KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

OPERATING SYSTEM.

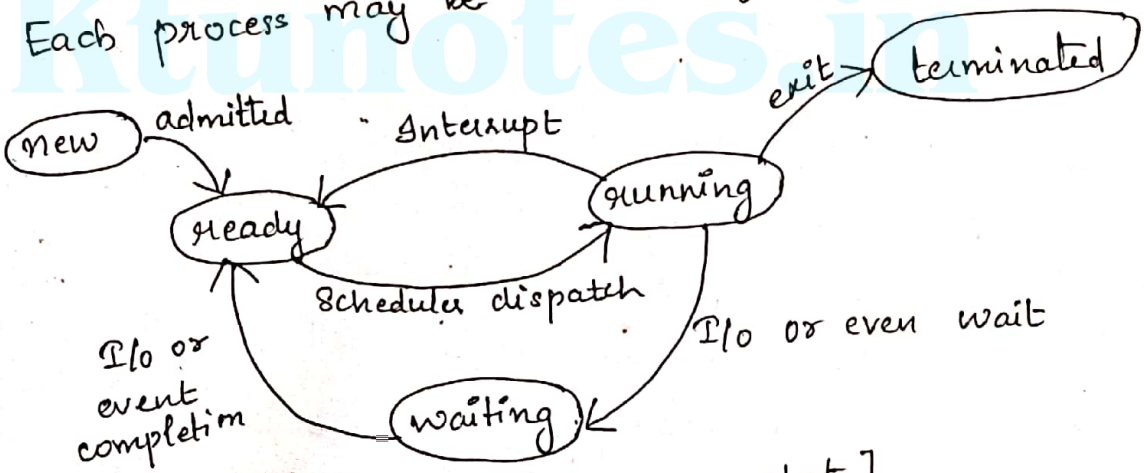
MODULE - 2

PROCESS MANAGEMENT.

* Process is a program in execution.
* Process need resources → CPU time, memory, files & I/O devices to accomplish its task.
* Resources are allocated to the process either when it is created or while it is executing.

Process State

* A process executes, it changes state.
* State of the process is defined by the current activity of that process.
* Each process may be in one of the foll. states:



[fig: Diagram of process state]

* **New**: The process is being created.
* **Running**: Instructions are being executed.
* **Waiting**: Process is waiting for some event to occur.
* **Ready**: Process is waiting to be assigned to a processor.
* **Terminated**: Process has finished execution.

* Only one process can be running on any processor at any instant, although many process may be ready & waiting.

Process Control Block.

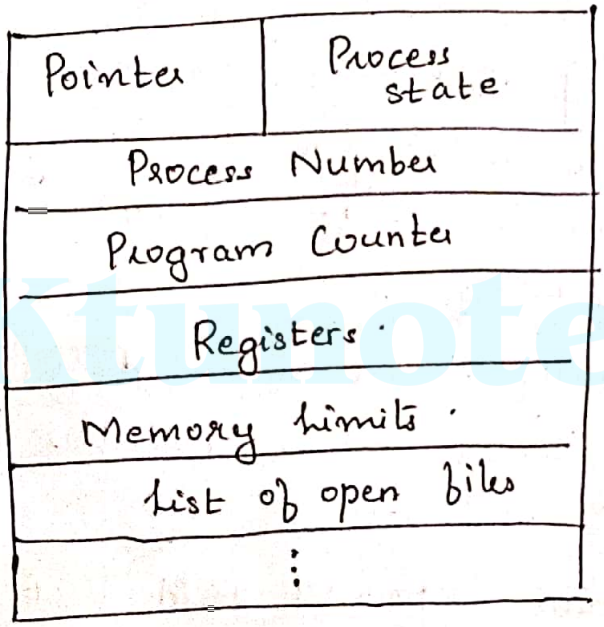
↳ Each process is represented in the OS by a process control block (PCB).

↳ PCB also called a task control block.

↳ PCB contains many peices of information associated with a specific process including:

* Process state : The state may be new, ready, running, waiting, halted & so on.
* Program counter : The counter indicates the address of the next instruction to be executed for this process.
* CPU Registers.
  - Registers vary in number & type, depending on the computer Architecture.
  - Registers include accumulators, index registers, stack pointer & general purpose registers (GPR)
  - Along with the pgm counter, this state information must be saved when an interrupt occurs.
* CPU scheduling information : This information includes process priority, pointers to scheduling queues & other scheduling parameters.

* Memory management information : This information include the value of base & limit registers, the page tables or the segment table, depending on the memory system.
* Accounting information : This information includes the amount of CPU & real time used, time limits, account number, job or process numbers.
* I/o status information : This information includes the list of I/o devices allocated to this process, a list of open files & so on.



[fig: Process Control Block]

Threads:

* Process is a pgm that performs single thread of execution.
* Single thread of ~~execu~~ control allows the process to perform only one task at one time.
* Modern OS have extended the process concept to allow a process to have multiple threads of execution & thus to perform more than one task at a time.

Process Scheduling.

