ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

# Selective Scheduling Algorithms of RTOS A Study of Task Based Scheduling

[1] Minal V. Domke, [2] Nikita R. Hatwar, [3] Mrs. Mrudula M. Gudadhe [4] Priyanka V. Thakare
[1][2][3][4] Asst. Professor, Department of Information Technology
Piyadarhini College of Engineering, Nagpur

*Abstract* - **In this paper, study of the classic real-time scheduling algorithms is done. Many papers have been published in the field of real-time scheduling. The problem of scheduling is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. The quality of real-time scheduling algorithm has a direct impact on real-time system's working. After studied popular scheduling algorithms mainly EDF and RM for periodic tasks with hard deadlines tried to describe performance parameters use to compare the performances of the various algorithms. Observation is that the choice of a scheduling algorithm is important in designing a real-time system .Conclusion by discussing the results of the survey and suggests future research directions in the field of RTOS.**

*Keywords:---* **DEADLINE, EDF,LLF, PCP ,RTOS, SCHEDULABILITY**

## I. INTRODUCTION

In real-time systems the correctness of systems depends not only on the computed results but also on the time at which results are produced. In other words, the major constraint in real-time system is timing requirements that must be guaranteed with accurate results. This leads to the notion of deadline which is a common thread among all real-time system models and the core of the difference between real-time systems and time-sharing systems. The deadline of a task is the point in time before which the task must complete its execution. There can be three types of deadlines, which are mentioned below. Soft Deadline: If the results produced after the deadline has passed and are still useful then this type of deadline is known as soft deadline. Reservation systems come under this category. Firm deadline: This deadline is one in which the results produced after the deadline is missed is of no utility. Infrequent deadline misses are tolerable. These types of deadlines are used in systems which are performing some important operations. Hard deadline: If catastrophe results on missing the deadline then this type of deadline is known as hard deadline. The systems which are performing critical applications like air traffic control come under this category.

There are two kinds of real-time tasks, depending on their arrival pattern: periodic tasks (the task has a regular inter-arrival time called the period, a deadline and a computation time) and aperiodic tasks (the task can arrive at any time; such a task is characterized by a computation time and a deadline; the latter is usually considered as soft). An essential component of a computer system is the scheduling mechanism that is the strategy by which the system decides which task should be executed at any given time. The problem of real-time scheduling is different from that of multiprogramming time-sharing scheduling because of the role of timing constraints in the evaluation of the system performance. Normal multiprogramming time-sharing systems are expected to process multiple job streams simultaneously, so the scheduling of these jobs has the goals of maximizing throughput and maintaining fairness. In real-time systems the primary performance is not to maximize throughput or maintain fairness, but instead to perform critical operations within a set of user-defined critical time constraints.

## II. METHODS AND MATERIAL

As a summary, we divide real-time applications into the following four types according to their timing attributes.

***1. Purely cyclic:*** Every task in a purely cyclic application executes periodically. Even I/O operations are polled. Moreover, its demands in (computing, communication, and storage) resources do not vary significantly from period to period. Most digital controllers and real-time monitors are of this type.

***2. Mostly cyclic:*** Most tasks in a mostly cyclic system execute periodically. The system must also respond to some external events (fault recovery and external commands) asynchronously.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

Examples are modern avionics and process control systems.

***3. Asynchronous and somewhat predictable:*** In applications such as multimedia communication, radar signal processing, and tracking, most tasks are not periodic. The duration between consecutive executions of a task may vary considerably, or the variations in the amounts of resources demanded in different periods may be large. However, these variations have either bounded ranges or known statistics.

***4. Asynchronous and unpredictable:*** Applications that react to asynchronous events and have tasks with high run-time complexity belong to this type. An example is intelligent real-time control systems [SKNL].

The *release time* of a job is the instant of time at which the job becomes available for execution. The job can be scheduled and executed at any time at or after its release time whenever its data and control dependency conditions are met. It is more natural to state the timing requirement of a job in terms of its *response time,* that is, the length of time from the release time of the job to the instant when it completes. We call the maximum allowable response time of a job its *relative deadline.* The deadline of a job, sometimes called its *absolute deadline,* is equal to its release time plus its relative deadline. In general, we call a constraint imposed on the timing behaviour of a job a *timing constraint.* In its simplest form, a timing constraint of a job can be specified in terms of its release time and relative or absolute deadlines, as illustrated by the above example. Some complex timing constraints cannot be specified conveniently in terms of release times and deadlines.

