algorithms, and programming as well as research and development.
- \* Make use of modern computer tools for creating innovative career paths, to become an entrepreneur or desire for higher studies.



### Program Educational Objectives (PEOs)

**PEO1:** Graduates expose strong skills and abilities to work in industries and research organizations.

**PEO3:** Graduates engage in team work to function as responsible professional with good ethical behavior and leadership skills.

**PEO3:** Graduates engage in life-long learning and innovations in multi disciplinary areas.



Process Management

The CPU can execute only one instruction and that instruction can belong to only one ~~instruction~~ of the programs residing in the memory. Therefore, the OS have to allocate the CPU time to various users based on a certain policy. This is done by the Process Management (PM) module.

Process - Process is a program under execution. It is an active entity and a unit of work. It has a limited time span. Process consists of program code (text section), system stack (parameters, return addresses, local variables), data section (global variables) and process control block (PCB) (process state, program counter, PID, registers).

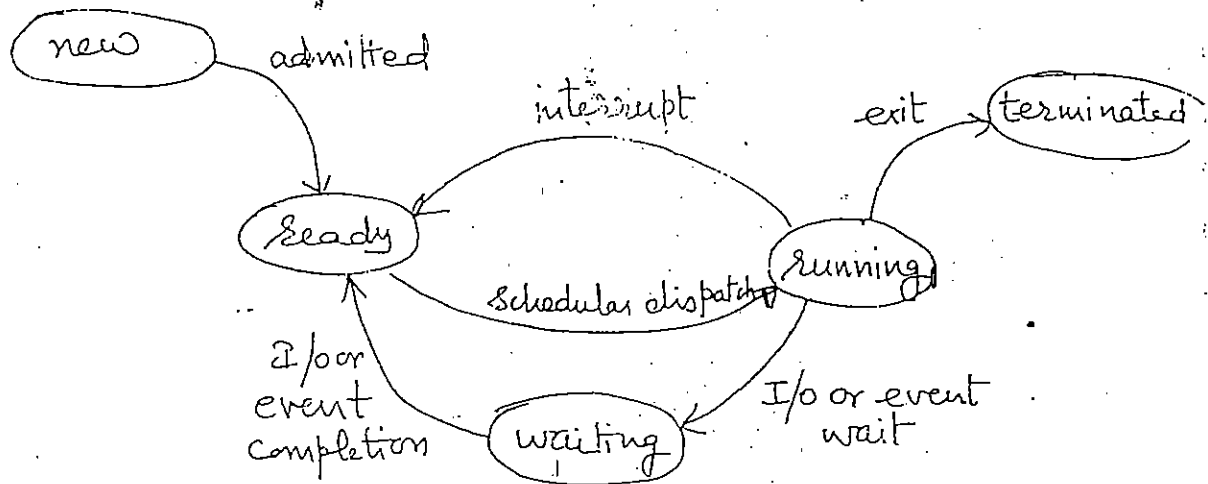
A program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources.

Process State

As a process executes, it changes its state. The state of a process is defined by the current activity of that process. Each 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 signal).

- Ready: The process is waiting to be assigned to processor.
- Terminated: The process has finished execution.



### Different process states

These states may vary across operating systems. Only one process can be running on any processor at any instant, although many processes may be ready and waiting.

Process Control Block

An OS considers a process to be the fundamental unit for resource allocation such as memory, swap space, IO Device, files owned by the process, CPU time etc

Whenever a process is created, a process control block (PCB), is created for the process by OS. This is a data structure that is used by an OS to keep track of all information concerning a process. The PCB of a process contains the following information:

Pointer	Process State
Process Identification Number	
Program Counter	
registers	
memory limits	
list of open files	

Process Control block (PCB)  
Fields in a PCB :-

- 1) Pointer : It is a pointer to the next PCB in the process scheduling list.
- 2) Process State : The state of process may be new, ready, running, waiting and so on.

3) Program Counter: The counter indicates the address of the next instruction to be executed for this process.

4) Process identification number: This is the unique number allocated by the OS to the process on creation.

5) Registers: The CPU registers include accumulator, index registers, stack pointers, and general purpose registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

6) Memory limits: Memory addresses in virtual memory.

7) List of open files: The files which were being used by the process.

8) Other information may be:

• CPU scheduling info: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• Memory management info: This information includes the values of the base and limit registers, page table or segment tables depending on the memory system used by the OS.

• Accounting info: This information includes the amount of CPU ~~exec~~ time used, job or process numbers etc.

• I/O status info: It includes the list of I/O devices allocated to this process.

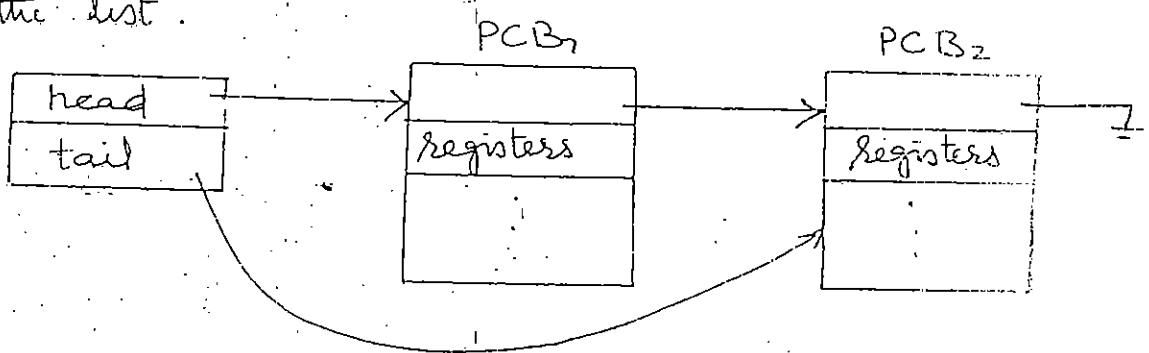
## Process Scheduling ✓

(27)

P. Scheduling refers to a set of policies and mechanisms built into the OS that governs the ~~operat~~ order in which processes should be submitted to the CPU for execution.

Scheduling Queues: As a process enters the system, it is put into a job queue. This queue consists of processes in the system.

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. Queue is stored as a link list. Header contains pointers to the first ~~the~~ first and final PCBs in the list.



### Ready Queue

The operating system also has other queues such as device queue - list of processes waiting for I/O device. Each device has its own device queue.

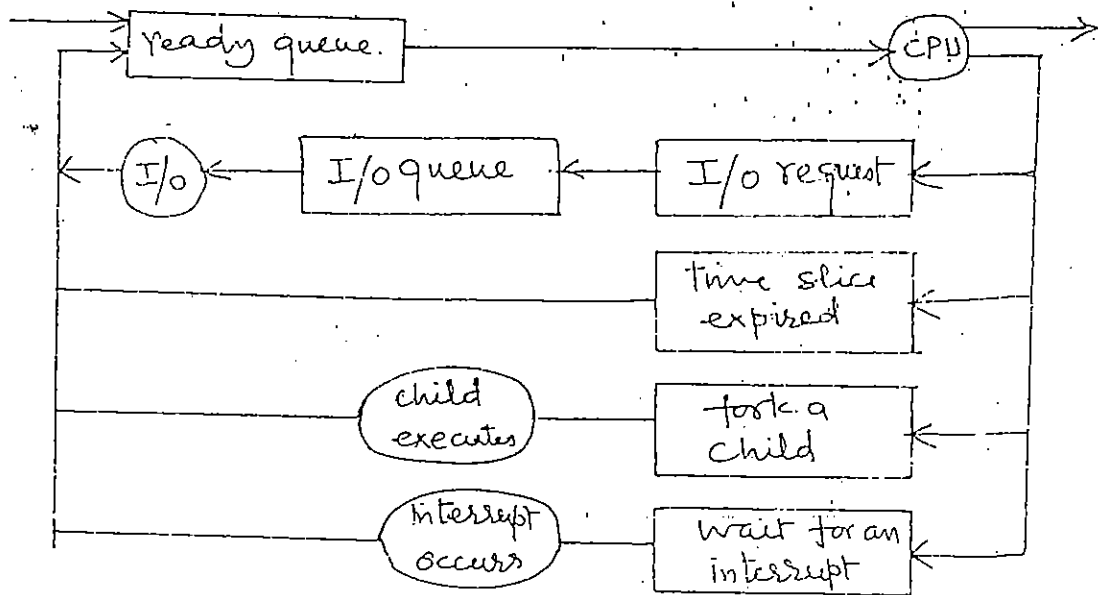
Process Scheduling can be represented by queuing diagram. Two types of queues are there: ready queue and ~~dev~~ device queue.