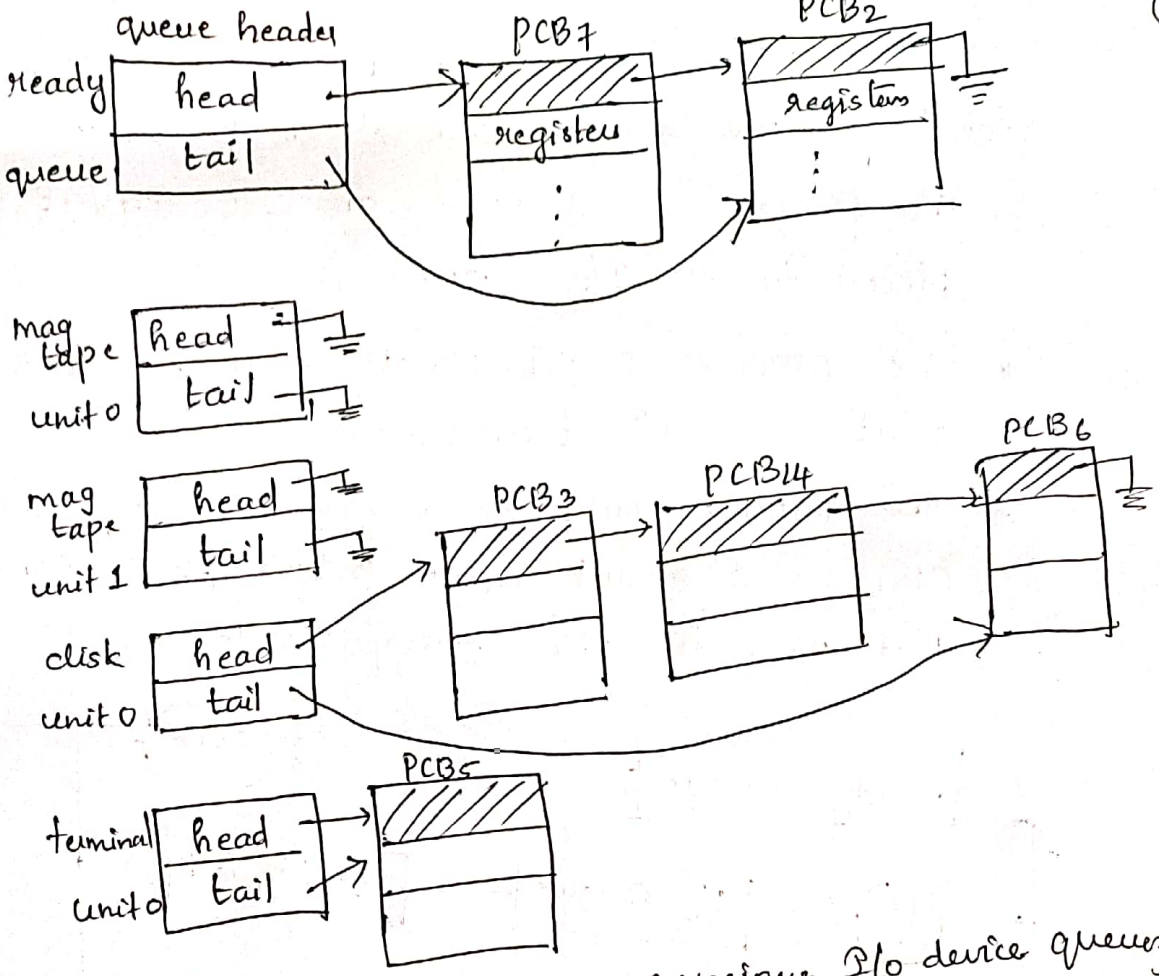
* Objective of multiprogramming is to have some processes running at all time, to maximize CPU utilization.
* Objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each pgm while it is running.

1. Scheduling Queues.

* As processes enter the system, they are put into a job queue.
* Queue consists of all processes in the system.
* Processes that are residing in main mly & are ready & waiting to execute are kept on a list called the ready queue.
* Queue is generally stored as linked list.
* A ready-queue header contains pointers to the first & final PCB in the list.
* Each PCB contains a pointer field that points to the next PCB in the ready queue.
* In case of an I/O request, a request may be to a dedicated tape drive or to a shared device such as disk.

Since the system has many processes the disk may be busy with the I/o request of some other process. The process therefore may have to wait for the disk.

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



(fig: The ready queue & various I/O device queues)

→ Common representation of process scheduling is a queueing diagram.

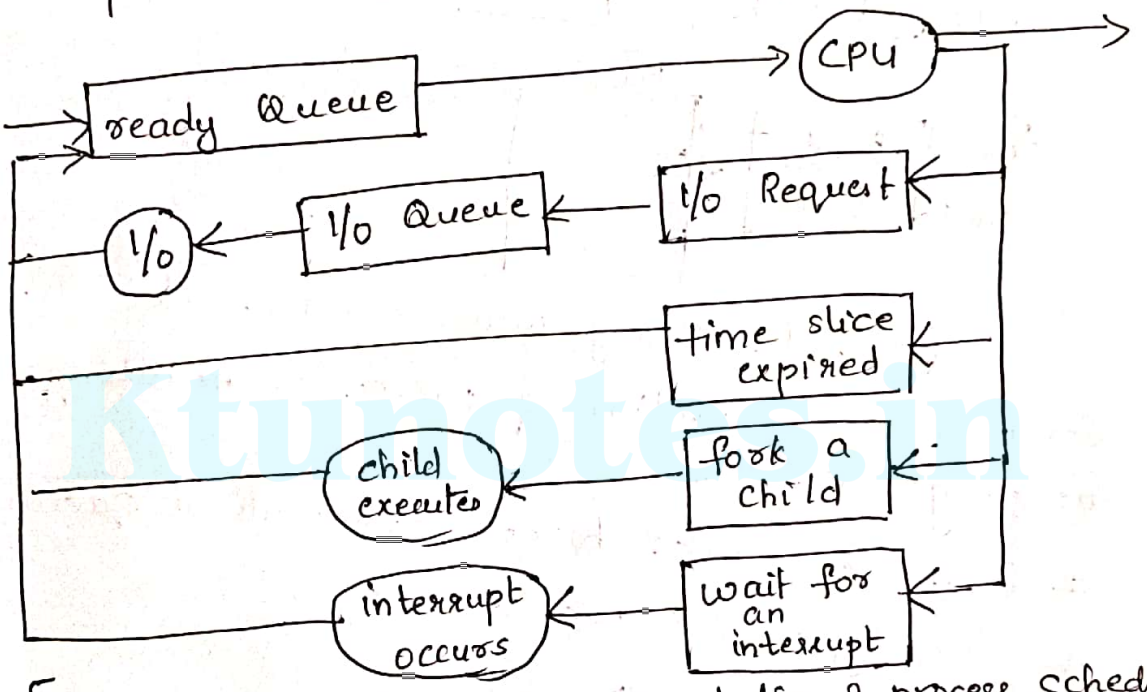
* Each rectangular boxes represents a queue.
* Two types of queues are present.
  - ✓ Ready Queue.
  - ✓ Set of device queues.
* Circle represent the resources that serve the queues, the arrow indicates the flow of processes in the system.

