International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622
International Conference on Industrial Automation and Computing (ICIAC- 12-13th April 2014)

RESEARCH ARTICLE OPEN ACCESS

Priority Based Assignment of Shared resources in RTOS

Ms. Raana Syeda*, Ms. Manju Ahuja**, Ms. Sneha Khatwani***
Mrs. Swara Pampatwar****

*(Department of Computer Science & Engg., Jhulelal Institute of Technology, Nagpur
Email: raana.syeda@gmail.com)
** (Department of Computer Science & Engg., Jhulelal Institute of Technology, Nagpur
Email: manju.ahuja@yahoo.com)
*** (Department of Computer Science & Engg., Jhulelal Institute of Technology, Nagpur
Email: sneha_khatwani@rediffmail.com)
**** (Department of Computer Science & Engg., Jhulelal Institute of Technology, Nagpur
Email: padmajaadgulwar@gmail.com)

ABSTRACT
A real time operating system (RTOS) is a system that allows completing the task in predictable timing constraints. RTOS have several characteristics such as multitasking, preemption, priority, predictable task synchronization, priority inheritance and known behavior. The major problem of RTOS is critical section problem. In RTOS, it is difficult make available resources to all processes in deterministic and predefined time constraint (deadline) according to their priorities. This paper deals with the study of priority based scheduling in RTOS with available semaphore based solutions of critical section problem. It also has two new semaphore based approaches for task synchronization in RTOS.

I. INTRODUCTION:

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests. It supports applications that must meet deadlines in addition to providing logically correct results. The main property of a real-time system is feasibility. It is the guarantees that task always meet their deadlines, when scheduled according to the chosen policy.

The design of an RTOS is essentially a balance between providing a reasonably rich feature set for application development and deployment and, not sacrificing predictability and timeliness. A basic RTOS will be equipped with the following features:

i. Multitasking and Preemptibility

An RTOS must be multi-tasked and preemptible to support multiple tasks in real-time applications. The scheduler should be able to preempt any task in the system and allocate the resource to the task that needs it most even at peak load.

ii. Task Priority

Preemption defines the capability to identify the task that needs a resource the most and allocates it the control to obtain the resource. In RTOS, such capability is achieved by assigning individual task with the appropriate priority level. Thus, it is important for RTOS to be equipped with this feature.

iii. Reliable and Sufficient Inter Task Communication Mechanism

For multiple tasks to communicate in a timely manner and to ensure data integrity among each other, reliable and sufficient inter-task communication and synchronization mechanisms are required.

iv. Priority Inheritance

To allow applications with stringent priority requirements to be implemented, RTOS must have a sufficient number of priority levels when using priority scheduling.

v. Predefined Short Latencies

An RTOS needs to have accurately defined short timing of its system calls. The behavior metrics are:

• Task switching latency: The time needed to save the context of a currently executing task and switching to another task is desirable to be short.

• Interrupt latency: The time elapsed between execution of the last instruction of the interrupted task and the first instruction in the interrupt handler.

• Interrupt dispatch latency. The time from the last instruction in the interrupt handler to the next task scheduled to run.

vi. Control of Memory Management

To ensure predictable response to an interrupt, an RTOS should provide way for task to lock its code and data into real memory.

II. TASK SCHEDULING

A real-time OS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time. The purpose of a real-time scheduling algorithm is to ensure that critical timing constraints, such as deadlines and response time, are met. When necessary, decisions are made that favor the most critical timing constraints, even at the cost of violating others. Real-time scheduling is also used to allocate processor time between tasks in soft real-time embedded systems.

Most RTOSs today control the execution of application software tasks by using priority-based pre-emptive scheduling. In this approach, software developers assign a numeric "priority" value to each task in their application software. The RTOS's task scheduler will allow tasks to run, and will switch among the tasks using the rule that "The highest priority task that is ready to run, should always be the task that is actually running.

2.1 Priority-based scheduling

Many real-time systems use preemptive multitasking, especially those with an underlying real-time operating system (RTOS). Relative priorities are assigned to tasks, and the RTOS always executes the ready task with highest priority.

The scheduling algorithm is the method in which priorities are assigned. Most algorithms are classified as static priority, dynamic priority, or mixed priority. A static-priority algorithm assigns all priorities at design time, and those priorities remain constant for the lifetime of the task. A dynamic-priority algorithm assigns priorities at runtime, based on execution parameters of tasks, such as upcoming deadlines. A mixed-priority algorithm has both static and dynamic components. Needless to say, static-priority algorithms tend to be simpler than algorithms that must compute priorities on the fly.

