Operating System

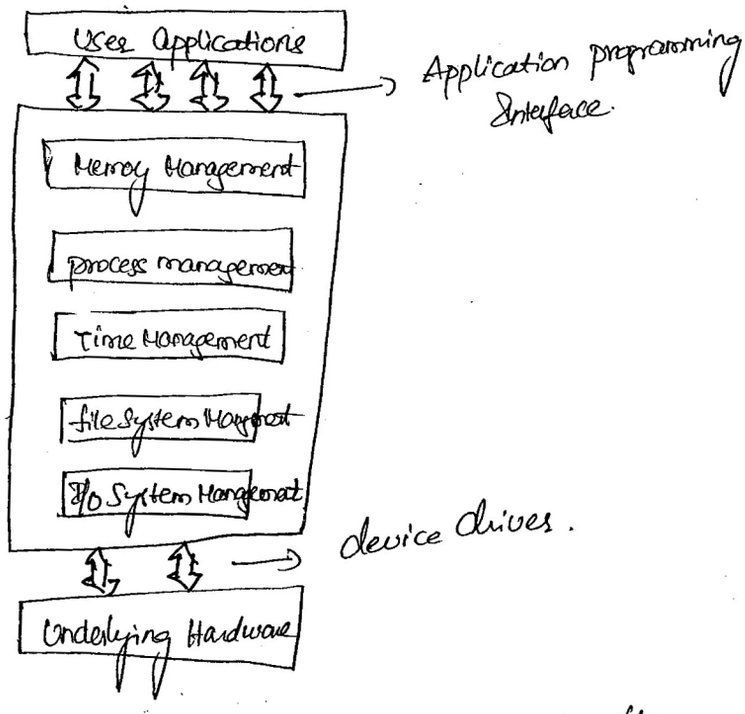
Basics:-

The Operating System act as a bridge b/w the User Applications/tasks And the Underlying System resources through a set of System functionalities And Services.

The 'OS' manages the System Resources and makes them Available to the User Applications on a need basis.

The Kernel:

The Kernel is the Core of the operating System And is responsible on managing the System resources And the Communication among the hardware And other System Services.



Process Management:-

The process management deals with managing the process/tasks. process/tasks includes setting up the memory space for the process, loading the process code in to memory space, scheduling And managing the execution of process.

Secondary Storage management:-

The Secondary storage management deals with managing the Secondary storage memory devices, if any connected to the system. Secondary memory is used as a backup medium for programs and data since the main memory is volatile. In most of systems, the Secondary storage is kept in hard disks. The Secondary storage management service of kernel deals with.

→ Disk storage allocation
→ disk scheduling (Time interval at which disk is accessed to be updated)
→ free space management

Protection Systems:-

Most of the modern operating systems are designed in such a way to support multiple users with different levels of accessing permissions (eg: windows 'XP' with user permissions like "Administrator", 'Restricted' etc)

The protection deals with implementing the security policies to restrict the access to both users and system resources. So, in multiuser supported os, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. This kind of protection is provided by the protection services running with the kernel.

Interrupt Handling:- kernel provides handles mechanism for all external/internal interrupts generated by the system. Depending upon the type of services a kernel may contain lesser number of services or more.

Kernel Space and User Space:-

The program code corresponding to the kernel applications/services are kept in a contiguous area of primary memory and is protected from unauthorized access by user programs/applications. The memory space at which kernel is located called as "kernel space". Similarly all user applications are located in specific area of primary memory and is referred as "user space". The partitioning of memory into kernel and user space is purely 'os' dependent.

File System management :- file is a collection of related information. A file could be a program (Source code or Executable), text files, image files, word documents, audio/video files, etc. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file management Service responsible for.

(a) The Creation, deletion and alteration of files.
(b) Creation, deletion and alteration of directories.
(c) Saving of files in Secondary storage memory (eg: Hard disk Storage)
(d) providing automatic allocation of file space based on the amount of free space available.
(e) providing flexible naming convention for the files.

I/O System (Device) Management :- Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the System. In a well structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided by a set of Application Programming Interfaces (APIs) Exposed by Kernel.

The kernel maintains a list of all I/O devices of the system. This list may be available in advance, at the time of building the kernel for some kernels, dynamically update the list when a new device is installed.

The kernel talks to the I/O device through a set of low level System calls, which are implemented in a Service called device drivers.

The Device manager is responsible for.

(a) Loading and Unloading device drivers.
(b) Exchanging information and the System Specific Control Signals to and from the device.

Task, process and threads

The term "task" refers to something that needs to be done. In our day-to-day life, we are bound to the execution of a number of tasks.

The task can be the one assigned by our managers or the one assigned by our professors/teachers or the one related to our personal or family needs. In addition, we will have an order of priority for executing these tasks. In the operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program.

Note :- Task is also known as "Job" in the operating system context. A program or part of it in execution is also called a "process". The terms "task", "job" and "process" refer to the same entity in the operating system context and most often they are used interchangeably.

Process :-

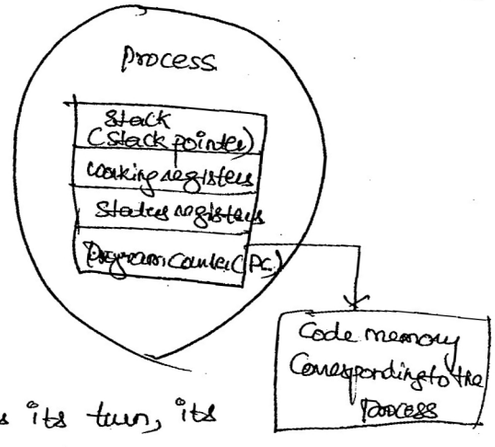
A "process is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. A process requires various system resources like 'cpu' for executing the process, memory for storing the code corresponding to the process, and data related variables, input-output information exchange, etc.

Structure of a process:-

The Concept of "process" leads to Concurrent execution of tasks and thereby the efficient utilisation of the 'cpu' and other system resources. Concurrent execution is achieved through the sharing of 'cpu' among the processes.

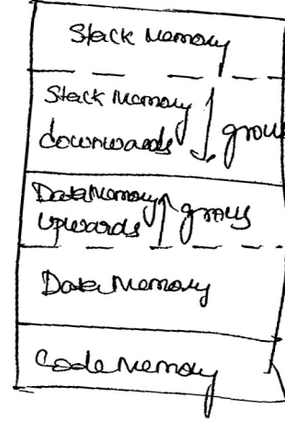
A process imitates a processor in properties and holds. (mimic)
a set of registers, process status, a program counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and code corresponding to the process.

Structure of process



When the process gets its turn, its registers and the program counter registers becomes mapped to the physical registers of the cpu. The memory occupied by the process is segregated into three regions, namely stack memory, Data memory and Code memory.

Memory organisation of process



The stack memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process. The code memory contains the program code (instructions) corresponding to the process.

On loading a process into main memory, a specific area of memory is allocated for the process. The stack memory usually starts at the highest memory address from the memory area allocated for the process.

Ex:- the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accomodate the variables local to the process.

Process States And State transition :-

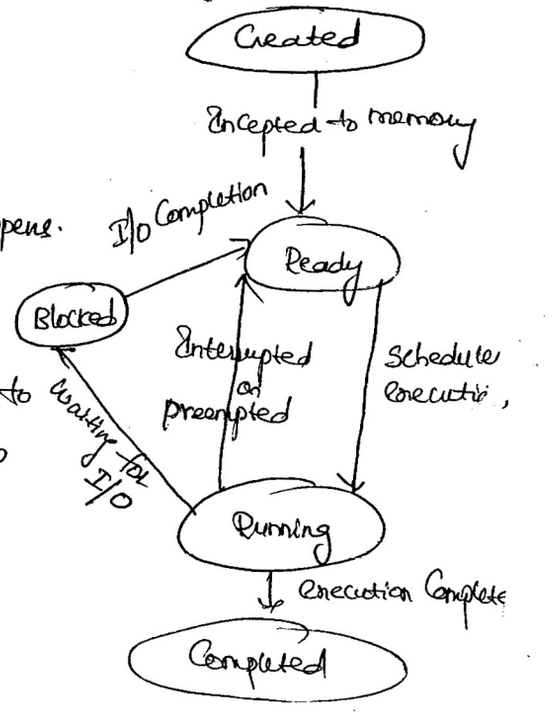
The process traverses through a series of states during (travels)
its transition from the newly created state to the terminated state.
The cycle through which a process changes its state from "newly created
to "execution completed" is known as "process life cycle". The state at
which the process is created is refered as "Created State". The
Operating System recognises a process in the created state. The
state, where a process is incepted into the memory and waiting
(entered)
the processor time for execution is known as "Ready State".

The state where the source code.
instructions corresponding to the process is
being executed is called "Running State".
↓
where execution happens.

Blocked State / wait State refers to a state where a
Running process is temporarily suspended from
execution and does not have immediate access to
resources. (The task waiting for user inputs to
unblock)

→ A state where the process completes its
execution is known as Completed State.

Process states and transitions

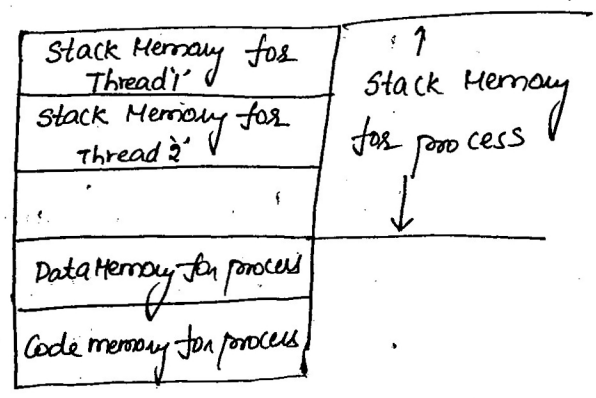


THREADS

→ A thread is a single sequential flow of control within a process. A thread is also known as light weight process. A process can have multiple threads of execution. The different threads which are part of a process, share the same address space, meaning they share the data memory, code memory.

