Technical Report No. 2005-499

Scheduling Algorithms for Real-Time Systems*

Arezou Mohammadi and Selim G. Akl

School of Computing
Queen's University
Kingston, Ontario
Canada K7L 3N6
E-mail:{arezou, akl}@cs.queensu.ca

July 15, 2005

Abstract

The problem of real-time scheduling spans a broad spectrum of algorithms from simple uniprocessor to highly sophisticated multiprocessor scheduling algorithms. In this paper, we study the characteristics and constraints of real-time tasks which should be scheduled to be executed. Analysis methods and the concept of optimality criteria, which leads to design appropriate scheduling algorithms, will also be addressed. Then, we study real-time scheduling algorithms for uniprocessor systems, which can be divided into two major classes: off-line and on-line. On-line algorithms are partitioned into either static or dynamic-priority based algorithms. We discuss both preemptive and non-preemptive static-priority based algorithms. For dynamic-priority based algorithms, we study the two subsets; namely, planning based and best effort scheduling algorithms. Some of the uniprocessor scheduling algorithms are illustrated by examples in the Appendix. Multiprocessor scheduling algorithms is another class of real-time scheduling algorithms which is discussed in the paper as well. We also describe techniques to deal with aperiodic and sporadic tasks, precedence constraints, and priority inversion.

---

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

1 Real-Time Systems

1.1 Introduction

In the physical world, the purpose of a real-time system is to have a physical effect within a chosen time-frame. Typically, a real-time system consists of a controlling system (computer) and a controlled system (environment). The controlling system interacts with its environment based on information available about the environment. On a real-time computer, which controls a device or process, sensors will provide readings at periodic intervals and the computer must respond by sending signals to actuators. There may be unexpected or irregular events and these must also receive a response. In all cases, there will be a time bound within which the response should be delivered. The ability of the computer to meet these demands depends on its capacity to perform the necessary computations in the given time. If a number of events occur close together, the computer will need to schedule the computations so that each response is provided within the required time bounds. It may be that, even so, the system is unable to meet all the possible unexpected demands. In this case we say that the system lacks sufficient resources; a system with unlimited resources and capable of processing at infinite speed could satisfy any such timing constraint. Failure to meet the timing constraint for a response can have different consequences; there may be no effect at all, or the effects may be minor or correctable, or the results may be catastrophic. Each task occurring in a real-time system has some timing properties. These timing properties should be considered when scheduling tasks on a real-time system. The timing properties of a given task refer to the following items [30, 33, 59, 21]:

- *Release time* (or *ready time*): Time at which the task is ready for processing.
- *Deadline*: Time by which execution of the task should be completed, after the task is released.
- *Minimum delay*: Minimum amount of time that must elapse before the execution of the task is started, after the task is released.
- *Maximum delay*: Maximum permitted amount of time that elapses before the execution of the task is started, after the task is released.
- *Worst case execution time*: Maximum time taken to complete the task, after the task is released. The worst case execution time is also referred to as the *worst case response time*.
- *Run time*: Time taken without interruption to complete the task, after the task is released.
- *Weight* (or *priority*): Relative urgency of the task.

Real-time systems span a broad spectrum of complexity from very simple micro-controllers to highly sophisticated, complex and distributed systems. Some examples of real-time systems include process control systems, flight control systems, flexible manufacturing applications, robotics, intelligent highway systems, and high speed and multimedia communication systems [30, 33, 59, 26, 12, 5, 21, 47].

The objective of a computer controller might be to command the robots to move parts from machines to conveyors in some required fashion without colliding with other objects. If the computer controlling a robot does not command it to stop or turn in time, the robot might collide with another object on the factory floor.

A real-time system will usually have to meet many demands within a limited time. The importance of the demands may vary with their nature (e.g. a safety-related demand may be more important than a simple data-logging demand) or with the time available for a response. So the allocation of the system resources needs to be planned so that all demands are met by the time of their respective deadlines. This is usually done using a scheduler which implements a scheduling policy that determines how the resources of the system are allocated to the program. Scheduling policies can be analyzed mathematically so the precision of the formal specification and program development stages can be complemented by a mathematical timing analysis of the program properties [30, 59, 12].

With large and variable processing loads, it may be necessary to have more than one processor in the system. If tasks have dependencies, calculating task completion times on a multiprocessor system is inherently more difficult than on a single-processor system.

The nature of the application may require distributed computing, with nodes connected by communication lines. The problem of finding completion times is then even more difficult, as communication between tasks can now take varying times [59].

1.2 Real-Time Systems

In this section we present a formal definition of real-time systems. As we mentioned in Section 1.1, real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. If the timing constraints of the system are not met, system failure is said to have occurred. Hence, it is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behavior requires that the system be *predictable*. Predictability means that when a task is activated it should be possible to determine its completion time with certainty. It is also desirable that the system attain a high degree of utilization while satisfying the timing constraints of the system [59, 33, 30, 12, 5].

It is imperative that the state of the environment, as received by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling

systems' activities may be disastrous. Therefore, periodic monitoring of the environment as well as timely processing of the sensed information is necessary [59, 30].

A real-time application is normally composed of multiple tasks with different levels of criticality. Although missing deadlines is not desirable in a real-time system, *soft real-time tasks* could miss some deadlines and the system could still work correctly. However, missing some deadlines for soft real-time tasks will lead to paying penalties. On the other hand, *hard real-time tasks* cannot miss any deadline, otherwise, undesirable or fatal results will be produced in the system. There exists another group of real-time tasks, namely *firm real-time tasks*, which are such that the sooner they finish their computations before their deadlines, the more rewards they gain [30, 59, 33].

We can formally define a real-time system as follows.

Consider a system consisting of a set of tasks, $T = \{\tau_1, \tau_2, ..., \tau_n\}$, where the worst case execution time of each task $\tau_i \in T$ is C_i. The system is said to be real-time if there exists at least one task $\tau_i \in T$, which falls into one of the following categories:

(1) Task τ_i is a **hard real-time** task. That is, the execution of the task τ_i should be completed by a given deadline D_i; i.e., $C_i \leq D_i$.

(2) Task τ_i is a **soft real-time** task. That is, the later the task τ_i finishes its computation after a given deadline D_i, the more penalty it pays. A penalty function $P(\tau_i)$ is defined for the task. If $C_i \leq D_i$, the penalty function $P(\tau_i)$ is zero. Otherwise $P(\tau_i) > 0$. The value of $P(\tau_i)$ is an increasing function of $C_i - D_i$.

(3) Task τ_i is a **firm real-time** task. That is, the earlier the task τ_i finishes its computation before a given deadline D_i, the more rewards it gains. A reward function $R(\tau_i)$ is defined for the task. If $C_i \geq D_i$, the reward function $R(\tau_i)$ is zero. Otherwise $R(\tau_i) > 0$. The value of $R(\tau_i)$ is an increasing function of $D_i - C_i$.

The set of real-time tasks $T = \{\tau_1, \tau_2, ..., \tau_n\}$ can be a combination of hard, firm, and soft real-time tasks.

Let T_S be the set of all soft real-time tasks in T; i.e., $T_S = \{\tau_{S,1}, \tau_{S,2}, ..., \tau_{S,l}\}$ with $\tau_{S,i} \in T$. The penalty function of the system is denoted by $P(T)$, where

$$P(T) = \sum_{i=1}^{l} P(\tau_{S,i})$$

Let T_F be the set of all firm real-time tasks in T; i.e., $T_F = \{\tau_{F,1}, \tau_{F,2}, ..., \tau_{F,k}\}$ with $\tau_{F,i} \in T$. The reward function of the system is denoted by $R(T)$, where

$$R(T) = \sum_{i=1}^{k} R(\tau_{F,i})$$

1.3 Problems That Seem Real-Time but Are Not

Sometimes the concept of real-time is misunderstood. The following cases are given to clarify this [69, 70].

- One will occasionally see references to “real-time” systems when what is meant is “on-line”, or “an interactive system with better response time than what we used to have”. This is not always correct. For instance, a system interacting with a human and waiting for a person’s response is not real-time. This is because the system is interacting with a human who can tolerate hundreds of milliseconds of delays without a problem. In other words, since no deadline is given for any task, it is not a real-time system.

  A real-life example is standing in a line waiting for the checkout in a grocery store. If the line can grow longer and longer without bound, the checkout process is not real-time. But, if the length of the line is bounded, customers should be served and output as rapidly, on average, as they arrive into the line. The grocer must lose business or pay a penalty if the line grows longer than the determined bound. In this case the system is real-time. The deadline of checkout process depends on the maximum length given for the line and the average serving time for each costumer.

- In digital signal processing (DSP), if a process requires 2.01 seconds to analyze or process 2.00 seconds of sound, it is not real-time. If it takes 1.99 seconds, it is (or can be made into) a real-time DSP process.

- One will also see references to real-time systems when what is meant is just “fast”. It might be worth pointing out that “real-time” is not necessarily synonymous with “fast”. For example consider a robot that has to pick up something from a conveyor belt. The object is moving, and the robot has a small window of time to pick it up. If the robot is late, the object won’t be there anymore, and thus the job will have been done incorrectly, even though the robot went to the right place. If the robot is too early there, the object won’t be there yet, and the robot may block it.

1.4 Real-Time Scheduling

For a given set of jobs, the general scheduling problem asks for an order according to which the jobs are to be executed such that various constraints are satisfied. Typically, a job is characterized by its execution time, ready time, deadline, and resource requirements. The execution of a job may or may not be interrupted (preemptive or non-preemptive scheduling). Over the set of jobs, there is a precedence relation which constrains the order of execution. Specially, the execution of a job cannot begin until the execution of all its predecessors (according to

the precedence relation) is completed. The system on which the jobs are to be executed is characterized by the amounts of resources available [22, 59, 30, 32, 27, 12].

The following goals should be considered in scheduling a real-time system: [30, 32, 27].

- Meeting the timing constraints of the system
- Preventing simultaneous access to shared resources and devices
- Attaining a high degree of utilization while satisfying the timing constraints of the system; however this is not a primary driver.
- Reducing the cost of context switches caused by preemption
- Reducing the communication cost in real-time distributed systems; we should find the optimal way to decompose the real-time application into smaller portions in order to have the minimum communication cost between mutual portions (each portion is assigned to a computer).

In addition, the following items are desired in advanced real-time systems:

- Considering a combination of hard, firm, and soft real-time activities, which implies the possibility of applying dynamic scheduling policies that respect the optimality criteria.
- Task scheduling for a real-time system whose behavior is dynamically adaptive, reconfigurable, reflexive and intelligent.
- Covering reliability, security, and safety.

Basically, the scheduling problem is to determine a schedule for the execution of the jobs so that they are all completed before the overall deadline [22, 59, 30, 32, 27, 12].

Given a real-time system, the appropriate scheduling approach should be designed based on the properties of the system and the tasks occurring in it. These properties are as follows [22, 59, 30, 32]:

- ***Soft/Hard/Firm real-time tasks***

  The real-time tasks are classified as hard, soft and firm real-time tasks. This is described in Section 1.2.

- ***Periodic/Aperiodic/Sporadic tasks***

  Periodic tasks are real-time tasks which are activated (released) regularly at fixed rates (periods). Normally, periodic tasks have constraints which indicates that instances of them must execute once per period P.

Aperiodic tasks are real-time tasks which are activated irregularly at some unknown and possibly unbounded rate. The time constraint is usually a deadline D.

Sporadic tasks are real-time tasks which are activated irregularly with some known bounded rate. The bounded rate is characterized by a minimum inter-arrival period, that is, a minimum interval of time between two successive activations. The time constraints is usually a deadline D.

An aperiodic task has a deadline by which it must start or finish, or it may have a constraint on both start and finish times. In the case of a periodic task, a period means once per period P or exactly P units apart. A majority of sensory processing is periodic in nature. For example, a radar that tracks flights produces data at a fixed rate [32, 29, 27, 12].

- ***Preemptive/Non-preemptive tasks***

  In some real-time scheduling algorithms, a task can be preempted if another task of higher priority becomes ready. In contrast, the execution of a non-preemptive task should be completed without interruption, once it is started [32, 30, 27, 12].

- ***Multiprocessor/Single processor systems***

  The number of the available processors is one of the main factors in deciding how to schedule a real-time system. In multiprocessor real-time systems, the scheduling algorithms should prevent simultaneous access to shared resources and devices. Additionally, the best strategy to reduce the communication cost should be provided [32, 27].

- ***Fixed/Dynamic priority tasks***

  In priority driven scheduling, a priority is assigned to each task. Assigning the priorities can be done statically or dynamically while the system is running [22, 59, 30, 32, 12].

- ***Flexible/Static systems***

  For scheduling a real-time system, we need to have enough information, such as deadline, minimum delay, maximum delay, run-time, and worst case execution time of each task. A majority of systems assume that much of this information is available a priori and, hence, are based on static design. However, some of the real-time systems are designed to be dynamic and flexible [22, 59, 30, 32, 12].

- ***Independent/Dependent tasks***

  Given a real-time system, a task that is going to start execution may require to receive the information provided by another task of the system. Therefore, execution of a task should be started after finishing the execution of the other task. This is the concept of dependency. The dependent tasks use shared memory or communicate data to transfer the

information generated by one task and required by the other one. While we decide about scheduling of a real-time system containing some dependent tasks, we should consider the order of the starting and finishing time of the tasks [22, 59, 30, 32].

1.5 Overview

This paper is organized as follows.

Section 2 contains a description of the process of modeling real-time problems, defining their optimality criteria, and providing the appropriate scheduling algorithms.We also study the two most popular algorithms that optimally schedule uniprocessor real-time systems.

In Section 3, the real-time scheduling algorithms are classified. We study off-line/on-line scheduling algorithms for uniprocessor/multiprocessor preemptive/non-preemptive fixed-priority/dynamic-priority systems. We also present some algorithms as examples for the classes of the algorithms.