To demonstrate the importance of a scheduling algorithm, consider a system with only two tasks, Task 1 and Task 2. Assume these are both periodic tasks with periods T1 and T2, and each has a deadline that is the beginning of its next cycle. Task 1 has T1 = 50 ms, and a worst-case execution time of C1 = 25 ms. Task 2 has T2 = 100 ms and C2 = 40 ms. Note that the utilization, Ui, of task i is Ci/Ti. Thus U1 = 50% and U2 = 40%. This means total requested utilization U = U1 + U2 = 90%. It seems logical that if utilization is less than 100%, there should be enough available CPU time to execute both tasks.

Let's consider a static priority scheduling algorithm. With two tasks, there are only two possibilities:

Case 1: Priority(t1) > Priority(t2)
Case 2: Priority(t1) < Priority(t2)

The two cases are shown in Figure 1. In Case 1, both tasks meet their respective deadlines. In Case 2, however, Task 1 misses a deadline, despite 10% idle time. This illustrates the importance of priority assignment.

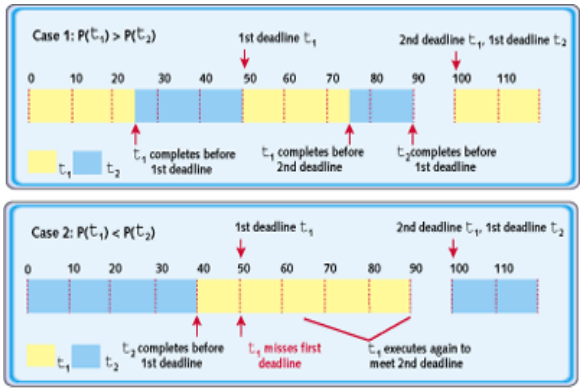

Figure 1: Both possible outcomes for static-priority scheduling with two tasks (T1=50, C1=25, T2=100, C2=40)

2.2 Setting priorities

2.2.1 Rate monotonic algorithm (RMA)

The rate monotonic algorithm (RMA) is a procedure for assigning fixed priorities to tasks to maximize their "schedulability." A task set is considered schedulable if all tasks meet all deadlines all the time. The algorithm is simple:

Assign the priority of each task according to its period, so that the shorter the period the higher the priority. In the example, the period of Task 1 is shorter than the period of Task 2. Following RMA's rule, we assign the higher priority to Task 1. This corresponds to Case 1 in Figure 1, which is the

priority assignment that succeeded in meeting all deadlines.

RMA is the optimal static-priority algorithm. If a task set cannot be scheduled using the RMA algorithm, it cannot be scheduled using any static-priority algorithm.

One major limitation of fixed-priority scheduling is that it is not always possible to fully utilize the CPU. Even though RMA is the optimal fixed-priority scheme, it has a worst-case schedulable bound of:

Wn = n * (21/n - 1)

Where n is the number of tasks in a system. As you would expect, the worst-case schedulable bound for one task is 100%. But, as the number of tasks increases, the schedulable bound decreases, eventually approaching its limit of about 69.3% (ln 2, to be precise).

It is theoretically possible for a set of tasks to require just 70% CPU utilization in sum and still not meet all their deadlines. For example, consider the case shown in Figure 2. The only change is that both the period and execution time of Task 2 have been lowered. Based on RMA, Task 1 is assigned higher priority. Despite only 90% utilization, Task 2 misses its first deadline. Reversing priorities would not have improved the situation.

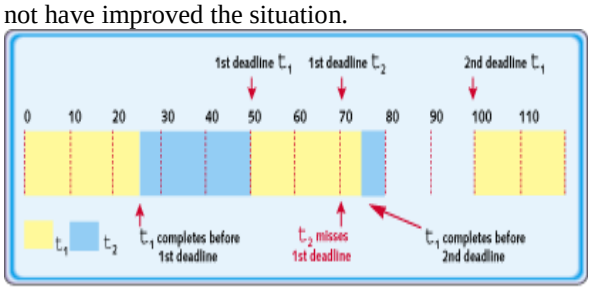

Figure 2: Some task sets aren't schedulable (T1=50, C1=25, T2=75, C2=30)

In this case, the only way to meet all deadlines is to use a dynamic scheduling algorithm, which, because it increases system complexity, is not available in many commercial RTOSes.