• A new process is initially put in ready queue. It waits in the ready queue until it is selected.

for execution. Once the process is assigned to 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 <sup>sub-</sup>process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be 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.



## Process Scheduling - Queuing representation

(22)

Schedulers: The operating system must select the processes from different queues for execution at that time or over the time. This selection process is carried out by the appropriate scheduler.

Long term scheduler: The function of the long term scheduler is to select the jobs from the pool of jobs and load them into main memory in ready queue. Thus, long term scheduler is also known as job scheduler.

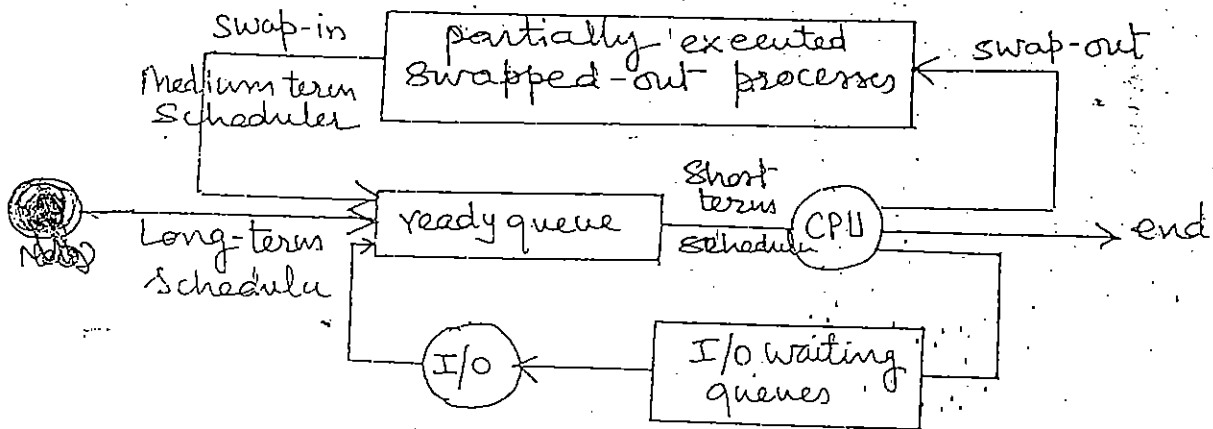
The long term scheduler must make a careful selection. Processes can be described either I/O bound or CPU bound. An I/O bound process spends more of its time doing I/O bound process tasks and CPU bound processes generate I/O requests infrequently, using most of the time CPU. Long term scheduler should select a good process mix of I/O & CPU bound processes. If all are I/O bound, the ready queue will always be empty. If all are CPU bound, I/O device queue will be empty and system will be imbalanced.

The long term scheduler executes much less frequently.

Short term scheduler: It selects the process (from main memory) which among the processes which are ready to execute and allocates the CPU to that process. It is also known as CPU scheduler. Short term scheduler executes at least once every 100 milliseconds.

It is also called as dispatcher.

Medium term Scheduler: Some OS, such as time sharing systems, may introduce an additional intermediate level of scheduling. It removes processes from memory. At some later time the process can be reintroduced into memory and its execution can be continued where it left off. This process is called swapping which is necessary to improve the process mix and to free up some main memory.



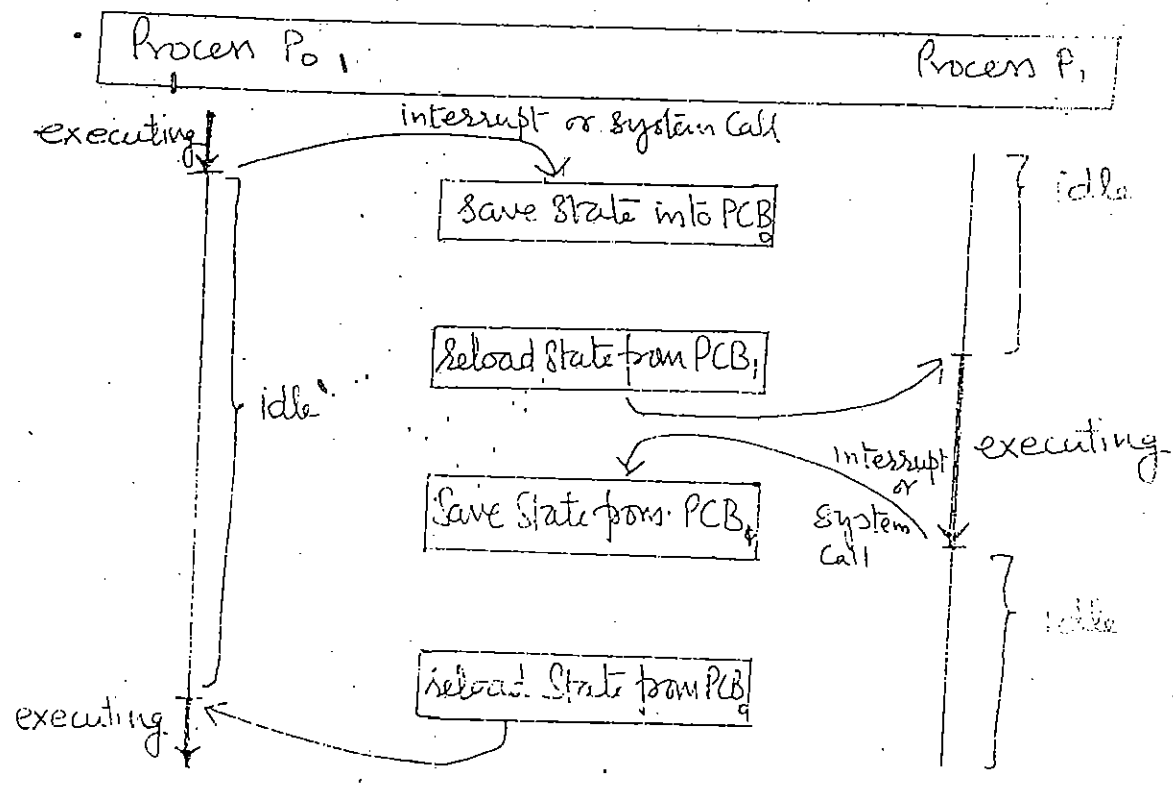
Different types of Schedulers and their actions

Diff b/w blocking & unblocking comp.

# Context Switch

Switching the CPU from one process to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory management information. Context switch time is pure overhead, because system does not perform any useful work (or CPU work) while switching.

The switching speed varies from machine to machine, depending on the memory speed, the no. of registers that must be copied etc.



2  
CPU  
state

CPU Switch from process to process

Cooperating Processes

Concurrent Processes

Cooperating processes  
Concurrent processes

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even though actual parallel processing is not achieved, interleaved execution provides major benefits in processing efficiency. In a multiple processor system it is possible not only to interleave processes but also to overlap them.

Thus, two processes are known as concurrent processes, if their execution can interleave or overlap in time.

The concurrent processes may be either independent processes or cooperating processes.

