UNIT 4

RTOS BASED EMBEDDED SYSTEM DESIGN

Part- A (2 marks)

1. Define process.

Process is a computational unit that processes on a CPU under the control of a scheduling kernel of an OS. It has a process structure, called Process control block. A process defines a sequentially executing program and its state.

2. What are the states of a process?

a. Running b. Ready c. Waiting

3. What is a thread?

Thread is a concept in Java and UNIX and it is a light weight sub process or process in an application program. It is controlled by the OS kernel. It has a process structure, called thread stack, at the memory. It has a unique ID .It have states in the system as follows: stating, running, blocked and finished.

4. Define scheduling.

This is defined as a process of selection which says that a process has the right to use the processor at given time.

5. What are the types of scheduling?

1. Time division multiple access scheduling. 2. Round robin scheduling.

6. Define round robin scheduling and priority scheduling

Round robin scheduling: This type of scheduling also employs the hyper period as an interval. The processes are run in the given order.

Priority scheduling: A simple scheduler maintains a priority queue of processes that are in the run able state.

7. Give the different styles of inter process communication.

1. Shared memory. 2. Message passing.

8. Differentiate pre-emptive and non pre-emptive multitasking.

Preemptive multitasking differs from non-preemptive multitasking in that the operating system can take control of *the* processor without the task's cooperation

9. What is meant by PCB?

Process Control Block' is abbreviated as PCB.PCB is a data structure which contains all the information and components regarding with the process.

10. Define Semaphore and mutex.

Semaphore provides a mechanism to let a task wait till another finishes. It is a way of synchronizing concurrent processing operations. When a semaphore is taken by

a task then that task has access to the necessary resources. When given the resources unlock. Mutex is a semaphore that gives at an instance two tasks mutually exclusive access to resources.

11. Define priority inversion & priority inheritance.

Priority inversion: A problem in which a low priority task inadvertently does not release the process for a higher priority task is called as Priority inversion.

Priority Inheritance: Priority inversion problems are eliminated by using a method called priority inheritance. The process priority will be increased to the maximum priority of any process which waits for any resource which has a resource lock. This is the programming methodology of priority inheritance.

12. What is RTOS?

An RTOS is an OS for response time controlled and event controlled processes. RTOS is an OS for embedded systems, as these have real time programming issues to solve.

13. What are the real time system level functions in UC/OS II? State some.

1. Initiating the OS before starting the use of the RTOS functions.
2. Starting the use of RTOS multi-tasking functions and running the states.
3. Starting the use of RTOS system clock.

14. Name any two mailbox related functions.

a.OS_Event *OSMboxCreate(void *mboxMsg)

b.Void *OSMboxAccept(OS_EVENT *mboxMsg)

15. Name any two queue related functions for the inter task communications.

a.OS_Event OSQCreate(void **QTop,unsigned byte qSize)

b.Unsigned byte OSQPostFront(OS_EVENT *QMsgPointer,void *qmsg).

UNIT – 4

PART B(16 marks)

1. Discuss deeply about the pre-emptive & non –pre-emptive scheduling with suitable diagrams.

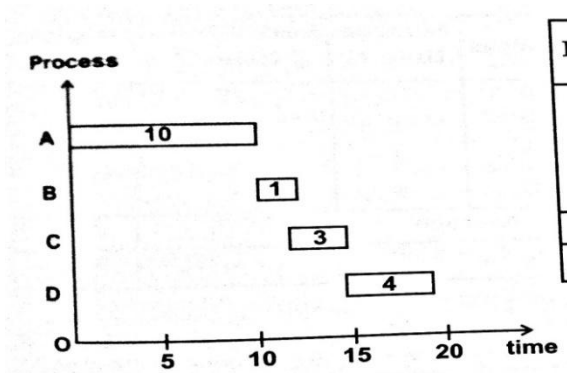
NON – PREEMPTIVE SCHEDULING:

Non- Preemptive scheduling is employed in non-preemptive multitasking systems. In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the wait state waiting for an I/O or system resource. The various types of non-preemptive scheduling algorithms are,

1. First Come First Served (FCFS)

- FCFS also known as first in first out (FIFO) is the simplest scheduling policy. Arriving jobs are inserted into the tail (rear) of the ready queue and process to be executed next is removed from the head (front) of the queue.
- FCFS performs better for long jobs. Relative importance of jobs measured only by arrival time. A long CPU bound job may hog the CPU and may force shorter jobs to wait prolonged periods. This in turn may lead to a lengthy queue of ready jobs and hence to the convoy effect.

Turnaround Time = Waiting Time + Service Time



Process	Service time t_s	Turnaround time t_t	Waiting time t_w
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
Average		13.25	8.75
$t_t/t_w = 1.51$		$t_t/t_s = 5.29$	

Fig. FCFS Example

Advantages:

1. Better for long process
2. Simple method
3. No starvation

Disadvantages:

1. Convoy effect occurs. Even very small process should wait for its turn to come to utilize the CPU. Short process behind long process results in lower CPU utilization.
2. Throughput is not emphasized.

2. Shortest Job First Scheduling (SJF):

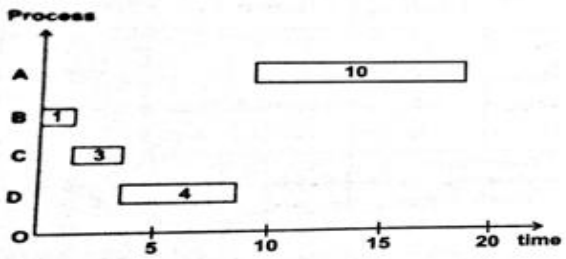
This algorithm associated with each process the length of the next CPU burst. Shortest job first scheduling is also called shortest process next (SPN). The process with the shortest expected processing time is selected for execution among the available process in the ready queue. Thus, a short process will jump to the head of the queue over long jobs.

If the next CPU bursts of two processes are the same then FCFS scheduling is cued to break the tie. SJF scheduling algorithm is probably optimal. It gives the minimum average time for a given set of processes. It cannot be implemented at the level of short

term CPU scheduling. There is no way of knowing the shortest CPU burst. SJF can be preemptive or non-preemptive.

A preemptive SJF algorithm will preempt the currently executing process, if the next CPU burst of newly arrived process may be shorter than what is left to the currently executing process.

A non-preemptive SJF algorithm will allow the currently running process to finish. Preemptive SJF scheduling is sometimes called shortest remaining time first algorithm.



Process	Service time t_s	Turnaround time t_t	Waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
Average		7.75	3.25
$t_t/t_w = 2.38$		$t_t/t_s = 1.83$	

Fig. SJF Example

Advantages:

1. It gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
2. Throughput is high.

Disadvantages:

1. Elapsed time must be recorded, it results an additional overhead on the processor.
2. Starvation may be possible for the longer processes.

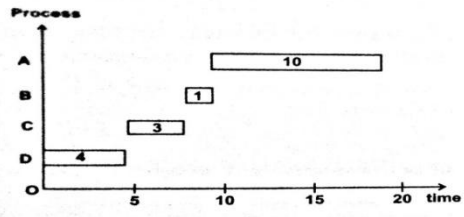
PRE-EMPTIVE SCHEDULING:

- In preemptive mode, currently running process may be interrupted and forces the currently active process to release the CPU on certain events such as a clock interrupt, some I/O interrupts or a system call and they moved to the ready state by the OS.
- When a new process arrives or when a interrupt occurs, preemptive policies may incur greater overhead than non-preemptive version but preemptive version may provide better results.
- It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

The various types of pre-emptive scheduling are

1. Priority – Based Scheduling:

- Each process is assigned a priority. The ready list contains an entry for each process ordered by its priority. The process at the beginning of the list (highest priority) is picked first.
- A variation of this scheme allows preemption of the current process when a higher priority process arrives.
- Another variation of the policy adds an aging scheme, whether the priority of a process increases as it remains in the ready queue. Hence, this will eventually execute to completion.
- If the equal priority process is in running state, after the completion of the present running process CPU is allocated to this, even though one more equal priority process is to arrive.



Process	Priority	Service time t_s	Turn around time t_t	Waiting time t_w
A	4	10	18	8
B	3	1	8	7
C	2	3	7	4
D	1	4	4	0
Average			9.25	4.75
$t_t/t_w = 1.98$			$t_t/t_s = 3.28$	

Fig. Priority Scheduling example

Advantage:

- Very good response for the highest priority process over non-pre-emptive version of it.

Disadvantage:

- Starvation may be possible for the lowest priority processes.

2. Write about the interrupt routines in RTOS.

Interrupts is a mechanism for alleviating the delay caused by uncertainty and for maximizing system performance.

Interrupt Mechanism

- Instead of polling the device or entering a wait state, the CPU continuously executing its instruction and performing useful work.
- When the I/O device is ready to transfer data, it sends an interrupt request to the CPU. This is done via a dedicated signal on the control bus.

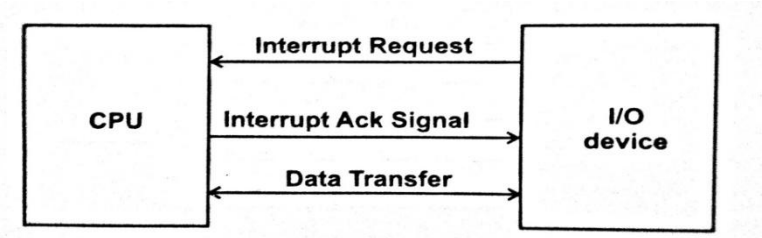


Fig. Handshaking signals of Interrupts

Scenario of Interrupt service Routine in RTOS:

Embedded application running on top of real time operating systems require Interrupt Service Routines (ISRs) to handle interrupt generated by external event. External events can be caused by just about anything, form an asynchronous character arrival on a UART to a periodic timer interrupt. ISRs have the responsibility of acknowledging the hardware condition and provide the initial handling of data sent or received as required by the interrupt.

An ISR often is responsible for providing the RTOS with information necessary to provide services to application threads. Examples include moving data into a buffer for processing, adding an entry to a queue for processing, setting a value to indicate that an event has occurred and so on.

