**CHAPTER 06:**

**RTOS AND INTER-PROCESS COMMUNICATION**

**Preview :**

6.1 Concepts of RTOS, Need of RTOS in Embedded systems

6.2 Multitasking

6.3 Task synchronization & Mutual Exclusion

6.4 Starvation, Deadlock, Multiple process

6.5 Basics of Inter-process Communication

In this chapter we will study what is real time operating system? Which is nothing but system whose response is predictable? Now it is useful in embedded system to handle multiple devices using RTOS. To handle multiple devices different algorithms like Round Robin, Round Robin with ISR, Function Queue Scheduling, RTOS can be used that is given in detail. And to achieve task synchronization with mutual exclusion semaphores can be used that will also be discussed in this chapter. Also

the concepts of deadlock, starvation will be discussed.

**6.1 Concepts of RTOS**

**6.1.1 General Purpose Operating System:**

An **operating system** (**OS**) is software that manages [computer hardware](http://en.wikipedia.org/wiki/Computer_hardware) and [software](http://en.wikipedia.org/wiki/Computer_software) resources and provides common [services](http://en.wikipedia.org/wiki/Operating_system_services) for [computer programs](http://en.wikipedia.org/wiki/Computer_program). The operating system is an essential component of the [system software](http://en.wikipedia.org/wiki/System_software) in a computer system. Application programs usually require an operating system to function.

[Time-sharing](http://en.wikipedia.org/wiki/Time-sharing) operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and [memory allocation](http://en.wikipedia.org/wiki/Dynamic_memory_allocation), the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently make a [system call](http://en.wikipedia.org/wiki/System_call) to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a computer—from [cellular phones](http://en.wikipedia.org/wiki/Cellular_phone) and [video game consoles](http://en.wikipedia.org/wiki/Video_game_console) to [supercomputers](http://en.wikipedia.org/wiki/Supercomputer) and [web servers](http://en.wikipedia.org/wiki/Web_server).

Examples of popular modern operating systems include [Android](http://en.wikipedia.org/wiki/Android_(operating_system)), [BSD](http://en.wikipedia.org/wiki/BSD), [iOS](http://en.wikipedia.org/wiki/IOS), [Linux](http://en.wikipedia.org/wiki/Linux), [OS X](http://en.wikipedia.org/wiki/OS_X), [QNX](http://en.wikipedia.org/wiki/QNX), [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows), [Windows Phone](http://en.wikipedia.org/wiki/Windows_Phone), and [IBM z/OS](http://en.wikipedia.org/wiki/IBM_z/OS). All these examples, except Windows, Windows Phone and z/OS, share roots in [UNIX](http://en.wikipedia.org/wiki/UNIX).

**A GPOS is** used for systems/applications that are not time critical.

**6.1.2 Real time operating system:-**

Real time can also refer to events simulated by a computer at the same speed that they would occur in real life. In [graphics](http://www.webopedia.com/TERM/G/graphics.html) [animation](http://www.webopedia.com/TERM/A/animation.html), for example, a real-time [program](http://www.webopedia.com/TERM/P/program.html) would display [objects](http://www.webopedia.com/TERM/O/object.html) moving across the [screen](http://www.webopedia.com/TERM/D/display_screen.html)at the same speed that they would actually move.

In real time application completion of task includes completing it in more than one dimension. In RTOS response is controllable and predictable.

 We can also say an **RTOS** is supposed to give quick and predictable response.

An RTOS is used for time critical systems.Example:- VxWorks, uCos  etc.

**6.1.2.1 Classification of RTOS**

**RTOS can be classified into three types :**

1. Hard RTOS: These type of RTOS strictly adhere to the deadline associated with the tasks. Missing on a deadline can have catastrophic affects.
2. Firm RTOS: These types of RTOS are also required to adhere to the deadlines because missing a deadline may not cause a catastrophic affect but could cause undesired affects, like a huge reduction in quality of a product which is highly undesired.
3. Soft RTOS:  In these type of RTOS, missing a deadline is acceptable. For example On-line Databases.

**6.1.3 Similarities between RTOS and normal OS are**

* Multi-tasking
* Management of software and hardware resources
* Providing services to applications

**Why RTOS is better than normal OS**

The basic difference of using a GPOS or an RTOS lies in the nature of the system – i.e whether the system is “time critical” or not! They provide deterministic time response.