In Section 4, we discuss some techniques to deal with precedence conditions, priority inversion, aperiodic and sporadic tasks while scheduling real-time systems.

Finally, Section 5 contains conclusions and some suggestions of open problems for future research.

In the Appendix, some of the real-time scheduling algorithms are illustrated using examples.

2 Methods and Analysis

2.1 Motivation

One concern in the analysis and development of strategies for task scheduling is the question of predictability of the system's behavior. The concept of predictability was defined in Section 1.2. If there is no sufficient knowledge to predict the system's behavior, especially if deadlines have to be met, the only way to solve the problem is to assume upper bounds on the processing times. If all deadlines are met with respect to these upper bounds, no deadlines will be exceeded for the real task processing times. This approach is often used in a broad class of computer control systems working in real-time environments, where a certain set of control programs must be processed before taking the next sample from the same sensing device. In the following sections, we study some of the methods and techniques that are used to model real-time problems, define their optimality criteria, and provide the appropriate scheduling algorithms [30, 32].

2.2 Scheduling Models and Problem Complexity

The scheduling problems considered in this paper are characterized by a set of *tasks* $T = \{\tau_1, \tau_2, ..., \tau_n\}$ and a set of *processors (machines)* $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$ on which the tasks are to be processed. Besides processors, tasks may require certain additional *resources* $R = \{R_1, R_2, ..., R_k\}$ during their execution. Scheduling, generally speaking, means the assignment of processors from π and resources from R to tasks from T in order to complete all tasks under certain imposed constraints. In classical scheduling theory it is also assumed that each task is to be processed by at most one processor at a time and each processor is capable of processing at most one task at a time [27].

We begin with an analysis of processors, $\pi = \{\pi_1, \pi_2, ..., \pi_m\}$. There are three different types of multiprocessor systems: *identical processors*, *uniform processors* and *unrelated processors*. They are discussed in Section 3.2.

By an *additional resource* we understand a "facility", besides processors, for which the tasks compete.

Let us now consider the assumptions associated with the task set T. In general, a task $\tau_i \in T$ is characterized by the following data.

- Release time R_j; if the ready times are the same for all tasks from T, then $R_j = 0$ is assumed for all j.
- Completion time C_j
- Deadline D_j; usually, penalty functions are defined in accordance with deadlines.
- Priority ω_j
- *Precedence constraints* among tasks. $\tau_i \prec \tau_j$ means that the processing of τ_i must be completed before τ_j can be started. In other words, set T is partially ordered by a precedence relation $\prec$. The tasks in set T are called *dependent* if the order of execution of at least two tasks in T is restricted by their relation. Otherwise, the tasks are called *independent*.

The following parameters can be calculated for each task τ_j, $j = 1, 2, ..., n$ processed in a given schedule:

- *Flow time* $F_j = C_j - R_j$ being the sum of waiting and processing times
- *Lateness* $L_j = C_j - D_j$
- *Tardiness* $K_j = max\{C_j - D_j, 0\}$

Next, some definitions concerning *schedules* and *optimality criteria* are discussed.

A schedule is an assignment of processors from set π (and possibly resources from set R) to tasks from set T in time such that the following conditions are satisfied:

- At every moment each processor is assigned to at most one task and each task is processed by at most one processor.
- The timing constraints of tasks in set T are considered.
- If tasks τ_i and τ_j, $i, j = 1, 2, ..., n$ are in relation $\tau_i \prec \tau_j$, the processing of τ_j is not started before τ_i has been completed.
- A schedule is called *preemptive* if each task may be preempted at any time and restarted later at no cost, perhaps on another processor. If preemption is not allowed we will call the schedule *non-preemptive*.
- Resources constraints are satisfied.

Depending on the type of application we are confronted with, different *performance measures* or *optimality criteria* are used to evaluate schedules. Among the most common measures in scheduling theory are *schedule length (makespan)* $C_{max} = max\{C_j\}$, and *mean flow time* $F = \frac{1}{n}\sum_{j=1}^{n} F_j$ or *mean weighted flow time* $F_w = (\sum_{j=1}^{n} F_j w_j)/(\sum_{j=1}^{n} w_j)$. Minimizing scheduling length is important from the viewpoint of the owner of a set of processors or machines, since it leads to both, the maximization of the processor utilization within makespan C_{max}, and the minimization of the maximum in-process time of the schedule set of tasks. The *mean flow time* criterion is important from the user's viewpoint since its minimization yields a minimization of the mean response time and the mean in-process time of the scheduled task set.

In real-time applications, performance measures are used that take lateness or tardiness of tasks into account. Examples are the *maximum lateness* $L_{max} = max\{L_i\}$, the number of *tardy tasks* $Y = \sum_{j=1}^{n} Y_j$, where $Y_j = 1$, if $C_j > D_j$, and 0 otherwise, or the *weighted number of tardy tasks* $Y_w = \sum_{j=1}^{n} w_j Y_j$, the *mean tardiness* $\overline{K} = \frac{1}{n}\sum_{j=1}^{n} K_j$ or the *mean weighted tardiness* $\overline{K}_w = (\sum_{j=1}^{n} w_j K_j)/(\sum_{j=1}^{n} w_j)$. These criteria involving deadlines are of great importance in many applications. These criteria are also of significance in computer control systems working in a real-time environment since their minimization leads to the construction of schedules with no late task whenever such schedules exist or if a task is not finished on time, the yet unprocessed part of it contributes to the schedule value that has to be minimized.

A schedule for which the value of a particular performance measure γ is at its minimum will be called *optimal*, and the corresponding value of γ will be denoted by γ^*.

Now we define a *scheduling problem* as a set of parameters as described above, together with an optimally criterion.

The criteria mentioned above are basic in the sense that they require specific approaches to the construction of schedules. A scheduling algorithm is an algorithm which constructs a schedule for a given problem.

Scheduling problems belong to the broad class of *combinatorial search problems*. Combinatorial search is among the hardest of common computational problems: the solution time can grow exponentially with the size of the problem [67, 32, 27]. We are given a set of n variables each of which can be assigned b possible values. The problem is to find an assignment for each variable that together satisfy some specified constraints. Fundamentally, the combinatorial search problem consists of finding those combinations of a discrete set of items that satisfy specified requirements. The number of possible combinations to consider grows very rapidly (e.g., exponentially or factorially) with the number of items, leading to potentially lengthy solution times and severely limiting the feasible size of such problems. Because of the exponentially large number of possibilities it appears (though no one knows for sure) that the time required to solve such problems must grow exponentially, in the worst case. These problems form the well-studied class of NP-hard problems [27].

In general, we are interested in optimization algorithms, but because of the inherent complexity of many problems of that type, also approximation or heuristic algorithms are applied. It is rather obvious that very often the time available for solving particular scheduling problems is seriously limited so that only low order polynomial-time algorithms can be applied [27].

2.3 A Simple Model

Let us consider a simple real-time system containing a periodic hard real-time task which should be processed on one processor [30]. The task does not require any extra resource. The priority of the task is fixed.

We define a simple real-time program as follows: Program H receives an event from a sensor every P units of time (i.e. the *inter-arrival time* is P). A task is defined as the processing of an event. In the worst case the task requires C units of computation time. The execution of the task should be completed by D time units after the task starts. If $D < C$, the deadline cannot be met. If $P < D$, the program must still process each event in a time $> P$ if no events are to be lost. Thus the deadline is effectively bounded by P and we need to handle only those cases where $C \leq D \leq P$ [59, 30, 32].

Now consider a program which receives events from *two* sensors. Inputs from Sensor 1 come every P_1 time units and each needs C_1 time units for computation; events from Sensor 2 come every P_2 time units and each needs C_2 time units. Assume the deadlines are the same as the periods, i.e. P_1 time units for Sensor 1 and P_2 time units for Sensor 2. Under what conditions will these deadlines be met?

More generally, if a program receives events from n such devices, how can it be determined

if the deadline for each device will be satisfied?

Before we begin to analyze this problem, we first express our assumptions as follows. We assume that the real-time program consists of a number of *independent tasks* that do not share data or communicate with each other. Also, we assume that each task is periodically invoked by the occurrence of a particular event [30, 32]. The system has one processor; the system periodically receives events from the external environment and these are not buffered. Each event is an invocation for a particular task. Note that events may be periodically produced by the environment or the system may have a timer that periodically creates the events. The processor is idle when it is not executing a task.

Let the tasks of program H be $\tau_1, \tau_2, \tau_3, ..., \tau_n$. Let the inter-arrival timer, or *period*, for invocation to task τ_i be P_i and the computation time for such invocation be C_i.

2.3.1 Scheduling for the Simple Model

One way to schedule the program is to analyze its tasks *statically* and determine their timing properties. These times can be used to create a ***fixed scheduling*** table according to which tasks will be dispatched for execution at run-time [22, 59, 30, 32, 27, 12]. Thus, the order of execution of tasks is fixed and it is assumed that their execution times are also fixed.

Typically, if tasks $\tau_1, \tau_2, ..., \tau_n$ have periods $P_1, P_2, P_3, ..., P_n$, the table must cover scheduling for length of time equal to the *least common multiple* of the periods, i.e. $lcm\{P_1, P_2, ..., P_n\}$, as that is the time in which each task will have an integral number of invocations. If any of the P_i are co-primes, this length of time can be extremely large so where possible it is advisable to choose values of P_i that are small multiples of a common value. We define a *hyper-period* as the period equal to the least common multiple of the periods $P_1, P_2, ..., P_n$ of the n periodic tasks.

Static scheduling has the significant advantage that the order of execution of tasks is determined *off-line* (before the execution of the program), so the run-time scheduling overheads can be very small. But it has some major disadvantages. This is discussed in Section 3.1.

In scheduling terms, a *priority* is usually a positive integer representing the urgency or importance assigned to an activity. By convention, the urgency is in inverse order to the numeric value of the priority, and priority 1 is the highest level of priority. We shall assume here that a task has a single, fixed priority. We can consider the following two simple scheduling disciplines:

Non-preemptive priority based execution: When the processor is idle, the ready task with the highest priority is chosen for execution; once chosen, a task is run to completion.

Preemptive priority based execution: When the processor is idle, the ready task with the highest priority is chosen for execution; at any time, execution of a task can be preempted if a task of higher priority becomes ready. Thus, at all times the processor is either idle or executing the ready task with the highest priority.

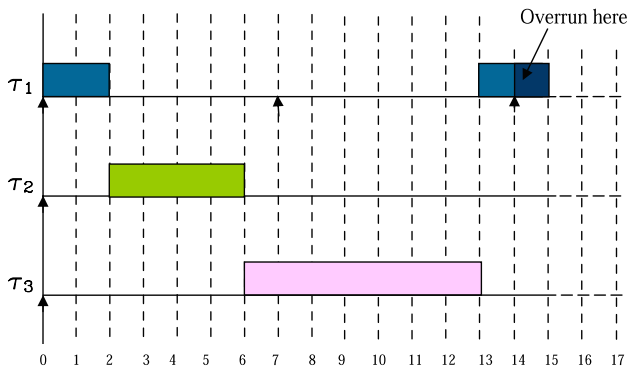

Figure 1: Priorities without preemption

	Priority	*Period*	*Computation time*
τ_1	1	7	2
τ_2	2	16	4
τ_3	3	31	7

Table 1: The priorities, repetition periods and computation times of the tasks τ_1, τ_2 and τ_3 for Example 2.1

Example 2.1 ([32]): Consider a program with 3 tasks τ_1, τ_2 and τ_3, that have the priorities, repetition periods and computation times defined in Table 1. Let the deadline D_i for each task τ_i be P_i. Assume that the tasks are scheduled according to priorities, with no pre-emption, as shown in Figure 1. The arrows in the figure represent the invocation times of the tasks.

If all three tasks have invocations at $time = 0$, task τ_1 will be chosen for execution as it has the highest priority. When it has completed it execution, task τ_2 will be executed until its completion at $time = 6$.

At that time, only task τ_3 is ready for execution and it will execute from $time = 6$ to $time = 13$, even though an invocation comes for task τ_1 at $time = 7$. So there is just one unit of time for task τ_1 to complete its computation requirement of two units and its new invocation will arrive before processing of the previous invocation is complete.

In some cases, the priorities allotted to tasks can be used to solve such problems; in this case, there is no allocation of priorities to tasks under which task τ_1 will meet its deadline. If we keep drawing the timing diagram represented in Figure 1, we can observe that between $time = 15$ and $time = 31$ (at which the next invocation for task τ_3 will arrive) the processor is not always busy and task τ_3 does not need to complete its execution until $time = 31$. If there were some way of making the processor available to tasks τ_1 and τ_2 when needed and then

returning it to task τ_3, they could all meet their deadlines.

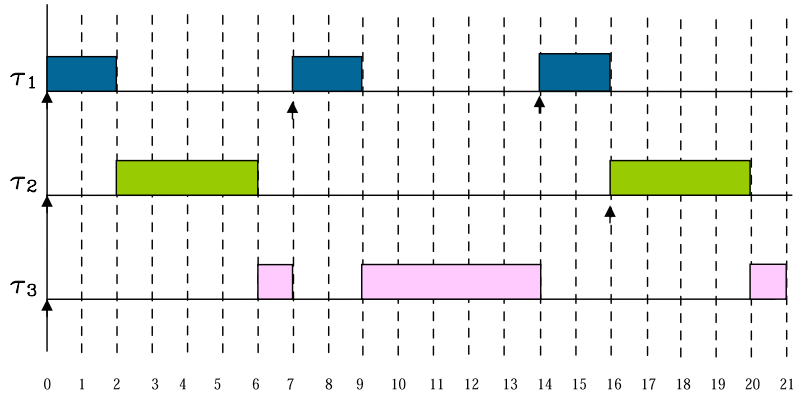

Figure 2: Priorities with preemption

This can be done using priorities with preemption: execution of task τ_3 will then be preempted at $time = 7$, allowing task τ_1 to complete its execution at $time = 9$ (Figure 2). Process τ_3 is preempted once more by task τ_1 at $time = 14$ and this is followed by the next execution of task τ_2 from $time = 16$ to $time = 20$ before task τ_3 completes the rest of its execution at $time = 21$.