Since application code execution is interrupted during the execution of an ISR, most application minimize the amount of code in the ISR and rely instead on non- ISR code (Thread or Task) to complete the processing. This allows the highest priority application code to be executed as quickly as possible and delayed as little as possible, even in situations with intense interrupt activity.

Interrupt Routines in RTOS Environment and Handling of Interrupt source calls:

- ISRS have the higher priorities over the RTOS function and the tasks. So ISR should not wait for a semaphore, mailbox, message or queue message.
- An ISR should not also wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run.
- Only the IPC accept function for these events can be used, not the post function.

There are 3 alternative systems for the RTOS is to respond to the hardware source calls from the interrupts.

1. Direct call to an ISR by an interrupting source and ISR sending an ISR Enter message:

- On an interrupt, the process running at the CPU is interrupted and the ISR corresponding to that source starts executing (1).

- A hardware source calls an ISR directly. The ISR just sends an ISR enter message to the RTOS. ISR enter message is to inform the RTOS than an ISR has taken control of the CPU (2).

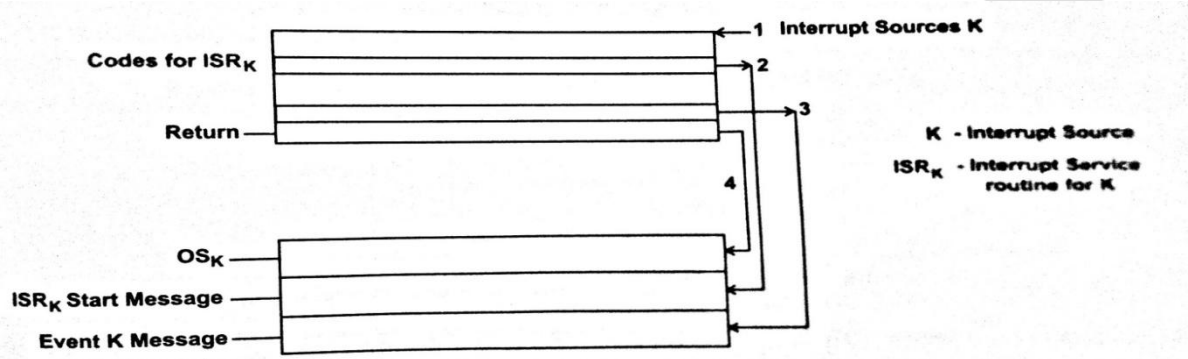


Fig. System-I (two memory blocks)

The case involves the two function such as ISR and OS function in two memory block.

- ISR code can send into a mailbox or message queue(3), but the task waiting for a mailbox or message queue does not start before the return from the ISR (4).
- When ISR finishes, it sends Exit message to OS.
- On return from ISR by retrieving saved context, the RTOS later on returns to the interrupted process or reschedules the process.
- RTOS action depends on the event messages, whether the task waiting for the event messages from the ISR is a task of higher priority than the interrupted task on the interrupt.
- The special ISR semaphore used in this case is OSISRSemPost ( ) which executes the ISR. OS ensures that OSISRSemPost is returned after any system call from the ISR.

2. RTOS first interrupting on an interrupt, then RTOS calling the corresponding ISR:

- On interrupt of a task, say, Nth task, the RTOS first gets itself the hardware source call (1) and initiates the corresponding ISR after saving the present processor status (2).
- Then the ISR (3) during execution then can post one or more outputs (4) for the events and messages into the mail boxes or queues.

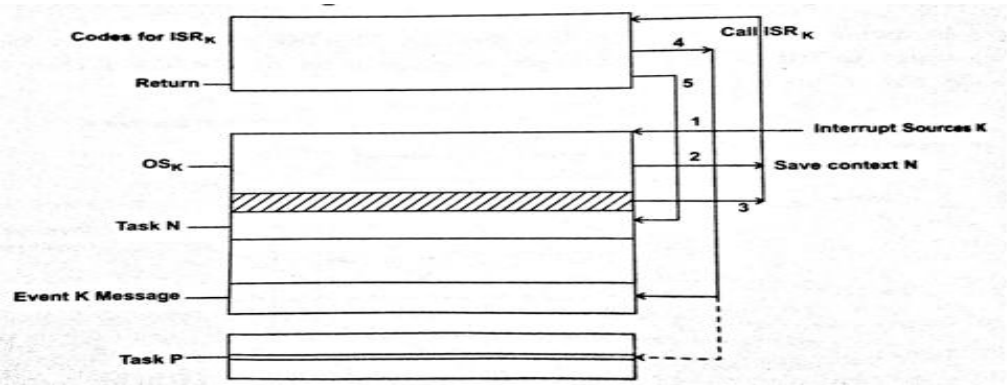


Figure: System 2 (Three memory blocks)

This case involves the one ISR function (ISR_k) and two processes (OS and p^{th} task) in three memory blocks other than the interrupted Nth task.

- When the interrupt source k is interrupted (1), OS finishes the critical code till the pre emption point and calls the ISR routine for interrupt k called as ISR_k (3) after saving the context of a previous task N onto a stack (2)
- After executing the ISRk routine, the ISR in step (4) can post the event or message to the OS for initiating the Nth or P^{th} task after the return (5) from the ISR and after retrieving the Nth or P^{th} task context.
- The OS initiates the Nth or P^{th} task based upon their priorities.
- The ISR must be short and it must put post the messages for another task.

3. RTOS First Interrupting on an Interrupt, then RTOS initiating the ISR and then an ISR:

- The two levels of ISR in RTOSes are Fast Level ISR (FLISR) and Slow Level ISR (SLISR). FLISR is also called as hardware interrupt ISR and SLISR is also called as software interrupt ISR. FLISR is just the ISR in RTOS and SLISR is called as Interrupt Service Thread (IST).
- FLISR reduces the interrupt latency and jitter for an interrupt service. A k^{th} IST is a thread to service an k^{th} interrupt source call. An IST function is referred as deferred procedure call of the ISR.

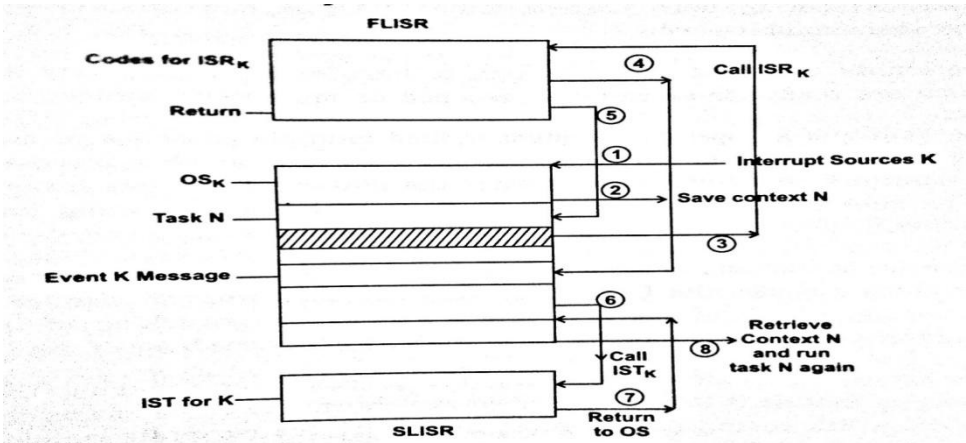


Fig. System 3

- When an interrupt source k is interrupted (1), OS finishes the critical code till the pre emption point and calls the ISR routine for interrupt k called as ISR_k (3) after saving the context of a previous task N onto a stack (2)
- The ISR during execution can send one or more outputs for the events and messages into the mailboxes or queues for the ISTs (4). The IST executes the device and platform independent code.
- The ISR just before the end enables further pre-emption from the same or other hardware sources (5). The ISR can post messages into the FIFO for the ISTs after recognizing the interrupt source and its priority. The ISTs in the FIFO that have received the messages from the ISR executes (6) as per their priorities on return (5) from the ISR.
- The ISR has the highest priority and preempts all pending ISTs and tasks, when no ISR or IST is pending execution in the FIFO, the interrupted task runs on return (7).

3) Write brief notes on i) Process ii) Threads iii) Tasks

i) PROCESS

Defn: Process is defined as a computational unit that processes on a CPU and whose state changes under the control of kernel of an OS. It has a state ,which at an instance defines by the process status(running, blocked or finished),process structure –its data, objects and resources and process control block.

A process runs on scheduling by OS (kernel) which gives the control of CPU to the process. Process runs instructions and the continuous changes of its state take place as the Program counter changes.

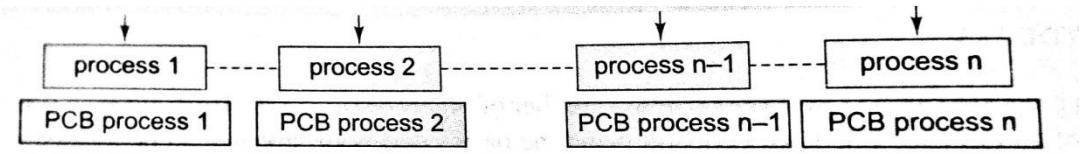


Fig: Processes

Process control block

PCB is a data structure having the information using which the OS controls the process state. The PCB stores in the protected memory addresses at kernel.The PCB consists of the following information about the process state.

1. Process ID,process priority,parent process,child process and address to the next process PCB which will run next.
2. Allocated program memory address blocks in physical memory and in secondary memory for the process codes.
3. Allocated process-specific data address blocks.
4. Allocated process heap addresses.
5. Allocated process stack addresses for the functions called during running of the process.
6. Allocated addresses of the CPU register
7. Process-stae sugnal mask.
8. Signals dispatch table
9. OS-allocated resources descriptors
10. Security restrictions and permissions.

ii) THREAD

Application program can be said to consist of a number of threads or a number of processes and threads

1.A thread consists of sequentially executable program(codes) under state-control by an OS.

2.The state of information of a thread is represented by thread- state(started, running, blocked or finished),thread structure –its data, objects and a subset of the process resources and thread-stack.

3. A thread is a light weight entity

Defn: A thread is a process or sub process within a process that has its own PC, its own SP and stack ,its own priority and its own variables that load into the processor registers on context switching and is processed concurrently along with other threads.