Independent processes :- The processes which are not affected by each other and they do not share any data, are called independent processes.

Cooperating processes :- The processes which are affected by each other or they share any data, are called cooperating processes.

Processes cooperation is needed due to following reasons.

- (i) Information sharing : Many processes need to share the piece of information such as files, data etc. we must provide an environment to allow concurrent access to these data type of resource.
- (ii) Computation speed up : A process task should be broken into smaller subtasks for faster execution, each will be executing in parallel with each other. These subtasks are cooperating processes because affect each other.
- (iii) Modularity : System may be developed in modular structure to give high extensibility & flexibility.
- (iv) Convenience : An individual user may have many tasks parallel such as: printing, editing etc.

31

where each thread may be running in parallel on a different processor. In a single processor architecture, the CPU generally moves between each thread so quickly as to create the illusion of parallelism, but in reality only one thread is running at a time.

### User and Kernel threads:

Support for threads may be provided at either the user level, for user threads, or by the kernel, for kernel threads.

#### 1) User threads:

- These are supported above the kernel and are implemented by a thread library at user level.
  - Library provides thread creation, scheduling, management with no support from kernel.
  - No kernel intervention is required.
  - User level threads are fast to create and manage.
- User thread libraries include: POSIX Thread, Solaris 2-UI Thread.

Drawback - If kernel is single-threaded, ~~then~~ any user-level thread gets blocked due to some system call. This causes the entire process to block, even if other process threads are available to run.

#### 2) Kernel threads

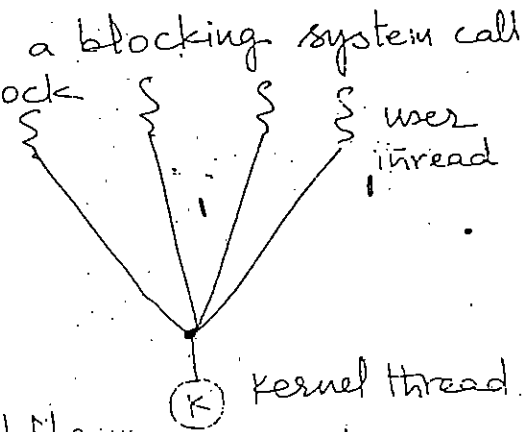
- Kernel threads are supported by OS.
- Kernel performs thread creation, scheduling, management in kernel space.
  - Kernel threads are slower than user threads (to create and manage).
  - If a ~~user~~ thread performs a blocking system call, the kernel can schedule another kernel thread.
  - In multiprocessor environment, the kernel can schedule threads on different processors. e.g. Windows 2000, Solaris 2, BeOS, Tru64 UNIX.

## Multithreading Models:

Many systems do support for both user and kernel threads, resulting in different multithreading models.

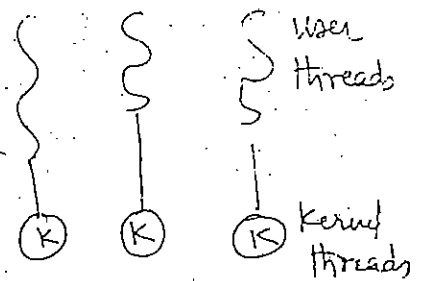
### Many-to-one Model:

- It maps many user-level threads to one kernel thread.
  - Thread management is carried out in user space so it is efficient.
  - But, if one thread makes a blocking system call then entire process will block.
  - Only one user thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- e.g. Green threads - a thread library in Solaris 2



### One-to-one Model:

- It maps each user thread to a kernel thread.
- It allows another thread to run when a thread makes a blocking system call which provides more concurrency.
- It also allows multiple threads to run in parallel on multiprocessors.
- But creating a user thread requires creating the corresponding kernel thread. This overhead can reduce the application performance.



e.g. - Windows NT, Windows 2000, and OS/2

(W)

## Unit 2 (Ch-5)

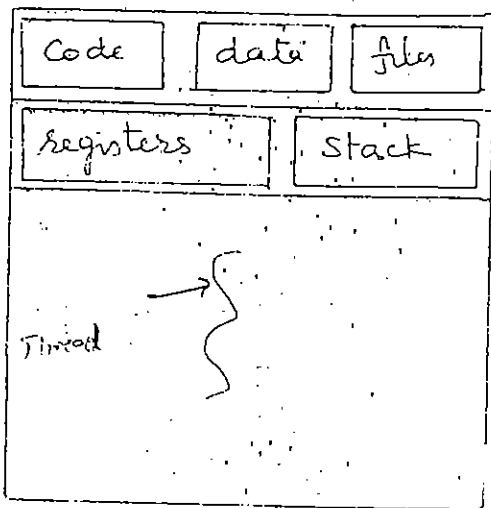
### Threads

30

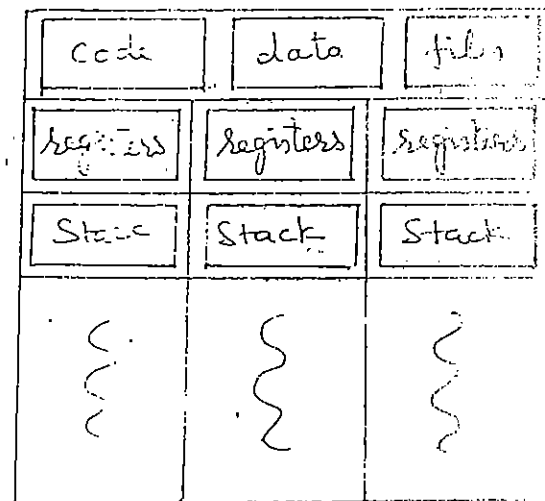
Multithreading refers to the ability of an OS to support multiple threads of execution within a single process.

A thread:

- referred as a light-weight process (LWP)
- a basic unit of CPU utilization
- it has a thread ID, a program counter, a register set and a stack.
- it shares code section, data section, open files etc. with other threads belonging to the same process.
- threads are tightly coupled.
- a single process control block and uses address space associated with the process and all threads belonging to this process.
- Each thread has its own thread control block having Id, counter, register set and other related info.



Single-threaded



Multi-threaded

e.g. Web browser — (i) Displaying message or text or image  
(ii) retrieves data from network

Word processor — (i) Displaying graphics  
(ii) Reading key strokes  
(iii) Spelling & grammar checking

Netflix Browser

If one application should be implemented as a set of related units of execution. There are two ways:  
1. Multiple processes - which are heavy weight and less efficient.

2. Multiple threads - Multiple threads are more useful because of following characteristics:

- Multiple threads are light weight because of share memory.

- Thread creation and termination processes are less time consuming.

- Context switch between two threads takes less time.

- Communication between threads is fast because threads share address space, intervention of kernel is not required.

Benefits: Benefits of multithreaded ~~process~~ can be broken down into 4 categories.

(1) Responsiveness: A process continues running even if one thread is blocked or performing lengthy operation which increases the responsiveness.

(2) Resource sharing: Threads share memory and resources of the process to which they belong. It allows an application to have several different threads of activity all within the same address space.

(3) Economy: Threads share memory and resources of the process which is economical. Also, thread managing is less time consuming than process management. e.g. In Solaris 2, creating a ~~thread~~ process is 30 times slower than creating a thread. Context switch in process is 5 times slower than in threads.

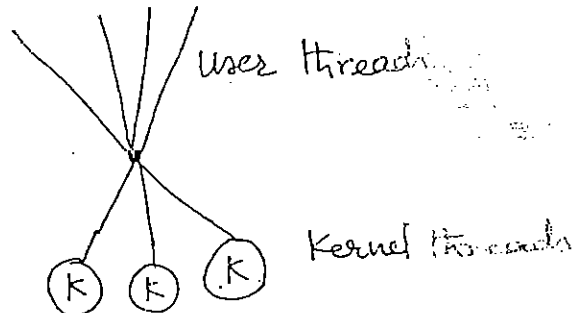
(4) Utilization of Multiprocessor Architecture -

Multithreading on a multiprocessor architecture increases the concurrency.