→ A new process is initially put in a ready queue.

→ It waits in the ready queue until it is selected for execution.

↳ Once the process is assigned to the CPU & is executing, one of the several events occurs:

* The process could issue an I/O request & then placed in an I/O queue.
* The process could create a new process & wait for its termination.
* The process could be removed forcibly from the CPU, as a result of an interrupt & can be put back in the ready queue.



[fig: Queueing diagram representation of process scheduling]

2. Schedulers

* A process migrates between the various scheduling queues throughout its lifetime.
* OS must select for scheduling purposes, process from these queues in some fashion.
* Selection process is carried out by the schedulers.

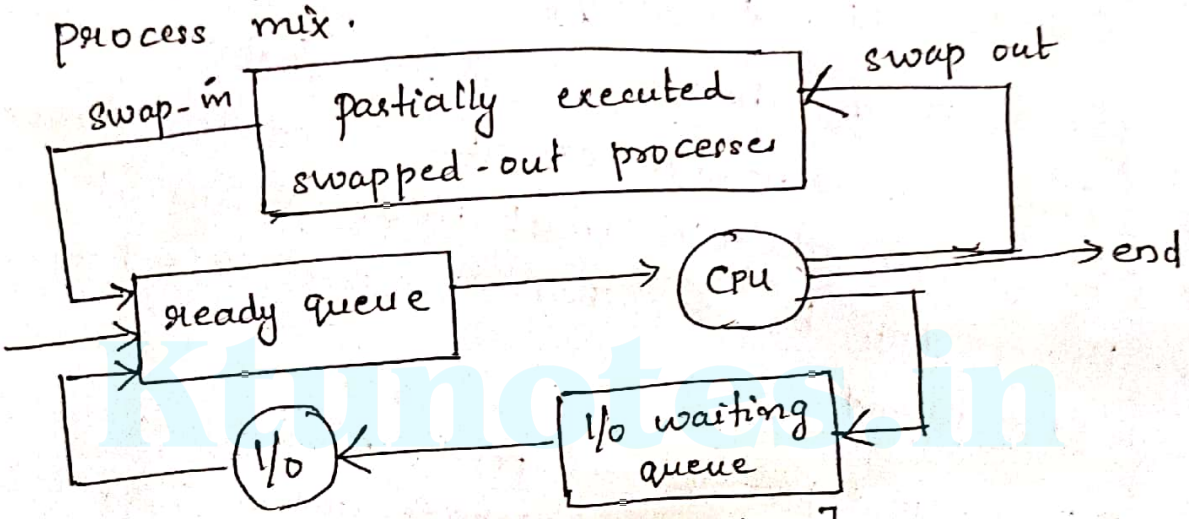
* Long-term scheduler, selects processes from job pool & loads them into memory for execution.
* Short-term scheduler, selects from among the processes that are ready to execute & allocates the CPU to one of them.
* Primary distinction b/w long-term scheduler & short-term scheduler is the frequency of their execution.
* Short-term scheduler must select a new process for the CPU frequently.
* Short-term scheduler must be fast.
* Long-term scheduler executes much less frequently.
* Long-term scheduler must make a careful selection.
* Process can be described as either I/O bound or CPU bound.
  - I/O bound process → spends more of its time doing I/O than it spends doing computation.
  - CPU bound process → generates I/O requests infrequently using more of its time doing computation than an I/O bound process uses.
* Long-term scheduler should select a good process mix of I/O bound & CPU-bound processes.
* If all processes are I/O bound, the ready queue will almost always be empty & the short term scheduler will have little to do.
* If all process are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused & again the system will be unbalanced.

* The system with the best performance will have a combination of CPU bound & I/O bound processes.

* Medium-term scheduler removes processes from memory & then reduces the degree of multiprogramming.

* Processes are swapped-out & is later swapped-in, by the medium-term scheduler.

* Swapping may be necessary to improve the process mix.



[fig: Medium-term scheduling]

3. Context switch.

* Switching the CPU to another process requires saving the state of old process & loading the ~~saved~~ saved state for the new process. This task is known as Context Switch.

* Context of a process is represented in the PCB of a process, it includes the value of the CPU registers, the process state & mly mgmt information.

* When context switch occurs, the kernel saves the context of old process in its PCB & loads the saved context of the new process scheduled to run.

* Context switch time is pure overhead, because the system does not useful work while switching.
Its speed varies from machine to machine, depending on the m/y speed, the no. of register that must be copied & existence of special instructions.

CPU Scheduling

* CPU Scheduling is the basis of multiprogrammed OS.
* By switching the CPU among processes, the OS can make the computer more productive.
* Objective of multiprogramming is to have process running at all time, inorder to maximize CPU utilization.
* In multiprogramming, we use the time productively.
* Several processes are kept in memory at one time.
* When one process has to wait, the OS take the CPU away from that process & gives the CPU to another process.

CPU Scheduler

* Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
* Selection process is carried out by the short-term scheduler.
* Scheduler selects from among the processes in memory that are ready to execute & allocates the CPU to one of them.

* A ready queue may be implemented as FIFO queue, priority queue, a tree.
* All the processes in the ready queue are waiting for a chance to run on the CPU.

Preemptive Scheduling.

* CPU scheduling decisions takes place under the foll. 4 circumstances:

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

* In circumstances 1 & 4, there is no choice in terms of scheduling.
* There is choice in circumstances 2 & 3.
→ when scheduling takes place under circumstance 1 & 4, we call it as non-preemptive scheduling
otherwise, it is preemptive Scheduling.

Non-Preemptive Scheduling.

In 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.

↳ Preemptive scheduling is problematic.

* Suppose two processes are sharing data.
  - The first in the midst of updating the data.
  - While second process try to read the data, which are currently in an inconsistent state.

Dispatcher.

* Major component involved in the CPU scheduling function is dispatcher.
* Dispatcher is the module that gives control of the CPU to the process selected by the short term scheduler.
* Dispatcher should be as fast as possible.
* ~~Time it taker for the dispatcher to sto~~
* It is invoked during every process switch.
* Time it takes for the dispatcher to stop one process & start another running is known as the dispatch latency.

