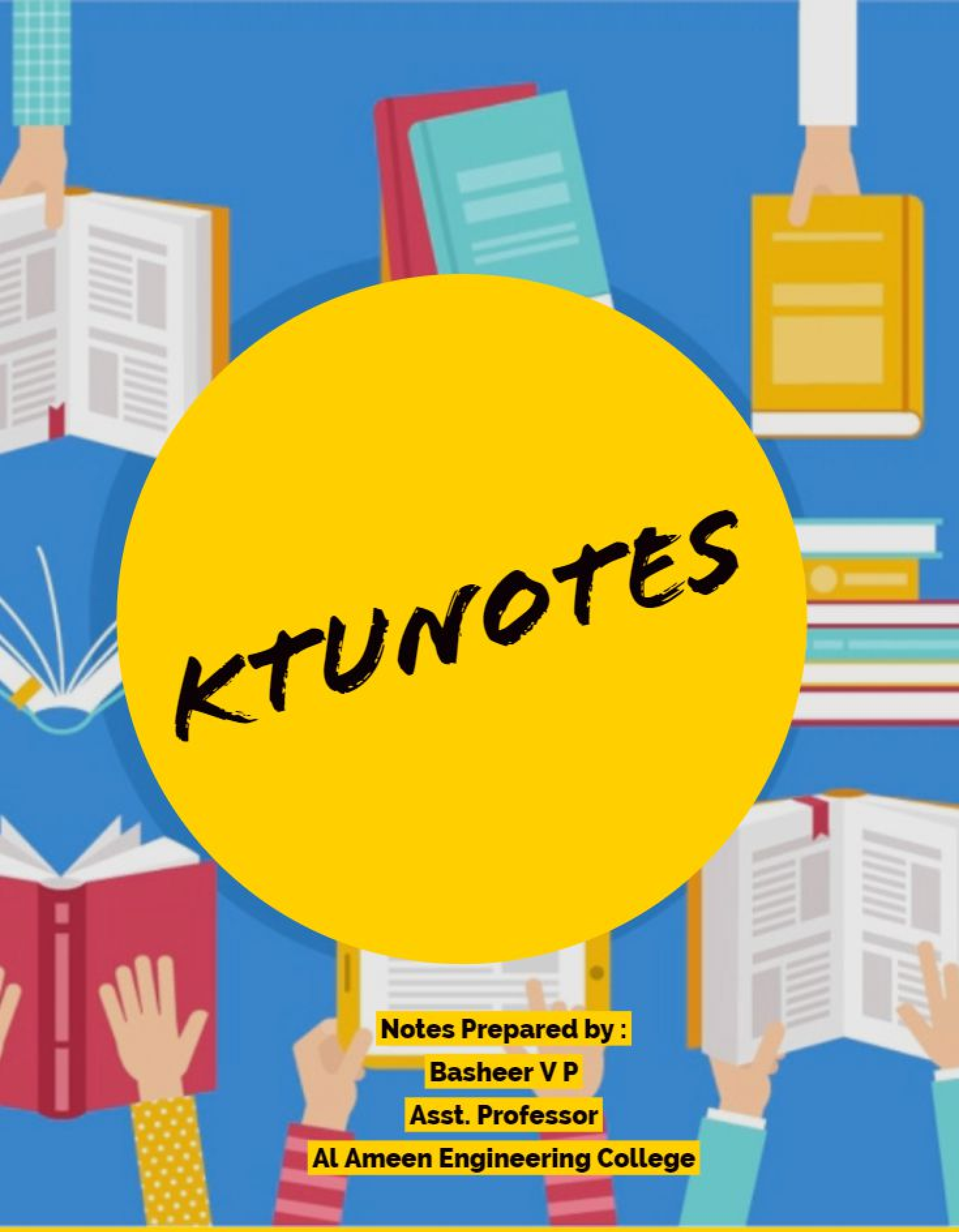


Notes Prepared by :
Basheer V P
Asst. Professor
Al Ameen Engineering College

EC 308 EMBEDDED SYSTEMS

MODULE-5

<u>SYLLABUS</u>

Inter Process Communication and Synchronization -Process, tasks and threads –Shared data– Inter process communication - Signals – Semaphore – Message Queues – Mailboxes – Pipes –Sockets – Remote Procedure Calls (RPCs).

5.1 Process

Process Features

- A process consists of executable program (codes), *state* of which is controlled by OS, the *state* during running of a process represented by process-status (running, blocked, or finished), process structure—its data, objects and resources, and process control block (PCB).
- Runs when it is scheduled to run by the OS (kernel)
- OS gives the control of the CPU on a process‘s request (system call).
- Runs by executing the instructions and the continuous changes of its state takes place as the program counter (PC) changes.
- Process is that executing unit of computation, which is controlled by some process (of the OS) for a scheduling mechanism that lets it execute on the CPU and by some process at OS for a resource management mechanism that lets it use the system- memory and other system resources such as network, file, display or printer.
- Application program can be said to consist of number of processes

Example - Mobile Phone Device embedded software

- Software highly complex.
- Number of functions, ISRs, processes threads, multiple physical and virtual device drivers, and several program objects that must be concurrently processed on a single processor.
- Voice encoding and convoluting process─ the device captures the spoken words through a speaker and generates the digital signals after analog to digital conversion, the digits are encoded and convoluted using a CODEC,
- Modulating process,
- Display process,
- GUIs (graphic user interfaces), and
- Key input process ─ for provisioning of the user interrupts

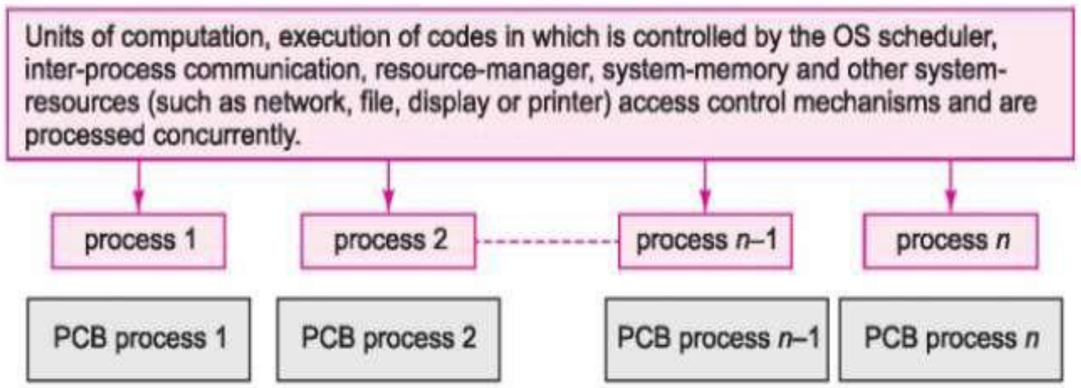
Process Control Block

- A data structure having the information using which the OS controls the Process state.
- Stores in protected memory area of the kernel.
- Consists of the information about the process state

Information about the process state at Process Control Block:

- Process ID,
- Process priority,
- Parent process (if any),
- Child process (if any), and
- Address to the next process PCB which will run,
- Allocated program memory address blocks in physical memory and in secondary (virtual) memory for the process-codes,
- Allocated process-specific data address blocks
- Allocated process-heap (data generated during the program run) addresses,
- Allocated process-stack addresses for the functions called during running of the process,
- Allocated addresses of CPU register-save area as a process context represents by CPU registers, which include the program counter and stack pointer
- Allocated addresses of CPU register-save area as a process context [Register-contents (define process context) include the program counter and stack pointer contents]
- process-state signal mask [when mask is set to 0 (active) the process is inhibited from running and when reset to 1, the process is allowed to run],
- Signals (messages) dispatch table [process IPC functions],
- OS allocated resources' descriptors (for example, file descriptors for open files, device descriptors for open (accessible) devices, device-buffer addresses and status, socket- descriptor for open socket), and
- Security restrictions and permissions.