A system can be of a single purpose or multiple purpose that is real time OS can be customized.

Example of a “time critical system” is – Automated Teller Machines (ATM). Here an ATM card user is supposed to get his money from the teller machine within 4 or 5 seconds from the moment he press the confirmation button.

**Latency issues:** Another major issue with a GPOS is unbounded dispatch latency, which most GPOS falls into. The more number of threads to schedule, latencies will get added up! An RTOS has no such issues because all the process and threads in it has got bounded latencies – which means – a process/thread will get executed within a specified time limit.

**6.1.4 Need and Specification of RTOS**

**Need or Requirements**

1. Some applications require system to respond to an event within specific amount of time(deadline).
2. System need to do multitasking in a manner in which all the concurrent tasks will be completed within correct time.
3. A scalable platform is needed using which the application development is done faster
4. There is need to control collection of devices connected to each other that provides the embedded system. This controlling can be done using real time operating system.

**6.1.5 Key Specifications or Features of RTOS:**

**1. Reliability**

* Reliable and time bound inter process mechanisms should be in place for processes to communicate with each other in a timely manner.
* Reliable system is the one which is available all the time. RTOS is reliable but it does not guarantee the reliability of the embedded system.

**2. Predictability**

* Context switching latency should be short. This means that the time taken while saving the context of current task and then switching over to another task should be short or predictable.
* The time taken between executing the last instruction of an interrupted task and executing the first instruction of interrupt handler should be predictable and short. This is also known as interrupt latency.
* Similarly the time taken between executing the last instruction of the interrupt handler and executing the next task should also be short and predictable. This is also known as interrupt dispatch latency.

**3. Performance**

* RTOS should be fast enough to fulfill the timing requirements. RTOS should have a deterministic behavior in terms of deadlines but its not true that the processing speed of an RTOS is fast. This ability of responsiveness of an RTOS does not mean that they are fast.
* All RTOS are same. As already discussed we have three types of RTOS (Hard, firm and soft).

**4. Compactness:-**

* RTOS mostly used in embedded systems that are extremely constraints as far as resource are concerned.
* As the name suggests, there is a deadline associated with tasks and an RTOS adheres to this deadline as missing a deadline can cause affects ranging from undesired to catastrophic.
* The example cell phone must be small, portable and low cost. These design requirements limit the system memory, which limits the size of application and operating system.

.**5. Scalability:**

* As used RTOS can be used in wide variety of embedded system, they must be able scale up or scale down according to application.
* RTOS should be capable of adding or deleting the modular components depending on how much functionality is needed.

**6.2 Multitasking**

* Embedded systems are generally application specific but need to perform several tasks concurrently for the same application.
* A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources.
* The inter-task communication facilities allow these tasks to synchronize and coordinate their activity.
* Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications.

To provide multitasking following activities need to provide.

* 1. Process Management
  2. Interprocess Communication and Synchronization
  3. Memory Management
  4. Input/output Management

**Consider example of washing machine which consist of following tasks**

Operation of wash motor

Timers

Water level Sencing

Keypad Scanning

The system has to perform more than one task at a time.

**6.2.1 Ways of achieving Scheduling**

**1. Round Robin**

In this scheduling, ready tasks are in a circular queue.

**Algorithm:**

void main()

{

while(1)

{

if(device 1 requires service)

Perform device 1 IO functions;

if(device 2 requires service)

Perform device 2 IO functions;

if(device 3 requires service)

Perform device 3 IO functions;

.

.

.

if(device N requires service)

Perform device N IO functions;

}

}

* It closely relates to function queue scheduling.
* Characterized by the absence of interrupts
* Consists of a main loop that checks each I/O device in turn and services them if needed.
* Cannot suffer from shared data problems
* Latency is limited by the maximum duration of a loop cycle.
* Attractive for simple environments

**Round-Robin Problems**

* If any device needs a response in less time than the duration of the loop the system won't function that is one of worst
* If A and B take 5ms each and Z needs a response time of less than 7ms its not possible.
* This can be mitigate somewhat by doing (A,Z,B,Z) in a loop instead of (A,B,Z).
* Scalability of this solution is poor.
* Even if absolute deadlines do not exist, overall response time may become unacceptably poor
* For example if in worst case scenario the processor runs through the loop with 5 devices to service in 100 ms and the deadline for each device servicing is 90 ms, this is also one of worst case.