## *A. PERIODIC TASK MODEL*

The *periodic task model* is a well-known deterministic workload model .With its various extensions, the model characterizes accurately many traditional hard real-time applications, such as digital control, real-time monitoring, and constant bit-rate voice/video transmission. Many scheduling algorithms based on this model have good performance and well-understood behaviour. There are now methods and tools to support the design, analysis, and validation of real-time systems that can be accurately characterized by the model. For these reasons, we want to know it well and be able to use it proficiently.

### *A.1 Periods, Execution Times, and Phases of Periodic Tasks*

In the periodic task model, each computation or data transmission that is executed repeatly at regular or semiregular time intervals in order to provide a function of the system on a continuing basis is modeled as a *period task.* Specifically, each periodic task, denoted by *Ti* , is a sequence of jobs. The *period pi* of the periodic task *Ti* is the minimum length of all time intervals between release times of consecutive jobs in *Ti.* Its *execution time* is the maximum execution time of all the jobs in it. With a slight abuse of the notation, we use *ei* to denote the execution time of the periodic task *Ti* , as well as that of all the jobs in it. At all times, the period and execution time of every periodic task in the system are known. This definition of periodic tasks differs from the one often found in real-time systems literature. In many published works, the term periodic task refers to a task that is truly periodic, that is, interrelease times of all jobs in a periodic task are equal to its period. This definition has led to the common misconception that scheduling and validation algorithms based on the periodic task model are applicable only when every periodic task is truly periodic. What are called periodic tasks here are sometimes called sporadic tasks in literature.A sporadic task is one whose interrelease times can be arbitrarily small.

The accuracy of the periodic task model decreases with increasing jitter in release times and variations in execution times. So, a periodic task is an inaccurate model of the transmission of a variable bit-rate video, because of the large variation in the execution times of jobs (i.e., transmission times of individual frames). A periodic task is also an inaccurate model of the transmission of cells on a real-time connection through a switched network that does not do traffic shaping at every switch, because large release-time jitters are possible.

We call the tasks in the system *T1, T2, . . . , Tn.2* When it is necessary to refer to the individual jobs in a task *Ti* , we call them *Ji,1, Ji,2* and so on, *Ji,k* being the *kth* job in *Ti.* When we want to talk about properties of individual jobs but are not interested in the tasks to which they belong, we also call the jobs *J1, J2,* and so on.
The release time *ri,1* of the first job *Ji,1* in each task *Ti* is called the phase of Ti. For the sake of convenience, we use *φi* to denote the phase of *Ti* , that is, *φi* = *ri,1.* In general, different tasks may have different phases. Some tasks are *in phase,* meaning that they have the same phase.
We use *H* to denote the least common multiple of *pi* for *i* = 1, 2, . . . *n.* A time interval of length *H* is called a *hyperperiod* of the periodic tasks. The (maximum) number

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

*N* of jobs in each hyperperiod is equal to _n *i=1 H/pi* . The length of a hyperperiod of three periodic tasks with periods 3, 4, and 10 is 60. The total number *N* of jobs in the hyperperiod is 41.

We call the ratio *ui* = *ei /pi* the *utilization* of the task *Ti* . *ui* is equal to the fraction of time a truly periodic task with period *pi* and execution time *ei* keeps a processor busy. It is an upper bound to the utilization of any task modeled by *Ti.* The *total utilization U* of all the tasks in the system is the sum of the utilizations of the individual tasks in it. So, if the execution times of the three periodic tasks are 1, 1, and 3, and their periods are 3, 4, and 10, respectively, then their utilizations are 0.33, 0.25 and 0.3. The total utilization of the tasks is 0.88; these tasks can keep a processor busy at most 88 percent of the time.