Process


Units of computation, execution of codes in which is controlled by the OS scheduler, inter-process communication, resource-manager, system-memory and other system-resources (such as network, file, display or printer) access control mechanisms and are processed concurrently.
process 1
process 2
process n–1
process n
PCB process 1
PCB process 2
PCB process n–1
PCB process n

5.2 Thread

Thread Features

- A thread consists of executable program (codes), *state* of which is controlled by OS,
- The state information– *thread-status* (running, blocked, or finished), *thread structure*—its data, objects and a subset of the process resources, and *thread-stack.*
- Considered a lightweight process and a process level controlled entity.[Light weight means its running does not depend on system resources] .

- Process… heavyweight
  - Process considered as a heavyweight process and a kernel-level controlled entity.
  - Process thus can have codes in secondary memory from which the pages can be swapped into the physical primary memory during running of the process. [Heavy weight means its running may depend on system resources]
  - May have process structure with the virtual memory map, file descriptors, user–ID, etc.
  - Can have multiple threads, which share the process structure thread
  - *A* process or sub-process within a process that has its own program counter, its own stack pointer and stack, its own priority parameter for its scheduling by a thread scheduler
  - Its' variables that load into the processor registers on context switching.
  - Has own signal mask at the kernel. Thread's signal mask
  - When unmasked lets the thread activate and run.
  - When masked, the thread is put into a queue of pending threads.

Multiprocessing OS

- A multiprocessing OS runs more than one processes.
- When a process consists of multiple threads, it is called multithreaded process.
- A thread can be considered as daughter process.
- A thread defines a minimum unit of a multithreaded process that an OS schedules onto the CPU and allocates other system resources.

Thread parameters

- Each thread has independent parameters ID, priority, program counter, stack pointer, CPU registers and its present status.
- Thread states─ starting, running, blocked (sleep) and finished

Thread's stack

- When a function in a thread in OS is called, the calling function state is placed on the stack top.
- When there is return the calling function takes the state information from the stack top
- A data structure having the information using which the OS controls the thread state.
- Stores in protected memory area of the kernel.
- Consists of the information about the thread state

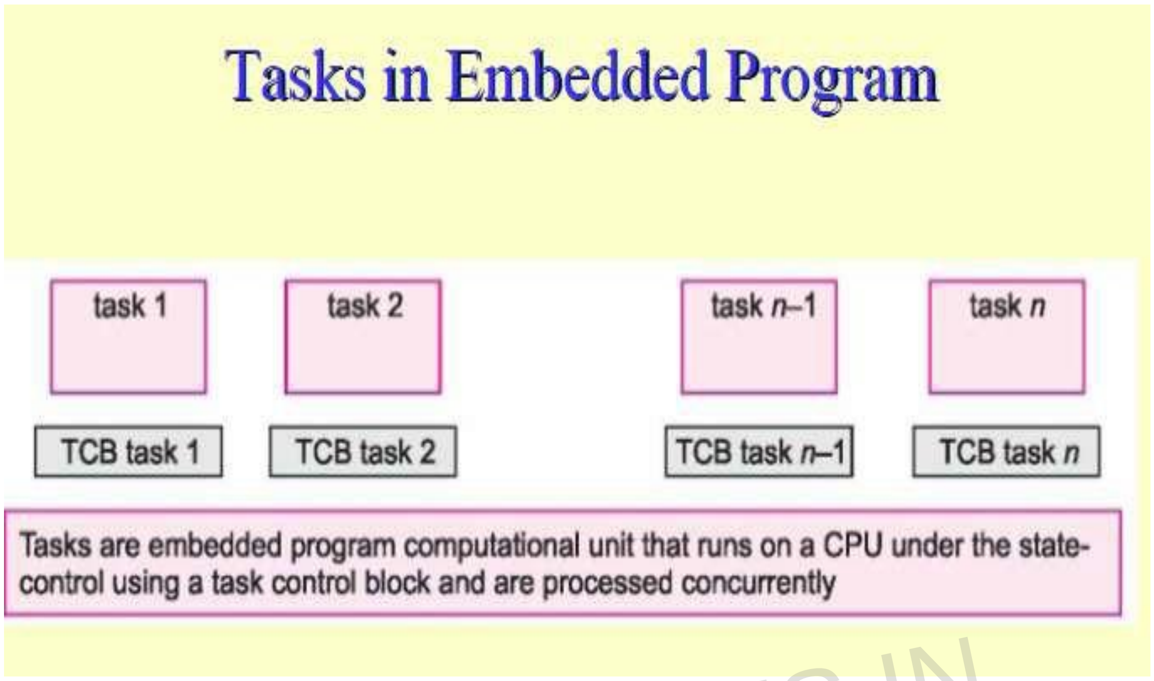
5.3 Task

Task Features

- An application program can also be said to be a program consisting of the tasks and task behaviors in various states that are controlled by OS.
- A task is like a process or thread in an OS.
- Task─ term used for the process in the RTOSes for the embedded systems.
    - For example, VxWorks and μCOS-II are the RTOSes, which use the term task.
- A task consists of executable program (codes), *state* of which is controlled by OS, the *state* during running of a task represented by information of process status (running, blocked, or finished),process-structure—its data, objects and resources, and task control block (PCB).
- Runs when it is scheduled to run by the OS (kernel), which gives the control of the CPU on a task request (system call) or a message.
- Runs by executing the instructions and the continuous changes of its state takes place as the program counter (PC) changes.
- Task is that executing unit of computation, which is controlled by some process at the OS scheduling mechanism, which lets it execute on the CPU and by some process at OS for a resource-management mechanism that lets it use the system memory and other system-resources such as network, file, display or printer.
- A task is an independent process.
- No task can call another task. [It is unlike a C (or C++) function, which can

call another function.]

- The task─ can send signal (s) or message(s) that can let another task run.
- The OS can only block a running task and let another task gain access of CPU to run the servicing codes



Tasks in Embedded Program

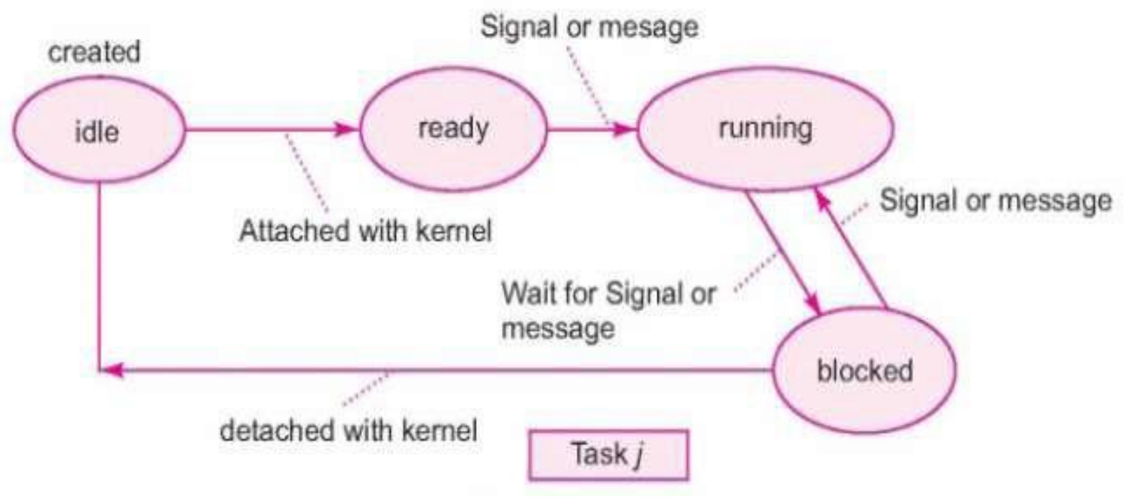
Tasks are embedded program computational unit that runs on a CPU under the state-control using a task control block and are processed concurrently

Task States

Following are the five states of a task that may have.