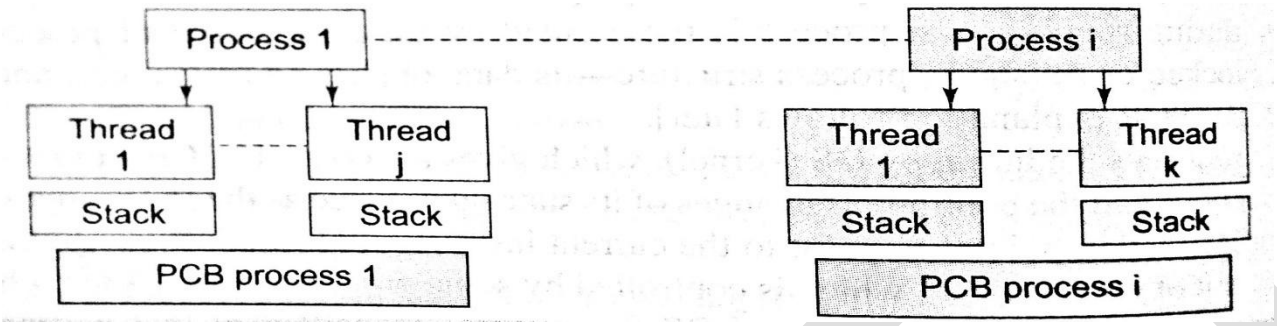


Fig :Threads

- A multiprocessing OS runs more than one process. When a process consists of multiple threads, it is called multithreaded process.
- A thread can be considered as a daughter process.
- A thread defines a minimum unit of a multithreaded process that an OS schedules onto the CPU and allocates the other system resources.
- Different threads of a process may share a common process structure.
- Multiple threads can share the data of the process.
- Thread is a concept used in Java or Unix.
- Thread is a process controlled entity

iii) TASKS

Task is the term used for the process in the RTOSes for the embedded systems. A task is similar to a process or thread in an OS.

Defn: Task is defined as an embedded program computational unit that runs on a CPU under the state-control of kernel of an OS. It has a state, which at an instance defines by status (running, blocked or finished),structure – its data, objects and resources and control block.

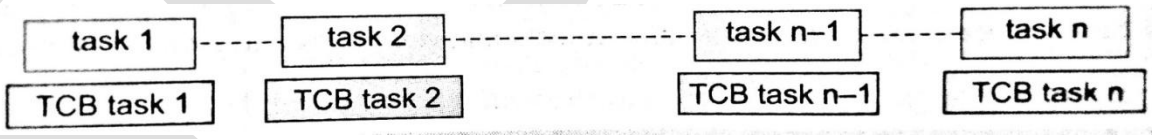


Fig: Tasks

- A task consists of a sequentially executable program under a state-control by an OS.
- The state information of a task is represented by the task state (running, blocked or finished),structure – its data, objects and resources and task control block.
- Embedded software for an application may consist of a number of tasks .
- Each task is independent in that it takes control of the CPU as scheduled by the scheduler at the OS.
- A task is an independent process

- No task can call another task
- The task can send signals and messages that can let another task waiting for this signal.

Task states

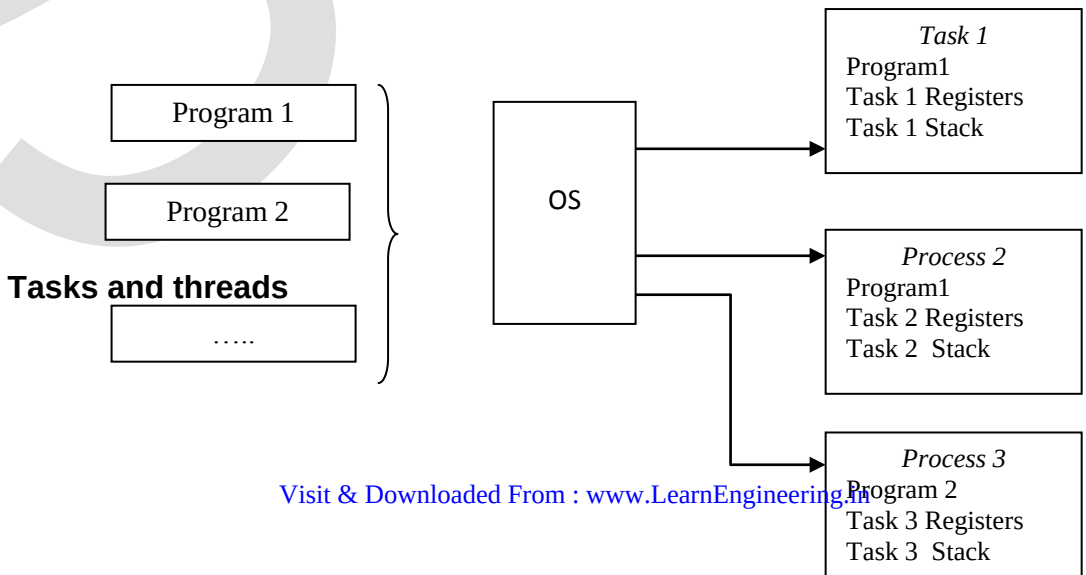
1. Idle(created)state
2. Ready(active)state
3. Running state
4. Blocked(waiting)state
5. Deleted(finished)state

4) Explain briefly on how embedded processors have improved efficiency with use of multitasking.

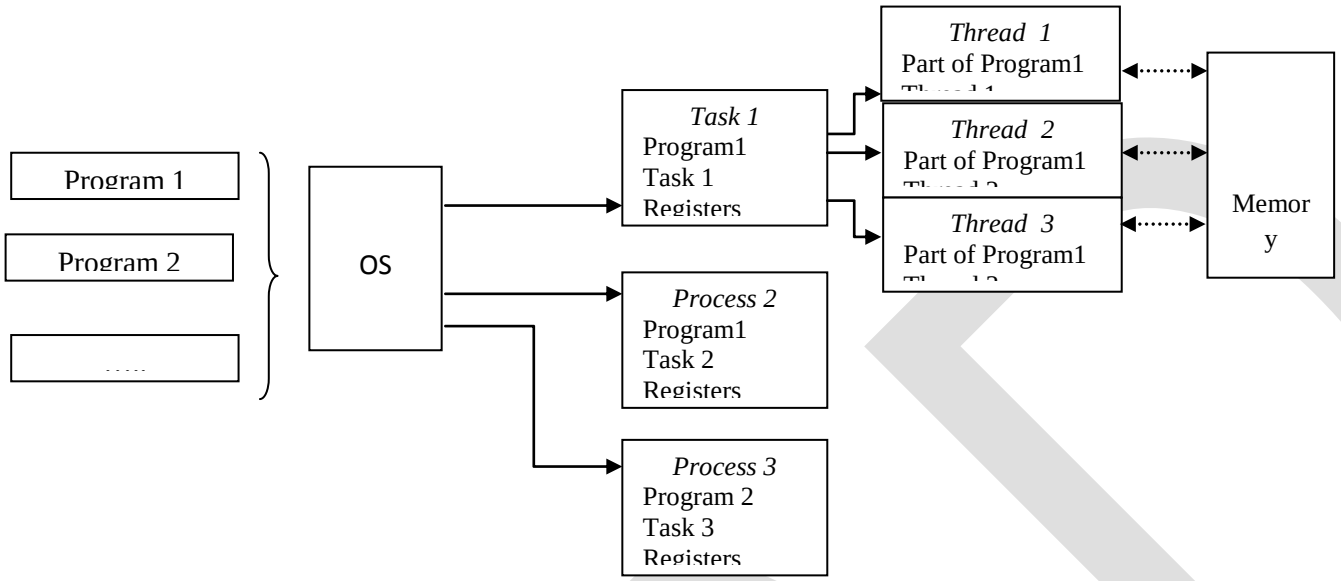
Embedded OSs manages all embedded software using tasks, and can be either unitasking or multitasking.

- In unitasking environments, only one task can exist at any given time.
- In multitasking OS, multiple tasks are allowed to exist simultaneously.
- In multitasking environment each process remain independent of others and do not affect other process.
- Some multitasking OSs also provides threads as an additional alternative means for encapsulating an instance of a program.
- Threads are created within the context of a task and depending on the OS the task can own one or more threads
- Threads of a task share the same resources like working directories, files, I/O devices, etc., but have their own PCs, stack and scheduling information.
- A task can contain at least one thread executing one program in one address space or can contain many threads executing different portions of one program in one address space.