**2. Round-Robin with Interrupts**

* Interrupt routines deal with the very urgent needs of devices
* Interrupt routines set flags to indicate the interrupt happened
* Main while loop polls the status of the interrupt flags and does any follow-up processing required by a set flag.

**Algorithm**

BOOL flag\_device1=FALSE;

BOOL flag\_device2=FALSE;

.

.

BOOL flag\_deviceN=FALSE;

void interruptISR\_device1(void)

{

Critical code for servicing of device 1;

flag\_device1=TRUE;

}

void interruptISR\_device2(void)

{

Critical code for servicing of device 2;

flag\_device2=TRUE;

}

.

.

.

void interruptISR\_deviceN(void)

{

Critical code for servicing of device 1;

flag\_deviceN=TRUE;

}

void main()

{

while(1)

{

if(flag\_device 1)

{

flag\_device1=FALSE;

Perform non critical code for servicing device 1 ;

}

if(flag\_device 2)

{

flag\_device2=FALSE;

Perform non critical code for servicing device 2;

}

.

.

if(flag\_device N)

{

Flag\_deviceN=FALSE;

Perform non critical code for servicing device N ;

}

}

}

**RR with Interrupts Advantages**

● More control over priorities. – device routines can be serviced in any order – processor interrupt priority settings can be used

● Interrupt routines get good response (low latency) – Main loop can be suspended – Interrupt routines are (must/should be) short. – Interrupt code inherently has a higher priority than task code

**RR with Interrupts disadvantages**

● More complicated than Round-Robin

● Context problems can occur – saving and restoring context inside interrupts routines becomes necessary when number of registers/resources is small.

● Shared data problems – Debugging becomes more complicated.

● Same latency/priority issues causing (A,C,B,C) still exist**.**

**3. Function-Queue Scheduling**

● Interrupt routines enqueue function pointers for follow-up work onto a queue.

● main routine just dequeues a pointer from the queue and makes a branch to that address.

void interruptISR\_device1(void)

{

Critical code for servicing of device 1;

Put function1 on queue of function pointers;

}

void interruptISR\_device2(void)

{

Critical code for servicing of device 2;

Put function2 on queue of function pointers;

}

.

.

.

void interruptISR\_deviceN(void)

{

Critical code for servicing of device N;

Put functionN on queue of function pointers;

}

void function1(void)

{

Perform non critical code for servicing device 1 ;

}

void function2(void)

{

Perform non critical code for servicing device 2 ;

}

.

.

. void functionN(void)

{

Perform non critical code for servicing device N ;

}

void main(void)

{

while(1)

{

while(function queue is empty);

call the first function in the queue and update the queue;

}

}

**Advantages**

* Latency for high priority devices can be reduced compared to Round-Robin with Interrupts (C,C,C,C,A,C,C,B) is simply a matter of the queuing algorithm.
* In Round-Robin with Interrupts every loop may end up executing every follow-up task.
* Function-Queue Scheduling guarantees that at most a single follow-up task is executed per loop iteration**.**

**Disadvantages**

● Latency for low priority tasks can increase.

● Low priority tasks can actually starve

● Queuing algorithm may be complex/costly to run/code.

● If a low priority follow-up task is very time consuming the latency for higher priority response times will suffer.

– Once you start processing C you must wait for completion before any other task will get attention.

**4. Real-Time Operating Systems**

* Interrupts signal the need for follow-up tasks. – But, unlike Function-Queue Scheduling, this is handled by the Real-Time Operating System and not by the interrupt routines manipulating flags or a queue.
* Instead of a loop deciding what to do next the RTOS decides.
* One follow-up task can be suspended by the RTOS in favoring of performing a higher priority task.
* Biggest difference compares to Function-Queue Sched.

**Algorithm:**

void interruptISR\_device1(void)

{

Critical code for servicing of device 1;

Set signal 1;

}

void interruptISR\_device2(void)

{

Critical code for servicing of device 2;

Set signal 2;

}

.

.