32

### 3. Many-to-many Model :

- It multiplexes many user-level threads to a smaller or equal no. of kernel threads.
- Number of kernel threads may be specific to either a particular application or a particular machine.



### Concurrency in multithreading Model :

1. Many to one model does not provide true concurrency because kernel can schedule only one thread at a time.
2. One to one model provides greater concurrency but the developer has to be careful not to create greater too many threads within an application.

### Threading Issues

#### 1. The fork and exec System Calls :

- fork() call either duplicates all threads or duplicates only the thread that invoked the fork().
- The exec() will replace the entire process including all threads and LWPs with the program which is specified in the parameter of exec().
- If fork() is followed by exec() immediately, it duplicates the process with the current thread.
- If fork() is not followed by exec(), it duplicates the process with all threads.

2. Cancellation: Thread cancellation is the task of terminating a thread before it has completed. e.g. if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.

The thread that is to be cancelled is referred to as the target thread.

Cancellation may occur in two different situations:

(i) Asynchronous Cancellation: One thread immediately terminates the target thread. e.g. most OS.

(ii) Deferred Cancellation: The target thread can periodically check if it should terminate itself, allowing the target thread an opportunity to terminate itself in an orderly fashion. e.g. Pthread API.

In asynchronous cancellation, a thread may be cancelled in the middle of use of a resource (such as file updation). Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

Deferred cancellation allows a thread to check if it should be cancelled at a point when it can safely be cancelled. Pthreads refer to such points as cancellation points.

### 3. Signal handling:

A signal is generated by the occurrence of a particular event. A generated signal is delivered to a process. Once signal is generated, it must be handled.

A signal may be received either synchronously or asynchronously, depending upon the source and the reason for the event being signalled.

### Types of Signals:

1. Synchronous: Signal is delivered to same process that performed the operation causing the signal e.g. illegal memory access, division by zero.

2. Asynchronous: Signal is generated by an event external to a running process i.e. delivered to different process e.g. timer expire.

Handlers: There are two possible handlers to handle the signal.

(i) A default signal handler: Every signal has a default signal handler that is run by the kernel.

(ii) User defined signal handler: The default action may be overridden by a user-defined signal handler a function. The user-defined function is called to handle the signal rather than the default action.

Both synchronous and asynchronous signals may be handled in different ways. Some signals may be simply ignored; others may be handled by terminating the program e.g. changing the size of a window, an illegal memory access.

Handling

signals in a single threaded process is straight forward. ~~signal~~ ~~process~~

### Different options for delivering signals:

1. Deliver the signal to the thread to which the signal applies - for synchronous signal.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process e.g. UNIX (multithreaded).
4. Assign a specific thread to receive all signals for the process e.g. Solaris 2.

4. Thread Pools: Whenever the server receives a request, it creates a separate thread to service the request. Some potential problems are:

1) Amount of time required to create the thread  
2) Thread will be discarded once it has completed its work.

3) Number of threads may be very large. Unlimited threads could exhaust system resources, such as CPU time or memory.

Above problems can be solved by thread pools.

• A number of threads can be created at process startup and placed into a pool.

• When server receives a request, it awakens a thread from this pool. If thread is available, the request can be passed to thread for service.

• When thread completes its service, it returns to the pool awaiting more work.

• If the thread completes its service, it returns to the pool.

• If the pool contains no available thread, the server waits until one thread becomes free.

Benefits of thread:

1) It is faster to service a request with an existing thread than waiting to create a thread.

2) A thread pool limits the number of threads that exist in system.

Thread-specific Data

Threads belonging to a process share the data of the process. This sharing of data provides one of the benefits of multithreaded programming.

In some cases, each thread may need its own copy of certain data, such data are thread-specific data. e.g. in a transaction processing system, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, thread-specific data is used.

Pthreads: Pthreads refer to the POSIX standard defining an API for thread creation and synchronization. This is a specification for thread behaviour, not an implementation.

Pthreads is a user level library. Libraries implementing the Pthreads specification are restricted to UNIX-based systems such as Solaris 2.

- All ~~thread~~ programs must include the pthread.h header file. • The statement pthread\_t tid declares the identifier for the thread created.

- Each thread has a set of attributes including stack size and scheduling information. The pthread\_attr\_t attr declaration represents the attributes for the thread.

- Attributes can be set in the function call pthread\_attr\_init(&attr).

- A separate thread is created with the pthread\_create function call in the following form.

```
pthread_create(&tid, &attr, runner, argv[1])
```

- After the creation of a separate thread, program has two threads: the initial thread in main and thread performing the runner function call in the runner function. After creating the second thread the main thread will wait for the runner thread to complete by calling the pthread\_join function.

- The runner thread will complete when it calls the function pthread\_exit.

Java threads : Java provides ~~the~~ support at the language level for the creation and management of threads. Java threads can not be classified as either user-level ~~library~~ or kernel-level because java threads are managed by the Java Virtual Machine (JVM) not by a user-level library or kernel.

All Java programs comprise at least a single thread of control. Even a simple java program consisting of only a main method runs as a single thread in the JVM. Java provides the commands that allow the developer to create and manipulate additional threads of control within the program.

### Thread Creation :

Thread can be created by creating a new class that is derived from the Thread class and to override the run method of the Thread class.

class Summation extends Thread

{

...

}

public class ThreadTester

{

Summation thrd = new Summation ( ... );

thrd.start();

}

An object of the derived class will run as a separate thread of control in the JVM. However, creating an object that is derived from the Thread class does not specifically create the new thread; rather, it is start method that actually creates the new thread.

calling the start method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the run method, making the thread eligible to be run by the JVM.

When the summation program runs, two threads are created by the JVM:

- ① The first thread is associated with the application and starts execution at the main method.
- ② The second thread is the summation thread that is created explicitly with the start method. The summation thread begins execution in its run method. The thread terminates when it exits from its run method.