Multitasking OS



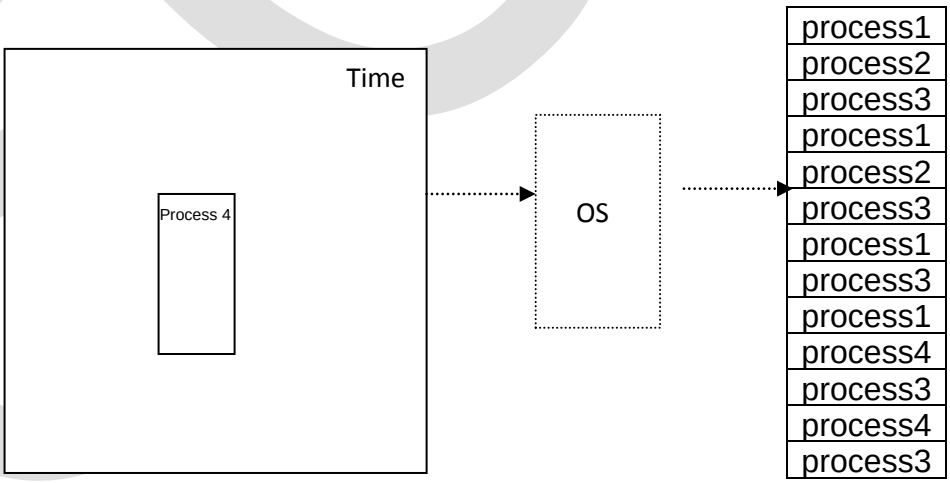
Tasks and threads



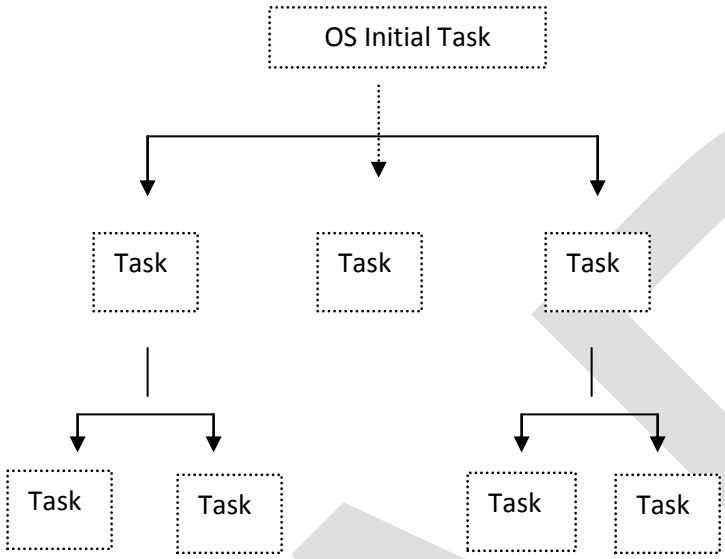
Multitasking and process management

- When an OS allows multiple tasks to coexist, one master processor or an embedded board can only execute one task or thread at any given time.
- Therefore multitasking embedded OSs must find some way of allocating each task a certain amount of time to use the master CPU and switching the master processor between the various tasks
- This is done through task implementation, scheduling, synchronization and inters task communication.
- Gives an illusion that single processors simultaneously run multiple tasks.

Interleaving tasks



Task hierarchy



- Tasks are structured as a hierarchy of parent and child tasks and when an embedded kernel starts up only one task exists.
- All tasks create their child task through system calls.
- The OS gains control and creates the Task Control Block (TCB)
- Memory is allocated for the new child task ,its TCB and the code to be executed by the child task.
- After the task is set up to run, the system call returns and the OS releases control back to the main program.

Types of Multitasking:

Multitasking involves the switching of execution among multiple tasks. It can be classified into different types.

1. ***Cooperative multitasking –*** this is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can hold the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as cooperative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.
2. ***Preemptive multitasking:*** This ensures that every task gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. In preemptive scheduling, the

currently running task is pre empted to give a chance to other tasks to execute. The preemption of task may be based on time slots or task/process priority.

3. ***Non-Preemptive Multitasking:*** In non- preemptive multitasking, the task which is currently given the CPU time is allowed to execute until it terminates or enters the Blocked/wait state waiting for an I/O or system resource. The cooperative and non-preemptive multitasking differs in their behavior when they are in the blocked/wait state.

In cooperative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/wait' state waiting for an I/O or a shared resource access or an event to occur where as in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

5) Explain briefly about inter process communication. A) Semaphores B) Mailbox C) Pipe

SEMAPHORE

Suppose that there are two trains. Assume that they use an identical track. When the first train A is to start on the track, a signal or token for A is set and the signal or token for the other train B is reset.

- Semaphore is used for signaling or notifying of a certain action and also for notifying the acceptance of the notice or signal.
- Release of a token is the occurrence of the event and acceptance of the token is taking note of that event.
- Let a Boolean variable, s represent a semaphore.
- s increments from 0 to 1 for signaling or notifying occurrence of an event from a section of codes in a task or thread.
- s decrements from 1 to 0 when the event is taken note by a section in another task waiting for that event and the waiting task codes start at another action

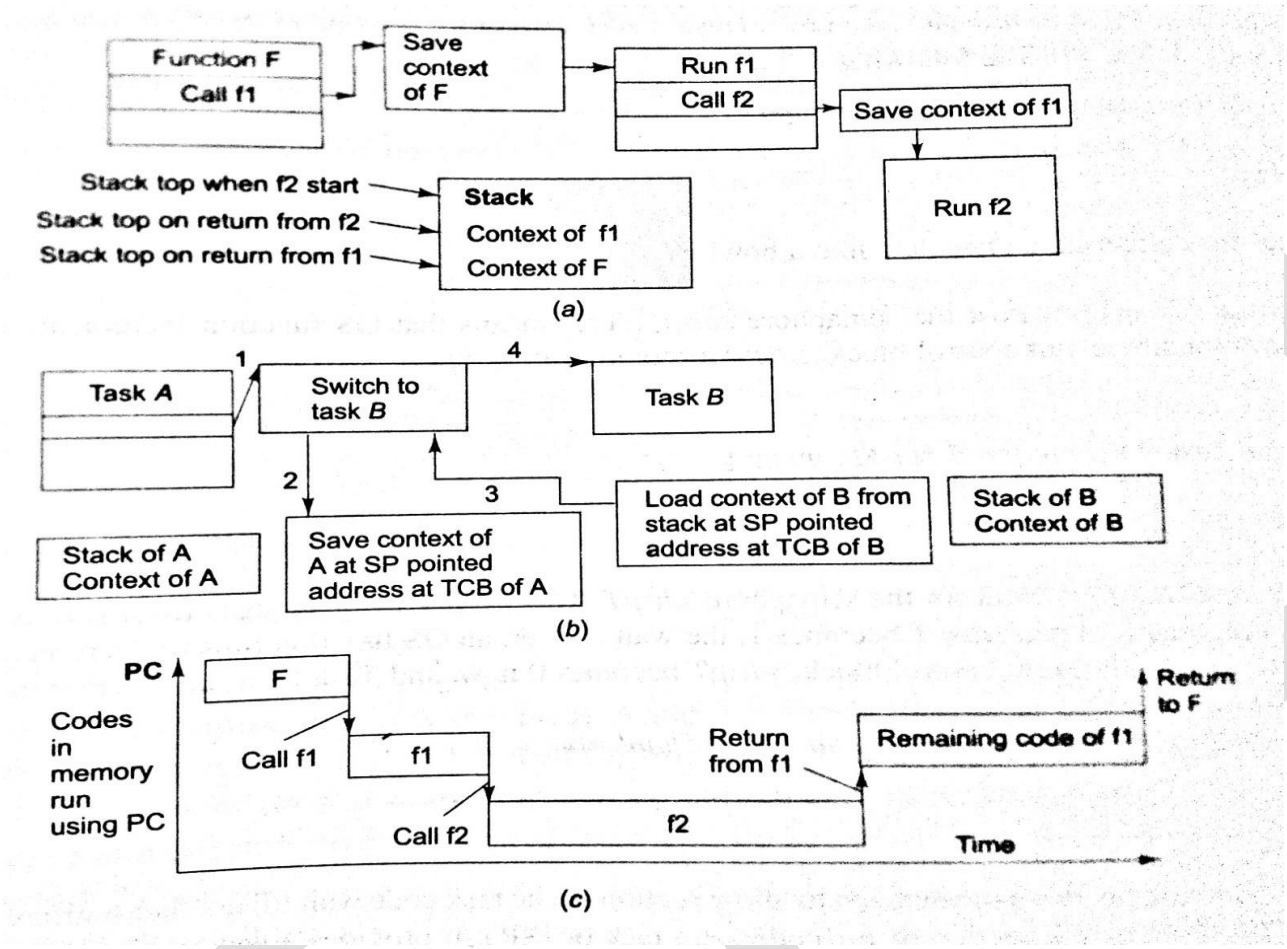


Fig: Action on the function call in a program, Action on pre-emption of task A by B, program counter assignments

Binary Semaphore

A semaphore is called binary semaphore when its value is 0, it is assumed that it has been taken or accepted and when it is 1, it is assumed that it has been released or sent or posted and no task has taken it yet.

- An ISR can release the token, A task can release the token as well accept the token or wait for taking the token.

Example

Consider an Automatic Chocolate Vending Machine(ACVM).After the task delivers the chocolate , it has to notify to the display task to run a waiting section of the code to display , “Collect the nice chocolate, Thank you, Visit Again”. The waiting section for the display of the thank you message takes this notice and then it starts the display of thank you message.

MAILBOX

- A message mailbox is for an IPC message that can be used only by a single destined task.
- The mailbox message is a message pointer or can be a message.
- The source(mail sender) is the task that sends the message pointer to a created mailbox .
- The destination is the place where the OSMBoxPend function waits for the mailbox message and reads it when received.

Example:

A mobile phone LCD display task

- In the mailbox, when the time and data message from a clock-process arrives , the time is displayed at side corner on top line.
- When the message is from another task to display a phone number, it is displayed at the middle.
- When the message is to display the signal strength of the antenna, it is displayed at the vertical bar on the left.

Mailbox types at the different operating systems (OSes)

1. Multiple Unlimited Messages Queueing up
2. One message per mailbox
3. Multiple messages with a priority parameter for each message

- A queue may be assumed a special case of a mailbox with provision for multiple messages or message pointers.
- An OS can provide for queue from which a read can be on a FIFO basis
- Or an OS can provide for multiple mailbox messages with each message having a priority parameter.
- Even if the messages are inserted in a different priority, the deletion is as per the assigned priority parameter.

OS functions for the Mailbox

- Create
- Write(Post)
- Accept
- Read(Pend)
- Query

RTOS functions for the Mailbox

1. **OSMBoxCreate** : Creates a box and initializes the mailbox contents with a NULL pointer
2. **OSMBoxPost**: Sends a message to the box
3. **OSMBoxAccept**: Reads the current message pointer after checking the presence yes or no.Deletes the mailbox when read.
4. **OSMBoxWait**:Waits for a mailbox message , which is read when received.
5. **OSMBoxQuery**: Queries the mailbox when read and not needed later.

PIPE

A message-pipe is a device for inserting (writing) and deleting(reading) from that between two given interconnected tasks or two set 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 *fread* with a file name to read from the named file. Pipes are also like Java PipeInputOutputStreams.

1. *fwrite* task can insert(write) to a pipe at the back pointer address,*pBACK
2. *fread* task can delete (read) from a pipe at the front pointer address,*pFRONT
3. In a pipe there are no fixed number of bytes per message, but there is a end pointer. Pipe can have a variable number of bytes per message between the initial and final pointers.
4. Pipe is unidirectional. One thread or task inserts into it and the other one deletes from it.

