International Journal of Advance Engineering and Research Development (IJAERD)
Volume 1,Issue 6,June 2014, e-ISSN: 2348 - 4470 , print-ISSN:2348-6406

Synchronization in Embedded Real-Time Operating Systems

Miss. Rani S. Lande[1] , Dr. M. S. Ali[2]

[1]*Department of Computer Science and Engineering, Prof. Ram Meghe College of Engineering & Management, Email id: lande.rani@gmail.com*
[2]*Department of Computer Science and Engineering, Prof. Ram Meghe College of Engineering & Management, Email id: softalis@hotmail.com*

Abstract - Real-time multiprocessor systems are now commonplace. Manufacturers of traditionally uniprocessor embedded systems are also shifting towards multi-processor or multi-core platforms to obtain high computing performance. Embedded systems are the computing devices hidden inside a vast array of everyday products and appliances such as cell phones, toys, handheld PDAs, cameras, etc. An embedded system is various type of computer system or computing device that performs a dedicated function and/or is designed for use with a specific embedded software application. The Real Time Operating System (RTOS) supports applications that meet deadlines in addition to providing logically correct results. In this paper, we will discuss the study and analysis of the scheduling and synchronization. It also presents a system that can be schedule multiple tasks using global EDF scheduler and share the resource among tasks using preemptive synchronization algorithm. The main objective is to share resource among task sets and reduce the average waiting time and jitter for task. It considers global dynamic-priority preemptive multiprocessor scheduling of constrained-deadline periodic tasks that share resources in a non-nested manner. In the addition, a resource-sharing protocol (i.e. priority inheritance protocol) has been discussed to decrease the blocking overhead of tasks, which reduces the total utilization and has the potential to reduce the number of required processors.

Keywords - Real-Time Operating System, Embedded System, Scheduling, Synchronization.

I. INTRODUCTION

The Embedded Real-time operating systems are designed to satisfy the strict requirements from embedded applications which need real-time responses. Some embedded systems run a scaled down version of operating system called Real Time Operating System (RTOS).

Real time embedded operating system needs better response time for real-time process. They have to complete the tasks before the deadlines. But it is difficult to fulfill the condition as embedded system have too many resources and are within different types. They will damage the performance of scheduling algorithms which will make embedded RTOS to lose their deadlines.

The scheduling problem consists of deciding the order of execution and also the period of execution of a set of tasks with certain known characteristics like periodicity and limited set of processing units, which is typically a single processor in embedded systems. Embedded system designers rely more on multi-processor or multi-core platforms to obtain high computing performance [1], [2]. A major issue in developing multi-core computing systems is how to utilize the available computing resources most effectively. There are two kinds of constraints faced by tasks executing in an embedded real-time environment: *Time Constraint and Resource Constraints.*

In real time world most of the tasks have a time constraint, a deadline in executing a particular job. The tasks are also required to have a good response time to increase the response time of the system and execute in a manner so that other tasks can also meet their deadlines. The other constraint that affects the design of an embedded real-time system is resource constraint. In embedded systems there is a limited RAM availability, limited CPU speed, power consumption constraint and most of other resource related constraints. An embedded system is designed to work optimally in spite of the resource constraint problems it has.

Due to the above-mentioned constraints (a combination) there is an immense pressure on embedded operating system performance. An embedded system's performance in presence of constraints is highly correlated with how smart the scheduler is. Up until now, a lot of research has been done on developing schedulers and synchronization algorithm that can meet these constraints. The paper discusses various designs relevant in embedded real-time world and follows them up with our analysis.

To increase the response time of the system, the tasks are also required having a good response time. The embedded systems have a resource constraints as a limited RAM availability, limited CPU Speed, power-consumption. A high performance scheduling and synchronization algorithm (i.e. smart scheduler and synchronizer) is required in order to make embedded operating system more efficient.

To enforce the performance of the embedded systems and satisfy the requirement of embedded real time systems, the paper developed a new real time scheduling and synchronization mechanism. Now a days, Multi-cores are common and applications with real-time constraints are implemented upon them. To enable such implementation, real time scheduling algorithm must have to develop.

Real time response is the core part of the entire embedded real time operating system. The mechanism presented in this paper focuses on the response time of the embedded systems. This work also considered several synchronization mechanisms and an optimal real-time scheduling algorithm based on multi-core processor, several resources sharing methods under multi-core processor for embedded real time systems.

II. BACKGROUND

Real-Time Systems can be distinguished based on the effect of missing their deadlines. They can be grouped into three categories: systems that have hard deadlines, soft deadlines and firm deadlines.

2.1 Real -Time System

The Real-time system is one that must perform operations within rigid timing constraints i.e., processing must be completed within the defined constraints otherwise the system will fail. The correctness of real time system does not depend only on the logical correctness but also on the time it takes to produce the result [1]. Real-time systems have well defined, fixed time constraints. There are **two main types of real-time systems:** Hard Real-Time System (HRT), Firm or Soft Real-Time System (SRT).

In Hard Real-Time System requires that fixed deadlines must be met otherwise disastrous/ catastrophic situation may arise whereas in Soft Real-Time System, missing an occasional deadline is undesirable, but nevertheless tolerable. System in which performance is degraded but not destroyed by failure to meet response time constraints is called soft real time systems.

2.1.1. Soft Real Time system

- Deadline overruns are tolerable, but not desired.
- There are no catastrophic consequences of missing one or more deadlines.
- There is a cost associated to overrunning, but this cost may be abstract.
- Often connected to Quality-of-Service.

2.1.2. Hard Real Time system

- An overrun in response time leads to potential loss of life and/or big financial damage.
- Many of these systems are considered to be safety critical.
- Sometimes they are “only” missioning critical, with the mission being very expensive.
- In general there is a cost function associated with the system.

2.1.3. Firm Real Time system

- The computation is obsolete if the job is not finished on time.
- Cost may be interpreted as loss of revenue.
- Typical examples are forecast systems.

2.2. Embedded System

An Embedded System: It is a combination of hardware and software to perform a specific task. An embedded computer is frequently a computer that is implemented for a particular purpose. In contrast, an average PC computer usually serves a number of purposes: checking email, surfing the internet, listening to music, word processing, etc... However, embedded systems usually only have a single task, or a very small number of related tasks that they are programmed to perform.

2.3. Embedded Real – Time Operating System

An Embedded Real Time System possesses the characteristics of both an embedded system and a real-time system. The embedded systems are resource limited, the memory capacity and processing power in an embedded system is limited as compared to a desktop computer and response time is one of the most important requirements. Some embedded systems run a scaled down version of operating system called Real Time Operating System (RTOS).

The choice of an operating system is important in designing a real time system. Designing a real time system involves choice for proper language, task portioning and merging and assigning priorities using a real time scheduler to manage response time. The depending upon scheduling objectives parallelism and communication may be balanced. The designer of scheduling policy must be determine critical tasks and assign them high priorities. However, care must be taken avoid starvation, which occurs when high priority tasks are always ready to run.

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. A scheduling algorithm is a set of rules that determines which task should be executed in any given instance. Due to the tasks criticality scheduling algorithms should be timely and predictable.

2.3.1 Basic Requirements of Scheduler in RTOS

The following are the basic requirements of Scheduler in RTOS.

- Multitasking and Preemptable
- Dynamic Deadline Identification
- Predictable Synchronization
- Sufficient Priority levels
- Predefined latencies

a) *Multitasking and Preemptable*: To support multitasks in real time applications an RTOS should be multitasking and preemptable. The Scheduler should be able to preempt any task in the system and give resource to the task that needs it.

b) *Dynamic Deadline Identification*: In regulate to achieve preemption; an RTOS should be able to dynamically identify the task with the earliest deadline. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation. It is also employed for lack of a better solution and error less.

c) *Predictable Synchronization*: Multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Predictable synchronization requires compromise. Ability to lock or unlock resources is the ways to achieve data integrity.

d) *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 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 protocol, need sufficient priority levels.

e) *Predefined latencies*: The timing of system calls must be defined using the following specifications:

▪ Task switching latency or 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 first instruction of the interrupt handler.
▪ Interrupt dispatch latency or the time to switch from the last instruction in the interrupt handler to the next task scheduled to run.

The objective of real-time task scheduler is to guarantee the deadline of tasks in the system as much as possible when we consider soft real time system. To achieve this goal, vast researches on real-time task scheduling have been conducted. Mostly all the real time systems in existence use preemption and multitasking.

2.4 Real Time Scheduling

Given a set of tasks Γ= {τ1, τ2,…… τn }, a set of m processors P={P1,P2,…..Pm } and a set of S resources R={R1,R2,…..RS}. There may exists precedence's and we are considering real-time systems, timing constraints are associated to each task. The goal of Real-time scheduling is to assign processors from P and resources from R to tasks from Γ in such a way that all task instances are completed under the imposed constraints. This problem in its general form is NP-complete. Therefore relaxed situations have to be enforced and/or proper heuristics have to be applied.

Before discussing embedded real-time system schedulers, the paper provide an introduction to certain system concepts that carry a lot of significance in embedded real-time systems.

➢ *Periodic Tasks* - The period of a task is the rate with which a particular task becomes ready for execution. Periodic tasks become ready at regular and fixed intervals. Periodic tasks are commonly found in applications such as avionics and process control accurate control requires continual sampling and processing data.

➢ *Deadlines* - All real time tasks have deadline by which a particular job has to be finished. There are scheduling algorithms designed to allow maximum tasks to meet their deadline.

➢ *Laxity* - Laxity is defined as the maximum time a task can wait and still meet the deadline. It can also be used as a measure of scheduling necessity.