.

void interruptISR\_deviceN(void)

{

Critical code for servicing of device N;

Set signal N;

}

void task1(void)

{

Wait for signal1;

Perform non critical code for servicing device 1 ;

}

void task1(void)

{

Wait for signal 2;

Perform non critical code for servicing device 2 ;

}

.

.

. void task1(void)

{

Wait for signalN;

Perform non critical code for servicing device N;

}

**RTOS Advantages**

● Suspension of tasks allows the worst case wait for the highest priority item to be zero.

● built-in scheduling mechanism yields a system with very stable response characteristics even when changes to the code occur.

● Widely available for purchase.

**RTOS Disadvantages**

● Can be Costly.

● RTOS are generally complicated and can consume a non-trivial amount of processor cycles.

**6.3 Task synchronization & Mutual Exclusion**

**6.3.1Task**

* TASK: Task is the term used for the processes in the RTOS for the embedded systems.
* Task can be defined as that executing unit of computation, which is controlled by some process at the OS scheduling mechanism.
* This scheduling mechanism allows task to execute on the CPU and OS for a resource-management mechanism so that it can use the system memory and other system-resources such as network, file, display or printer.

**6.3.2 States of a Task in a system**

1. Idle state [Not attached or not registered]
2. Ready State [Attached or registered]
3. Running state
4. Blocked (waiting) state
5. Terminated

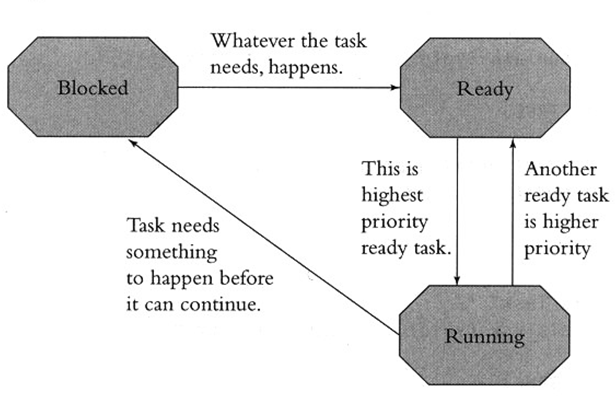


Fig6.1: Task states

The tasks states:

In embedded system there are number of tasks that are to execute concurrently. Each task in RTOS can be in one of the state given above.

**Idle state** is task is doing nothing just created task can be in this state that is not attached to anything.

**Running state** of task means that microprocessor/microcontroller is executing the instructions that completes task. At a time one and only one task can be in running task.

**Ready state** of a task means other task is in running state, but this task has all resources it needs to go to running state. Any number of tasks can be in this state.

**Blocked state** of a task means that task will not be able to execute currently even if the microprocessor/microcontroller becomes available. The task can be in this state because they do not have one or more resources or inputs that are needed for the task execution. Any number of tasks can be in this state.

**Scheduler :**

It is part of RTOS that keeps the track of state of each task and decides which one task should go into the running state.

Scheduler in RTOS is usually look at the priorities assigned to the tasks (by the programmer), among the tasks that are not in blocked state, the one with the highest priority is allowed to run while the other ready task wait.

There can be situations in task execution-

* + Scheduler/Task signal exchange for block-unblock of tasks via function calls
  + All tasks are blocked and scheduler idles forever (not desirable!)
  + Two or more tasks with same priority levels in Ready state (time-slice, FIFO)

Each task has its own private context which includes the register values, a program counter and a stack. All other data is shared amongstall the tasks in the system.

**6.3.3 Methods of task synchronization are**

* + 1. Semaphore: counting semaphore and binary semaphore
    2. mutex (short for mutual exclusion)
    3. Spinlock.
    4. Read/write locks.
    5. Barrier

**1. Semaphore: counting semaphore and binary semaphore**

We use semaphore for achieving task synchronization with mutual exclusion.

• A semaphore is created with initial\_count, which is the number of allowed holders of the semaphore lock. (initial\_count=1: binary sem)

When no process is being executed its value is 0.

Sem\_wait will decrease the count; while Sem\_signal will increase it.

That is a process which want to execute will acquire semaphore and execute Sem\_wait and decrements count. When it completes its execution will execute Sem\_signal and increments count. So now count becomes 0.

A task can get the semaphore when the count > 0; otherwise, block on it.

In general we can say: Semaphores are data structures with a count and two associated operations, give and take.