Functions at Operating systems

The OS functions for pipe are the following :

1. **pipeDevCreate** : Create a device which functions as pipe.
2. **open( )** :Opening the device to enable its use from beginning of its allocated buffer
3. **connect( )** :Connecting a thread or task by inserting bytes to the thread or task and deleting bytes from the pipe.
4. **write ( )** :Inserting from the bottom of the empty memory space in the buffer allotted to it.
5. **read( )** :Deleting from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. **close( )** :Closing the device to enable its use from beginning of its allocated buffer only after opening it.

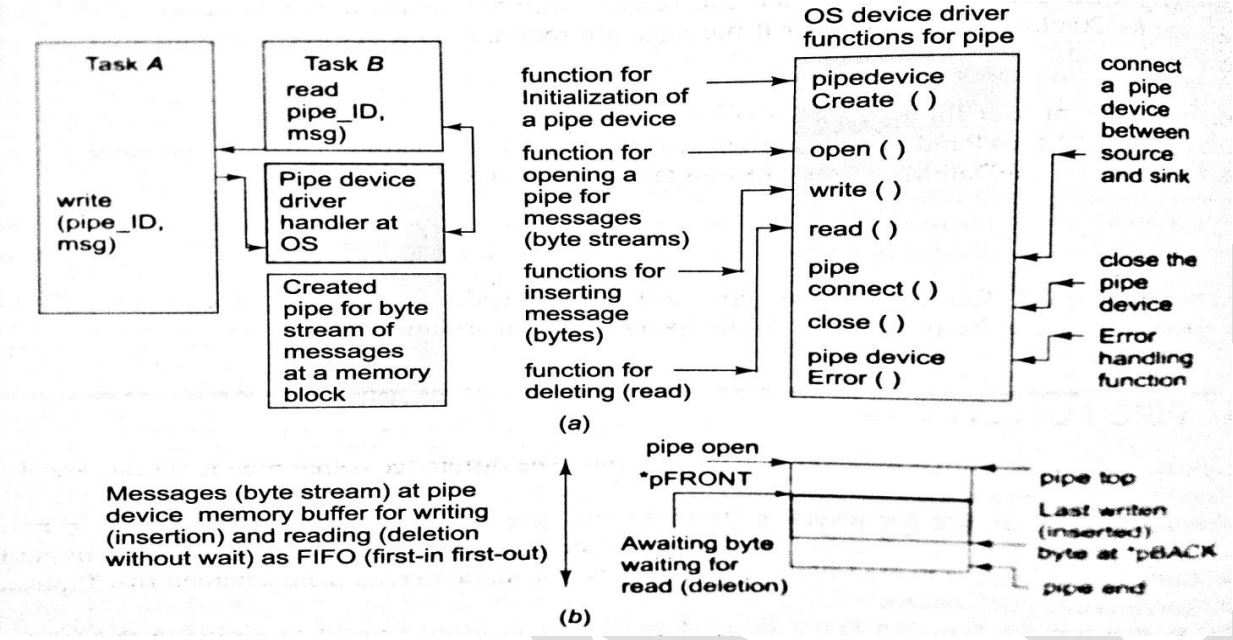


Fig: a) Function at operating system and use of write and read function by task A and task B, b)Pipe messages in a message buffer

6. Write short notes on i) shared memory, ii) message passing, iii) priority inheritance iv) priority inversion.

Shared Memory:

Shared Memory is an efficient means of passing data between programs. One program will create a memory portion which other processes can access.

A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two process and their address spaces. The rectangle box is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is past of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.

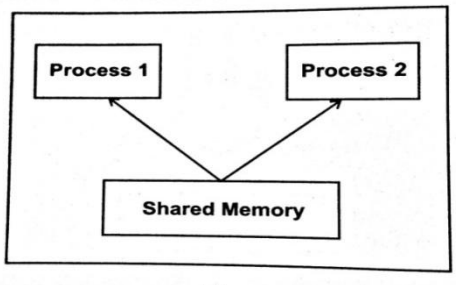


Fig.

Shared memory is a feature supported by UNIX system 5, including Linux, Sun OS and Solaris. One process must explicitly ask for an area, using a key to be shared by other processes. This process will be called the server. All other processes, the clients that know the shared area can access it. However, there is no protection to a shared memory and any process that knows it can access it freely. To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

A shared memory segment is identified by a unique integer, the shared memory ID. The shared memory itself is described by a structure of type shmid-ds in header file sys / shm.h. To use this file, files sys / types.h and sys / ipc.h must be included. Therefore, your program should start with the following lines:

include <sys / types.h>

include <sys / ipc.h>

include <sys / shm.h>

A general scheme of using shared memory is the following:

- For aserver, it should be started before any client. The server should perform the following tasks:
    1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call shmget ().
    2. Attach this shared memory to the server's address space with system call shmat ().
    3. Initialize the shared memory, if necessary.
    4. Do something and wait for all client's completion.
    5. Detach the shared memory with system call shmdt ().
    6. Remove the Shared memory with system call shmctl ().
- For the client part, the procedure is almost the same:
    1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
    2. Attach this shared memory to the client's address space.
    3. Use the memory.
    4. Detach all shared memory segments, if necessary.
    5. Exit.

Message Passing:

Most general purpose OS actually copy messages as twice as they transfer there from task to task via a message queue. In RTOS, the OS copies a pointer to the message,

delivers the pointer to the message - receiver task and then deletes the copy of the pointer with message sender task.

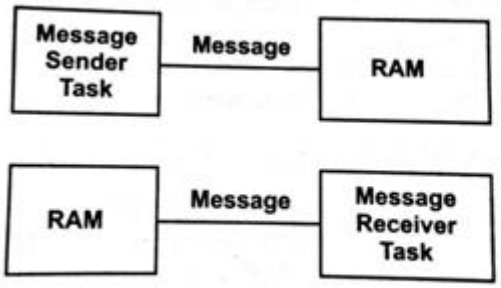


Fig. Message Passing in OS.

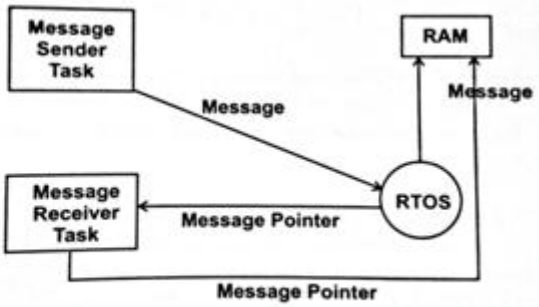


Fig. Message passing in RTOS

Message Passing is an asynchronous / synchronous information exchange mechanism used for Inter process or thread communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared where as only limited amount of info / data is passed through message passing. Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory.

Message passing constructs:

There are 2 basic message passing primitives such as send and receive.

Send primitive: Sends a message on a specified channel from one process to another,

Receive primitive: Receives a message on a specified channel from other processes.

Message passing is classified into message queue, Mail box and signalling.

Message Queue:

The process which wants to talk to another process posts the message to a First-In-First-Out queue called message queue, which stores the messages temporarily in a system defined memory object to pass it to the desired process.

- Messages are sent and received through send and receive methods. the messages are exchanged through a message queue.
- The message mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance from the thread to which the message is posted.
- In synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is

posted.

The features of a message queue IPC are

1. An OS provides for inserting and deleting the message pointers or messages.
2. Each queue for the message needs initialization before using functions in kernel for the queue.
3. Each Created queue has an ID.
4. Each queue has a user defined size.
5. When an OS call is to insert a message into the queue, the bytes are as per the pointed number of bytes.
6. When a queue becomes full, there is error handling function to handle that.

There is a presence of two pointers for queue head and tail memory locations are.

- Q HEAD and
- Q TAIL

The µC/OS - II Functions for a queue are

1. OSQ Create 2. OSQ POST 3. OSQ PEND 4. OSQ ACCEPT 5. OSQ FLUSH 6. OSQ QUERY 7. OSQ POST Front.

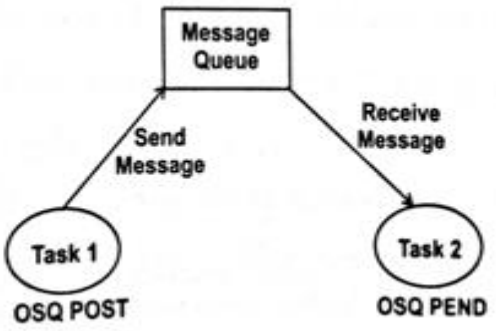


Figure: OS functions for Queue between task 1 and task 2

Priority Inversion

Priority inversion is the byproduct of the combination of blocking based process synchronization and preemptive priority scheduling. 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 share 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 pre emptying the low priority task.

Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronization mechanism ensures that a process will not access a shared resource, which is currently in use by

another process. The synchronization technique is used to avoid conflients in the concurrent access of the shared resources.

- 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 synchronized through a mutual exclusion mechanism like Binary semaphore.
- Imagine a situation where process C is ready and is picked up for execution by the schedules and Process C tries to access the 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 process A is of higher priority than process B, Process B is preempted and process A is scheduled for execution. Process A tries to access shared variable X which is currently accessed by process C. So Process A will not be able to access if and then it enters into the blocked state.
- Now process B gets the CPU and it continued its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task.
- 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 the high priority task which is supposed to be executed immediately when it was ready.
- Priority Inversion may be sporadic in nature but can lead to potential damages as a result of missing critical dead lines.