➢ *Jitter* - It is defined as the time between when a task became ready and when it actually got executed. For certain real time systems there is an additional constraint that all the tasks should have minimum jitter.

➢ *Schedulability* - A given set of tasks is considered to be schedulable if all the tasks can meet their deadline. In certain on-line scheduling algorithms a new task is subject to schedulability test, wherein it is verified that the new task is schedulable along with the already existing tasks. If the task is not schedulable the task is not permitted to enter the system.

➢ *Utilization* - It is the factor giving a notion of how much CPU is utilized by a given set of tasks.
Meeting deadlines, achieving high CPU utilization with minimum resource and time utilization are considered as the main goals of task scheduling.

2.4.1 Real Time Scheduling Paradigms

Real time scheduling techniques can be broadly divided into two categories: Static and Dynamic. This classification is done on the basis of time of scheduling the processes i.e. whether the
processes are to be scheduled on the compile time or run time.

➢ **Static scheduling**

In this technique, scheduling decisions are made at compile time. For scheduling, complete prior knowledge of task-set characteristics is required. The system's behaviour with static scheduling is deterministic. Static algorithms assign all priorities at design time (before the tasks are entered into the system based on statically defined criterion like deadline, criticality, periodicity etc.). All assigned priorities remain constant for the lifetime of a task. The advantage of using static scheduling procedure is that it involves almost no overhead in deciding which task to schedule. Static scheduling of tasks in embedded real-time systems often implies a tedious iterative design process. The reason for this is the lack of flexibility and expressive power in existing scheduling framework, which makes it difficult to both model the system accurately and provide correct optimizations. This causes systems to be over constrained due to statically decided rules of procedures.

- **Round-robin method**

The simplest of static scheduling procedures is round-robin method. The tasks are checked for readiness in a predetermined order with ready to execute task getting a CPU slice. Each task gets checked for schedulability once per cycle, with scheduling time bound by execution time of other tasks. Apart from simplicity this method has no advantages. The major disadvantage being that urgent tasks always have to wait for their turns, allowing non urgent tasks to execute before the urgent tasks. Also polling tasks for schedulability for readiness is not a good procedure at all. This type of scheduling works well in some simple embedded systems where software in the loop executes quickly and the loop can execute repeatedly at a very rapid rate.

- **Dynamic Scheduling**

Scheduling decisions are made at run time by selecting one task out of the set of ready tasks. Dynamic schedulers are flexible but also require run time in finding a substantial schedule. System's behaviour is non-deterministic. Dynamic algorithms assign priority at runtime, based on execution parameters of tasks. In a dynamic scheduling policy the tasks are dynamically chosen based on their priority dynamically, generally from ordered prioritized queue. The priorities can be assigned statically or dynamically based on different criterions like, deadline, criticality, periodicity etc. Dynamic scheduling can be *preemptive* or *non-preemptive*.

2.4.2 Classification Based on the Number of Processes to be Scheduled

This classification is done considering whether the scheduling is done on single processor or multiple processors.

- **Uniprocessor Scheduling**

If the scheduling is done on a single processor then it is known as uniprocessor scheduling. Round Robin, RM scheduling etc are the examples of uniprocessor scheduling algorithms.

- **Multiprocessor Scheduling**

If number of events occurring close together is high then we have to increase number of processors in the system. Such system is known as multiprocessor systems and scheduling techniques required to schedule a task on such system are known as multiprocessor scheduling algorithm. Global scheduling algorithms and partitioning scheduling algorithms fall under this category.

III. RELATED WORK

Scheduling

J. Anderson and S. Bauruah analyzed the trade-offs involved in scheduling independent, periodic real-time tasks on a multiprocessor. They proposed a new method for classification of scheduling algorithms for scheduling preemptive real time tasks on multiprocessors. Authors described some new classes of scheduling algorithms and also described known scheduling algorithms that fall under these classes [12].

J. Anderson and U. C. Devi proposed a new EDF-fm algorithm for scheduling soft real-time systems on multiprocessors under the condition no restriction on the total system utilization but requires per task utilizations to be at most one half of the processor capacity[11].

A. Srinivasan and J. Anderson considered the scheduling of soft real time task on multiprocessor using EDF algorithm with partitioning. They proved that the percentage of deadlines missed is very low for any threshold. Authors proved the performance (in terms of tardiness and percentage of missed deadline) of EPDF scales well as the number of processors increase [2].

Many techniques are provided by researchers to provide embedded hard real-time operating systems. Wei Hu and T. Chen proposed real time scheduling algorithm for real time embedded system which divides all tasks into different sets and schedule the tasks according to the nature of the set [9].

T. Baker presented the technical report which reviewed hard real-time multiprocessor scheduling and advances in the analysis of arbitrary sporadic task systems under fixed-priority and EDF scheduling policies [3].

B. Ward and J. Herman presented several techniques for managing shared caches on multi-core systems within the mixed-critically scheduling framework. They implemented it on a quad-core ARM machine [6].

B. Brandenburg and J. Anderson presented suspension based real-time locking protocol for clustered schedulers [4].

B. Brandenburg, J. Anderson and J. M. Calandrino produced an extension of the LITMUS^{RT} (Linux Testbed for Multiprocessor Scheduling in Real-Time systems) testbed that incorporates support for synchronization [5].

A. Easwaran and B. Andersson implemented P-PCP, resource sharing protocol and developed schedulability analysis for global fixed priority preemptive multiprocessor scheduling under the same protocol and PIP [7].

S. Khushu and J. Simmons presented a survey of scheduling and synchronization in embedded real-time operating systems [16].

C. Belwal and A. Cheng proposed utilization based necessary and sufficient scheduling condition for a Software Transactional Memory (STM) using lazy conflict detection [10].

R. Jejurikar and R. Gupta proposed algorithms to compute static slowdown factor for a periodic task set for that they consider the effect of blocking that arises due to task synchronization. They proved that the computed slow down factors save on an average 25%-30% energy as compared to known methods [12].

K. Jeffay presented an optimal on-line algorithm for scheduling a set of sporadic tasks. The result of algorithm based on the integration of synchronization for access to shared resources with EDF algorithm [13].

N. Guan, Wang Yiand Ge yu proposed to use cache space isolation technique to avoid cache contention for hard real-time tasks running on multi-cores with shared caches. They proposed scheduling for real-time task with both timing and cache space constraints, which allow each task to use a fixed number of cache partitions and make sure that at any time at most one running task occupied the cache partition [8].

A. Shah and K. Kotecha proposed a new scheduling algorithm Ant Colony Optimization (ACO) algorithm for scheduling Soft real-time tasks in uniprocessor RTOS with preemptive task sets. They observed that the proposed algorithm is equally optimal during underloaded condition and performs better than EDF in overloaded condition [15].

M. Fan and G. Quan proposed a new semi-partitioned algorithm to schedule real-time sporadic tasks on multi-core system under Rate-Monotonic scheduling policy. They proposed HSP-light algorithm to schedule light task set and HSP to schedule general task sets. Authors experimentally presented that the proposed Harmonic scheduling algorithm improved the scheduling performance as compared the other [14].

V. Salmani, S. Taghavi and Zargar presented a modified maximum urgency scheduling algorithm which combines the advantages of fixed and dynamic scheduling. *Modified Maximum Urgency First* (MMUF) scheduling algorithm is as an optimization of *Maximum Urgency First algorithm* (MUF) [43] which is used to predictably schedule dynamically changing systems. The MMUF is a preemptive mixed priority algorithm for predictable scheduling of periodic real-time tasks. It usually has less task preemption and hence, less related overhead. It also leads to less failed non-critical tasks in overloaded situations in which the CPU load factor is greater than 100% [18].

A. Habibi and V. Salmani used a job-level dynamic and practical version of LLF which is called Modified Least Laxity First (MLLF) algorithm instead of the traditional LLF and have compared its performance with EDF algorithm from many different aspects. The success ratio has been chosen as the key factor for evaluation of the algorithms. Authors experimentally proved that in case of job-level dynamic scheduling, deadline-based algorithms have supremacy over laxity-based ones. Furthermore, in most conditions, global policies show a better performance than partition policies. Authors also proved that in most conditions the performance of global laxity-based algorithm is much better than that of its corresponding partition and shows a very close behavior to that of global deadline-based and thus has the potential to be considered for future research [19].

Synchronization

In work on uniprocessor resource sharing, the *priority ceiling protocol* (PCP) [42] and the *stack-based resource allocation protocol* (SRP) [29] have received much attention. For multiprocessor systems, there has been a growing interest in the area of resource synchronization.

Rajkumar *et. al.* were the first to propose a semaphore-based protocol for resource sharing on multiprocessors. Two variants of PCP were presented by them for systems that use partitioned, fixed priority scheduling. Several protocols related to multiprocessor PCP have since been proposed for systems scheduled under partitioned, dynamic-priority (EDF) scheduling [41, 40].

Chen and Tripati proposed two extensions to the basic protocol, but these extensions were only valid for periodic (and not sporadic) task systems. Further, global critical sections were assumed to be non-preemptable and nesting was not allowed between global and local critical sections (each can be separately nested however) [33].

In later work, L`opez *et. al.* presented an implementation of SRP for partitioned EDF. However, this study required that tasks sharing resources be assigned to the same processor [38].

Recently, Gai *et. al.* also presented an implementation of SRP for partitioned EDF and compared it to PCP. They have implemented a FIFO-queue based spin-lock for global critical sections, which has the potential to waste processing time (tasks can busy-wait for other tasks accessing global critical sections). Further, accesses to different global critical sections are not allowed to be nested, and these critical sections are executed in a non-preemptive manner. The latter requires modifications in the kernel to disable preemptions [36].