→ The threads maintain their own thread status (CPU registers), program counter (PC) and stack.

Ex:- Memory organisation of a process and its associated threads.



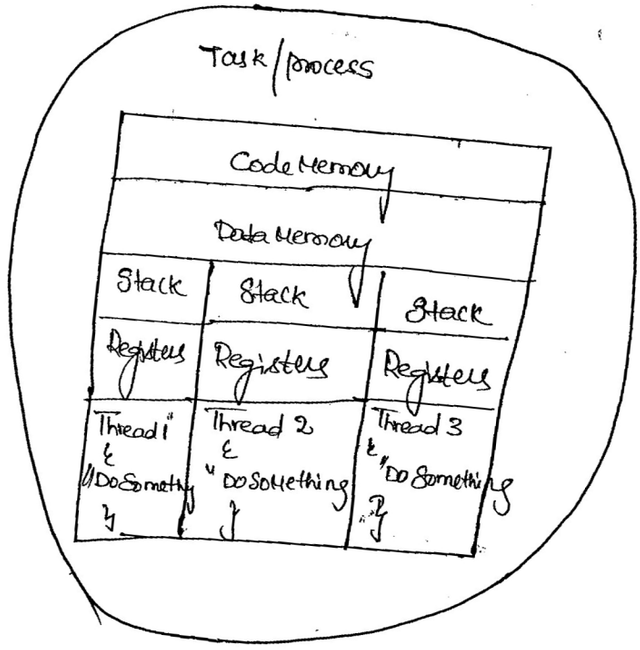
The Concept of Multithreading :-

A process in embedded application may be a complex or lengthy one. And it may contain various sub-operations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc. If all the subfunctions of task are executed in sequence, the CPU utilisation may not be efficient.

Ex:- If the process is waiting for user input, the CPU enters the wait state for the event, and the process execution also enters wait state.

Instead of this if the task/process is split into different threads carrying out the different subfunctionalities of the process, the cpu can be effectively utilised and when the thread corresponding to the I/o operation enters the wait state, another threads which do not require the I/o event for their operation can be switched in-to execution. This leads to more speedy execution of the process and the efficient utilisation of the processor time and resources.

Process with Multi-threads



If the process is split into multiple threads, which executes a part of the process, there will be a main thread and rest of the thread will be created within the main thread. Use of multiple threads to execute a process brings the following advantage.

(1) Better Memory utilisation. Multiple threads of the same process share the address space for data Memory.

(2) Since the process is split up into different threads, when one thread enters a wait state, the cpu can be utilised by other threads of the process that do not require the event. This speeds up the execution of the process

(8) Efficient cpu utilisation. The cpu is engaged all time. (9)

Thread Standards :- The Thread Standards deal with different Standards Available for thread Creation and management. These Standards are utilised by the operating systems for thread Creation and thread Management.

(1) POSIX threads :- posix Standards for portable operating System Enterface. It deals with Real-time Extensions.

(2) Win32 Threads :- Win32 thread are the threads Supported by Various flavours of windows operating System. The Win32 Application Program interface (Api's) libraries provide the Standard set of Win32 thread Creation and management functions.

Thread pre-emption :- Thread pre-emption is the Act of pre-empting the Currently Running thread (Stopping the Currently Running thread temporarily) Thread pre-emption Ability is solely dependent on the operating System. Thread pre-emption is performed for Sharing the cpu time among all the threads. The execution switching among threads are known as thread Context switching.

The thread falls into any one of the following types.

Userlevel thread :- User level thread do not have Kernel or Support and they exist in the Running process. Even if process Contains multiple user level threads, the OS treats it as Single thread and will not Switch the execution among different threads.

Note :- It is the Responsibility of process to Schedule each thread as and when Required.

Kernel level thread :-

Kernel level threads are individual units of execution, which the 'OS' treats as seperate threads. The 'OS' interrupts the execution of currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by 'OS'.

Many to one Model :- Here many user level threads are mapped to a single thread. In this model, the kernel treats all user level threads as single thread and the execution switching among the user level threads happen when a currently executing user level thread voluntarily blocks itself or relinquishes its CPU.

Threads V/S process

Thread	Process
(1) Thread is a single unit of execution and is part of process.	(1) Process is a program in execution and contain one or more threads.
(2) A thread does not have its own data memory. It shares the data memory and heap memory with other threads of same process.	(2) process has its own code memory, data memory and stack memory.
(3) A thread cannot live independently, it lives within the process.	(3) A process contains atleast one thread.
(4) There can be multiple threads in the process. The first thread occupies the start of the stack memory of process	(4) Threads within the process share the code, data and heap memory. each thread holds seperate memory area for the stack.
(5) Threads are very inexpensive to create	(5) process are very expen to create.
(6) Context switching is inexpensive and fast.	(6) Context switching is complex and it is comparatively slower.
(7) If a thread expires, the stack is reclaimed by the process	(7) By default a process has one thread and when that thread dies, process dies also.

Multiprocessing and Multitasking

In operating system Context multiprocessing describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing are known as multiprocessor systems. Multiprocessor systems possess multiple CPUs and can execute multiple processes simultaneously.

The ability of the operating system to have multiple programs in memory, which are ready for execution, is refered as multiprogramming. In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiprocesses by switching the execution among different processes.

Note :- The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as multitasking.

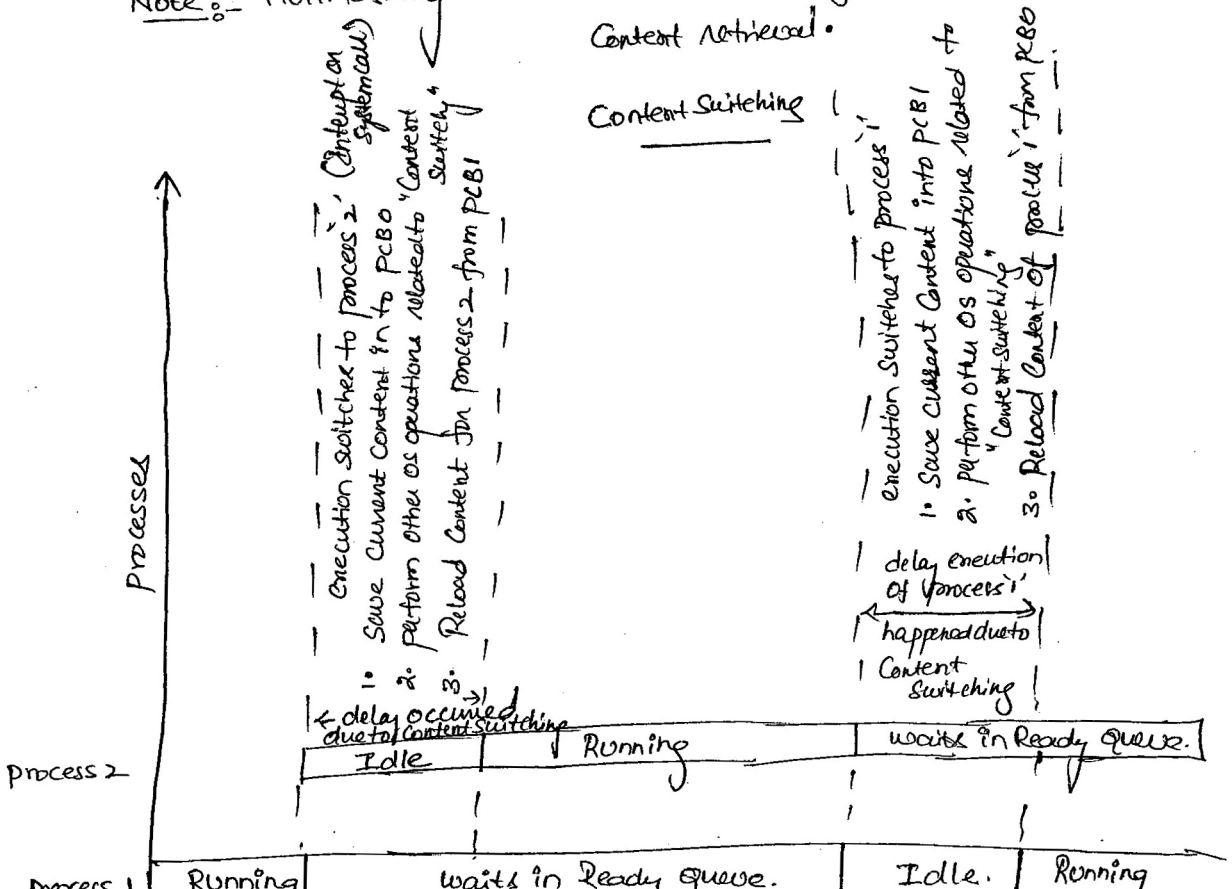
Multitasking involve the switching of CPU from executing one task to another. Whenever a 'cpu' switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to 'cpu' switching.

→ The act of switching 'CPU' among the processes or changing the current execution context is known as "Context switching".

→ The act of saving the current context which contains the context details (Register details, memory details, execution details, resource details etc) for the currently running process at the time of CPU switch is "Context saving".

The process of Retrieving the Saved Content details for a process which is going to be executed due to 'cpu' switching known as "Content retrieval"

Note :- Multitasking involves "Content switching, Content saving and Content retrieval.



Types of Multitasking :- (Multitasking can be classified in to different types)

(1) Co-operative Multitasking :- Here the process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU (releases)

In this method the task can hold the 'cpu' as much time as it wants

Note :- (1) This type of implementation depends on mercy of the tasks each other for getting the cpu time for execution, it is known as Co-operative multitasking.