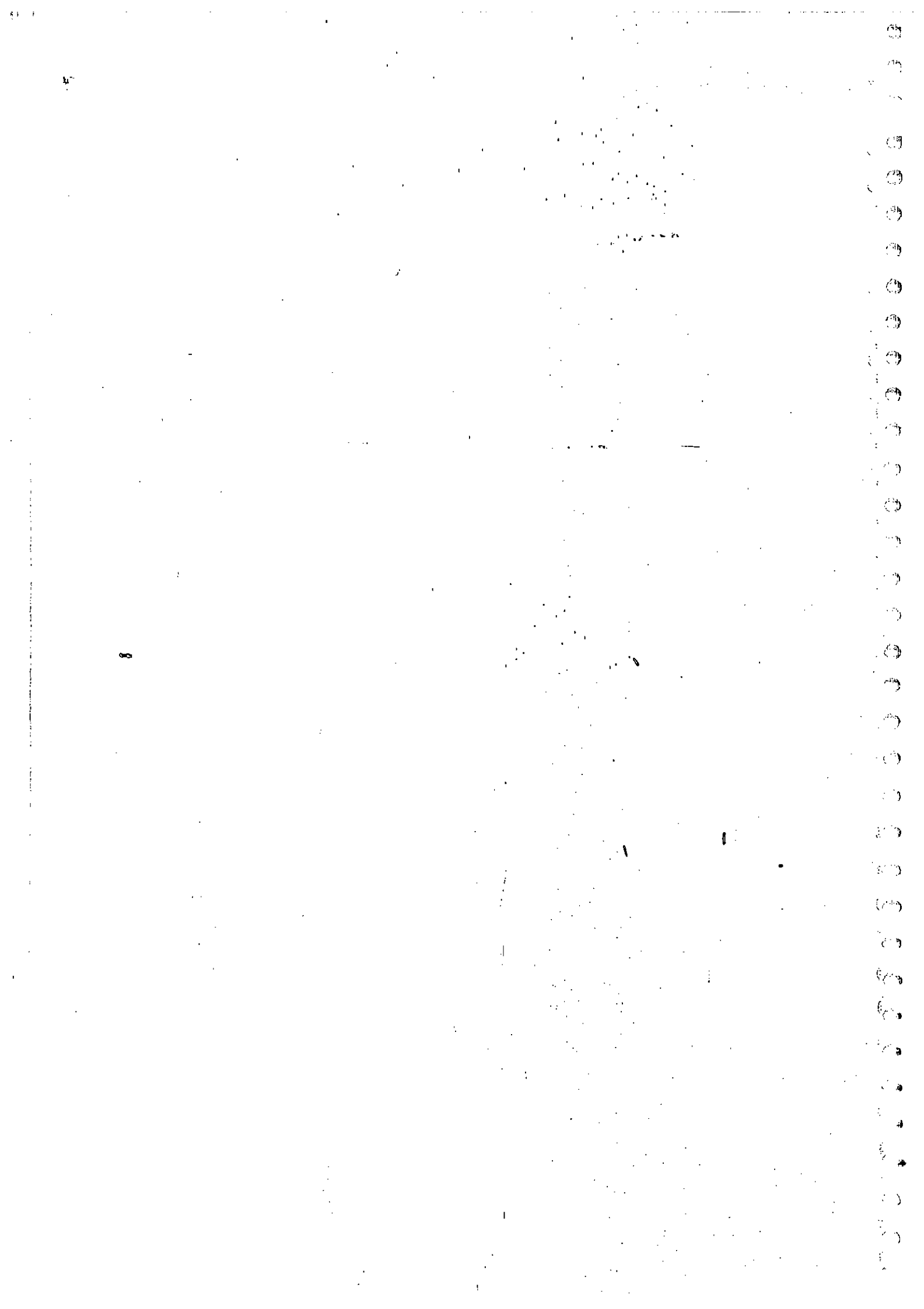
The JVM and the host operating system:

The typical implementation of the JVM is on the top of a host OS. This setup allows the JVM to hide the implementation details of the underlying OS and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying OS, instead leaving that decision to the particular implementation of the JVM.

Windows 95/98/NT/2000 use the one-to-one model, therefore, each Java thread for a JVM running on these OS maps to a kernel thread.

Solaris 2 initially implemented the JVM using the many-to-one model (called green threads).

Java ver 1.1 with Solaris 2.6 was implemented using the many-to-many model.

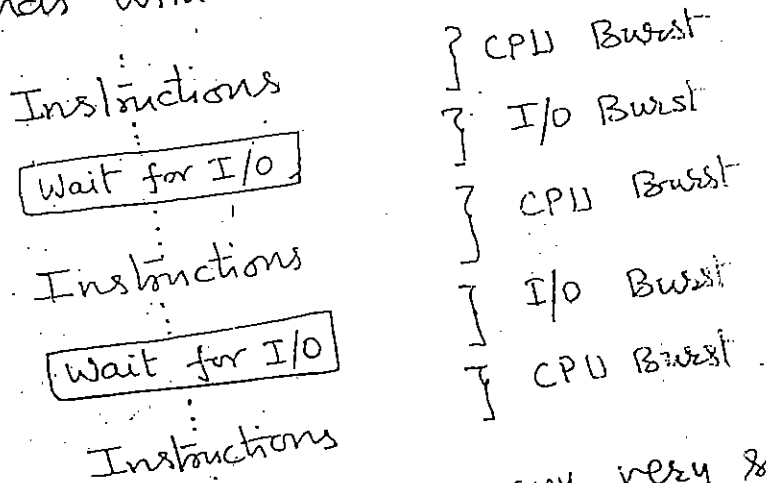


Unit-3 (Chapter 8)

34

CPU Scheduling - CPU Scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. Thus, Objective of CPU scheduling is maximize CPU utilization.

CPU-I/O Burst Cycle: Process execution consists of a cycle of CPU execution and I/O wait alternating between these two cycles. Process execution starts with CPU burst and ends with CPU burst.



An I/O-bound program, have many very short CPU bursts. A CPU-bound program may have a few very long CPU bursts. Type of programs in a system helps to select the CPU-scheduling algorithm.

CPU Scheduler :- This is also known as short term scheduler. It selects the process from ready queue and allocates the CPU for execution on the basis of some scheduling algorithm. The ready queue may be implemented as a FIFO queue, a priority queue, a tree or an orderly.

Neelam Bawa

linked list. The records in the queue are PCBs of the process.

Dispatcher: The module dispatcher 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 time taken by dispatcher to stop one process and start another process is known as dispatch latency.

Scheduling Criteria:

Non-preemptive Scheduling:

When a process switches from running state to waiting state due to either some I/O services or need of OS service. It voluntarily surrenders the control of CPU and it can not be forced to leave the ownership of the processor when a higher-priority process ~~comes~~ comes for execution. This is followed in some OS like windows 3.1, Apple Macintosh OS.

Preemptive Scheduling: A running process may be replaced by another process on the basis of priority, shorter turn CPU burst etc. Preemption necessitates more frequent execution of the scheduler. Preemptive scheduling is more responsive, but it

imposes higher overhead, since each pr thus it incurs a cost.

\* Scheduling Criteria

Different CPU-Scheduling have different properties and may favour one class of processes over another. There can be many criteria to choose one particular scheduling algorithm for a particular situation.

1. CPU-Utilization :- Processor should be busy as much as possible giving CPU utilization on higher side

2. Throughput :- Throughput refers to the amount of work completed in a unit of time. The higher the number, the more work is being done by the system.

3. Turnaround time :- It is time required from the submission of job to completion of job. It is the sum of execution time and waiting time. Scheduling selection should reduce the turnaround time.

4. Waiting time :- It is time spent in waiting in a ready queue. It should be as less as possible.

5. Response time :- It is time required from the submission of a request until the first response is produced.

Schedulers try to maximise the average performance of a system by maximising CPU utilization and throughput, by minimising the turnaround time, waiting time, response time.

## \* Scheduling Algorithms:

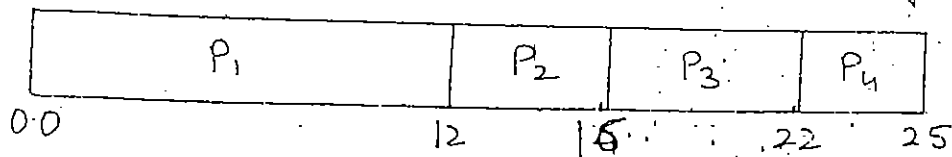
### 1. First-Come, first-served Algorithm: (or First in first out)

It is the simplest algorithm to implement. When a process becomes ready, it joins ready queue at the tail, when CPU is free, process at the head of the queue gets CPU for execution. The average waiting time under the FCFS policy is quite long. It is Non-preemptive i.e process starts execution, it leaves CPU voluntarily only.

Example:

Process	Burst time (m.s.)
P <sub>1</sub>	12
P <sub>2</sub>	4
P <sub>3</sub>	6
P <sub>4</sub>	3

All processes have arrived at 0.0 time in order of P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>



Gantt Chart

Waiting time for P<sub>1</sub> = 0

" " " P<sub>2</sub> = 12

" " " P<sub>3</sub> = 16

" " " P<sub>4</sub> = 22

$$\text{Average Waiting time} = (0 + 12 + 16 + 22) / 4 = 50 / 4 = 12.5 \text{ m.s.}$$

$$\text{Average turn around time} = (12 + 16 + 22 + 25) / 4 = 75 / 4 = 18.75 \text{ m.s.}$$

Convoy effect: If all the processes wait for one big process to get off the CPU, it is known as convoy effect. FCFS Policy suffers due to this effect.