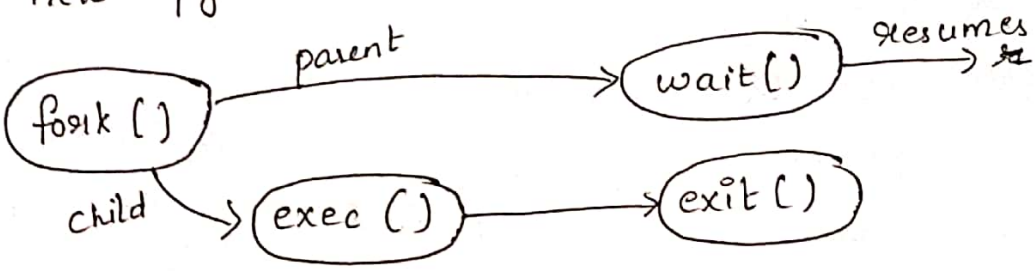
Operations on Processes.

* System must provide mechanisms for:
  - Process creation
  - Process Termination

Process Creation.

* A process may create several new processes, via a create-process system call, during the course of execution.
* The creating process is called a parent process & the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.
* Generally, process identified & managed via a process identifier (pid).
* Resource sharing options.
  - Parent & children share all resources.
  - Children share subset of parents resources.
* Execution options.
  - Parent & children execute concurrently.
  - Parent waits until children terminate..
* Address space of new process.
  - The child process is a duplicate of the parent process.
  - The child process has a new pgm loaded into it.

* fork() system call creates a new process.
* exec() system call used after a fork() to replace the process memory space with a new pgm.



Process Termination

* Process executes last statement & then asks the OS to delete it using the exit() system call.
* Return status data from child to parent [via wait()]
* A process can cause the termination of another process via an appropriate system call.
  Such a system call can be invoked only by the parent of the process that is to be terminated.
* A parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
* A parent may terminate the execution of one of its children for a variety of reasons, such as:
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - The parent is exiting & the OS does not allow a child to continue if its parent terminates.

* Some OS do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated. This phenomenon referred to as cascading termination.

* The parent process may wait for termination of a child process by using the wait() system call. The call returns status information & the pid of the terminated process.

INTER PROCESS COMMUNICATION

* Inter-process Communication (IPC) refers to the coordination of activities among cooperating processes.

* Two modes of IPC:
  - Shared Memory
  - Message Passing.

1. Shared Memory Systems.

* IPC communication using shared m/y requires communicating processes to establish a region of shared memory.

* A shared m/y region resides in the address space of the process creating the shared m/y segment.

* Other processes that wish to communicate using this shared m/y segment must attach it to their address space.

* They can then exchange information by reading & writing data in the shared areas.

* The communication is under the control of the uses processes not the OS.

Example for cooperating processes:

→ Producer – Consumer problem.

* A producer produces information that is consumed by a consumer process.
* One solution to the producer-consumer problem uses shared mly.
* A buffer which reside in a region of mly that is shared by the producer & consumer process is used.
* The producer & consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
* Two types of buffers can be used:
  - Unbounded buffer places no ~~partical~~ limit on the size of the buffer.
  - Bounded buffer assumes that there is a fixed buffer size.

Bounded Buffer – Shared Memory Solution.

* Shared data.

```
#define BUFFER_SIZE 10.
typedef struct
{ ....
} item;
item buffer [BUFFER_SIZE];
  int in = 0;
  int out = 0;
```

* The shared buffer is implemented as a circular array with two logical pointers: in & out.

* The variable 'in' points to the next free position in the buffer.
'out' points to the first full position in the buffer.

* The buffer is empty when in == out;
The buffer is full when ((in+1) % BUFFER_SIZE) == out)

Producer

```
item next_produced;
while (true)
{
    /* produce an item in next produced */
while((in+1) % BUFFER_SIZE) == out);
                /* do nothing */
buffer [in] = next_produced;
in = (in+1) % BUFFER_SIZE;
}
```

Consumer

```
item next_consumed;
while (true)
{   while (in == out);
                /* do nothing */
    next_consumed = buffer [out];
    out = (out+1) % BUFFER_SIZE;
        /* consume item in next consumed */
}
```

2. Message Passing Systems

* Message Passing provides a mechanism to allow processes to communicate & to synchronize their actions without sharing the same address space.
* A message-passing facility provides 2 operations:
  - send (message)
  - receive (message).
* If processes P & Q want to communicate, they must send messages to & receive messages from each other. A communication link must exist b/w them.

Naming

* Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.
* Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() & receive() are defined as:
  - send (P, message) – Send a message to process P.
  - receive (Q, message) – Receive a msg from process Q.
* Symmetry in addressing – Both the sender process & the receiver process must name the other to communicate.
* Assymetry in addressing – Here only the sender names the recipient; the recipient is not required to name the sender.
  - send (P, message) – Send a msg to process P.
  - receive (id, message) – Receive a msg from any process.

* With indirect communication, the msgs are sent to and received from mailboxes or ports.

* Two processes can communicate only if the processes have a shared mail box.
  - send (A, message) – Send a msg to mailbox A.
  - receive (A, message) – Receive a msg from mailbox A.

* The OS must provide a mechanism that allows a process to do the foll:
  - Create a new mailbox.
  - Send & receive msgs through the mailbox.
  - Delete a mailbox.

Synchronization:

* Message passing may be either blocking or non-blocking also known as synchronous & asynchronous.
  - Blocking send – The sending process is blocked until the msg is received by the receiving process or by the mailbox.
  - Non-Blocking send – The sending process sends the msg & resumes operation.
  - Blocking receive – The receiver blocks until a msg is available.
  - Non-blocking receive – The receiver retrieves either a valid msg or a null.