A '**give**' (signal) operation increments the count and returns immediately.

A '**take**' operation decrements the count and returns immediately, unless the count is already zero. In this case the operation blocks until another process gives the semaphore.

• Semaphores have the semantics to determine which of the possibly many blocked processes attempting to take a semaphore will awaken first when it is given, for example first-in first-out (FIFO) or priority-based (PRIO).

• There are two types of semaphores, optimized to address different classes of problems:

**Binary:-**

• A binary semaphore is a semaphore with a maximum count of 1. Binary semaphores are useful to enforce mutual exclusion: only one process can have the semaphore at any point of time, and then other takers will block until the holder returns it. Shared data will remain consistent, since there is no possibility of being interrupted during access.

When Sem\_wait( i.e any one task is in execution) value is 0 and when sem\_signal is 1.

**Counting:-**

• A counting semaphore is similar to the binary semaphore, it but keeps track of the number of times the semaphore is given. Optimized for guarding multiple instances of a resource.

**2. Mutex:-**

Similar to a binary semaphore, but mutex has an owner.

A mutual exclusion (mutex) semaphore is a special binary semaphore that supports ownership, recursive access, task deletion safety, and one or more protocols for avoiding problems which can occur in mutual exclusion.

As given the available and unavailable states in binary and counting semaphores, the states of a mutex are unlocked or locked (0 or 1, respectively).

A mutex is initially created in the unlocked state, in which it can be acquired by a task. After being acquired, the mutex moves to the locked state.

do {

wait(mutex);

/\* critical section \*/

signal(mutex);

/\* remainder section \*/

} while (true);

Conversely, when the task releases the mutex, the mutex returns to the unlocked state. Note that some kernels might use the terms lock and unlock for a mutex instead of acquire and release.

Depending on the implementation, a mutex can support additional features not found in binary or counting semaphores. These key differentiating features include ownership, recursive locking, task deletion safety, and priority inversion avoidance protocols.Figure illustrates the state diagram of a mutex.



Fig 6.2: State diagram of a process

**3. Spinlock: lock mechanism for multi-processor systems,**

* A task wanting to get spinlock has to get a lock shared by all processors.
* The operating systems provide different locking mechanisms depending on the application developers’ needs.
* Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue.
* Mutexes are useful for locking resources.
* Solaris 2 uses adaptive mutexes, meaning that the mutex is implemented with a spin lock on multiprocessor machines.

**4.** **Read/write locks: protect from concurrent write, while allow concurrent read** .

Many tasks can get a read lock; but only one task can get a write lock. Before a task gets the write lock, all read locks have to be released.

**5.** **Barrier: to synchronize a lot of tasks,** they should wait until all of them have reached a certain barrier. A **barrier** is a type of [synchronization](http://en.wikipedia.org/wiki/Synchronization_(computer_science)) method.

A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Many collective routines and directive-based parallel languages impose implicit barriers.

Use of many [Task](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx)objects and a [Barrier](http://msdn.microsoft.com/en-us/library/system.threading.barrier(v=vs.110).aspx)to synchronize them.

The problem pops out when many tasks use a Barrier to synchronize at a certain point in the elaboration. Let’s say a function F can be parallelized into two blocks of parallel code A and B, with a synchronization point between A and B, i.e., after a local computation each task needs the partial results from the other tasks to complete its computation. The single-threaded version can be sketched as:

sequential\_F() {

sequential\_A();

//sync is implicit

sequential\_B();

}

The parallel version:

parallel\_F() {

parallel\_A();

wait\_for\_sync();

parallel\_B();

}

There are at least two ways to implement parallel\_F in C#: (1) writing a single task with a Barrier, or (2) writing two tasks with an external synchronization point.

**6.4 Starvation and Deadlock**

**6.4.1 Starvation:-**

CPU starvation occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.

The possibility of blocked states is extremely important in real-time systems because without blocked states, lower priority tasks could not run. If higher priority tasks are not designed to block, CPU starvation can result.

**6.4.2 Deadlock:-**

* Deadlock is the situation in which multiple concurrent threads of execution are blocked permanently because of resource requirements that can never be satisfied.
* A typical real-time system has multiple types of resources and multiple concurrent threads of execution contending for these resources. Each thread of execution can acquire multiple resources of various types throughout its lifetime.
* Deadlock exist in a system in which the underlying RTOS permits resource sharing among multiple threads of execution.