2 Shortest Job-first Scheduling: ~~SJF~~ (SJF) When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.

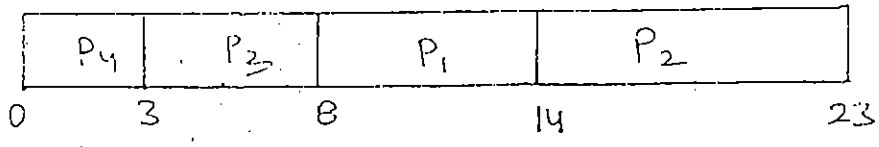
Difficulties in SJF: To know the length of next CPU burst, it is very difficult, thus it can't be implemented at the level of short-term scheduling. It is good policy for long-term scheduling because user can specify process time limit while submitting the job. Only prediction of CPU burst time is possible on the basis of previous burst time of same type or same process.

Types of SJF

Nonpreemptive <sup>SJF</sup>: It allows the currently running process to finish its CPU burst irrespective of burst time of newly arrived processes in the system.

Process	Burst time
P <sub>1</sub>	6
P <sub>2</sub>	9
P <sub>3</sub>	5
P <sub>4</sub>	3

All processes arrive at same time i.e. 0.0 in order of P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>



Gantt Chart

Waiting time for  $P_1 = 8$

" " "  $P_2 = 14$

" " "  $P_3 = 3$

" " "  $P_4 = 0$

Average Waiting time =  $(8 + 14 + 3 + 0) / 4 = 6.25 \text{ ms}$

Average Turnaround time =  $(14 + 23 + 8 + 3) / 4 = 12 \text{ ms}$

(SRT) Preemptive SJF or Shortest-remaining time first -

Preemptive  
SJF

When a new process arrives at the ready queue, the burst time of new process is compared with remaining burst time of running process.

If new process burst time is less than remaining burst time of running process, running process will be preempted and new process will start executing.

Example:

Process	Arrival time	Burst time
$P_1$	0	8
$P_2$	1	5
$P_3$	3	10
$P_4$	4	3

At time 0 : only  $P_1$  →  $P_1$  runs

At time 1 :  $P_1 = 7$  } →  $P_2$  runs  
 $P_2 = 5$  }

At time 3 :  $P_1 = 7$  } →  $P_2$  runs  
 $P_2 = 3$  }

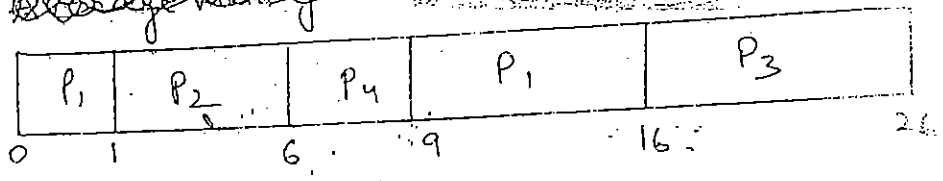
At time 4 :  $P_3 = 10$  }

39

At time 4:  $P_1 = 7$   
 $P_2 = 2$   
 $P_3 = 10$   
 $P_4 = 3$  }  $P_2$  runs then  $P_4, P_1, P_3$

Sequence  $P_1 \rightarrow P_2 \rightarrow P_4 \rightarrow P_1 \rightarrow P_3$   
 time (1) (5) (8) (7) (10)

~~Average waiting time~~



Gantt Chart

Average Waiting time =  $[(9-1-0) + (1-1) + (8-3) + (16-9)]/4$   
 $= 5.75$  ms

Average turn around time =  $[(16-0) + (6-1) + (26-3) + (9-4)]/4$   
 $= 12.25$  ms

\* 3. Priority Scheduling :- ~~the~~ ~~is~~ 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.

Priorities can be defined internally on the basis of some factors such as time limits, memory requirements, no. of open files etc. Priorities can be defined externally also on the basis of factors such as importance & type of process, payable process, processes belonging to different processes.

Problem in Priority Scheduling:  
Indefinite blocking (or Starvation): In a heavily loaded system, a continuous stream of higher priority processes can prevent a low-priority process from ever getting the CPU.

Solution is aging, a technique of gradually increasing the priority of processes that wait in the system for a long time.

Types of Priority Scheduling:

Non-preemptive: This algorithm puts the new process at the head ready queue. When running process finishes execution or terminates itself then only highest priority process from ready queue will be sent for execution.

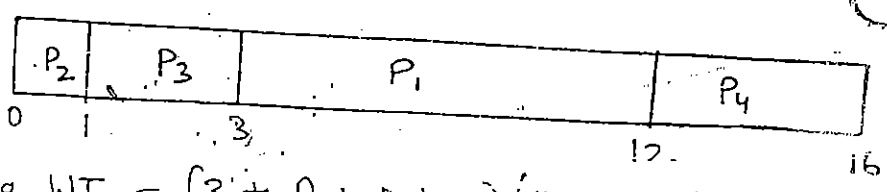
Preemptive: The priority of newly arrived process is compared with running process. If the priority of newly arrived process is higher than currently running process, the process is preempted and new process gets CPU for execution.

Example for Nonpreemptive priority Scheduling:

Process	Burst time	Priority
P <sub>1</sub>	9	3
P <sub>2</sub>	1	1
P <sub>3</sub>	2	2
P <sub>4</sub>	4	4

All the processes arrived at time 0.0 in order of P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>.

(40)

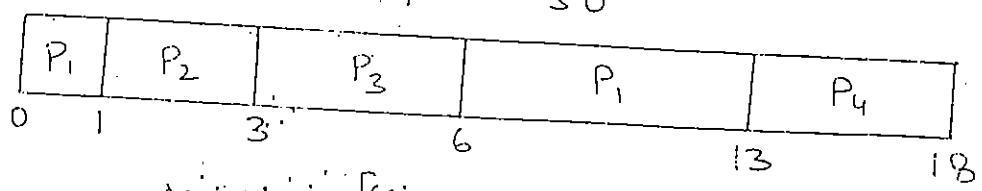


$$\text{Avg. W.T.} = (3 + 0 + 1 + 12) / 4 = 4$$

$$\text{Avg. Turnaround time} = (12 + 1 + 3 + 16) / 4 = 8$$

Example for Preemptive priority scheduling:

Process	Burst time	Priority	Arrival time
P <sub>1</sub>	8	3	0
P <sub>2</sub>	2	1	1
P <sub>3</sub>	3	2	2
P <sub>4</sub>	5	4	3



$$\text{Avg. W.T.} = [(6 - 1 - 0) + (1 - 1) + (3 - 2) + (3 - 3)] / 4$$

$$= 4$$

$$\text{Avg. Turn around time} = [(13 - 0) + (3 - 1) + (6 - 2) + (18 - 3)] / 4$$

$$= 8.5$$

4. Round Robin Scheduling:

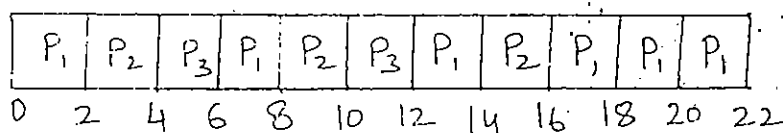
It is a preemptive scheduling specially designed for time-sharing system. The processor time is divided into time slices or time quanta, which are allocated to processes. No process can run for more than one time slice when there are others waiting in the ready queue. Ready queue is implemented as FIFO queue of processes. New processes are added to the tail of the ready queue.

Each process executes either equal to <sup>one</sup> time quantum and is preempted and sent to tail of the queue.  
 Or if burst time is less than time quantum, it finishes execution and exits.

Performance of RR algorithm depends on the size of time quantum. If the time quantum is very large, RR is same as FCFS policy. If the time quantum is very small, RR is overhead in context switching will be more.