Sometimes a particular set of tasks will have total utilization above the worst-case schedulable bound and still be schedulable with fixed priorities. Figure 1's Case 1 is a perfect example. Schedulability then depends on the specifics of the periods and execution times of each task in the set, which must be analyzed by hand. Only if the total utilization is less than Wn can you skip that step and assume that all the tasks will meet all their deadlines.

RMA Guidelines

To benefit most from using a fixed-priority preemptive RTOS, consider the following rules of thumb:

- Always assign priorities according to RMA. Manually assigning fixed priorities will not give you a better solution.
- If total utilization is less than or equal to Wn, all tasks will meet all deadlines, so no additional work needs to be done.
- If total utilization is greater than Wn, an analysis of the specific task set is needed, to verify whether or not it will be schedulable.
- To achieve 100% utilization when using fixed priorities, assign periods so that all tasks are harmonic. This means that for each task, its period is an exact multiple of every other task that has a shorter period.

III. SYNCHRONIZATION

Because tasks share resources, events outside the scheduler's control can prevent the highest priority ready task from running when it should. If this happens, a critical deadline could be missed, causing the system to fail. Priority inversion is the term for a scenario in which the highest-priority ready task fails to run when it should.

3.1 Resource sharing

Tasks need to share resources to communicate and process data. This aspect of multi-threaded programming is not specific to real-time or embedded systems.

Any time two tasks share a resource, such as a memory buffer, in a system that employs a priority-based scheduler, one of them will usually have a higher priority. The higher-priority task expects to be run as soon as it is ready. However, if the lower-priority task is using their shared resource when the higher-priority task becomes ready to run, the higher-priority task must wait for the lower-priority task to finish with it. The higher-priority task is "pending" on the resource. If the higher-priority task has a critical deadline that it must meet, the worst-case "lockout time" for all of its shared resources must be calculated and taken into account in the design. If the cumulative lockout times are too long, the resource-sharing scheme must be redesigned.

Since worst-case delays resulting from the sharing of resources can be calculated at design time, the only way they can affect the performance of the system is if no one properly accounts for them.

3.2 Critical Section

One of the methods of controlling access to a shared resource is that we can declare a section of code to be critical and we then regulate access to that section. This section of code is called as critical section. The system must ensure that only one task can be in its critical section, no other task is allowed

to enter in its critical section. Thus the execution of critical section by the tasks is mutually exclusive.

Semaphore is the simplest and widely used synchronization tool, available in operating system or sometimes built into a programming language as a language construct, used for mutual exclusion. Semaphore is also used as a signaling tool. A task can use the resource (or enter into its critical section) if and only if the resource is free.

Semaphore has following advantages as compared to other synchronization tools:

- Simple to implement and use
- It is the most primitive and widely used construct for synchronization and communication in all operating systems
- It can be used to implement other synchronization tools Monitors, protected data type, bounded buffers, mailbox etc.

Apart from the advantages semaphores also have following disadvantages:

- May leads to deadlocks and starvation
- Loss of mutual exclusion if carefully not used
- Blocking tasks with higher priorities
- Priority inversion

The following are the challenges for task synchronization in RTOS:

- Critical section (data, service, code) protected by lock mechanism e.g. Semaphore etc. In a RTOS, the maximum time a task can be delayed because of locks held by other tasks should be less than its timing constraints.
- Race condition may leads to deadlock, livelock and starvation. Some deadlock avoidance/prevention algorithms are too complicate and non-deterministic for real-time execution. Simplicity is preferred, like
  - All tasks always take locks in the same order.
  - Allow each task to hold only one resource.
- Priority inversion using priority-based task scheduling and locking primitives should know the "priority inversion" danger: a medium-priority job runs while a high priority task is ready to proceed.

IV. VARIOUS IMPLEMENTATIONS OF SEMAPHORES

Following two semaphore implementations are widely used for task synchronization:

• Busy-Wait Implementation – In this implementation, the wait(S) operation checks for the availability of the resource. If the resource is free then it locks the resource and allowed to enter the task in critical section. If the resource is busy then it waits till the resource becomes free. After completing the operation in critical section, the task will release the resource by making a call to the signal(S) operation.

• Queuing Implementation – In this implementation, the wait(S) operation checks for the availability of the resource. If the resource is free then it locks the resource and allowed to enter the task in critical section. If the resource is busy then it adds the task into the waiting queue at rear of queue. After completing the operation in critical section, the task will first check the waiting queue if it is not free then handover the resource to the first task otherwise release the resource by making a call to the signal(S) operation. The queue can be implemented as simple queue where the task ids can be stored.