(i) Idle state [Not attached or not registered]
(ii) Ready State [Attached or registered]
(iii) Running state
(iv) Blocked (waiting) state
(v) Delayed for a preset period

Task States



Idle (created) state

- The task has been created and memory allotted to its structure however, it is not ready and is not schedulable by kernel.

Ready (Active) State

- The created task is ready and is schedulable by the kernel but not running at present as another higher priority task is scheduled to run and gets the system resources at this instance.

Running state

- Executing the codes and getting the system resources at this instance. It will run till it needs some IPC (input) or wait for an event or till it gets pre-empted by another higher priority task than this one.
-

Blocked (waiting) state

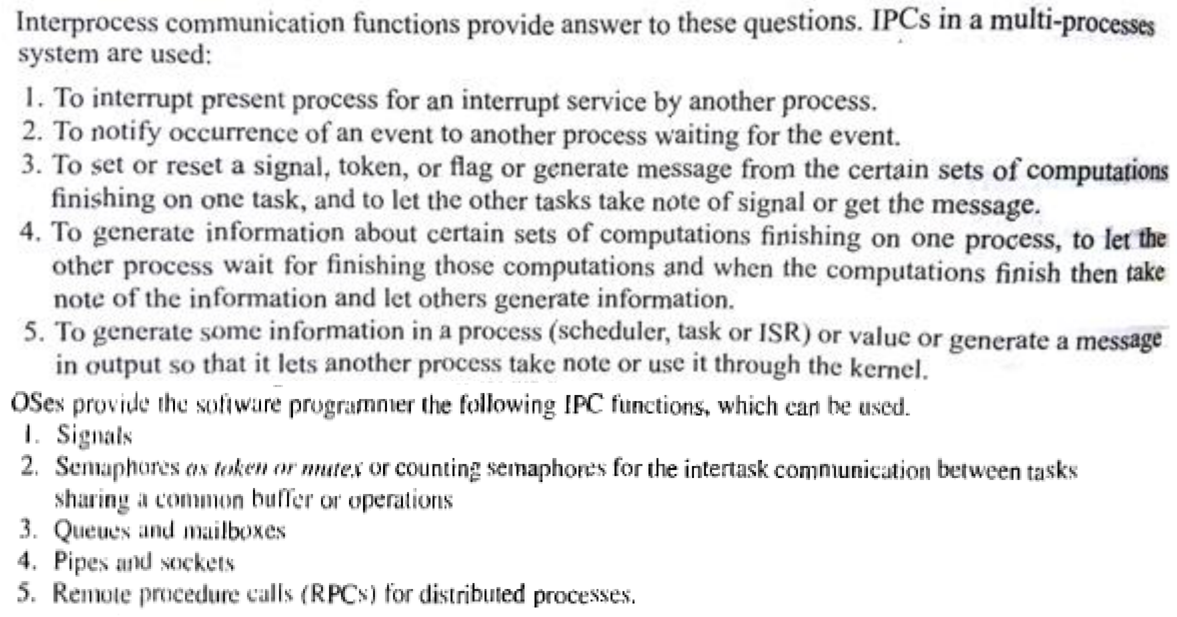
- Execution of task codes suspends after saving the needed parameters into its Context. It needs some IPC (input) or it needs to wait for an event or wait for higher priority task to block to enable running after blocking.

Deleted (finished) state

- Deleted Task─ The created task has memory deallotted to its structure. It frees the memory. Task has to be re-created.

5.4 Inter Process Communication (IPC)

- Inter process communication (IPC) means that a process (scheduler or task or ISR) generates some information by setting or resetting a Token or value, or generates an output so that it lets another process take note or to signal to OS for starting a process or use it under the control of an OS.



Interprocess communication functions provide answer to these questions. IPCs in a multi-processes system are used:

1. To interrupt present process for an interrupt service by another process.
2. To notify occurrence of an event to another process waiting for the event.
3. To set or reset a signal, token, or flag or generate message from the certain sets of computations finishing on one task, and to let the other tasks take note of signal or get the message.
4. To generate information about certain sets of computations finishing on one process, to let the other process wait for finishing those computations and when the computations finish then take note of the information and let others generate information.
5. To generate some information in a process (scheduler, task or ISR) or value or generate a message in output so that it lets another process take note or use it through the kernel.

OSes provide the software programmer the following IPC functions, which can be used.

1. Signals
2. Semaphores *as token or mutex* or counting semaphores for the intertask communication between tasks sharing a common buffer or operations
3. Queues and mailboxes
4. Pipes and sockets
5. Remote procedure calls (RPCs) for distributed processes.

5.5 Signals

Features:

- One way for inter process communication is to use an OS function signal ( ).
- It is provided in Unix, Linux and several RTOSes.
- Unix and Linux OSes use signals profusely and have thirty-one different types of Signals for the various events.
- Unless masked by a signal mask, the signal allows the execution of the **signal handling function** and allows the handler to run just as a hardware interrupt allows the execution of an ISR.
- When a signal is sent from a process, OS interrupts the process execution and calls the function for signal handling. On return from the signal handler, the process continues as before.
- Signal provides the shortest communication.

The following are the signal related IPC functions

1. SigHandler ( )
   - This function is used to create a signal handler corresponding to a signal identified by the signal number and define a pointer to signal context. The signal context saves the registers on signal.

2. Connect ()
   - This function is used to connect an interrupt vector to a signal number, with signaled handler function and signal handler arguments. The interrupt vector provides the program counter value for the signal handler function address.

3. signal ( )
   - It is the function signal ( ) to send a signal identified by a number to a signal handler task.
4. Mask ()
   - This function is used to mask the signal
5. Unmask ()
   - This function is used to unmask the signal
6. Ignore ()
   - This function is used to ignore the signal

Application Example:

- An important application of the signals is to handle exception.
- An exception is a process that is executed on a specific reported run-time condition.
- A signal reports an error (called 'Exception') during the running of a task and then lets the scheduler initiate an error-handling process or ISR, for example, initiate error-login task.

Advantage of using a Signal

- It takes the shortest possible CPU time.

Drawbacks of using a Signal

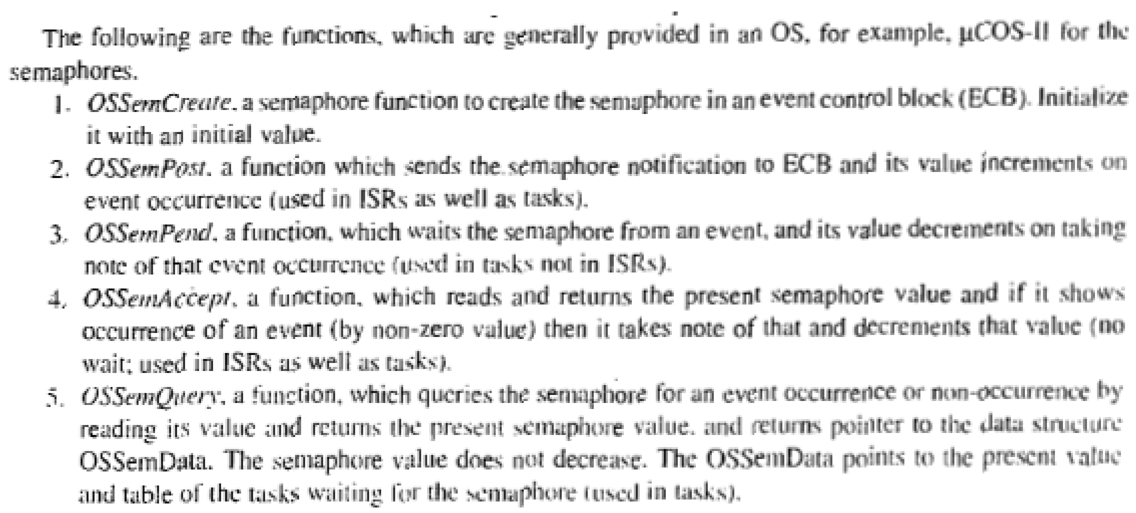
- Signal is handled only by a very high priority process (service routine). That may disrupt the usual schedule and usual priority inheritance mechanism.
- Signal may cause reentrancy problem [process not retuning to state identical to the one before signal handler process executed].

5.6 Semaphore

- A semaphore is special kind of shared program variable.
- A semaphore is a **kernel object that one or more tasks can acquire or release** for the purpose of synchronization or mutual exclusion.
- Mutual exclusion is a provision by which only one task at a time can access a shared resource (port, memory block, etc.)
- The value of a semaphore is a non-negative integer.

- If the integer data is only allowed to take the values 0 and 1 then the semaphore is referred to as a **binary semaphore**.
- **Binary semaphore** allows only one process at a time to access the shared resource.
- If the integer data is allowed to take any non-negative value then the semaphore is referred to as a **General Semaphore or Counting Semaphore**.
- **Counting Semaphore** allows N > 1 processes to access the resource simultaneously. Instead of having states empty and full, it uses a counter S. The counter is initialised to N, with S = 0 corresponding to the full state.
- OS provides the semaphore IPC functions for creating, releasing and taking of the semaphore

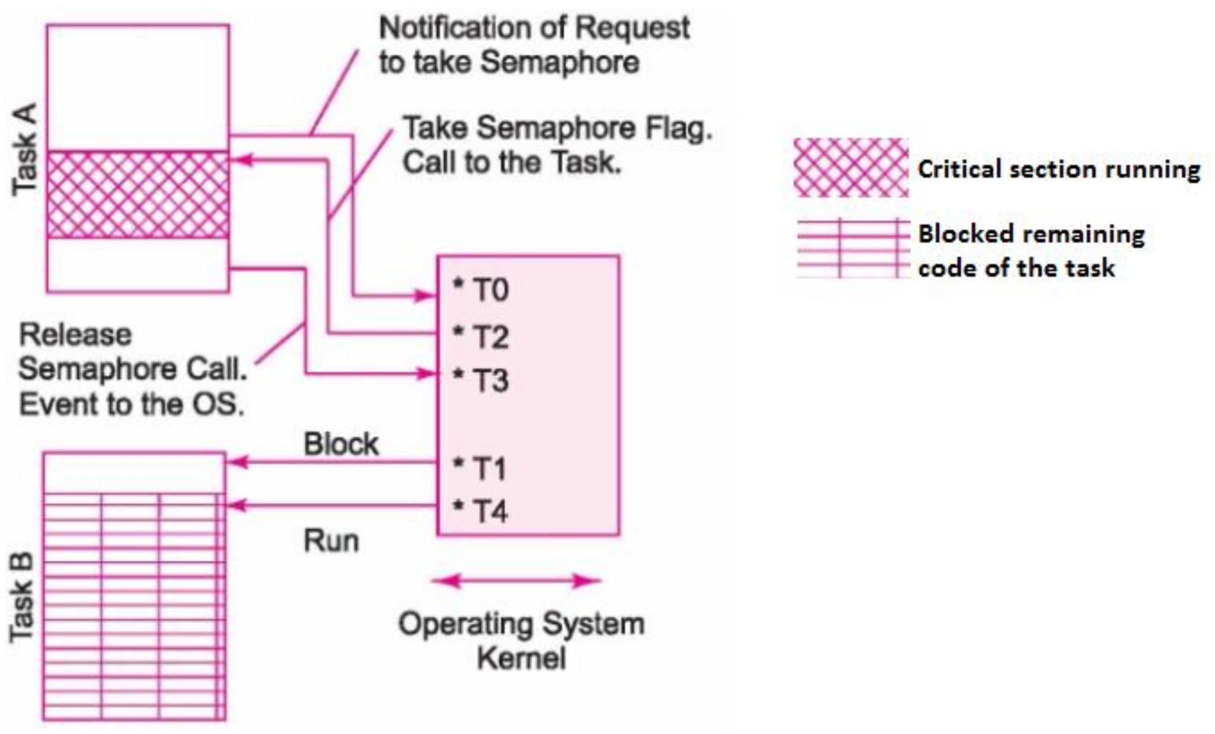
Semaphore functions:


The following are the functions, which are generally provided in an OS, for example, µCOS-II for the semaphores.

1. *OSSemCreate*, a semaphore function to create the semaphore in an event control block (ECB). Initialize it with an initial value.
2. *OSSemPost*, a function which sends the semaphore notification to ECB and its value increments on event occurrence (used in ISRs as well as tasks).
3. *OSSemPend*, a function, which waits the semaphore from an event, and its value decrements on taking note of that event occurrence (used in tasks not in ISRs).
4. *OSSemAccept*, a function, which reads and returns the present semaphore value and if it shows occurrence of an event (by non-zero value) then it takes note of that and decrements that value (no wait; used in ISRs as well as tasks).
5. *OSSemQuery*, a function, which queries the semaphore for an event occurrence or non-occurrence by reading its value and returns the present semaphore value, and returns pointer to the data structure OSSemData. The semaphore value does not decrease. The OSSemData points to the present value and table of the tasks waiting for the semaphore (used in tasks).

<u>Semaphore as a resource key and for critical sections having shared for critical sections having shared resource (s):</u>

- Shared Resources like shared memory buffer are to be used only by one task (process or thread) at an instance.
- OS Functions provide for the use of a semaphore resource key for running of the codes in critical section.
- Let a task A, when getting access to a resource/critical section notifies to the OS to have taken the semaphore (take notice)That is, an OS function *OSSemPend()* runs to notify. The OS returns the semaphore as taken (accepted) by decrementing the semaphore from 1 to 0.
- Now the task A accesses the resource.
- The task A, after completing the access to a resource/critical section it notiufies to the OS to have posted that semaphore (post notice). That is an OS function *OSSemPost()* runs to notify. The OS returns the semaphore as released by incrementing the semaphore from 0 to 1.
- Figure below shows the use of semaphore between A and B. It shows the five sequential actions at five different times.


Notification of Request to take Semaphore
Take Semaphore Flag. Call to the Task.
Critical section running
Blocked remaining code of the task
Task A
* T0
* T2
* T3
Release Semaphore Call. Event to the OS.
Block
* T1
* T4
Run
Task B
Operating System Kernel

Mutex Semaphore for use as resource key:

- Mutex means mutually exclusive key.
- Mutex is a binary semaphore usable for protecting use of resource by other task section at an instance
- Let the semaphore sm has an initial value = 1
- When the semaphore is taken by a task the semaphore sm decrements from 1 to 0 and the waiting task codes starts.
- Assume that the sm increments from 0 to 1 for signalling or notifying end of use of the semaphore that section of codes in the task.
- When sm = 0 ─assumed that it has been taken (or accepted) and other task code section has not taken it yet and using the resource
- When sm = 1─assumed that it has been released (or sent or posted) and other task code section can now take the key and use the resource.

P and V semaphores

- An efficient synchronisation mechanism
- P and V semaphores represent by integers in place of binary or unsigned integers.
- The semaphore, apart from initialization, is accessed only through two standard atomic operations - P and V.
- P semaphore function signals that the task requires a resource and if not available waits for it.
- V semaphore function signals which the task passes to the OS that the resource is now free for the other users.

5.7 <u>Message Queues</u>

- A message queue is an inter process communication mechanism which uses an n byte memory block as queue.
- The message pointers post into a queue by the tasks either at the back as in a queue or at the front as in a stack.
- Every OS provides message queue IPC functions.

<u>Features:</u>

1. OS provides for inserting and deleting the message-pointers or messages.
2. Each queue for a message need initialization (creation) before using the functions in the scheduler for the message queue
3. There may be a provision for multiple queues for the multiple types or destinations of messages. Each queue may have an ID.
4. Each queue either has a user definable size (upper limit for number of bytes) or a fixed pre-defined size assigned by the scheduler.
5. When an RTOS call is to insert into the queue, the bytes are as per the pointed number of bytes. For example, for an integer or float variable as a pointer, there will be four bytes inserted per call. If the pointer is for an array of 8 integers, then 32 bytes will be inserted into the queue.
6. When a queue becomes full, there may be a need for error handling and user codes for blocking the task(s). There may not be self-blocking.

<u>Queue IPC functions</u>

- Figure (a) shows the memory blocks at OS for inserting deleting and other functions.
- Figure (b) shows the functions for the queue in the OS.
- Figure (c) shows a queue message block with the messages or message pointers.
- Two pointers *QHEAD and *QTAIL are for queue head and tail memory locations

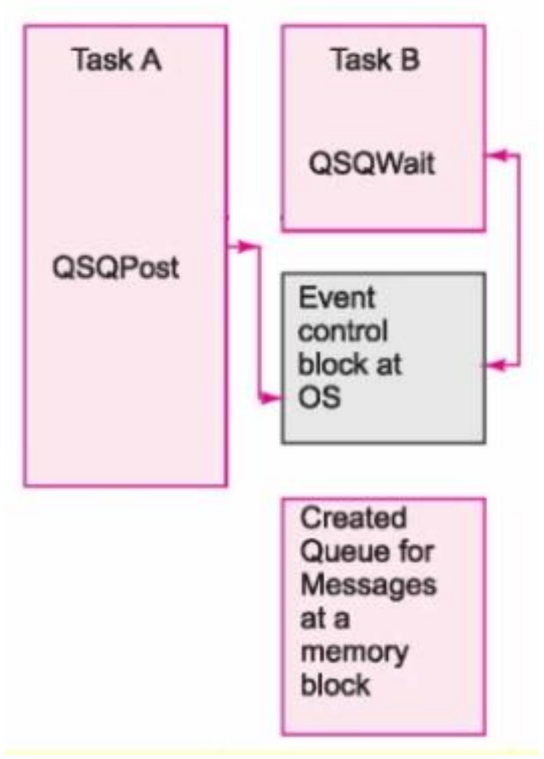

Task A
QSQPost
Task B
QSQWait
Event control block at OS
Created Queue for Messages at a memory block

Figure (a) Queue handling by tasks

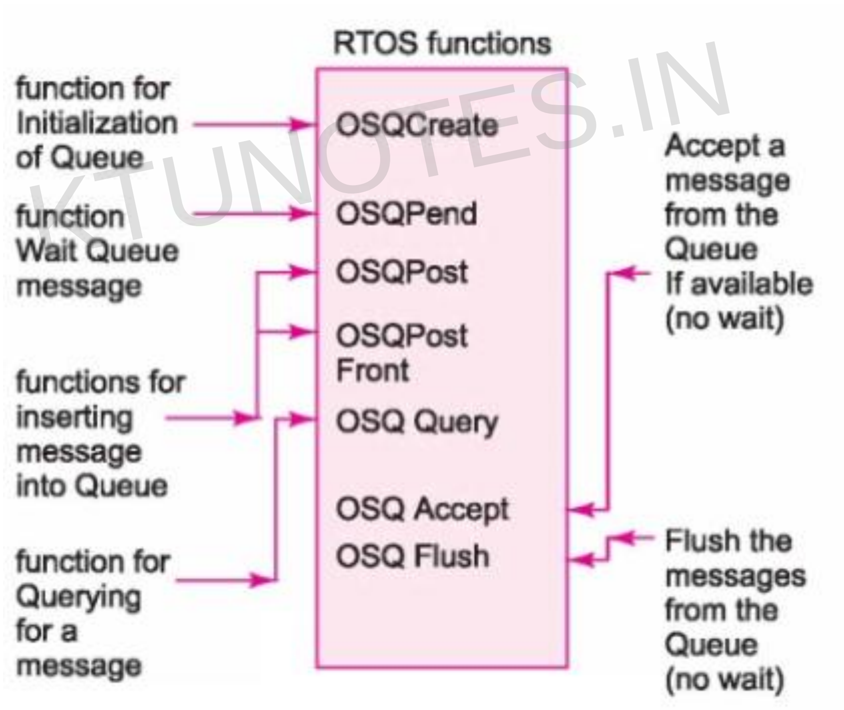

RTOS functions
function for Initialization of Queue
function Wait Queue message
functions for inserting message into Queue
function for Querying for a message
OSQCreate
OSQPend
OSQPost
OSQPost Front
OSQ Query
OSQ Accept
OSQ Flush
Accept a message from the Queue If available (no wait)
Flush the messages from the Queue (no wait)

Figure (b) Queue Functions

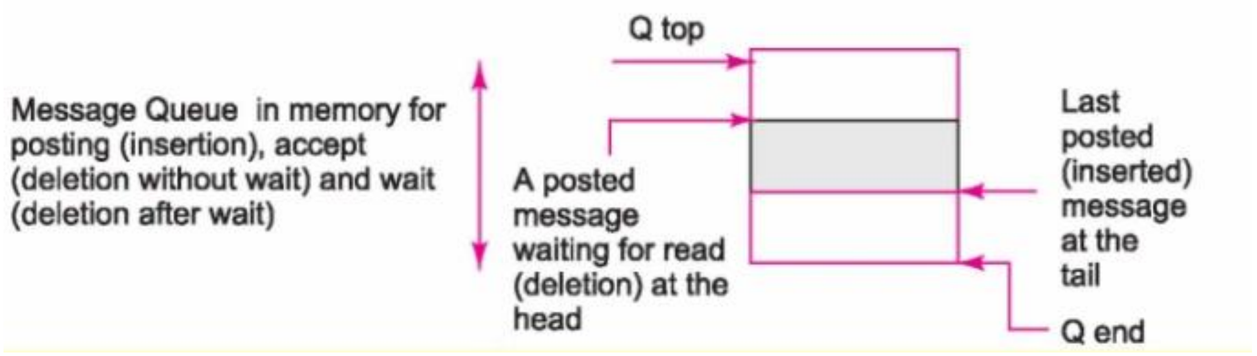

Q top
Message Queue in memory for posting (insertion), accept (deletion without wait) and wait (deletion after wait)
A posted message waiting for read (deletion) at the head
Last posted (inserted) message at the tail
Q end

Figure (c) Queue message block

OSQCreate

- ➢ This function is used to create a queue and initialize the queue.

OSQPost

- ➢ This function is used to post a message to the message block as per the queue tail pointer, *QTAIL.

OSQPend

- ➢ This function is used to wait for a queue message at the queue and reads and deletes that when received.

OSQAccept

- ➢ This function is used to delete the present queue head after checking its presence yes or no and after the deletion the queue head pointer increments.

OSQFlush

- ➢ This function is used to deletes messages from queue head to tail, after the flush the queue head and tail points to QTop, pointer to start of the queue.

OSQQuery

- ➢ This function is used to query the queue message-block when read and but the message is not deleted. The function returns pointer to the message queue *QHEAD if there are the messages in the queue or else NULL. It return a pointer to data structure of the queue data structure which has *QHEAD, number of queued messages, size of the queue and. table of tasks waiting for the messages from the queue.

OSQPostFront

- ➢ This function is used to send a message as per the queue front pointer, *QHEAD. Use of this function is made in the following situations. A message is urgent or is of higher priority than all the previously posted message into the queue.

5.8 Mailbox

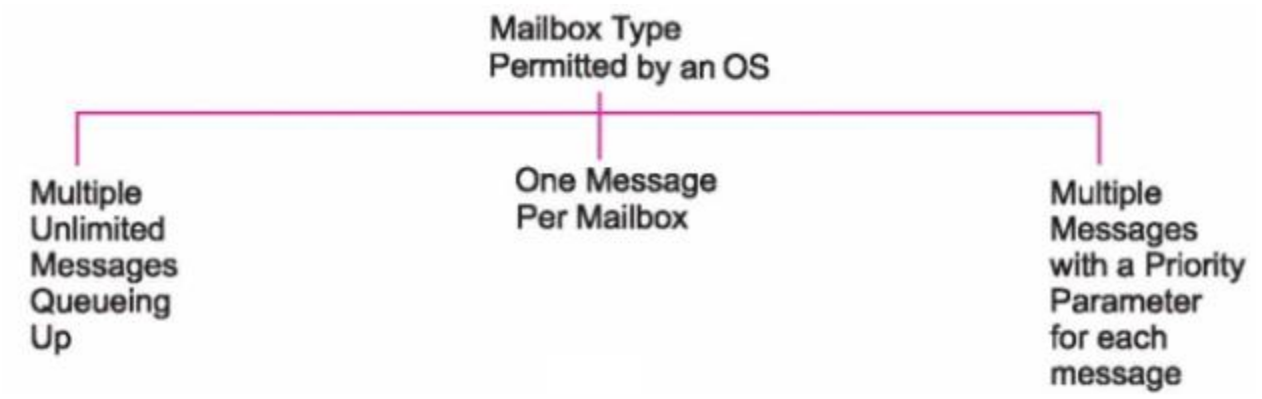
Features

- ➢ Mailbox (for message) is an IPC through a message-block at an OS that can be used only by a single destined task.
- ➢ The source (mail sender) is the task that sends the message pointer to a created mailbox.
- ➢ OS provides for inserting and deleting message into the mailbox message-pointer.
- ➢ Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to Null.

- There may be a provision for multiple mailboxes for the multiple types or destinations of messages.
- Each mailbox has an ID.
- Each mailbox usually has one message pointer only, which can point to message.

Types:

- There are three types of mailboxes as given below.


Mailbox Type Permitted by an OS
Multiple Unlimited Messages Queueing Up
One Message Per Mailbox
Multiple Messages with a Priority Parameter for each message

- One type is only one message per mailbox is available.
- Another type is mailbox with provision for multiple messages or message pointers.
- Third one is OS can provide multiple mailbox messages with each message having a priority parameter. The read (deletion) can then only be on priority basis in case mailbox has multiple messages.

Mailbox Functions

1. OSMBoxCreate creates a box and initializes the mailbox contents with a NULL pointer.
2. OSMBoxPost sends (writes) a message to the box.
3. OSMBoxWait (pend) waits for a mailbox message, which is read when received.
4. OSMBoxAccept reads the current message pointer after checking the presence yes or no (no wait). Deletes the mailbox when read.
5. OSMBoxQuery queries the mailbox when *read* and not needed later.

5.9 Pipes

- Pipe is a device used for the inter process communication.
- A message-pipe is a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.
- Writing and reading from a pipe is like using a C command *fwrite* with a file name to write into a named file, and C command *fread* with a file name to read into a named file.

Write and read using Pipe

- One task using the function *fwrite* in a set of tasks can write to a pipe at the back pointer address, *pBACK.
- One task using the function *fread* in a set of tasks can read (delete) from a pipe at the front pointer address, *pFRONT.
- In a pipe there may be no fixed number of bytes per message but there is end pointer.
- A pipe can therefore be limited and have a variable number of bytes per message between the initial and final pointers.
- Pipe is unidirectional. One thread or task inserts into it and other one deletes from it.

Pipe Functions

- Figure (a) shows the write and read the pipe using device drivers.
- Figure (b) shows the functions for the pipe in the OS.
- Figure (c) shows a pipe messages in the message buffer.

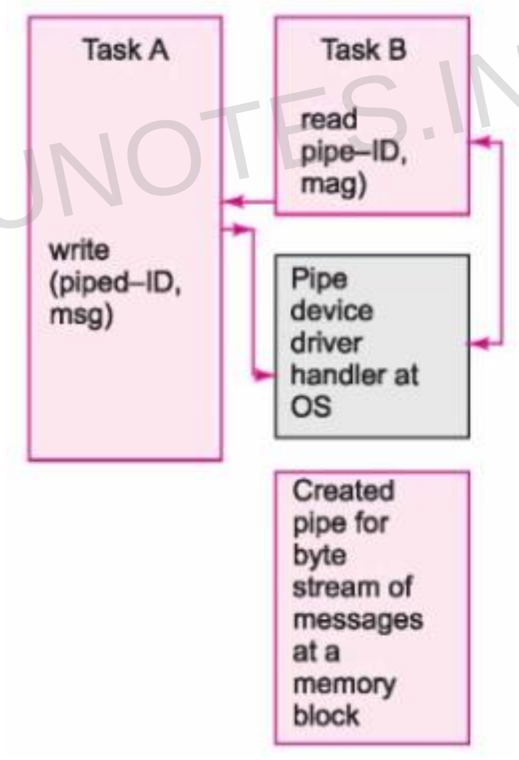

Task A
write (piped–ID, msg)
Task B
read pipe–ID, mag)
Pipe device driver handler at OS
Created pipe for byte stream of messages at a memory block

Figure (a) Pipe handling by tasks

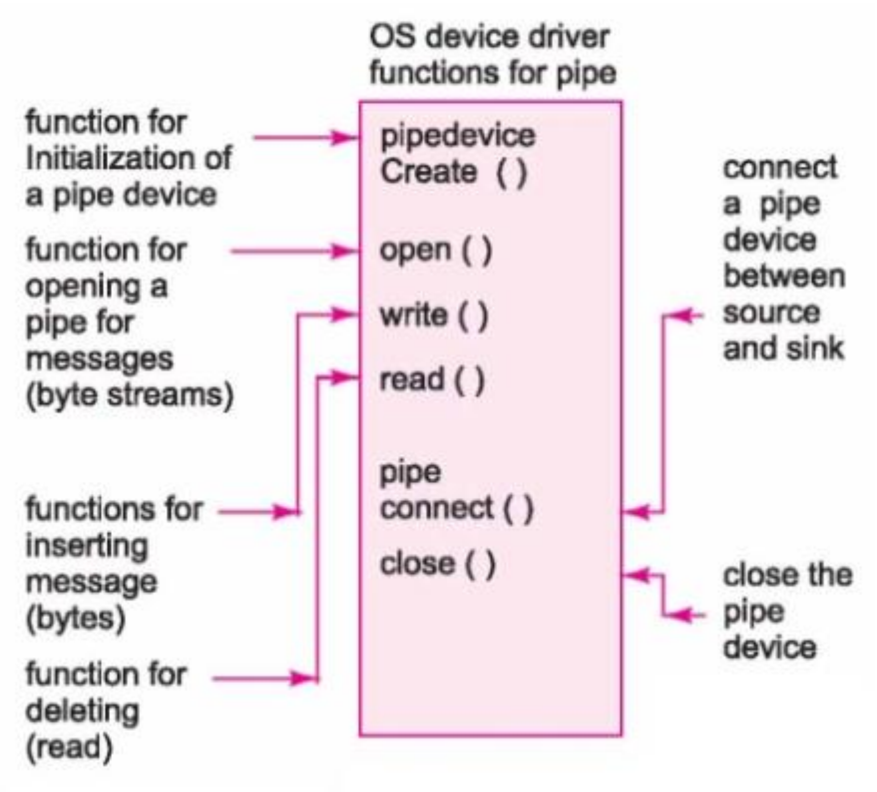

OS device driver functions for pipe
function for Initialization of a pipe device
pipedevice Create ( )
function for opening a pipe for messages (byte streams)
open ( )
write ( )
read ( )
pipe connect ( )
close ( )
connect a pipe device between source and sink
functions for inserting message (bytes)
function for deleting (read)
close the pipe device

Figure (b) Pipe Functions

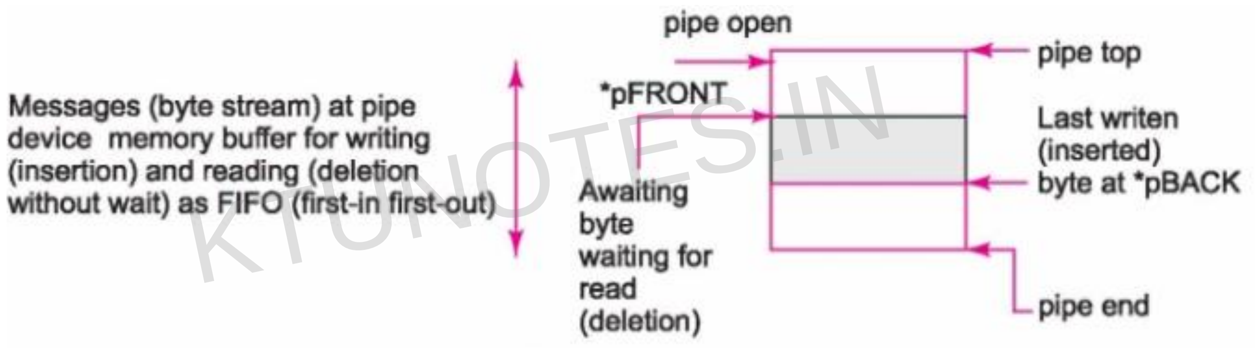

pipe open
pipe top
*pFRONT
Messages (byte stream) at pipe device memory buffer for writing (insertion) and reading (deletion without wait) as FIFO (first-in first-out)
Last writen (inserted) byte at *pBACK
Awaiting byte waiting for read (deletion)
pipe end

Figure (c) Pipe message block

The OS functions for pipe are the following:

1. *pipeDevCreate* for creating a device, which functions as pipe.
2. *open* ( ) for opening the device to enable its use from beginning of its allocated buffer, its use is with options and restrictions (or permissions) defined at the time of opening.
3. *connect* ( ) for connecting a thread or task inserting bytes to the thread or task deleting bytes from the pipe.
4. *write* ( ) function for inserting (writing) from the bottom of the empty memory space in the buffer allotted to it.
5. *read* ( ) function for deleting (reading) from the pipe from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. *close* ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

5.10 Sockets

- Provides a device like mechanism for bi-direction communication.

Need for Sockets for the Inter Process Communication

- A pipe could be used for inserting the byte steam by a process and deleting the bytes from the stream by another process.
- However, for example, we need that the card information to be transferred from a process A as byte stream to the host machine process B and the B sends messages as byte stream to the A
- There is need for bi-directional communication between A and B.
- We need that the A and B ID or address information when communicating must also be specified either for the destination alone or for the source and destination both. [The messages in a letter are sent along with address specification.]
- We need to use a protocol for communication
- A protocol, for example, provides along with the byte stream information of the address or port of the destination or addresses or ports of source and destination both or the protocol may provide for the addresses and ports of source and destination in case of the remote processes.
- For example, UDP (user datagram protocol) is used as connectionless protocol.
- UDP header contains source port (optional) and destination port numbers, length of the datagram and checksum.
- Port means a process or task for specific application.
- Port number specifies the process.
- Datagram means a data, which is independent need not in sequence with the previously sent data.
- Checksum is sum of the bytes to enable the checking of the erroneous data transfer.
- TCP (transport control protocol) is used as connection oriented protocol.

Features:

- Socket Provides for a bi-directional pipe like device, which also use a protocol between source and destination processes for transferring the bytes.
- Provides for establishing and closing a connection between source and destination processes using a protocol for transferring the bytes.
- May provide for listening from multiple sources or multicasting to multiple destinations.
- Two tasks at two distinct places or locally interconnect through the sockets.
- Multiple tasks at multiple distinct places interconnect through the sockets to a socket at a server process.
- The client and server sockets can run on same CPU or at distant CPUs on the Internet.

- Sockets can be using a different domain. For example, a socket domain can be TCP (transport control protocol) , another socket domain may be UDP(transport control protocol), the card and host example socket domain is different.
- Two processes (or sections of a task) at two sets of ports interconnect (perform inter process communication) through a socket at each.
- A socket stream between two specified sections (ports)─at the specified sets (addresses) sent with a port-specific protocol.
- Each socket ─process address (similar to a network or IP address) and section (similar to a port) specification. The sections (or ports or tasks) and sets of processes (addresses) may be on the same computer or on a network.
- There has to be a specific protocol in which the messages at the socket interconnect between (i, j) and (m, n). [i and m are process addresses, and j and n are port or section specifications]
- A pipe does not have protocol based inter-processor communication, while socket provides that.
- A socket can be a client-server socket. Client Socket and server socket functions are different.
- A socket can be a peer-to-peer socket IPC. At source and destination sockets have similar functions.

<u>Socket Functions</u>

- Figure (a) shows the write by a server task and read by a client task using device drivers.
- Figure (b) shows the functions for the socket for both server and client in the OS.
- Figure (c) shows byte stream at socket device.

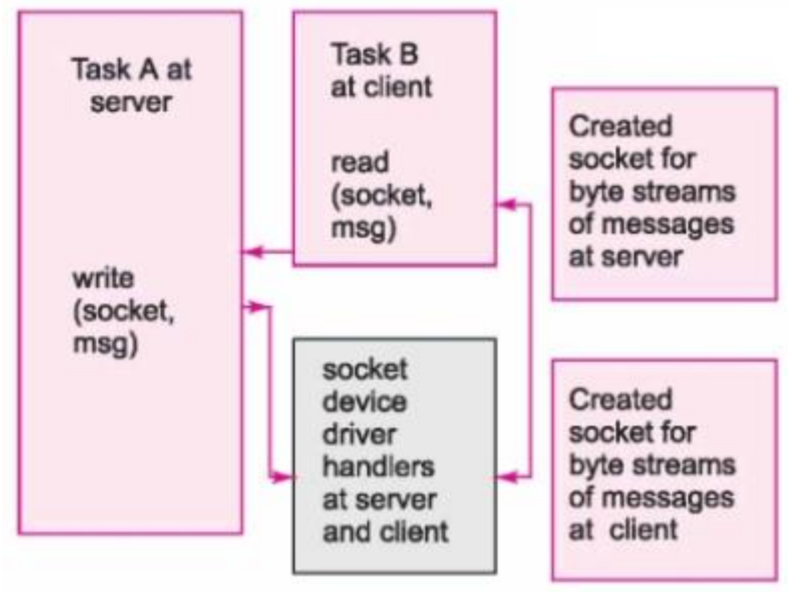

Task A at server
write (socket, msg)
Task B at client
read (socket, msg)
socket device driver handlers at server and client
Created socket for byte streams of messages at server
Created socket for byte streams of messages at client

Figure (a) Socket handling by tasks

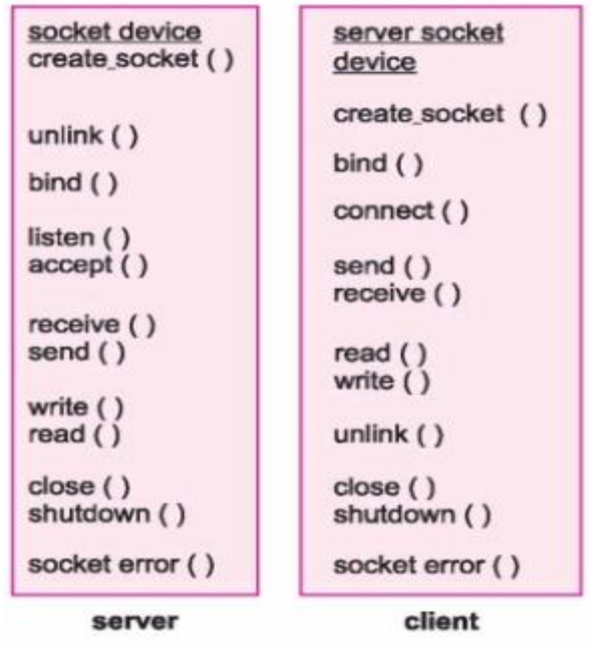


Figure (b) Socket Functions for both server and client.

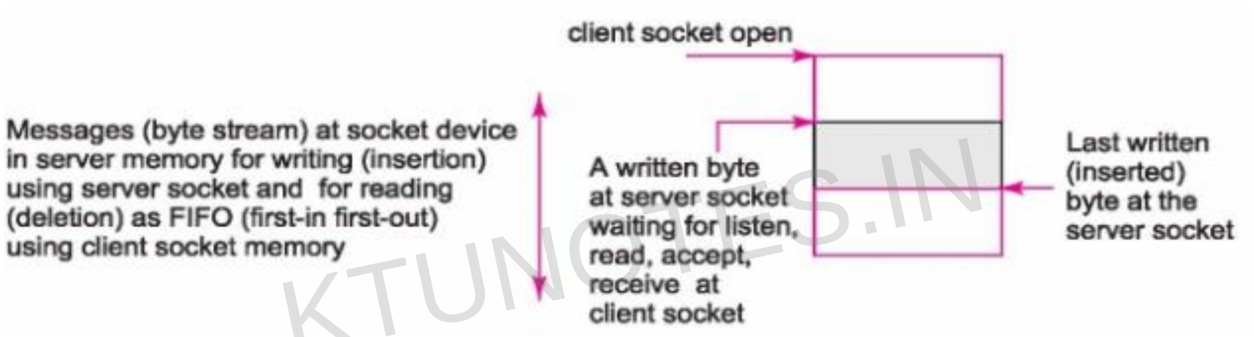

client socket open
Messages (byte stream) at socket device in server memory for writing (insertion) using server socket and for reading (deletion) as FIFO (first-in first-out) using client socket memory
A written byte at server socket waiting for listen, read, accept, receive at client socket
Last written (inserted) byte at the server socket

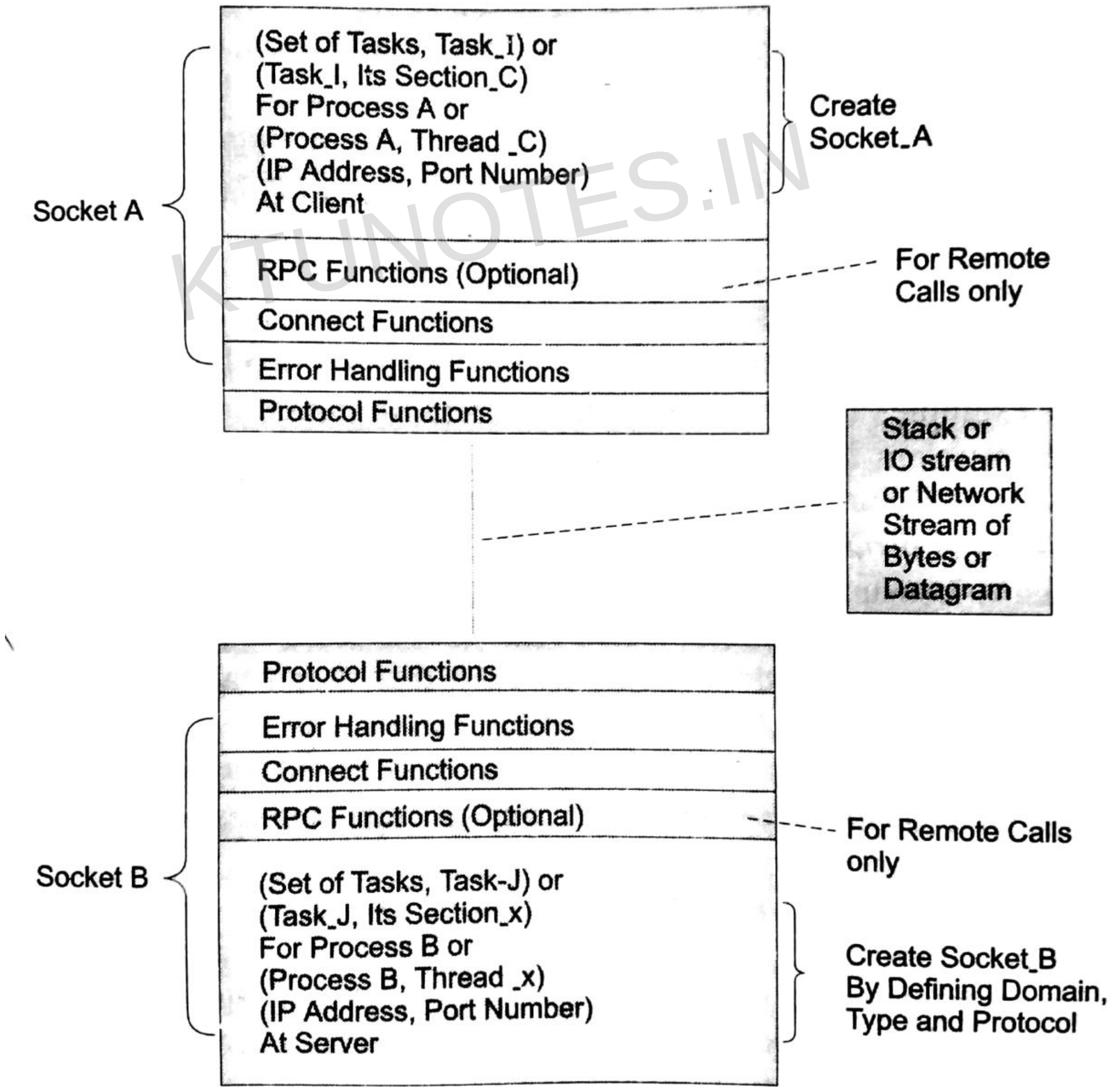
Figure (c) Socket message stream

The OS functions for socket in Unix are as following.

1. The socket ( ) [in place of open ( ) in case of pipe] gives a socket descriptor sfd. The socket ( ) enables its use from beginning of its allocated buffer at the socket address, its use with option and restrictions or permissions defined at the time of opening. A socket can be a stream, SOCK_STREAM or UDP datagram SOCK_DGRAM.
2. The unlink ( ) before the bind ( ).
3. The bind ( ) for binding a thread or task inserting bytes into the socket to the thread or task and deleting bytes from the socket. bind ( ) the socket descriptor to an address in the Unix domain. bind (sfd, (struct sockaddr *)&local, len); where len is string length. sockaddr is a data structure with a record of 16-bit unsigned num and a path for the file and a data structure struct sockaddr_un {unsigned short num; char path[108]; }
4. The listen (sfd, 16) function for listening 16 queued connections from the client socket.
5. The accept ( ) accepts the client connection and gives a second socket descriptor.
6. The recv ( ) function for deleting (reading) and receiving from the socket from the bottom of unread memory spaces in buffer. The buffer has messages after writing into the socket.
7. The send ( ) function for inserting (writing) and sending from the socket from the bottom of the memory spaces in the buffer filled after writing into the socket.
8. The close ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

5.11 RPC (remote procedure call)

- A method used for connecting two remotely placed functions by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.
- The RTOS can provide for the use of RPCs. These permits distributed environment for the embedded systems.
- The OS IPC function allows a function or method to run at another address space of shared network or other remote computer.
- The client makes the call to the function that is local or remote and the server response is either remote or local in the call.
- Both systems work in the peer-to-peer communication mode. Each system in peer-to-peer mode can make an RPC.
- An RPC permits remote invocation of the processes in the distributed systems.
- The RPC provides the inter task communication when a task is at system1 and another task is at system 2.
- Figure below shows the initialised virtual sockets between the client set of tasks and a server set of tasks at RTOS.


(Set of Tasks, Task_I) or
(Task_I, Its Section_C)
For Process A or
(Process A, Thread _C)
(IP Address, Port Number)
At Client
Create Socket_A
Socket A
RPC Functions (Optional)
For Remote Calls only
Connect Functions
Error Handling Functions
Protocol Functions
Stack or IO stream or Network Stream of Bytes or Datagram
Protocol Functions
Error Handling Functions
Connect Functions
RPC Functions (Optional)
For Remote Calls only
Socket B
(Set of Tasks, Task-J) or
(Task_J, Its Section_x)
For Process B or
(Process B, Thread _x)
(IP Address, Port Number)
At Server
Create Socket_B By Defining Domain, Type and Protocol