Example:

Process	Burst time	Time quantum = 2
P <sub>1</sub>	12	All processes arrived at time 0.0 in the order of P <sub>1</sub> , P <sub>2</sub> , P <sub>3</sub>
P <sub>2</sub>	6	
P <sub>3</sub>	4	



$$\text{Avg W.T.} = \frac{[(20 - 10) + (14 - 4) - (10 - 2)]}{3}$$

$$= 9.33$$

$$\text{Avg Turnaround time} = \frac{(22 + 16 + 12)}{3} = 16.66$$

#### \* 5. Multilevel Queue Scheduling:

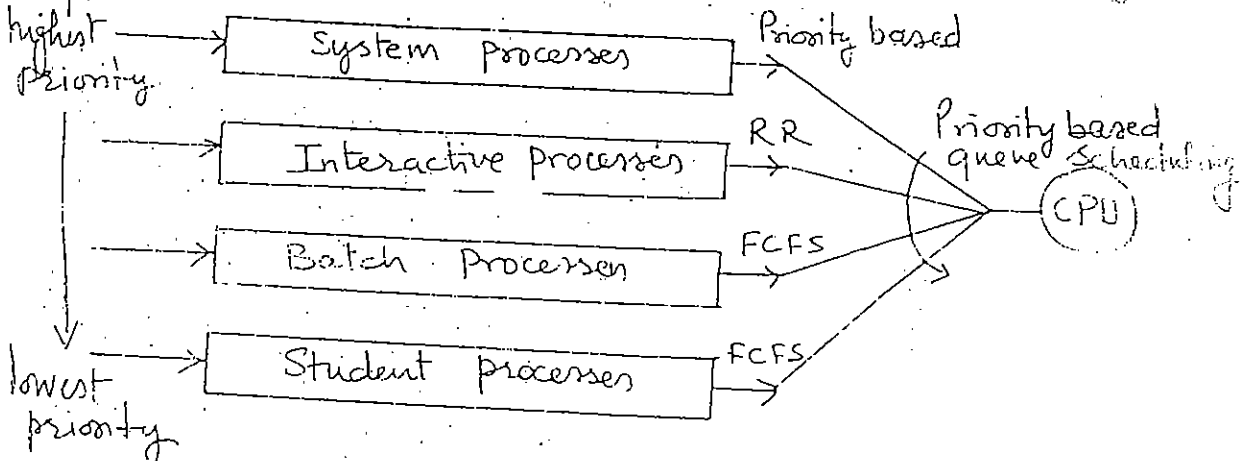
No single CPU scheduling is proper with a mixed system with time-critical systems processes, a multitude of interactive processes, very long non-interactive jobs. One approach is to combine several scheduling algorithms.

A ready queue is partitioned into several separate queues. A process is permanently assigned to one queue based on some attribute of process such as

memory size, process priority or process type. Each queue has its own scheduling algorithm. There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

For example ① Separate queues may be used for foreground and background processes. The foreground queue may be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. Foreground queue may have absolute priority over the background queue. When the higher priority queues become empty, a lower priority queue is serviced. A low priority process may be preempted by a higher priority ~~arrival~~ queue due to new arrival.

② If four queues are there on the basis of process type, each having one scheduling policy. Each queue has absolute priority over lower-priority queues.



Queues scheduling can be implemented on the basis of time slice. Each queue gets a certain portion of CPU time, which can be scheduled among the various processes in its queue.

## 6. Multilevel feedback Queue Scheduling:

This algorithm allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue. I/O bound and interactive processes may continue in higher priority queues. If a process waits too long in a lower priority queue, it may be moved to a higher-priority queue. This is aging which prevents starvation.

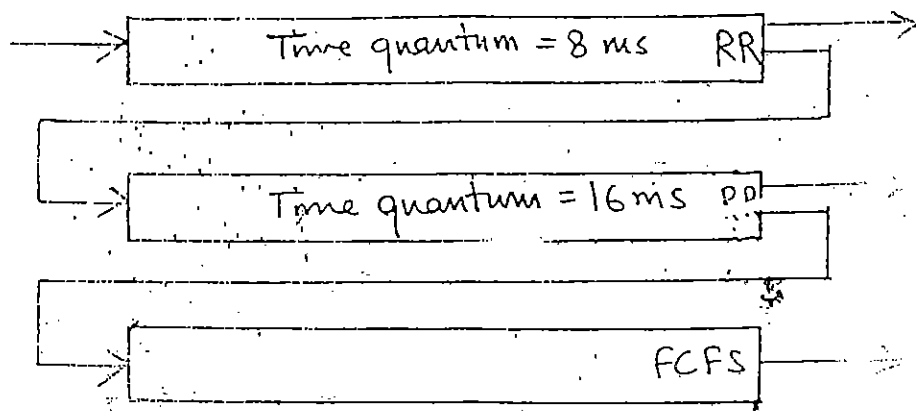
For example: Performance of this algorithm depends on the following parameters:

1. The no. of queues.
2. The Scheduling algorithm for each queue
3. The method to determine when to upgrade a process to a higher priority queue
4. The method to determine when to demote a process to a lower priority queue.

This is most general and complex algorithm and can be configured to match a specific system.  
For example:

If this algorithm is implemented with three queues the scheduler first executes all processes in queue 0 then 1 and then 2. \*A process that arrives for queue 0 will preempt a process in queue 1 and so on. A process in queue 0 gets a time quantum of 8 ms. If a process does not finish in 8ms it again joins tail of queue 1 and so on. Thus the process with short CPU burst will get higher priority. (Fig next page)

\* Queue 0 and 1 scheduling scheme is RR and queue 2 has FCFS.



\* 7. Multiple processor Scheduling :

In a homogeneous multiprocessor environment (identical processors having same functionalities), two approaches can be followed

1) Separate queue for each processor : It is possible to provide a separate queue for each processor. But one processor can be idle with an empty queue while another processor may be very busy with highly loaded queue.

2. A common ready queue : All processes go into one queue and are scheduled onto any available processor. For such implementation, two approaches can be followed

(i) Self-Scheduling : Each processor is self-scheduling and examines the common ready queue and selects a process to execute. In this scheme, multiple processors ~~will~~<sup>may</sup> try to access and update a common data structure. We must ensure that processors do not choose the same process.

(ii) Master-Slave Structure : One processor <sup>acts</sup> as scheduler for other processors which is called master processor. In some systems, master processor may have all

scheduling decisions, I/O processing and other activities. The other processors only execute user code. Thus one processor accesses the system data structures avoiding the need for data sharing. But it may lead to bottleneck as one CPU is performing all the operations.

## 8. Real-time Scheduling:

Hard real-time system: In hard real-time system a critical task should be completed within a guaranteed amount of time or at a particular time. A process is submitted with requirement of amount of time needed to finish I/O.

Either scheduler can reject the process or admit the process guaranteeing that the process will complete on time. This is known as resource reservation. This guarantee is not possible with general purpose system. Thus, the hard real-time systems are composed of special-purpose software running on hardware dedicated to their critical processes.

Soft real-time systems: Here, critical processes are getting priority higher than the priority of less important processes. Thus, the system should have the priority scheduling algorithm, and the priority of real-time processes must not degrade over time. Aging of general processes should be disallowed. The dispatch latency must be small. The smaller the latency, the faster a real-time process can start execution.

Some system calls are complex and I/O devices are slow and OS is forced to wait for system

call to complete or for an I/O block to take place before doing a context switch.

Solution: ① System calls can be preemptible to keep dispatch latency low. Preemption points can be inserted at safe locations where presence of <sup>ready</sup> real process is checked. Safe points are those where kernel data structures are not being modified.

② All kernel data structures must be protected through the use of various synchronization mechanisms. Thus, kernel data structure can be preemptible, because high priority process can not modify the data structure if low priority process is modifying the data structure.