- *Resource synchronization under global scheduling algorithms*

There have been a few recent studies [34, 37, 32].Under global EDF, Devi *et. al.* proposed a FIFO-queue based spinlock implementation for non-nested critical sections. They also modified the global EDF scheduler to enforce non-preemptive critical sections [6].

Holman and Anderson have proposed various techniques for implementing non-nested critical sections under Pfair [30] global scheduling. They allow FIFO-queue based access to locked resources and present different techniques for handling short and long critical sections [37].

➢ *Resource synchronization under partitioned scheduling algorithms*

Flexible Multiprocessor Locking Protocol (FMLP), proposed by Block *et. al.* can be used under partitioned EDF, global EDF, and Pfair scheduling. They handle short critical sections using FIFO-queue based spin-locks, and long critical sections using priority inheritance similar to PCP. Under partitioned scheduling, global critical sections are required to be non-preemptive. Further, nested critical sections are required to have group locks (separately for short and long sections), thus negating the benefits of nesting [32].

IV. SYSTEM AND TASK MODEL

4.1 System And Task Model

All the tasks are assumed to be periodic. The system knows about arrival time, period, required execution time and deadline of the task in priori. There are no precedence constraints on the task; they can run in any order relative to each other as long as their deadlines are met. A task is ready to execute as it arrives in the system.

The present work assumed that the system is not having resource contention problem. The task set is assumed to be preemptive. It also assumed that preemption and the scheduling algorithm incurs no overhead. In soft real-time systems, each task has a positive value.

4.1.1 Task model

The paper assumes that jobs are generated by periodic tasks and are scheduled on a multiprocessor platform comprised of m identical processors. A real-time system with shared resources is specified using *p* shared resources $R_1.....R_p$ and η periodic tasks $T = \{T_l......T_\eta\}$.Each periodic task $T_i(1 \le i \le \eta)$ is characterized as (T_i, C_i, D_i) where T_i denotes the minimum inter-arrival time, C_i the worst-case execution time, and D_i the relative deadline. Each job of task T_i requires C_i units of processing capacity within D_i time units from its release, and this processing capacity must be supplied sequentially, i.e., the job cannot be scheduled on more than one processor at any given time instant. Further, any two successive jobs of this task must be released at least T_i time units apart. This work consider constrained-deadline tasks, i.e., $D_i \le T_i$.

4.1.2 Job blocking

A job J of task τ_j is said to be directly blocked at time t on a request for resource R_k, if the three conditions below are true:
1) at time t, job J is one of the *m* highest effective priority jobs with remaining execution time;
2) at time t, resource R_k is locked by a job having lower base-priority than J
3) Job J made a request for resource R_k and this request has not been granted until time t.

Note that the above definition of blocking does not include the case when R_k is locked by a job having higher priority than J. This is consistent with the notion or blocking (that of being associated with priority-inversion) in the standard literature on uniprocessor systems. A job is said to be ready for execution whenever it is not directly blocked and not requesting a resource locked by a higher base- priority job.

Blocking time is defined as the time for which a low priority task can delay the execution of high priority task. Blocking can lead to tasks missing deadlines. Since blocking is an important phenomenon. The present work will explain the concept and methods to reduce and quantify Blocking times here. A lower priority task blocks the execution of high priority task. This phenomenon is called *priority inversion*.

➢ *Priority Inheritance (PIP)*

A *priority inheritance* mechanism is used to avoid priority inversion. In this algorithm a lower priority task inherits the priority of the highest priority task that gets blocked. The priority of lower task is increased once the higher priority task tries to lock the semaphore. The priority inheritance does solve problems of blocking to some extent but does not completely solve the problem of unpredictable delays. The priority inheritance mechanism cannot solve circular blocking which can lead to deadlocks.

To solve these problems there is a mechanism called the *priority ceiling protocol* (PCP). Each semaphore has an associated ceiling that it attains once it locks that semaphore. When the task releases the semaphore the priority is reverted to old one.

- *PIP for multiprocessors*

Under PIP, whenever a job *J* of task τ_i is directly blocked on a resource R_k the effective-priority of the job that is holding resource R_k (say *J* ' of task τ_j) is raised to i (priority inheritance). In addition to direct blocking, job *J* may also experience interference from other lower priority jobs under PIP. For instance, this can happen when the effective-priority of a lower priority job is raised above i because of priority inheritance. On uniprocessors, PIP ensures that *J* only experiences direct blocking (from job *J* ') and lower priority interference from carry-in jobs; jobs that are released before *J*'s release time. This is because any lower-base-priority job that is released after *J* cannot execute until *J* finishes its processing requirements. In other words, only those lower-base-priority jobs that hold a resource when *J* is released, can potentially interfere with, *J*'s executions.

4.1.3 Multiprocessor scheduling

From a different standpoint, scheduling algorithms can be classified as global or partitioned. Global algorithms use only one queue for all the tasks in the system, while in partitioned algorithms each processor has its own private scheduling queue.

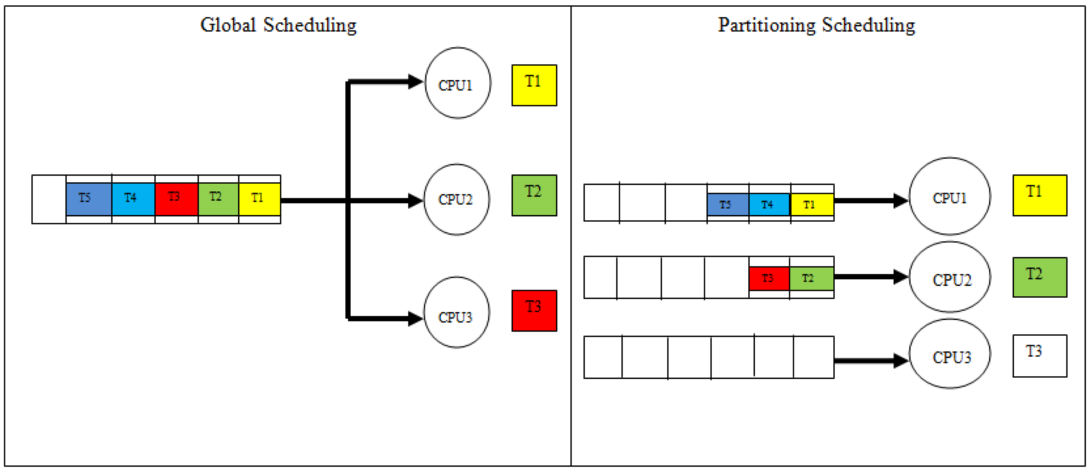


Figure1. Types of Microprocessor Scheduling

What is notable to say is that the proposed synchronization mechanism is independent from the specific characteristics of the scheduler, and works with dynamic priority, and under global approach. Therefore, in the remainder of the paper, it is assumed without loss of generality that the scheduling algorithm is global EDF. The present work considers global scheduling, i.e., a job is not assigned to any specific processor; instead jobs which have arrived but whose execution have not finished are stored in a system-wide ready queue shared between processors. This works focus on global dynamic priority preemptive scheduling. Every task has a priority assign based on EDF. We assume that priorities are unique and therefore we order tasks (with no loss of generality) such that for every pair or tasks $(\tau_i....\tau_\eta)$ it holds that is task τ_i has higher priority than task τ_j then $i < j$. Let the priority of a task be a positive integer such that a low number signifies a high priority; i.e., priority level 1 is the highest priority level and priority level η is the lowest priority level.

V. SYSTEM ARCHITECTURE

Multicore processors are today standard building blocks in embedded computer systems but their use for applications with real-time requirements is non-trivial. This is because although a comprehensive toolbox of scheduling theories are available for a computer with a single processor; such a comprehensive toolbox is currently not available for multicores. Real-time applications tend to be organized as a set of concurrently executing tasks which need to share resources (for example data structures or I/O devices). Clearly, a resource-sharing protocol is a crucial component in multicore-based embedded real-time systems.

When designing a resource access protocol for real-time applications, there are two important objectives: 1) at runtime, we must devise scheduling schemes and resource access protocols to reduce the *waiting-time* or *blocking-time* of a task; 2) off-line, we must be able to bound the waiting-time and include it in a schedulability analysis.

As processes enter the system, they are put into a *job queue*, which contains all the processes in the system. as *job scheduler*, select process from this pool and loads them into memory for execution. *Ready Queue* contains all processes residing in main memory and are ready and waiting to execute.

CPUScheduler, selects from among the processes (according to EDF) that are ready to execute and allocates the CPU to one of them.