Buffering

* Whether communication is direct or indirect, msgs exchanged by communicating processes reside in a temporary queue.
* Such queues can be implemented in 3 ways:
  - Zero Capacity – The Queue has a maximum length of zero; thus, the link cannot have any msgs waiting in it. In this case, the sender must block until the recipient receives the msg.
  - Bounded capacity – The queue has finite length n; thus atmost n messages can reside in it. If the queue is not full when a new msg is sent, the msg is placed in the queue, & the sender can continue execution w/o waiting. The link's capacity is finite. If the link is full, the sender must block until space is available in the queue.
  - Unbounded Capacity – The queue's length is infinite; thus any no. of msgs can wait in it. The sender never blocks.

PIPES

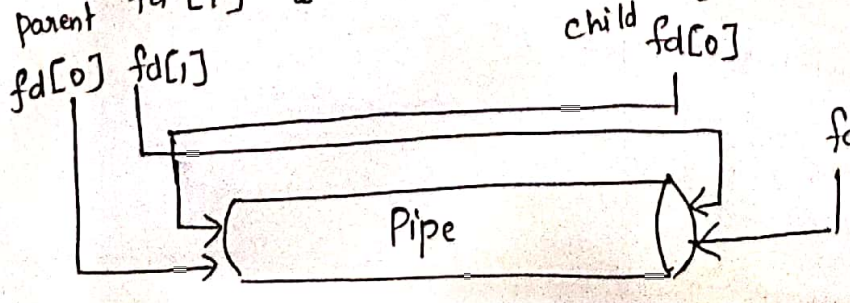
* A pipe acts as a conduit allowing two processes to communicate.
* In implementing a pipe, 4 issues must be considered:
  1. Does the pipe allow unidirectional communication or bi-directional communication?
  2. If 2-way comm is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

3. Must a relationship (such as parent-child) exist b/w the communicating processes?

4. Can the pipes communicate over a n/w or must the communicating processes reside on the same machine?

Ordinary Pipes.

* Ordinary pipes allow two processes to communicate in a producer-consumer fashion.
  The producer writes to one end of the pipe (write end) & the consumer reads from the other end (read end).
* As a result, ordinary pipes are unidirectional, allowing only one-way communication.
* If a two-way comm^n is required, two pipes must be used, with each pipe sending data in a different direction.
* Ordinary pipes are constructed using the function pipe (int fd[]).
  This function creates a pipe that is accessed through the int fd[] file descriptors.
  fd[0] is the read-end of the pipe.
  fd[1] is the write-end of the pipe.



[fig: File descriptors for an ordinary pipe]

Named Pipes

* Ordinary pipes provide a simple comm^n mechanism b/w a pair of processes.
  However, ordinary pipes exist only while the processes are communicating with one another.
  Once the processes have finished communicating & terminated, the ordinary pipe ceases to exist.

* Named pipes provide a much more powerful comm^n tool.
  Comm^n can be bidirectional & no parent-child relationship is required.
  Onced a named pipe is established, several processes can use it for comm^n.
  A named pipe has several writers.
  Additionally, named pipes continue to exist after communicating processes have finished.

* Named pipes are referred to as FIFOs.
  Once created, they appear as typical files in the file system.
  A FIFO is created with the mkfifo() system call & manipulated with the ordinary open(), read(), write() & close() system call.
  It will continue to exist until it is explicitly deleted from the file s/m.
  Although FIFOs allow bidirectional comm^n, only half-duplex transmission is permited. If data must travel in both directions, two FIFOs are typically used.

SCHEDULING CRITERIA

↳ **CPU utilization.**

✓ Need to keep the CPU as busy as possible.

↳ **Throughput.**

✓ Work is being done if the CPU is busy executing processes.

✓ One measure of work is the no. of processes completed per ~~unit~~time unit called throughput.

✓ for long processes → 1 process/hr
for short transaction → 10 processes/hr.

↳ **Turnaround Time**

✓ We need to know how long it takes to execute a particular process.

✓ The interval from the time of submission of a process to the time of completion is turn around time.

✓ i.e, it is the sum of periods spent:
* waiting to get into m/y.
* waiting in the ready queue.
* executing on the CPU
* doing I/o.

↳ **Waiting Time.**

✓ It is the amount of time that a process spends waiting in the ready queue.

✓ waiting time is the sum of periods spent waiting in the ready queue.

↳ Response Time

✓ Measure of time from time of submission of a request until the first response is produced.

✓ This measure is response time.
ie, it is the amount of time it takes to start responding, but not the time that it takes to output that response.

* Maximize → CPU utilization & throughput
Minimize → Turn Around time, waiting time & Response Time.

Scheduling Algorithms.

↳ Decide which of the process in the ready queue is to be allocated the CPU.

1. First Come First Served Scheduling (FCFS)

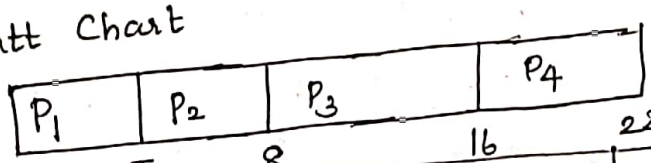
* The process that requests the CPU first is allocated the CPU first.
* Implementation of FIFO policy is easily managed with FIFO Queue.
* When CPU free, it is allocated to the process at the head of the queue.
* Running process is removed from queue.

Q1)

Process	Arrival Time	Burst Time
P_1	0	5
P_2	1	3
P_3	2	8
P_4	3	6

Soln:

Gnatt Chart



Process	A.T	B.T	Completion Time (CT)	(CT-AT) Turn Around Time (TAT)	(TAT-BT) Waiting Time
P_1	0	5	5	0	0
P_2	1	3	8	7	4
P_3	2	8	16	14	6
P_4	3	6	22	19	13
				45	23

TAT = CT − AT.

WT = TAT − BT

Total TAT = 45

Avg TAT = 45/4 = 11.2

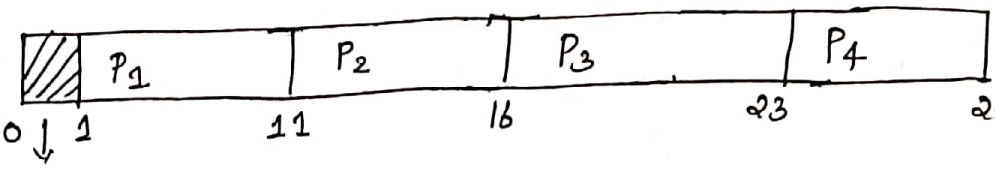
Total W.T = 23

Avg W.T = 23/4 = 5.7

Q2)

Process	Arrival Time	Burst Time
P_1	1	10
P_2	2	5
P_3	3	7
P_4	4	6

Soln:



Process	A.T	B.T	C.T	(CT-AT) T.A.T	(TAT-BT) W.T
P_1	1	10	11	10	0
P_2	2	5	16	14	9
P_3	3	7	23	20	13
P_4	4	6	29	25	19
				69	41

Total TAT = 69

Avg TAT = 69/4 = ____

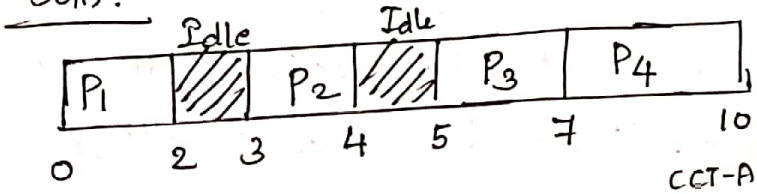
Total WT = 41

Avg WT = 41/4 = ____

Q3)

Process	Arrival Time	Burst Time
P_1	0	2
P_2	3	1
P_3	5	2
P_4	7	3

Soln:



Process	A.T	B.T	C.T	(CT-AT) TAT	(TAT-BT) W.T
P_1	0	2	2	2	0
P_2	3	1	4	1	0
P_3	5	2	7	2	0
P_4	7	3	10	3	0
				8	0

TAT = 8

Avg TAT = 8/4 = 2

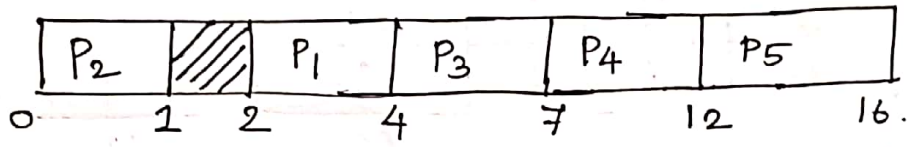
W.T = 0

Q4)

Process	A.T	B.T.
P1	2	2
P2	0	1
P3	2	3
P4	3	5
P5	4	4

Soln:

Gnatt Chart



Process	A.T	B.T	C.T	TAT	W.T.
P1	2	2	4	2	0
P2	0	1	1	1	0
P3	2	3	7	5.	2
P4	3	5	12	9	4
P5	4	4	16	12	8
				24	14

Avg TAT = 24/5 = 5.8

Avg W.T = 14/5 = 2.8

2. Shortest Job First Scheduling (SJF) - Non Preemptive

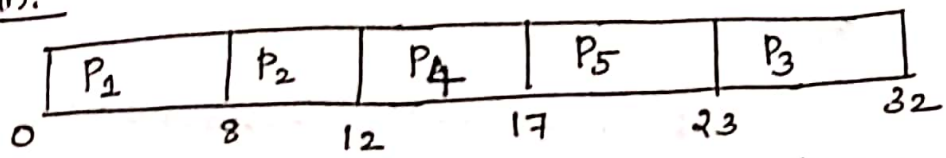
* Here when the CPU is available, it is assigned to the process that has the smallest next CPU burst.
* If two process have the same next CPU burst, FCFS scheduling is used.
* Burst Time.

Q1)

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5
P_5	4	6

P_2 → shortest but arrives only @ 1 so allocate P_1

Soln:



Process	A.T	B.T	C.T	TAT (CT-AT)	W.T (TAT-BT)
P_1	0	8	8	8	0
P_2	1	4	12	11	7
P_3	2	9	32	30	21
P_4	3	5	17	14	9
P_5	4	6	23	19	13
				82	50.

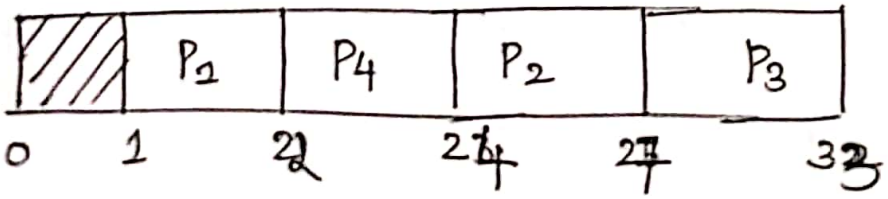
Avg TAT = 82/5 = 16.5

Avg WT = 50/5 = 10

Q2)

Process	AT	B.T.
P_1	1	21
P_2	2	3
P_3	3	6
P_4	4	2

Soln:



Process	AT	BT	C.T	TAT	WT
P_1	1	21	22	21	0
P_2	2	3	27	25	22
P_3	3	6	33	30	24
P_4	4	2	24	20	18
				96	64

Avg TAT = 96/4

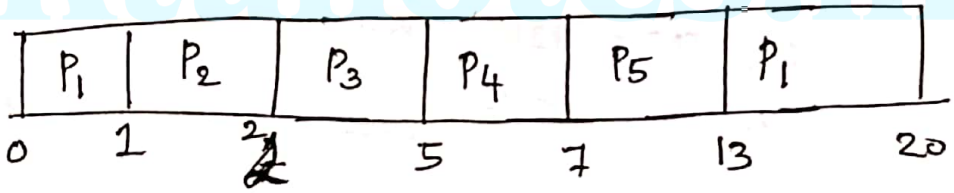
Avg WT = 64/4

Shortest Job first (SJF) – Preemption (Shortest Remaining Time first (SRTF))

(Q1)

Process	AT	BT
P_1	0	8 (7)
P_2	1	1 (0)
P_3	2	3 (2)
P_4	3	2
P_5	4	6

Soln:



Process	AT	BT	C.T	TAT.	W.T
P_1	0	8	20	20	12
P_2	1	1	2	1	0
P_3	2	3	5	3	0
P_4	3	2	7	4	2
P_5	4	6	13	9	3
				37	17