Then, high priority process has to wait for longer lower priority process which is known as priority inversion. This problem can be solved by priority-inheritance protocol, in which all the <sup>low</sup> priority processes while accessing data structures inherit the high priority. Once they finish, they acquire their original priority.

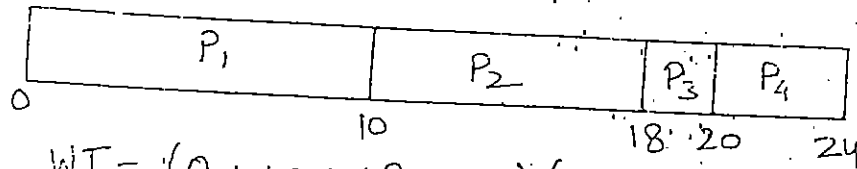
## Problems

Q.1. 4 batch jobs  $P_1, P_2, P_3, P_4$  arrive at a computer center at most the same order. The estimated running times are 10, 8, 2, 4 ms. The priorities are 3, 4, 1, 2. Time quantum is 2 ms. Draw Gantt chart and compute <sup>Avg.</sup> waiting and avg. turn around time for following algorithms:  
 (i) FCFS, (ii) SJF (preemptive or non-preemptive)  
 (iii) Priority and (iv) RR.

Solution:

FCFS:

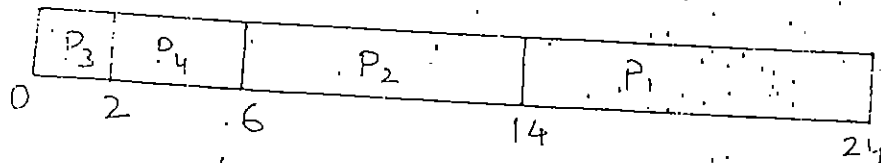
Process	Burst time	Priority
$P_1$	10	3
$P_2$	8	4
$P_3$	2	1
$P_4$	4	2



$$\text{Avg. WT} = (0 + 10 + 18 + 20) / 4 = 12$$

$$\text{Avg. TAT} = (10 + 18 + 20 + 24) / 4 = 18$$

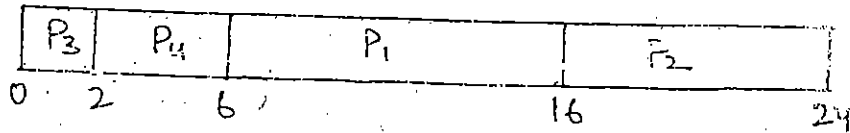
SJF:



$$\text{Avg. WT} = (14 + 6 + 0 + 2) / 4 = 5.5$$

$$\text{Avg. TAT} = (24 + 14 + 2 + 6) / 4 = 11.5$$

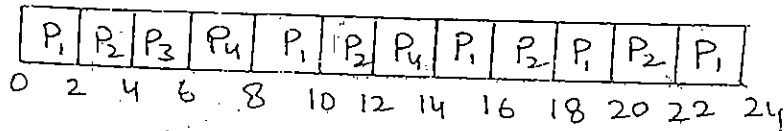
Priority based:



$$\text{Avg WT} = (6 + 16 + 0 + 2) / 4 = 6$$

$$\text{Avg TAT} = (16 + 24 + 2 + 6) / 4 = 12$$

Round Robin



$$\text{Avg WT} = [(22 - 8) + (20 - 6) + 4 + (12 - 2)] / 4$$

$$= (14 + 14 + 4 + 10) / 4$$

$$= 10.5$$

$$\text{Avg TAT} = (24 + 22 + 6 + 12) / 4$$

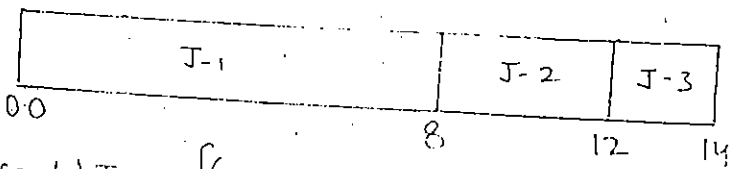
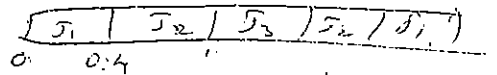
$$= 16.5$$

Q.2. The following jobs arrive at times indicated, each job will run listed amount of time

Jobs	Arrival time	Burst time	Priority	Complete Avg waiting time and average turn around time
1	0:0	8	2	
2	2:4	4	1	
3	1:0	2	3	

Solution

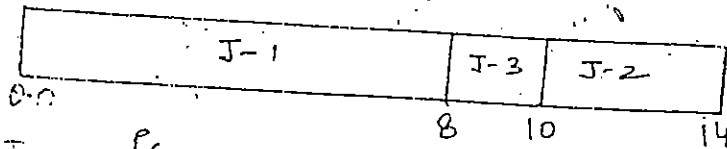
PCFS



$$\text{Avg WT} = [(0 - 0) + (8 - 4) + (12 - 10)] / 3 = 6.2$$

$$\text{Avg TAT} = [(8 - 0) + (12 - 4) + (14 - 10)] / 3 = 10.86$$

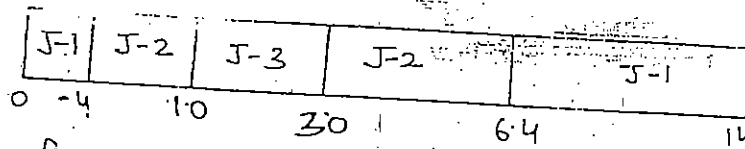
### SJF Non-preemptive



$$\text{Avg WT} = [(0-0) + (10-4) + (8-10)] / 3 = 5.53$$

$$\text{Avg TAT} = [(8-0) + (14-4) + (10-10)] / 3 = 10.2$$

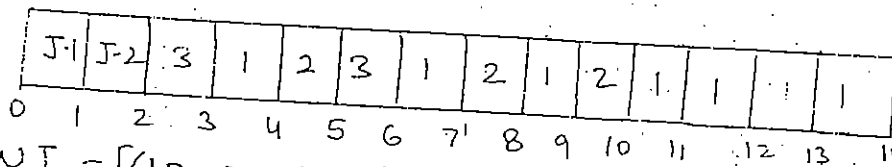
### SJF preemptive



$$\text{Avg WT} = [(6.4 - 4 - 0) + (3.0 - 6.4) + (1 - 1)] / 3 = 2.66$$

$$\text{Avg TAT} = [(14 - 0) + (6.4 - 4) + (3.0 - 1)] / 3 = 7.33$$

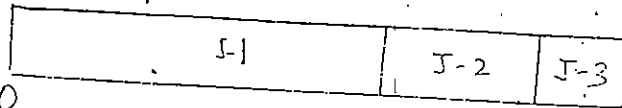
### Round Robin



$$\text{Avg WT} = [(13 - 7 - 0) + (9 - 3 - 4) + (5 - 1 - 1)] / 3 = 4.86$$

$$\text{Avg TAT} = [(14 - 0) + (10 - 4) + (6 - 1)] / 3 = 9.53$$

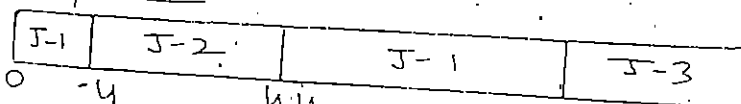
### Priority (Non-preemptive)



$$\text{Avg WT} = [(0-0) + (8-4) + (12-14)] / 3 = 2$$

$$\text{Avg TAT} = [(8-0) + (12-4) + (14-14)] / 3 = 8.66$$

### Priority (Preemptive)



$$\text{Avg WT} = [(4.4 - 4 - 0) + (4 - 4) + (12 - 1)] / 3 = 5.0$$

$$\text{Avg TAT} = [(12 - 0) + (4.4 - 4) + (14 - 1)] / 3 = 9.66$$