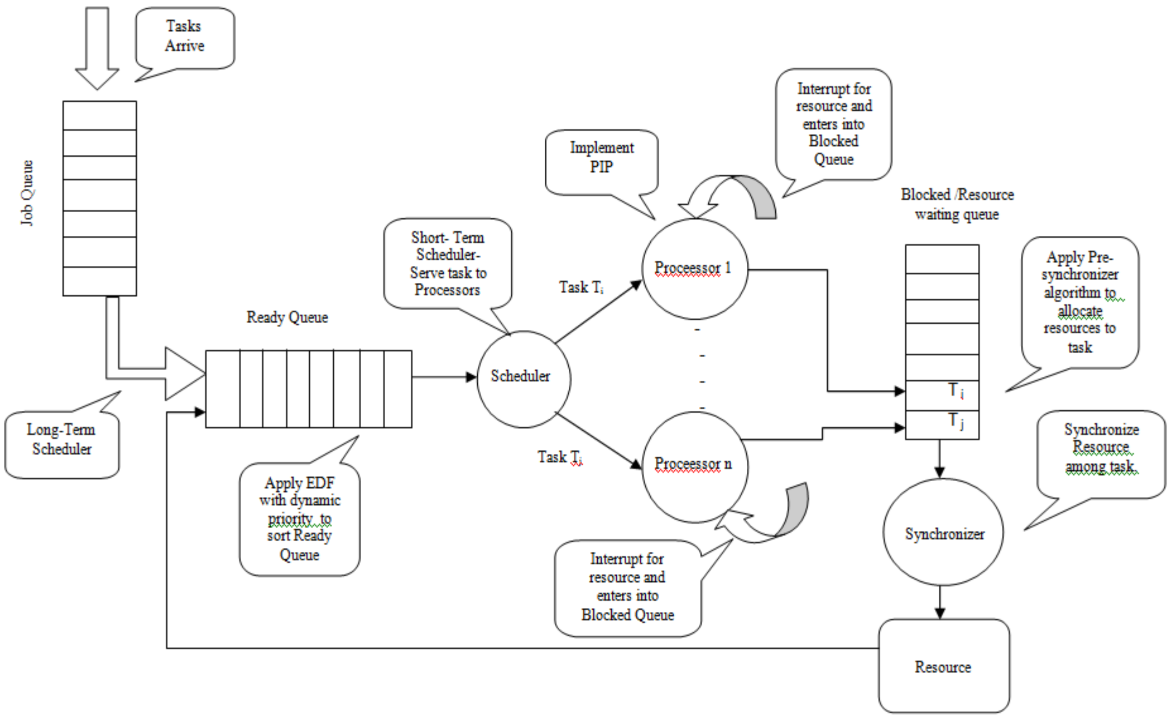

Tasks Arrive
Job Queue
Long-Term Scheduler
Ready Queue
Apply EDF with dynamic priority to sort Ready Queue
Short- Term Scheduler- Serve task to Processors
Scheduler
Implement PIP
Task T_i
Task T_j
Processor 1
Processor n
Interrupt for resource and enters into Blocked Queue
Interrupt for resource and enters into Blocked Queue
Blocked /Resource waiting queue
T_i
T_j
Apply Pre-synchronizer algorithm to allocate resources to task
Synchronize Resource among task
Synchronizer
Resource

Figure 2. System architecture

Suppose at the time of execution of a job, job *issue requests* for exclusive access to resources. If a request is not satisfied immediately, then the issuing job is said to be *blocked* and inserted into device queue. *Device queue* contains processes waiting for a particular I/O device.

The main goal of the above system architecture is to reduce the size of blocked queue. The present work present a resource sharing protocol based on this idea. It allows parallelism (granting requests) as much as possible, yet keeping the blocking time within limits.

5.1 Flow of system

Table 1. Task Model

Task Name	Arrival Time	Execution Time	Deadline	Dependency	Resource	Resource Reqd.
J1	**0**	**4**	**5**	**J3**	-------	---------
J2	**0**	**9**	**10**	--------	**R**	**6**
J3	**2**	**2**	**12**	--------	**R**	**1**

This section explains the Pre-synchronizer protocol using examples illustrated in Table 4.1. Let first assign priority to J1, J2 and J3 according to EDF. In this scenario job *J*1 has higher priority than job *J*2 and *J*2 has higher priority than job *J*3, and these jobs are scheduled on 2 identical processors and there is only one resource i. e. 'R'. Job J1 is dependent on job J3. Further, job *J*2 requests resource *R* and job *J*3 also requests resource *R*.

In Figure 3, both job *J*1 and J2 arrives and executing on processor P1 and processor P2, respectively. After some time job J3 arrives and waiting in *Ready Queue*. After some time interval, in between execution of job J1 requires job J3 and blocks. As soon as job J3 start execution on P1; it requests to lock resource *R*. At the same time job J2 also requests to lock resource R. This (Job J2) request is granted because J2 having higher priority and job J2 locks resource *R*. Then job *J*3 blocks because it needs *R*. It delays the execution of *J*3 and increase the blocking time of *J*1 as it is dependents on job J3**.**

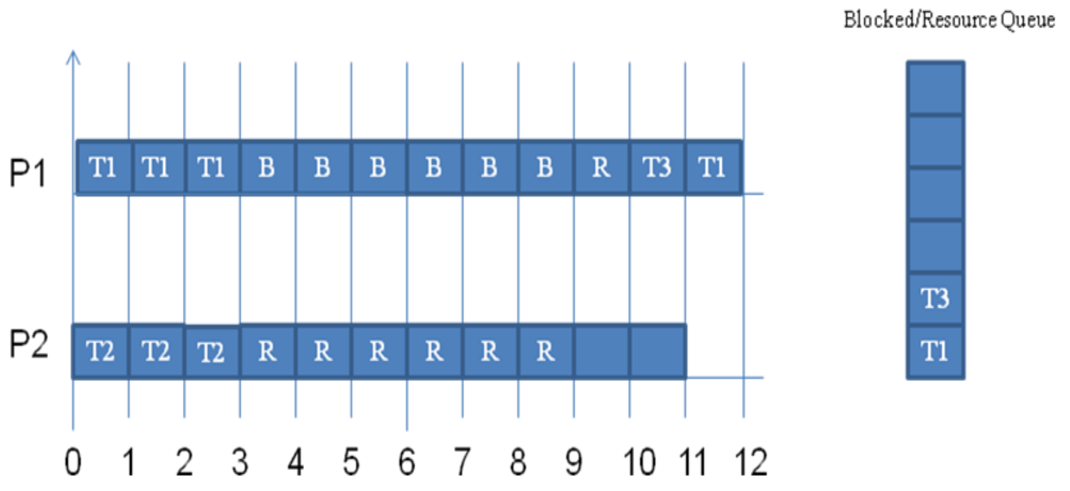


Figure 3. Global Scheduling without Pre-synchronizer protocol

Figure 4 shows Global Scheduling after applying pre-synchronizer protocol. Both job *J*1 and J2 arrives and executing on processor P1 and processor P2, respectively. After some time job J3 arrives and waiting in *Ready Queue*.

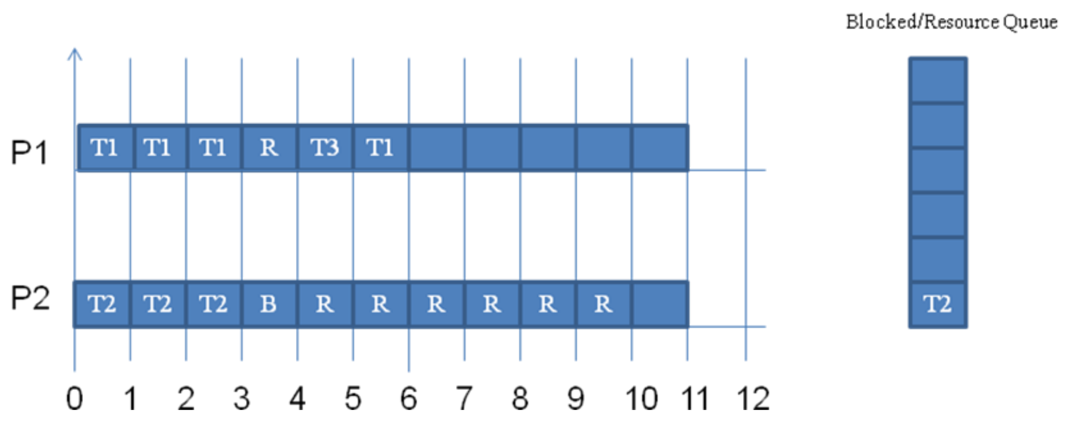


Figure 4. Global scheduling with Pre-synchronizer protocol

After some time interval job J1 requires job J3 and blocks. As soon as job J3 start execution on P1; it requests to lock resource *R*. At the same time job J2 also requests to lock resource R. This (Job J2) request is granted because J2 having higher priority and job J2 locks resource *R*. Then job *J*3 blocks because it needs *R*. It delays the execution of *J*3 and increase the blocking time of *J*1 as it is dependents on job J3**.** The pre-synchronizer protocol will allow *J*3 to lock *R a*s job J3 requires R for less time as compare to job J2 and does not increase the blocking time of *J*1. This scenario is depicted in the figure 4**.** In summary, when *J*3 requests resource *R*, it is granted access. So that it does not unnecessarily increasing the blocking of *J*1.

The adjustment of blocking parameters in Lines 24 and 28 of Algorithm 2 ensures this property. *The Pre-synchronizer protocol thus allows lower priority jobs to lock resource and thereby improves parallelism when compared to other existing approaches.*

The latter is true because of the following reasons. The protocol allows a lower priority job to lock a resource. *This constraint checks, for each higher priority task* τ_{κ}*, whether the maximum blocking time* $CS_{i,j}$ *is smaller than permitted blocking time of resource* R_j $\left(\lfloor R_k \rfloor_j\right)$.

VI. PROPOSED WORK

This works aims at the Synchronization in Embedded Real-Time operating systems. Real-time systems have a finite set of resources, and hence, have a finite processing capacity. A RTOS is a multitasking operating system designed to meet strict deadlines (Real-Time). Many embedded systems require software to respond to inputs and events within a defined short period. RTOS is designed to control an embedded system and deliver the real-time responsiveness and determinism required by the controlled device. Applications run under the control of the RTOS, which schedules allocated CPU time.

6.1 Understanding the problem

This section introduce about problem of priority-inversion and the idea of how this is solved in the context of uniprocessor scheduling. The next section then discuss that transferring this idea to multiprocessor scheduling can cause a limitation in efficient use of available processing capacity (platform parallelism).