V. SEMAPHORE IMPLEMENTATION WITH PRIORITIES

The busy-wait implementation and queuing implementations are not suitable for the real-time systems due non-deterministic nature and not support for priorities. Therefore, an implementation is required which will be deterministic in nature and can handle the priorities. For working in the real-time system, a new implementation is proposed based on the queuing implementation of semaphore with priorities. There are two variants for the implementation is discussed for proposed implementation of semaphore:

• Priority Queuing Semaphore Implementation
•Multiple Priorities Queuing Semaphore Implementation

A. Priority Queuing Semaphore Implementation

In this approach, a single priority queue is used in which the task ids (or TCBs) are ordered in descending order based on their priority value. If two tasks of same priority are there then these will be stored in FIFO order. In other words, if a task waits for a resource which is used by some other executing task then it will be added into the priority queue at the appropriate position according to its priority value. The implementation details are described below:

Data structure:

The dynamic priority queue is used which will be implemented by using linked list so that there will not be any limitation for the waiting tasks. The number of waiting tasks will be restricted based on the available/allotted memory space. The structure of the node of linked list will be as follows (figure 3):

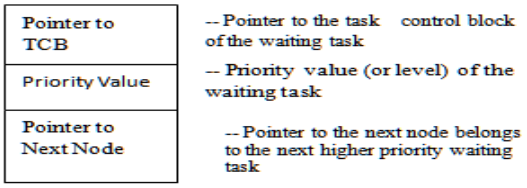

Figure 3: Data Structure for priority queuing semaphore implementation

Psuedocode and logic description of wait(S) operation:

```
wait(S)
{
        if(S == 0)
        then
                insert(PQ(S), TCB, priority);
        else
                S = S-1;
}

Insert(PQ(S), TCB, priority)
{
        Disable task switching
        Newnode->TCB;
        Newnode->Priority = priority;
        start = S;
        while (Start->Next->Priority >= priority)
                Start = Start->Next;
        Newnode->Next = Start->Next;
        Start->Next = Newnode;
        Enable task switching
}
```

The wait(S) operation first check whether the resource is free by checking the value of S if it is free then it will decrease the value of S and allow the task to enter in the critical section. If the resource is not free then the TCB of the requesting task will be inserted in the priority queue along with its priority with the help of another function insert(). The insert() function first prepare a node for the new waiting task, then search for appropriate position in the priority queue to add the node of the waiting task. The insert() function must be re-entrant. Therefore, the task switching is disabled at the beginning of this function and enabled at the end of the function.

Psuedocode and logic description of signal(S) operation:

```
Signal(s)
{
        Tasknode=delete(PQ(S),TCB)
        if Tasknode!= NULL
        then
                allocate the resource to the
Tasknode;
        else
                S = S+1;
}

Delete(PQ(S), TCB)
{
        Disable task switching
        node = S;
        S = S->Next;
        TCB = node->TCB;
        free(node);
        Enable task switching
}
```

The signal(S) operation first checks whether any other task waits for the resource which is going to free by the current task. if any task waiting for then it will select highest priority task from the priority queue with the help of delete() function and allocate the resource to that task. If there is no task waiting for the resource then it will increase the value of S by one to make the resource free. The delete() function will return the TCB of the highest priority task (which at the front of the queue). The delete() function must be re-entrant. Therefore, the task switching is disabled at the beginning of this function and enabled at the end of the function.

B. Multiple Priorities Queuing Semaphore Implementation

In this approach, a separate queue is maintained for each priority level. The request tasks will be added at rear of the queue which belongs to its priority value. The resource will be allocated to the highest priority waiting task. If there is no task waiting in the highest priority queue then it will check the next priority level queue. The implementation details are described below:

Data structure:

The dynamic multiple queues are used for each priority level which will be implemented by using header linked list so that there will not be any limitation for the waiting tasks. The number of waiting tasks will be restricted based on the available/allotted memory space. The first node of the list contains the information about the number of tasks belongs to that priority level are waiting for resources, pointer to the TCB of first waiting task, and the pointer to the last node of the queue. The structure of the node of linked list will be as follows (figure 4 and figure 5):

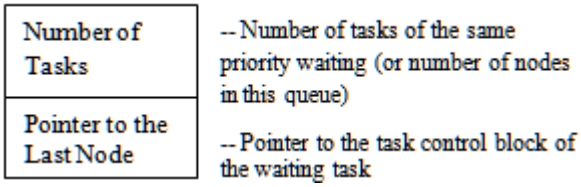

Figure 5: Data Structure for multiple priority queuing semaphore implementation (Header Node)