(2) If the current executing task is non-cooperative then other task has to wait for long time to get the cpu.

(2) Preemptive Multitasking :-

→ Preemptive multitasking ensures that every task gets a chance to execute. When and how much time a process gets is dependent on the implementation of scheduling.

→ As the name indicates, preemptive scheduling Multitasking, the currently running task is preempted to give a chance to other tasks to execute. (stopped)

Note :- The preemption may be based on timeslots and task/process priority.

(3) Non-preemptive Multitasking :-

In non-preemptive multitasking, the process/task, which is currently given the 'cpu' time is allowed to execute until it terminates (enters into Completed State) or enters the blocked/wait state, waiting for an I/O or System Resource).

In Non-preemptive multitasking the currently executing task releases the 'cpu' when it waits for an I/O or System resource or an event to occur.

Task Scheduling

Multitasking involves the execution switching among the different tasks. The ability of an operating system to hold multiple processes in a memory and switch the CPU from executing one process to another process is known as Multitasking.

Note :- There should be some mechanism in place to share the 'cpu' among different tasks and to decide which 'task'/'process' is to be executed at a given point of time

Scheduling :- Determining which task/process is to be executed at a given point of time is known as task/process scheduling. The kernel service which implements the scheduling algorithm is known as 'scheduler'.

The selection of scheduling algorithm should consider the following factors.

CPU Utilisation :- The scheduling algorithm should always make the 'cpu' utilisation high. 'cpu' utilisation is a direct measure of how much percentage of 'cpu' is being utilised.

Throughput : This gives an indication of number of tasks/processes executed per unit of time. The throughput for a good scheduler should always be higher.

Turn Around time :- It is the amount of time taken by a process for completing its execution. It includes the time spent by process/task in ready queue, and time spent in executi It should be minimum for good scheduling algorithm.

Waiting time :- It is the amount of time spent by process/task in 'Ready' queue. waiting to get the 'cpu' time for execution. The waiting time should be. minimal for good scheduling algorithm.

The various queues maintained by 'os' in association with 'cpu' scheduling are:

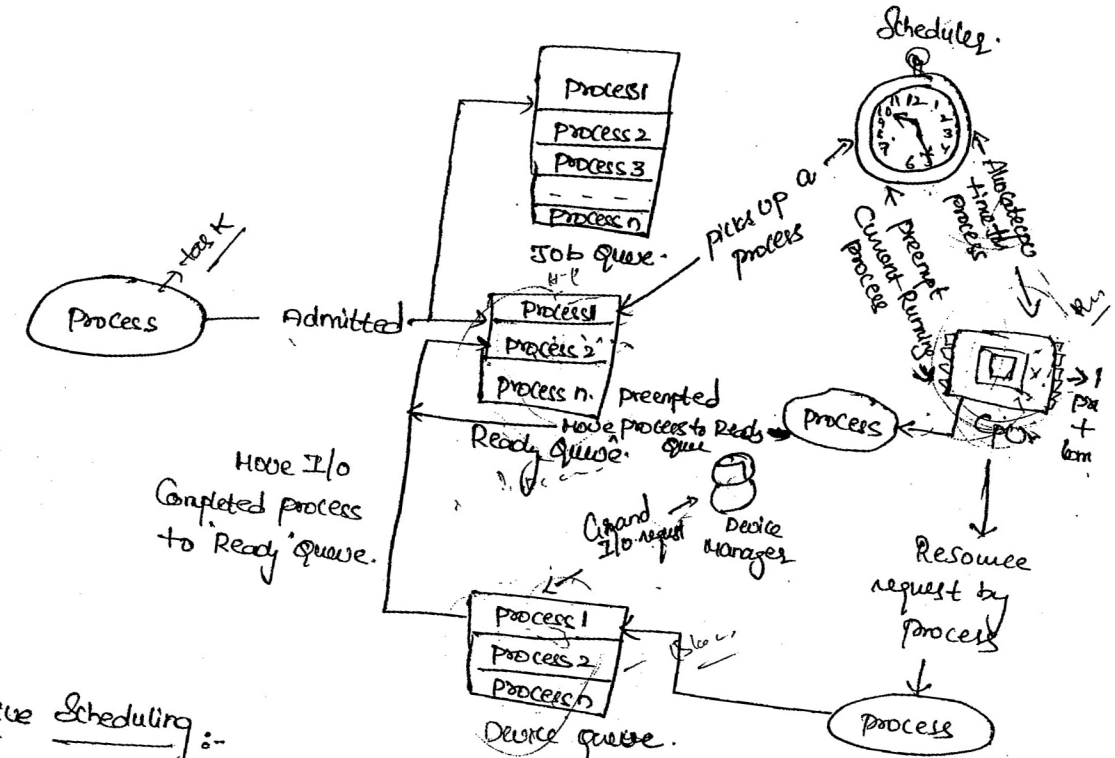
Job queue :- Job queue contains all the processes/tasks in the system.

Ready Queue :- Contains all the tasks/processes which are ready for execution and waiting for 'cpu' to get their turn for execution. This queue is empty when there is no process/task ready for running.

Device Queue :- It contains the set of processes/tasks which are waiting for an I/o device.

A process migrates through all these queues during its journey from 'admitted' to 'Completed' Stage. The diagram represents the transition of process through various queues.

The transition through various queues:-



Non-preemptive Scheduling:-

In this Scheduling type, the Currently Executing task/process is Allowed to Run Until it terminates or enters the 'wait' state. The various types of non-preemptive Scheduling are.

(1) FCFS (First Come first Serve Scheduling):- It Allocates cpu time to the task based on the order in which they enter in to 'Ready' Queue. The first entered task is Served first (i,e) the task which is put first in Ready Queue is Served first.

Ex:- Ticked reservation system where people need to stand in Queue and the first person standing in Queue is Serviced first.

(2) Last Come first Served Scheduling (LCFS):- It allocates Cpu time to the task/process based on the order in which they are entered in the Ready Queue. The last entered task is Serviced first. (i,e) process which put last in Ready Queue is Served first.

(3) Shortest Job first Scheduling (SJF):-

(13) (18)

[In 'SJF', the process with the shorest estimated Run-time is Scheduled first, followed by next Shortest process, And so on.] This Algorithm Sorts the 'Ready Queue' each time a process/task relinquishes the 'Cpu' to pick the proce with least (Shortest) estimated Completion/Run time.

Priority Based Scheduling:-

[Priority Based non-preemptive scheduling algorithm ensures that a proce with high priority is Serviced at the ealiest Compared to other low priority Proces in 'Ready' queue.] The Shortest Job first (SJF) algorithm Can be viewed as a priority based Scheduling where each task is prioritised in the Order of time required to Complet the task. The lower the time required for Completing a process the higher is its priorit in SJF algorithm. The priority is a number ranging from '0' to maximum priority Supported by OS.

Ex:- Windows 'CE' Supports 256 levels of priority (i.e '0' indicates highest priority and 255 indicates the lowest priority)

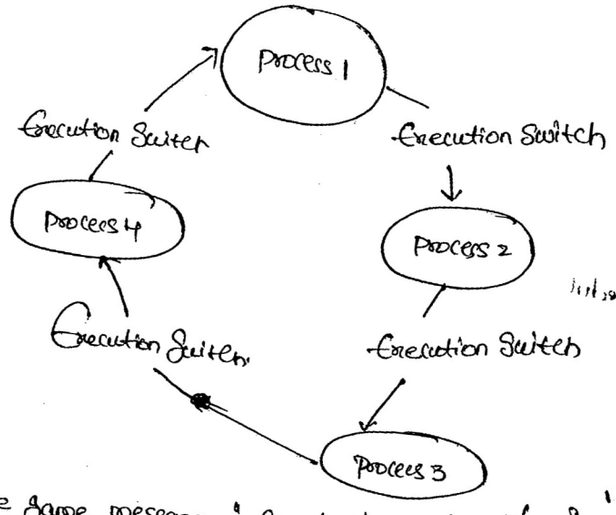
PRE-emptive SCHEDULING:-

In preemptive Scheduling, every task in the 'Ready Queue' gets a Chance to Execute. In this Scheduling the Scheduler Can prempt (Stop temporarily) the Currently Executing task/process And Select Another task from the 'Ready' Queue for Execution. A 'task' which is preempted by the Scheduler is moved to the 'Ready' Queue.

The Various types of pre-emptive Scheduling:-

Premptive (SJF) Scheduling:- This Scheduling Sorts the 'Ready' queue. When a new process enters the 'Ready' queue And Checks whether the Execution-time of the new process is Shorter than the remaining of the total estimated time for the Currently Executing Process/task.
If the Execution time of new process is less, the Currently Executing process is prempted And the new process is Scheduled for Execution.

Round Robin Scheduling :-



'Round Robin' brings the same message 'Equal chance to all'. In 'Round Robin' Scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot. The execution starts with picking up the first process in 'Ready' queue. It is executed for a pre-defined time and when the pre-defined time elapses or the process completes the next process in the 'Ready' queue is selected for execution. Once each process in the 'Ready queue' is executed for pre-defined time period, the Scheduler comes back and picks the first process in the 'Ready' queue for execution & the same sequence is repeated.

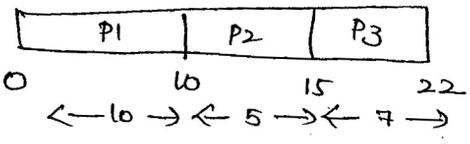
The set of functions that the 'RTX51' kernel provides for managing the tasks :-

Task State	State Descriptions
Running	The task that is currently running is in Running state. Only one task at a time may be in this state. The os_running_task_id kernel call returns the task number (ID) of current executing task.
Ready	Tasks which are ready to run are in the Ready state. A task may be made ready immediately by setting its ready flag using the os_set_ready function.
Waiting	Tasks which are waiting for an event are in waiting state. Once the event occurs, that task is switched to Ready state. The os_wait function is used for placing a task in waiting state.

Priority Based Scheduling :- In preemptive priority scheduling any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in non-preemptive scheduling the high priority process entering the 'Ready' queue is scheduled only after the current executing process completes its execution.

Ex:1 Three processes with process IDS P1, P2, P3 with estimated Completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turnaround time (TAT) for each process and the Average waiting time and Turn Around time.

The Sequence of execution of the processes by the cpu is represented as



Sol:-

Assume the cpu is readily available at the time of Arival of P1, P1 starts executing without any waiting in the Ready Queue. Hence the waiting time for P1 is zero. The waiting time for all processes are given as.

waiting time for P1 = 0ms (P1 starts executing first)

waiting time for P2 = 10ms (P2 starts executing after Completing P1)

waiting time for P3 = 15ms (P3 starts executing after Completing P1 and P2)

Average waiting time = (waiting time for all processes) / No of processes.

= (waiting time for (P1+P2+P3)) / 3

= (0+10+15)/3 = 25/3 = 8.33 milliseconds

Turn Around time (TAT) for P1 = 10ms (Time Spent in Ready Queue + Execution time)

" " for P2 = 15ms (-Do-)

" " for P3 = 22ms (-Do-)

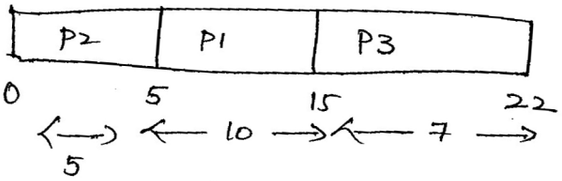
Average Turn Around time = (Turn around time for all processes) / No of processes

= (10+15+22)/3 = 15.66 milliseconds.

Ex: 2

Calculate the waiting time and (TAT) for each process and the average waiting time and Turn Around time (TAT) for the above example ① if the process enter the "Ready" queue together in the order P2, P1 and P3.

The Sequence of execution of processes by cpu is represented as



Assume the 'cpu' is readily available at the time of arrival of P2, P2 starts executing without any waiting in the "Ready" queue.

Hence waiting time for P2 is zero.

waiting time for P2 = 0ms (P2 starts executing first)

" " P1 = 5ms (P1 starts executing after completing P2)

" " P3 = 15ms (P3 starts executing after completing P2 and P1)

Average waiting time = waiting time for (P2+P1+P3)/3

= (0+5+15)/3 = 6.66 milliseconds.

Turn around time (TAT) for P2 = 5ms (Time spent in Ready qu execution time)

" for P1 = 15ms (-Do-)

" for P3 = 22ms (-Do-)

Average turn around time = (Turn around time for (P2+P1+P3))/3

= (5+15+22)/3

= 14 milliseconds.

Ex: 3

Three processes with process ID's P1, P2, P3 with estimated Completion time 6, 4, 2 m.seconds respectively, enters the ready queue together in the order P1, P2 and P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around time in RR algorithm with time slice = 2ms.

Solution:-

The scheduler sorts the Ready Queue based on the FCFS policy and picks up the first process 'P1' from the Ready queue and execute it for the time slice '2ms'. When the time slice is expired, 'P1' is preempted and 'P2' is scheduled for execution. The time slice expires after 2ms of execution of 'P2'. Now 'P2' is preempted and 'P3' is picked up for execution. 'P3' completes its execution within the timeslice and the scheduler picks 'P1' again for execution for the next timeslice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as

P1	P2	P3	P1	P2	P1
0 – 2	2 – 4	4 – 6	6 – 8	8 – 10	10 – 12
←2→	←2→	←2→	←2→	←2→	←2→

The waiting time for all the processes given as.

waiting time for P1 = ('P1' starts executing first and waits for two timeslices to get execution back)

⇒ 2ms + 2ms + 2ms = 6ms

waiting time for P2 = (P2 starts executing after completion of 'P1' and waits for two timeslices to get the cpu time)

= (2ms + 2ms + 2ms) = 6ms

waiting time for P3 = (P3 starts executing after completing the first time slices for 'P1' and 'P2' and completes its execution in a single time slice)

= 2ms + 2ms + 2ms

W.T for P3 = 4ms.

Average waiting time = (waiting time for all the processes) / No of process

= (6 + 6 + 4)ms / 3 = 5.33 milliseconds.//

Turn around time (TAT) for P1 = 12ms (Time Spent in Ready queue + execution time)

=> 6ms + 6ms = 12ms

Turn around time (TAT) for P2 = Time spent in Ready queue + execution tim

=> 6ms + 4ms => 10ms

Turn around time (TAT) for P3 = Time spent in Ready queu + execution tim

=> 4ms + 2ms => 6ms

Average Turn around time = (Time around time for all processes) / No of processes

= (12ms + 10ms + 6ms) / 3 = 9.33 milliseconds

Note :- 'RR' Scheduling involves lot of overhead in maintaining the time slice information for every process which is currently being executed.//

Putting them Altogether

→ The first piece of code Represents a process (process1) with normal priority And it performs a task which Requires 7.5 Units of Execution time. After performing this task, the process sleeps for 17.5 units of execution time And this is Repeated forever.

→ The Second piece of Code Represents a process (process 2) with priority above normal and it performs a task which requires 10 units of Execution time. After performing this task, the process sleeps for 5 units of execution time And this is Repeated forever.

Note:- process '2' is of high priority Compared to process '1' as this priority is above Normal.

Now Examine what happens if these processes are Executed on a Real-time kernel, with pre-emptive priority based Scheduling policy.

Imagine process '1' and process '2' are ready for execution. Both of them enters the 'Ready' queue and the Scheduler picks up process '2' for execution Since it is of higher priority Compared to process '1'. process '2' Starts Executing and runs until it executes the Sleep Instruction (i.e after 10 units of Execution time). when the sleep instruction is executed, process '2' enters the wait state, Since process '1' is waiting for its turn in the 'Ready' queue the Scheduler picks up it for execution, Resulting in Context Switch. The process Control Block (PCB) for process '2' is updated with the values of program Counter (PC), Stack pointer (SP) etc, at the time of Context Switch.

The estimated task execution time for process '1' is 7.5 units of execution time and the sleeping time for process '2' is 5 units of execution. After 5 units of execution time, process '2' enters the 'Ready State' and moves to the 'Ready Queue'. Since it is of higher priority compared to running process, the running process is preempted and process '2' is scheduled for execution. process '1' is moved to 'Ready' Queue, resulting in Context Switching.

Now the PCB of process '1' is updated with the current values of program counter and stack pointer (sp) etc. Here this Context Switching happens b/w process '1' and process '2'.

⇒ In order to keep the 'cpu' always busy, the scheduler runs a dummy task called IDLE process (task). This task executes some dummy task and keeps the cpu engaged.

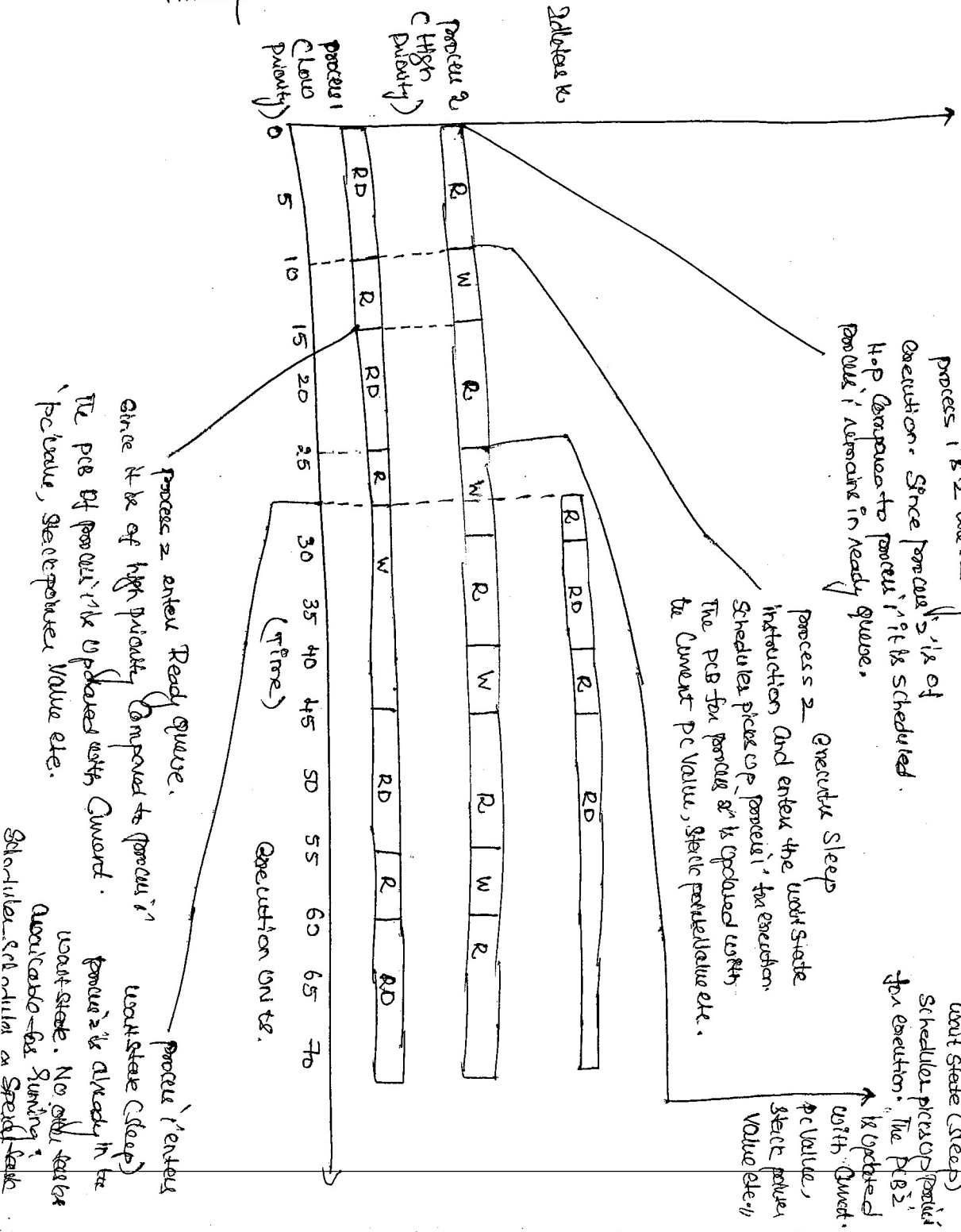
The implementation of the IDle process (task) is dependent on the kernel. It is simply an endless loop.

```
Void Idle_process (Void)
{
// Simply wait...
// Do nothing....
while (1);
}
```

The Real time kernel deployed in embedded systems, where the operating power is a big constraint (like systems which are battery powered) the idle task is used for putting the 'cpu' into Idle mode for Saving Power.

Ex:- RTX51 Tiny Real time Kernel, where the IDle task sets the 8051 'cpu' to idle mode, a power saving mode. In this mode, the program execution is halted and all peripherals and the interrupt system continues its operation.

Once 'cpu' is put into 'Idle mode', it comes out of this mode. when an interrupt occurs or when the RTX51 Tiny Timer tick interrupt occurs.



Task Communication

In a Multitasking System Multiple tasks Run Concurrently And each process may or may not interact between. Based on the degree of Interaction the process Running on 'OS' are Classified as

(1) Co-operating process :- In this Co-operative interaction Model one process requires the inputs from other processes to Complete its execution.

(2) Competing process :- The Competing process do not share anything among themselves but they share the system resources such as file, display device etc.

Co-operative processes exchanges information And Communicate through the following methods.

Co-operative through Sharing :- The Co-operative process exchange data through some shared resources.

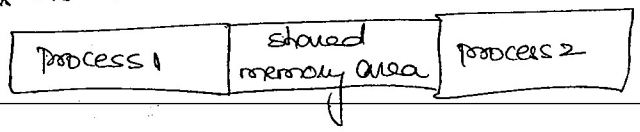
Co-operation through Communication :- No data is shared b/w the processes. But they Communicate for synchronisation.

The mechanism through which processes Communicate each other. is known as Interprocess Communication (Ipc). The various types of Interprocess Communication (IPC) mechanisms adopted by process are. Kernel dependent.

Some of the 'Ipc' are explained as :-

I (1) Shared Memory :-

Process share some area of the memory to Communicate Among them. Information to be Communicated by the process is written to the shared memory area. other process which requires this information can read from that shared memory area.



The Implementation of Shared Memory Concept is kernel dependant. A few among them are

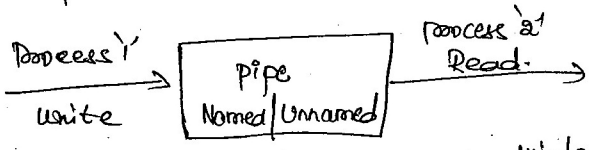
(a) Pipes :-

→ Pipe is a section of the shared memory used by processes for Communicating. Pipes follows the client-server architecture. A process which creates a pipe is known as pipe server and a process which connects to a pipe is known as pipe client.

→ The pipes may be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow.

Ex:- A unidirectional pipe allows the process connecting at one end of the pipe to write to pipe and the process connected at the other end of the pipe to read the data whereas bidirectional pipe allows both reading and writing at one end.

The pipe can be visualized as

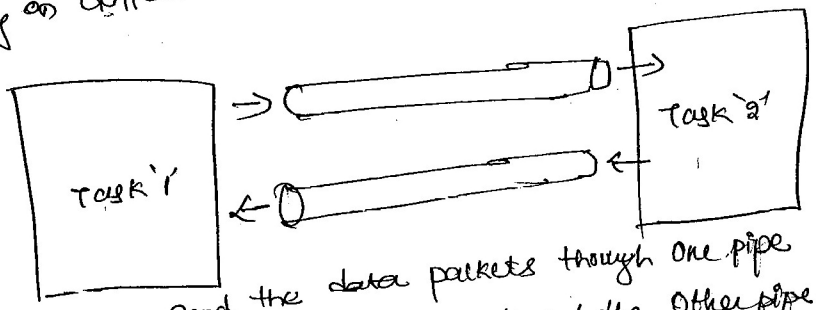


The implementation of pipe is OS dependant. Windows Desktop OS's supports two types of 'pipes' for interprocess communication, they are.

(1) Anonymous pipe :- These pipes are unnamed, unidirectional pipes used for data transfer between two processes.

(2) Named pipes :- Named pipe is a named, unidirectional or bidirectional pipe for data exchange b/w processes. Named pipes can be used for Communicating b/w processes running on the same machine or b/w processes running on different machines connected to network.

Ex:-



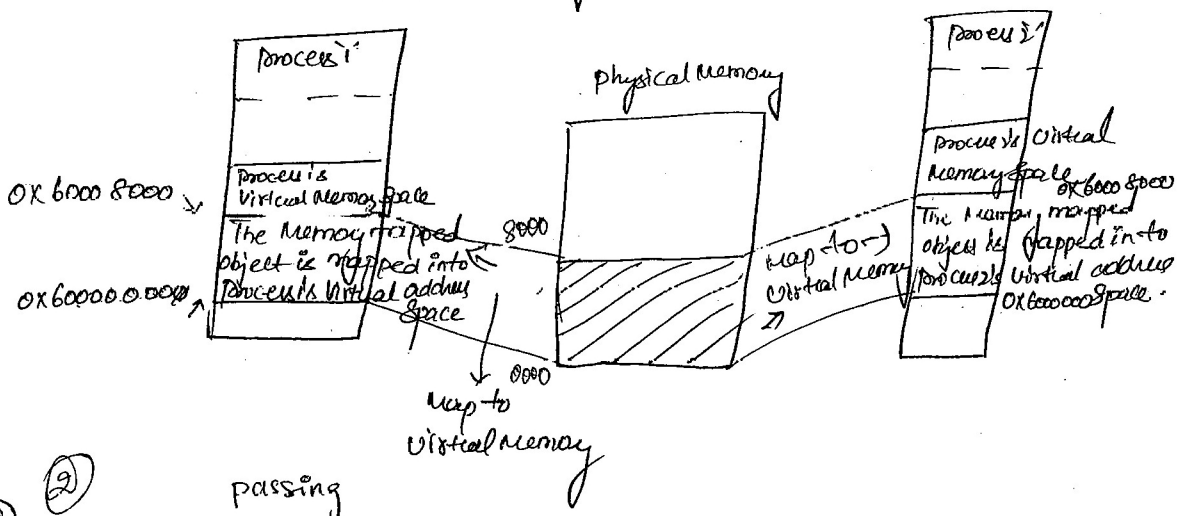
Note :- one task may send the data packets through one pipe and the other task may send Acknowledgements through the other pipe.

(b) Memory Mapped objects :-

It is a shared memory technique adopted by certain 'RTOS' for allocating a shared block of memory which can be accessed by multiple process simultaneously.

In this approach a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area to its virtual address space. All read and write operation to this virtual address space by process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for data sharing.

Concept of memory mapped object.



(2) Message passing :-

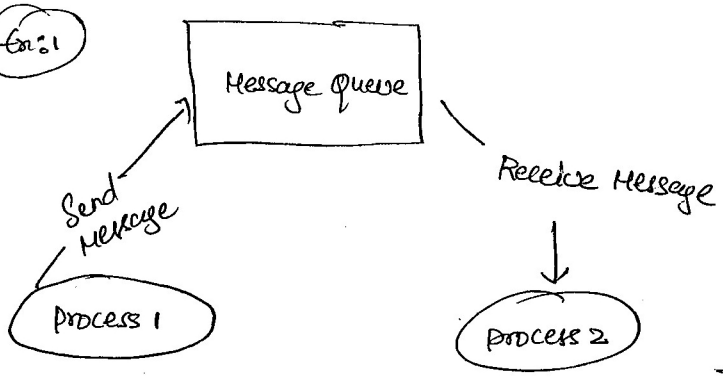
Note :- Message passing is an information exchange mechanism used for Interprocess Communication. The major difference b/w shared memory and messaging passing technique is that, through shared memory lots of data can be shared whereas only limited Amount of data is shared through messaging passing. Also message passing is relatively fast and free from the overheads compared to shared memory. The Message passing is classified into

(a) Message queue :- Usualy the process which wants to talk to another process posts the message to (FIFO) queue called 'message queue'

first in first out

The Message Queue Stores the message temporarily to pass it to the destined process. The Messages are sent and received through Send and Relcive Command.

Ex:1



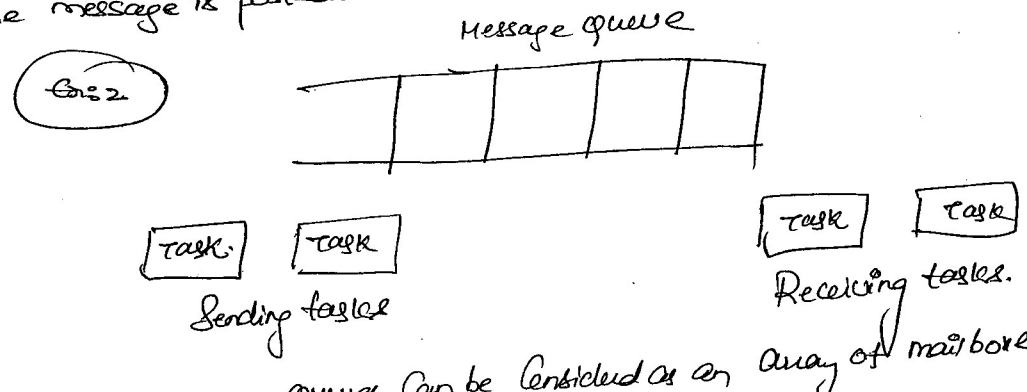
The Messages are exchanged through message queue. The implementation of Message queue is Kernel dependent. The kernel picks up the message from the message queue one at a time and examines the message for finding the destination process and then past the message to the message queue of the Corresponding process.

The Messaging Mechanism is Clautified in to Synchronous and asynchronous based on the behaviours of message posting thread/process/subprogram/program

→ In asynchronous messaging, the message posted process just past the message to the queue and it will not wait for an acceptance from the process to which message is posted.

→ In synchronous Messaging the process which past a message enters waiting state and waits for the message result from the process to which the message is posted.

Ex:2



Note:- Message queues can be Considered as an array of mailboxes.

Some of Applications of Musge Queues are

1. Taking the input from keyboard
2. To display output
3. Reading voltages from Sensors etc.
4. Data packet transmission in a N/w.

application, the highest priority task or the first in the queue can take the message.

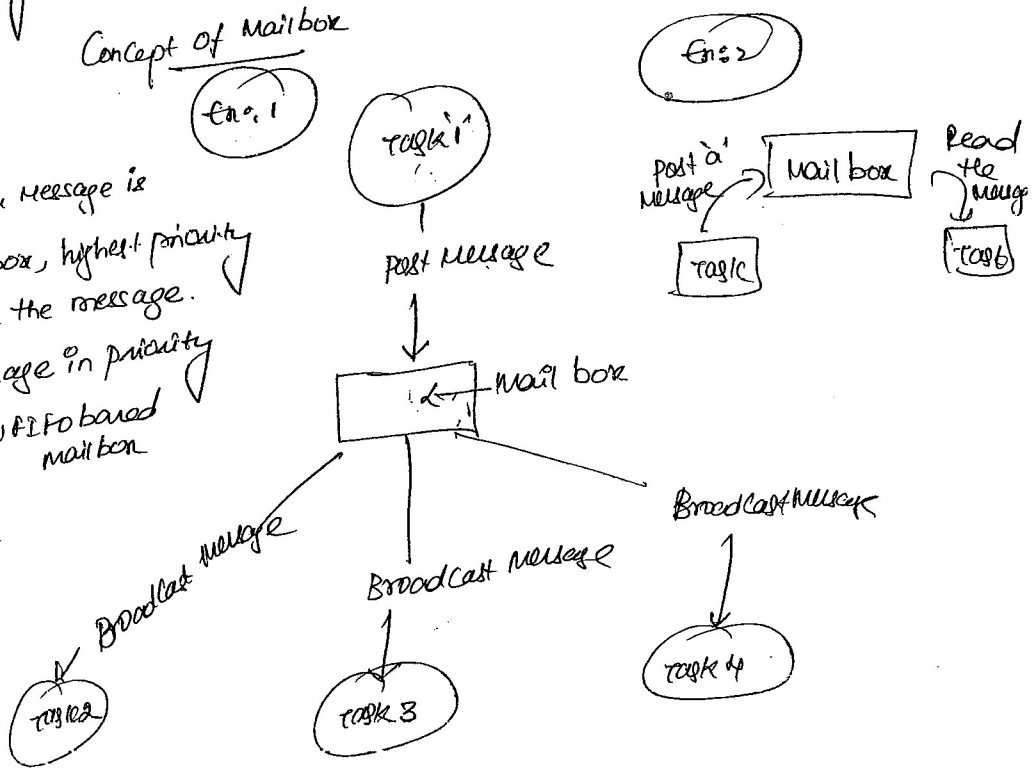
time of creating a queue, the queue is given a name, size (length), sending task waiting list and receiving task waiting list.

Mailbox :-

Mailbox is an alternate form of "message queue" and it is used in certain Real-time operating systems for IPC. It is usually one way messaging. The task which wants to send a message to other task creates a mailbox for posting the messages. The processes which are interested in receiving the messages posted to the mailbox can subscribe to the mailbox. The process which creates the mailbox known as mailbox server and the one who posts the message to the mailbox is called mailbox client. The mailbox creation, reading and writing are achived through kernel.

Note :- Mailbox and Message queue are same in functionality the only difference is in number of messages supported by them. Mailbox is used for exchanging a single message b/w two tasks or b/w Interrupt Service Routine and task.

Concept of Mailbox

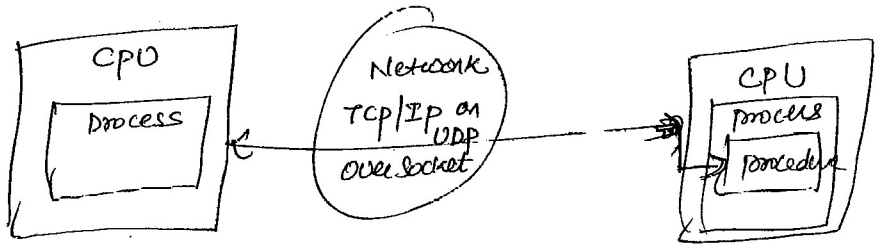


Note :- when a message is sent to the mailbox, highest priority task waiting for the message is given the message in priority based mailbox (or) FIFO based mailbox

(3) Remote procedure call (Rpc) :-

Remote procedure call is the interprocess Commun. used by a process to call a procedure of another proces the same 'cpu' or on a different 'cpu' which is inter Con. Network. The 'cpu' which initiates the Rpc request is known as Client. It is possible to implement Rpc Communication with different invocation interfaces. The Rpc can be Synchronous (Blocking) or Asynchronous (Non-blocking).

In Synchronous the process which calls the remote procedure is blocked until it receives a response back from other procedure. In Asynchronous Rpc calls, the calling process continues its execution while the remote process performs the execution of procedure. The result is returned back to the caller through mechanism called callback functions.



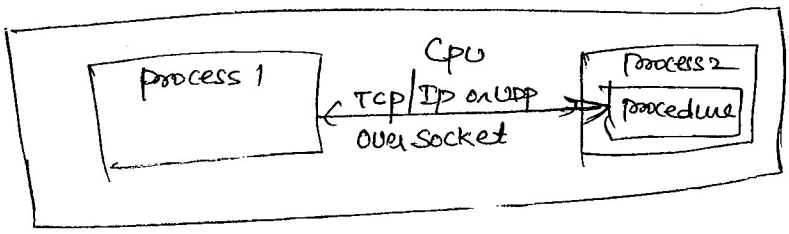
process running on different cpu's which are Networked.

Note:2

TCP along with IP to send data in the form of message units b/w Computers over the Internet.

IP → takes care of handling the actual delivery of data.

TCP takes care of keeping track of individual units (packets) for efficient routing through Internet.



process running on same CPU

Note:1

TCP → Transmission Control Protocol provides reliable, ordered & error checked delivery of data b/w processes running on different CPU

N/W :→ A tele Communication N/W that allows Computers to exchange data. (A group of two or more Computers linked together)

The Connection b/w them are established using either Cable or wireless Media.

Socket is a two way Communication link b/w two processes running on the N/W.

Task Synchronisation

In a multitasking environment, multiple processes run concurrently and share the system resource. Each process has its own boundary wall and they communicate with each other with different Ipc mechanisms including shared memory and variables. Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this. what could be the result in these scenarios, obviously unexpected results. How these issues can be addressed?

Note:- The solution is, make each process aware of the access of a shared resource either directly or indirectly. The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as Task/process synchronisation.

The major task communication synchronisation issues observed in multitasking and the commonly adopted techniques to overcome these issues are.

(1) priority inversion issue
(2) priority Inheritance issue.
(3) priority Ceiling

Priority inversion problem :-

"Priority inversion" is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task. Priority based preemptive Shedduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronisation mechanism (like mutex semaphore, etc) ensures that a process will not access a shared resource, which is currently in use by another Process.

The Synchronisation technique is only interested in avoiding conflicts that may arise due to the concurrent access of the shared resources not at all bothered about the priority of the process which tries to access the shared resource.

Let Process A, Process B and Process C be three Processes with priorities High, Medium and low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronised through a mutual exclusion mechanism like Binary semaphore S. Imagine a situation where Process C is ready and is picked up for execution by the Shedduler and 'Process tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing shared variable 'X'. Immediately after 'Process C' acquires the 'semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A is of higher priority than 'Process B', 'Process B' is preempted and 'Process A' is shedduled for execution. 'Process A' involves accessing of shared variables 'X' which is currently being accessed by 'Process C'. 'Process A' will not be able to access it. Thus 'Process A' is put into blocked state (this condition is called Pending on resource) Now 'Process B' gets the CPU and by 'Process C'. Since 'Process C' acquired the semaphore for signalling the accesses of the shared variable 'X' 'Process A' will not be

able to access it. Thus 'process A' is put into blocked state (This condition is called Pending on resource) Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state to preempted by another high priority task. The highest priority Process 'Process A' has to wait till Process C' gets a chance to execute and release the semaphore. This Produces unwanted delay in the execution of high priority task which is supposed to be executed immediately when it was 'Ready':

Priority Inheritance :-

→ A low-priority task that is currently accessing a shared resource requested by a high priority task temporarily "inherits" the priority of that high-priority task, from the moment the high priority task raises the request.

→ Boosting the priority of the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource and thereby reduces the delay in waiting to get the resource requested by high priority task.

→ The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource.

Implementation of priority inheritance work around in the priority inversion problem

Note :- priority inheritance is only a work around and it will not eliminate the delay in waiting the high priority task to get the resource from the low priority task. The only thing is it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moments at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU. (A true form of selfishness)

Priority ceiling :-

A priority is associated with each shared resources. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level is called ceiling priority. whenever a task accesses a shared resources the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which access the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the resource is also shared. This eliminates the preemption of the task by other medium priority tasks leading to priority inversion.

Handling Priority Inversion problems with priority Inheritance.
Process
Time
Process 'a' (High priority)
Process 'b' (Med. priority)
Process 'c' (Low priority)
Process 'c' acquires the shared variable
Process 'b' enters 'Ready' state and preempts Process 'c'
Process 'a' enters 'Ready' state and preempts Process 'b'.
Process 'a' tries to access the resource which is currently acquired by process C. Priority of process 'c' is increased to high.
Delay in execution of Process 'a' happened due to resource waiting
Process C released the shared resource
Priority of process C is boosted to its original value. Process A starts executing since it is of highest priority.
Process A completes its execution. Process starts executing since it is of higher priority
Completed to process C
Process B completes its execution
Process C starts executing.
Running
Waiting in Ready Queue
Running
Waiting in Ready Queue
Running
Running
Waiting
Running

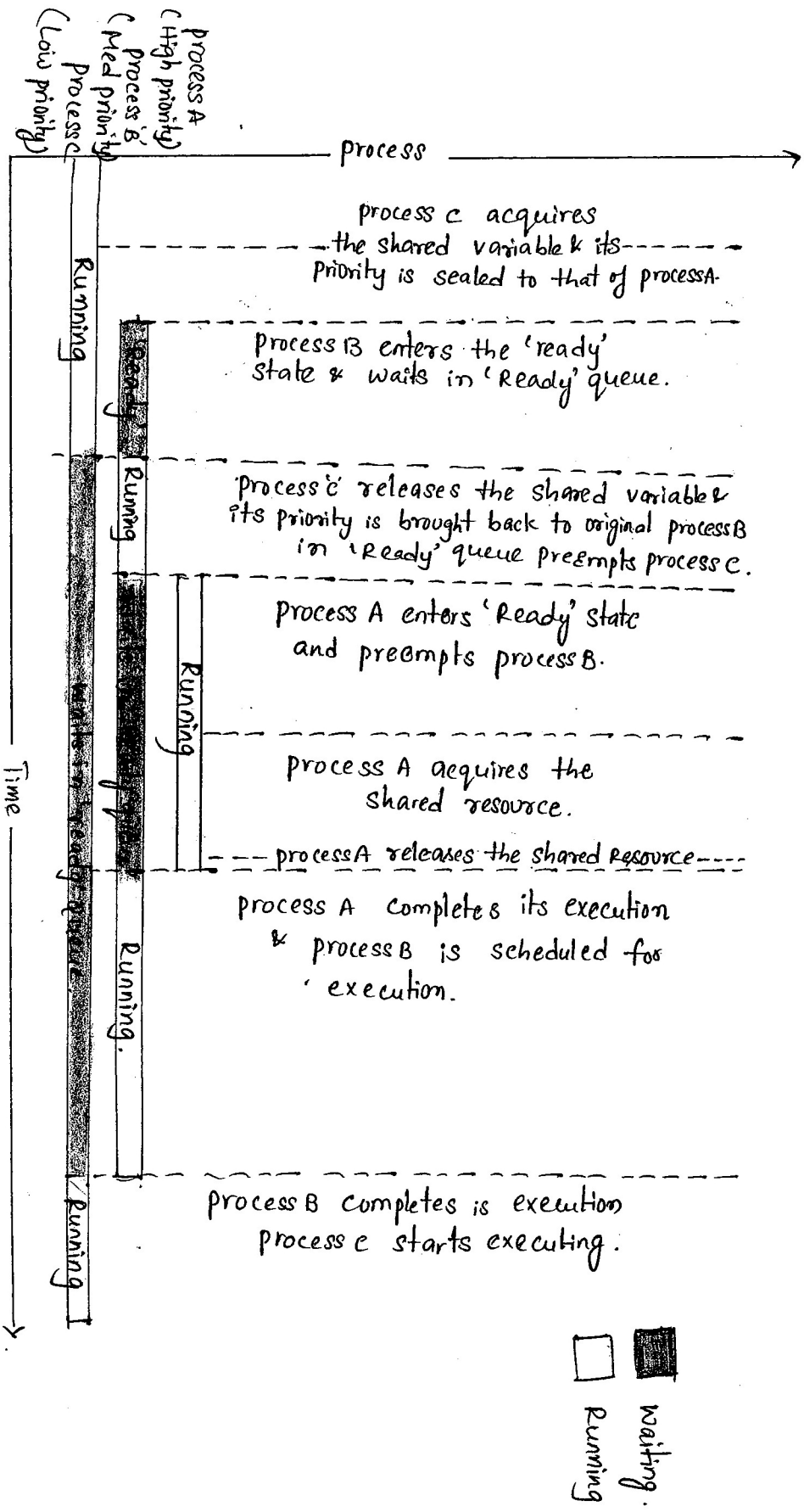
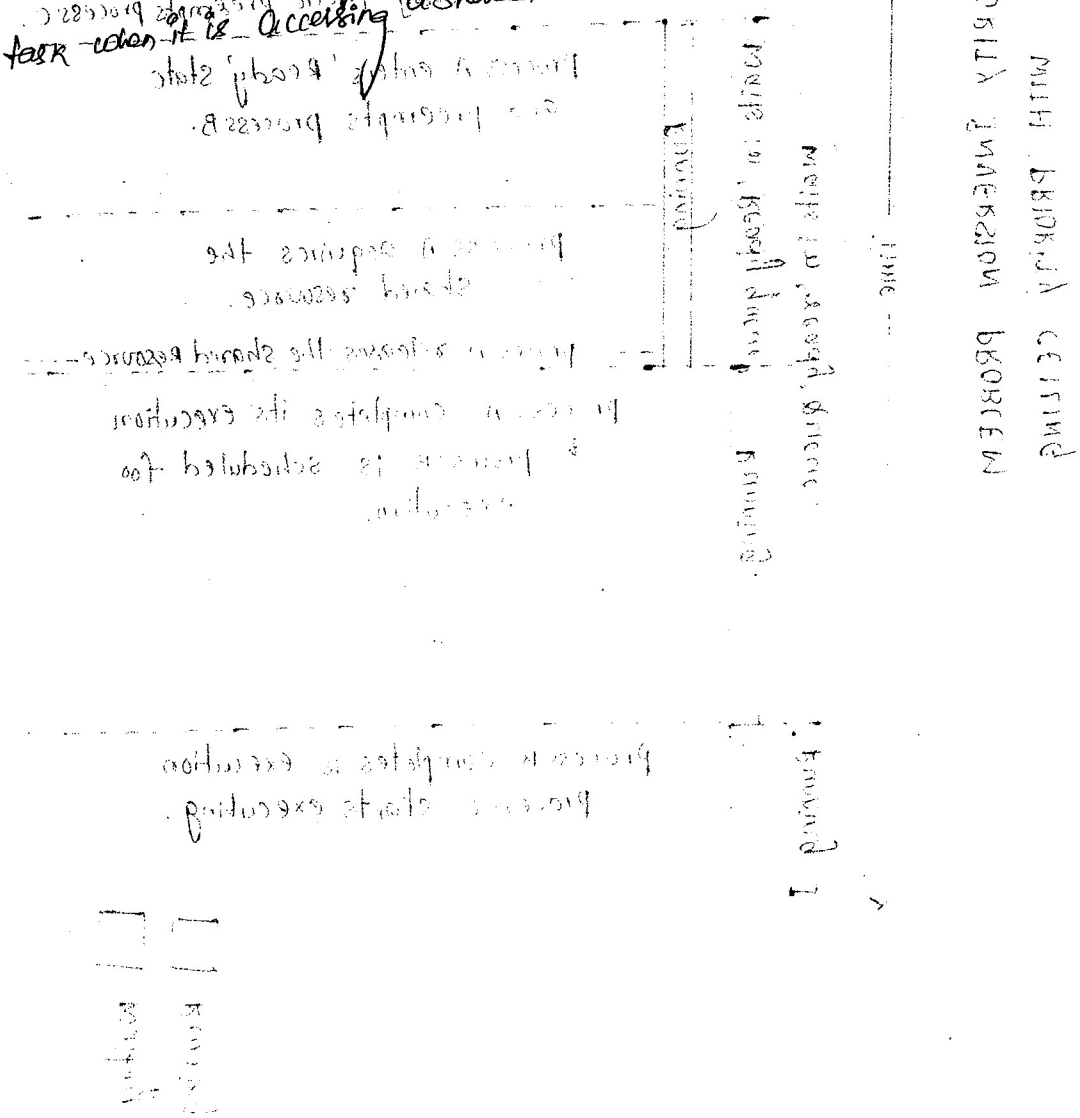


fig: HANDLING PRIORITY INVERSION PROBLEM WITH PRIORITY CEILING

The priority of the task is brought back to the original level once the task completes the accessing of the shared resource.

The biggest drawback of priority ceiling is that it may produce hidden priority inversion, with priority ceiling technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority task among which the resource is shared and other tasks with priorities higher than that of low priority task is not allowed to preempt the low priority task when it is accessing a shared resource.



Deadlock :- a deadlock condition creates a situation where none of the processes are able to make any progress in their execution resulting in a set of deadlocked processes.

Ex: A situation very similar to our traffic jam issues on a junction.

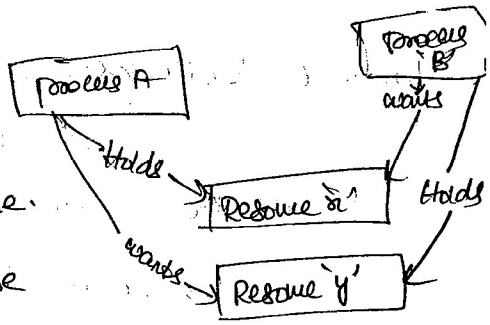
In its simplest form "deadlock" is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process.

Ex:- process 'A' holds a resource 'x' and it wants a resource 'y' held by process 'B'. process 'B' is currently holding resource 'y' and it wants the resource 'x' which is currently held by process 'A'. Both hold the respective resources and they compete each other to get the resources held by respective processes. The result of competition is "deadlock".

Note :- As none of the process will be able to access the resources held by other processes since they are locked.

The conditions favouring deadlock situation are listed :-

Scenarios leading to deadlock



No Resource preemption :-

The criteria that operating system cannot take back a resource from a process process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Mutual Exclusion :- The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion.

Ex:- Access display hardware in embedded device.

The 'OS' adopt following techniques to detect and prevent deadlock Conditions.

Ignore deadlock :- Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large Compared to the chance of happening a deadlock.

Detect and Recover :- (This Suggest detect deadlock and recovery from it)

Operating System keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock Condition can be detected by analysing the resource graph by graph analyser algorithms. Once a deadlock Condition is detected, the System can terminate a process or preempt the resource to break the deadlocking cycle.

Avoid deadlock :- Deadlock is avoided by the Careful resource allocation techniques by the Operating System. It is similar to traffic light mechanism at junctions to avoid traffic jams.

Prevent deadlocks :-

Prevent the deadlock Condition by negating one of the Conditions favouring the situation.

⇒ ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/guidelines

(1) A process must request all its required resource and the resource should be allocated before the process begins its execution.

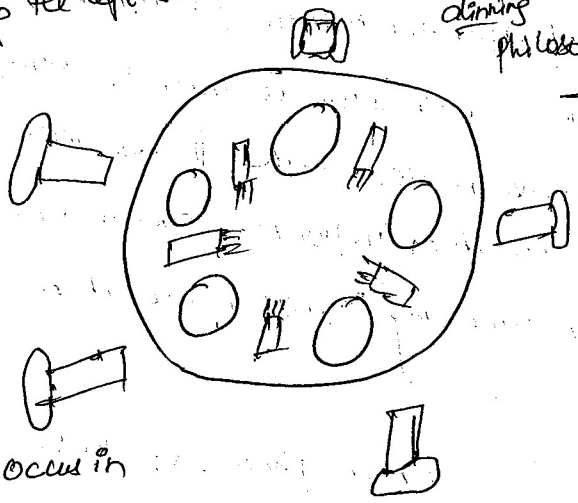
(2) Grant resource allocation request from process only if the process does not hold a resource Currently.

The Dining philosophers problem:-

The Dinning philosophers problem is an interesting example for Synchronisation issues in Resource Utilisation.

Five philosophers are sitting around a round table, involved in eating and brainstorming. At any point of time each philosopher will be in any one of three states; eating, hungry or brainstorming. For eating each philosophers require 2 forks. There are only 5 forks available on the dining table and they are arranged in a fashion one fork in between two philosophers. The philosophers can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then right fork.

dining philosophers problem



The various Scenarios that occur in the Situation.

(1) All the philosophers involved in brainstorming together and try to eat together. each philosopher picks up the left fork and is unable to proceed as required two forks for eating

Note:- philosopher 'i' thinks that the philosopher 'i' sitting to the right of him/her will put the fork down and waits for it. hence they will not make any progress in eating and result in Starvation (process does not get the resources to continue its execution for a long time).

and deadlock.

(2) All the philosophers start brainstorming together. one of the philosophers is hungry and he picks up the left fork, when the philosopher is about to pick up the right fork, the philosophers sitting to him right also become hungry and tries to grab the left fork resulting a Race Condition (multiple process compete each other to access and shared data concurrently)

(3) All the philosophers involve in brainstorming together and try to eat together. each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating. here the philosophers anticipates that the adjacently sitting philosophers will put his/her fork down and wait for a fixed duration and after this puts the fork down. this condition leads to Livelock.

Livelock :- when a process enters in wait state for a resource and continues in that state forever without making any progress in the executions in livelock process always does something but is unable to make any progress in execution completion.

Solution :- when a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork hence each philosopher acquires a semaphore before picking up the fork. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry by signaling the semaphores associated with forks.

Semaphore :-

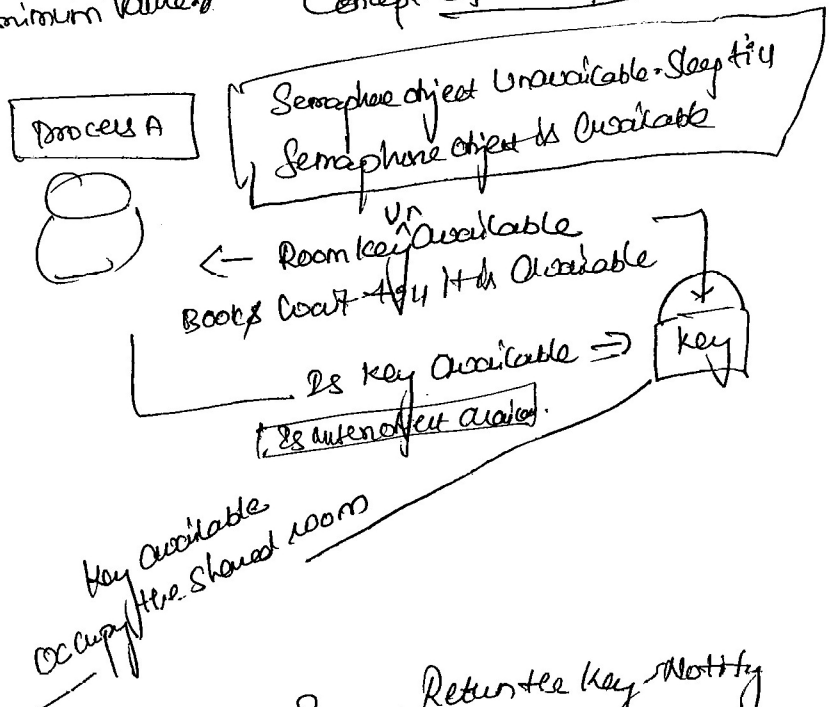
Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it.

Based upon the implementation of the sharing limitation of shared resource, semaphores are classified in to two namely Binary Semaphore, Counting Semaphore.

The binary Semaphore (Mutex) provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it.

Unlike binary Semaphore, the "Counting Semaphore" limits the access of resources by a fixed number of processes. Counting Semaphore maintains count between zero an maximum value. and provides the usage of resource to maximum value.

Concept of Semaphore :-



Device Drivers

(1) Device Drivers is a piece of Software that acts a bridge b/w the operating system and the hardware.

→ In operating system based product architecture, the user applications talk to the operating system kernel for all necessary information exchange including communication with hardware peripherals.

→ The 'os' kernel routes to the connected hardware peripheral. The 'os' provides interfaces in the form of Application programming Interfaces (API) for accessing the hardware.

→ The device driver abstracts the hardware from the user applications.

→ Device drivers are responsible for initiating and managing the communica with the hardware peripherals. They are responsible for establishing the connectivity, initializing the hardware (setting up various registers for hardware device and transfering

→ An embedded product may have different types of hardware data components like wi-fi module, storage device interfaces, filesystems etc. The initialization of these devices and protocols required for communicatin with them is different. Hence all these requirements are implemented in drivers. Hence each hardware require unique driver component.

→ Certain drivers come as part of the 'os' kernel and certain. drivers need to be installed on the fly. Ex:- The program storage memory for an embedded product (flash memory requires a flash driver to read and write data from/to it. This driver should come as the part of the 'os' kernel.

→ The drivers which are part of the 'os' known as Built-in drivers (or on-board drivers. These drivers are loaded by 'os' at the time of booting the device and always kept in RAM.

→ Drivers which need to be installed for accessing a device are known as installable drivers. These drivers are loaded by the 'os' on a need basis.

→ whenever the device is connected, the 'OS' loads the corresponding driver to memory. when the device is removed, the driver is unloaded (from memory).

→ The 'OS' maintains the record of the drivers corresponding to each hardware.

→ The implementation of driver is 'OS' dependent.

→ The drivers can run on either user space or kernel space. Drivers which run in user space are known as user mode drivers and drivers run on kernel space is called kernel mode drivers.

Note: If error occurs in user mode driver it won't affect the services of kernel. On the other hand, if error occurs in kernel mode driver, it may lead to the kernel crash.

→ The way how the device driver is written and how the interrupts are handled in it are operating system dependent. The device driver implements the following

(1) Device initialisation and Interrupt Configuration
(2) Interrupt handling and processing
(3) client interfacing (Interfacing with user applications)

(1) The Device (Hardware) initialisation part of driver deals with configuring the different registers of the device. Ex:1 Configuring the I/O port line of processor as input or output line and setting its associated registers for building an. General purpose Input output (GPIO) driver.

(2) The interrupt configuring needs to be configuring the interrupts that are associated with the hardware.

The basic interrupt configuration involves the following.

(a) Set the interrupt type (Edge/level triggered), enable the interrupts, and set the interrupt priorities.