A job in *Ti* that is released at *t* must complete *Di* units of time after *t* ; *Di* is the (relative) *deadline* of the task *Ti* . We will omit the word "relative" except where it is unclear whether by deadline, we mean a relative or absolute deadline.We will often assume that for every task a job is released and becomes ready at the beginning of each period and must complete by the end of the period. In other words, *Di* is equal to *pi* for all *n.* This requirement is consistent with the throughput requirement that the system can keep up with all the work demanded of it at all times. However, in general, *Di* can have an arbitrary value. In particular, it can be shorter than *pi* . Giving a task a short relative deadline is a way to specify that variations in the response times of individual jobs (i.e., jitters in their completion times) of the task must be sufficiently small. Sometimes, each job in a task may not be ready when it is released. (For example, when a computation job is released, its input data are first transferred to memory. Until this operation completes, the computation job is not ready.) The time between the ready time of each job and the end of the period is shorter than the period. Similarly, there may be some operation to perform after the job completes but before the next job is released. Sometimes, a job may be composed of dependent jobs that must be executed in sequence. A way to enforce the dependency relation among them is to delay the release of a job later in the sequence while advancing the deadline of a job earlier in the sequence. The relative deadlines of jobs may be shortened for these reasons as well.

### *A.2 Aperiodic and Sporadic Tasks*

Earlier, we pointed out that a real-time system is invariably required to respond to external events, and to respond, it executes aperiodic or sporadic jobs whose release times are not known a priori. An operator's adjustment of the sensitivity setting of a radar surveillance system is an example. The radar system must continue to operate, but in addition, it also must respond to the operator's command. Similarly, when a pilot changes the autopilot from cruise mode to standby mode, the system must respond by reconfiguring itself, while continuing to execute the control tasks that fly the airplane. A command and control system must process sporadic data messages, in addition to the continuous voice and video traffic.

In the periodic task model, the workload generated in response to these unexpected events is captured by aperiodic and sporadic tasks. Each *aperiodic* or *sporadic task* is a stream of aperiodic or sporadic jobs, respectively. The interarrival times between consecutive jobs in such a task may vary widely and, in particular, can be arbitrarily small. The jobs in each task model the work done by the system in response to events of the same type. For example, the jobs that execute to change the detection threshold of the radar system are in one task; thejobs that change the operation mode of the autopilot are in one task; and the jobs that process sporadic data messages are in one task, and so on.

Specifically, the jobs in each aperiodic task are similar in the sense that they have the same statistical behavior and the same timing requirement. Their interarrival times are identically distributed random variables with some probability distribution *A(x).* Similarly, the execution times of jobs in each aperiodic (or sporadic) task are identically distributed random variables, each distributed according to the probability distribution *B(x).* These assumptions mean that the statistical behavior of the system and its environment do not change with time, that is, the system is stationary. That the system is stationary is usually valid in time intervals of length on the order of *H,* in particular, within any hyperperiod of the periodic tasks during which no periodic tasks are added or deleted.

We say that a task is *aperiodic* if the jobs in it have either soft deadlines or no deadlines. The task to adjust radar's sensitivity is an example. We want the system to be responsive, that is, to complete each adjustment as soon as possible. On the other hand, a late response is annoying but tolerable. We therefore want to optimize the responsiveness of the system for the aperiodic jobs, but never at the expense of hard real-time tasks whose deadlines must be met at all times.

In contrast, an autopilot system is required to respond to a pilot's command to disengage the autopilot and take over the control manually within a specific time. Similarly, when a transient fault occurs, a fault-tolerant system may be required to detect the fault and recover from it in time. The jobs that execute in response to these events have hard deadlines. Tasks containing jobs that are released at random time instants and have hard deadlines are *sporadic tasks.* We treat them as hard real-time tasks. Our primary concern is to ensure that their deadlines are always met; minimizing their response times is of secondary importance.