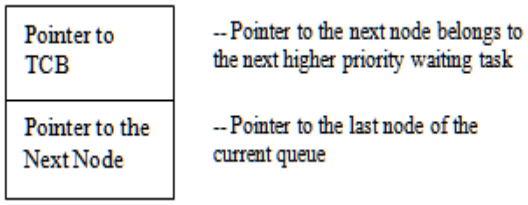

Figure 4: Data Structure for multiple priority queuing semaphore implementation (Header Node)

Psuedocode and logic description of wait(S) operation:

```
wait(S)

{
        if(S == 0)
                then
                insert(PQ(S), TCB, priority);
        else
                S = S-1;
}

Insert(PQ(S), TCB, priority)

{
        Disable task switching
        Newnode->TCB;
        Newnode->Next == NULL;
        PQ[priority]->no_of_tasks++;
        PQ[priority]->last_node->Next = Newnode;
        Enable task switching
}
```

The wait(S) operation first check whether the resource is free by checking the value of S if it is free then it will decrease the value of S and allow the task to enter in the critical section. If the resource is not free then the TCB of the requesting task will be inserted in the queue of its own priority value with the help of another function insert(). The insert() function first prepare a node for the new waiting task, then append the newnode in the queue at the rear. The insert() function must be re-entrant. Therefore, the task switching is disabled at the beginning of this function and enabled at the end of the function.

Psuedocode and logic description of signal(S) operation:

```
signal(S)
{
        tasknode = delete(PQ(S), TCB)
        if tasknode != NULL
                then
                allocate the resource to the
        tasknode
        else
                S = S+1;
}

Delete(PQ(S), TCB)

{
        Disable task switching
        i = highestpriority;
        while (PQ[i]->no_of_task == 0)
                i++;
                node = PQ[i]->Next;
                PQ[i]->Next = node->Next;
TCB = node->TCB;
free(node);
PQ[i]->no_of_task--;
        Enable task switching
}
```

The signal(S) operation first checks whether any other task waits for the resource which is going to free by the current task. if any task waiting for then it will select highest priority task from the priority queue with the help of delete() function and allocate the resource to that task. If there is no task waiting for the resource then it will increase the value of S by one to make the resource free. The delete() function will return the TCB of the highest priority task (which at the front of the queue). The delete() function must be re-entrant. Therefore, the task switching is disabled at the beginning of this function and enabled at the end of the function.

VI. PRIORITY INVERSION

The real trouble arises at run-time, when a medium-priority task preempts a lower-priority task using a shared resource on which the higher-priority task is pending. If the higher-priority task is otherwise ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur.

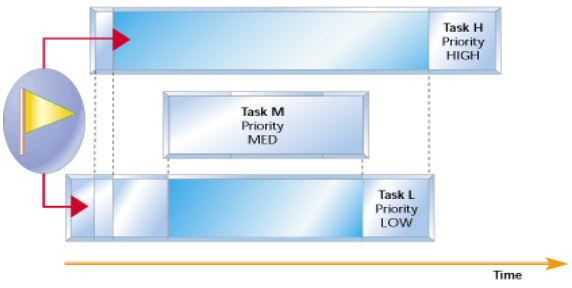

Figure 6: Priority inversion timeline

This dangerous sequence of events is illustrated in Figure 6. Low-priority Task L and high-priority Task H share a resource. Shortly after Task L takes the resource, Task H becomes ready to run. However, Task H must wait for Task L to finish with the resource, so it ends. Before Task L finishes with the resource, Task M becomes ready to run, preempting Task L. While Task M (and perhaps additional intermediate-priority tasks) runs, Task H, the highest-priority task in the system, remains in a pending state.

To avoid unbounded priority inversion when accessing shared resources, preemptive scheduling requires the implementation of specific concurrency control protocols, such as Priority Inheritance, Priority Ceiling or Stack Resource Policy.

1. Priority inheritance:

This technique mandates that a lower-priority task inherit the priority of any higher-priority task pending on a resource they share. This priority change should take place as soon as the high-priority task begins to end; it should end when the resource is released. This requires help from the operating system.

2. Priority Ceiling or Stack Resource Policy:

Priority ceilings associate a priority with each resource; the scheduler then transfers that priority to any task that accesses the resource. The priority assigned to the resource is the priority of its highest-priority user, plus one. Once a task finishes with the resource, its priority returns to normal. A beneficial feature of the priority ceiling solution is that tasks can share resources simply by changing their priorities, thus eliminating the need for mutexes.