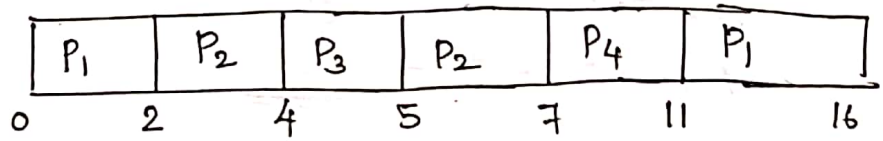
Avg TAT = 37/5

Avg WT = 17/5

(Q2)

Process	AT	B.T
P_1	0	7 (5)
P_2	2	4 (2)(0)
P_3	4	1 (0)
P_4	5	4

Soln:



Process	AT	BT	CT	TAT	WT
P_1	0	7	16	16	9
P_2	2	4	7	5	1
P_3	4	1	5	1	0
P_4	5.	4	11	6	2
				28	12

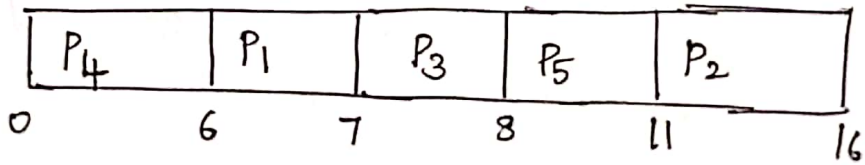
Avg TAT = 28/4

Avg WT = 12/4

(Q3) Solve using SJF

Process	AT	BT
P_1	2	1
P_2	1	5
P_3	4	1
P_4	0	6
P_5	2	3.

Soln:



Process	AT	BT	CT	TAT	WT
P_1	2	1	7	5	~~4~~ 4
P_2	1	5	16	15	10
P_3	4	1	8	4	~~3~~ 3
P_4	0	6	6	6	0
P_5	2	3	11	9	6
				39	23

Avg TAT = 39/5 = 7.8

Avg WT = 23/5 = 4.6

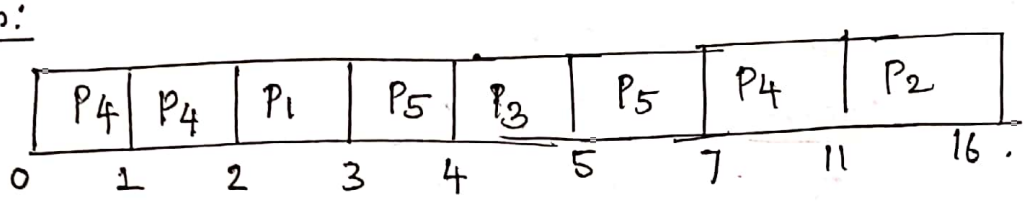
Q4) Solve using ~~SPSTF~~ SRTF

Process	AT	BT
P_1	2	1
P_2	1	5
P_3	4	1
P_4	0	6 (5)(4)
P_5	2	3

P_4 P_2 P_1 P_5 P_3

4 5 $\frac{3}{2}$ 1

Soln:



Process	AT	BT	CT	TAT	WT
P_1	2	1	3	1	0
P_2	1	5	16	15	10
P_3	4	1	5	1	0
P_4	0	6	11	11	5
P_5	2	3	7	5	2
				33	17

Avg TAT = 33/5 = 6.6

Avg WT = 17/5 = 3.4

3. Priority Scheduling

* CPU is allocated to the process with the highest priority.
* Equal priority processes are scheduled in FCFS order.
* ~~The~~ Low numbes represents high priority

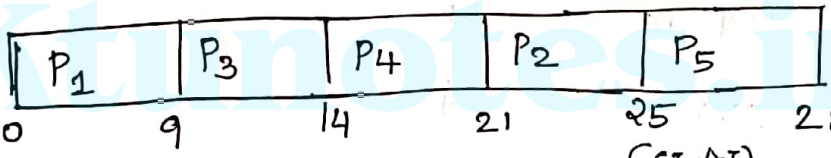
Non-Preemptive

Q1)

Priority	Process ID	Arrival Time	Burst Time
5	P_1	0	9
3	P_2	1	4
1	P_3	2	5
2	P_4	3	7
4	P_5	4	3

Soln:

Grantt Chart



Priority	Process	AT	BT	CT	TAT (CT-AT)	WT (TAT-BT)
5	P_1	0	9	9	9	0
3	P_2	1	4	25	24	20
1	P_3	2	5	14	12	7
2	P_4	3	7	21	18	11
4	P_5	4	3	28	24	21
			28		87	59

Avg TAT = 87/5

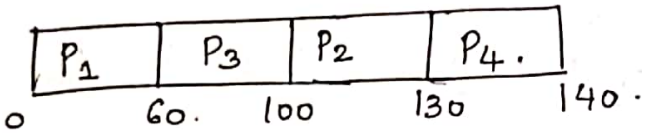
Avg WT = 59/5

Q2)

Priority	Process ID	AT	BT
3	P_1	0	60
2	P_2	3	30
1	P_3	4	40
4	P_4	9	10

Soln:

P_3, P_2, P_1, P_4



Priority	Process ID	AT	BT	CT	TAT	WT
3	P_1	0	60	60	60	0
2	P_2	3	30	130	127	97
1	P_3	4	40	100	96	56
4	P_4	9	10	140	131	121
			140			

Avg TAT = TAT/4

Avg WT = WT/4

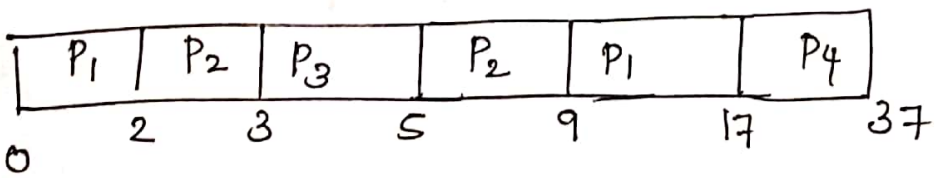
Preemptive

Q1.)

Priority	Process ID	AT	BT.
2	P_1	0	10 /8 /0
1	P_2	2	5 /4 /0
0	P_3	3	2 /0
3	P_4	5	20

Soln:

P_3, P_2, P_1, P_4



Priority	Process ID	AT	BT	CT	TAT	WT
2	P_1	0	10	17	17	7
1	P_2	2	5	9	7	2
0	P_3	3	2	5	2	0
3	P_4	5	20	37	32	12
			37		58	21

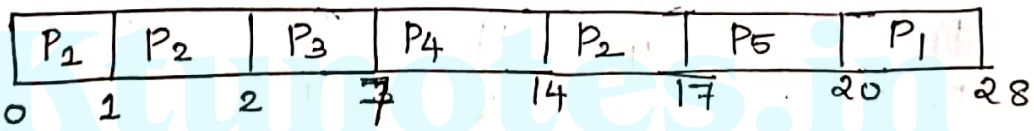
Avg TAT = 58/4

Avg WT = 21/4

Q2)

Priority	Process ID	AT	BT
5	P_1	0	9 |8|0
3	P_2	1	4 |3|0
1	P_3	2	5 |0
2	P_4	3	7 |0
4	P_5	4	3 |0

Soln: P_3, P_4, P_2, P_5, P_1



Priority	Process ID	AT	BT	CT	TAT	WT.
5	P_1	0	9	28	28	19
3	P_2	1	4	17	16	12
1	P_3	2	5	7	5	0
2	P_4	3	7	14	11	4
4	P_5	4	3	20	16	13
			28		76	48

Avg TAT = 76/5.

Avg WT = 48/5

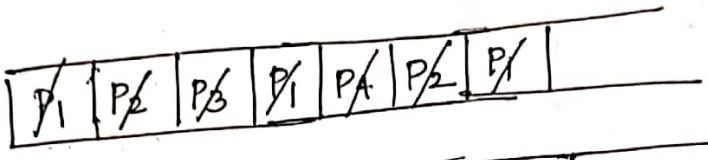
4. Round Robin. [Time Quantum + AT]

Q1) Time Quantum = 2.

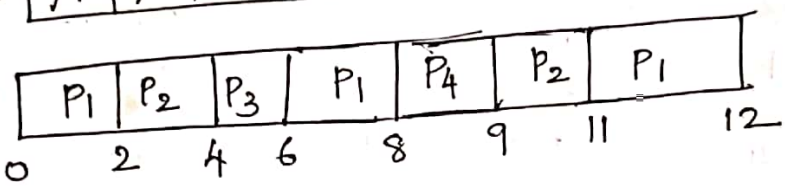
Process	AT	BT
P_1	0	5 /3/1
P_2	1	4 /2/0
P_3	2	2 /0
P_4	4	1 /0

Soln:

Ready Queue



Guant Chart



Process	AT	BT	CT	TAT	WT
P_1	0	5	12	12	7
P_2	1	4	11	10	6
P_3	2	2	6	4	2
P_4	4	1	9	5	4
					19

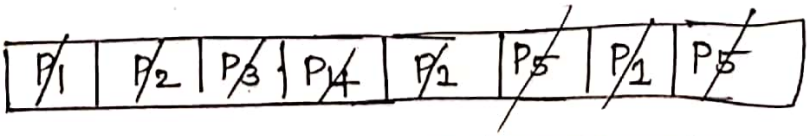
Avg WT = 19/4

Q2) ~~Priority~~ Time Quantum = 3.

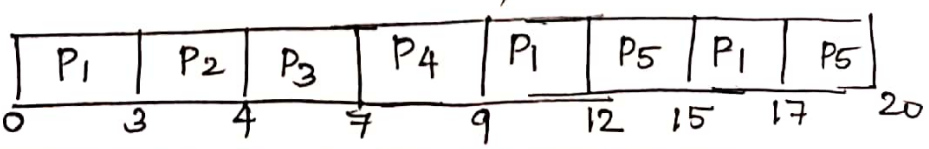
Process	AT	BT
P_1	0	8 /5/2/0.
P_2	1	1 /0
P_3	2	3 /0
P_4	3	2 /0.
P_5	4	6 /3/0.

Soln:

Ready Queue



Guant Chart



Process	AT	BT	CT	TAT	WT
P_1	0	8	17	17	9
P_2	1	1	4	3	2
P_3	2	3	7	5	2
P_4	3	2	9	6	4
P_5	4	6	20	16	10

Avg TAT = TAT/5

Avg WT = WT/5

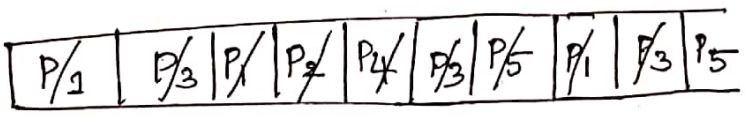
Q3) Time Quantum = 3.

Process	AT	BT
P_1	0	8 |5|2 |0
P_2	5	2 |0
P_3	1	7 |4|1 |0
P_4	6	3 |0
P_5	8	5 |2 |0

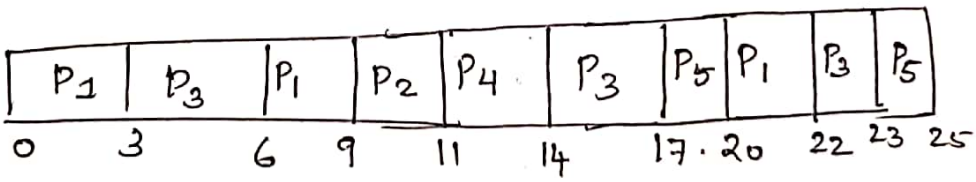
Soln:

newly arrived process - first

Ready Queue



Guantt Chart



Process	AT	BT	CT	TAT	WT
P_1	0	8	22	22	14
P_2	5	2	11	6	4
P_3	1	7	23	22	15
P_4	6	3	14	8	5
P_5	8	5	25	17	12
				75	50

TAT = 75/5 = 15 WT = 50/5 = 10