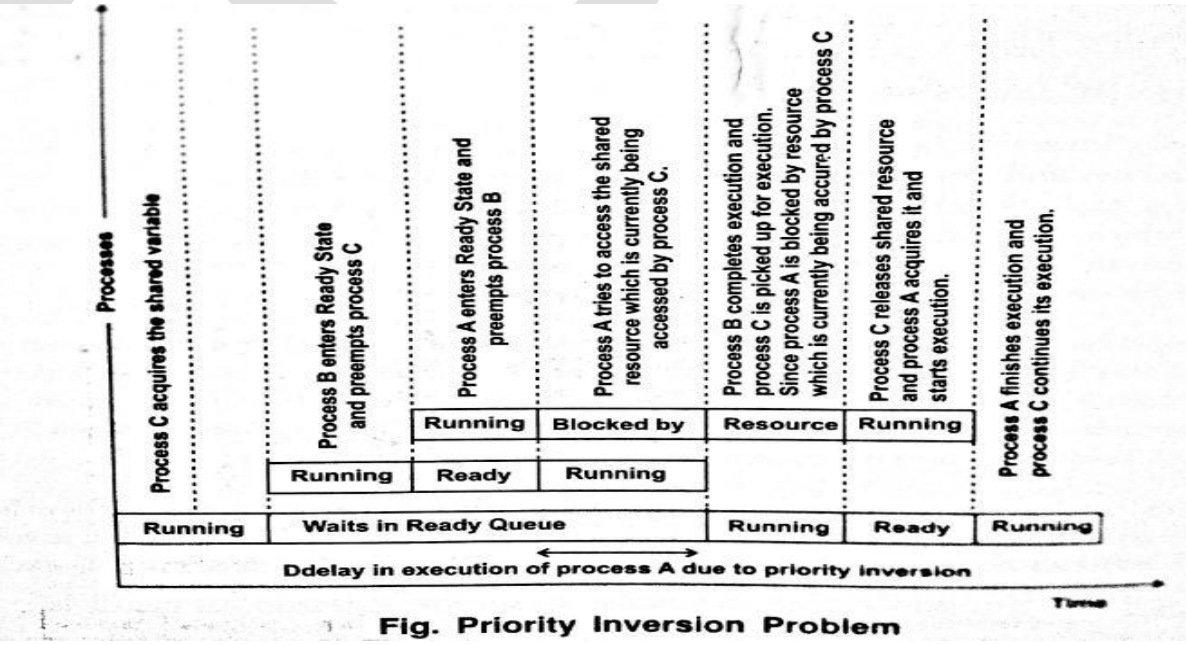


Fig. Priority Inversion Problem

Priority Inheritance

A low-priority task that is currently accessing (by holding the lock) a shared resource requested by 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 to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other task 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 the high priority task. The priority of the low priority task which is temporarily boosted to high is brought to the original value when it relaeases the shared resource. Implementation of priority inheritance work around in the priority inversion problem discussed for process A, Process B and process C example will change the execution sequence as shown in figure.

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 that it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPI – A true form of selfishness. Priority inheritance handles priority inversion at the cost of run-time overhead at schedules. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust.

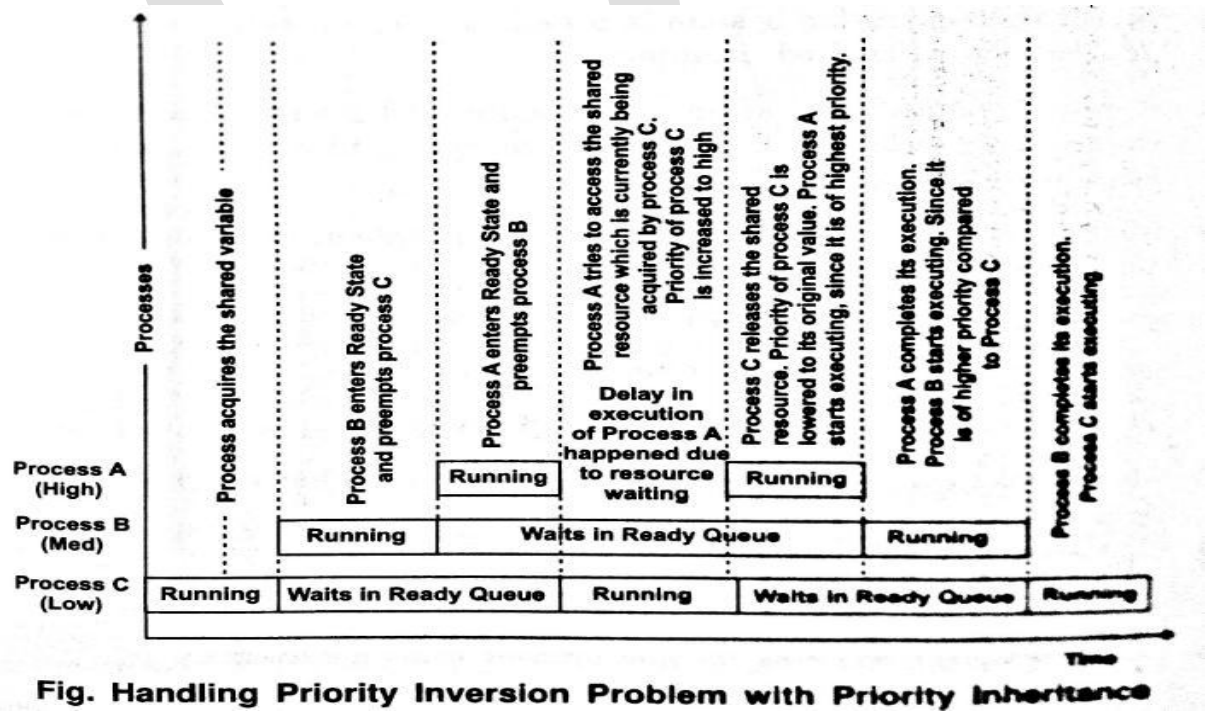


Fig. Handling Priority Inversion Problem with Priority Inheritance

7. Write about the features of µC/OS- II , Vx Works and RT Linux .

i) µC/OS – II

- µC/OS – II is a highly portable, ROMable, very scalable, preemptive real time, deterministic, multitasking kernel.
- It can manage up to 64 tasks (56 uses task available)
- It has connectivity with µc/GUI and µc/ FS (GUI and file systems for µc/OS – II)
- It is ported to more than 100 microprocessors and microcontrollers.
- It is simple to use and simple to implement but very effective compared to the price/performance ratio.
- It supports all type of processors from 8-bit to 64-bit.

Task Management Services:

- Task features
- Task creation
- Task stack and stack checking
- Task deletion
- Change a task's priority
- Suspend and resume a task
- Get information about a task

Task feature:

- µc/OS- II can manage up to 64 tasks.
- The four highest priority tasks and the four lowest priority tasks are reserved for its own use. This leaves us with 56 application tasks.
- The lower the value of the priority, the higher the priority of the task.
- The task priority number also serves as the task identifier.

Rate monotonic scheduling:

- In rate monotonic scheduling tasks with the highest rate of execution are given the highest priority.

Assumptions

1. All tasks are periodic
2. Tasks do not synchronize with one another, share resources, etc.
3. Preemptive scheduling is used (always runs the highest priority task that is ready)

- Under these assumptions, let n be the number of tasks, Ei be the execution time of task i, and Ti be the period of task i. Then, all deadlines will be met if the following inequality is satisfied.

  $\Sigma Ei/Ti \leq n(2^{1/n} - 1)$

- Example: suppose we have 3 tasks. Task 1 runs at 100Hz and task 2 ms. Task 2 runs at 50Hz and takes 1ms. Task 3 runs at 66.7 Hz and takes 7 ms. Apply RMS theory.

  $(2/10) + (1/20) + (7/15) = 0.707 \leq 3(2^{1/3} - 1) = 0.780$

  Thus, all the deadlines will be met.

General solution:

As n goes infinity, the right hand side of the inequality goes to in (2) = 0.6931. Thus you should design your system to use less than 60 – 70% of the CPU.

Task creation:

Two functions for creating a task are

- OS task create ( )
- OS task create E X t ( )

Task Management:

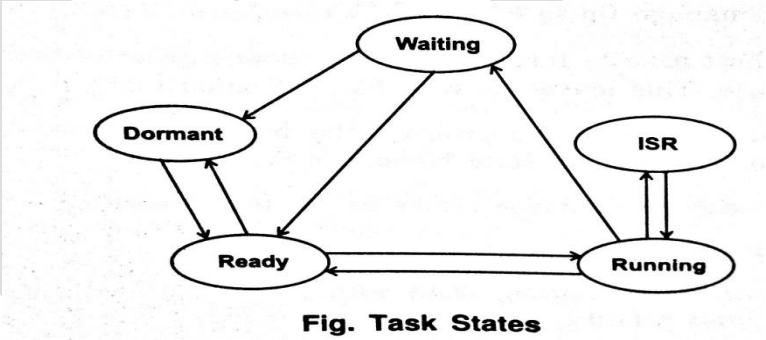


Fig. Task States

- After the task is created, the task has to get a stack in which it will store its data.
- A stack must consist of contiguous memory locations.
- It is necessary to determine how much stack space a task actually uses.
- Deleting a task means the task will be returned to its dormant state and does not mean that the code for the task will be deleted. The calling task can delete itself.
- If another task tries to delete the current task, the resources are not freed and thus are lost. So the task has to delete itself after it uses its resources.
- Priority of the calling task or another task can be changed at run time.
- A task can suspend itself or another task; a suspended task can resume itself.

- A task can obtain information about itself or other tasks. This information can be used to know what the task is doing at a particular time.

Memory Management:

The memory management includes:

1. Initializing the memory manager
2. Creating a memory partition
3. Obtaining status of a memory partition
4. Obtaining a memory block
5. Returning a memory block
6. Waiting for memory blocks from a memory partition.

- Each memory partition consists of several fixed – sized memory blocks.
- A task obtains memory blocks from the memory partition.

A task must create a memory partition before it can be used.

- Allocation and de-allocation of these fixed – sized memory blocks is done in constant time and is deterministic.
- Multiple memory partitions can exist, so a task can obtain memory blocks of different sizes.
- A specific memory block should be returned to its memory partitions from which it came.