2.4 Methods of Analysis

Timing diagrams provide a good way to visualize and even to calculate the timing properties of simple programs. But they have obvious limits, not least of which is that a very long time might be required to reach the point that a deadline is missed. Checking the feasibility of a uniprocessor periodic real-time scheduling algorithm, we need to keep drawing some timing diagrams for a duration that is equal to the least common multiple of the periods [30, 27].

A better method of analysis would be to derive conditions to be satisfied by the timing properties of a program for it to meet its deadlines. Let an *implementation* consist of a hardware platform and the scheduler under which the program is executed. An implementation is called *feasible* if every execution of the program will meet all its deadlines. We should look for the conditions that are *necessary* to ensure that an implementation is feasible. The aim is to find necessary conditions that are also *sufficient*, so that if they are satisfied, an implementation is guaranteed to be feasible [22, 59, 30, 32, 27].

It is shown in [30, 32] how we can examine the conditions that are necessary so that we make sure that the scheduling is feasible. We should find the condition to ensure that the total computation time needed for the task, and for all those of higher priority, is smaller than the period of each task. If we assume that P_i/P_j, $1 \leq j \leq i-1$, represents an integer division,

then we can say that there are P_i/P_j invocations for task P_j in the time P_i and each invocation will need a computation time of C_j. However, if P_i is not exactly divisible by P_j, then either $\lceil P_i/P_j \rceil$ is an overestimate of the number of invocations or $\lfloor P_i/P_j \rfloor$ is an underestimate. We avoid the approximation resulting from integer division by considering an interval M_i which is the *least common multiple* of all periods up to P_i:

$$M_i = lcm(\{P_1, P_2, ..., P_i\})$$

Therefore, as shown in [30, 32], we can conclude that the necessary condition to make sure that the scheduling is feasible is:

$$\sum_{j=1}^{i}(C_j \times \frac{M_i/P_j}{M_j}) \leq 1 \tag{1}$$

Since M_i is exactly divisible by all P_j, $j < i$, the number of invocations at any level j within M_i is exactly M_i/M_j.

Equation (1) is the *Load Relation* and must be satisfied by any feasible implementation. However, this condition *averages* the computational requirements over each lcm period.

Example 2.2 ([32]): Consider a program with two tasks τ_1 and τ_2 that have the priorities, repetition periods and computation times defined as follows. Let the deadline D_i for each task τ_i be equal to P_i.

	Priority	*Period*	*Computation time*
τ_1	1	12	5
τ_2	2	4	2

Since the computation time of task τ_1 exceeds the period of task τ_2, the implementation is infeasible, though it does satisfy condition (1).

Actually, condition (1) fails to take account of an important requirement of any feasible implementation. Not only the *average* load must be smaller than 1 over the interval M_i, but the load must at all times be sufficiently small for the deadlines to be met. More precisely, if at any time t there are u time units left for the next deadline at priority level i, the total computation requirement at time t for level i and all higher levels must be smaller than u. But while on the one hand it is necessary that at every instant there is sufficient computation time remaining for all deadlines to be met, it is important to remember that once a deadline at level i has been met there is no further need to make provision for computation at that level up to the end of the current period.

Based on the properties of the real-time system, the parameters of the system, and the algorithm applied for scheduling, we can determine the sufficient condition of the feasibility test of the scheduling algorithm. The sufficient condition is obtained by calculating the utilization bounds associated with scheduling algorithm. For the systems containing more than one

processor, we not only should decide about the appropriate scheduling algorithm, but we also have to specify the *allocation algorithm* which assigns the tasks to the available processors. For multiprocessor real-time systems, calculating the utilization bounds associated with *(scheduling algorithm, allocation algorithm)* pairs leads us to achieving the sufficient conditions of the feasibility test, analogous to those known for uniprocessors. This approach has several interesting features: it allows us to carry out fast schedulability tests and to qualify the influence of certain parameters, such as the number of processors, on scheduling. For some algorithms, this bound considers not only the number of processors, but also the number of the tasks and their sizes [22, 6, 23, 30, 32].

Let us study the above concepts on the *Rate-Monotonic* algorithm (RM) and *Earliest Deadline First* algorithm (EDF) [32, 22, 59, 30]. Both the RM and EDF algorithms are *optimal* real-time scheduling algorithms. An optimal real-time scheduling algorithm is one which may fail to meet a deadline only if no other scheduling algorithm can meet the deadline. The following assumptions are made for both the RM and EDF algorithms.

(**a**) No task has any nonpreemptable section and the cost of preemption is negligible.

(**b**) Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.

(**c**) All tasks are independent; there are no precedence constraints.

2.5 RM Scheduling

The RM scheduling algorithm is one of the most widely studied and used in practice [22, 59, 30, 32, 27]. It is a uniprocessor static-priority preemptive scheme. For the RM scheduling algorithm, in addition to assumptions (**a**) to (**c**), we assume that all tasks are periodic and the priority of task τ_i is higher than the priority of task τ_j, where $i < j$. The RM scheduling algorithm is an example of priority driven algorithms with static priority assignment in the sense that the priorities of all instances are known even before their arrival. The priorities of all instances of each task are the same. They are determined only by the period of the task. A periodic task consists of an infinite sequence of instances with periodic ready times, where the deadline of a request could be less than, greater than, or equal to the ready time of the succeeding instance. Furthermore, the execution times of all the instances of a task are the same. A periodic task τ_i is characterized by three parameters P_i, the period of the instance, C_i, the execution time, and D_i, the deadline of the tasks. The utilization factor of a set of n periodic tasks is defined by $\sum_{i=1}^{n} C_i/P_i$, where $P_1, P_2, ..., P_n$ are the periods and $C_1, C_2, ..., C_n$ are the execution times of the n tasks. If $\sum_{i=1}^{n} C_i/P_i \leq n(2^{1/n} - 1)$, where n is the number of tasks to be scheduled, then the RM algorithm will schedule all the tasks to meet their respective

deadlines. Note that this is a sufficient, but not a necessary, condition. That is, there may be task sets with a utilization greater than $n(2^{1/n} - 1)$ that are schedulable by the RM algorithm.

A given set of tasks is said to be RM-schedulable if the RM algorithm produces a schedule that meets all the deadlines. The sufficient and necessary conditions for feasibility of RM scheduling is studied in [32] as follows.

Given a set of n periodic tasks $\tau_1, \tau_2, ..., \tau_n$, whose periods and execution times are $P_1, P_2, ..., P_n$ and $C_1, C_2, ..., C_n$, respectively, we suppose task τ_i completes executing at t. We consider the following notation:

$$W_i(t) = \sum_{j=1}^{i} C_j \lceil \frac{t}{P_j} \rceil = t - \textit{idle time}$$

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L = \min_{0 \le t \le P_i} L_i(t)$$

Task τ_i can be feasibly scheduled using RM *if and only if* $L_i(t) \le 1$. In this case $\tau_1, \tau_2, ..., \tau_{i-1}$ are also feasibly scheduled.

Thus far, we have only considered periodic tasks. As defined in Section 1.4, sporadic tasks are released irregularly, often in response to some event in the operating environment. While sporadic tasks do not have periods associated with them, there must be some maximum rate at which they can be released. That is, we must have some minimum interarrival time between the release time of successive iterations of sporadic tasks. Otherwise, there is no limit to the amount of workload that sporadic tasks can add to the system and it will be impossible to guarantee that deadlines are met. The different approaches to deal with aperiodic and sporadic tasks are outlined in Section 4.1 and Section 4.2.

One drawback of the RM algorithm is that task priorities are defined by their periods. Sometimes, we must change the task priorities to ensure that all critical tasks get completed. Suppose that we are given a set of tasks containing two tasks τ_i and τ_j, where $P_i < P_j$, but τ_j is a critical task and τ_i is a noncritical task. We check the feasibility of the RM scheduling algorithm for the tasks $\tau_1, \tau_2, ..., \tau_n$. Suppose that if we take the worst-case execution times of the tasks, we cannot guarantee the schedulability of the tasks. However, in the average case, they are all RM-schedulable. The problem is how to arrange matters so that all the critical tasks meet their deadlines under the RM algorithm even in the worst case, while the noncritical tasks, such as τ_i, meet their deadlines in many other cases. The solution is either of the following two methods.

- We lengthen the period of the noncritical task, i.e. τ_i, by a factor of k. The original task should also be replaced by k tasks, each phased by the appropriate amount. The

parameter k should be chosen such that we obtain $P_i' > P_j$ (see [32, Example 3.10] for an example).

- We reduce the period of the critical task, i.e. τ_j, by a factor of k. Then we should replace the original task by one whose (both worst case and average case) execution time is also reduced by a factor of k. The parameter k should be chosen such that we obtain $P_i > P_j'$ (see [32, Example 3.10] for an example).

So far, we have assumed that the relative deadline of a task is equal to its period. If we relax this assumption, the RM algorithm is no longer an optimum static-priority scheduling algorithm. When $D_i \leq P_i$, at most one initiation of the same task can be alive at any one time. However, when $D_i > P_i$, it is possible for multiple initiations of the same task to be alive simultaneously. For the latter case, we have to check a number of initiations to obtain the worst-case response time. Therefore, checking for RM-schedulability for the case $D_i > P_i$ is much harder than for the case $D_i \leq P_i$. Suppose we have a task set for which there exists a γ such that $D_i = \gamma P_i$, for each task τ_i. In [32], the necessary and sufficient condition for the tasks of the set to be RM-schedulable is given.

The RM algorithm takes $O((N+\alpha)^2)$ time in the worst case, where N is the total number of the requests in each hyper-period of n periodic tasks in the system and α is the number of aperiodic tasks.

Two examples scheduled by RM algorithm are presented in the Appendix.

2.6 EDF Scheduling

The EDF scheduling algorithm is a priority driven algorithm in which higher priority is assigned to the request that has earlier deadline, and a higher priority request always preempts a lower priority one [60, 22, 59, 30, 32, 27]. This scheduling algorithm is an example of priority driven algorithms with *dynamic priority* assignment in the sense that the priority of a request is assigned as the request arrives. EDF is also called the *deadline-monotonic* scheduling algorithm. Suppose each time a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their deadlines. If sorted lists are used, the EDF algorithm takes $O((N+\alpha)^2)$ time in the worst case, where N is the total number of the requests in each hyper-period of n periodic tasks in the system and α is the number of aperiodic tasks.

For the EDF algorithm, we make all the assumptions we made for the RM algorithm, except that the tasks do not have to be periodic.

EDF is an optimal uniprocessor scheduling algorithm. That is, if EDF cannot feasibly schedule a task set on a uniprocessor, there is no other scheduling algorithm that can. This can be proved by using a *time slice swapping* techniques. Using this technique, we can show that any valid schedule for any task set can be transformed into a valid EDF schedule.

If all tasks are periodic and have relative deadlines equal to their periods, they can be feasibly scheduled by EDF *if and only if* $\sum_{i=1}^{n} C_i/P_i \leq 1$. There is no simple schedulability test corresponding to the case where the relative deadlines do not all equal the periods; in such a case, we actually have to develop a schedule using the EDF algorithm to see if all deadlines are met over a given interval of time. The following is the schedulability test for EDF under this case.

Define $U = \sum_{i=1}^{n} C_i/P_i$, $D_{max} = \max_{1 \leq i \leq n}\{D_i\}$ and $P = lcm(P_1, ..., P_n)$, where lcm stands for least common multiple. Consider $h(t)$ to be the sum of the execution times of all tasks whose absolute deadlines are smaller than t. A task set of n tasks is *not* EDF-feasible *if and only if*

- $U < 1$ *or*
- there exists $t < \min\{P + D_{max},\ \frac{U}{1-U} \max_{1 \leq i \leq n}\{P_i - D_i\}\}$ such that $h(t) > t$

Very little is known about algorithms that produce an optimal solution. This is due to either of the following reasons.

- Some real-time scheduling problems are NP-complete. Therefore, we cannot say whether there is any polynomial time algorithm for the problems. For this group, we should search for heuristic algorithms. Given a heuristic algorithm, we should investigate for the sufficient conditions for feasible scheduling. The sufficient conditions are used to determine whether a given task set can be scheduled feasibly by the algorithm upon the available processors. Many researches have also focused on searching for heuristic algorithms whose results are compared to the optimal results. In fact, for problems in this class the optimal solution cannot be obtained in polynomial time. Approximation algorithms are polynomial time heuristic algorithms whose performance is compared with the optimal performance.

- As for the second group of real-time scheduling problems, there exists polynomial algorithms which provide feasible schedule of any task set which satisfy some specific conditions. For example any set of periodic tasks which satisfy $\sum_{i=1}^{n} C_i/P_i \leq 1$ is guaranteed to be feasibly scheduled using EDF. Recall that an optimal scheduling algorithm is one which may fail to meet a deadline only if no other scheduling algorithm can meet the deadline. Therefore, a feasible scheduling algorithm is optimal if there is no other feasible algorithm with looser conditions. In order to prove optimality of a scheduling algorithm, the feasibility conditions of the algorithm must be known. For example there is no dynamic-priority scheduling algorithm that can successfully schedule a set of periodic tasks where $\sum_{i=1}^{n} C_i/P_i > 1$. Therefore, EDF is an optimal algorithm.