Following code shows situation of deadlock

char X;

char Y;

RTOS semaphoreX;

RTOS semaphoreY;

void task1(void)

{

Takesemaphore(semaphoreX);

Takesemaphore(semaphoreY);

x=y;

releasesemaphore(semaphoreX);

releasesemaphore(semaphoreY);

}

void task2(void)

{

Takesemaphore(semaphoreY);

Takesemaphore(semaphoreX);

y=x;

releasesemaphore(semaphoreY);

releasesemaphore(semaphoreX);

}

* Consider what happens if task1 calls a function to take semaphore and gets it, but before it can call the function to take semaphoreY, the RTOS stops the execution of task1 and runs the task2.
* The task 2 calls function to take semaphoreY and gets semaphoreY. When the task2 calls the function to take semaphore, it is blocked because task1 has already taken the semaphore.
* The RTOS will now switch back to task1, which now calls the function to take semaphoreY, but task2 has semaphoreY, so the task1 is now blocked.

**6.5 Basics of Inter-process Communication**

* Interprocess communication is set of techniques for exchange of data among multiple threads in one or more process.
* 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

**6.5.1 The interprocess communication can be explained with any of the following methods.**

1. MessageQueue

2. Pipeline

3. Socket device

4. Remote procedure call (RPC) for distributed processes.

5. Semaphore

6. Signal

**1. Message Queue:**

* Message queue provides an asynchronous communication protocol, that is sender and receiver need not be present at the time of communication or need not to interact with the message queue at the same time.
* Messages placed onto the queue are stored until receipient retrieves them.
* Most message queue has set limits on size of message to be transmitted in a single message. Those queues that do not have limits are called as mailboxes.
* Many RTOSes such as VxWorks use Message queue as primary IPC.

**2. Pipeline**

* Pipeline is set of processes chained by their standard streams, so that the output of each buffering a sending stream may produce 5000 bytes per second, and a receiving program may only receive 100 bytes per second. No data is lost

Fig shows pipeline of three programs

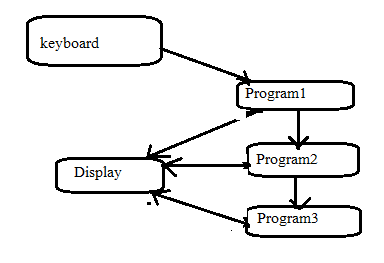


Fig 6.3 pipeline of three programs run on a text terminal

* When receiving program is ready to read data the OS send this data from buffer, then removes that data from buffer.
* If the buffer(pipeline) fills up the sending program is suspended until the receiving program has a chance to read some data and make room in buffer.

**3. Remote Procedure Call**

* RPC is an interprocess communication technology that allows a program to call a subroutine or procedure to execute in another address space without the programmer explicitly coding the details for the system.
* That is programmer use same code wheather the subroutine is local to the executing program or remote.

**4. Semaphore**

* A semaphore is protected variable which include a method for restricting access to resources such as shared memory in multiprogramming environment.
* A counting semaphore is a counter for a set of available resources, rather than a locked/unlocked flag of single resources.

**5. Signal**

* Signal is limited form of IPC used in UNIX
* It is asynchronous notification sent to a process in order to notify it of event has occured
* When a signal is sent to a process the operating system interrupts the process normal flow of execution.

**QUESTION BANK**

**2M Questions**

1. List the different hardware units present in an embedded system?
2. Give the advantages of an embedded system?
3. What is compiler?
4. What is debugger? Why it is required?
5. What is the difference between emulator & simulator?
6. What is the difference between RTOS & desktop OS?
7. What is multitasking?
8. Which are the different scheduling algorithms used in RTOS?

**4M Questions**

1. What do you meant by task Synchronization and mutual exclusion? [SUMMER 12]
2. Enlist the scheduling algorithm and explain any one. [winter 12]
3. Describe the specifications of RTOS. [Winter 12]
4. Define the term starvation and deadlock with respect to multiple

process.[Summer 12]

1. What do you meant by virtual device driver? [SUMMER 13]
2. Describe various operating system functionality in RTOS. [Summer 12]
3. Define RTOS. Why RTOS is needed?[ Winter 12]