On the basis of nature of the task we can compare different scheduling algorithms for RTOS under different parameters. Better scheduling algorithm - Earliest Deadline First (deadline driven scheduling) is too complex to be implementedinreal-timeoperatingsystem [11].Timmerman [10] describes the framework for evaluation of realtime operating systems. Baskiyar [13] and et al. have made an extensive survey on memory management and scheduling in RTOS. A worst case response time analysis of real time tasks under hierarchical fixed priority pre-emptive scheduling is done by Bril and Cuijpers. Yaasuwanth [3] and et. al. have developed an modified RR algorithm for scheduling in real time systems. Recently, a number of CPU scheduling algorithms have been developed for predictable allocation of processor [12]. From the work done by the various researchers in the field of real time scheduling; so far, it has been observed that

**a.** Scheduling should be done in order to guarantee the schedule of the processes fairly and throughput must be maximum.
**b.** Real time scheduling algorithms are always pre-emptive which can perform better if the pre-emption is limited.
**c.** Static priority scheduling algorithms are used for scheduling real time tasks for maximum CPU utilization but it can be increased more using dynamic priorities.
**d.** The schedulability of scheduling algorithm must be checked using schedulability tests. **e.** Starvation should not be there which means a particular process should not be held indefinitely. Allocation of resources should be such that all the processes get proper CPU time in order to prevent starvation. **f.** In case of priority based algorithms, there should be fairness in the pre-emption policy. Low priority tasks should not wait indefinitely because of higher priority tasks.

## III.CLASSIFICATION OF SCHEDULING ALGORITHMS

A *scheduler* provides a policy for ordering the execution of tasks on the processor, according to some criteria. Schedulers produce a schedule for a given set of processes. There are several classifications of schedulers. Here are the most important:

- ***Optimal or non-optimal:*** An optimal scheduler can schedule a task set if the task set is schedulable by some scheduler.

- ***Preemptive or non-preemptive:*** A preemptive scheduler can decide to suspend a task (before finishing its execution) and restart it later, generally, because a higher priority task becomes ready. Non-preemptive schedulers do not suspend tasks in this way. Once a task has started, it cannot be suspended involuntarily.

- ***Static or dynamic:*** Static schedulers calculate the execution order of tasks before run-time. It requires knowledge of task characteristics but produces little run-time overhead. However, it cannot deal with aperiodic or non-predicted events. Some references about this kind of schedulers can be found in [20]. Dynamic schedulers, on the contrary, make decisions during the run-time of the system. This allows to design a more flexible system, but it means some overhead.

Three commonly used approaches to scheduling realtime systems: clock-driven, weighted round-robin and priority-driven. The subsequent five chapters will study in depth the clock-driven and priority-driven approaches

- ***Clock – driven Scheduling***

As the name implies, when scheduling is *clock-driven* (also called *time-driven),* decisions on what jobs execute at what times are made at specific time instants. These instants are chosen a priori before the system begins execution. Typically, in a system that uses clock-driven scheduling, all the parameters of hard real-time jobs are fixed and known. A schedule of the jobs is computed off-line and is stored for use at run time. The scheduler schedules the jobsaccording to this schedule at each scheduling decision time. In this way, scheduling overhead during run-time can be minimized.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

A frequently adopted choice is to make scheduling decisions at regularly spaced time instants. One way to implement a scheduler that makes scheduling decisions periodically is to use a hardware timer. The timer is set to expire periodically without the intervention of the scheduler. When the system is initialized, the scheduler selects and schedules the job(s) that will execute until the next scheduling decision time and then blocks itself waiting for the expiration of the timer. When the timer expires, the scheduler awakes and repeats these actions.

- ***Weighted round-robin Scheduling***

The round-robin approach is commonly used for scheduling time-shared applications. When jobs are scheduled on a round-robin basis, every job joins a First-in-first-out (FIFO) queue when it becomes ready for execution. The job at the head of the queue executes for at most one time slice. (A time slice is the basic granule of time that is allocated to jobs. In a timeshared environment, a time slice is typically in the order of tens of milliseconds.) If the job does not complete by the end of the time slice, it is preempted and placed at the end of the queue to wait for its next turn. When there are *n* ready jobs in the queue, each job gets one time slice every *n* time slices, that is, every *round.* Because the length of the time slice is relatively short, the execution of every job begins almost immediately after it becomes ready.

In essence, each job gets 1/nth share of the processor when there are *n* jobs ready for execution. This iswhy the round-robin algorithm is also called the processor-sharing algorithm. The *weighted round-robin algorithm* has been used for scheduling real-time traffic in high-speed switched networks. It builds on the basic round-robin scheme. Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different *weights.* Here, the weight of a job refers to the fraction of processor time allocated to the job. Specifically, a job with weight *wt* gets *wt* time slices every round, and the length of a round is equal to the sum of the weights of all the ready jobs. By adjusting the weights of jobs, we can speed up or retard the progress of each job toward its completion.

By giving each job a fraction of the processor, a round-robin scheduler delays the completion of every job. If it is used to schedule precedence constrained jobs, the response timeof a chain of jobs can be unduly large. For this reason, the weighted round-robin approach is not suitable for scheduling such jobs. On the other hand, a successor job may be able to incrementally consume what is produced by a predecessor (e.g., as in the case of a UNIX pipe). In this case, weighted round-robin scheduling is a reasonable approach, since a job and its successors can execute concurrently in a pipelined fashion. As an example, we consider the two sets of jobs, **J1** = {J1,1, *J1,2}* and **J2** = {J2,1, *J2,2},* shown in Figure 1-a ,Figure 1-b. The release times of all jobs are 0, and their execution times are 1. *J1,1* and *J2,1* execute on processor *P1,* and *J1,2* and *J2,2* execute on processor *P2.* Suppose that *J1,1* is the predecessor of *J1,2,* and *J2,1* is the predecessor of *J2,2.*

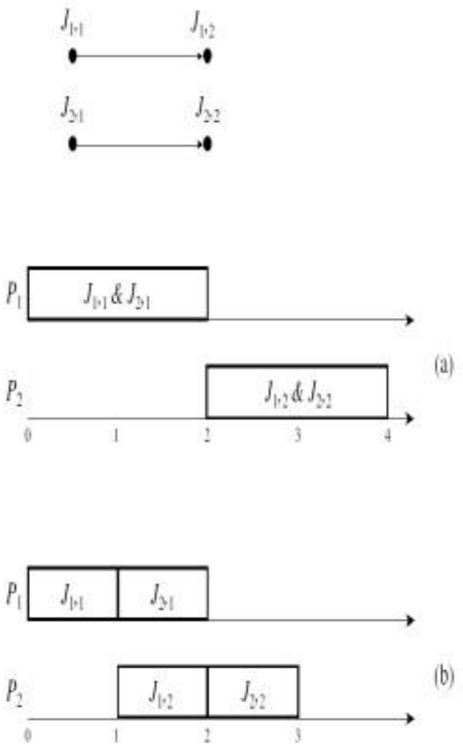

***Figure 1. Example illustrating Round-robin Scheduling***

- ***Priority – driven Scheduling***

The term *priority-driven* algorithms refers to a large class of scheduling algorithms that never leave any resource idle intentionally. Stated in another way, a resource idles only when no job requiring the resource is ready for execution. Scheduling decisions are made when events such as releases and completions of jobs occur. Hence, priority-driven algorithms are *event-driven.*

Other commonly used names for this approach are *greedy scheduling, list scheduling* and *work-conserving scheduling.* A priority-driven algorithm is greedy because it tries to make locally optimal decisions. Leaving a resource idle while some job is ready to use the resource is not locally optimal. So when a processor or resource is available and some job can use it to make progress, such an algorithm never makes the job wait. We will return shortly

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

to illustrate that greed does not always pay; sometimes it is better to have some jobs wait even when they are ready to execute and the resources they require are available.
The term list scheduling is also descriptive because any priority-driven algorithm can be implemented by assigning priorities to jobs. Jobs ready for execution are placed in one or more queues ordered by the priorities of the jobs. At any scheduling decision time, the jobs with the highest priorities are scheduled and executed on the available processors. Hence, a priority-driven scheduling algorithm is defined to a great extent by the list of priorities it assigns to jobs; the priority list and other rules, such as whether preemption is allowed, define the scheduling algorithm completely.

### *3.1Priority Driven Scheduling*

#### 3.1.1 Fixed-priority scheduling

The most important scheduling algorithms in this category are Rate Monotonic (RM) [2] and Deadline Monotonic (DM) [4]. The former assigns the higher priority to the task with the shortest period, assuming that periods are equal to deadlines. The latter assigns the highest priority to the task with the shortest deadline. Both algorithms are optimal. Fixed-priority scheduling has been widely studied and the most important real-time operating systems have a fixed-priority scheduler.

#### *3.1.1.1 Schedulability analysis*

The first test was provided by Lui and Layland [2]. It is based on the processor utilization of the task set. The total utilization of the set is the sum of the utilizations of all tasks in the set, which is obtained as the quotient of execution time by the period. This utilization is compared with the utilization bound (that depends on the number of tasks). Thus, if the utilization of the set is less or equal to the utilization bound, the task set is schedulable. This schedulability constraint is a sufficient, but not a necessary condition. That is, there are task sets that can be scheduled using a rate monotonic priority algorithm, but which break the utilization bound. A sufficient and necessary condition was developed by Lehoczky [10] and Audsley. This test is based on the worst case response time of every task. If, in the worst case, a task finishes its execution before its deadline, the task will be schedulable. The worst case response time of a task occurs in the first activation. Moreover, this test is valid for any priority assignation, and it informs not only whether the set is feasible or not, but the task or tasks that miss its deadline.

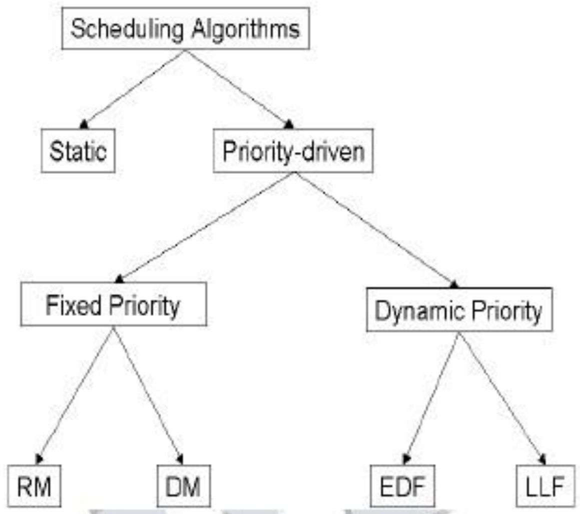

*Figure 2. Scheduler's classification*

***3.1.2 Dynamic-priority scheduling*** Within this category, Earliest Deadline First (EDF) [2] and Least Laxity First (LLF) [13] are the most important. Both are optimal, if any algorithm can find a schedule where all tasks meet their deadline then EDF can meet the deadlines. In EDF, the highest priority task is the task with the nearest absolute deadline. The absolute deadline is the point in time in which it arrives the deadline of the current activation of the task. LLF assigns priorities depending on the *laxity,* being the task with the lower laxity, the highest priority task. The term *laxity* refers to the interval between the current time and the deadline, minus the execution time that remains to execute. Dynamic-priority algorithms have interesting properties when compared to fixed-priority. They achieve high processor utilization, and they can adapt to dynamic environments, where task parameters are unknown. On the contrary, real-time systems community is reluctant to use dynamic-priority algorithms mainly because of the instability in case of overloads. It is also not possible to known what task miss its deadline if the system is not feasible.

#### *3.1.2.1Schedulability analysis*

In [2] it is proved that EDF can guarantee schedulability of tasks when the processor utilization is less than 100%. In this case, deadlines have to be equal than periods, but Dertouzos also proved that EDF is optimal when deadlines are less than periods.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 4, Issue 3, March 2017

## IV. BASIC PARAMETERS THAT AFFECT THE PERFORMANCE IN RTOS [13]

***4.1 Multi-tasking and preemptable:*** To support multiple tasks in real-time applications, an RTOS must be multi-tasking and preemptable. The scheduler should be able to preempt any task in the system and give the resource to the task that needs it most. An RTOS should also handle multiple levels of interrupts to handle multiple priority levels.

***4.2 Dynamic deadline identification:*** In order to achieve preemption, an RTOS should be able to dynamically identify the task with the earliest deadline [11]. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation. Although such an approach is error prone, nonetheless it is employed for lack of a better solution.

***4.3 Predictable synchronization:*** For multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Semantic integrity as well as timelinessconstitutespredictability.Predictable synchronization requires compromises.

***4.4 Sufficient Priority Levels:*** When using prioritized task scheduling, the RTOS must have a sufficient number of priority levels, for effective implementation. Priority inversion occurs when a higher priority task must wait on a lower priority task to release a resource and in turn the lower priority task is waiting upon a medium priority task. Two workarounds in dealing with priority inversion, namely priority inheritance and priority ceiling protocols (PCP), need sufficient priority levels.

***4.5 Predefined latencies:*** The timing of system calls must be defined using the following specifications: • Task switching latency or the time to save the context of a currently executing task and switch to another. • Interrupt latency or the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the interrupt handler.

## V. RESULTS AND DISSCUSIONS

***5.1 Scheduling with fixed priority algorithm (RM)***

We schedule the following task set with FP (RM priority assignment).

$\tau 1 = (1, 4)$, $\tau 2 = (2, 6)$, $\tau 3 = (3, 8)$.

U =1/4 +2/6+3/8 =23/24 The utilization is greater than the bound: there is a deadline miss.

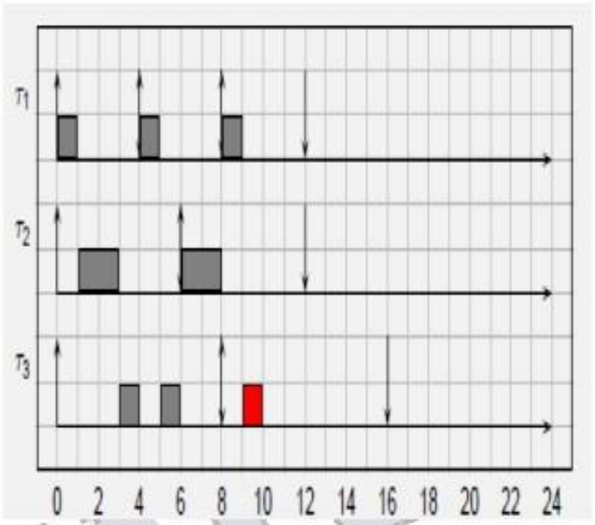

***Figure 3. Timeline for RM***

Observe that at time 6, even if the deadline of task τ 3 is very close, the scheduler decides to schedule task τ2. This is the main reason why τ3 misses its deadline!

***5.2 Scheduling with dynamic priority algorithm (EDF)***

Now we schedule the same above task set with EDF. τ 1 = (1, 4), τ 2 = (2, 6), τ3 = (3, 8). *U* = 1/4 + 2/6 + 3/8 = 23/24 Again, the utilization is same.

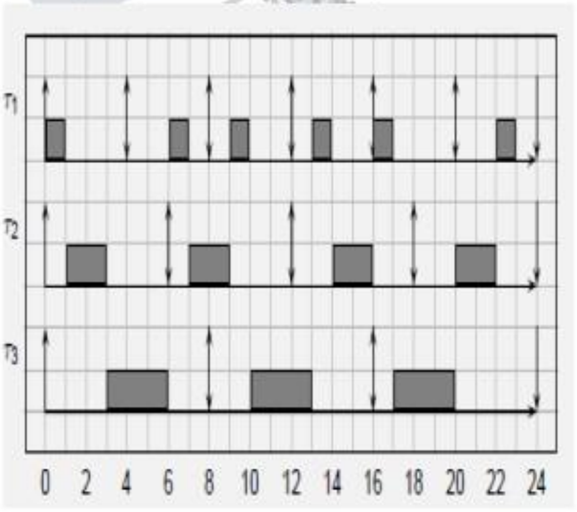

***Figure 4. Timeline for EDF***

However, no deadline miss in the hyperperiod.
Observe that at time 6, the problem does not appear, as the earliest deadline job (the one of τ 3) is executed. EDF is an optimal algorithm, in the sense that if a task set if schedulable, then it is schedulable by EDF. EDF can schedule all task sets that can be scheduled by fixed priority, but not vice versa. There is no need to define

priorities. In fixed priority, in case of offsets, there is not an optimal priority assignment that is valid for all task sets. In general, EDF has less context switches. In the previous example, the number of context switches in the first interval of time: in particular, at time 4 there is no context switch in EDF, while there is one in fixed priority. As no of context switches are less processor utilization is greater, less idle times. EDF is less predictable: Looking back at the example, the response time of task τ1: in fixed priority is always constant and minimum; in EDF is variable. If we want to reduce the response time of a task, in fixed priority is only sufficient to give him an higher priority; in EDF we cannot do anything; we have less control over the execution. Fixed priority can be implemented with a very low overhead even on very small hardware platforms (for example, by using only interrupts); EDF instead requires more overhead to be implemented. Computing the response time in EDF is very difficult. EDF is still optimal when relative deadlines are not equal to the periods.

## VI. CONCLUSION AND FUTURE SCOPE

From the comparative study it can be concluded that since the concept of "time" is of such importance in real-time application systems, and since these systems typically involve various contending processes, the concept of scheduling is integral to real-time system design and analysis. Scheduling and schedulability analysis enables these guarantees to be provided. From the comparison of real time scheduling algorithms, it is clear that earliest deadline first is the efficient scheduling algorithm if the CPU utilization is not more than 100% but does scale well when the system is overloaded. In the experimental environment, although EDF scheduling algorithm can meet the needs of real-time applications, in practical applications, it is still needed to be improved to meet the evolving needs of real-time systems. In future a new algorithm should be developed which is a mix of fixed and dynamic priority or a scheduling algorithm switch automatically between EDF algorithm and fixed based scheduling algorithm to deal overloaded and under loaded conditions. The new algorithm will be very useful when future workload of the system is changeable.

## REFERENCES

[1] Arnoldo Diaz, Ruben Batista and Oskardie Castro, 2013. Realtss: A real-time scheduling simulator. International Conference on Electrical and Electronics Engineering.

[2] C. Liu and James Leyland, January 1973. *Scheduling algorithm for multiprogramming in a hard real-time environment.* Journal of the Association for Computing Machinery, 20(1): 46-61.

[3] C. Yaashuwanth and R. Ramesh, 2010. *Design of real time scheduler simulator and development ofmodified round robin architecture.* International Journal of Computer Applications.

[4]J.Leung and J. Whitehead, 1982. Performance Evaluation, On the complexity of fixed-priority schedulings of periodic, real-time tasks.

[5] Fengxiang Zhang and Alan Burns, September, 2009. *Schedulability analysis for real-time systems with EDF scheduling.* IEEE Transactions on computers, vol. 58, no. 9.

[6] Hamid Arabnejad and Jorge G. Barbosa, 2013. *List scheduling algorithm for heterogeneous systems by an optimistic cost table.* IEEE.

[7] M. Kaladevi and Dr. S. Sathiyabama , 2010. *A comparative study of scheduling algorithms for real time task.* International Journal of Advances in Science and Technology, Vol. 1, No. 4.

[8] Moonju Park and Heemin Park, 2012. *An efficient test method for rate monotonic schedulability.* IEEE.

[9] J. Lehoczky, L. Sha, and Y. Ding, 1989 *The rate monotonic scheduling algorithm: Exact characterization and average case behaviour.* IEEE Real-Time Systems Symposium, 166-171.

[10] Peng Li and Binoy Ravindran September, 2004. *Fast, Best-Effort Real-Time Scheduling Algorithms.* IEEE Transactions on computers, vol. 53, no. 9.

[11] A.Mok and M.Dertouzos, 1978. Multiprocessor scheduling in a hard real-time environment. 7th Texas Conference on Computing Systems.

[12] R. Le Moigne, O. Pasquier, J-P. Calvez 2004. A Generic RTOS Model for Real-time Systems Simulation with SystemC. IEEE.

[13] S. Baskiyar and N. Meghanathan, 2005. *A Survey On Real Time Systems.* Informatica (29), 233-240.

[14] Jane W.S. Liu , Real Time Systems, Prentice Hall Publication,