Time management:

- *Clock Tick:* A clock tick is a periodic time source to keep track of time delays and time outs.
  - Time intervals: 10~100 ms
  - The faster the tick rate, the higher the overhead imposed on the system.
- Whenever a clock ticks occurs, µC/OS – II increments a 32 – bit counter.
  - The counter starts at zero and rolls over to 2^{32-1} ticks.
- A task can be delayed and a delayed task can be also be resumed.
- Five services:
  - OSTime_DLY ( )
  - OSTime_DLYHMSM ( )
  - OSTime_DLYResume ( )
  - OSTime_GET ( )
  - OSTime_Set ( )

Inter task communication:

Inter task or inter process communication in µC/OS takes place using,

- Semaphore
- Message mailbox
- Message queues

Tasks and interrupt service routines (ISR) can interact with each other through an ECB (Event Control Block)

Single task waiting:

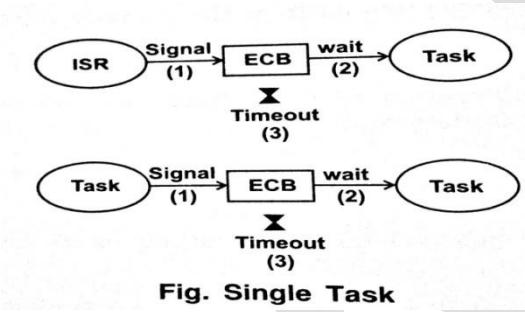


Fig. Single Task

Multiple tasks waiting and signalling:

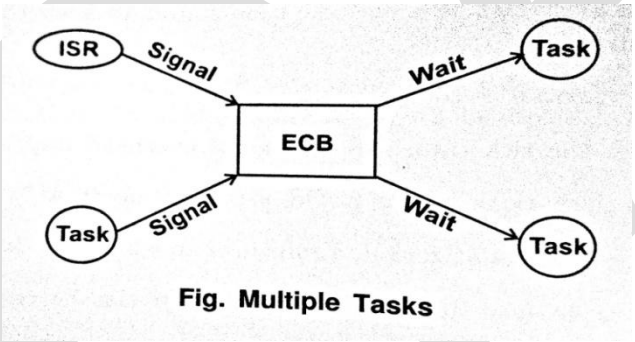


Fig. Multiple Tasks

Tasks can wait and signal along with an optional time out.

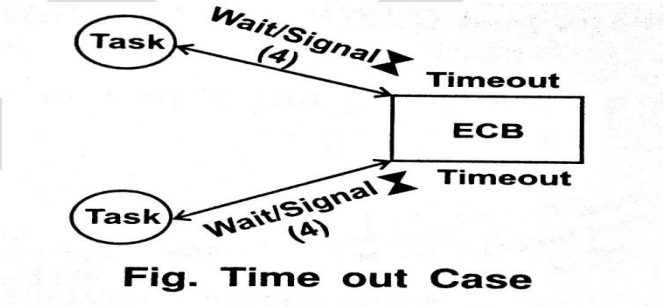


Fig. Time out Case

- µC/OS-II semaphores consist of two elements
  - 16 – bit unsigned integer count.
  - List of tasks waiting for semaphore.
- µC/OS – II provides

  Create, post, pend accept and query services.

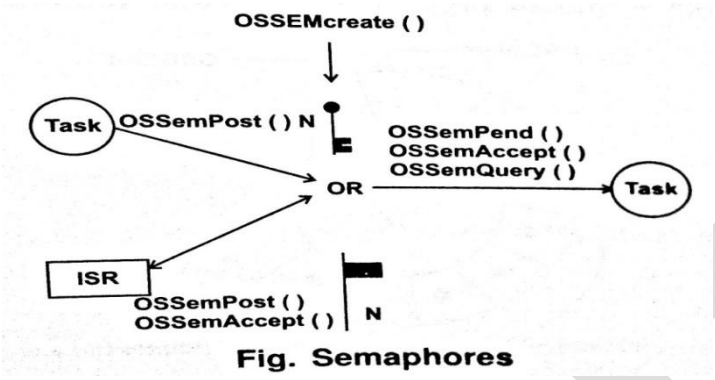


Fig. Semaphores

- µC/OS – II message mailbox: an µC/OS – II object that allows a task or ISR to send a pointer sized variable (pointing to a message) to another task.

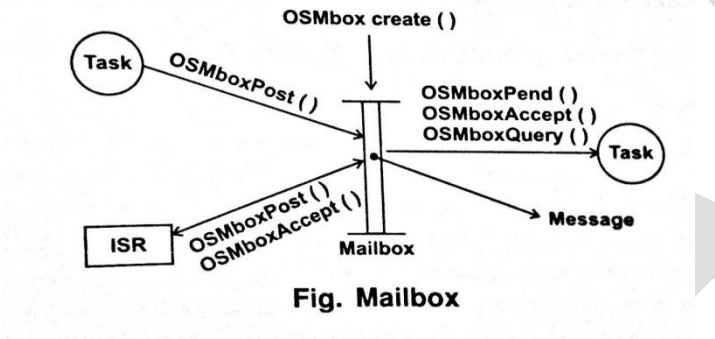


Fig. Mailbox

- µC/OS – II message – queues
- Available services: create, post (FIFO), postfront (LIFO), pend, accept – query, flush.

N = Addition of entries in the queue: queue full if post or post front called N times before a pend or accept.

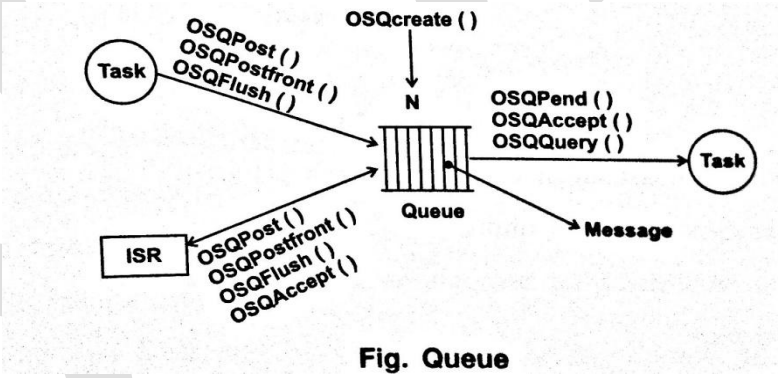


Fig. Queue

- µC/OS – II message – queue organized at circular buffers.

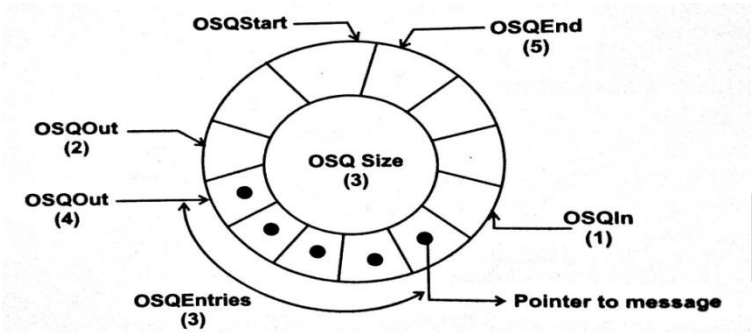


Fig. Circular buffer

ii) Vx Works RTOS:

Introduction:

Vx works is a real time operating system developed as proprietary software by wind river systems. It in a high performance, Unix like, scalable RTOS and supports ARM, Pentium, Intel X-scale, super H and other popular processors for embedded system design. Vx works design is hierarchical and well suited for real time applications. It support kernel mode execution of tasks for fast execution of application codes. Vx works is supported with powerful development tools that make it easy and efficient to use.

Many simulation tools, time performance analysis tools, debug and test tools are provided. Making Vx works as an RTOS that supports development of almost any embedded application, providing a complete solution for all stages of the design cycle. The latest version Vx works 6.9 in the first commercial grade RTOS to fully support both 32-bit and 64-bit processing on Intel Architecture.

Basic Features:

1. Multi tasking environment using standard POSIX scheduler.
2. Ability to run two concurrent operating systems on a single processing layer.
3. Multiple file systems and systems that enable advanced multimedia functionality.
4. Synchronization using a full range of IPC option.
5. Different context saving mechanisms for the tasks and ISRs.
6. Virtual IO devices including pipes and sockets.
7. Victual memory management functions.
8. Power management functions to enhance the ability to control power consumption.
9. Interconnecting functions that support large number of protocols.
10. Pipe drivers for DPCS.
11. Network transparent sockets.

12. Network derivers for shared memory and Ethernet.
13. RAM dist drivers for memory resident files.
14. Processor arbitration layer to enable application system design by user when using new versions of processor architecture.

Architecture:

Vx Works was initially a development and network environment for VRTX. Later wind River systems developed. Their own micro kernal. So the Vx works is of "client-server" architecture from the beginning. The heart of the Vx works run-time system is the wind microkernel. This micro kernel supports a full range of real-time features including multi-tasking, scheduling. Inter task synchronization communication and memory management. All the other functionality is implemented as processes.

Vx Works is highly scalable. By including or excluding various models, Vx works can be configured for the use in small embedded system with tough memory constraints to complex systems where more functions are needed. Furthermore, individual modules themselves are scalable. Individual functions may be removed from the library or specific kernel synchronization objects may be omitted if they are not required by the application.

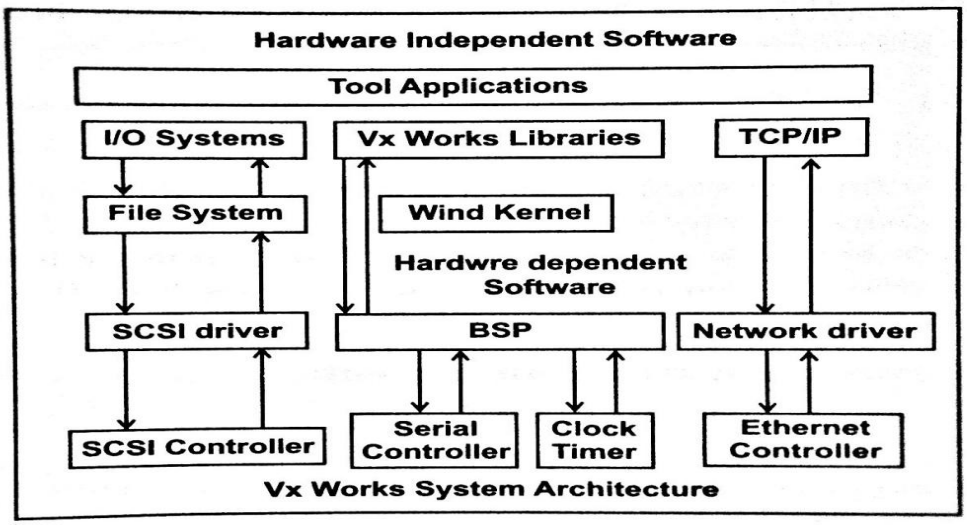


Fig. Vx Works System Architecture

Task Management:

The Vx works real-time kernels provides a basic multitasking environment. Vx works offers both posix and a proprietary scheduling mechanisms. Both preemptive priority and round robin scheduling mechanism are available. The difference between POSIX and wind scheduling is that wind scheduling is that wind scheduling mechanism are available.