But these two solutions introduce additional overhead and complexity, whereas non-preemptive scheduling automatically prevents unbounded priority inversion. On the other hand, fully non-preemptive scheduling is too inflexible for certain applications and could introduce large blocking times that would prevent guaranteeing the schedulability of the task set.

To overcome such difficulties, different scheduling approaches have been proposed in the literature to avoid arbitrary preemptions and limit the length of non-preemptive execution.[2]

1) Fixed Preemption Points (FPP): According to this model, each task is divided into a number of nonpreemptive chunks (also called subjobs) by inserting predefined preemption points in the task code. If a higher priority task arrives between two preemption points of the running task, preemption is deferred until the next preemption point.[2]

2) Floating Non-Preemptive Regions (NPR): Another approach is to define for each task τi a maximum interval Qi in which the task can execute nonpreemptively. Since the mode switching is triggered by the arrival time of higher priority tasks, which is unknown a priori, in this model, the non-preemptive regions have no fixed start time, and are considered to be "floating" in the task code.[2]

3) Preemption Thresholds: A different approach for limiting preemptions is based on the concept of preemption thresholds, proposed by Wang and Saksena under fixed priority systems. This method allows a task to disable preemption up to a specified priority, which is called preemption threshold. Each task is assigned a regular priority and a preemption threshold, and the preemption is allowed to take place only when the priority of arriving task is higher than the threshold of the running task. This work has been later improved by Regehr in [2].

7. COMPARISON AMONG VARIOUS SEMAPHORE IMPLEMENTATIONS

Basis	Busy-Wait Implementation	Queuing Implementation	Priority queue based Implementations
Complexity	Non-deterministic	Deterministic	Deterministic
Prioritization	No concept of priorities	No concept of priorities	Arrange the tasks according to their priority.
Data Structure	No additional data structure is required	Simple circular queue data structure is required to store the pointers to the TCBs of the waiting tasks	Priority Queues or multiple queue are required to store the pointers to the TCBs of the waiting tasks

The comparison among busy-wait, queuing and proposed implementations of semaphores is described in the table 1:

TABLE 1 Comparison Between Various Semaphore Implementations

Deadlock	May leads to deadlock	There will not be any deadlock	There will not be any deadlock due to the pre-emptive nature
Starvation	There is a high possibility of starvation	No possibility of starvation due to FIFO	The low priority tasks can be starved by the high priority tasks. The solution is priority inversion.
Pre-emption within the Critical Section	Not allowed	Not allowed	A low priority process may be pre-empted if any other high priority process is ready.
CPU Overhead	CPU time is wasted in testing of semaphore	There is little CPU overhead for inserting the request in to queue and retrieving the tasks from the waiting queue	There is overhead included for inserting the task in waiting queue at appropriate position or in the appropriate priority queue. Similarly in retrieval of the task from the waiting queue.

VII. CONCLUSION

The task synchronization is one of the important issues in the general operating systems as well as in the real-time operating systems. Due to the pre-emption during the execution of critical section, the task synchronization becomes more crucial in RTOS. It is requires to take care about the priorities of the processes during task synchronization.

Sometimes, the higher priority tasks may be blocked by the lower priority tasks because they have acquired the resources earlier but before release they were pre-empted by the higher priority task. This situation may leads to the deadlocks. To avoid the deadlocks in this situation we can use priority inversion in which temporarily the priority of the lower priority process will be increased so that it can get the CPU time and can complete its critical section execution and can release the resource. After releasing the appropriate resource the priority of that task was reverted back.

Therefore, use some implementation for task synchronization in RTOS which can support priorities, deadlines, priority inversion to accomplish the tasks in real-time environment.

REFERENCES

[1]. Manoj K. Gupta, Rakesh K. Arora," Priority Queue Based Implementation of Semaphore for RTOS", Published in International Journal of Advanced Engineering & Application, Jan 2011

[2]. Gang Yao, Giorgio Buttazzo and Marko Bertogna," Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points", Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA 2010), Macau, China, August 23-25, 2010.

[3]. Giorgio,"Why Real Time Computing", Proceedings of the ANIPLA International Congress on Methadology on emerging Technology,2006

[4]. Stewart, David and Michael Barr. "Rate Monotonic Scheduling," Embedded Systems Programming, March 2002, pp. 79-80.

[5]. Krithi Ramamritham," Scheduling Algorithms and Operating Systems Support for Real-Time Systems" PROCEEDINGS OF THE IEEE, VOL. *82*. NO. I , January 1994.