Uniprocessor systems

Suppose task set *T* is scheduled on a single processor using dynamic-priority scheduling. It is assumed that the priority of a job is assigned according to EDF and is not affected by whether the job is holding a resource or not. Table 1 shows an example of three jobs where *J*2 and *J*3 request some shared resource *R* and job *J*1 never requests this resource. It is assumed that *J*1 has higher priority than *J*2 and *J*2 has higher priority than *J*3.

First *J*1 and *J*2 are released. *J*1 executes and then it requests job *J*3. Then *J*1 is blocked because of *J*3. Then *J*2 starts its execution and it request resource *R*. *J*2 is granted the resource *R* and it continues executing holding the resource. Before it releases *R* however, in the meantime, *J*3 is released for execution and it is scheduled by the dispatcher. As soon as *J*3 starts, it requests resource *R*. This request is denied since *R* is held by job *J*2, *i.e.*, job *J*3 is blocked and cannot execute further. The job *J*2 executes for a long time and during its execution the deadline of job *J*1 expires.

In the above example, even when a higher priority job *J*1 is blocked by a lower priority job *J*3 (on a shared resource *R* held by *J*2), a medium priority job *J*2 is allowed to execute and eventually delay the execution of job *J*1. Although it is inevitable that *J*1 must block until *J*3 releases resource *R*, *J*1 must not be required to wait for job *J*2 to finish executing because *J*2 is not holding any resource required by *J*1.

The research community has invented protocols to reduce this effect (priority-inversion), and those protocols give jobs that hold shared resources temporarily a higher priority. Figure 3 shows the same jobs. But now the priority of job *J*3 is promoted when it holds resource *R*. In this way, we can see that job *J*1 will meet its deadline, because job *J*2 is not allowed to execute in between.

There are different ways to promote the priority of a job that holds a resource; (i) the job could be scheduled non-preemptively or (ii) the job could be assigned the ceiling priority of the resource or (iii) the job could inherit (transitively) the maximum priority among all jobs that are presently blocked on the same resource.

The latter approach can be combined with a test that is performed whenever any job requests a shared resource; a lower priority task inherits the priority of the highest priority task that gets blocked. The priority of lower task is increased once the higher priority task tries to lock the semaphore.

Multiprocessor systems

Suppose we have the same scenario as in Table 1, but now the jobs are scheduled on a multiprocessor platform comprised of 2 processors. Then, it is possible to schedule job *J*2 without having to preempt the execution of job *J*3, and therefore *J*2 will not interfere with the execution of job *J*1. This brings us to the question, "Is it okay to schedule a medium priority job as long as it does not preempt any resource holding lower priority job?" Although the answer seems positive from the previous example, this is not true in all cases.

Let us consider three jobs *J*1, *J*2 and *J*3 as in Table 1 shows an example of three jobs where *J*2 and *J*3 request some shared resource *R* and job *J*1 never requests this resource. It is assumed that *J*1 has higher priority than *J*2 and *J*2 has higher priority than *J*3.

First *J*1 and *J*2 are released. *J*1 executes and then it requests job *J*3. Then *J*1 is blocked because of *J*3. Then *J*2 starts its execution and it request resource *R*. *J*2 is granted the resource *R* and it continues executing holding the resource. Before it releases *R* however, in the meantime, *J*3 is released for execution and it is scheduled by the dispatcher. As soon as *J*3 starts, it requests resource *R*. This request is denied since *R* is held by job *J*2, *i.e.*, job *J*3 is blocked and cannot execute further. The job *J*2 executes for a long time and during its execution the deadline of job *J*1 expires.

In the above example, even when a higher priority job *J*1 is blocked on lower priority job *J*3, a medium priority job *J*2 is allowed to execute and eventually delay the execution of job *J*1. Although it is inevitable that *J*1 must block until *J*3 executes, *J*1 must not be required to wait for job *J*2 to finish executing because *J*1 is not holding any resource required by *J*2.

An improved resource sharing scenario in which job *J*2 is denied access to resource *R*2 is illustrated in Figure 4. This then begs the question, "When should a request for shared resource be granted?".A very safe approach would be to grant access to only one resource at a time, but this would limit parallelism. And this limited parallelism would imply that more work must be done at later times, which in turn can cause deadline misses. Another approach would be to use a PCP-like protocol (as in uniprocessors) and decide that job *J*2 should be denied resource $R2$, because $J2$ does not have higher priority than the ceilings of all locked resources (namely $R1$). But this can also unnecessarily limit parallelism resulting in the aforementioned performance drawback.

If job $J2$ would have released resource $R2$ just prior to when job $J1$ requested access to the same resource, then it would not have affected the finishing time of job $J1$. Thus we can see that a resource request from a medium priority job can be granted if the resource is released before any other higher priority job requests it. Or more generally, a resource request can be granted as long as the maximum blocking time suffered by any higher priority job is guaranteed to be within pre-defined bounds. The next section present, pre-synchronizer, a resource sharing protocol based on this idea. It allows parallelism (granting requests) as much as possible, yet keeping the blocking time within limits.

6.2 Pre-synchronizer resource sharing protocol

The main idea:

From the discussion in the previous section we can draw the following conclusions about the design of a protocol which avoids priority inversion and allows a large degree of parallel execution:

• A priority-inheritance mechanism should be used in order to avoid priority inversion but a PCP-like mechanism should not be used (because it would restrict parallel execution too much).
• If a high-priority task requests to execute but it does not request a shared resource then this task should be allowed to continue to execute.
• There should be a mechanism for preventing deadlock. (This is needed since we do not use PCP.)
• For each task-resource pair, there should be an associated counter variable. This counter specifies the amount of blocking that the task can tolerate to be blocked when requesting the resource. For every resource request, the protocol should check so that granting the request does not violate any tolerated blocking of any other task-resource pairs.
• If a task is blocked for one time unit because the task requested a resource then the corresponding counter of this task-resource pair should be decremented by one (since the amount of tolerable blocking is one time unit less). In this section, the paper present a protocol based on these ideas and this protocol will avoid priority inversion and allow a large degree of parallel execution.

Firstly, notations are present that is needed. Then present the algorithm for global scheduling; it uses the counters as mentioned above. The paper will then show how the counters as updated and discuss subtle issues with the protocol.

Notations: We use the following notations.

• t : Denotes the current time instant.
• LPB_i : Denotes the *L*ower *P*riority *B*locking for jobs of task τ_i. The present protocol guarantees that for each resource access by jobs of task τ_i, the maximum time for which this job will be blocked by lower priority jobs is at most LPB_i. The present protocol guarantees a value of $\max\{CS_{k,l}\}$ for LPB_i, where $k > i$ and l ranges over resource that job of τ_i access.
• $MTR_{k,l}$: *M*inimum *T*ime to *R*equest resource R_l. For example, suppose jobs of τ_k request resource R_l in three nesting during their execution; 1) R_j requested and then R_l with a minimum gap of 10 time units, 2) R_l alone requested, and 3) R_j requested and then R_l with a minimum gap of 5 time units. Then, $MTR_{k,l}$ in this case is 5.
• $\lfloor R_l \rfloor_k$:For each task τ_k and each resource R_l, $\lfloor R_l \rfloor_\kappa$ denotes the maximum blocking (in future) that the currently active job of τ_k can incur in its current resource nesting. If the job is currently not in any nesting or if

$\lfloor R_l \rfloor_\kappa$ is currently irrelevant, then it is set to ∞. The value is initialized to ∞, and only updated by the present protocol.

• PTY_i: Priority of jobs of task τ_i at the current time instant. PTY_i is initialized to i, but can be temporarily modified by pre-synchronizer protocol (PIP-like updates). A job of task τ_i has higher priority than a job of task τ_j if $PTY_i < PTY_j$, or $PTY_i = PTY_j$ and $i < j$.

6.2.1 Pre-synchronizer Protocol

The **Pre-synchronizer** resource sharing protocol is given by Algorithm 1 and the update to $\lfloor R_l \rfloor_\kappa$ is performed by Algorithm 2.

Both these algorithms are executed at each time instant *t*, with Algorithm 2 being executed first. The present paper previously explained the **Pre-synchronizer** protocol using examples illustrated in Figure 4.1

Algorithm 1 Global scheduling with resource sharing

```
1: $n\_assigned \longleftarrow 0$
2: for each ready job $J$ in priority order (based on $PTY_i$) do
3:     if $n\_assigned < m$ then
4:         if $J$ is not requesting any resource then
5:             Execute job $J$
6:             $n\_assigned \leftarrow n\_assigned + 1$
7:         else
8:             Let $R_j$ denote the resource requested.
9:             if all resources in the nesting to which
10:                this request belongs are unlocked then
11:                if $\forall\kappa : \lfloor R_J \rfloor_\kappa \geq CS_{i,j}$ or
12:                    $PTY_i < PTY_k$ then
13:                    Execute $J$ and set $PTY_i$ equal to the
14:                    smallest $PTY_k$ such that $\lfloor R_J \rfloor_\kappa$
15:                    has a finite value (this update to $PTY_i$
16:                    is reset when resource $R_j$ is released).
17:                    $n\_assigned \leftarrow n\_assigned + 1$
18:                end if
19:            end if
20:        end if
21:    end if
22: end for
```

Algorithm 2 Update rules for $\lfloor R_l \rfloor_\kappa$

```
1: if A job of task $\tau_i$ performs an *outermost* request for
2:     resource $R_j$ (first request of a nested access) then
3:     $\lfloor R_j \rfloor_i \leftarrow LPB_i$
4:     if $R_j$ is currently locked by a job of task $\tau_k$ and
```

```
5:                 $PTY_k > PTY_i$ **then**
6:                 $PTY_k \leftarrow PTY_i$ (this update to $PTY_k$ is reset
7:                 when resource $R_j$ is released).
8:         **end if**
9:         $\lfloor R_l \rfloor_i \leftarrow MTR_{i,l}$ for each non-outermost resource $R_l$
10:        in this nested access.
11:        **for** each non-outermost resource $R_l$ in this
12:                nested access **do**
13:                **if** $R_l$ is currently locked by a job of task $\tau_k$ and
14:                        $PTY_k > PTY_i$ **then**
15:                                $PTY_k \leftarrow PTY_i$ (this update to $PTY_k$ is reset
16:                                when resource $R_l$ is released).
17:                **end if**
18:        **end for**
19: **end if**
20:  **if** a job of task $\tau_i$ is granted access to a resource $R_j$ (in response
         to an earlier request) **then**
21:                $\lfloor R_j \rfloor_i \leftarrow \infty$
22: **end if**
23: **if** A job of task $\tau_i$ is blocked in the interval $(t-1, t]$ **then**
24:        $\lfloor R_l \rfloor_i \leftarrow \lfloor R_l \rfloor_i - 1$, for all $l.s.t. \lfloor R_l \rfloor_i \neq \infty$
25: **else**
26:        **if** A job of task $\tau_j$ is directly blocking some job in the
27:                interval $(t-1, t]$ **then**
28:        $\lfloor R_l \rfloor_i \leftarrow \lfloor R_l \rfloor_i - 1$, for all $l.s.t. \lfloor R_l \rfloor_i \neq \infty$
29:        **end if**
30: **end if**
```

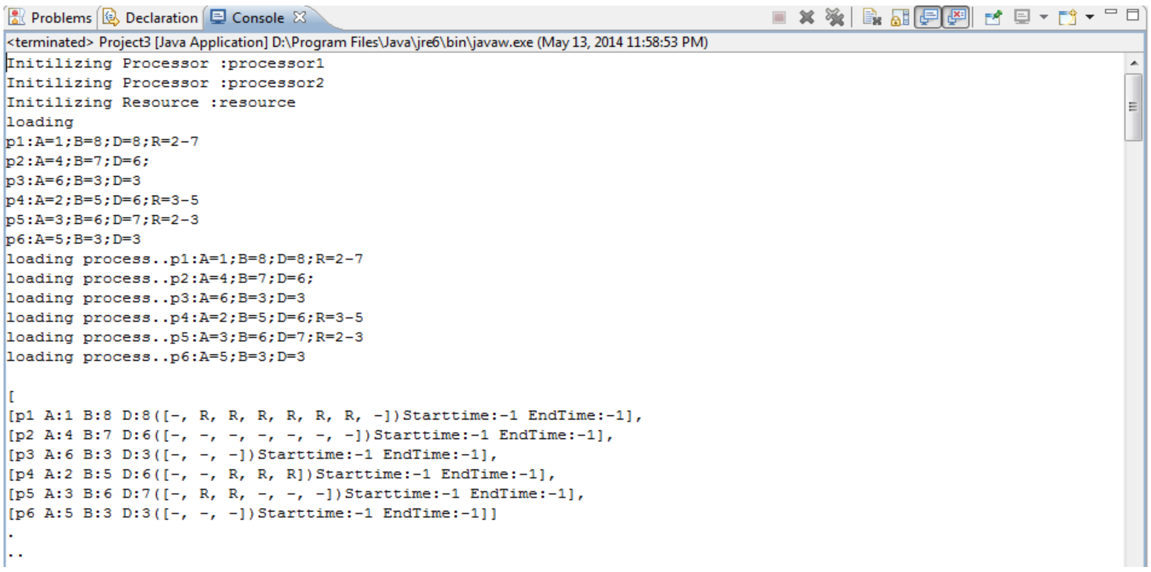
The Pre-synchronizer protocol thus allows lower priority jobs to lock resources even when dependent resources are locked by higher priority jobs, and thereby improves parallelism when compared to other existing approaches. It can be shown that the Pre-synchronizer protocol prevents deadlocks (due to check in Line 9 of Algorithm 1), and ensures that the maximum lower priority blocking suffered by any job of task τ_i is LPB_i. The latter is true because of the following reasons.

• The protocol allows a lower priority job to lock a resource if it does not violate the blocking constraint in Line 11 of Algorithm 1. This constraint checks, for each higher priority task τ_k, whether the maximum blocking time $CS_{i,j}$ is smaller than permitted blocking time of resource R_j $\left(\lfloor R_k \rfloor_j\right)$.

VII. EXPERIMENTAL RESULTS

7.1 Results

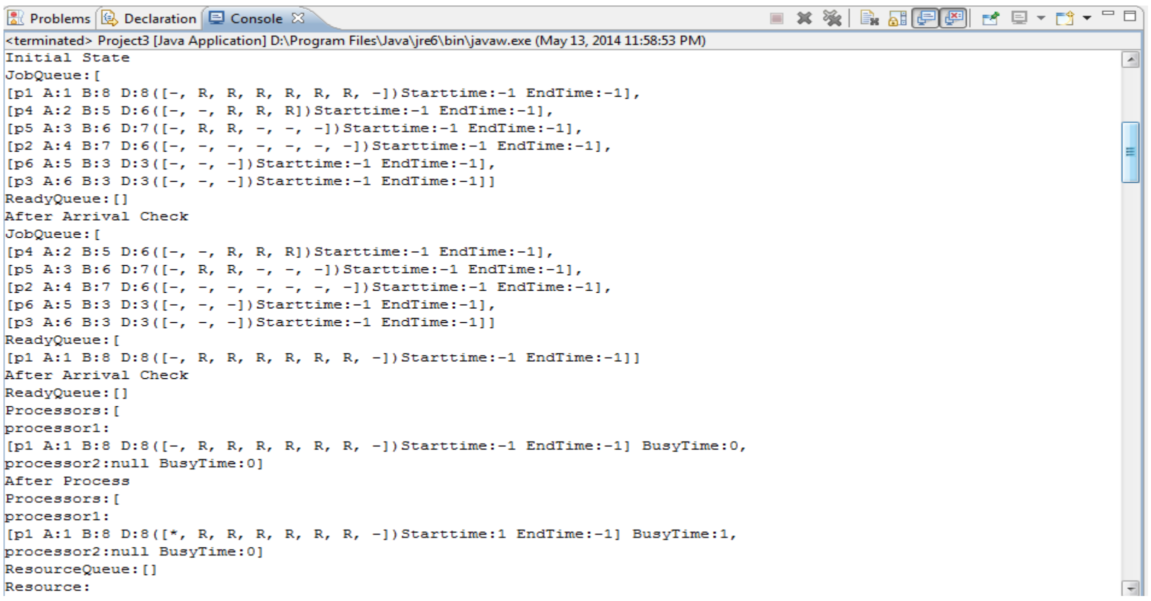
Figure 5 shows initialization of the tasks with different parameters like A i.e. Arrival Time, B i.e. Burst Time, D i.e. Deadline and R i.e. Resource required in particular clock cycle. For ex. In above figure, Processor1 and Processor2 are two processors and processes initialized with p4: A=2, B=5,D=6 and R=3-5 means Process 4 arrives at 2 clock cycle, 5 is burst time for process 4, 6 is deadline(Relative) for process. And R indicates it require resource from 3 clock cycle to 5 clock cycle.



```
<terminated> Project3 [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (May 13, 2014 11:58:53 PM)
Initilizing Processor :processor1
Initilizing Processor :processor2
Initilizing Resource :resource
loading
p1:A=1;B=8;D=8;R=2-7
p2:A=4;B=7;D=6;
p3:A=6;B=3;D=3
p4:A=2;B=5;D=6;R=3-5
p5:A=3;B=6;D=7;R=2-3
p6:A=5;B=3;D=3
loading process..p1:A=1;B=8;D=8;R=2-7
loading process..p2:A=4;B=7;D=6;
loading process..p3:A=6;B=3;D=3
loading process..p4:A=2;B=5;D=6;R=3-5
loading process..p5:A=3;B=6;D=7;R=2-3
loading process..p6:A=5;B=3;D=3

[
[p1 A:1 B:8 D:8([-, R, R, R, R, R, R, -])Starttime:-1 EndTime:-1],
[p2 A:4 B:7 D:6([-, -, -, -, -, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1],
[p5 A:3 B:6 D:7([-, R, R, -, -, -])Starttime:-1 EndTime:-1],
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
.
..
```

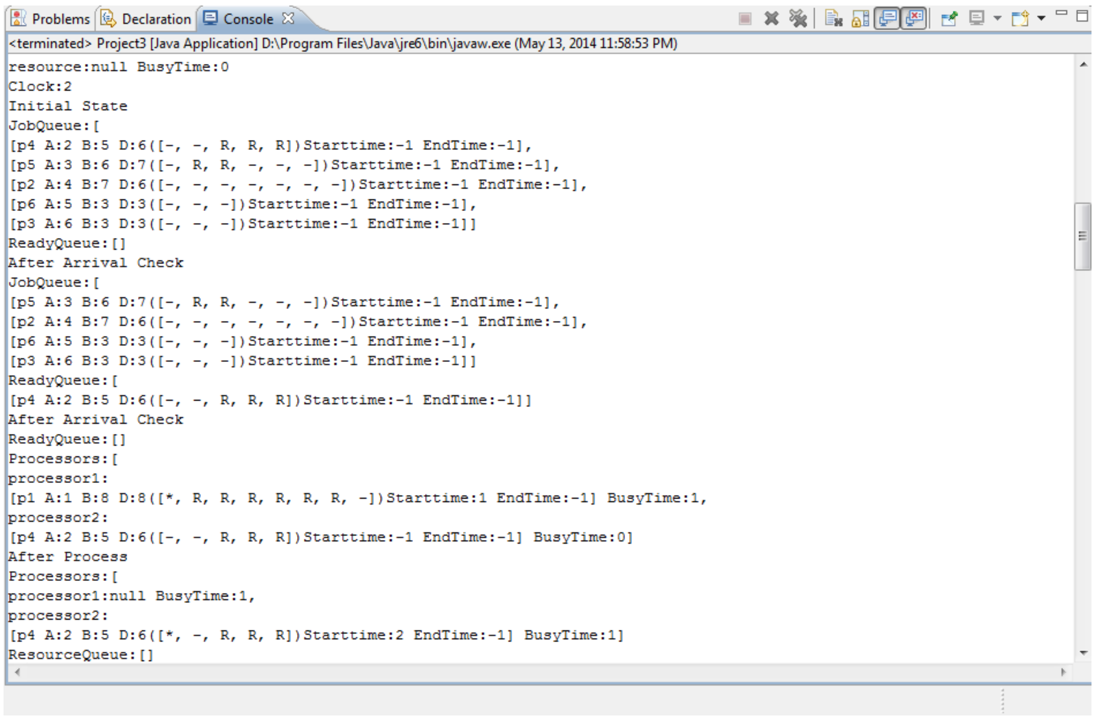
Figure 5. Process Initialization and Processor



```
<terminated> Project3 [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (May 13, 2014 11:58:53 PM)
Initial State
JobQueue:[
[p1 A:1 B:8 D:8([-, R, R, R, R, R, R, -])Starttime:-1 EndTime:-1],
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1],
[p5 A:3 B:6 D:7([-, R, R, -, -, -])Starttime:-1 EndTime:-1],
[p2 A:4 B:7 D:6([-, -, -, -, -, -, -])Starttime:-1 EndTime:-1],
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[]
After Arrival Check
JobQueue:[
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1],
[p5 A:3 B:6 D:7([-, R, R, -, -, -])Starttime:-1 EndTime:-1],
[p2 A:4 B:7 D:6([-, -, -, -, -, -, -])Starttime:-1 EndTime:-1],
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[
[p1 A:1 B:8 D:8([-, R, R, R, R, R, R, -])Starttime:-1 EndTime:-1]]
After Arrival Check
ReadyQueue:[]
Processors:[
processor1:
[p1 A:1 B:8 D:8([-, R, R, R, R, R, R, -])Starttime:-1 EndTime:-1] BusyTime:0,
processor2:null BusyTime:0]
After Process
Processors:[
processor1:
[p1 A:1 B:8 D:8([*, R, R, R, R, R, R, -])Starttime:1 EndTime:-1] BusyTime:1,
processor2:null BusyTime:0]
ResourceQueue:[]
Resource:
```

Figure 6. Allocation of Processes

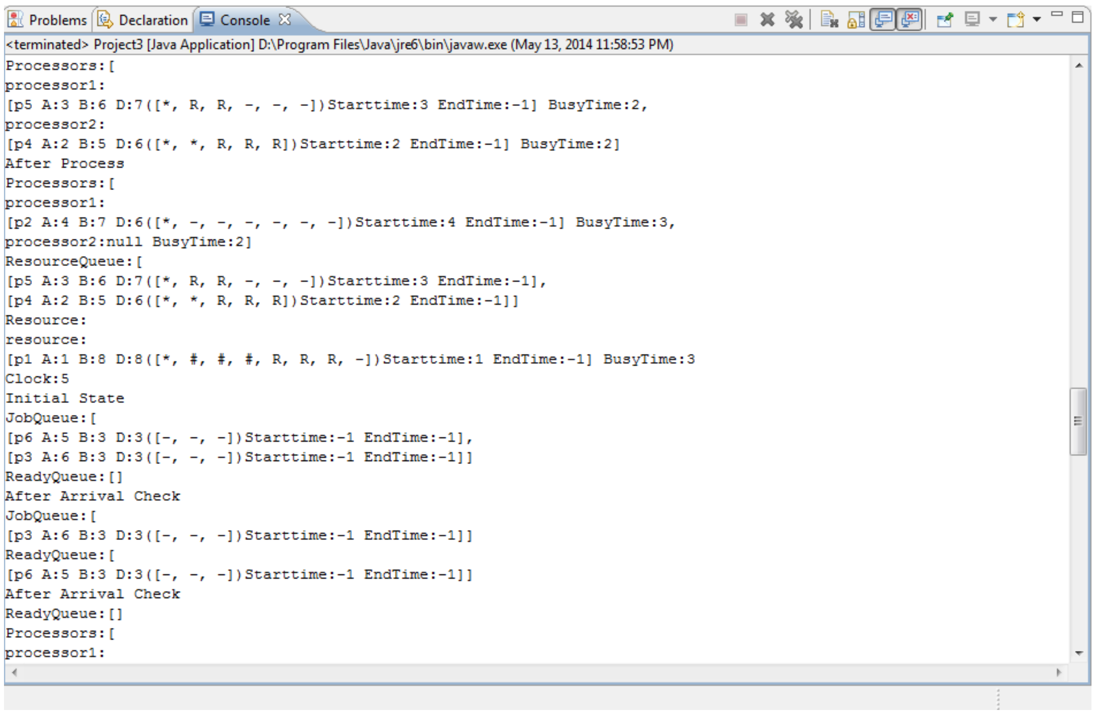
Figure 6 shows simulation for allocation of processes to *Job Queue* and then *Long Term Schedular* allocates that processes from job queue to R*eady Queue*. Then *CPUSchedular* sort ready queue according to EDF and allocate processor to processes for execution.



```
<terminated> Project3 [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (May 13, 2014 11:58:53 PM)
resource:null BusyTime:0
Clock:2
Initial State
JobQueue:[
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1],
[p5 A:3 B:6 D:7([-, R, R, -, -, -])Starttime:-1 EndTime:-1],
[p2 A:4 B:7 D:6([-, -, -, -, -, -, -])Starttime:-1 EndTime:-1],
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[]
After Arrival Check
JobQueue:[
[p5 A:3 B:6 D:7([-, R, R, -, -, -])Starttime:-1 EndTime:-1],
[p2 A:4 B:7 D:6([-, -, -, -, -, -, -])Starttime:-1 EndTime:-1],
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1]]
After Arrival Check
ReadyQueue:[]
Processors:[
processor1:
[p1 A:1 B:8 D:8([*, R, R, R, R, R, R, -])Starttime:1 EndTime:-1] BusyTime:1,
processor2:
[p4 A:2 B:5 D:6([-, -, R, R, R])Starttime:-1 EndTime:-1] BusyTime:0]
After Process
Processors:[
processor1:null BusyTime:1,
processor2:
[p4 A:2 B:5 D:6([*, -, R, R, R])Starttime:2 EndTime:-1] BusyTime:1]
ResourceQueue:[]
```

Figure 7. Execution of task on multiprocessor

In Figure 7, '*' indicates execution of process and 'R' indicates resource required in particular clock cycle.



```
<terminated> Project3 [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (May 13, 2014 11:58:53 PM)
Processors:[
processor1:
[p5 A:3 B:6 D:7([*, R, R, -, -, -])Starttime:3 EndTime:-1] BusyTime:2,
processor2:
[p4 A:2 B:5 D:6([*, *, R, R, R])Starttime:2 EndTime:-1] BusyTime:2]
After Process
Processors:[
processor1:
[p2 A:4 B:7 D:6([*, -, -, -, -, -, -])Starttime:4 EndTime:-1] BusyTime:3,
processor2:null BusyTime:2]
ResourceQueue:[
[p5 A:3 B:6 D:7([*, R, R, -, -, -])Starttime:3 EndTime:-1],
[p4 A:2 B:5 D:6([*, *, R, R, R])Starttime:2 EndTime:-1]]
Resource:
resource:
[p1 A:1 B:8 D:8([*, #, #, #, R, R, R, -])Starttime:1 EndTime:-1] BusyTime:3
Clock:5
Initial State
JobQueue:[
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1],
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[]
After Arrival Check
JobQueue:[
[p3 A:6 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
ReadyQueue:[
[p6 A:5 B:3 D:3([-, -, -])Starttime:-1 EndTime:-1]]
After Arrival Check
ReadyQueue:[]
Processors:[
processor1:
```

Figure 8. Allocation of Resource

In Figure 8, shows allocation of resource to process which is indicated using '#'. 'Starttime' indicates starting time of execution of particular process on processor. And 'Endtime' indicates end of execution of process. Firstly, process from ready queue is allocated to processor which is free (Bydefault, at first time Processor 1).

If Processor 1 is busy then next process allocate to Processor 2. The present work assumed scheduling is *preemptive* .So in between execution of current process any high priority (earliest deadline) process arrives; initial process gets preempted by newly arrived process and adds to ready queue. In between execution, if process requires resource then it adds to *Resource Queue* and resource get allocate resource according to *Pre-synchronizer*.

VIII. CONCLUSION

The paper has taken an initial step towards developing a generic resource-sharing framework for periodic real-time tasks scheduled on a multiprocessor under global EDF. The present work has discussed that there is a tradeoff between blocking and parallelism, and work has proposed the Pre-synchronizer protocol which allows as much parallelism as possible, keeping blocking within limits.

We may note that we are not the first ones to propose that a request for a resource should undergo a check, to calculate the time when the resource will be released. In fact, SIRAP [1] (a protocol for hierarchical scheduling) used such a test to decide if a job which requests a resource will finish execution before its budget expires, and if the answer is no then the request is denied.

Although the Pre-synchronizer protocol addressed some issues concerning the efficient use of parallelism in task executions, some open questions still remain.

- "Moving from uniprocessors to multiprocessors, whether it is still relevant to treat processors in a special manner when compared to other shared resources?".

In multiprocessors, there is a very clear trade-off between mutually exclusive access to shared resources and ability to exploit processing parallelism. Then, it would be interesting to consider processors as just another shared resource (although preemptable), and integrate their scheduling directly into the resource sharing protocol.

- "How to integrate the loss of parallelism due to shared resources in schedulability analysis?"

There are two factors leading to loss of parallelism; blocking from lower priority jobs and blocking from higher priority jobs. Higher priority blocking arises when processors are idle because higher priority jobs have locked resources required by lower priority jobs. The former is accounted for in the blocking factor (*LPB* in our case). However, accounting for the latter is still an open problem.

IX. FUTURE SCOPE

In the future we plan to work further on the resource management issues on multi-core platforms and we will investigate the possibility of improvement of the existing protocols as well as development of new approaches. One future work will be to extend our global algorithm to other synchronization protocols, e.g, Multiple Stack Resource Protocol (MSRP), Flexible Multiprocessor Locking Protocol (FMLP) under partitioned scheduling.

This work has focused on resource management on multi-cores where resources are protected by semaphores. In a fault-tolerant system, applications have to be protected from other applications that may malfunction. If the applications are allowed to access shared memory, a malfunctioning application may corrupt parts of the memory that is also shared by other applications. To avoid this, the applications are isolated such that each of them can only access its dedicated portion of memory. However, in this case using resource sharing protocols that rely on shared memory (semaphores) is not feasible. In the future, we aim to work on resource management among real-time applications on multi-cores by means of message passing.

REFERENCES

[1] A. Shah and K. Kotecha, “Scheduling Algorithm for Real-Time Operating Systems Using ACO”, proceedings of the International Conference on Computational Intelligence and Communication Networks (CICN), pp.617-621,2010.

[2] A. Srinivasan and J. Anderson, “Efficient Scheduling of Soft Real-Time Applications on Multiprocessors”, Journal of Embedded Computing, Volume 1, Number 2, pp. 285-302, 2005.

[3] Baker, T, “What to make of multicore processors for reliable real-time systems?” In Proceedings of the 15th Ada-Europe International Conference on Reliable Software Technologies, LNCS 6106, pages 1–18, 2010.

[4] B. Brandenburg and J. Anderson, “Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks”, Proceedings of the ACM International Conference on Embedded Software, pp. 69-78, October 2011. Winner, best paper award, October 2011..

[5] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson, “Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?”, Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 342-353, April 2008..

[6] B. Ward, J. Herman, C. Kenna, and J. Anderson, “Making Shared Caches More Predictable on Multicore Platforms”, Proceedings of the 25th Euromicro Conference on Real-Time Systems, pp. 157-167. Winner, outstanding paper award, July 2013.

[7] Easwaran, A. and Andersson, B.,“Resource sharing in global fixed-priority preemptive multiprocessor scheduling”. In Proceedings of the 30th IEEE Real-Time Systems Symposium, pages 377–386, 2009..

[8] Guan, N., Stigge, M., Yi, W., and Yu, G, “Cache-aware scheduling and analysis for multicores”. In Proceedings of the 7th ACM International Conference on Embedded Software, pages 245– 254, 2009.

[9] Hu Wei and C. Tianzhou, “Embedded hard real-time scheduling algorithm based on task’s resource requirement”, Proceedings of Int. J. High Performance Computing and Networking, Vol. 6, Nos. ¾, 2010..

[10] H. Cho, Binoy Ravindran and E. Douglas Jensen, “Synchronization for an optimal real-time scheduling algorithm on multiprocessors” [Online] Available.(Accessed: Aug 6,2013)

[11] J. Anderson, V. Bud, and, U. Devi, “An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems”, Proceedings of the 17th Euromicro Conference on Real-Time Systems, pp. 199-208, July 2005.

[12] J. Carpenter, Shelly Funk, Philip Holman, A. Srinivasan and J. Anderson, Sanjoy Baruah, “A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms” [Online] Available. (Accessed: Aug 7,2013)

[13] Jeffay, K. “Scheduling sporadic tasks with shared resources in hard-real-time systems”. In Proceedings of the 13th IEEE Real-Time Systems Symposium, pages 89–99, 1992.

[14] M. Fan and Q. Gang, “Harmonic-Aware Multi-Core Scheduling For Fixed-Priority Real-Time Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol: pp, Issue:99, 2013.

[15] M. Kaladevi and Dr.S.Sathiyabama, “A Comparative Study of Scheduling Algorithms for Real Time Task”, Proceeding of International Journal of Advances in Science and Technology, Vol. 1, No.4, 2010.

[16] S. Khushu and J. Simmons, “Scheduling and Synchronization in Embedded Real-Time Operating Systems”, March 2001, [Online] Available. (Accessed: Aug 7,2013)

[17] Xie Bin,Yan Like and C. Tianzhou, “Real-Time Scheduling Algorithm for Embedded Systems with various Resource Requirement” [Online] Available (Accessed: Aug 8,2013)

[18] V. Salmani, S.Taghavi Zargar, and M. Naghibzadeh, “A Modified Maximum Urgency First Scheduling Algorithm for Real-Time Tasks”, World Academy of Science, Engineering and Technology 9, 2005.

[19] A.Habibi and V. Salmani, “Quantitative Comparison of Job-level Dynamic Scheduling Policies in Parallel Real-time Systems” [Online] Available (Accessed: Aug 4,2014).

[20] B. Andersson, S. Baruah, and J. Jonsson, “Static-Priority Scheduling on Multiprocessors”. In Proc. IEEE Real-Time Systems Symposium (RTSS), Dec 2001.

[21] B. Andersson, “Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%”. In Proc. ACM International Conference on Principles of Distributed Systems (OPODIS), volume 5401, pages 73–88, 2008.

[22] S. Kato and N. Yamasaki., “Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors”. In IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr. 2009.

[23] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, “Partitioned Fixed- Priority Preemptive Scheduling for Multi-core Processors”. In Euromicro Conference on Real-Time Systems (ECRTS), Jul. 2009.

[24] Sudarshan K. Dhall and C. L. Liu., “On a Real-Time Scheduling Problem”. Operations Research, 26(1):127–140, 1978.

[25] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, “A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms”. In Handbook on Scheduling Algorithms, Methods, and Models. Chapman Hall/CRC, Boca, 2004.

[26] M. Herlihy and I. Wing, “Linearizability: a correctness condition for concurrent objects”. ACM Transactions on Prograrnrning Languages and Svstems, J 2(3):463-492, 1990.

[27] T. P. Baker, “Stack-based scheduling for realtime processes”. Journal of Real-Time Systems. 3(1):67-99, 1991.

[28] L. Sha, R. Rajkumar, and P. Lehoczky, “ Priority inheritance protocols: An approach to real-time synchronization”. IEEE Transactions on Computers, 39(9): 1175-1185, 1990.
[29] T. P. Baker, “Stack-based scheduling for realtime processes”. Journal of Real-Time Systems, 3(1):67–99, 1991.
[30] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “ Proportionate progress: a notion of fairness in resource allocation”. Algorithmica, 15(6):600–625, 1996.
[31] M. Behnam, I. Shin, T. Nolte, and M. Nolin, “ SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems”. pages 279–288, 2007.
[32] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, “ A flexible real-time locking protocol for multiprocessors”. In Proc. of Real-time and Embedded Computing Systems and Applications Conference, pages 47–56, 2007.
[33] C.-M. Chen and S. K. Tripathi, “ Multiprocessor priority ceiling based protocols. Technical report”, 1994.
[34] Um. C. Devi, H. Leontyev, and J. H. Anderson, “ Efficient synchronization under global edf scheduling on multiprocessors”, In Proc. of Euromicro Conference on Real-Time Systems, pages 75–84, 2006.
[35] E. W. Dijkstra, “Co-operating sequential processes”. Programming Languages, pages 43–112, 1968.
[36] P. Gai, G. Lipari, and M. Di Natale, “Minimizing memory utilization of real-time task sets in single and multiprocessor systems-on-a-chip”, In Proc. of IEEE Real-Time Systems Symposium, page 73, 2001.
[37] P. Holman and J. H. Anderson, “Locking in pfairscheduled multiprocessor systems”, In Proc. of IEEE Real-Time Systems Symposium, page 149, 2002.
[38] J. M. L´opez, J. L. D´ıaz, and F. D. Garc´ıa.,“ Utilization bounds for EDF scheduling on real-time multiprocessor systems”, Journal of Real-Time Systems, 28(1):39–68, 2004.
[39] A.K. Mok., “ Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.”, PhD Thesis, Department of Computer Science, Massachusetts Institute of Technology (MIT), 1983.
[40] R. Rajkumar, “Synchronization in Real-Time Systems: A Priority Inheritance Approach”. Kluwer Academic Publishers,1991.
[41] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-time synchronization protocols for multiprocessors”. pp 259–269,1988.
[42] L. Sha, R. Rajkumar, and J. P. Lehoczky, “ Priority inheritance protocols: An approach to real-time synchronization”, IEEE Transactions on Computers, 39(9):1175– 1185,1990.
[43] D. B. Stewart, and P. k. Khosla, August, “Real-Time Scheduling of Dynamically Reconfigurable Systems,” in Proc. IEEE International Conference on Systems Engineering, Dayton Ohio, pp. 139-142, 1991.