The optimal algorithm for a real-time scheduling problem is not unique. For instance, in addition to EDF algorithm, there is another optimal dynamic-priority scheduling algorithm, which is the least laxity first (LLF) algorithm. The laxity of a process is defined as the deadline minus remaining computation time. In other words, the laxity of a job is the maximal amount of time that the job can wait and still meet its deadline. The algorithm gives the highest priority to the active job with the smallest laxity. Then the job with the highest priority is executed. While a process is executing, it can be preempted by another whose laxity has decreased to below that of the running process. A problem arises with this scheme when two processes have similar laxities. One process will run for a short while and then get preempted by the other and vice versa. Thus, many context switches occur in the lifetime of the processes. The least laxity first algorithm is an optimal scheduling algorithm for systems with periodic real-time tasks [26, 68, 43]. If each time a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their laxities. In this case, the worst case time complexity of the LLF algorithm is $O((N+\alpha)^2)$, where N is the total number of the requests in each hyper-period of n periodic tasks in the system and α is the number of aperiodic tasks.

The EDF and LLF algorithms are illustrated using examples in the Appendix.

Although many people have worked on feasibility analysis of polynomial algorithms, still further investigation is required. Verification of optimality of scheduling algorithms is another subject that should be studied further.

3 Scheduling Algorithms of Real-Time Systems

The goals for real-time scheduling are completing tasks within specific time constraints and preventing from simultaneous access to shared resources and devices [22, 30, 32, 27]. Although system resource utilization is of interest, it is not a primary driver. In fact, predictability and temporal correctness are the principal concerns. The algorithms used, or proposed for use, in real-time scheduling vary from relatively simple to extremely complex.

The topic of real-time scheduling algorithms can be studied for either uniprocessor or multiprocessor systems. We first study uniprocessor real-time scheduling algorithms.

3.1 Uniprocessor Scheduling Algorithms

The set of uniprocessor real-time scheduling algorithms is divided into two major subsets, namely *off-line scheduling* algorithms and *on-line scheduling* algorithms.

Off-line algorithms **(Pre-run-time scheduling)** generate scheduling information prior to system execution [22, 59, 30, 32, 27, 60]. The scheduling information is then utilized by the

system during runtime. The EDF algorithm and the off-line algorithm provided in [20] are examples of off-line scheduling algorithms.

In systems using off-line scheduling, there is generally, if not always, a required ordering of the execution of processes. This can be accommodated by using precedence relations that are enforced during off-line scheduling. Preventing simultaneous access to shared resources and devices is another function that a priority based preemptive off-line algorithm must enforce. This can be accomplished by defining which portion of a process cannot be preempted by another task and then defining exclusion constraints and enforcing them during the off-line algorithm. In Section 4.4, we study the methods that address this problem.

Another goal that may be desired for off-line schedules is reducing the cost of context switches caused by preemption. This can be done by choosing algorithms that do not result in a large number of preemptions, such as the EDF algorithm. It is also desirable to increase the chances that a feasible schedule can be found. If the input to the chosen off-line scheduling algorithm is exactly the input to the real-time system and not an approximation, then the mathematical off-line algorithms are more likely to find a feasible schedule. In a predictable environment, these algorithms can guarantee system performance. Off-line algorithms are good for applications where all characteristics are known a priori and change very infrequently. A fairly complete characterization of all processes involved, such as execution times, deadlines, and ready times are required for off-line scheduling. The off-line algorithms need large amount of off-line processing time to produce the final schedule and due to this they are quite inflexible. Any change to the system processes requires starting the scheduling problem over from the beginning. In addition, these algorithms cannot handle an environment that is not completely predictable. Although a strict off-line scheduler has no provision for handling aperiodic tasks, it is possible to translate an aperiodic process into a periodic one, thus allowing aperiodic processes to be scheduled using off-line scheduling. A major advantage of off-line scheduling is significant reduction in run-time resources, including processing time, for scheduling. However, since it is inflexible, any change requires re-computing the entire schedule [22, 30, 32, 27].

The real advantage of off-line scheduling is that in a predictable environment it can guarantee system performance.

On-line algorithms generate scheduling information while the system is running [22, 30, 32, 27]. The on-line schedulers do not assume any knowledge of process characteristics which have not arrived yet. These algorithms require a large amount of run-time processing time. However, if different modes or some form of error handling is desired, multiple off-line schedules can be computed, one for each alternate situation. At run-time, a small on-line scheduler can choose the proper one.

One of the severe problems that can occur with priority based preemptive on-line algorithms is *priority inversion* [22, 32, 65]. This occurs when a lower priority task is using a resource which is required by a higher priority task and this causes blocking the higher priority task by

the lower priority one. Methods of coping with this problem are discussed in Section 4.4.

The major advantage of on-line scheduling is that there is no requirement to know tasks characteristics in advance and they tend to be flexible and easily adaptable to environment changes. However, the basic assumption that the system has no knowledge of process characteristics for tasks that have not yet arrived, severely restricts the potential for the system to meet timing and resource sharing requirements. If the scheduler does not have such knowledge, it is impossible to guarantee that system timing constraints will be met. Despite the disadvantages of on-line scheduling, this method is used for scheduling of many real-time systems because it does work reasonably well under most circumstances and it is flexible.

On-line scheduling algorithms can be divided into *Static-priority based* algorithms and *Dynamic-priority based* algorithms, which are discussed as follows.

- **Static-priority based algorithms**

  Static-priority based algorithms are relatively simple to implement but lack flexibility. They are arguably the most common in practice and have a fairly complete theory. They work well with fixed periodic tasks but do not handle aperiodic tasks particularly well, although there are some methods to adapt the algorithms so that they can also effectively handle aperiodic tasks. Static priority-based scheduling algorithms have two disadvantages, which have received a significant amount of study. Their low processor utilization and poor handling of aperiodic and soft-deadline tasks have prompted researchers to search for ways to combat these deficiencies [22].

  On-line Static-priority based algorithms may be either *preemptive* or *non-preemptive* [35, 22, 32, 65, 3, 11, 10]. For example, the Rate-monotonic algorithm and the *Rate-monotonic deferred server* (DS) scheduling algorithm are in the class of Preemptive Static-priority based algorithms [22, 32]. The DS algorithm has a time complexity in $O((N+\alpha)^2)$, where α is the number of active aperiodic requests and N is the total number of the requests in each hyper-period of n periodic tasks in the system.

  Many real-time systems have the characteristic in which the order of task execution is known a priori and each task must complete before another task can start. These systems can be scheduled non-preemptively. This scheduling technique, which is called non-preemptive static-priority based algorithms, avoids the overhead associated with multiple context switches per task. This property improves processor utilization. Additionally, tasks are guaranteed of meeting execution deadlines [22, 30, 32, 27].

  The two following non-preemptive algorithms attempt to provide high processor utilization while preserving task deadline guarantees and system schedulability.

  - *Parametric dispatching algorithm ([25, 22]):* This algorithm uses a calendar of functions, which maintains for each task τ_i two functions, Min_i and Max_i, de-

scribing the upper and lower bounds on allowable start times for the task. During an off-line component, the timing constraints between tasks are analyzed to generate the calendar of functions. Then, during system execution, these bounds are passed to dispatcher which then determines when within the window to start execution of the task. This decision can be based on whether there are other non-real-time tasks waiting to execute. The worst case time complexities of the off-line and on-line components of the Parametric dispatching algorithm are $O(n)$ and $O((N+\kappa)^2 \log(N+\kappa))$, respectively, where n is the number of periodic tasks, N is the total number of the requests of real-time task in each hyper-period of n periodic tasks in the system, and κ is the number of the requests of non-real-time tasks.

– *Predictive algorithm ([52, 22]):* This algorithm depends upon known a priori task execution and arrival times. When it is time to schedule a task for execution, the scheduler not only looks at the first task in the ready queue, but also looks at the deadlines for tasks that are predicted to arrive prior to the first task's completion. If a later task is expected to arrive with an earlier deadline than the current task, the scheduler may insert CPU idle time and wait for the pending arrival if this will produce a better schedule. In particular, the insertion of idle time may keep the pending task from missing its deadline. The Predictive algorithm takes $O(n^2)$ time in the worst case, where n is the number of tasks.

These algorithms both have drawbacks when applied to real-world systems. Both algorithms require significant a priori knowledge of the system tasks, both execution times and ordering. Therefore, they are quite rigid and inflexible.

- **Dynamic-priority based algorithms**

  Dynamic-priority based algorithms require a large amount of on-line resources. However, this allows them to be extremely flexible. Many dynamic-priority based algorithms also contain an off-line component. This reduces the amount of on-line resources required while still retaining the flexibility of a dynamic algorithm. There are two subsets of dynamic algorithms: *planning based* and *best effort*. They attempt to provide better response to aperiodic tasks or soft tasks while still meeting the timing constraints of the hard periodic tasks. This is often accomplished by utilization of spare processor capacity to service soft and aperiodic tasks [22, 32, 27, 26].

  Planning Based Algorithms guarantee that if a task is accepted for execution, the task and all previous tasks accepted by the algorithm will meet their time constraints [22, 32].

  The planning based algorithms attempt to improve the response and performance of a system to aperiodic and soft real-time tasks while continuing to guarantee meeting the

deadlines of the hard real-time tasks. The traditional way of handling aperiodic and soft real-time tasks in a system that contained periodic tasks with hard deadlines is to allow the aperiodic or soft real-time tasks to run in the background. By this method, the aperiodic or soft real-time tasks get served only when the processor has nothing else to do. The result of this method is unpredictable and normally rather poor response to these tasks. The other approach used was to model aperiodic tasks as periodic tasks with a period equal to the minimum time between their arrivals and then schedule them using the same algorithm as for the real periodic tasks. This tended to be extremely wasteful of CPU cycles because the minimum period between arrivals is usually significantly smaller than the average. Many researchers have tried to counter these problems by proposing a variety of approaches that utilize spare processor time in a more structured form than simple background processing [51, 55]. Some of these algorithms attempt to identify and capture spare processor capacity and use it to execute aperiodic and soft real-time tasks. Other utilize a more dynamic scheduling method in which aperiodic tasks are executed instead of a higher priority periodic task, when the system can confirm that doing so will not jeopardize the timely completion of the periodic tasks [41, 57, 61].

The general model for these types of algorithms is a system where all periodic tasks have hard deadlines equal to the end of their period, their period is constant, and their worsts case execution times are constant. All aperiodic tasks are assumed to have no deadlines and their arrival or ready times are unknown.

Planning based algorithms tend to be quite flexible in servicing aperiodic tasks while still maintaining the completion guarantees for hard-deadline tasks. Most of the algorithms also provide a form of guarantee for aperiodic tasks. They reject a task for execution if they cannot guarantee its on-time completion. Most of the planning based algorithms can provide higher processor utilization than static priority-based algorithm while still guaranteeing on-time completion of accepted tasks.

The Earliest Deadline First scheduling [37, 60, 32] is one of the first planning based algorithms proposed. It provides the basis for many of the algorithms currently being studied and used. The LLF algorithm is another planning based algorithm.

The Dynamic Priority Exchange Server, Dynamic Sporadic Server, Total Bandwidth Server, Earliest Deadline Late Server, and Improved Priority Exchange Server are examples of planning based algorithms, which work under EDF scheduling. They are discussed in Section 4.2.

Best Effort Algorithms seek to provide the best benefit to the application tasks in overload conditions. The Best Effort scheduling algorithms seek to provide the best benefit to the application tasks. The best benefit that can be accrued by an application task is based on application-specified benefit functions such as the energy consumption func-

tion [62, 48]. More precisely, the objective of the algorithms is to maximize the *accrued benefit ratio*, which is defined as the ratio of total accrued benefit to the sum of all task benefits [36, 22, 32].

There exist many best effort real-time scheduling algorithms. Two of the most prominent of them are the Dependent Activity Scheduling Algorithm (DASA) [16] and the Lockes Best Effort Scheduling Algorithm (LBESA) [38]. DASA and LBESA are equivalent to the Earliest Deadline First (EDF) algorithm during underloaded conditions [16], where EDF is optimal and guarantees that all deadlines are always satisfied. In the event of an overload situation, DASA and LBESA seek to maximize the aggregate task benefit.

The DASA algorithm makes scheduling decisions using the concept of benefit densities. The benefit density of a task is the benefit accrued per unit time by the execution of the task. The objective of DASA is to compute a schedule that will maximize the aggregate task benefit. The aggregate task benefit is the cumulative sum of the benefit accrued by the execution of the tasks. Thus, since task benefit functions are step-benefit functions, a schedule that satisfies all deadlines of all tasks will yield the maximum aggregate benefit.

LBESA [38] is another best effort real-time scheduling algorithm. It is similar to DASA in that both algorithms schedule tasks using the notion of benefit densities and are equivalent to EDF during underload situations. However, the algorithms differ in the way they reject tasks during overload situations. In [16], it is shown that DASA is generally better than LBESA in terms of aggregate accrued task benefit. While DASA examines tasks in the ready queue in decreasing order of their benefit densities for determining feasibility, LBESA examines tasks in the increasing order of task deadlines. Like DASA, LBESA also inserts each task into a tentative schedule at its deadline-position and checks the feasibility of the schedule. Tasks are maintained in increasing deadline-order in the tentative schedule. If the insertion of a task into the tentative schedule results in an infeasible schedule, then, unlike DASA, LBESA removes the least benefit density task from the tentative schedule. LBESA continuously removes the least benefit density task from the tentative schedule until the tentative schedule becomes feasible. Once all tasks in the ready queue have been examined and a feasible tentative schedule is thus constructed, LBESA selects the earliest deadline task from the tentative schedule.

Both the DASA and LBESA algorithms take $O((N + \alpha)^2)$ time in the worst case, where N is the total number of the requests in each hyper-period of n periodic tasks in the system and α is the number of aperiodic tasks.

3.2 Multiprocessor Scheduling Algorithms

The scheduling of real-time systems has been much studied, particularly upon uniprocessor platforms, that is, upon machines in which there is exactly one shared processor available, and all the jobs in the system are required to execute on this single shared processor. In multiprocessor platforms there are several processors available upon which these jobs may execute. The Pfari scheduling is one of the few known optimal methods for scheduling tasks on multiprocessor systems [7]. However, the optimal assignment of tasks to processors is, in almost all practical cases, an NP-hard problem [24, 44, 35]. Therefore, we must make do with heuristics. The heuristics cannot guarantee that an allocation will be found that permits all tasks to be feasibly scheduled. All that we can hope is to allocate the tasks, check their feasibility, and, if the allocation is not feasible, modify the allocation to try to render its schedule feasible. So far, many heuristic multiprocessor scheduling algorithms have been provided (see, for example, [7, 46, 42, 2, 4, 23, 6, 63, 34, 28, 19, 1, 32]).

When checking an allocation for feasibility, we must account for communication costs. For example, suppose that task τ_2 cannot start before receiving the output of task τ_1. If both tasks are allocated to the same processor, then the communication cost is zero. If they are allocated to separate processors, the communication cost is positive and must be taken into account while checking for feasibility.

The following assumptions may be made to design a multiprocessor scheduling algorithm:

- *Job preemption is permitted*

  That is, a job executing on a processor may be preempted prior to completing execution, and its execution may be resumed later. We may assume that there is no penalty associated with such preemption.

- *Job migration is permitted*

  That is, a job that has been preempted on a particular processor may resume execution on a different processor. Once again, we may assume that there is no penalty associated with such migration.

- *Job parallelism is forbidden*

  That is, each job may execute on at most one processor at any given instant in time.

Real-time scheduling theorists have extensively studied uniprocessor real-time scheduling algorithms. Recently, steps have been taken towards obtaining a better understanding of multiprocessors real-time scheduling. Scheduling theorists distinguish between at least three different kinds of multiprocessor machines:

- *Identical parallel machines*

  These are multiprocessors in which all the processors are identical, in the sense that they have the same computing power.

- *Uniform parallel machines*

  By contrast, each processor in a uniform parallel machine is characterized by its own computing capacity, with the interpretation that a job that executes on a processor of computing capacity s for t time units completes $s \times t$ units of execution. Actually, identical parallel machines are a special case of uniform parallel machines, in which the computing capacities of all processors are equal.

- *Unrelated parallel machines*

  In unrelated parallel machines, there is an execution rate $r_{i,j}$ associated with each job-processor ordered pair (J_i, π_j), with the interpretation that job J_i completes $(r_{i,j} \times t)$ units of execution by executing on processor π_j for t time units.

Multiprocessor scheduling techniques fall into two general category:

- **Global Scheduling Algorithms**

  Global scheduling algorithms store the tasks that have arrived but not finished their execution in one queue which is shared among all processors. Suppose there exist m processors. At every moment the m highest priority tasks of the queue are selected for execution on the m processors using preemption and migration if necessary [23, 32].

  The *focused addressing and bidding algorithm* is an example of global scheduling algorithms [32]. The main idea of the algorithm is as follows. Each processor maintains a status table that indicates which tasks it has already committed to run. In addition, each processor maintains a table of the surplus computational capacity at every other processor in the system. The time axis is divided into windows, which are intervals of fixed duration, and each processor regularly sends to its colleagues the fraction of the next window that is currently free.

  On the other hand, an overloaded processor checks its surplus information and selects a processor that seems to be most likely to be able to successfully execute that task by its deadline. It ships the tasks out to that processor, which is called selected task. However, the surplus information may have been out of date and it is possible that the selected processor will not have the free time to execute the task. In order to avoid this problem, and in parallel with sending out the task to the selected processor, the originating processor asks other lightly loaded processors how quickly they can successfully process the task.

The replies are sent to the selected processor. If the selected processor is unable to process the task successfully, it can review the replies to see which other processor is most likely to be able to do so, and transfers the task to that processor.

- **Partitioning Scheduling Algorithms**

  Partitioning scheduling algorithms partition the set of tasks such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to migrate, hence the multiprocessor scheduling problem is transformed to many uniprocessor scheduling problems [23, 32].

  The *next fit algorithm for RM scheduling* is a multiprocessor scheduling algorithm that works based on the partitioning strategy [32]. In this algorithm, we define a set of classes of the tasks. The tasks, which are in the same class, are guaranteed to satisfy the RM-schedulability on one processor. We allocate tasks one by one to the appropriate processor class until all the tasks have been assigned. Then, with this assignment, we run the RM scheduling algorithm on each processor.

Global strategies have several disadvantages versus partitioning strategies. Partitioning usually has a low scheduling overhead compared to global scheduling, because tasks do not need to migrate across processors. Furthermore, partitioning strategies reduce a multiprocessor scheduling problem to a set of uniprocessor ones and then well-known uniprocessor scheduling algorithms can be applied to each processor. However, partitioning has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is an NP-complete problem. Thus, tasks are usually partitioned using non-optimal heuristics. Second, as shown in [13], task systems exist that are schedulable if and only if tasks are not partitioned. Still, partitioning approaches are widely used by system designers. In addition to the above approaches, we can apply hybrid partitioning/global strategies. For instance, each job can be assigned to a single processor, while a task is allowed to migrate.

4 Constraints of Real-Time Systems

Many industrial applications with real-time demands are composed of tasks of various types and constraints. Arrival patterns and importance, for example, determine whether tasks are periodic, aperiodic, or sporadic, and soft, firm, or hard. The controlling real-time system has to provide for a combined set of such task types. The same holds for the various constraints on tasks. In addition to basic temporal constraints, such as periods, start-times, deadlines, and synchronization demands such as precedence, or mutual exclusion, a system has to fulfill complex application demands which cannot be expressed directly with basic constraints. An example for complex demands is a control application that may require constraints on individual instances,

rather than periods. The set of types and constraints of tasks determines the scheduling algorithm during system design. Adding constraints, however, increases scheduling overhead or requires the development of new appropriate scheduling algorithms. Consequently, a designer given an application composed of mixed tasks and constraints has to choose which constraints to focus on in the selection of a scheduling algorithm; others have to be accommodated as well as possible.

4.1 Scheduling of Sporadic Tasks

Sporadic Tasks are released irregularly, often in response to some event in the operating environment. While sporadic tasks do not have periods associated with them, there must be some maximum rate at which they can be released. That is, we must have some minimum interval time between the release of successive iterations of sporadic tasks. Some approaches to deal with sporadic tasks are outlined as follows [32].

- The first method is to simply consider sporadic tasks as periodic tasks with a period equal to their minimum interarrival time.
- The other approach is to define a fictitious periodic task of highest priority and of some chosen fictitious execution period. During the time that this task is scheduled to run on the processor, the processor is available to run any sporadic tasks that may be awaiting service. Outside this time, the processor attends to the periodic tasks. This method is the simplest approach for the problem.
- The *Deferred Server* is another approach, which wastes less bandwidth. Here, whenever the processor is scheduled to run sporadic tasks and finds no such tasks awaiting service, it starts executing the periodic tasks in order of priority. However, if a sporadic task arrives, it preempts the periodic task and can occupy a total time up to the time allotted for sporadic tasks.

4.2 Scheduling of Aperiodic Tasks

Real-time scheduling algorithms that deal with a combination of mixed sets of periodic real-time tasks and aperiodic tasks have been studied extensively [55, 66, 60, 53, 54]. The objective is to reduce the average response time of aperiodic requests without compromising the deadlines of the periodic tasks. Several approaches for servicing aperiodic requests are discussed as follows.

A *Background Server* executes at low priority, and makes use of any extra CPU cycles, without any guarantee that it ever executes. Background Server for aperiodic requests executes whenever the processor is idle (i.e. not executing any periodic tasks and no periodic tasks are

pending). If the load of the periodic task set is high, then utilization left for background service is low, and background service opportunities are relatively infrequent.

The *Polling Server* executes as a high-priority periodic task, and every cycle checks if an event needs to be processed. If not, it goes to sleep until its next cycle and its reserved execution time for that cycle is lost, even if an aperiodic event arrives only a short time after. This results in poor aperiodic response time. Polling consists of creating a periodic task for servicing aperiodic requests. At regular intervals, the polling task is started and services any pending aperiodic requests. However, if no aperiodic requests are pending, the polling task suspends itself until its next period and the time originally allocated for aperiodic service is not preserved for aperiodic execution but is instead used by periodic tasks. Note that if an aperiodic request occurs just after the polling task has suspended, then the aperiodic request must wait until the beginning of the next polling task period or until background processing resumes before being serviced. Even though polling tasks and background processing can provide time for servicing aperiodic requests, they have the drawback that the average wait and response times for these algorithms can be long, especially for background processing.

The purpose of the *Priority Exchange* and *Deferrable Servers* is to improve the aperiodic response time by preserving execution time until required. Taking advantage of the fact that, typically, there is no benefit in early completion of the periodic tasks, the Deferrable Server algorithm assigns higher priority to aperiodic tasks up until the point where the periodic tasks would start to miss their deadlines. Guaranteed alert-class aperiodic service and greatly reduced response times for soft deadline aperiodic tasks are important features of the Deferrable Server algorithm, and both are obtained with the hard deadlines of the periodic tasks still being guaranteed.

The Priority Exchange server allows for better CPU utilization, but is much more complex to implement than the Deferrable Server.

The Priority Exchange technique adds to the task set an aperiodic server that services the aperiodic requests as they arrive. The aperiodic server has the highest priority and executes when an aperiodic task arrives. When there are no aperiodic tasks to service, the server exchanges its priority with the task of next highest priority to allow it to execute.

The *Sporadic Server* is based on the Deferrable Server; but provides with less complexity the same schedulable utilization as the Priority Exchange server. Similarly to other servers, this method is characterized by a period P_S and a capacity C_S, which is preserved for possible aperiodic requests. Unlike other server algorithms, however, the capacity is not replenished at its full value at the beginning of each server period, but only when it has been consumed. The times at which the replenishments occur are chosen according to a replenishment rule, which allows the system to achieve full processor utilization. The Sporadic Server has a fixed priority chosen according to the Rate Monotonic algorithm, that is, according to its period P_S. The Sporadic Server algorithm improves response times for soft-deadline aperiodic tasks and can

guarantee hard deadlines for both periodic and aperiodic tasks.

The above aperiodic servers are designed to operate in conjunction with the Rate Monotonic algorithm [66, 60]. We discuss some other servers that can operate in conjunction with deadline-based scheduling algorithms, such as Earliest Deadline First, as follows.

The *Dynamic Priority Exchange* server is an aperiodic service technique, which can be viewed as an extension to the Priority Exchange server, adapted to work with deadline-based scheduling algorithms. The main idea of the algorithm is to let the server trade its run-time with the run-time of lower priority tasks in case there are no aperiodic requests pending. In this way, the server run-time is only exchanged with periodic tasks, but never wasted unless there are idle times. It is simply preserved, even if at a lower priority, and it can be later reclaimed when aperiodic requests enter the system [60, 55, 56].

The *Dynamic Sporadic Server* is another aperiodic service strategy, which extends the Sporadic Server to work under dynamic EDF scheduler. The main difference between the classical Sporadic Server and its dynamic version consists in the way the priority is assigned to the server. Dynamic Sporadic Server has a dynamic priority assigned through a suitable deadline. The methods of deadline assignment and capacity replenishment are described in [60, 55, 56].

Looking at the characteristics of Sporadic Server, we can realize that when the server has a long period, the execution of the aperiodic requests can be delayed significantly, and this is regardless of the aperiodic execution times. There are two possible approaches to reduce the aperiodic response times. The first is to use a Sporadic Server with a shorter period. This solution, however, increases the run-time overhead of the algorithm because, to keep the server utilization constant, the capacity has to be reduced proportionally, but this causes more frequent replenishment and increases the number of context switches with periodic tasks [60]. A second approach is to assign a possible earlier deadline to each aperiodic request. The assignment must be done in such a way that the overall processor utilization of the aperiodic load never exceeds a specified maximum value U_S. This is the main idea behind another aperiodic service mechanism, which is the *Total Bandwidth Server* [55, 56]. The Total Bandwidth Server is able to provide good aperiodic responsiveness with extreme simplicity. However, a better performance can still be achieved through more complex algorithms. This is possible because, when the requests arrive, the active periodic instances may have enough slack time to be safely preempted. Using the available slack of periodic tasks for advancing the execution of aperiodic requests is the basic principle adopted by the *Earliest Deadline Late Server* [55, 56, 60]. The basic idea behind the Earliest Deadline Late Server is to postpone the execution of periodic tasks as long as possible and use the idle times of periodic schedule to execute aperiodic requests sooner. It is proved that the Earliest Deadline Late Server is optimal, that is, the response times of aperiodic requests under this algorithm are the best achievable [60].

Although optimal, the Earliest Deadline Late Server has too much overhead to be considered practical. However, its main idea can be usefully adopted to develop a less complex

algorithm which still maintains a nearly optimal behavior. The expensive computation of the idle times can be avoided by using the mechanism of priority exchange. With this mechanism the system can easily keep track of the time advanced to periodic tasks and possibly reclaim it at the right priority level. The idle time of the Earliest Deadline Late algorithm can be pre-computed off-line and the server can use them to schedule aperiodic requests, when there are any, or to advance the execution of periodic tasks. In the latter case, the pre-computed idle time can be saved as aperiodic capacity. When an aperiodic request arrives, the scheduler gives the highest priority to the aperiodic request if all of the periodic tasks can wait while still meeting their deadlines. The idea described above is used by the algorithm called *Improved Priority Exchange* [55, 56, 60]. There are two main advantages to this approach. First, a far more efficient replenishment policy is achieved for the server. Second, the resulting server is no longer periodic and it can always run at the highest priority in the system.

In this section, we introduced a set of most popular algorithms that provide good response time to aperiodic tasks in real-time systems. The algorithms differ in their performance and implementation complexity.

4.3 Precedence and Exclusion Conditions

Suppose we have a set of tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. For each task τ_i we are given the worst-case execution time C_i, the deadline D_i, and the release time R_i. We say τ_i *precedes* τ_j if τ_i is in the precedence set of τ_j, that is, τ_j needs the output of τ_i and we cannot start executing τ_j until τ_i has finished executing. Task τ_i *excludes* τ_j if τ_i is not allowed to preempt τ_j. The sentence "τ_i *preempts* τ_j" is true if whenever τ_i is ready to run and τ_j is currently running, τ_j is always preempted by τ_i. Some relations between a given pair of distinct tasks are inconsistent with some other relations. For example, we cannot have both "τ_i precedes τ_j" and "τ_j precedes τ_i". Also, τ_i cannot precede τ_j when τ_j preempts τ_i. There are a few more examples of inconsistent relations.

Having a set of real-time tasks with some precedence and exclusion conditions, we should provide a scheduling algorithm such that not only all deadlines can be met, but also precedence and exclusion conditions can be handled successfully. This scheduling problem is an NP-complete problem [32]. Some heuristic algorithms have been provided for the problem in [32, 49, 39, 31, 17, 58].

Generally, the input of any scheduling problem with precedence constraints consists of a set of real-time tasks and a precedence graph, where a deadline, a release time and an execution time is specified for each task. Sometimes the release time of a job may be later than that of its successors, or its deadline may be earlier than that specified for its predecessors. This condition makes no sense. Therefore, we should derive an effective release time or effective deadline consistent with all precedence constraints, and schedule using that [45]. We apply the

following method in order to achieve an effective release time:

- If a job has no predecessors, its effective release time is its release time.
- If it has predecessors, its effective release time is the maximum of its release time and the effective release times of its predecessors.

An effective deadline can be found as follows.

- If a job has no successors, its effective deadline is its deadline.
- It if has successors, its effective deadline is the minimum of its deadline and the effective deadline of its successors.

On the other hand, an exclusion relation between a given pair of tasks can be reduced to a combination of preemption and precedence relation [32].

4.4 Priority Inversion

In a preemptive priority based real-time system, sometimes tasks may need to access resources that cannot be shared. For example, a task may be writing to a block in memory. Until this is completed, no other task can access that block, either for reading or for writing. The method of ensuring exclusive access is to guard the critical sections with binary semaphores. When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked. If it is, the task is stopped and cannot proceed further until that semaphore is unlocked. If it is not, the task locks the semaphore and enters the critical section. When a task exits the critical section, it unlocks the corresponding semaphore [32, 50, 22].

The following example represents an undesired behavior of the above method. Consider tasks τ_1, τ_2, and τ_3, listed in descending order of priority, which share a processor. There exists a critical section S that is used by both τ_1 and τ_3. It is possible for τ_1 to issue a request for the critical section S when it is locked by τ_3. Meanwhile τ_2 may preempt τ_3. This means that τ_2 which is of lower priority than τ_1, is able to delay τ_1 indirectly. When a lower priority task locks a critical section shared with the higher priority task, the *priority inheritance protocol* is used to prevent a medium priority task from preempting the lower priority task. Consider two tasks τ_i and τ_j, where $\tau_i \succ \tau_j$, which need a critical section S. Task τ_j inherits the priority of τ_i as long as it blocks τ_i. When τ_j exits the critical section that caused the block, it reverts to the priority it had when it entered that section.

Although the Priority Inheritance Protocol prevents unbounded blocking of a higher priority task by a lower priority task, it does not guarantee that mutual deadlocks will not occur. It also suffers from the possibility of *chained blocking*, which happens because a high priority task is

likely to be blocked whenever it wants to enter a critical section. If the task has several critical sections, it can be blocked for a considerable amount of time [22].

The *Priority Ceiling Protocol* is another protocol that can be used to prevent a medium priority task from preempting the lower priority task [32, 60, 15, 14, 37, 50]. Also, under this protocol, deadlocks cannot occur and a task can be blocked at most once by a lower priority task. In this protocol, when a task tries to hold a resource, the resource is made available only if the resource is free, and only if the priority of the task is greater than or equal to the current highest priority ceiling in the system. Such a rule can cause early blockings in the sense that a task can be blocked even if the resource it wants to access is free. This access rule guarantees that any possible future task is blocked at most once by the lower priority task, which is currently holding a resource, and for a duration of at most B, where B is defined as the greatest execution time of any critical section used by the lower priority task [32, 60].

The *Priority Ceiling Emulation*, which is a combination of the two previous methods, has been introduced to avoid chained blocking and mutual deadlocks. With this method, the priority of a low priority task is raised high enough to prevent it being preempted by a medium priority task. To accomplish this, the highest priority of any task that will lock a resource is kept as an attribute of that resource. Whenever a task is granted access to that resource, its priority is temporarily raised to the maximum priority associated with the resource. When the task has finished with the resource, the task is returned to its original priority.

5 Conclusions and Open Problems

5.1 Summary and Conclusions

A real time system is a system that must satisfy explicit bounded response-time constraints, otherwise risk severe consequences including failure. Failure happens when a system cannot satisfy one or more of the requirements laid out in the formal system specification.

For a given set of tasks the general scheduling problem asks for an order according to which the tasks are to be executed such that various constraints are satisfied. For a given set of real-time tasks, we are asked to devise a feasible allocation/schedule. The release time, the deadline and the execution time of the tasks are some of the parameters that should be considered for scheduling. The deadline may be hard, soft or firm. Other issues to be considered are as follows. Sometimes, a resource must be exclusively held by a task. Tasks may have precedence constraints. A task may be periodic, aperiodic, or sporadic. The schedule may be preemptive or non-preemptive. Less critical tasks must be allowed to be preempted by higher critical ones when it is necessary to meet deadlines. For the real-time systems in which tasks arrive extensively we have to use more than one processor to guarantee that tasks are feasibly scheduled. Therefore, the number of available processors is another parameter to consider. The available

processors may be identical, uniform or unrelated.

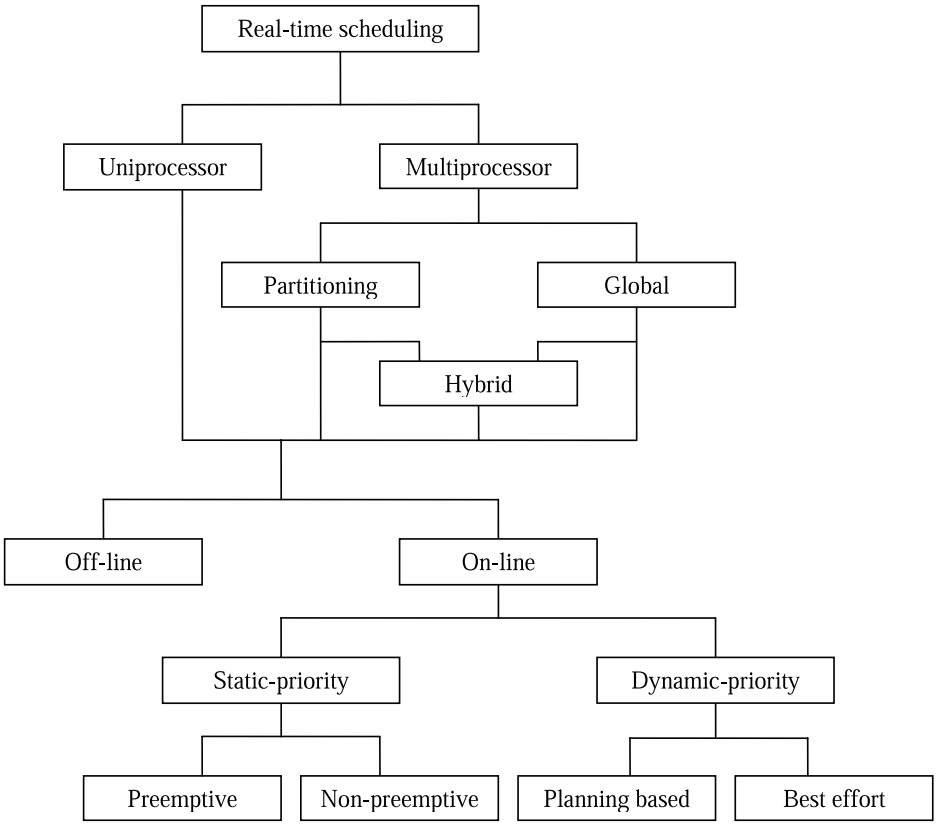

Figure 3: Real-time scheduling algorithms

In this paper, the concept of real-time systems and the characteristics of real-time tasks are described. Also, the concept of utilization bound and optimality criteria, which leads to design appropriate scheduling algorithms, are addressed. The techniques to handle aperiodic and periodic tasks, precedence constraints, and priority inversion are explained. Scheduling of real-time systems is categorized and a description for each class of algorithms is provided. Also, some algorithms are presented to clarify the different classes of algorithms. For real-time multiprocessor systems, we discuss the main strategies, namely partitioning and global strategies, to allocate/schedule the real-time tasks upon the processors. The different classes of the real-time scheduling algorithms studied in this paper are summarized in Figure 3. The techniques studied in Chapter 4 can be adopted to many algorithms in various classes of real-time scheduling algorithms. Some of the uniprocessor real-time scheduling algorithms are illustrated using examples in the Appendix.

Scheduling algorithms for real-time systems have been studied extensively. This paper does not cover all the existing real-time scheduling algorithms. We have not discussed subjects such as fault-tolerant real-time scheduling algorithms and scheduling of reward functions [32]. Also, we have not mentioned time complexity issues and many major theorems about the feasibility and optimality conditions of the real-time scheduling algorithms. There exist many approximation algorithms for real-time systems (see, for example, [64, 8, 9, 18, 40]) that we did not have an opportunity to discuss in this paper. We tried to present the main ideas and classes of real-time scheduling. This paper is organized such that a computer scientist who is not familiar with real-time scheduling, can obtain enough knowledge about this area to be able to analyze and categorize any real-time scheduling problem.

5.2 Open Problems

As we mentioned earlier, there are two main strategies to deal with multiprocessor scheduling problems: partitioning strategy and global strategy, each of which has its advantages and disadvantages. Real-time scheduling problems for multiprocessor systems have mostly been studied for simple system models. Little work has been done on more complex systems. In this section, we provide a list of multiprocessor real-time scheduling problems that require further research. For each problem, scheduling algorithms are to be developed that may fall into either of the above strategies. In addition, designing a suitable hybrid partitioning/global scheduling algorithm, one can take advantage of both methods. Providing suitable hybrid scheduling algorithms that yield the best solutions for each of the following problems is one of the interesting areas of research.

A list of open problems for multiprocessor real-time scheduling is as follows.

Consider a set of hard, soft, and firm real-time tasks, $T = \{\tau_1, \tau_2, ..., \tau_n\}$, where the worst case execution time of each task $\tau_i \in T$ is C_i.

(1) If the real-time tasks are hard, periodic, preemptive and have fixed priorities, then find the minimum number of the processors required to guarantee that all deadlines are met. Some heuristic algorithms have already been proposed, however we believe better algorithms with improved performance can be developed.

(2) Suppose in a system consisting of m identical processors, real-time tasks are preemptive and have fixed priorities. Hard real-time tasks are periodic. Communication cost is negligible. Find a schedule that minimizes mean response time while guaranteeing that all deadlines are met.

(3) Suppose there exist m identical processors, real-time tasks are preemptive and have fixed priorities, a penalty function $P(\tau_i)$ is assigned to each soft real-time task, and a reward

function $R(\tau_i)$ is determined for each firm real-time task. Communication cost is negligible. Find a schedule that guarantees all deadlines are met and $\frac{P(T)}{R(T)}$ is minimized.

(4) Consider the conditions of problem (3), except that communication cost is non-trivial. Give a schedule that minimize the communication cost. Minimizing the number of migrations is one way to reduce the communication cost.

(5) Suppose there exist m identical processors, real-time tasks are aperiodic, preemptive, and have fixed priorities. Communication cost is negligible. Find a schedule that not only guarantees that all deadlines are met, but also minimizes mean response time. Find the utilization bound of the algorithm.

(6) Consider the conditions of problem (5), except that tasks are non-preemptive. Find a schedule that not only guarantees that all deadlines are met, but also minimizes mean response time. Find the utilization bound of the algorithm.

(7) Solve all of the previous problems, i.e, problems (1)-(6), when the tasks are dynamic priority tasks.

(8) Solve all of the previous problems, i.e, problems (1)-(7), when the processors are uniform.

We may apply either of the following approaches to solve each of the above problems:

- The vast majority of the optimization allocating/scheduling problems on real-time systems with more than two processors are NP-hard. In those cases where the problems listed above are NP-hard, one of the following approaches could be used.

  (a) Since the problem is NP-hard, one should strive to obtain a polynomial-time guaranteed-approximation algorithm. Indeed, for some scheduling problems, a heuristic algorithm may be found that runs in polynomial time in the size of the problem and delivers an approximate solution whose ratio to the optimal solution is guaranteed to be no larger than a given constant or a certain function of the size of the problem. However, for most NP-hard problems guaranteeing such an approximate solution is itself an NP-complete problem. In this case, the amount of improvement of the heuristic algorithm with respect to the existing algorithms should be measured via simulation.

  A challenging problem in real-time systems theory is calculating the utilization bounds associated with each allocation/schedule algorithm. The obtained utilization bound allows not only to test the schedulability of any given task set for the

scheduling algorithm, but also it allows to quantify the effect of certain parameters such as the number of the processors, the size of the tasks, and the number of preemptions on schedulability. Calculation of utilization bounds of multiprocessor scheduling for real-time systems is one of the major research directions that should be further investigated.

(b) If we reduce the scheduling problem into a known NP-complete problem A, such as bin-packing or discrete knapsack problem, the existing approximation algorithms for problem A can be applied to the scheduling problem.

- Consider each of the aforementioned problems. The second possibility is developing a polynomial time algorithm that provides an optimal feasible schedule for the problem. The optimality of the algorithm should be proved. We must prove that the algorithm may fail to meet a deadline only if no other scheduling algorithm can meet the deadline. In order to prove optimality, we need to have the utilization bounds associated with the algorithm. The utilization bounds enable an admission controller to decide whether an incoming task can meet its deadline based on utilization-related metrics. In fact, the utilization bounds express the sufficient conditions required for feasibility of the algorithm.

References

[1] B. Andersson and J. Jonsson, *"The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50 percent,"* 15th Euromicro Conference on Real-Time Systems (ECRTS'03), Porto, Portugal, July 02-04, 2003.

[2] J. Anderson and A. Srinivasan, *"Early release fair scheduling,"* In Proceedings of the EuroMicro Conference on Real-Time Systems, IEEE Computer Society Press, pp. 35-43, Stockholm, Sweden, June 2000.

[3] N. Audsley, A. Burns, M. Richardson, K. W. Tindell, and A. J. Wellings, *"Applying new scheduling theory to static priority preemptive scheduling,"* Software Engineering Journal, pp. 284-292, 1983.

[4] H. Aydin, P. Mejia-Alvarez, R. Melhem, and D. Mosse, *"Optimal reward-based scheduling of periodic real-time tasks,"* In Proceedings of the Real-Time Systems Symposium, IEEE Computer Society Press, Phoenix, AZ, December, 1999.

[5] J. W. de Bakker, C. Huizing, W. P. de Roever and G. Rozenberg, *"Real-Time: Thory in Practice,"* Preceedings of REX Workshop, Mook, The Netherlands, Springer-Verlag company, June 3-7, 1991.

[6] J. M. Bans, A. Arenas, and J. Labarta, *"Efficient Scheme to Allocate Soft-Aperiodic Tasks in Multiprocessor Hard Real-Time Systems,"* PDPTA 2002, pp. 809-815.

[7] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel, *"Proportionate progress: A notion of fairness in resource allocation,"* Algorithmica , Volume 15, Number 6, pp. 600-625, June, 1996.

[8] P. Berman and B. DasGupta, *"Improvements in Throughput Maximization for Real-Time Scheduling,"* Department of Computer Science, Yale University, New Haven, CT 06511, January 31, 2000.

[9] S. A. Brandt, *"Performance Analysis of Dynamic Soft Real-Time Systems,"* The 20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001), April, 2001.

[10] A. Burns, *"Preemptive priority based scheduling: An appropriate engineering approach,"* Technical Report, YCS-93-214, Department of Computer Science, university of York, UK, 1993.

[11] A. Burns, *"Scheduling hard real-time systems: A review,"* Software Engineering Journal, Number 5, May, 1991.

[12] G. C. Buttazzo, *"Hard Real-Time Computing Systems: predictable scheduling algorithms and applications,"* Springer company, 2005.

[13] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, *"A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms,"* Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Edited by J. Y. Leung, Published by CRC Press, Boca Raton, FL, USA, 2004.

[14] M. Chen and K. Lin, *"A Priority Ceiling Protocol for Multiple-Instance Resources,"* Proc. of the Real-Time Systems Symposium, 1991.

[15] M. Chen and K. Lin, *"Dynamic Priority Ceiling: A Concurrency Control Protocol for Real-Time Systems,"* Real-Time Systems Journal 2, 1990.

[16] R. K. Clark, *"Scheduling Dependent Real-Time Activities,"* PhD dissertation, Carnegie Mellon Univ., 1990.

[17] L. Cucu, R. Kocik and Y. Sorel, *"Real-time scheduling for systems with precedence, periodicity and latency constraints,"* RTS Embedded Systems 2002, Paris, 26-28 March, 2002.

[18] B. Dasgupta and M. A. Palis, *"Online Real-Time Preemptive Scheduling of Jobs with Deadlines on Multiple Machines,"* Journal of Scheduling, Volume 4, Number 6, pp. 297-312, November, 2001.

[19] D. A. El-Kebbe, *"Real-Time Hybrid Task Scheduling Upon Multiprocessor Production Stages,"* International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, 22-26 April, 2003.

[20] G. Fohler, T. Lennvall, and G. Buttazzo, *"Improved Handling of Soft Aperiodic Tasks in Offline Scheduled Real-Time Systems using Total Bandwidth Server,"* In Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation, Nice, France, October, 2001.

[21] W. Fornaciari, P. di Milano, *"Real Time Operating Systems Scheduling Lecturer,"* www.elet elet.polimi polimi.it/ fornacia it/ fornacia.

[22] K. Frazer, *"Real-time Operating System Scheduling Algorithms,"* , 1997.

[23] S. Funk, J. Goossens, and S. Baruah, *"On-line Scheduling on Uniform Multiprocessors,"* , 22nd IEEE Real-Time Systems Symposium (RTSS'01), pp. 183-192, London, England, December, 2001.

[24] M. Garey, D. Johnson, *"Complexity Results for Multiprocessor Scheduling under Resource Constraints,"* SICOMP, Volume 4, Number 4, pp. 397-411, 1975.

[25] R. Gerber, S. Hong and M. Saksena, , *"GuaranteeingReal-Time Requirements with Resource-Based Calibrationof Periodic Processes,"* IEEE Transactions on Software Engineering, Volume 21, Number 7, July, 1995.

[26] J. Goossens and P. Richard, *"Overview of real-time scheduling problems,"* Euro Workshop on Project Management and Scheduling, 2004.

[27] W. A. Halang and A. D. Stoyenko, *"Real Time Computing,"* NATO ASI Series, Series F: Computer and Systems Sciences, Volume 127, Springer-Verlag company, 1994.

[28] P. Holman and J. H. Anderson, *"Using Supertasks to Improve Processor Utilization in Multiprocessor Real-Time Systems,"* 15th Euromicro Conference on Real-Time Systems (ECRTS'03), Porto, Portugal, 2-4 July, 2003.

[29] D. Isovic and G. Fohler, *"Efficient Scheduling of Sporadic, Aperiodic and Periodic Tasks with Complex Constraints,"* In Proceedings of the 21st IEEE RTSS, Florida, USA, November, 2000.

[30] M. Joseph, *"Real-time Systems: Specification, Verification and Analysis,"* Prentice Hall, 1996.

[31] S. Kodase, S. Wang, Z. Gu and K. G. Shin, *" Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-Time Systems Using Shared Buffers,"* The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 181-188, 2003.

[32] C. M. Krishna and K. G. Shin, *"Real-Time Systems,"* MIT Press and McGraw-Hill Company, 1997.

[33] P. A. Laplante, *"Real-time Systems Design and Analysis, An Engineer Handbook,"* IEEE Computer Society, IEEE Press, 1993.

[34] S. Lauzac and R. Melhem, *"An Improved Rate-Monotonic Admission Control and Its Applications,"* IEEE Transactions on Computers, Volume 52, Number 3, pp. 337-350, March, 2003.

[35] J. Y.-T. Leung and J. Whitehead, *"On the complexity of fixed priority scheduling of periodic real-time tasks,"* Performance Evaluation, Volume 2, pp. 237-250, 1982.

[36] P. Li and B. Ravindran, *"Fast, Best-Effort Real-Time Scheduling Algorithms,"* IEEE Transactions on Computers, Volume 53, Number 9, pp. 1159-1175, September, 2004.

[37] C. L. Liu and J. W. Layland, *"Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment,"* Journal of the ACM , Volume 20, Number 1, pp. 46-61, 1973.

[38] C. D. Locke, *"Best-Effort Decision Making for Real-Time Scheduling,"* PhD dissertation, Carnegie Mellon University, 1986.

[39] J. Luo and N. K. Jha, *"Power-conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-time Embedded Systems,"* Proceedings of ICCAD, pp. 357364, November, 2000.

[40] G. Manimaran and C. S. Ram Murthy, *"An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems,"* IEEE Transaction Parallel and Distributed Systems, Volume 9, Number 3, pp. 312-319, March, 1998.

[41] F. W. Miller, *"the Performance of a Mixed Priority Real-Time Scheduling Algorithm,"* Operating System Review, Volume 26, Number 4, pp. 5-13, October, 1992.

[42] M. Moir and S. Ramamurthy, *"Pfair scheduling of fixed and migrating tasks on multiple resources,"* In Proceedings of the Real-Time Systems Symposium, IEEE Computer Society Press, Phoenix, AZ, December, 1999.

[43] A. K. Mok, *"Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment,"* Technical Report, Massachusetts Institute of Technology, June, 1983.

[44] A. K. Mok, *"Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment,"* Ph.D. thesis. Department of Electronic Engineering and Computer Sciences, Mass. Inst. Technol., Cambridge MA, May, 1983.

[45] C. Perkins, *"Course Notes: Overview of Real-Time Scheduling, Real-Time and Embedded Systems (M) Lecture 3,"* University of Glasgow, Department of Computing Science 2004-2005 Academic Year.

[46] C. A. Phillips, C. Stein, E. Torng, and J. Wein, *"Optimal time-critical scheduling via resource augmentation,"* In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 140-149, El Paso, Texas, 4-6 May, 1997.

[47] S. Schneider, *"Concurrent and Real-time systems, The CSP Approach,"* John Wiley and Sons LTD, 2000.

[48] G. Quan, L. Niu, J. P. Davis, *"Power Aware Scheduling for Real-Time Systems with (m; k)-Guarantee,"* CNDS, 2004.

[49] , K. Sandstrm and C. Norstrm, " *Managing Complex Temporal Requirements in Real-Time Control Systems*," The 9th IEEE Conference on Engineering of Computer-Based Systems, pp. 81-84, Sweden, 2002.

[50] L. Sha, R. Rajkumar and J. P. Lehoczky, "*Priority Inheritance Protocol; an Approach to Real-Time Synchronization*," IEEE Transactions on Computers, Volume 39, Number 9, 1990.

[51] K. G. Shin and Y. Chang, "*A Reservation-Based Algorithm for scheduling Both Periodic and Aperiodic Real-Time Tasks*," IEEE Transactions on Computers, Volume 44, Number 12, pp. 1405-1419, December, 1995.

[52] H. Singh, "*Scheduling Techniques for real-time applications consisting of periodic task sets*," In Proceedings of the IEEE Workshop on Real-Time Applications, pp. 12-15 , 21-22 July, 1994.

[53] B. Sprunt, "*Aperiodic Task Scheduling for Real-Time Systems*," Ph.D. Thesis, Department of Electrical and Computer Engineering Carnegie Mellon University, August, 1990.

[54] B. Sprunt, J. Lehoczky, and L. Sha, "*Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm*," In Proceedings of the 9th Real-Time Systems Symposium, pp. 251-258. IEEE, Huntsville, AL, December, 1988.

[55] M. Spuri and G. C. Buttazzo, "*Efficient Aperiodic Service under Earliest Deadline Scheduling*," In Proceedings IEEE Real-Time Systems Symposium, pp. 2-11, San Juan, Puerto Rice, 7-9 December, 1994.

[56] M. Spuri and G. Buttazzo, "*Scheduling Aperiodic Tasks in Dynamic Priority Systems*," The Journal of Real-Time Systems.

[57] M. Spuri, G. Buttazzo, nd F. Sensini, "*Robust Aperiodic Scheduling under Dynamic Priority Systems*," In Proceedings IEEE Real-Time Systems Symposium, pp. 210-219, Pisa, Italy, 5-9 December, 1995.

[58] M. Spuri and J. A. Stankovic, "*How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling*," IEEE Transactions on Computers, Volume 43, Number 12, pp. 1407-1412, December, 1994.

[59] J. A. Stankovic and K. Ramamritham, , "*Tutorial Hard Real-Time Systems*," IEEE Computer Society Press, 1988.

[60] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *"Deadline Scheduling for Real-Time Systems, EDF and related algorithms,"* Kluwer Academia Publishers, 1998.

[61] T. Tia, J. W. Liu, and M. Shankar, *"Algorithms and Optimality of Scheduling Soft Aperiodic Request in Fixed Priority Preemptive Systems,"* The Journal of Real-Time Systems, Volume 10, Number 1, pp. 23-43, January, 1996.

[62] J. Wang, B. Ravindran, and T. Martin, *"A Power-Aware, Best-Effort Real-Time Task Scheduling Algorithm,"* IEEE Workshop on Software Technologies for Future Embedded Systems p. 21.

[63] Z. Xiangbin and T. Shiliang, *"An improved dynamic scheduling algorithm for multiprocessor real-time systems,"* PDCAT'2003. In Proceedings of the Fourth International Conference on Publication, pp. 710- 714, 27-29 August, 2003.

[64] M. Xiong, K.-Y. Lam and B. Liang, *"Quality of Service Gaurantee for Temporal Consistency of Real-Time Objects,"* The 24th IEEE Real-time System Symposium (RTSS2003), Cancun, Mexico, December, 2003.

[65] http://www.netrino.com/Publications/Glossary/PriorityInversion.html

[66] http://www.ee.umd.edu/serts/bib/thesis/dstewart2.pdf

[67] http://www-2.cs.cmu.edu/afs/cs/project/jair/pub/volume4/hogg96a-html/node2.html

[68] http://www.cs.pitt.edu/ melhem/courses/3530/L1.pdf

[69] http://www.omimo.be/encyc/publications/faq/rtfaq.htm

[70] http://c2.com./cgi/wiki?RealTime

Appendix: Examples

In this chapter we present the timing diagrams of the schedules provided by some real-time scheduling algorithms, namely the earliest deadline first (EDF), the rate-monotonic (RM), and the least laxity first (LLF) algorithms, on two given sets of tasks.

	Period	*Computation time*	*First invocation time*	*Deadline*
τ_1	2	0.5	0	2
τ_2	6	2	1	6
τ_3	10	1.8	3	10

Table 2: The repetition periods, computation times, and deadlines of the tasks τ_1, τ_2 and τ_3 for Example A.1

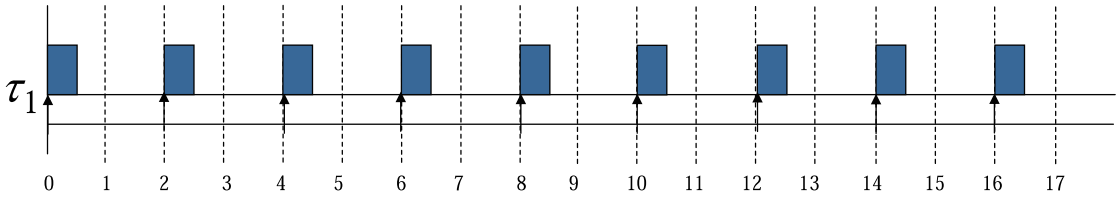

Figure 4: The timing diagram of task τ_1 defined in Table 2, before scheduling

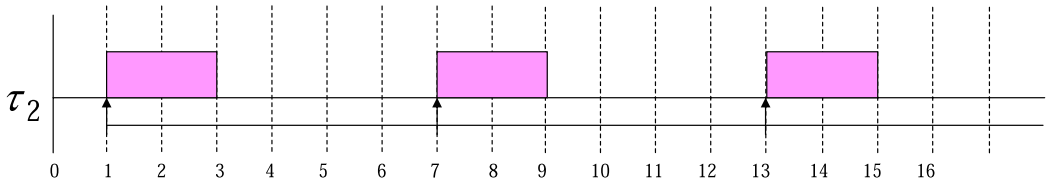

Figure 5: The timing diagram of task τ_2 defined in Table 2, before scheduling

Example A.1: Consider a system consisting of three tasks τ_1, τ_2 and τ_3, that have the repetition periods, computation times, the first invocation times and deadlines defined in Table 2. The deadline D_i of each task τ_i is P_i and tasks are preemptive. Figures 4, 5 and 6 present the timing diagram of each task τ_1, τ_2 and τ_3, respectively, before scheduling.

- *Earliest deadline first algorithm*

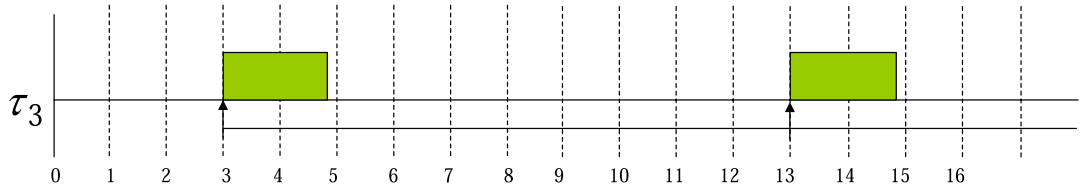

Figure 6: The timing diagram of task τ_3 defined in Table 2, before scheduling

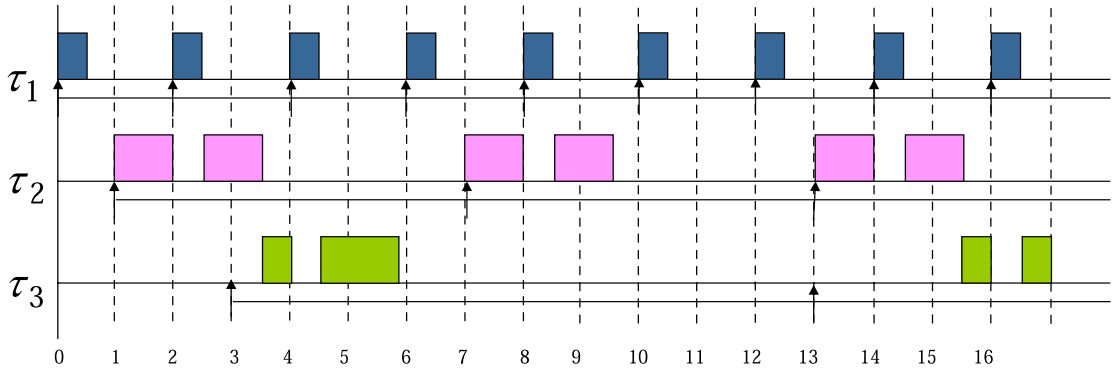

Figure 7: The timing diagram of the schedule provided by any of the earliest deadline first, rate monotonic, least laxity first algorithms on the tasks set defined in Table 2

Figure 7 presents a portion of the timing diagram of the schedule provided by the EDF algorithm on the tasks set defined in Table 2. Between time interval 0 and 17 we observe that no deadline is missed.

- *Rate monotonic algorithm*

  As shown in Figure 7, if we schedule the tasks set by the RM algorithm, no deadline is missed between time interval 0 and 17.

- *Least laxity first algorithm*

  Similar to the previous two scheduling algorithms, the least laxity first algorithm provides a schedule such that all deadlines are met between time interval 0 and 17 (see Figure 7).

For Example A.1, the timing diagrams of the schedules provided by the earliest deadline first, rate monotonic, and least laxity first algorithms happen to be the same, as indicated in Figure 7.

	Period	*Computation time*	*First invocation time*	*Deadline*
τ_1	2	0.5	0	2
τ_2	6	4	1	6
τ_3	3	1.8	3	10

Table 3: The repetition periods, computation times and deadlines of the tasks τ_1, τ_2 and τ_3 for Example A.2

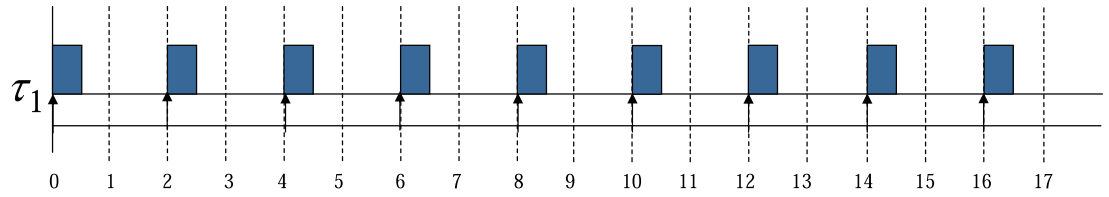

Figure 8: The timing diagram of task τ_1 defined in Table 3, before scheduling

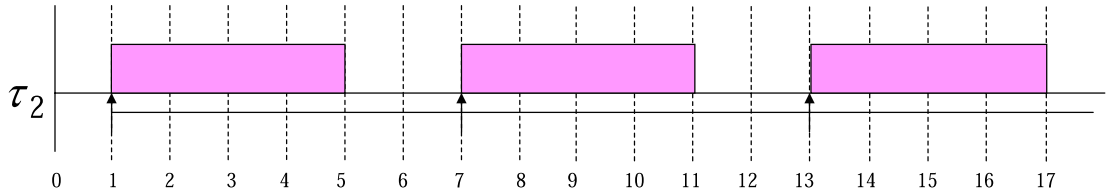

Figure 9: The timing diagram of task τ_2 defined in Table 3, before scheduling

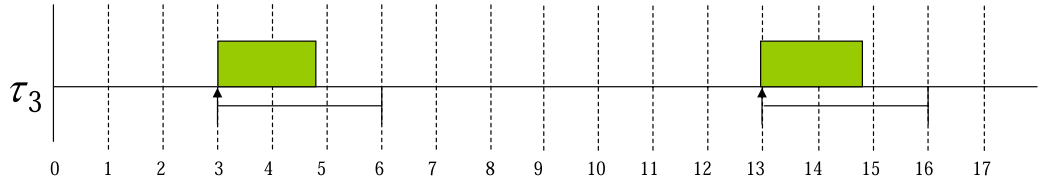

Figure 10: The timing diagram of task τ_3 defined in Table 3, before scheduling

Example A.2: Consider a system consisting of three tasks τ_1, τ_2 and τ_3, that have the repetition periods, computation times, first invocation times and deadlines defined in Table 3. The tasks are preemptive. The timing diagrams in Figures 8, 9 and 10 present the timing diagram of each task τ_1, τ_2 and τ_3, respectively, before scheduling.

- *Earliest deadline first algorithm*

  As presented in Figure 11, the uniprocessor real-time system consisting of the tasks set defined in Table 3 is not EDF-schedulable, because while the execution of the first invocation of the task τ_2 is not finished yet, the new invocation of the task arrives. In other words, an overrun condition happens.

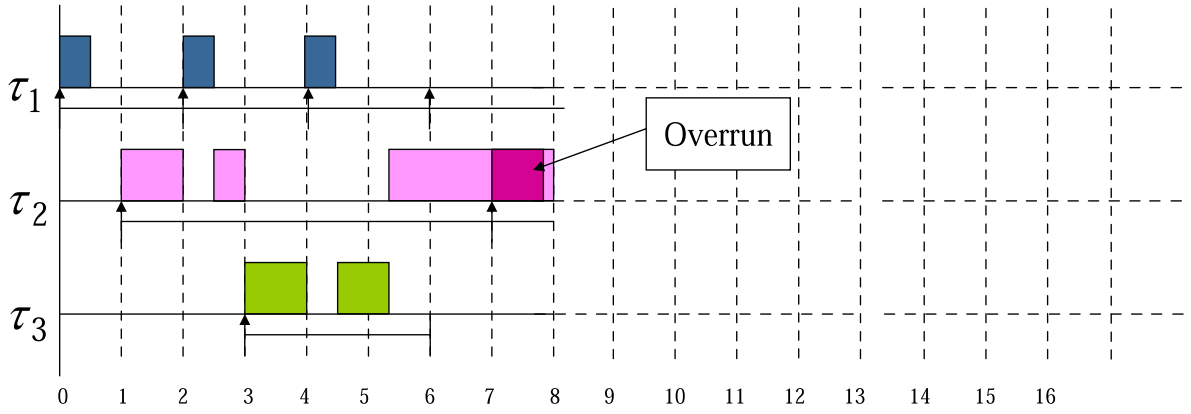

Figure 11: The timing diagram of the schedule provided by the earliest deadline algorithm on the tasks set defined in Table 3

- *Rate monotonic algorithm*

  As shown in Figure 12, the uniprocessor real-time system consisting of the tasks set defined in Table 3 is not RM-schedulable. The reason is that the deadline of the first invocation of the task τ_3 is missed. The execution of the first invocation is required to be finished by time 6, but the schedule could not make it.

- *Least laxity first algorithm*

  Figure 13 presents a portion of the timing diagram of the schedule provided by the least laxity first algorithm on the tasks set defined in Table 3. As shown in the figure, the deadline of the third invocation of the task τ_1 can not be met. we conclude that the uniprocessor real-time system consisting of the tasks set defined in Table 3 is not LLF-schedulable.

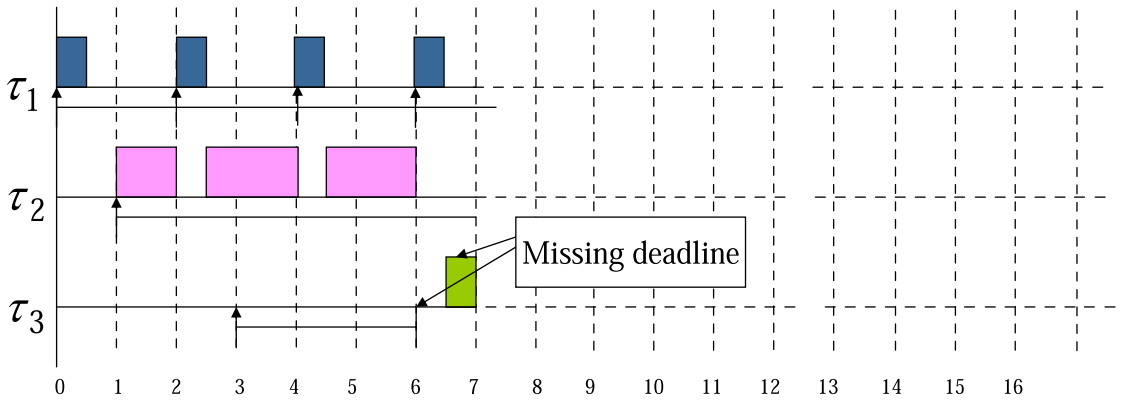

Figure 12: The timing diagram of the schedule provided by the rate monotonic algorithm on the tasks set defined in Table 3

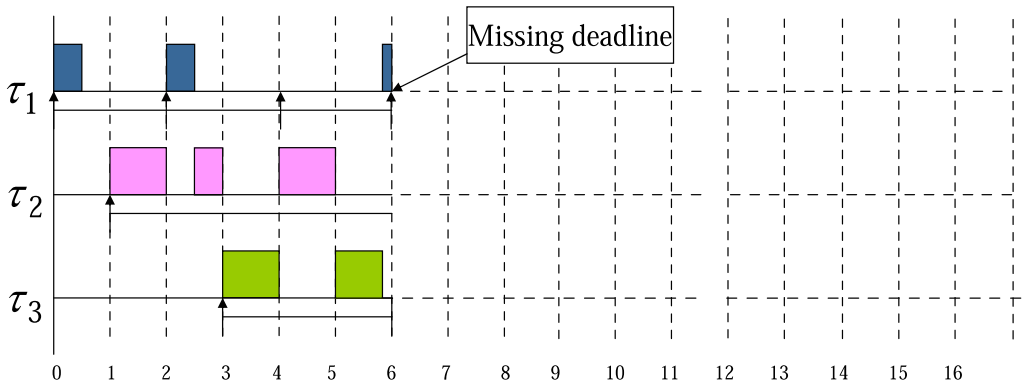

Figure 13: The timing diagram of the schedule provided by the least laxity first algorithm on the tasks set defined in Table 3