The difference between a POSIX and wind scheduling is that wind scheduling applies the scheduling algorithm on a system wide basis, whereas POSIX scheduling algorithms are applied on a process by process basis.

In Vx works, the states encountered by the task are of 8 different types.

1. Suspended: Idle state just after creation or stated where execution is inhibited.
2. Ready: Waiting for running and CPU access in case scheduled by the scholar but not waiting for a message through IPC.
3. Pending: The task is blocked as it waits for a message from the IPC or from a resource only then will the CPO be able to process further.
4. Delayed: Send to sleep for a certain time interval.
5. Delayed + suspended: Delayed and then suspended if it is not pre-emptied during the delay period.
6. Pended for an IPC _ suspended: Pended and then suspended if the blocked state does not change.
7. Pended for an IPC + delayed: Pended and than pre-emptied after the delayed time interval.
8. Pended for an IPC + suspended: Pended and suspended after delayed time interval.

Kernel library functions are included in the header file 'Vx works.h' and 'kernel Lib.h'. Task and system library functions are included in 'task Lib.h' and 'sys Lib.h' User task priorities are between 101 and 255. Lowest priority means task of highest priority number (255). System tasks have the priorities from 0 to 99. For tasks, the highest priority is 100 by default..

The functions involved in task management:

1. ***Task Spawn function:*** It is used for creating and activating a task. Prototype is unsigned int task ID = task spawn (name, priority, options, stack size, main, argo, art1, arg2, ... arg9).

2. ***Task suspending and Resuming functions:*** Task suspend (task ID): inhibits the execution of task indentified by task Id.

Task Resume (task ID): Resumes the execution of the task identified by task ID.

Task Restart (task ID): First terminates a task and then spawn again with its original assignes arguments.

3. ***Task deletion and deletion protection function:*** Task Delete (task Id): this permanently inhibits the execution f the task identified by task Id and cancels the allocations of the memory block for the task stack and TCB.

Many time each task should itself execute the codes for the following:

- Memory de-allocation.
- Ensure that the waiting task gets the desired IPC
- Close a file, which was opened there.
- Delete child tasks when the parent task executes the exit ( ) function.

4. Delaying a task to let a lower priority task get access:

intSysClkRateGet ( ) returns the frequency of the system ticks. Therefore to delay by 0.25 seconds, the function task Delay (SysclkRateGet ( ) /4) is used.

Memory Management:

In Vx works, all systems and all application tasks share the same address space. This means that faulty application could accidently access system resources and compromise the stability of entire system. An optional tool named Vx VMI is available that can be used to allow each task to have its own address space. Default physical page size used is 8 KB. Virtual memory support is available with Vx VMI tool. Vx works does not offer privilege protection. The privilege level is always 0.

Interrupts:

To achieve the fastest possible response to external interrupts, Interrupt service routines in Vx works run in a special context outside of any thread's context, so that there are no thread context switches involved. The C function that the user attaches to a interrupt vector is not actual ISR. Interrupts cannot directly vector to C functions.

The ISR's address is stored in the interrupt vector table and is called directly from the hardware. The ISR performs some initial work and then calls the C function that was attached by the user. For this reason, we use the term interrupt handler to designate the user installed C handler function. Vx Works uses an ISR design that is different from a task design.

The features of the ISR in Vx works are:

1. ISRs have the highest priorities and can pre-empt any running task.
2. An ISR inhibits the execution of tasks till return.
3. An ISR does not execute like a task and does not have regular task context.
4. An ISR should not use mutex semaphore.
5. ISR should just write the required data at the memory or buffer.
6. ISR should not use floating – point functions as these take longer time to execute.

Performance:

- *Real time performance:* Capable of dealing with the most demanding time constraints, Vx works is a high-performance RTOS tuned for both determinism and responsiveness.
- *Reliability:* A high – reliability RTOS, Vx works provides certification evidence required by strict security standard. Even for non-safety – critical systems, Vx works is counted on to run forever, error free.
- *Scalability:* An indispensable RTOS foundation for very small – scale devices, large scale networking systems and everything in between, Vx works is the first RTOS to provide full 64-bit processing to support the over growing data requirements for embedded real time systems. Vx works is scalable in terms of memory foot print and functionality so that it can be tuned as per the requirements of the project.
- *Interrupt latencies:* The time elapsed between the execution of the last instruction of the interrupted thread and the first instruction in the interrupts handler to the next task scheduled to run is interrupt dispatch latency. Vx works exhibits an Interrupt latency of 1.4 to 2.6 micro seconds and dispatch latency of 1.6 to 2.4 ----s
- *Priority inheritance:* Vx works has a priority inheritance mechanism that exhibits an optimal performance, which is essential for an RTOS.
- *Foot print:* Vx works has a completely configurable and tunable small memory foot print for today's memory – constrained systems. The user can control how much of the operating system he needs.

Applications:

Vx Works RTOS is widely used in the market, for a great variety of applications. Its reliability makes it a popular choice for safety critical applications Vx works has been success fully used in both military and civilian avionics, including the Apache Attack Helicopter, Boeing 787, 747-8 and Airbus A 400 M. It is also used in on ground avionic systems such as in both civilian and military radar stations. Another safety critical application that entrusts Vx works is BMW's i – Drive system.

However, Vx works is also widely used in non-safety – critical applications where performance is at premium. The xerox phasor, a post – script printer is controlled by a Vx works powered platform link sys wireless routers use Vx works for operating switches.

Vx works has been used in several space application's. In space crafts, where design challenges are greatly increased by the need of extremely low power consumption and lack of access to regular maintenance, Vx works RTOs can be chosen as the operating

system for on Board Computer (OBC). For example 'clementine' launched in 1994 is running Vx works 5.1 on a MIPS – based CPU responsible for the star Tracker and image processing algorithms. The 'spirt' and opportunity mars exploration rovers were installed with Vx works. Vx works is also used as operating system in several industrial robots and distributed control systems.

Summary:

The need to develop for real-time embedded applications is always a challenge, especially when expensive hardware is at risk. The complex nature of such systems require many special design considerations an understanding of physical systems, and efficient Management of limited resources perhaps one of the most difficult choice the embedded system designers have to make in which operating system they are going ot use. It is critical to have operating system that will be able to be fail-safe, secure, scalable, fast and robust in multi task management, while being friendly to the application developers, Vx works is an RTOs which meets almost all of these requirements.

iii) RT Linux

RT linux is a hard real time RTOS microkernel that runs the entire linux operating system as a fully preemptive process. It was developed by Victor Yodaiken, Michael Barabanov and others at the New Mexico Institute of Mining and Technology and then as a commercial product at FSM Labs. FSM Labs has 2 editions of RT Linux.

- RT Linux pro and RT Linux free. RT Linux pro is the priced edition and RT Linux is the open source release. RT Linux support hard real time applications. The Linux kernel has been modified by adding a layer of software between the hardware and the Linux kernel. This additional layer is called 'Virtual Machine'. A footprint of 4 MB is required for RT Linux.

The new layer, RT Linux layer has a separate task Scheduler. This task scheduler assigns lowest priority to the standard to the standard Linux kernel. Any task that has to met real-time constraints will run under RT Linux. Interrupts from Linux are disabled to achieve real time performance.

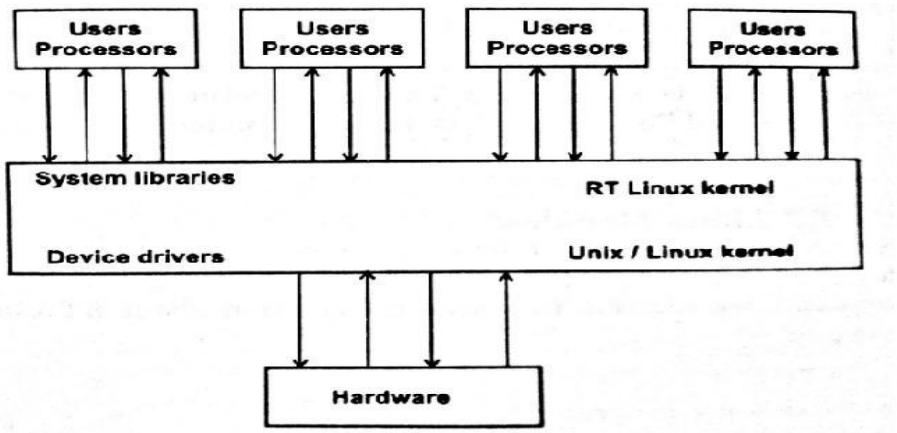


Fig. RT Linux kernel

In RT Linux location of the various files are:

- RT Linux will be installed in the directory/usr/rtlinux-XXX, where XXX is the version name.
- /usr/rtlinux/include contains all the include files necessary for development project.
- /usr/doc/rtlinux/man contains the manual pages for the RT Linux.
- /usr/rtlinux/modules contains the core RT Linux modules

The two important aspects while doing programming in RT linux are:

- By default, the RT Linux tasks do not have access to the computer's floating points unit. Hence need to explicitly set the permissions for every RT Linux task.
- It cannot pass arguments from the command prompt.

RT Linux Modules:

RT Linux programe are not created as stand-alone units, they are created as modules. Which are loaded into the linux kernel space. The C source files are complied into object files using the gcc command with the argument C flag. In the C file the main ( ) function gets replced with the following lines.

- Int init-module ( ).
- Void cleanup_module ( ).

Init-module is called when the module is first loaded into the kernel. This function returns O if the module is successfully loaded. It returns a negative value in case of failure. When the module is loaded is to be unloaded, the cleanup_module ( ) is called.

Executing the RT Linux Modules:

In RT Linux, Load and stop user modules are using the RT Linux command. Using this command, we can obtain status information about RT Linux modules. The command syntax is: