Giorgio Buttazzo

Hard Real-Time Computing Systems

Predictable Scheduling Algorithms and Applications

Fourth Edition

Springer

Hard Real-Time Computing Systems

Giorgio Buttazzo

Hard Real-Time Computing Systems

Predictable Scheduling Algorithms and Applications

Fourth Edition

Springer

Giorgio Buttazzo
Sant'Anna School of Advanced Studies
Pisa, Italy

ISBN 978-3-031-45409-7 ISBN 978-3-031-45410-3 (eBook)
https://doi.org/10.1007/978-3-031-45410-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Real-time computing plays a crucial role in our society since an increasing number of complex systems rely, in part or completely, on computer control. Examples of applications that require real-time computing include nuclear power plants, railway switching systems, automotive and avionic systems, air traffic control, telecommunications, robotics, and military systems. In the last years, real-time computing was also demanded in new applications areas, such as medical equipment, consumer electronics, multimedia systems, flight simulation systems, virtual reality, and interactive games.

In spite of this large application domain, most of the current real-time systems are still designed and implemented using low-level programming and empirical techniques, without the support of a scientific methodology. This approach results in a lack of reliability, which in critical applications may cause serious environmental damage or even loss of life.

This book is a basic treatise on real-time computing, with particular emphasis on predictable scheduling algorithms. The main objectives of the book are to introduce the basic concepts of real-time computing, illustrate the most significant results in the field, and provide the basic methodologies for designing predictable computing systems useful in supporting critical control applications.

The book is written for instructional use and is organized to enable readers without a strong knowledge of the subject matter to quickly grasp the material. Technical concepts are clearly defined at the beginning of each chapter, and algorithm descriptions are corroborated through concrete examples, illustrations, and tables.

Contents of the Chapters

Chapter 1 presents a general introduction to real-time computing and real-time operating systems. It introduces the basic terminology and concepts used in the

book, discusses the typical application domains, and clearly illustrates the main characteristics that distinguish real-time processing from other types of computing.

Chapter 2 introduces the general problem of scheduling a set of tasks on a uniprocessor system. Objectives, performance metrics, and hypotheses are clearly presented, and the algorithms proposed in the literature are classified in a taxonomy, which provides a useful reference framework for understanding the various approaches. At the end of the chapter, a number of scheduling anomalies are illustrated to show that real-time computing is not equivalent to fast computing.

The rest of the book is dedicated to specific scheduling algorithms, which are presented as a function of the task characteristics.

Chapter 3 introduces a number of real-time scheduling algorithms for handling aperiodic tasks with explicit deadlines. Each algorithm is examined in regard to the task set assumptions, formal properties, performance, and implementation complexity.

Chapter 4 treats the problem of scheduling a set of real-time tasks with periodic activation requirements. In particular, three classical algorithms are presented in detail: Rate Monotonic, Earliest Deadline First, and Deadline Monotonic. A schedulability test is derived for each algorithm.

Chapter 5 deals with the problem of scheduling hybrid sets consisting of hard periodic and soft aperiodic tasks, in the context of fixed-priority assignments. Several algorithms proposed in the literature are analyzed in detail. Each algorithm is compared with respect to the assumptions made on the task set, its formal properties, its performance, and its implementation complexity.

Chapter 6 considers the same problem addressed in Chap. 5, but in the context of a dynamic priority assignment. Performance results and comparisons are presented at the end of the chapter.

Chapter 7 introduces the problem of scheduling a set of real-time tasks that may interact through shared resources and hence have both time and resource constraints. Three resource access protocols are described in detail: the Priority Inheritance Protocol, the Priority Ceiling Protocol, and the Stack Resource Policy. These protocols are essential for achieving predictable behavior, since they bound the maximum blocking time of a process when accessing shared resources.

Chapter 8 is dedicated to non-preemptive and limited-preemptive scheduling, often used in industrial applications to make task execution more predictable and reduce the run time overhead introduced by arbitrary preemptions. Different solutions are presented, analyzed, and compared in terms of implementation complexity, predictability, and efficacy.

Chapter 9 deals with the problem of real-time scheduling during overload conditions; that is, those situations in which the total processor demand exceeds the available processing time. These conditions are critical for real-time systems, since not all tasks can complete within their timing constraints. This chapter introduces new metrics for evaluating the performance of a system and presents a new class of scheduling algorithms capable of achieving graceful degradation in overload conditions.

Chapter 10 describes some basic guidelines that should be considered during the design and the development of a hard real-time kernel for critical control applications. An example of a small real-time kernel is presented. The problem of time predictable inter-task communication is also discussed, and a particular communication mechanism for exchanging asynchronous messages among periodic tasks is illustrated. The final section shows how the runtime overhead of the kernel can be evaluated and taken into account in the guarantee tests.

Chapter 11 discusses some crucial issues related to the design of real-time applications. A few robot control examples are presented to illustrate how time constraints can be derived from the application requirements, even though they are not explicitly specified by the user. Then, the kernel primitives presented in Chap. 10 are used to illustrate a concrete programming example of real-time tasks for sensory processing and control activities. Finally, some considerations are presented to explain how to improve predictability and safety in cyber-physical real-time systems that exploit machine learning algorithms.

Chapter 12 illustrates how to develop a C library to simplify the implementation of real-time periodic tasks in Linux, thus hiding many of the low-level details that are visible when programming at the operating system level. Such an abstraction layer is built on top of the Pthread library and allows the user to create periodic tasks with given periods and deadlines, manage time in a more practical way, and check for deadline misses.

Chapter 13 concludes the book by presenting a number of real-time operating systems, including standard interfaces (like RT-Posix, APEX, OSEK, and Micro-ITRON), commercial operating systems (like VxWorks, QNX, OSE), and open source kernels (like Shark, Erika, Marte, and Linux real-time extensions).

Difference with the Third Edition

This book contains several changes and additions with respect to the previous edition. Several parts have been added to illustrate the most recent methods proposed in the real-time literature, mistakes and typos have been corrected, and some concepts have been further clarified, based on the observations received from readers.

The most significant additions are the following:

- In Chap. 4 on periodic task scheduling, several improvements of the Response Time Analysis have been included, in order to reduce the number of iterations of the algorithm and speed up the schedulability test. Then, an approximated method based on a response time upper bound is presented to derive a sufficient schedulability test with linear time complexity. Finally, Sect. 4.5.3 on Workload Analysis has been extended to explain the method more clearly using examples and figures.

- In Chap. 7 on resource access protocols, a more precise method for computing the worst-case blocking times is presented and illustrated with an example.
- In Chap. 9 on handling overload conditions, the BROE protocol is presented to manage the budget exhaustion in reservation servers that share resources with other tasks. Also, a new section has been added to explain the variable-rate task model, used to dynamically adapt the computational load in applications where task activation rates depend on a physical system variable, as for engine control tasks in automotive applications.
- In Chap. 11 on application design issues, a new section has been added to explain how to improve predictability and safety in cyber-physical real-time systems that exploit machine learning algorithms.
- Chapter 12 is new and has been added to present a library, built on top of the Pthread library, for implementing periodic real-time tasks under Linux.
- The bibliography has been updated with additional references.

Acknowledgments

This work is the result of over 30 years of research and teaching activity in the field of real-time systems. The majority of the material presented in this book is based on class notes for an operating systems course taught at the University of Pisa, University of Pavia, and Sant'Anna School of Advanced Studies.

Though this book carries the name of a single author, it has been positively influenced by a number of people to whom I am indebted. Foremost, I would like to thank all my students, who contributed to improve its readability and clarity.

A personal note of appreciation goes to Paolo Ancilotti, who at the beginning of my career gave me the opportunity to teach these topics in his course on operating systems. Moreover, I would like to acknowledge the contributions of Jack Stankovic, Krithi Ramamritham, Herman Kopetz, John Lehoczky, Lui Sha, Alan Burns, Gerhard Fohler, Sanjoy Baruah, and Gerard Le Lann. Their inputs enhanced the overall quality of this work. I would also like to thank the Springer editorial staff for the support I received during the preparation of the manuscript.

Special appreciation goes to Marco Spuri and Fabrizio Sensini, who gave a substantial contribution to the development of dynamic scheduling algorithms for aperiodic service; Benedetto Allotta, who worked with me in approaching some problems related to control theory and robotics applications; Luca Abeni, for his contribution on resource reservation; Giuseppe Lipari and Marco Caccamo, for their work on resource sharing in dynamic scheduling; and Enrico Bini, for his novel approach on the analysis of fixed priority systems.

I also wish to thank Antonino Gambuzza, Marco Di Natale, Giacomo Borlizzi, Stefano Petrucci, Enrico Rebaudo, Fabrizio Sensini, Gerardo Lamastra, Giuseppe Lipari, Antonino Casile, Fabio Conticelli, Paolo Della Capanna, and Marco Caccamo, who gave a valuable contribution to the development of the HARTIK

real-time kernel, and Claudio Scordino for his contribution in the description of Linux-related kernels.

A special thanks goes to Paolo Gai, who was the major developer of two real-time kernels, SHARK and ERIKA, used for many years as educational kernels in several real-time courses around the world, and to all the guys that have been involved in its maintenance, as Tullio Facchinetti, Mauro Marinoni, and Gianluca Franchino.

Finally, I would like to thank Alessandro Biondi for his exceptional work on modeling and analysis of rate adaptive tasks, and my PhD students, Federico Nesti, Giulio Rossolini, and Fabio Brau, for their essential contributions to a new research direction aimed at enhancing the safety and the security of AI-powered cyber-physical systems.

Pisa, Italy Giorgio Buttazzo

Contents

Chapter 1
A General View

1.1 Introduction

Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [SR88]. A reaction that occurs too late could be useless or even dangerous. Today, real-time computing plays a crucial role in our society, since an increasing number of complex systems rely, in part or completely, on computer control. Examples of applications that require real-time computing include:

- Chemical and nuclear plant control,
- Control of complex production processes,
- Railway switching systems,
- Automotive applications,
- Flight control systems,
- Environmental acquisition and monitoring,
- Telecommunication systems,
- Medical systems,
- Industrial automation,
- Robotics,
- Military systems,
- Space missions,
- Consumer electronic devices,
- Multimedia systems,
- Smart toys, and
- Virtual reality.

In many cases, the real-time computer running the application is embedded into the system to be controlled. Embedded systems span from small portable devices

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_1

(e.g., cellular phones, cameras, navigators, ECG Holter devices, smart toys) to larger systems (e.g., industrial robots, cars, aircrafts).

Despite this large application domain, many researchers, developers, and technical managers have serious misconceptions about real-time computing [Sta88], and most of today's real-time control systems are still designed using ad hoc techniques and heuristic approaches. Very often, control applications with stringent time constraints are implemented by writing large portions of code in assembly language, programming timers, writing low-level drivers for device handling, and manipulating task and interrupt priorities. Although the code produced by these techniques can be optimized to run very efficiently, this approach has the following disadvantages:

- **Tedious programming**. The implementation of large and complex applications in assembly language is much more difficult and time-consuming than high-level programming. Moreover, the efficiency of the code strongly depends on the programmer's ability.
- **Difficult code understanding**. Except for the programmers who develop the application, very few people can fully understand the functionality of the software produced. Clever hand-coding introduces additional complexity and makes a program more difficult to comprehend.
- **Difficult software maintainability**. As the complexity of the application software increases, the modification of large assembly programs becomes difficult even for the original programmer.
- **Difficult verification of time constraints**. Without the support of specific tools and methodologies for code and schedulability analysis, the verification of timing constraints becomes practically impossible.

The major consequence of this approach is that the control software produced by empirical techniques can be highly unpredictable. If all critical time constraints cannot be verified a priori and the operating system does not include specific mechanisms for handling real-time tasks, the system could apparently work well for a period of time, but it could collapse in certain rare, but possible, situations. The consequences of a failure can sometimes be catastrophic and may injure people, or cause serious damages to the environment.

A high percentage of accidents that occur in nuclear power plants, space missions, or defense systems are often caused by software bugs in the control system. In some cases, these accidents have caused huge economic losses or even catastrophic consequences, including the loss of human lives.

As an example, the first flight of the space shuttle was delayed, at considerable cost, because of a timing bug that arose from a transient CPU overload during system initialization on one of the redundant processors dedicated to the control of the aircraft [Sta88]. Although the Shuttle control system was intensively tested, the timing error was never discovered before. Later, by analyzing the code of the processes, it has been found that there was only a 1 in 67 probability (about 1.5%) that a transient overload during initialization could push the redundant processor out of synchronization.

Another software bug was discovered on the real-time control system of the Patriot missiles, used to protect Saudi Arabia during the Gulf War.[1] When a Patriot radar sights a flying object, the onboard computer calculates its trajectory, and, to ensure that no missiles are launched in vain, it performs a verification. If the flying object passes through a specific location, computed based on the predicted trajectory, then the Patriot is launched against the target; otherwise, the phenomenon is classified as a false alarm.

On February 25, 1991, the radar sighted a Scud missile directed at Saudi Arabia, and the onboard computer predicted its trajectory, performed the verification, but classified the event as a false alarm. A few minutes later, the Scud fell on the city of Dhahran, causing victims and enormous economic damage. Later on, it was discovered that, because of a long interrupt handling routine running with disable interrupts, the real-time clock of the onboard computer was missing some clock interrupts, thus accumulating a delay of about 57 μs/min. The day of the accident, the computer had been working for about 100 h (an exceptional condition that was never experienced before), thus accumulating a total delay of 343 ms. Such a delay caused a prediction error in the verification phase of 687 m! The bug was corrected on February 26, the day after the accident, by inserting a few preemption points inside the long interrupt handler.

The examples of failures described above show that software testing, although important, does not represent a solution for achieving predictability in real-time systems. This is mainly due to the fact that, in real-time control applications, the program flow depends on input sensory data and environmental conditions, which cannot be fully replicated during the testing phase. As a consequence, the testing phase can provide only a *partial* verification of the software behavior, relative to the particular subset of data provided as input.

A more robust guarantee of the performance of a real-time system under all possible operating conditions can be achieved only by using more sophisticated design methodologies, combined with a static analysis of the source code and specific operating systems mechanisms, purposely designed to support computation under timing constraints. Moreover, in critical applications, the control system must be capable of handling all anticipated scenarios, including peak-load situations, and its design must be driven by pessimistic assumptions on the events generated by the environment.

In 1949, an aeronautical engineer of the US Air Force, Captain Ed Murphy, observed the evolution of his experiments and said: “If something can go wrong, it will go wrong.” Several years later, Captain Ed Murphy became famous around the world, not for his work in avionics but for his phrase, simple but ineluctable, today known as *Murphy’s law* [Blo77, Blo80, Blo88]. Since that time, many other laws on existential pessimism have been formulated to describe unfortunate events in a humorous fashion. Due to the relevance that pessimistic assumptions have on

[1] *L’Espresso*, Vol. XXXVIII, No. 14, 5 April 1992, p. 167.

Table 1.1 Murphy's laws on real-time systems

Murphy's general law
If something can go wrong, it will go wrong.
Murphy's constant
Damage to an object is proportional to its value.
Naeser's law
One can make something bombproof, not jinx-proof.
Troutman postulates
1. *Any software bug will tend to maximize the damage.* 2. *The worst software bug will be discovered 6 months after the field test.*
Green's law
If a system is designed to be tolerant to a set of faults, there will always exist an idiot so skilled to cause a nontolerated fault.
Corollary
Dummies are always more skilled than measures taken to keep them from harm.
Johnson's first law
If a system stops working, it will do it at the worst possible time.
Sodd's second law
Sooner or later, the worst possible combination of circumstances will happen.
Corollary
A system must always be designed to resist the worst possible combination of circumstances.

the design of real-time systems, Table 1.1 lists the most significant laws on the topic, which a software engineer should always keep in mind.

1.2 What Does Real Time Mean?

1.2.1 The Concept of Time

The main characteristic that distinguishes real-time computing from other types of computation is time. Let us consider the meaning of the words *time* and *real* more closely.

The word *time* means that the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

The word *real* indicates that the reaction of the systems to external events must occur *during* their evolution. As a consequence, the system time (internal time) must be measured using the same time scale used for measuring the time in the controlled environment (external time).

Although the term *real time* is frequently used in many application fields, it is subject to different interpretations, not always correct. Often, people say that a control system operates in real time if it is able to *quickly* react to external events. According to this interpretation, a system is considered to be real time if it is fast. The term *fast*, however, has a relative meaning and does not capture the main properties that characterize these types of systems.

In nature, living beings act in real time in their habitat independently of their speed. For example, the reactions of a turtle to external stimuli coming from its natural habitat are as effective as those of a cat with respect to its habitat. In fact, although the turtle is much slower than a cat, in terms of absolute speed, the events that it has to deal with are proportional to the actions it can coordinate, and this is a necessary condition for any animal to survive within an environment.

On the contrary, if the environment in which a biological system lives is modified by introducing events that evolve more rapidly than it can handle, its actions will no longer be as effective, and the survival of the animal is compromised. Thus, a quick fly can still be caught by a flyswatter, a mouse can be captured by a trap, or a cat can be run down by a speeding car. In these examples, the flyswatter, the trap, and the car represent unusual and anomalous events for the animals, out of their range of capabilities, which can seriously jeopardize their survival. The cartoons in Fig. 1.1 schematically illustrate the concept expressed above.

The previous examples show that the concept of time is not an intrinsic property of a control system, either natural or artificial, but it is strictly related to the environment in which the system operates. It does not make sense to design

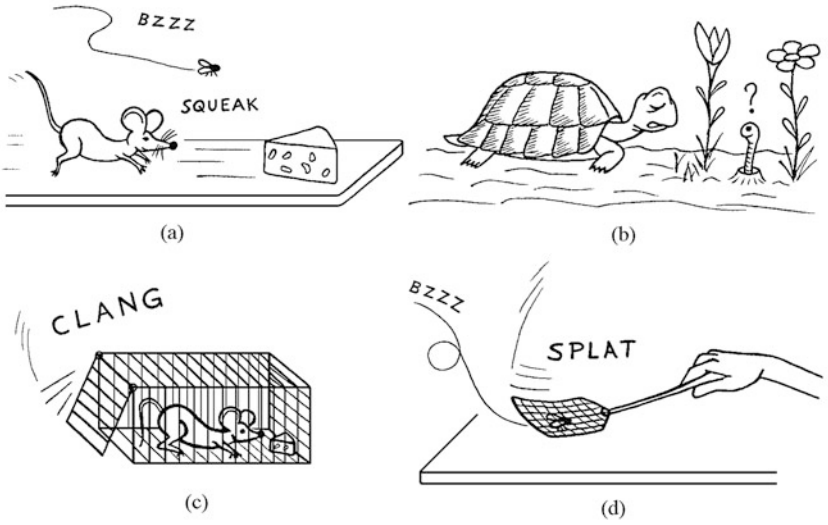


Fig. 1.1 Both the mouse (**a**) and the turtle (**b**) behave in real time with respect to their natural habitat. Nevertheless, the survival of fast animals such as a mouse and a fly can be jeopardized by events (**c** and **d**) quicker than their reactive capabilities

a real-time computing system for flight control without considering the timing characteristics of the aircraft and the environment in which it has to operate.

As an example, the Ariane 5 launcher ended in a failure 37 s after initiation of the flight sequence, because the characteristics of the launcher were not taken into account in the implementation of the control software [Bab97, JM97]. On June 4, 1996, at an altitude of about 3700 m, the launcher started deflecting from its correct path, and a few seconds later it was destroyed by its automated self-destruct system. The failure was caused by an operand error originated in a routine called by the Inertial Reference System for converting accelerometric data from 64-bit floating point to 16-bit signed integer.

One value was too large to be converted, and the program was not explicitly designed to handle the integer overflow error, so the Inertial Reference System halted, as specified in other requirements, leaving the launcher without inertial guidance. The conversion error occurred because the control software was reused from the Ariane 4 vehicle, whose dynamics was different from that of the Ariane 5. In particular, the variable containing the horizontal velocity of the rocket went out of range (since larger than the maximum value planned for the Ariane 4), so generating the error that caused the loss of guidance.

The examples considered above indicate that the environment is always an essential component of any real-time system. Figure 1.2 shows a block diagram of a typical real-time architecture for controlling a physical system.

Some people erroneously believe that it is not worth investing in real-time research because advances in computer hardware will take care of any real-time requirements. Although advances in computer hardware technology will improve system throughput and will increase the computational speed in terms of millions of instructions per second (MIPS), this does not mean that the timing constraints of an application will be met automatically.

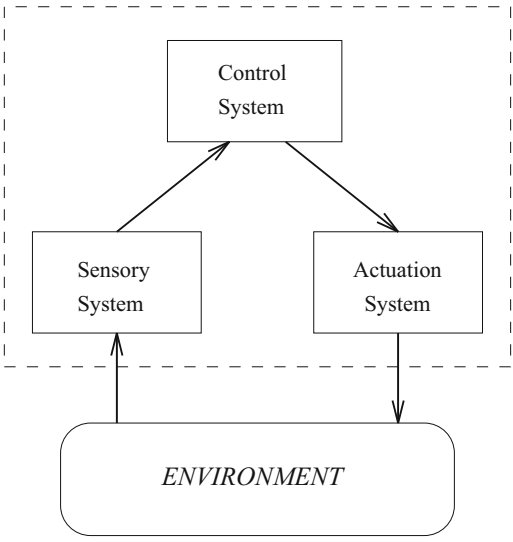


Fig. 1.2 Block diagram of a generic real-time control system

In fact, whereas the objective of fast computing is to minimize the average response time of a given set of tasks, the objective of real-time computing is to meet the individual timing requirement of each task [Sta88].

However short the average response time can be, without a scientific methodology, we will never be able to guarantee the individual timing requirements of each task in all possible circumstances. When several computational activities have different timing constraints, average performance has little significance for the correct behavior of the system. To better understand this issue, it is worth thinking about this little story:[2]

There was a man who drowned crossing a stream with an average depth of six inches.

Hence, rather than being fast, a real-time computing system should be predictable. And one safe way to achieve predictability is to investigate and employ new methodologies at every stage of the development of an application, from design to testing.

At the process level, the main difference between a real-time and a non-real-time task is that a real-time task is characterized by a *deadline*, which is the maximum time within which it must complete its execution.

In critical applications, a result produced after the deadline is not only late but wrong! Depending on the consequences that may occur because of a missed deadline, a real-time task can be distinguished in three categories:

- **Hard**: A real-time task is said to be *hard* if producing the results after its deadline may cause catastrophic consequences on the system under control.
- **Firm**: A real-time task is said to be *firm* if producing the results after its deadline is useless for the system, but does not cause any damage.
- **Soft**: A real-time task is said to be *soft* if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

A real-time operating system that is able to handle hard real-time tasks is called a *hard real-time system*. Typically, real-world applications include hard, firm, and soft activities; therefore, a hard real-time system should be designed to handle all such task categories using different strategies. In general, when an application consists of a hybrid task set, all hard tasks should be guaranteed offline, firm tasks should be guaranteed on line, aborting them if their deadline cannot be met, and soft tasks should be handled to minimize their average response time.

Examples of hard tasks can be found on safety-critical systems, and are typically related to sensing, actuation, and control activities, like:

- Sensory data acquisition;
- Data filtering and prediction;
- Detection of critical conditions;
- Data fusion and image processing;
- Actuator servoing;

[2] From John Stankovic's notes.

- Low-level control of critical system components; and
- Action planning for systems that tightly interact with the environment.

Examples of firm activities can be found in networked applications and multimedia systems, where skipping a packet or a video frame is less critical than processing it with a long delay. Thus, they include:

- Video playing;
- Audio/video encoding and decoding;
- On-line image processing;
- Sensory data transmission in distributed systems.

Soft tasks are typically related to system-user interactions. Thus, they include:

- The command interpreter of the user interface;
- Handling input data from the keyboard;
- Displaying messages on the screen;
- Representation of system state variables;
- Graphical activities; and
- Saving report data.

1.2.2 Limits of Current Real-Time Systems

Most of the real-time computing systems used to support control applications are based on kernels [AL86, Rea86, HHPD87, SBG86], which are modified versions of time-sharing operating systems. As a consequence, they have the same basic features found in time-sharing systems, which are not suited to support real-time activities. The main characteristics of such real-time systems include:

- **Multitasking**. A support for concurrent programming is provided through a set of system calls for process management (such as *create*, *activate*, *terminate*, *delay*, *suspend*, and *resume*). Many of these primitives do not take time into account and, even worse, introduce unbounded delays on tasks' execution time that may cause hard tasks to miss their deadlines in an unpredictable way.
- **Priority-based scheduling**. This scheduling mechanism is quite flexible, since it allows the implementation of several strategies for process management just by changing the rule for assigning priorities to tasks. Nevertheless, when application tasks have explicit time requirements, mapping timing constraints into a set of priorities may not be simple, especially in dynamic environments. The major problem comes from the fact that these kernels have a limited number of priority levels (typically 128 or 256), whereas task deadlines can vary in a much wider range. Moreover, in dynamic environments, the arrival of a new task may require the remapping of the entire set of priorities.
- **Ability to quickly respond to external interrupts**. This feature is usually obtained by setting interrupt priorities higher than process priorities and by reducing the portions of code executed with interrupts disabled. Note that,

although this approach increases the reactivity of the system to external events, it introduces unbounded delays on processes' execution. In fact, an application process will be always interrupted by a driver, even though it is more important than the device that is going to be served. Moreover, in the general case, the number of interrupts that a process can experience during its execution cannot be bounded in advance, since it depends on the particular environmental conditions.

- **Basic mechanisms for process communication and synchronization**. Binary semaphores are typically used to synchronize tasks and achieve mutual exclusion on shared resources. However, if no access protocols are used to enter critical sections, classical semaphores can cause a number of undesirable phenomena, such as priority inversion, chained blocking, and deadlock, which again introduce unbounded delays on real-time activities.
- **Small kernel and fast context switch**. This feature reduces system overhead, thus improving the average response time of the task set. However, a small average response time on the task set does not provide any guarantee on the individual deadlines of the tasks. On the other hand, a small kernel implies limited functionality, which affects the predictability of the system.
- **Support of a real-time clock as an internal time reference**. This is an essential feature for any real-time kernel that handles time-critical activities that interact with the environment. Nevertheless, in most commercial kernels, this is the only mechanism for time management. In many cases, there are no primitives for explicitly specifying timing constraints (such as deadlines) on tasks, and there is no mechanism for automatic activation of periodic tasks.

From the above features, it is easy to see that those types of real-time kernels are developed under the same basic assumptions made in time-sharing systems, where tasks are considered as unknown activities activated at random instants. Except for the priority, no other parameters are provided to the system. As a consequence, computation times, timing constraints, shared resources, or possible precedence relations among tasks are not considered in the scheduling algorithm, and hence no guarantee can be performed.

The only objectives that can be pursued with these systems is a quick reaction to external events and a "small" average response time for the other tasks. Although this may be acceptable for some soft applications, the lack of any form of guarantee precludes the use of these systems for those control applications that require stringent timing constraints that must be met to ensure safe behavior of the system.

1.2.3 Desirable Features of Real-Time Systems

Complex control applications that require hard timing constraints on tasks' execution need to be supported by highly predictable operating systems. Predictability can be achieved only by introducing radical changes in the basic design paradigms found in classical time-sharing systems.

For example, in any real-time control system, the code of each task is known a priori and hence can be analyzed to determine its characteristics in terms of computation time, resources, and precedence relations with other tasks. Therefore, there is no need to consider a task as an unknown processing entity; rather, its parameters can be used by the operating system to verify its schedulability within the specified timing requirements. Moreover, all hard tasks should be handled by the scheduler to meet their individual deadlines, not to reduce their average response time.

In addition, in any typical real-time application, the various control activities can be seen as members of a team acting together to accomplish one common goal, which can be the control of a nuclear power plant or an aircraft. This means that tasks are not all independent, and it is not strictly necessary to support independent address spaces.

In summary, there are some very important basic properties that real-time systems must have to support critical applications. They include:

- **Timeliness**. Results have to be correct not only in their value but also in the time domain. As a consequence, the operating system must provide specific kernel mechanisms for time management and for handling tasks with explicit timing constraints and different criticality.
- **Predictability**. To achieve a desired level of performance, the system must be analyzable to predict the consequences of any scheduling decision. In safety-critical applications, all timing requirements should be guaranteed offline, before putting system in operation. If some task cannot be guaranteed within its time constraints, the system must notify this fact in advance, so that alternative actions can be planned to handle the exception.
- **Efficiency**. Most of real-time systems are embedded into small devices with severe constraints in terms of space, weight, energy, memory, and computational power. In these systems, an efficient management of the available resources by the operating system is essential for achieving a desired performance.
- **Robustness**. Real-time systems must not collapse when they are subject to peak-load conditions, so they must be designed to manage all anticipated load scenarios. Overload management and adaptation behavior are essential features to handle systems with variable resource needs and high load variations.
- **Fault tolerance**. Single hardware and software failures should not cause the system to crash. Therefore, critical components of the real-time system have to be designed to be fault tolerant.
- **Maintainability**. The architecture of a real-time system should be designed according to a modular structure to ensure that possible system modifications are easy to perform.

1.3 Achieving Predictability

One of the most important properties that a hard real-time system should have is predictability [SR90]. That is, based on the kernel features and on the information associated with each task, the system should be able to predict the evolution of the tasks and guarantee in advance that all critical timing constraints will be met. The reliability of the guarantee, however, depends on a range of factors, which involve the architectural features of the hardware and the mechanisms and policies adopted in the kernel, up to the programming language used to implement the application.

The first component that affects the predictability of the scheduling is the processor itself. The internal characteristics of the processor, such as instruction prefetch, pipelining, cache memory, and direct memory access (DMA) mechanisms, are the first cause of nondeterminism. In fact, although these features improve the average performance of the processor, they introduce nondeterministic factors that prevent a precise analysis of the worst-case execution times. Other important components that influence the execution of the task set are the internal characteristics of the real-time kernel, such as the scheduling algorithm, the synchronization mechanism, the types of semaphores, the memory management policy, the communication semantics, and the interrupt handling mechanism.

In the rest of this chapter, the main sources of nondeterminism are considered in more detail, from the physical level up to the programming level.

1.3.1 DMA

Direct memory access (DMA) is a technique used by many peripheral devices to transfer data between the device and the main memory. The purpose of DMA is to relieve the central processing unit (CPU) of the task of controlling the input/output (I/O) transfer. Since both the CPU and the I/O device share the same bus, the CPU has to be blocked when the DMA device is performing a data transfer. Several different transfer methods exist.

One of the most common methods is called *cycle stealing*, according to which the DMA device steals a CPU memory cycle in order to execute a data transfer. During the DMA operation, the I/O transfer and the CPU program execution run in parallel. However, if the CPU and the DMA device require a memory cycle at the same time, the bus is assigned to the DMA device, and the CPU waits until the DMA cycle is completed. Using the cycle stealing method, there is no way of predicting how many times the CPU will have to wait for DMA during the execution of a task; hence, the response time of a task cannot be precisely determined.

A possible solution to this problem is to adopt a different technique, which requires the DMA device to use the memory *time-slice method* [SR88]. According to this method, each memory cycle is split into two adjacent time slots: one reserved for the CPU and the other for the DMA device. This solution is more expensive than

cycle stealing but more predictable. In fact, since the CPU and DMA device do not conflict, the response time of the tasks does not increase due to DMA operations and hence can be predicted with higher accuracy.

1.3.2 Cache

The cache is a fast memory that is inserted as a buffer between the CPU and the random access memory (RAM) to speed up processes' execution. It is physically located after the memory management unit (MMU) and is not visible at the software programming level. Once the physical address of a memory location is determined, the hardware checks whether the requested information is stored in the cache: if it is, data are read from the cache; otherwise, the information is taken from the RAM, and the content of the accessed location is copied in the cache along with a set of adjacent locations. In this way, if the next memory access is done to one of these locations, the requested data can be read from the cache, without having to access the memory.

This buffering technique is motivated by the fact that statistically the most frequent accesses to the main memory are limited to a small address space, a phenomenon called *program locality*. For example, it has been observed that with a 1-Mb memory and an 8-Kbyte cache, the data requested from a program are found in the cache 80% of the time (*hit ratio*).

The need for having a fast cache appeared when memory was much slower. Today, however, since memory has an access time almost comparable to that of the cache, the main motivation for having a cache is not only to speed up process execution but also to reduce conflicts with other devices. In any case, the cache is considered as a processor attribute that speeds up the activities of a computer.

In real-time systems, the cache introduces some degree of nondeterminism. In fact, although statistically the requested data are found in the cache 80% of the time, it is also true that in the other 20% of the cases, the performance degrades. This happens because, when data is not found in the cache (cache fault or miss), the access time to memory is longer, due to the additional data transfer from RAM to cache. Furthermore, when performing write operations in memory, the use of the cache is even more expensive in terms of access time because any modification made on the cache must be copied to the memory in order to maintain data consistency. Statistical observations show that 90% of the memory accesses are for read operations, whereas only 10% are for writes. Statistical observations, however, can provide only an estimation of the average behavior of an application, but cannot be used for deriving worst-case bounds.

In preemptive systems, the cache behavior is also affected by the number of preemptions. In fact, preemption destroys program locality and heavily increases the number of cache misses due to the lines evicted by the preempting task. Moreover, the cache-related preemption delay (CRPD) depends on the specific point at which preemption takes place; therefore, it is very difficult to precisely

estimate [AG08, GA07]. Bui et al. [BCSM08] showed that on a PowerPC MPC7410 with 2-MByte two-way associative L2 cache, the WCET increment due to cache interference can be as large as 33% of the WCET measured in non-preemptive mode.

1.3.3 Interrupts

Interrupts generated by I/O peripheral devices represent a big problem for the predictability of a real-time system because, if not properly handled, they can introduce unbounded delays during process execution. In almost any operating system, the arrival of an interrupt signal causes the execution of a service routine (*driver*), dedicated to the management of its associated device. The advantage of this method is to encapsulate all hardware details of the device inside the driver, which acts as a server for the application tasks. For example, in order to get data from an I/O device, each task must enable the hardware to generate interrupts, wait for the interrupt, and read the data from a memory buffer shared with the driver, according to the following protocol:

<enable device interrupts>
<wait for interrupt>
<get the result>

In many operating systems, interrupts are served using a fixed priority scheme, according to which each driver is scheduled based on a static priority, higher than process priorities. This assignment rule is motivated by the fact that interrupt handling routines usually deal with I/O devices that have real-time constraints, whereas most application programs do not. In the context of real-time systems, however, this assumption is certainly not valid because a control process could be more urgent than an interrupt handling routine. Since, in general, it is very difficult to bound a priori the number of interrupts that a task may experience, the delay introduced by the interrupt mechanism on tasks' execution becomes unpredictable.

In order to reduce the interference of the drivers on the application tasks and still perform I/O operations with the external world, the peripheral devices must be handled in a different way. In the following, three possible techniques are illustrated.

1.3.3.1 Approach A

The most radical solution to eliminate interrupt interference is to disable all external interrupts, except the one from the timer (necessary for basic system operations). In this case, all peripheral devices must be handled by the application tasks, which have direct access to the registers of the interfacing boards. Since no interrupt is generated, data transfer takes place through polling.

The direct access to I/O devices allows great programming flexibility and eliminates the delays caused by the drivers' execution. As a result, the time needed for transferring data can be precisely evaluated and charged to the task that performs the operation. Another advantage of this approach is that the kernel does not need to be modified as the I/O devices are replaced or added.

The main disadvantage of this solution is a low processor efficiency on I/O operations, due to the busy wait of the tasks while accessing the device registers. Another minor problem is that the application tasks must have the knowledge of all low-level details of the devices that they want to handle. However, this can be easily solved by encapsulating all device-dependent routines in a set of library functions that can be called by the application tasks. This approach is adopted in RK, a research hard real-time kernel designed to support multisensory robotics applications [LKP88].

1.3.3.2 Approach B

As in the previous approach, all interrupts from external devices are disabled, except the one from the timer. Unlike the previous solution, however, the devices are not directly handled by the application tasks but are managed in turn by dedicated kernel routines, periodically activated by the timer.

This approach eliminates the unbounded delays due to the execution of interrupt drivers and confines all I/O operations to one or more periodic kernel tasks, whose computational load can be computed once and for all and taken into account through a specific utilization factor. In some real-time systems, I/O devices are subdivided into two classes based on their speed: slow devices are multiplexed and served by a single cyclical I/O process running at a low rate, whereas fast devices are served by dedicated periodic system tasks, running at higher frequencies. The advantage of this approach with respect to the previous one is that all hardware details of the peripheral devices can be encapsulated into kernel procedures and do not need to be known to the application tasks.

Because the interrupts are disabled, the major problem of this approach is due to the busy wait of the kernel I/O handling routines, which makes the system less efficient during the I/O operations. With respect to the previous approach, this case is characterized by a higher system overhead, due to the communication required among the application tasks and the I/O kernel routines for exchanging I/O data. Finally, since the device handling routines are part of the kernel, it has to be modified when some device is replaced or added. This type of solution is adopted in the MARS system [DRSK89, KDK+89].

1.3.3.3 Approach C

A third approach that can be adopted in real-time systems to deal with the I/O devices is to leave all external interrupts enabled while reducing the drivers to

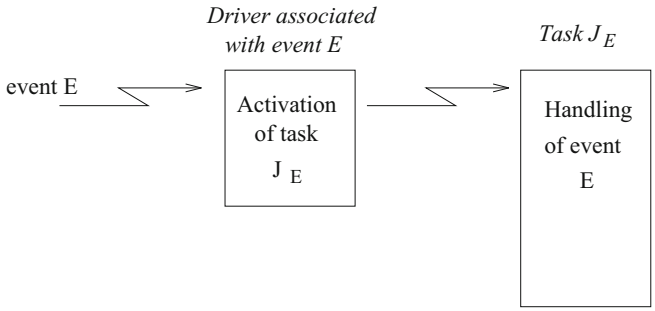


Fig. 1.3 Activation of a device-handling task

the least possible size. According to this method, the only purpose of each driver is to activate a proper task that will take care of the device management. Once activated, the device manager task executes under the direct control of the operating system, and it is guaranteed and scheduled just like any other application task. In this way, the priority that can be assigned to the device handling task is completely independent from other priorities and can be set according to the application requirements. Thus, a control task can have a higher priority than a device- handling task.

The idea behind this approach is schematically illustrated in Fig. 1.3. The occurrence of event E generates an interrupt, which causes the execution of a driver associated with that interrupt. Unlike the traditional approach, this driver does not handle the device directly but only activates a dedicated task, J_E, which will be the actual device manager.

The major advantage of this approach with respect to the previous ones is to eliminate the busy wait during I/O operations. Moreover, compared to the traditional technique, the unbounded delays introduced by the drivers during tasks' execution are also drastically reduced (although not completely removed), so the task execution times become more predictable. As a matter of fact, a little unbounded overhead due to the execution of the small drivers still remains in the system, and it should be taken into account in the guarantee mechanism. However, it can be neglected in most practical cases. This type of solution is adopted in the ARTS system [TK88, TM89], in HARTIK [BDN93, But93], and in SPRING [SR91].

1.3.4 System Calls

System predictability also depends on how the kernel primitives are implemented. In order to precisely evaluate the worst-case execution time of each task, all kernel calls should be characterized by a bounded execution time, used by the guarantee

mechanism while performing the schedulability analysis of the application. In addition, in order to simplify this analysis, it would be desirable that each kernel primitive be preemptable. In fact, any nonpreemptable section could possibly delay the activation or the execution of critical activities, causing a timing fault to hard deadlines.

1.3.5 Semaphores

The typical semaphore mechanism used in traditional operating systems is not suited for implementing real-time applications because it is subject to the priority inversion phenomenon, which occurs when a high-priority task is blocked by a low-priority task for an unbounded interval of time. Priority inversion must absolutely be avoided in real-time systems, since it introduces nondeterministic delays on the execution of critical tasks.

For the mutual exclusion problem, priority inversion can be avoided by adopting particular protocols that must be used every time a task wants to enter a critical section. For instance, efficient solutions are provided by *Basic Priority Inheritance* [SRL90], *Priority Ceiling* [SRL90], and *Stack Resource Policy* [Bak91]. These protocols will be described and analyzed in Chap. 7. The basic idea behind these protocols is to modify the priority of the tasks based on the current resource usage and control the resource assignment through a test executed at the entrance of each critical section. The aim of the test is to bound the maximum blocking time of the tasks that share critical sections.

The implementation of such protocols may require a substantial modification of the kernel, which concerns not only the *wait* and *signal* calls but also some data structures and mechanisms for task management.

1.3.6 Memory Management

Similarly to other kernel mechanisms, memory management techniques must not introduce nondeterministic delays during the execution of real-time activities. For example, demand paging schemes are not suitable to real-time applications subject to rigid time constraints because of the large and unpredictable delays caused by page faults and page replacements. Typical solutions adopted in most real-time systems adhere to a memory segmentation rule with a fixed memory management scheme. Static partitioning is particularly efficient when application programs require similar amounts of memory.

In general, static allocation schemes for resources and memory management increase the predictability of the system but reduce its flexibility in dynamic environments. Therefore, depending on the particular application requirements, the

system designer has to make the most suitable choices for balancing predictability against flexibility.

1.3.7 Programming Language

Besides the hardware characteristics of the physical machine and the internal mechanisms implemented in the kernel, there are other factors that can determine the predictability of a real-time system. One of these factors is certainly the programming language used to develop the application. As the complexity of real-time systems increases, high demand will be placed on the programming abstractions provided by languages.

Unfortunately, current programming languages are not expressive enough to prescribe certain timing behavior and hence are not suited for realizing predictable real-time applications. For example, the Ada language (demanded by the Department of Defense of the United States for implementing embedded real-time concurrent applications) does not allow the definition of explicit time constraints on tasks' execution. The *delay* statement puts only a lower bound on the time the task is suspended, and there is no language support to guarantee that a task cannot be delayed longer than a desired upper bound. The existence of nondeterministic constructs, such as the *select* statement, prevents the performing of a reliable worst-case analysis of the concurrent activities. Moreover, the lack of protocols for accessing shared resources allows a high-priority task to wait for a low-priority task for an unbounded duration. As a consequence, if a real-time application is implemented using the Ada language, the resulting timing behavior of the system is likely to be unpredictable.

Recently, new high-level languages have been proposed to support the development of hard real-time applications. For example, *Real-Time Euclid* [KS86] is a programming language specifically designed to address reliability and guaranteed schedulability issues in real-time systems. To achieve this goal, Real-Time Euclid forces the programmer to specify time bounds and timeout exceptions in all loops, waits, and device accessing statements. Moreover, it imposes several programming restrictions, such as the ones listed below:

- **Absence of dynamic data structures**. Third-generation languages normally permit the use of dynamic arrays, pointers, and arbitrarily long strings. In real-time languages, however, these features must be eliminated because they would prevent a correct evaluation of the time required to allocate and deallocate dynamic structures.
- **Absence of recursion**. If recursive calls were permitted, the schedulability analyzer could not determine the execution time of subprograms involving recursion or how much storage will be required during execution.

- **Time-bounded loops**. In order to estimate the duration of the cycles at compile time, Real-Time Euclid forces the programmer to specify for each loop construct the maximum number of iterations.

Real-Time Euclid also allows the classification of processes as periodic or aperiodic and provides statements for specifying task timing constraints, such as activation time and period, as well as system timing parameters, such as the time resolution.

Another high-level language for programming hard real-time applications is *Real-Time Concurrent C* [GR91]. It extends Concurrent C by providing facilities to specify periodicity and deadline constraints, to seek guarantees that timing constraints will be met, and to perform alternative actions when either the timing constraints cannot be met or guarantees are not available. With respect to Real-Time Euclid, which has been designed to support static real-time systems, where guarantees are made at compile time, Real-Time Concurrent C is oriented to dynamic systems, where tasks can be activated at runtime. Another important feature of Real-Time Concurrent C is that it permits the association of a deadline with any statement, using the following construct:

within deadline *(d) statement-1*
[**else** *statement-2*]

If the execution of *statement-1* starts at time t and is not completed at time $(t + d)$, then its execution is terminated and *statement-2*, if specified, is executed.

Clearly, any real-time construct introduced in a language must be supported by the operating system through dedicated kernel services, which must be designed to be efficient and analyzable. Among all kernel mechanisms that influence predictability, the scheduling algorithm is certainly the most important factor, since it is responsible for satisfying timing and resource contention requirements.

In the rest of this book, several scheduling algorithms are illustrated and analyzed under different constraints and assumptions. Each algorithm is characterized in terms of performance and complexity to assist a designer in the development of reliable real-time applications.

Exercises

1.1 Explain the difference between fast computing and real-time computing.
1.2 What are the main limitations of the current real-time kernels for the development of critical control applications?
1.3 Discuss the features that a real-time system should have for exhibiting a predictable timing behavior.
1.4 Describe the approaches that can be used in a real-time system to handle peripheral I/O devices in a predictable fashion.
1.5 Which programming restrictions should be used in a programming language to permit the analysis of real-time applications? Suggest some extensions that could be included in a language for real-time systems.

Chapter 2
Basic Concepts

2.1 Introduction

Over the last few years, several algorithms and methodologies have been proposed in the literature to improve the predictability of real-time systems. In order to present these results, we need to define some basic concepts that will be used throughout the book. We begin with the most important software entity treated by any operating system, the *process*. A process is a computation that is executed by the CPU in a sequential fashion. In this text, the terms *process* and *task* are used as synonyms. However, it is worth saying that some authors prefer to distinguish them and define a task as a sequential execution of code that does not suspend itself during execution, whereas a process is a more complex computational activity, which can be composed by many tasks.

When a single processor has to execute a set of concurrent tasks—that is, tasks that can overlap in time—the CPU has to be assigned to the various tasks according to a predefined criterion, called a *scheduling policy*. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm* . The specific operation of allocating the CPU to a task selected by the scheduling algorithm is referred to as *dispatching*.

Thus, a task that could potentially execute on the CPU can be either in execution if it has been selected by the scheduling algorithm or waiting for the CPU if another task is executing. A task that can potentially execute on the processor, independently on its actual availability, is called an *active* task. A task waiting for the processor is called a *ready* task, whereas the task in execution is called a *running* task. All ready tasks waiting for the processor are kept in a queue, called *ready queue*. Operating systems that handle different types of tasks may have more than one ready queue.

In many operating systems that allow dynamic task activation, the running task can be interrupted at any point, so that a more important task that arrives in the system can immediately gain the processor and does not need to wait in the ready queue. In this case, the running task is interrupted and inserted in the ready queue,

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_2

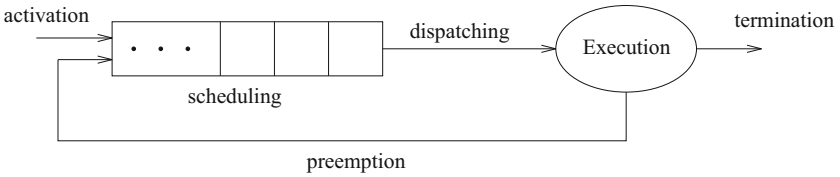


Fig. 2.1 Queue of ready tasks waiting for execution

while the CPU is assigned to the most important ready task that just arrived. The operation of suspending the running task and inserting it into the ready queue is called *preemption*. Figure 2.1 schematically illustrates the concepts presented above. In dynamic real-time systems, preemption is important for three reasons [SZ92]:

- Tasks performing exception handling may need to preempt existing tasks so that responses to exceptions may be issued in a timely fashion.
- When application tasks have different levels of criticality, expressing task importance, preemption permits executing the most critical activities as soon as they arrive.
- More efficient schedules can be produced to improve system responsiveness.

On the other hand, preemption destroys program locality and introduces a runtime overhead that inflates the execution time of tasks. As a consequence, limiting preemptions in real-time schedules can have beneficial effects in terms of schedulability. This issue will be investigated in Chap. 8.

Given a set of tasks, $J = \{J_1, \ldots, J_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion. More formally, a schedule can be defined as a function $\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}$ such that $\forall t \in \mathbf{R}^+$, $\exists t_1, t_2$ such that $t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)\ \sigma(t) = \sigma(t')$. In other words, $\sigma(t)$ is an integer step function and $\sigma(t) = k$, with $k > 0$, means that task J_k is executing at time t, while $\sigma(t) = 0$ means that the CPU is idle. Figure 2.2 shows an example of schedule obtained by executing three tasks: J_1, J_2, and J_3.

- At times t_1, t_2, t_3, and t_4, the processor performs a *context switch*.
- Each interval $[t_i, t_{i+1})$ in which $\sigma(t)$ is constant is called *time slice*. Interval $[x, y)$ identifies all values of t such that $x \leq t < y$.
- A *preemptive* schedule is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy. In preemptive schedules, tasks may be executed in disjointed interval of times.
- A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints.
- A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule.

An example of preemptive schedule is shown in Fig. 2.3.

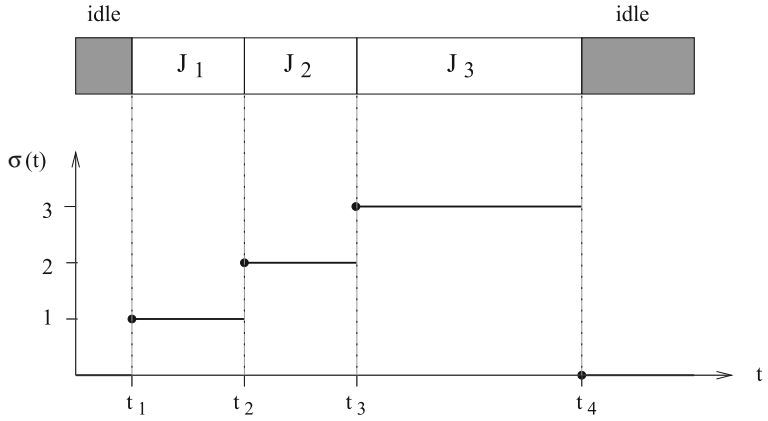


Fig. 2.2 Schedule obtained by executing three tasks J_1, J_2, and J_3

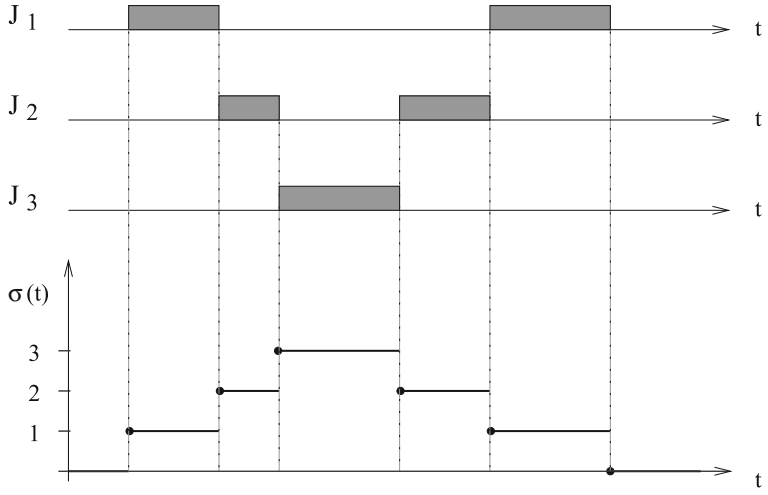


Fig. 2.3 Example of a preemptive schedule

2.2 Types of Task Constraints

Typical constraints that can be specified on real-time tasks are of three classes: timing constraints, precedence relations, and mutual exclusion constraints on shared resources.

2.2.1 Timing Constraints

Real-time systems are characterized by computational activities with stringent timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a task is the *deadline*, which represents the time before which a process should complete its execution without causing any damage to the system. If a deadline is specified with respect to the task arrival time, it is called a *relative deadline*, whereas if it is specified with respect to time zero, it is called an *absolute deadline*. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in three categories:

- **Hard**: A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the system under control.
- **Firm**: A real-time task is said to be *firm* if missing its deadline does not cause any damage to the system, but the output has no value.
- **Soft**: A real-time task is said to be *soft* if missing its deadline has still some utility for the system, although causing a performance degradation.

In general, a real-time task J_i can be characterized by the following parameters:

- **Arrival time $\mathbf{a_i}$**: is the time at which a task becomes ready for execution; it is also referred to as *request time* or *release time* and indicated by r_i;
- **Computation time $\mathbf{C_i}$**: is the time necessary to the processor for executing the task without interruption;
- **Absolute Deadline $\mathbf{d_i}$**: is the time before which a task should be completed to avoid damage to the system;
- **Relative Deadline $\mathbf{D_i}$**: is the difference between the absolute deadline and the request time: $D_i = d_i - r_i$;
- **Start time $\mathbf{s_i}$**: is the time at which a task starts its execution;
- **Finishing time $\mathbf{f_i}$**: is the time at which a task finishes its execution;
- **Response time $\mathbf{R_i}$**: is the difference between the finishing time and the request time: $R_i = f_i - r_i$;
- **Criticality**: is a parameter related to the consequences of missing the deadline (typically, it can be hard, firm, or soft);
- **Value $\mathbf{v_i}$**: represents the relative importance of the task with respect to the other tasks in the system;
- **Lateness $\mathbf{L_i}$**: $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative;
- **Tardiness** or *Exceeding time* $\mathbf{E_i}$: $E_i = max(0, L_i)$ is the time a task stays active after its deadline;
- **Laxity** or *Slack time* $\mathbf{X_i}$: $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Some of the parameters defined above are illustrated in Fig. 2.4.

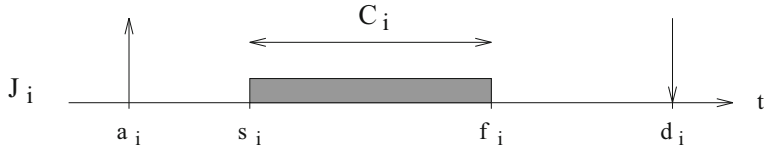


Fig. 2.4 Typical parameters of a real-time task

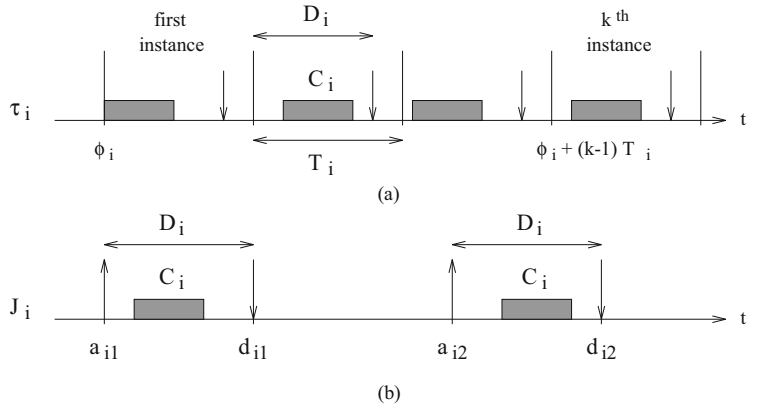


Fig. 2.5 Sequence of instances for a periodic and an aperiodic task

Another timing characteristic that can be specified on a real-time task concerns the regularity of its activation. In particular, tasks can be defined as *periodic* or *aperiodic*. Periodic tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, which are regularly activated at a constant rate. For the sake of clarity, from now on, a periodic task will be denoted by τ_i whereas an aperiodic task by J_i. The generic k^{th} job of a periodic task τ_i will be denoted by $\tau_{i,k}$.

The activation time of the first periodic instance ($\tau_{i,1}$) is called *phase*. If ϕ_i is the phase of task τ_i, the activation time of the k^{th} instance is given by $\phi_i + (k-1)T_i$, where T_i is the activation *period* of the task. In many practical cases, a periodic process can be completely characterized by its computation time C_i, its period T_i, and its relative deadline D_i.

Aperiodic tasks also consist of an infinite sequence of identical jobs (or instances); however, their activations are not regularly interleaved. An aperiodic task where consecutive jobs are separated by a minimum interarrival time is called a *sporadic task*. Figure 2.5 shows an example of task instances for a periodic and an aperiodic task.

2.2.2 Precedence Constraints

In certain applications, computational activities cannot be executed in arbitrary order but have to respect some precedence relations defined at the design stage. Such precedence relations are usually described through a directed acyclic graph G, where tasks are represented by nodes and precedence relations by arrows. A precedence graph G induces a partial order on the task set:

- The notation $J_a \prec J_b$ specifies that task J_a is a *predecessor* of task J_b, meaning that G contains a directed path from node J_a to node J_b.
- The notation $J_a \rightarrow J_b$ specifies that task J_a is an *immediate predecessor* of J_b, meaning that G contains an arc directed from node J_a to node J_b.

Figure 2.6 illustrates a directed acyclic graph that describes the precedence constraints among five tasks. From the graph structure, we observe that task J_1 is the only one that can start executing since it does not have predecessors. Tasks with no predecessors are called *beginning tasks*. As J_1 is completed, either J_2 or J_3 can start. Task J_4 can start only when J_2 is completed, whereas J_5 must wait the completion of J_2 and J_3. Tasks with no successors, as J_4 and J_5, are called *ending tasks*.

In order to understand how precedence graphs can be derived from tasks' relations, let us consider the application illustrated in Fig. 2.7. Here, a number of objects moving on a conveyor belt must be recognized and classified using a stereo vision system, consisting of two cameras mounted in a suitable location. Suppose that the recognition process is carried out by integrating the two-dimensional features of the top view of the objects with the height information extracted by the pixel disparity on the two images. As a consequence, the computational activities of the application can be organized by defining the following tasks:

- Two tasks (one for each camera) dedicated to image acquisition, whose objective is to transfer the image from the camera to the processor memory (they are identified by *acq1* and *acq2*);

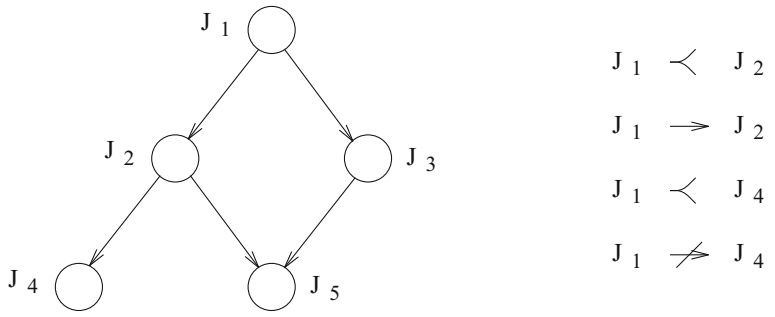


Fig. 2.6 Precedence relations among five tasks

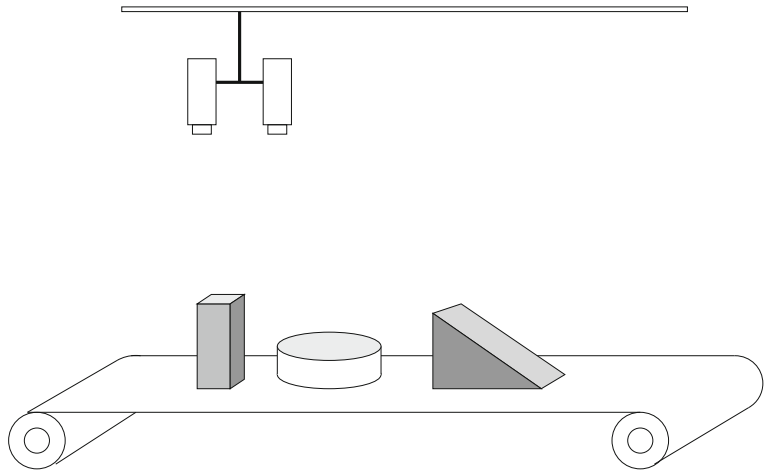


Fig. 2.7 Industrial application which requires a visual recognition of objects on a conveyor belt

- Two tasks (one for each camera) dedicated to low-level image processing (typical operations performed at this level include digital filtering for noise reduction and edge detection; we identify these tasks as *edge1* and *edge2*);
- A task for extracting two-dimensional features from the object contours (it is referred to as *shape*);
- A task for computing the pixel disparities from the two images (it is referred to as *disp*);
- A task for determining the object height from the results achieved by the *disp* task (it is referred to as *H*);
- A task performing the final recognition (this task integrates the geometrical features of the object contour with the height information and tries to match these data with those stored in the data base; it is referred to as *rec*).

From the logic relations existing among the computations, it is easy to see that tasks *acq1* and *acq2* can be executed in parallel before any other activity. Tasks *edge1* and *edge2* can also be executed in parallel, but each task cannot start before the associated acquisition task completes. Task *shape* is based on the object contour extracted by the low-level image processing; therefore, it must wait the termination of both *edge1* and *edge2*. The same is true for task *disp*, which however can be executed in parallel with task *shape*. Then, task *H* can only start as *disp* completes and, finally, task *rec* must wait the completion of *H* and *shape*. The resulting precedence graph is shown in Fig. 2.8.

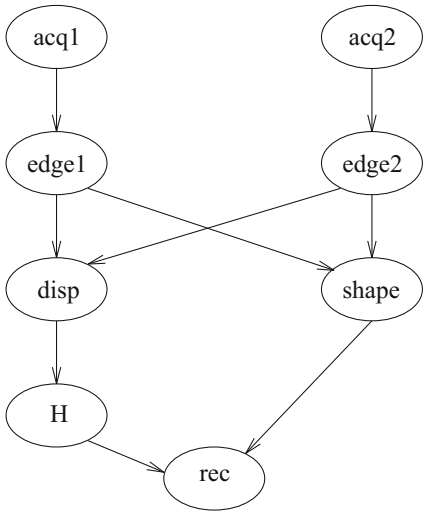


Fig. 2.8 Precedence graph associated with the robotic application

2.2.3 Resource Constraints

From a process point of view, a *resource* is any software structure that can be used by the process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*.

To maintain data consistency, many shared resources do not allow simultaneous accesses by competing tasks, but require their mutual exclusion. This means that a task cannot access a resource R if another task is inside R manipulating its data structures. In this case, R is called a *mutually exclusive resource*. A piece of code executed under mutual exclusion constraints is called a *critical section*.

To better understand why mutual exclusion is important for guaranteeing data consistency, consider the application illustrated in Fig. 2.9, where two tasks cooperate to track a moving object: task τ_W gets the object coordinates from a sensor and writes them into a shared buffer R, containing two variables (x, y); task τ_D reads the variables from the buffer and plots a point on the screen to display the object trajectory.

If the access to the buffer is not mutually exclusive, task τ_W (having lower priority than τ_D) may be preempted while updating the variables, so leaving the buffer in an inconsistent state. The situation is illustrated in Fig. 2.10, where, at time t, the (x, y) variables have values (1,2). If τ_W is preempted after updating x and before updating y, τ_D will display the object in (4,2), which is neither the old nor the new position.

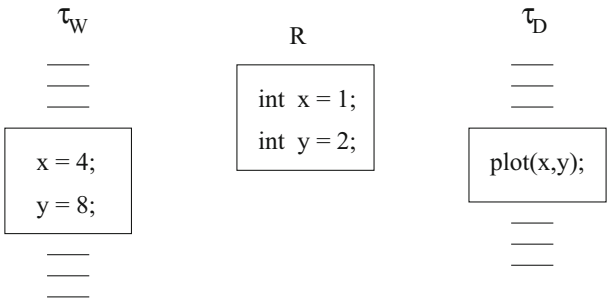


Fig. 2.9 Two tasks sharing a buffer with two variables

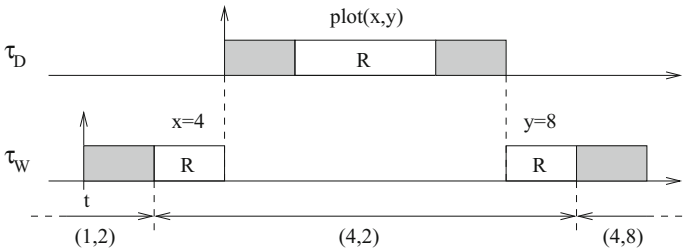
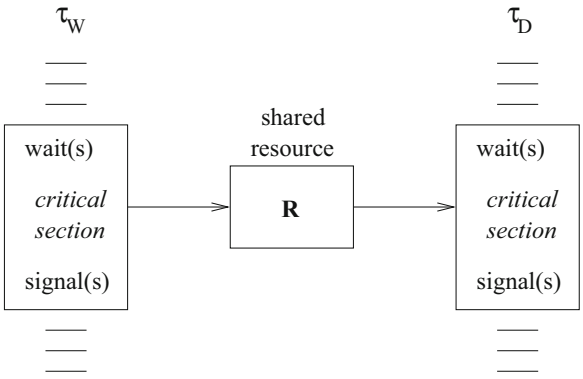


Fig. 2.10 Example of schedule creating data inconsistency

Fig. 2.11 Structure of two tasks that share a mutually exclusive resource protected by a semaphore



To ensure a correct access to exclusive resources, operating systems provide a synchronization mechanism (e.g., semaphores) that can be used to create critical sections of code. In the following, when we say that two or more tasks have resource constraints, we mean that they share resources that are accessed through a synchronization mechanism.

To avoid the problem illustrated in Fig. 2.10, both tasks have to encapsulate the instructions that manipulate the shared variables into a critical section. If a binary semaphore *s* is used for this purpose, then each critical section must begin with a *wait(s)* primitive and must end with a *signal(s)* primitive, as shown in Fig. 2.11.

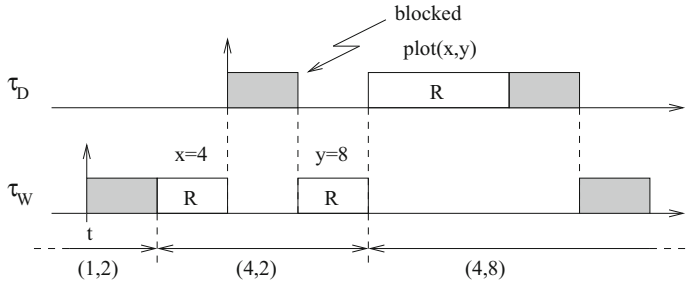


Fig. 2.12 Example of schedule when the resource is protected by a semaphore

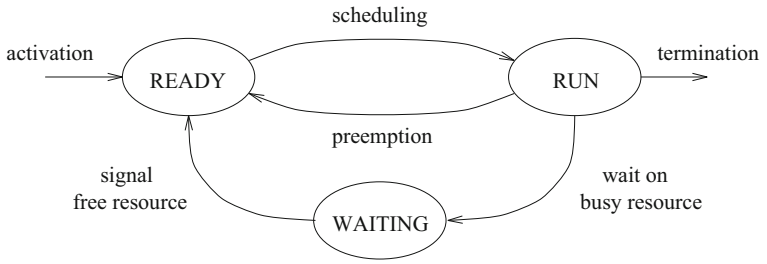


Fig. 2.13 Waiting state caused by resource constraints

If the resource is free, the *wait(s)* primitive executed by τ_W notifies that a task is using the resource, which becomes locked until the task executes the *signal(s)*. Hence, if τ_D preempts τ_W inside the critical section, it is blocked as soon as it executes *wait(s)*, and the processor is given back to τ_W. When τ_W exits its critical section by executing *signal(s)*, then τ_D is resumed, and the processor is given to the ready task with the highest priority. The resulting schedule is depicted in Fig. 2.12.

A task waiting for an exclusive resource is said to be *blocked* on that resource. All tasks blocked on the same resource are kept in a queue associated with the semaphore protecting the resource. When a running task executes a *wait* primitive on a locked semaphore, it enters a *waiting* state, until another task executes a *signal* primitive that unlocks the semaphore. Notice that, when a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Fig. 2.13.

2.3 Definition of Scheduling Problems

In general, to define a scheduling problem, we need to specify three sets: a set of n tasks $J = \{J_1, J_2, \ldots, J_n\}$, a set of m processors $P = \{P_1, P_2, \ldots, P_m\}$, and a set of s types of resources $R = \{R_1, R_2, \ldots, R_s\}$. Moreover, precedence

relations among tasks can be specified through a directed acyclic graph, and timing constraints can be associated with each task. In this context, scheduling means to assign processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints [B+93]. This problem, in its general form, has been shown to be NP-complete [GJ79] and hence computationally intractable.

Indeed, the complexity of scheduling algorithms is of high relevance in dynamic real-time systems, where scheduling decisions must be taken on-line during task execution. A *polynomial algorithm* is one whose time complexity grows as a polynomial function p of the input length n of an instance. The complexity of such algorithms is denoted by $O(p(n))$. Each algorithm whose complexity function cannot be bounded in that way is called an *exponential time algorithm*. In particular, **NP** is the class of all decision problems that can be solved in polynomial time by a *non*deterministic Turing machine.

A problem Q is said to be NP-*complete* if $Q \in \mathbf{NP}$ and, for every $Q' \in \mathbf{NP}$, Q' is polynomially transformable to Q [GJ79]. A decision problem Q is said to be NP-*hard* if all problems in **NP** are polynomially transformable to Q, but we cannot show that $Q \in \mathbf{NP}$.

Let us consider two algorithms with complexity functions n and 5^n, respectively, and let us assume that an elementary step for these algorithms lasts 1 μs. If the input length of the instance is $n = 30$, then it is easy to calculate that the polynomial algorithm can solve the problem in 30 μs, whereas the other needs about $3 \cdot 10^5$ centuries. This example illustrates that the difference between polynomial and exponential time algorithms is large, and, hence, it may have a strong influence on the performance of dynamic real-time systems. As a consequence, one of the research objectives on real-time scheduling is to restrict our attention to simpler, but still practical, problems that can be solved in polynomial time complexity.

In order to reduce the complexity of constructing a feasible schedule, one may simplify the computer architecture (e.g., by restricting to the case of uniprocessor systems), or one may adopt a preemptive model, use fixed priorities, remove precedence and/or resource constraints, assume simultaneous task activation, homogeneous task sets (solely periodic or solely aperiodic activities), and so on. The assumptions made on the system or on the tasks are typically used to classify the various scheduling algorithms proposed in the literature.

2.3.1 Classification of Scheduling Algorithms

Among the great variety of algorithms proposed for scheduling real-time tasks, we can identify the following main classes:

- **Preemptive vs. Non-preemptive**:
  - In preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.

  - In non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as the task terminates its execution.

- **Static vs. Dynamic**:
  - Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
  - Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.

- **Offline vs. On-Line**:
  - We say that a scheduling algorithm is used offline if it is executed on the entire task set before actual task activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
  - We say that a scheduling algorithm is used on-line if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.

- **Optimal vs. Heuristic**:
  - An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then an algorithm is said to be optimal if it is able to find a feasible schedule, if there exists one.
  - An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A heuristic algorithm tends toward the optimal schedule, but does not guarantee to find it.

Moreover, an algorithm is said to be *clairvoyant* if it knows the future, that is, if it knows in advance the arrival times of all the tasks. Although such an algorithm does not exist in reality, it can be used for comparing the performance of real algorithms against the best possible one.

2.3.1.1 Guarantee-Based algorithms

In hard real-time applications that require highly predictable behavior, the feasibility of the schedule should be guaranteed in advance, that is, before task execution. In this way, if a critical task cannot be scheduled within its deadline, the system is still in time to execute an alternative action, attempting to avoid catastrophic consequences. In order to check the feasibility of the schedule before tasks' execution, the system has to plan its actions by looking ahead in the future and by assuming a worst-case scenario.

In static real-time systems, where the task set is fixed and known a priori, all task activations can be pre-calculated offline, and the entire schedule can be stored in a table that contains all guaranteed tasks arranged in the proper order. Then, at

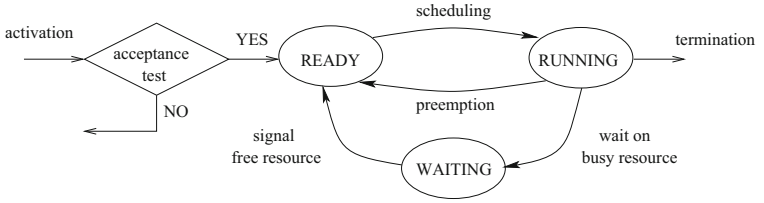


Fig. 2.14 Scheme of the guarantee mechanism used in dynamic real-time systems

runtime, a dispatcher simply removes the next task from the table and puts it in the running state. The main advantage of the static approach is that the runtime overhead does not depend on the complexity of the scheduling algorithm. This allows very sophisticated algorithms to be used to solve complex problems or find optimal scheduling sequences. On the other hand, however, the resulting system is quite inflexible to environmental changes; thus, predictability strongly relies on the observance of the hypotheses made on the environment.

In dynamic real-time systems (typically consisting of firm tasks), tasks can be created at runtime; hence, the guarantee must be done *on-line* every time a new task is created. A scheme of the guarantee mechanism typically adopted in dynamic real-time systems is illustrated in Fig. 2.14.

If J is the current task set that has been previously guaranteed, a newly arrived task J_{new} is accepted into the system if and only if the task set $J' = J \cup \{J_{new}\}$ is found schedulable. If J' is not schedulable, then task J_{new} is rejected to preserve the feasibility of the current task set.

It is worth noting that, since the guarantee mechanism is based on worst-case assumptions, a task could unnecessarily be rejected. This means that the guarantee of firm tasks is achieved at the cost of a lower efficiency. On the other hand, the benefit of having a guarantee mechanism is that potential overload situations can be detected in advance to avoid negative effects on the system. One of the most dangerous phenomena caused by a transient overload is called *domino effect*. It refers to the situation in which the arrival of a new task causes *all* previously guaranteed tasks to miss their deadlines. Let us consider, for example, the situation depicted in Fig. 2.15, where tasks are scheduled based on their absolute deadlines.

At time t_0, if task J_{new} were accepted, all other tasks (previously schedulable) would miss their deadlines. In planned-based algorithms, this situation is detected at time t_0, when the guarantee is performed and causes task J_{new} to be rejected.

In summary, the guarantee test ensures that, once a task is accepted, it will complete within its deadline and, moreover, its execution will not jeopardize the feasibility of the tasks that have been previously guaranteed.

2.3.1.2 Best-Effort Algorithms

In certain real-time applications, computational activities have soft timing constraints that should be met whenever possible to satisfy system requirements. In

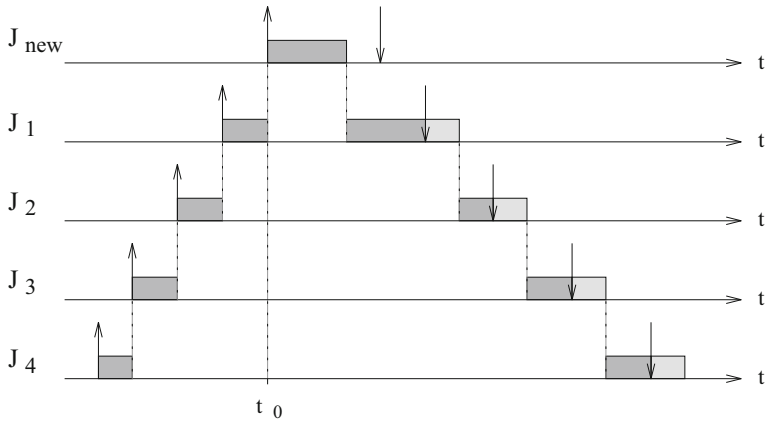


Fig. 2.15 Example of domino effect

these systems, missing soft deadlines does not cause catastrophic consequences, but only a performance degradation.

For example, in typical multimedia applications, the objective of the computing system is to handle different types of information (such as text, graphics, images, and sound) in order to achieve a certain quality of service for the users. In this case, the timing constraints associated with the computational activities depend on the quality of service requested by the users; hence, missing a deadline may only affect the performance of the system.

To efficiently support soft real-time applications that do not have hard timing requirements, a *best-effort* approach may be adopted for scheduling.

A best-effort scheduling algorithm tries to "do its best" to meet deadlines, but there is no guarantee of finding a feasible schedule. In a best-effort approach, tasks may be enqueued according to policies that take time constraints into account; however, since feasibility is not checked, a task may be aborted during its execution. On the other hand, best-effort algorithms perform much better than guarantee-based schemes in the average case. In fact, whereas the pessimistic assumptions made in the guarantee mechanism may unnecessarily cause task rejections, in best-effort algorithms a task is aborted only under real overload conditions.

2.3.2 Metrics for Performance Evaluation

The performance of scheduling algorithms is typically evaluated through a cost function defined over the task set. For example, classical scheduling algorithms try to minimize the average response time, the total completion time, the weighted sum of completion times, or the maximum lateness. When deadlines are considered, they are usually added as constraints, imposing that all tasks must meet their deadlines.

Table 2.1 Example of cost functions

Average response time:

$$\overline{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

Total completion time:

$$t_c = \max_i(f_i) - \min_i(a_i)$$

Weighted sum of completion times:

$$t_w = \sum_{i=1}^{n} w_i f_i$$

Maximum lateness:

$$L_{max} = \max_i(f_i - d_i)$$

Maximum number of late tasks:

$$N_{late} = \sum_{i=1}^{n} miss(f_i)$$

where

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

If some deadlines cannot be met with an algorithm A, the schedule is said to be infeasible by A. Table 2.1 shows some common cost functions used for evaluating the performance of a scheduling algorithm.

The metrics adopted in the scheduling algorithm has strong implications on the performance of the real-time system [SSDNB95], and it must be carefully chosen according to the specific application to be developed. For example, the average response time is generally not of interest for real-time applications because there

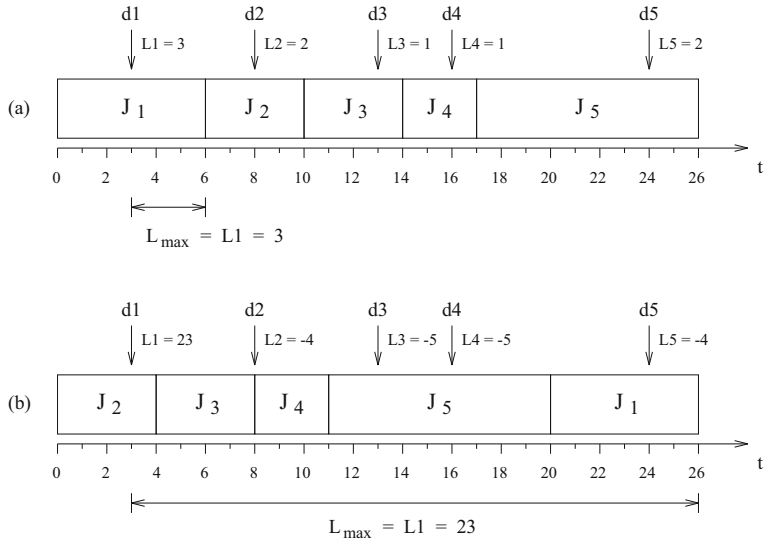


Fig. 2.16 The schedule in a minimizes the maximum lateness, but all tasks miss their deadline. The schedule in b has a greater maximum lateness, but four tasks out of five complete before their deadline

is not direct assessment of individual timing properties such as periods or deadlines. The same is true for minimizing the total completion time. The weighted sum of completion times is relevant when tasks have different importance values that they impart to the system on completion. Minimizing the maximum lateness can be useful at design time when resources can be added until the maximum lateness achieved on the task set is less than or equal to zero. In that case, no task misses its deadline. In general, however, minimizing the maximum lateness does not minimize the number of tasks that miss their deadlines and does not necessarily prevent one or more tasks from missing their deadline.

Let us consider, for example, the case depicted in Fig. 2.16. The schedule shown in Fig. 2.16a minimizes the maximum lateness, but all tasks miss their deadline. On the other hand, the schedule shown in Fig. 2.16b has a greater maximum lateness, but four tasks out of five complete before their deadline.

When tasks have soft deadlines and the application concern is to meet as many deadlines as possible (without a priori guarantee), then the scheduling algorithm should use a cost function that minimizes the number of late tasks.

In other applications, the benefit of executing a task may depend not only on the task importance but also on the time at which it is completed. This can be described by means of specific *utility functions*, which describe the value associated with the task as a function of its completion time.

Figure 2.17 illustrates some typical utility functions that can be defined on the application tasks. For instance, non-real-time tasks (a) do not have deadlines; thus, the value achieved by the system is proportional to the task importance and does not

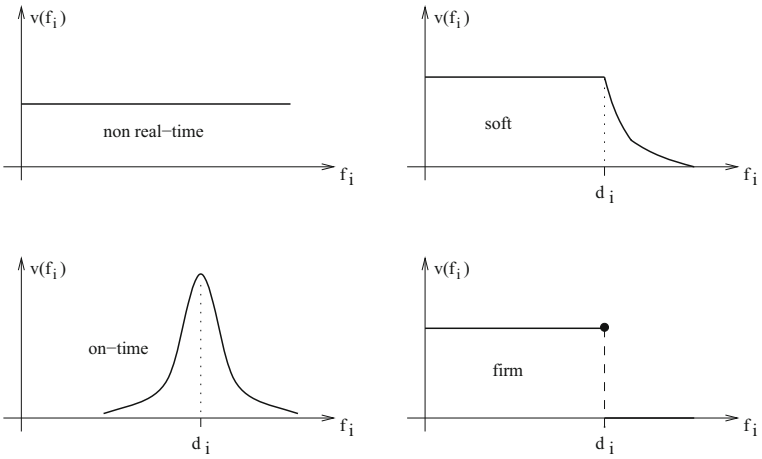


Fig. 2.17 Example of cost functions for different types of tasks

depend on the completion time. Soft tasks (b) have noncritical deadlines; therefore, the value gained by the system is constant if the task finishes before its deadline but decreases with the exceeding time. In some cases (c), it is required to execute a task *on time*, that is, not too early and not too late with respect to a given deadline. Hence, the value achieved by the system is high if the task is completed around the deadline, but it rapidly decreases with the absolute value of the lateness. Such types of constraints are typical when playing notes, since human ear is quite sensitive to time jitter.

In other cases (d), executing a task after its deadline does not cause catastrophic consequences, but there is no benefit for the system; thus, the utility function is zero after the deadline.

When utility functions are defined on the tasks, the performance of a scheduling algorithm can be measured by the *cumulative value*, given by the sum of the utility functions computed at each completion time:

$$Cumulative\_value = \sum_{i=1}^{n} v(f_i).$$

This type of metrics is very useful for evaluating the performance of a system during overload conditions, and it is considered in more detail in Chap. 9.

2.4 Scheduling Anomalies

In this section we describe some singular examples that clearly illustrate that real-time computing is not equivalent to fast computing, since, for example, an

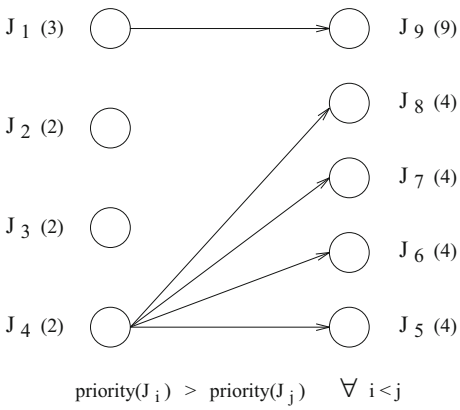


Fig. 2.18 Precedence graph of the task set J; numbers in parentheses indicate computation times

increase of computational power in the supporting hardware does not always cause an improvement of performance. These particular situations, called Richard's anomalies, have been described by Graham in 1976 and refer to task sets with precedence relations executed in a multiprocessor environment.

Designers should be aware of such insidious anomalies, in order to take the proper countermeasures to avoid them. The most important anomalies are expressed by the following theorem [Gra76, SSDNB95].

Theorem 2.1 (Graham) *If a task set is optimally scheduled on a multiprocessor with some priority assignment, a fixed number of processors, fixed execution times, and precedence constraints, then increasing the number of processors, reducing execution times, or weakening the precedence constraints can increase the schedule length.*

This result implies that if tasks have deadlines, then adding resources (e.g., an extra processor) or relaxing constraints (less precedence among tasks or fewer execution times requirements) can make things worse. A few examples can best illustrate why this theorem is true.

Let us consider a task set consisting of nine tasks $J = \{J_1, J_2, \ldots, J_9\}$, sorted by decreasing priorities, so that J_i priority is greater than J_j priority if and only if $i < j$. Moreover, tasks are subject to precedence constraints that are described through the graph shown in Fig. 2.18. Computation times are indicated in parentheses.

If this task set is executed on a parallel machine with three processors, where the highest-priority task is assigned to the first available processor, the resulting schedule σ^* is illustrated in Fig. 2.19, where the global completion time is $t_c = 12$ units of time.

Now we will show that adding an extra processor, reducing tasks' execution times, or weakening precedence constraints will increase the global completion time of the task set.

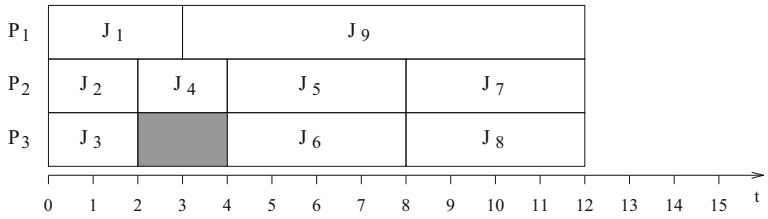


Fig. 2.19 Optimal schedule of task set J on a three-processor machine

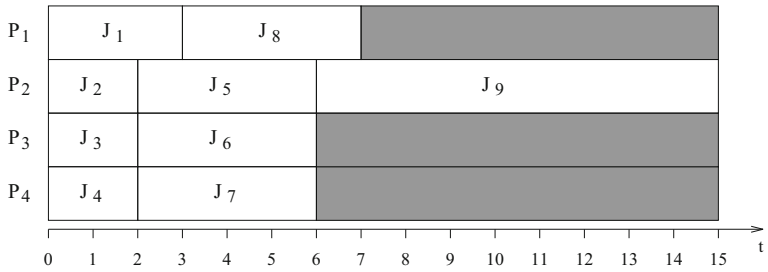


Fig. 2.20 Schedule of task set J on a four-processor machine

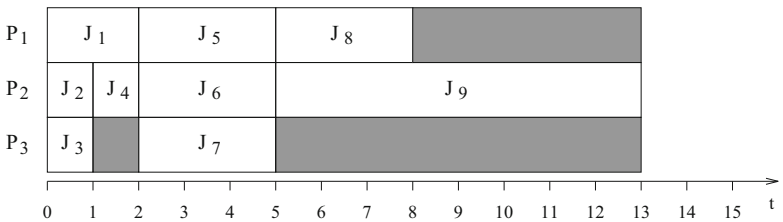


Fig. 2.21 Schedule of task set J on three processors, with computation times reduced by one unit of time

Number of Processors Increased

If we execute the task set J on a more powerful machine consisting of four processors, we obtain the schedule illustrated in Fig. 2.20, which is characterized by a global completion time of $t_c = 15$ units of time.

Computation Times Reduced

One could think that the global completion time of the task set J could be improved by reducing tasks' computation times of each task. However, we can surprisingly see that if we reduce the computation time of each task by one unit of time, the schedule length will increase with respect to the optimal schedule σ^*, and the global completion time will be $t_c = 13$, as shown in Fig. 2.21.

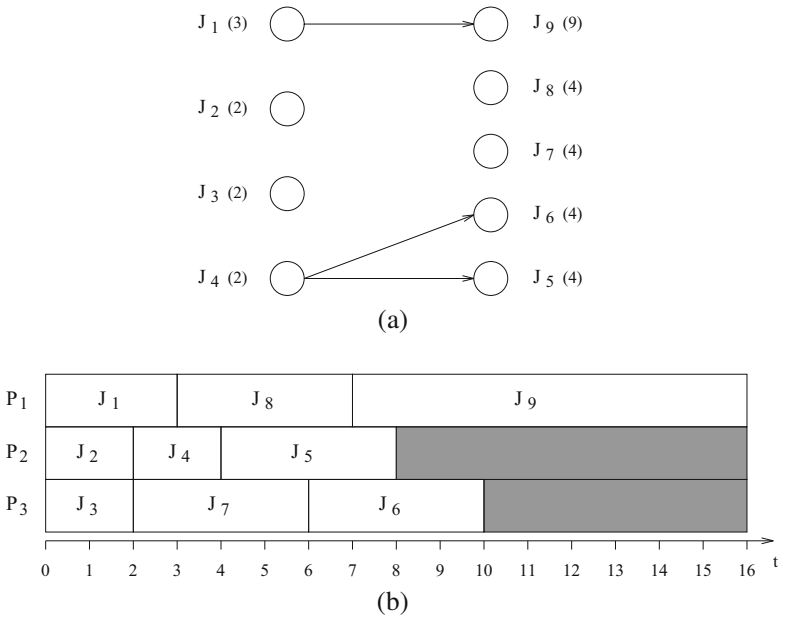


Fig. 2.22 (**a**) Precedence graph of task set J obtained by removing the constraints on tasks J_7 and J_8. (**b**) Schedule of task set J on three processors, with precedence constraints weakened

Precedence Constraints Weakened

Scheduling anomalies can also arise if we remove precedence constraints from the directed acyclic graph depicted in Fig. 2.18. For instance, if we remove the precedence relations between task J_4 and tasks J_7 and J_8 (see Fig. 2.22a), we obtain the schedule shown in Fig. 2.22b, which is characterized by a global completion time of $t_c = 16$ units of time.

Anomalies Under Resource Constraints

The following example shows that, in the presence of shared resources, the schedule length of a task set can increase when reducing tasks' computation times. Consider the case illustrated in Fig. 2.23, where five tasks are statically allocated on two processors: tasks J_1 and J_2 on processor P1 and tasks J_3, J_4, and J_5 on processor P2 (tasks are indexed by decreasing priority). Moreover, tasks J_2 and J_4 share the same resource in exclusive mode; hence, their execution cannot overlap in time. A schedule of this task set is shown in Fig. 2.23a, where the total completion time is $t_c = 17$.

If we now reduce the computation time of task J_1 on the first processor, then J_2 can begin earlier and take the resource before task J_4. As a consequence, task J_4 must now block over the shared resource and possibly miss its deadline. This situation is illustrated in Fig. 2.23b. As we can see, the blocking time experienced by J_4 causes a delay in the execution of J_5 (which may also miss its deadline), increasing the total completion time of the task set from 17 to 22.

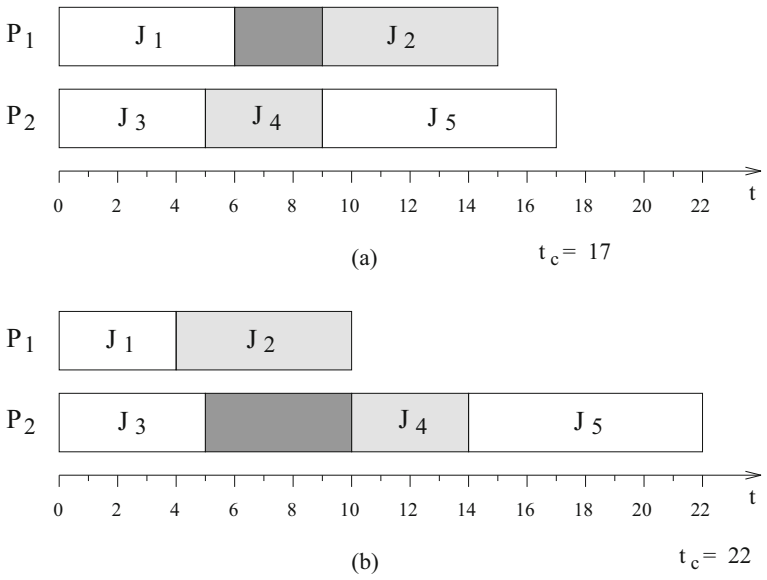


Fig. 2.23 Example of anomaly under resource constraints. If J_2 and J_4 share the same resource in exclusive mode, the optimal schedule length (**a**) increases if the computation time of task J_1 is reduced (**b**). Task are statically allocated on the processors

Notice that the scheduling anomaly illustrated by the previous example is particularly insidious for hard real-time systems, because tasks are guaranteed based on their worst-case behavior, but they may complete before their worst-case computation time. A simple solution that avoids the anomaly is to keep the processor idle if tasks complete earlier, but this can be very inefficient. There are algorithms, such as the one proposed by Shen [SRS93], which tries to reclaim such an idle time while addressing the anomalies so that they will not occur.

If tasks share mutually exclusive resources, scheduling anomalies can also occur in a uniprocessor system when changing the processor speed [But06]. In particular, the anomaly can be expressed as follows:

> A real-time application that is feasible on a given processor, can become infeasible when running on a faster processor.

Figure 2.24 illustrates a simple example where two tasks, τ_1 and τ_2, share a common resource (critical sections are represented by light gray areas). Task τ_1 has a higher priority, arrives at time $t = 2$, and has a relative deadline $D_1 = 7$. Task τ_2, having lower priority, arrives at time $t = 0$ and has a relative deadline $D_2 = 23$. Suppose that, when the tasks are executed at a certain speed S_1, τ_1 has a computation time $C_1 = 6$, (where 2 units of time are spent in the critical section), whereas τ_2 has a computation time $C_2 = 18$ (where 12 units of time are spent in the critical section). As shown in Fig. 2.24a, if τ_1 arrives just before τ_2 enters its critical section, it is able to complete before its deadline, without experiencing any blocking. However, if the

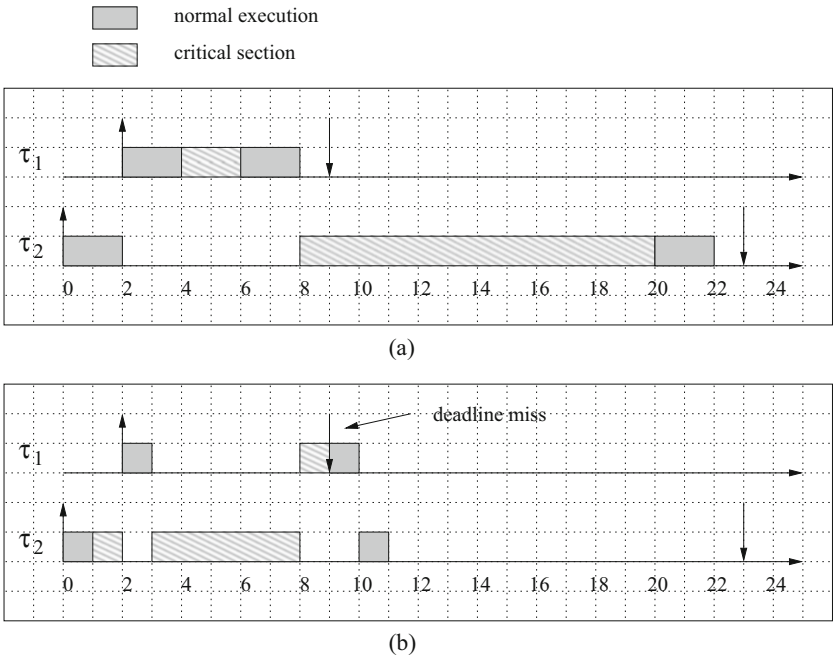


Fig. 2.24 Scheduling anomaly in the presence of resource constraints: task τ_1 meets its deadline when the processor is executing at a certain speed S_1 (**a**), but misses its deadline when the speed is doubled (**b**)

same task set is executed at a double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Fig. 2.24b. This happens because, when τ_1 arrives, τ_2 already granted its resource, causing an extra blocking in the execution of τ_1, due to mutual exclusion.

Note that, although the average response time of the task set is reduced on the faster processor (from 14 to 9.5 units of time), the response time of task τ_1 increases when doubling the speed, because of the extra blocking on the shared resource.

Anomalies Under Non-preemptive Scheduling

Similar situations can occur in non-preemptive scheduling. Figure 2.25 illustrates an anomalous behavior occurring in a set of three real-time tasks, τ_1, τ_2, and τ_3, running in a non-preemptive fashion. Tasks are assigned a fixed priority proportional to their relative deadline; thus, τ_1 is the task with the highest priority and τ_3 is the task with the lowest priority. As shown in Fig. 2.25a, when tasks are executed at speed S_1, τ_1 has a computation time $C_1 = 2$ and completes at time $t = 6$. However, if the same task set is executed with double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in Fig. 2.25b. This happens because, when τ_1 arrives, τ_3 already started its execution and cannot be preempted (due to the non-preemptive mode).

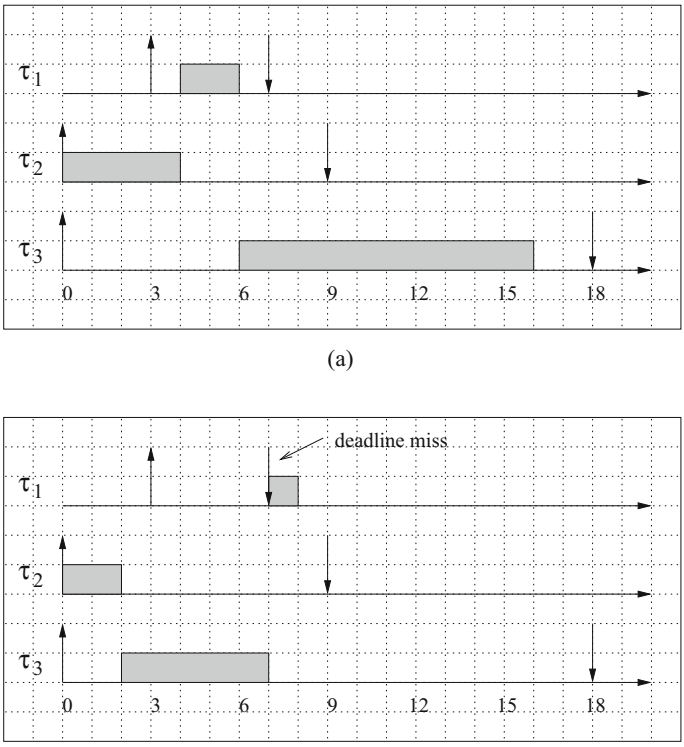


Fig. 2.25 Scheduling anomaly in the presence of non-preemptive tasks: task τ_1 meets its deadline when the processor is executing at speed S_1 (**a**), but misses its deadline when the speed is doubled (**b**)

It is worth observing that a set of non-preemptive tasks can be considered as a special case of a set of tasks sharing a single resource (the processor) for their entire execution. According to this view, each task executes as it were inside a big critical section with a length equal to the task computation time. Once a task starts executing, it behaves as it were locking a common semaphore, thus preventing all the other tasks from taking the processor.

Anomalies Using a Delay Primitive

Another timing anomaly can occur when tasks using shared resources explicitly suspend themselves through a $delay(T)$ system call, which suspend the execution of the calling task for T units of time. Figure 2.26a shows a case in which τ_1 is feasible and has a slack time of six units when running at the highest priority, suggesting that it could easily tolerate a delay of two units. However, if τ_1 executes a $delay(2)$ at time $t = 2$, it gives the opportunity to τ_2 to lock the shared resource. Hence, when

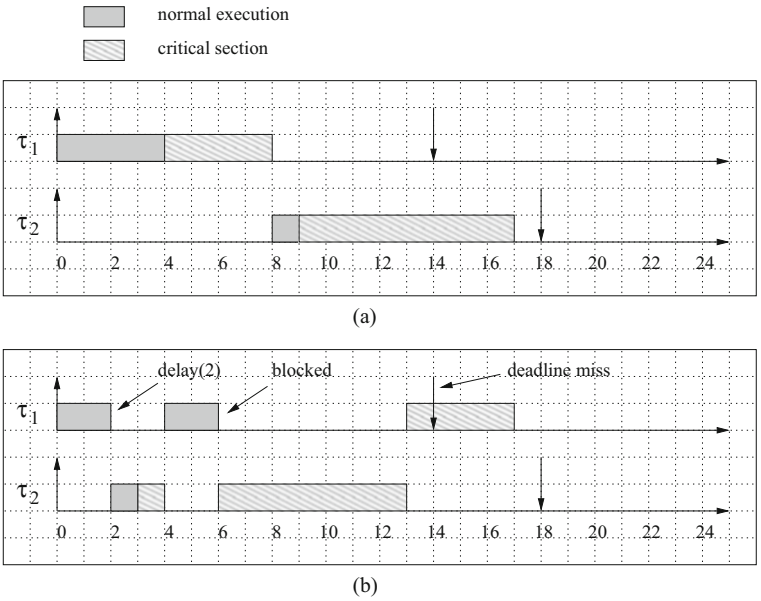


Fig. 2.26 Scheduling anomaly in the presence of a delay: τ_1 has a slack of six units of time when running at the highest priority (**a**), but cannot tolerate a self-suspension of two units (**b**)

τ_1 resumes, it has to block on the semaphore for 7 units, thus missing its deadline, as shown in Fig. 2.26b.

The example shown in Fig. 2.26 illustrates an anomaly in which a task with a large slack cannot tolerate a self-suspension of a much smaller value. Figure 2.27 shows another case in which the suspension of a task can also cause a longer delay in a different task, even without sharing any resource. When τ_1 is assigned a higher priority than τ_2, the resulting schedule shown in Fig. 2.27a is feasible, with a slack for τ_1 of three units of time. However, if the third instance of τ_1 executes a $delay(1)$ after one unit of execution, τ_2 will miss its deadline.

Exercises

2.1 Give the formal definition of a schedule, explaining the difference between preemptive and non-preemptive scheduling.
2.2 Explain the difference between periodic and aperiodic tasks, and describe the main timing parameters that can be defined for a real-time activity.

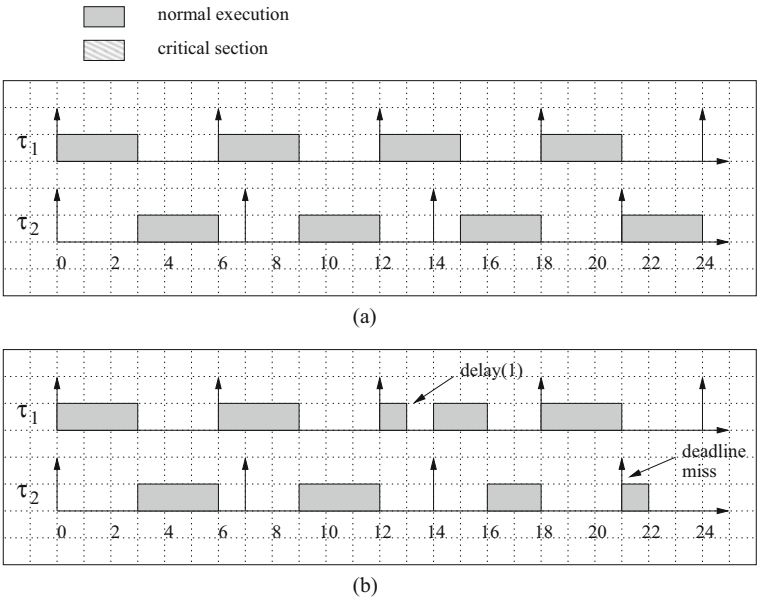


Fig. 2.27 Scheduling anomaly in the presence of a delay: two tasks are feasible without delays (**a**), but a delay in τ_1 causes a deadline miss in τ_2 (**b**)

2.3 Describe a real-time application as a number of tasks with precedence relations, and draw the corresponding precedence graph.
2.4 Discuss the difference between static and dynamic, on-line and offline, optimal, and heuristic scheduling algorithms.
2.5 Provide an example of domino effect, caused by the arrival of a task J^*, in a feasible set of three tasks.

Chapter 3
Aperiodic Task Scheduling

3.1 Introduction

In this chapter we present a variety of algorithms for scheduling real-time aperiodic tasks on a single machine environment. Each algorithm represents a solution for a particular scheduling problem, which is expressed through a set of assumptions on the task set and by an optimality criterion to be used on the schedule. The restrictions made on the task set are aimed at simplifying the algorithm in terms of time complexity. When no restrictions are applied on the application tasks, the complexity can be reduced by employing heuristic approaches, which do not guarantee to find the optimal solution to a problem but can still guarantee a feasible schedule in a wide range of situations.

Although the algorithms described in this chapter are presented for scheduling aperiodic tasks on uniprocessor systems, many of them can be extended to work on multiprocessor or distributed architectures and deal with more complex task models.

To facilitate the description of the scheduling problems presented in this chapter, we introduce a systematic notation that could serve as a basis for a classification scheme. Such a notation, proposed by Graham et al. [GLLK79], classifies all algorithms using three fields $\alpha \mid \beta \mid \gamma$, having the following meaning:

- The first field α describes the machine environment on which the task set has to be scheduled (uniprocessor, multiprocessor, distributed architecture, and so on).
- The second field β describes task and resource characteristics (preemptive, independent versus precedence constrained, synchronous activations, and so on).
- The third field γ indicates the optimality criterion (performance measure) to be followed in the schedule.

For example:

- $1 \mid prec \mid L_{max}$ denotes the problem of scheduling a set of tasks with precedence constraints on a uniprocessor machine in order to minimize the maximum

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_3

lateness. If no additional constraints are indicated in the second field, preemption is allowed at any time, and tasks can have arbitrary arrivals.

- $3 \mid no\_preem \mid \sum f_i$ denotes the problem of scheduling a set of tasks on a three-processor machine. Preemption is not allowed and the objective is to minimize the sum of the finishing times. Since no other constraints are indicated in the second field, tasks do not have precedence nor resource constraints but have arbitrary arrival times.
- $2 \mid sync \mid \sum Late_i$ denotes the problem of scheduling a set of tasks on a two-processor machine. Tasks have synchronous arrival times and do not have other constraints. The objective is to minimize the number of late tasks.

3.2 Jackson's Algorithm

The problem considered by this algorithm is $1 \mid sync \mid L_{max}$. That is, a set $\mathcal{J}$ of n aperiodic tasks has to be scheduled on a single processor, minimizing the maximum lateness. All tasks consist of a single job, have synchronous arrival times, but can have different computation times and deadlines. No other constraints are considered; hence, tasks must be independent, that is, cannot have precedence relations and cannot share resources in exclusive mode.

Notice that, since all tasks arrive at the same time, preemption is not an issue in this problem. In fact, preemption is effective only when tasks may arrive dynamically and newly arriving tasks have higher priority than currently executing tasks.

Without loss of generality, we assume that all tasks are activated at time $t = 0$, so that each job J_i can be completely characterized by two parameters: a computation time C_i and a relative deadline D_i (which, in this case, is also equal to the absolute deadline). Thus, the task set $\mathcal{J}$ can be denoted as

$$\mathcal{J} = \{J_i(C_i, D_i),\ i = 1, \ldots, n\}.$$

A simple algorithm that solves this problem was found by Jackson in 1955. It is called *Earliest Due Date* (EDD) and can be expressed by the following rule [Jac55].

Theorem 3.1 (Jackson's Rule) *Given a set of n independent tasks, any algorithm that executes the tasks in order of nondecreasing deadlines is optimal with respect to minimizing the maximum lateness.*

Proof Jackson's theorem can be proved by a simple interchange argument. Let σ be a schedule produced by any algorithm A. If A is different than EDD, then there exist two tasks J_a and J_b, with $d_a \leq d_b$, such that J_b immediately precedes J_a in σ. Now, let σ' be a schedule obtained from σ by exchanging J_a with J_b, so that J_a immediately precedes J_b in σ'.

As illustrated in Fig. 3.1, interchanging the position of J_a and J_b in σ cannot increase the maximum lateness. In fact, the maximum lateness between J_a and J_b

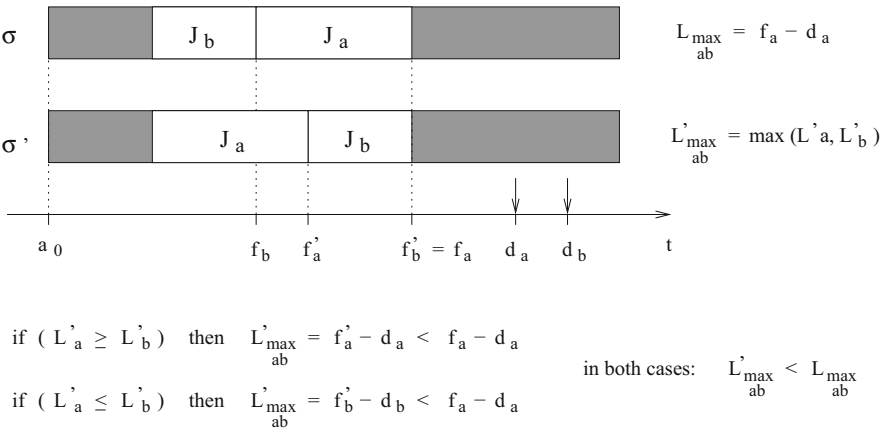


Fig. 3.1 Optimality of Jackson's algorithm

in σ is $L_{max}(a, b) = f_a - d_a$, whereas the maximum lateness between J_a and J_b in σ' can be written as $L'_{max}(a, b) = max(L'_a, L'_b)$. Two cases must be considered:

1. If $L'_a \geq L'_b$, then $L'_{max}(a, b) = f'_a - d_a$, and, since $f'_a < f_a$, we have $L'_{max}(a, b) < L_{max}(a, b)$.
2. If $L'_a \leq L'_b$, then $L'_{max}(a, b) = f'_b - d_b = f_a - d_b$, and, since $d_a < d_b$, we have $L'_{max}(a, b) < L_{max}(a, b)$.

Since, in both cases, $L'_{max}(a, b) < L_{max}(a, b)$, we can conclude that interchanging J_a and J_b in σ cannot increase the maximum lateness of the task set. By a finite number of such transpositions, σ can be transformed in σ_{EDD}, and, since in each transposition the maximum lateness cannot increase, σ_{EDD} is optimal. □

The complexity required by Jackson's algorithm to build the optimal schedule is due to the procedure that sorts the tasks by increasing deadlines. Hence, if the task set consists of n tasks, the complexity of the EDD algorithm is $O(n \log n)$.

3.2.1 Examples

3.2.1.1 Example 1

Consider a set of five tasks, simultaneously activated at time $t = 0$, whose parameters (worst-case computation times and deadlines) are indicated in the table shown in Fig. 3.2. The schedule of the tasks produced by the EDD algorithm is also depicted in Fig. 3.2. The maximum lateness is equal to -1, and it is due to task J_4, which completes a unit of time before its deadline. Since the maximum lateness is negative, we can conclude that all tasks have been executed within their deadlines.

	J_1	J_2	J_3	J_4	J_5
C_i	1	1	1	3	2
d_i	3	10	7	8	5

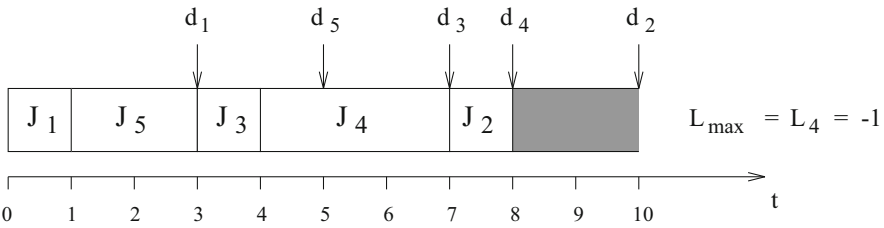


Fig. 3.2 A feasible schedule produced by Jackson's algorithm

Notice that the optimality of the EDD algorithm cannot guarantee the feasibility of the schedule for any task set. It only guarantees that, if there exists a feasible schedule for a task set, then EDD will find it.

3.2.1.2 Example 2

Figure 3.3 illustrates an example in which the task set cannot be feasibly scheduled. Still, however, EDD produces the optimal schedule that minimizes the maximum lateness. Notice that, since J_4 misses its deadline, the maximum lateness is greater than zero ($L_{max} = L_4 = 2$).

3.2.2 Guarantee

To guarantee that a set of tasks can be feasibly scheduled by the EDD algorithm, we need to show that, in the worst case, all tasks can complete before their deadlines. This means that we have to show that for each task, the worst-case finishing time f_i is less than or equal to its absolute deadline d_i:

$$\forall i = 1, \ldots, n \quad f_i \leq d_i.$$

If tasks have hard timing requirements, such a schedulability analysis must be done before actual tasks' execution. Without loss of generality, we can assume that tasks $J_1, J_2, \ldots, J_n$ are activated at time $t = 0$ and are listed by increasing deadlines, so

	J_1	J_2	J_3	J_4	J_5
C_i	1	2	1	4	2
d_i	2	5	4	8	6

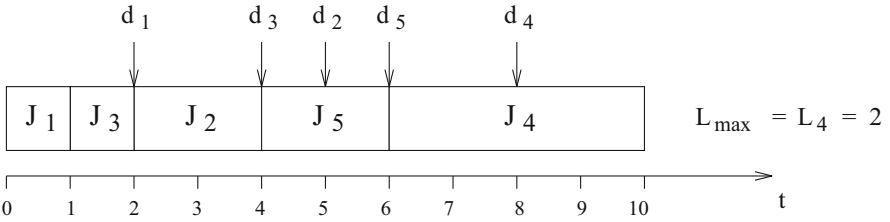


Fig. 3.3 An infeasible schedule produced by Jackson's algorithm

that J_1 is the task with the earliest deadline. In this case, we have that, for each task J_i, $d_i = D_i$, hence the worst-case finishing time of task J_i can easily be computed as

$$f_i = \sum_{k=1}^{i} C_k.$$

Therefore, if the task set consists of n tasks, the guarantee test can be performed by verifying the following n conditions:

$$\forall i = 1, \ldots, n \quad \sum_{k=1}^{i} C_k \leq D_i. \tag{3.1}$$

3.3 Horn's Algorithm

If tasks are not synchronous but can have arbitrary arrival times (i.e., tasks can be activated dynamically during execution), then preemption becomes an important factor. In general, a scheduling problem in which preemption is allowed is always easier than its non-preemptive counterpart. In a non-preemptive scheduling algorithm, the scheduler must ensure that a newly arriving task will never need to interrupt a currently executing task in order to meet its own deadline. This guarantee requires a considerable amount of searching. If preemption is allowed, however, this

searching is unnecessary, since a task can be interrupted if a more important task arrives [WR91].

In 1974, Horn found an elegant solution to the problem of scheduling a set of n-independent tasks on a uniprocessor system, when tasks may have dynamic arrivals and preemption is allowed ($1 \mid preem \mid L_{max}$).

The algorithm, called *Earliest Deadline First* (EDF), can be expressed by the following theorem [Hor74].

Theorem 3.2 (Horn) *Given a set of n-independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.*

This result can be proved by an interchange argument similar to the one used by Jackson. The formal proof of the EDF optimality has been given by Dertouzos in 1974 [Der74], and it is illustrated below. The complexity of the algorithm is $O(n)$ per task, if the ready queue is implemented as a list, or $O(n \log n)$ per task, if the ready queue is implemented as a heap.

3.3.1 EDF Optimality

The original proof provided by Dertouzos [Der74] shows that EDF is optimal in the sense of feasibility. This means that if there exists a feasible schedule for a task set $\mathcal{J}$, then EDF is able to find it. The proof can easily be extended to show that EDF also minimizes the maximum lateness. This is more general because an algorithm that minimizes the maximum lateness is also optimal in the sense of feasibility. The contrary is not true.

Using the same approach proposed by Dertouzos, let σ be the schedule produced by a generic algorithm A and let σ_{EDF} be the schedule obtained by the EDF algorithm. Since preemption is allowed, each task can be executed in disjointed time intervals. Without loss of generality, the schedule σ can be divided into *time slices* of one unit of time each. To simplify the formulation of the proof, let us define the following abbreviations:

$\sigma(t)$ identifies the task executing in the slice $[t, t+1)$.[1]
$E(t)$ identifies the ready task that, at time t, has the earliest deadline.
$t_E(t)$ is the time ($\geq t$) at which the next slice of task $E(t)$ begins its execution in the current schedule.

If $\sigma \neq \sigma_{EDF}$, then in σ there exists a time t such that $\sigma(t) \neq E(t)$. As illustrated in Fig. 3.4, the basic idea used in the proof is that interchanging the position of $\sigma(t)$ and $E(t)$ cannot increase the maximum lateness. If the schedule σ starts at time

[1] [a,b) denotes an interval of values x such that $a \leq x < b$.

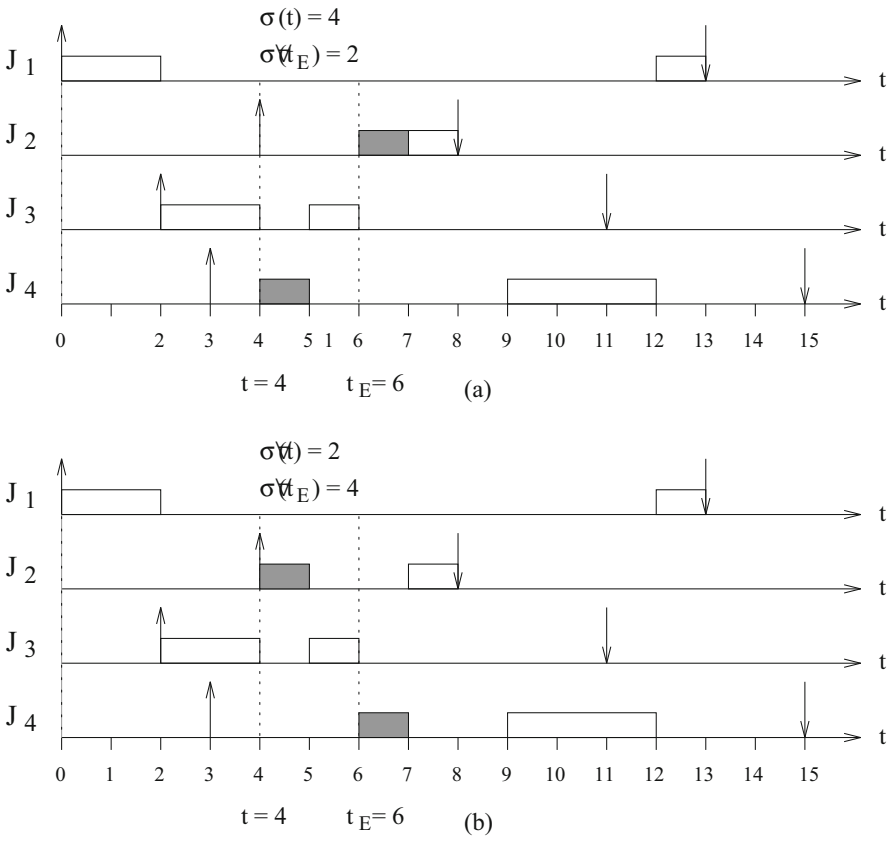


Fig. 3.4 Proof of the optimality of the EDF algorithm. (**a**) schedule σ at time $t = 4$. (**b**) new schedule obtained after a transposition

$t = 0$ and D is the latest deadline of the task set ($D = \max_i\{d_i\}$), then σ_{EDF} can be obtained from σ by at most D transpositions.

The algorithm used by Dertouzos to transform any schedule σ into an EDF schedule is illustrated in Fig. 3.5. For each time slice t, the algorithm verifies whether the task $\sigma(t)$ scheduled in the slice t is the one with the earliest deadline, $E(t)$. If it is, nothing is done; otherwise, a transposition takes place and the slices at t and t_E are exchanged (see Fig. 3.4). In particular, the slice of task $E(t)$ is anticipated at time t, while the slice of task $\sigma(t)$ is postponed at time t_E. Using the same argument adopted in the proof of Jackson's theorem, it is easy to show that after each transposition, the maximum lateness cannot increase; therefore, EDF is optimal.

By applying the interchange algorithm to the schedule shown in Fig. 3.4a, the first transposition occurs at time $t = 4$. At this time, in fact, the CPU is assigned to J_4, but the task with the earliest deadline is J_2, which is scheduled at time $t_E = 6$.

Fig. 3.5 Transformation algorithm used by Dertouzos to prove the optimality of EDF

```
Algorithm: interchange
{
        for (t=0 to D-1) {
            if (σ(t) ≠ E(t)) {
                σ(t_E) = σ(t);
                σ(t) = E(t);
            }
        }
}
```

As a consequence, the two slices in gray are exchanged, and the resulting schedule is shown in Fig. 3.4b. The algorithm examines all slices, until $t = D$, performing a slice exchange when necessary.

To show that a transposition preserves the schedulability, note that, at any instant, each slice in σ can be either anticipated or postponed up to t_E. If a slice is anticipated, the feasibility of that task is obviously preserved. If a slice of J_i is postponed at t_E and σ is feasible, it must be $(t_E + 1) \leq d_E$, being d_E the earliest deadline. Since $d_E \leq d_i$ for any i, then we have $t_E + 1 \leq d_i$, which guarantees the schedulability of the slice postponed at t_E.

3.3.2 Example

An example of schedule produced by the EDF algorithm on a set of five tasks is shown in Fig. 3.6. At time $t = 0$, tasks J_1 and J_2 arrive and, since $d_1 < d_2$, the processor is assigned to J_1, which completes at time $t = 1$. At time $t = 2$, when J_2 is executing, task J_3 arrives and preempts J_2, being $d_3 < d_2$. Note that, at time $t = 3$, the arrival of J_4 does not interrupt the execution of J_3, because $d_3 < d_4$. As J_3 is completed, the processor is assigned to J_2, which resumes and executes until completion. Then, J_4 starts at $t = 5$, but, at time $t = 6$, it is preempted by J_5, which has an earlier deadline. Task J_4 resumes at time $t = 8$, when J_5 is completed. Notice that all tasks meet their deadlines and the maximum lateness is $L_{max} = L_2 = 0$.

3.3.3 Guarantee

When tasks have dynamic activations and the arrival times are not known a priori, the guarantee test has to be done dynamically, whenever a new task enters the system. Let $\mathcal{J}$ be the current set of active tasks, which have been previously guaranteed, and let J_{new} be a newly arrived task. In order to accept J_{new} in the

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	3	2	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9

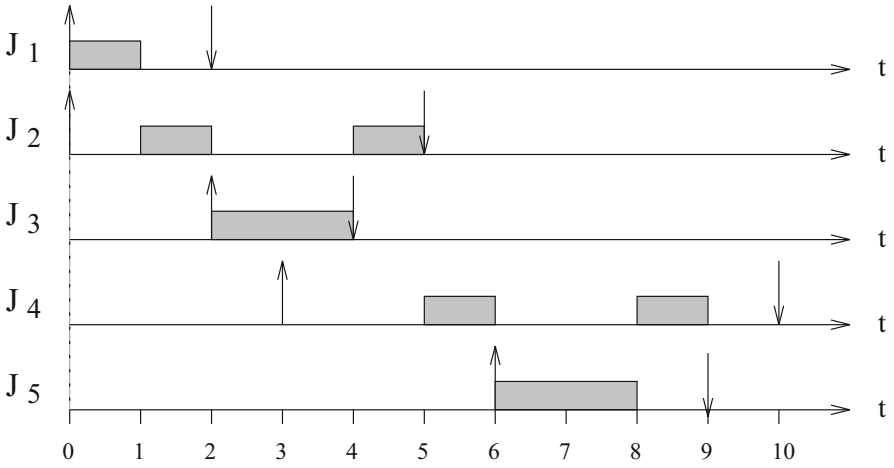


Fig. 3.6 Example of EDF schedule

system, we have to guarantee that the new task set $\mathcal{J}' = \mathcal{J} \cup \{J_{new}\}$ is also schedulable.

Following the same approach used in EDD, to guarantee that the set $\mathcal{J}'$ is feasibly schedulable by EDF, we need to show that, in the worst case, all tasks in $\mathcal{J}'$ will complete before their deadlines. This means that we have to show that, for each task, the worst-case finishing time f_i is less than or equal to its deadline d_i.

Without loss of generality, we can assume that all tasks in $\mathcal{J}'$ (including J_{new}) are ordered by increasing deadlines, so that J_1 is the task with the earliest deadline. Moreover, since tasks are preemptable, when J_{new} arrives at time t, some tasks could have been partially executed. Thus, let $c_i(t)$ be the remaining worst-case execution time of task J_i (notice that $c_i(t)$ has an initial value equal to C_i and can be updated whenever J_i is preempted). Hence, at time t, the worst-case finishing time of task J_i can easily be computed as

Fig. 3.7 EDF guarantee algorithm

```
Algorithm: EDF_guarantee(J, J_new)
{
        J' = J ∪ {J_new};    // ordered by deadline
        t = current_time();
        f_0 = t;
        for (each J_i ∈ J') {
            f_i = f_{i-1} + c_i(t);
            if (f_i > d_i) return(INFEASIBLE);
        }
        return(FEASIBLE);
}
```

$$f_i = t + \sum_{k=1}^{i} c_k(t).$$

Thus, the schedulability can be guaranteed by the following conditions:

$$\forall i = 1, \ldots, n \qquad \sum_{k=1}^{i} c_k(t) \leq d_i - t. \tag{3.2}$$

Noting that $f_i = f_{i-1} + c_i(t)$ (where $f_0 = t$ by definition), the dynamic guarantee test can be performed in $O(n)$ by executing the algorithm shown in Fig. 3.7.

3.4 Non-preemptive Scheduling

When preemption is not allowed and tasks can have arbitrary arrivals, the problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [GLLK79, LRKB77, KIM78]. Figure 3.8 illustrates an example that shows that EDF is no longer optimal if tasks cannot be preempted during their execution. In fact, although a feasible schedule exists for that task set (see Fig. 3.8a), EDF does not produce a feasible schedule (see Fig. 3.8b), since J_2 executes one unit of time after its deadline. This happens because EDF immediately assigns the processor to task J_1; thus, when J_2 arrives at time $t = 1$, J_1 cannot be preempted. J_2 can start only at time $t = 4$, after J_1 completion, but it is too late to meet its deadline.

Notice, however, that in the optimal schedule shown in Fig. 3.8a, the processor remains idle in the interval $[0, 1)$ although J_1 is ready to execute. If arrival times are not known a priori, then no on-line algorithm can decide whether to stay idle at time 0 or execute task J_1. A scheduling algorithm that does not permit the processor

	J_1	J_2
a_i	0	1
C_i	4	2
d_i	7	5

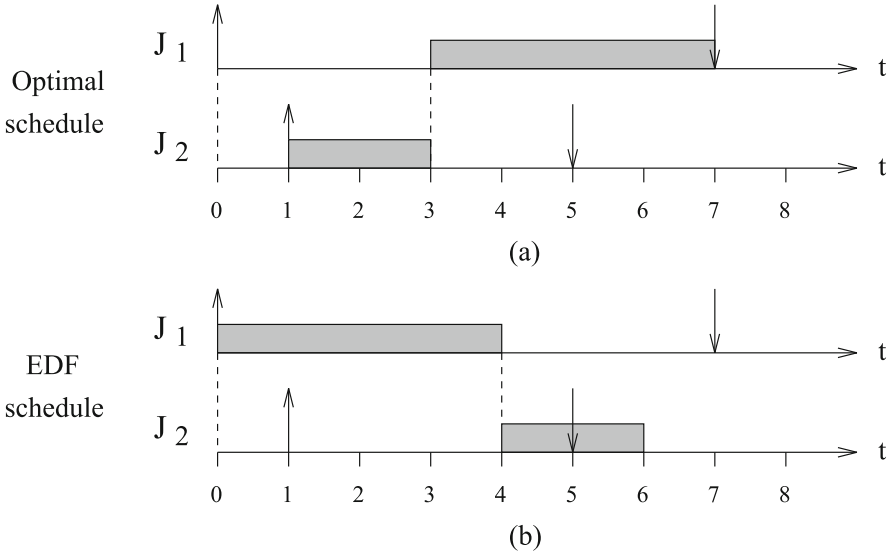


Fig. 3.8 EDF is not optimal in a non-preemptive model. In fact, although there exists a feasible schedule (**a**), the schedule produced by EDF (**b**) is infeasible

to be idle when there are active jobs is called a *non-idle* algorithm. By restricting to the case of non-idle scheduling algorithms, Jeffay, Stanat, and Martel [JSM91] proved that EDF is still optimal in a non-preemptive task model.

When arrival times are known a priori, non-preemptive scheduling problems are usually treated by branch-and-bound algorithms that perform well in the average case but degrade to exponential complexity in the worst case. The structure of the search space is a search tree, represented in Fig. 3.9, where the root is an *empty schedule*, an intermediate vertex is a *partial schedule*, and a terminal vertex (*leaf*) is a *complete schedule*. Since not all leaves correspond to feasible schedules, the goal of the scheduling algorithm is to search for a leaf that corresponds to a feasible schedule.

At each step of the search, the partial schedule associated with a vertex is extended by inserting a new task. If n is the total number of tasks in the set, the

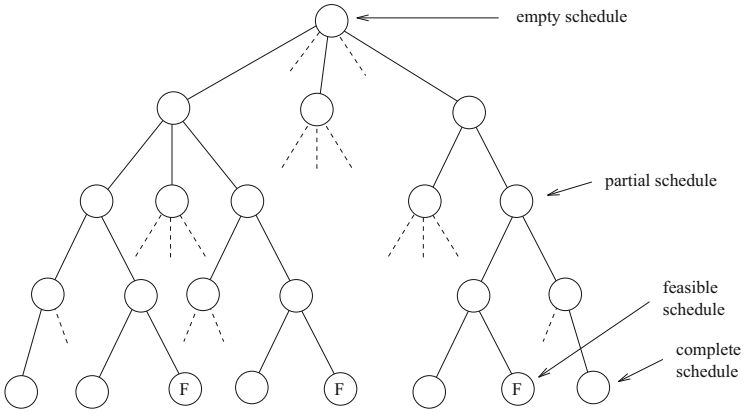


Fig. 3.9 Search tree for producing a non-preemptive schedule

length of a path from the root to a leaf (*tree depth*) is also n, whereas the total number of leaves is $n!$ (n factorial). An optimal algorithm, in the worst case, may make an exhaustive search to find the optimal schedule in such a tree, and this may require to analyze n paths of length $n!$, with a complexity of $O(n \cdot n!)$. Clearly, this approach is computationally intractable and cannot be used in practical systems when the number of tasks is high.

In this section, two scheduling approaches are presented, whose objective is to limit the search space and reduce the computational complexity of the algorithm. The first algorithm uses additional information to prune the tree and reduce the complexity in the average case. The second algorithm adopts suitable heuristics to follow promising paths on the tree and build a complete schedule in polynomial time. Heuristic algorithms may produce a feasible schedule in polynomial time; however, they do not guarantee to find it, since they do not explore all possible solutions.

3.4.1 Bratley's Algorithm (1 | *no_preem* | *feasible*)

The following algorithm was proposed by Bratley et al. in 1971 [BFR71] to solve the problem of finding a feasible schedule of a set of non-preemptive tasks with arbitrary arrival times. The algorithm starts with an empty schedule and, at each step of the search, visits a new vertex and adds a task in the partial schedule. With respect to the exhaustive search, Bratley's algorithm uses a pruning technique to determine when a current search can be reasonably abandoned. In particular, a branch is abandoned when:

- The addition of any node to the current path causes a missed deadline;
- A feasible schedule is found at the current path.

	J_1	J_2	J_3	J_4
a_i	4	1	1	0
C_i	2	1	2	2
d_i	7	5	6	4

Number in the node = scheduled task

Number outside the node = finishing time

$J_i^{\dagger}$ = task that misses its deadline


= feasible schedule

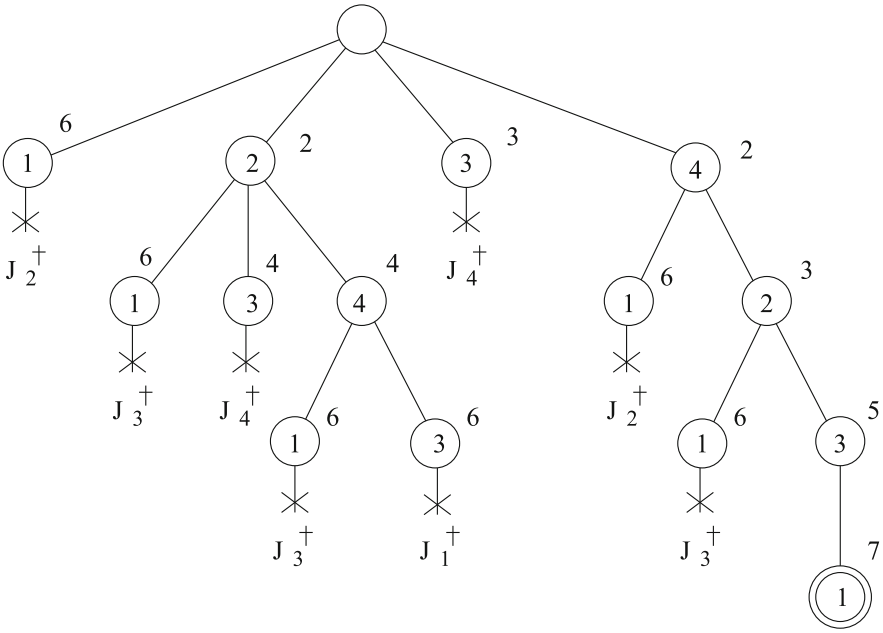


Fig. 3.10 Example of search performed by Bratley's algorithm

To better understand the pruning technique adopted by the algorithm, consider the task set shown in Fig. 3.10, which also illustrates the paths analyzed in the tree space.

To follow the evolution of the algorithm, the numbers inside each node of the tree indicate which task is being scheduled in that path, whereas the numbers beside the nodes represent the time at which the indicated task completes its execution. Whenever the addition of any node to the current path causes a missed deadline, the corresponding branch is abandoned and the task causing the timing fault is labeled with a (†).

In the example, the first task considered for extending the empty schedule is J_1, whose index is written in the first node of the leftmost branch of the tree. Since J_1

arrives at $t = 4$ and requires two units of processing time, its worst-case finishing time is $f_1 = 6$, indicated beside the correspondent node. Before expanding the branch, however, the pruning mechanism checks whether the addition of any node to the current path may cause a timing fault, and it discovers that task J_2 would miss its deadline, if added. As a consequence, the search on this branch is abandoned and a considerable amount of computation is avoided.

In the average case, pruning techniques are very effective for reducing the search space. Nevertheless, the worst-case complexity of the algorithm is still $O(n \cdot n!)$. For this reason, Bratley's algorithm can only be used in offline mode, when all task parameters (including the arrival times) are known in advance. This can be the case of a time-triggered system, where tasks are activated at predefined instants by a timer process.

As in most offline real-time systems, the resulting schedule produced by Bratley's algorithm can be stored in a data structure, called *task activation list*. Then, at runtime, a dispatcher simply extracts the next task from the activation list and puts it in execution.

3.4.2 The Spring Algorithm

Here we describe the scheduling algorithm adopted in the *Spring* kernel [SR87, SR91], a hard real-time kernel designed at the University of Massachusetts at Amherst by Stankovic and Ramamritham to support critical control applications in dynamic environments. The objective of the algorithm is to find a feasible schedule when tasks have different types of constraints, such as precedence relations, resource constraints, arbitrary arrivals, non-preemptive properties, and importance levels. The Spring algorithm is used in a distributed computer architecture and can also be extended to include fault-tolerance requirements.

Clearly, this problem is NP-hard, and finding a feasible schedule would be too expensive in terms of computation time, especially for dynamic systems. In order to make the algorithm computationally tractable even in the worst case, the search is driven by a *heuristic function* H, which actively directs the scheduling to a plausible path. On each level of the search, function H is applied to each of the tasks that remain to be scheduled. The task with the smallest value determined by the heuristic function H is selected to extend the current schedule.

The heuristic function is a very flexible mechanism that allows to easily define and modify the scheduling policy of the kernel. For example, if $H = a_i$ (arrival time), the algorithm behaves as first come first served, if $H = C_i$ (computation time), it works as Shortest Job First, whereas if $H = d_i$ (deadline), the algorithm is equivalent to Earliest Deadline First.

To consider resource constraints in the scheduling algorithm, each task J_i has to declare a binary array of resources $R_i = [R_1(i), \ldots, R_r(i)]$, where $R_k(i) = 0$ if J_i does not use resource R_k, and $R_k(i) = 1$ if J_i uses R_k in exclusive mode. Given a partial schedule, the algorithm determines, for each resource R_k, the earliest time

the resource is available. This time is denoted as EAT_k (Earliest Available Time). Thus, the earliest start time that a task J_i can begin the execution without blocking on shared resources is

$$T_{est}(i) = \max[a_i, \max_k(EAT_k)],$$

where a_i is the arrival time of J_i. Once T_{est} is calculated for all the tasks, a possible search strategy is to select the task with the smallest value of T_{est}. Composed heuristic functions can also be used to integrate relevant information on the tasks, such as

$$H = d + W \cdot C$$
$$H = d + W \cdot T_{est},$$

where W is a weight that can be adjusted for different application environments. Figure 3.11 shows some possible heuristic functions that can be used in Spring to direct the search process.

In order to handle precedence constraints, another factor E, called *eligibility*, is added to the heuristic function. A task becomes eligible to execute ($E_i = 1$) only when all its ancestors in the precedence graph are completed. If a task is not eligible, then $E_i = \infty$; hence, it cannot be selected for extending a partial schedule.

While extending a partial schedule, the algorithm determines whether the current schedule is *strongly feasible*, that is, also feasible by extending it with any of the remaining tasks. If a partial schedule is found not to be strongly feasible, the algorithm stops the search process and announces that the task set is not schedulable; otherwise, the search continues until a complete feasible schedule is met. Since a feasible schedule is reached through n nodes and each partial schedule requires the

H = a	First Come First Served (FCFS)
H = C	Shortest Job First (SJF)
H = d	Earliest Deadline First (EDF)
$H = T_{est}$	Earliest Start Time First (ESTF)
H = d + w C	EDF + SJF
$H = d + w\ T_{est}$	EDF + ESTF

Fig. 3.11 Example of heuristic functions that can be adopted in the Spring algorithm

evaluation of at most n heuristic functions, the complexity of the Spring algorithm is $O(n^2)$.

Backtracking can be used to continue the search after a failure. In this case, the algorithm returns to the previous partial schedule and extends it by the task with the second smallest heuristic value. To restrict the overhead of backtracking, however, the maximum number of possible backtracks must be limited. Another method to reduce the complexity is to restrict the number of evaluations of the heuristic function. To do that, if a partial schedule is found to be strongly feasible, the heuristic function is applied not to all the remaining tasks but only to the k remaining tasks with the earliest deadlines. Given that only k tasks are considered at each step, the complexity becomes $O(kn)$. If the value of k is constant (and small, compared to the task set size), then the complexity becomes linearly proportional to the number of tasks.

A disadvantage of the heuristic scheduling approach is that it is not optimal. This means that, if there exists a feasible schedule, the Spring algorithm may not find it.

3.5 Scheduling with Precedence Constraints

The problem of finding an optimal schedule for a set of tasks with precedence relations is in general NP-hard. However, optimal algorithms that solve the problem in polynomial time can be found under particular assumptions on the tasks. In this section we present two algorithms that minimize the maximum lateness by assuming synchronous activations and preemptive scheduling, respectively.

3.5.1 Latest Deadline First $(1 \mid prec, sync \mid L_{max})$

In 1973, Lawler [Law73] presented an optimal algorithm that minimizes the maximum lateness of a set of tasks with precedence relations and simultaneous arrival times. The algorithm is called *Latest Deadline First* (LDF) and can be executed in polynomial time with respect to the number of tasks in the set.

Given a set $\mathcal{J}$ of n tasks and a directed acyclic graph (DAG) describing their precedence relations, LDF builds the scheduling queue from tail to head: among the tasks without successors or whose successors have been all selected, LDF selects the task with the latest deadline to be scheduled last. This procedure is repeated until all tasks in the set are selected. At runtime, tasks are extracted from the head of the queue, so that the first task inserted in the queue will be executed last, whereas the last task inserted in the queue will be executed first.

The correctness of this rule is proved as follows. Let $\mathcal{J}$ be the complete set of tasks to be scheduled, let $\Gamma \subseteq \mathcal{J}$ be the subset of tasks without successors, and let J_l be the task in Γ with the latest deadline d_l. If σ is any schedule that does not follow the EDL rule, then the last scheduled task, say J_k, will not be the one with the latest

deadline; thus, $d_k \leq d_l$. Since J_l is scheduled before J_k, let us partition Γ into four subsets, so that $\Gamma = A \cup \{J_l\} \cup B \cup \{J_k\}$. Clearly, in σ the maximum lateness for Γ is greater or equal to $L_k = f - d_k$, where $f = \sum_{i=1}^{n} C_i$ is the finishing time of task J_k.

We show that moving J_l to the end of the schedule cannot increase the maximum lateness in Γ, which proves the optimality of LDF. To do that, let σ^* be the schedule obtained from σ after moving task J_l to the end of the queue and shifting all other tasks to the left. The two schedules σ and σ^* are depicted in Fig. 3.12. Clearly, in σ^* the maximum lateness for Γ is given by

$$L^*_{max}(\Gamma) = \max[L^*_{max}(A),\ L^*_{max}(B),\ L^*_k,\ L^*_l].$$

Each argument of the max function is no greater than $L_{max}(\Gamma)$. In fact,

- $L^*_{max}(A) = L_{max}(A) \leq L_{max}(\Gamma)$ because A is not moved;
- $L^*_{max}(B) \leq L_{max}(B) \leq L_{max}(\Gamma)$ because B starts earlier in σ^*;
- $L^*_k \leq L_k \leq L_{max}(\Gamma)$ because task J_k starts earlier in σ^*;
- $L^*_l = f - d_l \leq f - d_k \leq L_{max}(\Gamma)$ because $d_k \leq d_l$.

Since $L^*_{max}(\Gamma) \leq L_{max}(\Gamma)$, moving J_l to the end of the schedule does not increase the maximum lateness in Γ. This means that scheduling last the task J_l with the latest deadline minimizes the maximum lateness in Γ. Then, removing this task from $\mathcal{J}$ and repeating the argument for the remaining $n - 1$ tasks in the set $\mathcal{J} - \{J_l\}$, LDF can find the second-to-last task in the schedule, and so on. The complexity of the LDF algorithm is $O(n^2)$, since for each of the n steps, it needs to visit the precedence graph to find the subset Γ with no successors.

Consider the example depicted in Fig. 3.13, which shows the parameters of six tasks together with their precedence graph. The numbers beside each node of the graph indicate task deadlines. Figure 3.13 also shows the schedule produced by EDF to highlight the differences between the two approaches. The EDF schedule is constructed by selecting the task with the earliest deadline among the current

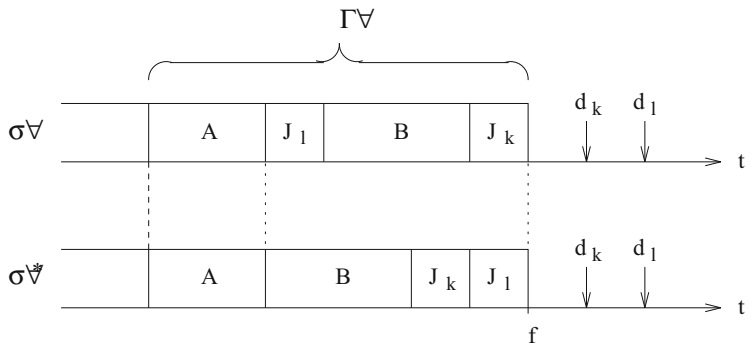


Fig. 3.12 Proof of LDF optimality

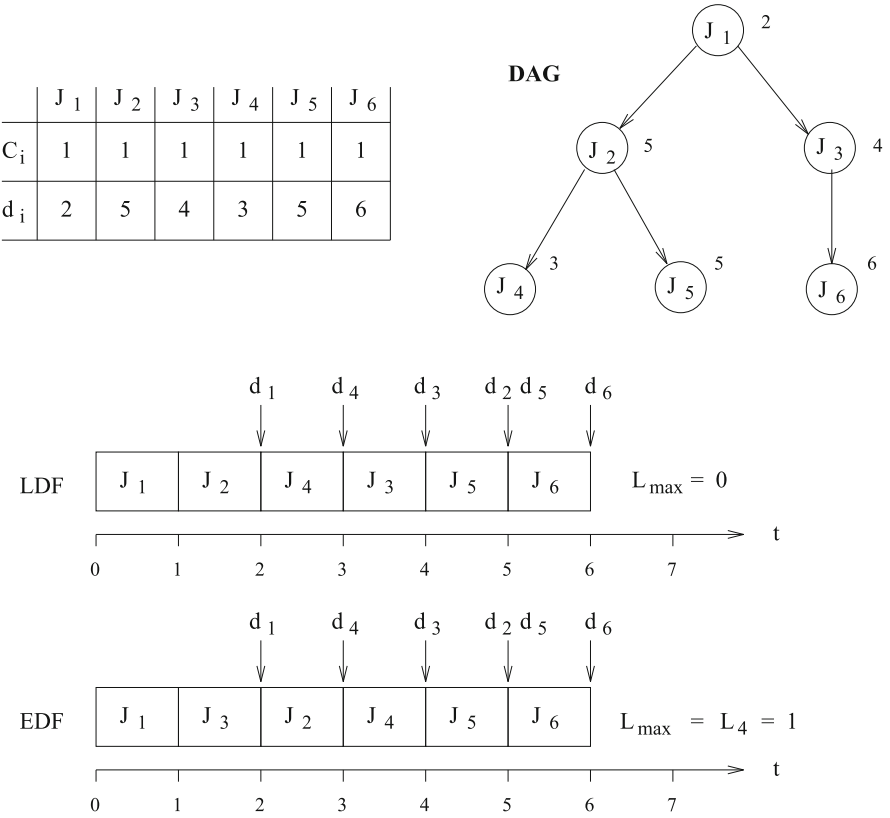


Fig. 3.13 Comparison between schedules produced by LDF and EDF on a set of tasks with precedence constraints

eligible tasks. Notice that EDF is not optimal under precedence constraints, since it achieves a greater L_{max} with respect to LDF.

3.5.2 *EDF with Precedence Constraints* ***(1 | prec, preem | L_{max})***

The problem of scheduling a set of n tasks with precedence constraints and dynamic activations can be solved in polynomial time complexity only if tasks are preemptable. In 1990, Chetto, Silly, and Bouchentouf [CSB90] presented an algorithm that solves this problem in elegant fashion. The basic idea of their approach is to transform a set $\mathcal{J}$ of dependent tasks into a set $\mathcal{J}^*$ of independent tasks by an adequate modification of timing parameters. Then, tasks are scheduled by the Earliest Deadline First (EDF) algorithm. The transformation algorithm

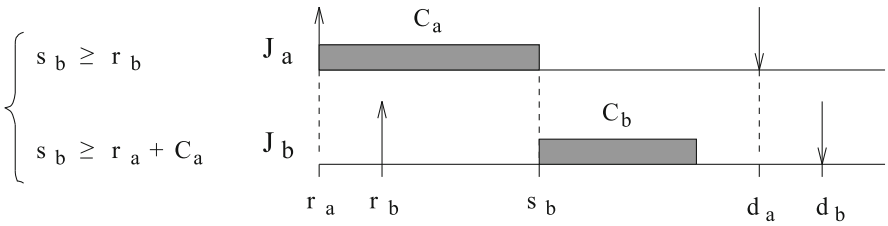


Fig. 3.14 If $J_a \rightarrow J_b$, then the release time of J_b can be replaced by $\max(r_b, r_a + C_a)$

ensures that $\mathcal{J}$ is schedulable and the precedence constraints are obeyed if and only if $\mathcal{J}^*$ is schedulable. Basically, all release times and deadlines are modified so that each task cannot start before its predecessors and cannot preempt their successors.

3.5.2.1 Modification of the Release Times

The rule for modifying tasks' release times is based on the following observation. Given two tasks J_a and J_b, such that $J_a \rightarrow J_b$ (i.e., J_a is an immediate predecessor of J_b), then in any valid schedule that meets precedence constraints, the following conditions must be satisfied (see Fig. 3.14):

$s_b \geq r_b$ (i.e., J_b must start the execution not earlier than its release time);
$s_b \geq r_a + C_a$ (i.e., J_b must start the execution not earlier than the minimum finishing time of J_a).

Therefore, the release time r_b of J_b can be replaced by the maximum between r_b and $(r_a + C_a)$ without changing the problem. Let r_b^* be the new release time of J_b. Then,

$$r_b^* = \max(r_b, r_a + C_a).$$

The algorithm that modifies the release times can be implemented in $O(n^2)$ and can be described as follows:

1. For any initial node of the precedence graph, set $r_i^* = r_i$.
2. Select a task J_i such that its release time has not been modified but the release times of all immediate predecessors J_h have been modified. If no such task exists, exit.
3. Set $r_i^* = \max[r_i, \max(r_h^* + C_h : J_h \rightarrow J_i)]$.
4. Return to step 2.

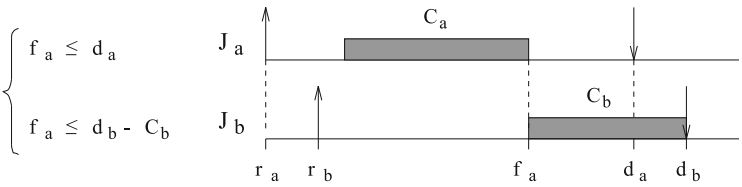


Fig. 3.15 If $J_a \rightarrow J_b$, then the deadline of J_a can be replaced by $\min(d_a, d_b - C_b)$

3.5.2.2 Modification of the Deadlines

The rule for modifying tasks' deadlines is based on the following observation. Given two tasks J_a and J_b, such that $J_a \rightarrow J_b$ (i.e., J_a is an immediate predecessor of J_b), then in any feasible schedule that meets the precedence constraints, the following conditions must be satisfied (see Fig. 3.15):

$f_a \leq d_a$ (i.e., J_a must finish the execution within its deadline);
$f_a \leq d_b - C_b$ (i.e., J_a must finish the execution not later than the maximum start time of J_b).

Therefore, the deadline d_a of J_a can be replaced by the minimum between d_a and $(d_b - C_b)$ without changing the problem. Let d_a^* be the new deadline of J_a. Then,

$$d_a^* = \min(d_a, d_b - C_b).$$

The algorithm that modifies the deadlines can be implemented in $O(n^2)$ and can be described as follows:

1. For any terminal node of the precedence graph, set $d_i^* = d_i$.
2. Select a task J_i such that its deadline has not been modified but the deadlines of all immediate successors J_k have been modified. If no such task exists, exit.
3. Set $d_i^* = \min[d_i, \ \min(d_k^* - C_k : J_i \rightarrow J_k)]$.
4. Return to step 2.

3.5.2.3 Proof of Optimality

The transformation algorithm ensures that if there exists a feasible schedule for the modified task set $\mathcal{J}^*$ under EDF, then the original task set $\mathcal{J}$ is also schedulable, that is, all tasks in $\mathcal{J}$ meet both precedence and timing constraints. In fact, if $\mathcal{J}^*$ is schedulable, all modified tasks start at or after time r_i^* and are completed at or before time d_i^*. Since $r_i^* \geq r_i$ and $d_i^* \leq d_i$, the schedulability of $\mathcal{J}^*$ implies that $\mathcal{J}$ is also schedulable.

To show that precedence relations in $\mathcal{J}$ are not violated, consider the example illustrated in Fig. 3.16, where J_1 must precede J_2 (i.e., $J_1 \rightarrow J_2$), but J_2 arrives before J_1 and has an earlier deadline. Clearly, if the two tasks are executed

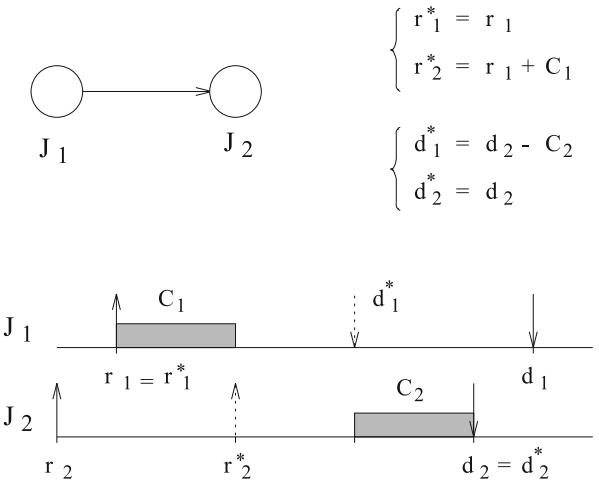


Fig. 3.16 The transformation algorithm preserves the timing and the precedence constraints

under EDF, their precedence relation cannot be met. However, if we apply the transformation algorithm, the time constraints are modified as follows:

$$\begin{cases} r_1^* = r_1 \\ r_2^* = \max(r_2, r_1 + C_1) \end{cases} \qquad \begin{cases} d_1^* = \min(d_1, d_2 - C_2) \\ d_2^* = d_2 \end{cases}$$

This means that, since $r_2^* > r_1^*$, J_2 cannot start before J_1. Moreover, J_2 cannot preempt J_1 because $d_1^* < d_2^*$ and, based on EDF, the processor is assigned to the task with the earliest deadline. Hence, the precedence relation is respected.

In general, for any pair of tasks such that $J_i \prec J_j$, we have $r_i^* \leq r_j^*$ and $d_i^* \leq d_j^*$. This means that, if J_i is in execution, then all successors of J_i cannot start before r_i because $r_i^* \leq r_j^*$. Moreover, they cannot preempt J_i because $d_i^* \leq d_j^*$ and, according to EDF, the processor is assigned to the ready task having the earliest deadline. Therefore, both timing and precedence constraints specified for task set $\mathcal{J}$ are guaranteed by the schedulability of the modified set $\mathcal{J}^*$.

3.6 Summary

The scheduling algorithms described in this chapter for handling real-time tasks with aperiodic arrivals can be compared in terms of assumptions on the task set and computational complexity. Figure 3.17 summarizes the main characteristics of such algorithms and can be used for selecting the most appropriate scheduling policy for a particular problem.

	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	**EDD** (Jackson '55) *O(n logn)* Optimal	**EDF** (Horn '74) $O(n^2)$ Optimal	***Tree search*** (Bratley '71) *O(n n!)* Optimal
precedence constraints	**LDF** (Lawler '73) $O(n^2)$ Optimal	**EDF** * (Chetto et al. '90) $O(n^2)$ Optimal	**Spring** (Stankovic & Ramamritham '87) $O(n^2)$ Heuristic

Fig. 3.17 Scheduling algorithms for aperiodic tasks

Exercises

3.1 Check whether the Earliest Due Date (EDD) algorithm produces a feasible schedule for the following task set (all tasks are synchronous and start at time $t = 0$).

	J_1	J_2	J_3	J_4
C_i	4	5	2	3
D_i	9	16	5	10

3.2 Write an algorithm for finding the maximum lateness of a task set scheduled by the EDD algorithm.

3.3 Draw the full scheduling tree for the following set of non-preemptive tasks and mark the branches that are pruned by Bratley's algorithm.

	J_1	J_2	J_3	J_4
a_i	0	4	2	6
C_i	6	2	4	2
D_i	18	8	9	10

3.4 On the scheduling tree developed in the previous exercise, find the path produced by the Spring algorithm using the following heuristic function: $H = a + C + D$. Then, find a heuristic function that produces a feasible schedule.

3.5 Given seven tasks, A, B, C, D, E, F, and G, construct the precedence graph from the following precedence relations:

$$
\begin{array}{ll}
A \rightarrow C & \\
B \rightarrow C & B \rightarrow D \\
C \rightarrow E & C \rightarrow F \\
D \rightarrow F & D \rightarrow G
\end{array}
$$

Then, assuming that all tasks arrive at time $t = 0$, have deadline D = 25, and computation times 2, 3, 3, 5, 1, 2, 5, respectively, modify their arrival times and deadlines to schedule them by EDF.

Chapter 4
Periodic Task Scheduling

4.1 Introduction

In many real-time control applications, periodic activities represent the major computational demand in the system. Periodic tasks typically arise from sensory data acquisition, low-level servoing, control loops, action planning, and system monitoring. Such activities need to be cyclically executed at specific rates, which can be derived from the application requirements. Some specific examples of real-time applications are illustrated in Chap. 11.

When a control application consists of several concurrent periodic tasks with individual timing constraints, the operating system has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline (which, in general, could be different than its period).

In this chapter three basic algorithms for handling periodic tasks are described in detail: Rate Monotonic, Earliest Deadline First, and Deadline Monotonic. Schedulability analysis is performed for each algorithm in order to derive a guarantee test for generic task sets. To facilitate the description of the scheduling results presented in this chapter, the following notation is introduced:

Γ denotes a set of periodic tasks;
τ_i denotes a generic periodic task;
$\tau_{i,j}$ denotes the jth instance of task τ_i;
$r_{i,j}$ denotes the release time of the jth instance of task τ_i;
Φ_i denotes the *phase* of task τ_i, that is, the release time of its first instance ($\Phi_i = r_{i,1}$);
D_i denotes the relative deadline of task τ_i;
$d_{i,j}$ denotes the absolute deadline of the jth instance of task τ_i ($d_{i,j} = \Phi_i + (j-1)T_i + D_i$).
$s_{i,j}$ denotes the start time of the jth instance of task τ_i, that is, the time at which it starts executing.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_4

$f_{i,j}$ denotes the finishing time of the jth instance of task τ_i, that is, the time at which it completes the execution.

Moreover, in order to simplify the schedulability analysis, the following hypotheses are assumed on the tasks:

A1. The instances of a periodic task τ_i are regularly activated at a constant rate. The interval T_i between two consecutive activations is the *period* of the task.
A2. All instances of a periodic task τ_i have the same worst-case execution time C_i.
A3. All instances of a periodic task τ_i have the same relative deadline D_i, which is equal to the period T_i.
A4. All tasks in Γ are independent; that is, there are no precedence relations and no resource constraints.

In addition, the following assumptions are implicitly made:

A5. No task can suspend itself, for example, on I/O operations.
A6. All tasks are released as soon as they arrive.
A7. All overheads in the kernel are assumed to be zero.

Notice that assumptions A1 and A2 are not restrictive because in many control applications, each periodic activity requires the execution of the same routine at regular intervals; therefore, both T_i and C_i are constant for every instance. On the other hand, assumptions A3 and A4 could be too tight for practical applications. However, the four assumptions are initially considered to derive some important results on periodic task scheduling; then, such results are extended to deal with more realistic cases, in which assumptions A3 and A4 are relaxed. In particular, the problem of scheduling a set of tasks under resource constraints is considered in detail in Chap. 7.

In those cases in which the assumptions A1, A2, A3, and A4 hold, a periodic task τ_i can be completely characterized by the following three parameters: its phase Φ_i, its period T_i, and its worst-case computation time C_i. Thus, a set of periodic tasks can be denoted by

$$\Gamma = \{\tau_i(\Phi_i, T_i, C_i),\ i = 1, \ldots, n\}.$$

The release time $r_{i,k}$ and the absolute deadline $d_{i,k}$ of the generic kth instance can then be computed as

$$r_{i,k} = \Phi_i + (k-1)T_i$$
$$d_{i,k} = r_{i,k} + T_i = \Phi_i + kT_i.$$

Other parameters that are typically defined on a periodic task are described below:

- **Hyperperiod**. It is the minimum interval of time after which the schedule repeats itself. If H is the length of such an interval, then the schedule in $[0, H)$ is the

same as that in $[kH, (k+1)H]$, for any integer $k > 0$. For a set of periodic tasks synchronously activated at time $t = 0$, the hyperperiod is given by the least common multiple of the periods:

$$H = lcm(T_1, \ldots, T_n).$$

- **Job response time**. It is the time (measured from the release time) at which the job is terminated:

$$R_{i,k} = f_{i,k} - r_{i,k}.$$

- **Task response time**. It is the maximum response time among all the jobs:

$$R_i = \max_k R_{i,k}.$$

- **Critical instant** of a task. It is the time at which the release of a task will produce the largest task response time.
- **Critical time zone** of a task. It is the interval between the critical instant and the response time of the corresponding request of the task.
- **Relative Start Time Jitter** of a task. It is the maximum deviation of the start time of two consecutive instances:

$$RRJ_i = \max_k |(s_{i,k} - r_{i,k}) - (s_{i,k-1} - r_{i,k-1})|.$$

- **Absolute Start Time Jitter** of a task. It is the maximum deviation of the start time among all instances:

$$ARJ_i = \max_k (s_{i,k} - r_{i,k}) - \min_k (s_{i,k} - r_{i,k}).$$

- **Relative Finishing Jitter** of a task. It is the maximum deviation of the finishing time of two consecutive instances:

$$RFJ_i = \max_k |(f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1})|.$$

- **Absolute Finishing Jitter** of a task. It is the maximum deviation of the finishing time among all instances:

$$AFJ_i = \max_k (f_{i,k} - r_{i,k}) - \min_k (f_{i,k} - r_{i,k}).$$

In this context, a periodic task τ_i is said to be *feasible* if all its instances finish within their deadlines. A task set Γ is said to be *schedulable* (or *feasible*) if all tasks in Γ are feasible.

4.1.1 Processor Utilization Factor

Given a set Γ of n periodic tasks, the *processor utilization factor* U is the fraction of processor time spent in the execution of the task set [LL73]. Since C_i/T_i is the fraction of processor time spent in executing task τ_i, the utilization factor for n tasks is given by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

The processor utilization factor provides a measure of the computational load on the CPU due to the periodic task set. Although the CPU utilization can be improved by increasing tasks' computation times or by decreasing their periods, there exists a maximum value of U below which Γ is schedulable and above which Γ is not schedulable. Such a limit depends on the task set (i.e., on the particular relations among tasks' periods) and on the algorithm used to schedule the tasks. Let $U_{ub}(\Gamma, A)$ be the upper bound of the processor utilization factor for a task set Γ under a given algorithm A. When $U = U_{ub}(\Gamma, A)$, the set Γ is said to *fully utilize* the processor. In this situation, Γ is schedulable by A, but an increase in the computation time in any of the tasks will make the set infeasible.

Figure 4.1 shows an example of two tasks (where τ_1 has higher priority than τ_2) in which $U_{ub} = 5/6 \simeq 0.833$. In fact, if any execution time is increased by epsilon, the task set becomes infeasible, since the first job of τ_2 misses its deadline. Figure 4.2 shows another example in which $U_{ub} = 0.9$. Notice that setting $T_1 = 4$ and $T_2 = 8$, U_{ub} becomes 1.0.

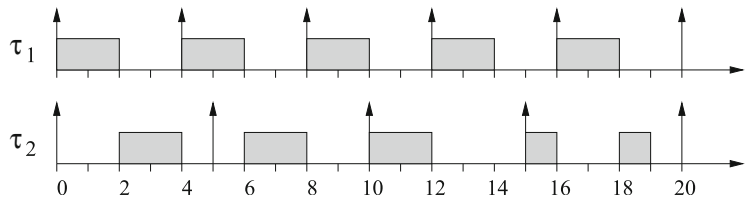


Fig. 4.1 A task set with $U_{ub} = 5/6$

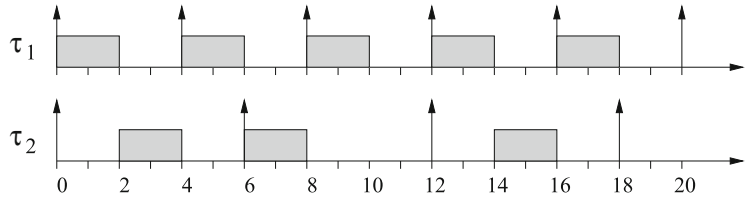


Fig. 4.2 A task set with $U_{ub} = 0.9$

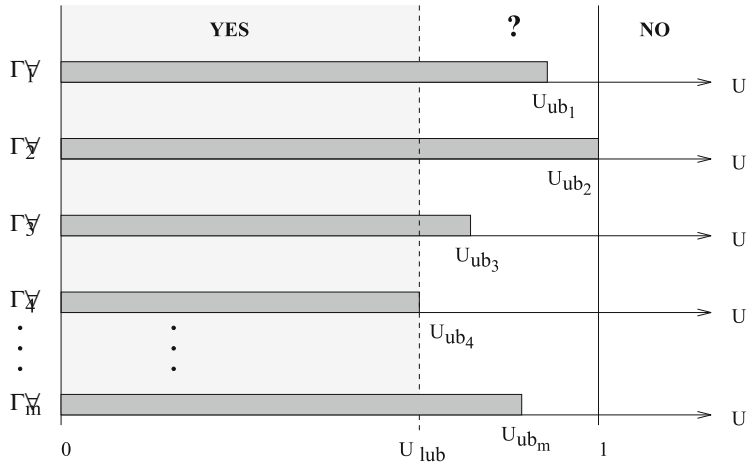


Fig. 4.3 Meaning of the least upper bound of the processor utilization factor

For a given algorithm A, the *least upper bound* $U_{lub}(A)$ of the processor utilization factor is the minimum of the utilization factors over all task sets that fully utilize the processor:

$$U_{lub}(A) = \min_{\Gamma} U_{ub}(\Gamma, A).$$

Figure 4.3 graphically illustrates the meaning of U_{lub} for a scheduling algorithm A. The task sets Γ_i shown in the figure differ for the number of tasks and for the configuration of their periods. When scheduled by the algorithm A, each task set Γ_i fully utilizes the processor when its utilization factor U_i (varied by changing tasks' computation times) reaches a particular upper bound U_{ub_i}. If $U_i \leq U_{ub_i}$, then Γ_i is schedulable, else Γ_i is not schedulable. Notice that each task set may have a different upper bound. Since $U_{lub}(A)$ is the minimum of all upper bounds, any task set having a processor utilization factor below $U_{lub}(A)$ is certainly schedulable by A.

U_{lub} defines an important characteristic of a scheduling algorithm because it allows to easily verify the schedulability of a task set. In fact, any task set whose processor utilization factor is less than or equal to this bound is schedulable by the algorithm. On the other hand, when $U_{lub} < U \leq 1.0$, the schedulability can be achieved only if the task periods are suitably related.

If the utilization factor of a task set is greater than 1.0, the task set cannot be scheduled by any algorithm. To show this result, let H be the hyperperiod of the task set. If $U > 1$, we also have $UH > H$, which can be written as

$$\sum_{i=1}^{n} \frac{H}{T_i} C_i > H.$$

The factor (H/T_i) represents the (integer) number of times τ_i is executed in the hyperperiod, whereas the quantity $(H/T_i)C_i$ is the total computation time requested by τ_i in the hyperperiod. Hence, the sum on the left-hand side represents the total computational demand requested by the task set in $[0, H)$. Clearly, if the total demand exceeds the available processor time, there is no feasible schedule for the task set.

4.2 Timeline Scheduling

Timeline scheduling (TS), also known as a cyclic executive, is one of the most used approaches to handle periodic tasks in defense military systems and traffic control systems. The method consists in dividing the temporal axis into slices of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements. A timer synchronizes the activation of the tasks at the beginning of each time slice. In order to illustrate this method, consider the following example, in which three tasks, A, B, and C, need to be executed with a frequency of 40, 20, and 10 Hz, respectively. By analyzing the task periods ($T_A = 25$ ms, $T_B = 50$ ms, $T_A = 100$ ms), it is easy to verify that the optimal length for the time slice is 25 ms, which is the greatest common divisor of the periods. Hence, to meet the required frequencies, task A needs to be executed every time slice, task B every two slices, and task C every four slices. A possible scheduling solution for this task set is illustrated in Fig. 4.4.

The duration of the time slice is also called a minor cycle, whereas the minimum period after which the schedule repeats itself is called a major cycle. In general, the major cycle is equal to the least common multiple of all the periods (in the example it is equal to 100 ms). In order to guarantee a priori that a schedule is feasible on a particular processor, it is sufficient to know the task worst-case execution times and verify that the sum of the executions within each time slice is less than or equal to the minor cycle. In the example shown in Fig. 4.4, if C_A, C_B, and C_C denote the execution times of the tasks, it is sufficient to verify that

$$\begin{cases} C_A + C_B \leq 25\,\text{ms} \\ C_A + C_C \leq 25\,\text{ms} \end{cases}$$

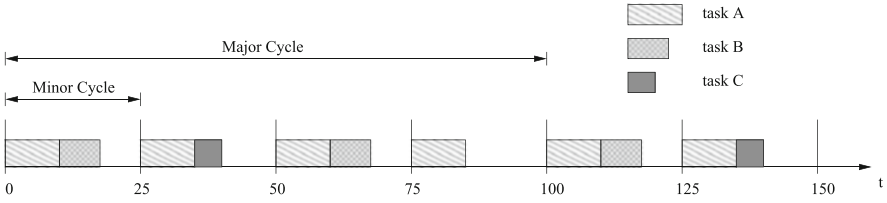


Fig. 4.4 Example of timeline scheduling

The major relevant advantage of timeline scheduling is its simplicity. The method can be implemented by programming a timer to interrupt with a period equal to the minor cycle and by writing a main program that calls the tasks in the order given in the major cycle, inserting a time synchronization point at the beginning of each minor cycle. Since the task sequence is not decided by a scheduling algorithm in the kernel, but it is triggered by the calls made by the main program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same, it can be easily visualized, and it is not affected by jitter (i.e., task start times and response times are not subject to large variations).

In spite of these advantages, timeline scheduling has some problems. For example, it is very fragile during overload conditions. If a task does not terminate at the minor cycle boundary, we can either let it continue or abort it. In both cases, however, the system may enter in a risky situation. In fact, if we leave the failing task in execution, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). On the other hand, if the failing task is aborted while it is modifying some global data structure, the system may be left in an inconsistent state, jeopardizing the correct system behavior.

Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch. Considering the previous example, if task B is updated to B' and the code change is such that $C_A + C_{B'} > 25$ ms, then we have to divide B' in two or more pieces to be allocated in the available intervals of the timeline. Changing the task frequencies may cause even more radical changes in the schedule. For example, if the frequency of task B changes from 20 Hz to 25 Hz (i.e., T_B changes from 50 to 40 ms), the previous schedule is not valid anymore, because the new minor cycle is equal to 5 ms and the new major cycle is equal to 200 ms. Note that after this change, being the minor cycle much shorter than before, all the procedures may need to be divided into small pieces to fit in the new time slots.

Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence. The problems outlined above can be solved by using priority-based scheduling algorithms.

4.3 Rate Monotonic Scheduling

The Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates (i.e., with shorter periods) will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: a priority P_i is assigned to the task before execution and does not change over time. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with shorter period.

In 1973, Liu and Layland [LL73] showed that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. Liu and Layland also derived the least upper bound of the processor utilization factor for a generic set of n periodic tasks. These issues are discussed in detail in the following subsections.

4.3.1 Optimality

In order to prove the optimality of the RM algorithm, we first show that a critical instant for any task occurs whenever the task is released simultaneously with all higher-priority tasks. Let $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be the set of periodic tasks ordered by increasing periods, with τ_n being the task with the longest period. According to RM, τ_n will also be the task with the lowest priority.

As shown in Fig. 4.5a, the response time of task τ_n is delayed by the interference of τ_i with higher priority. Moreover, from Fig. 4.5b it is clear that advancing the release time of τ_i the completion time of τ_n may only increase, reaching its maximum value when $\Phi_i = \Phi_n$. Note that if we continue to advance the activation of τ_i, that is, for $\Phi_n - C_i < \Phi_i < \Phi_n$, the interference on τ_n due to the first instance of τ_i reduces; hence, the response time of τ_n is largest when it is released simultaneously with τ_i. Repeating the argument for all τ_i, $i = 1, \ldots, n-1$, we can prove that the worst response time of τ_n occurs when it is released simultaneously with all higher-priority tasks. Since the same argument applies to any task whose

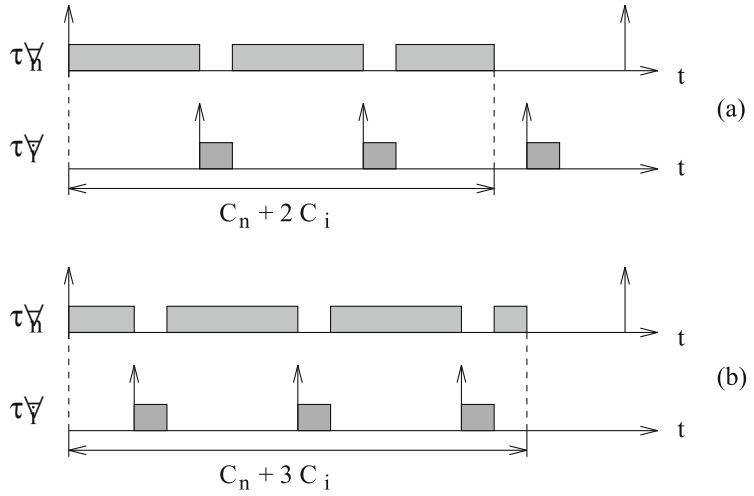


Fig. 4.5 (**a**) The response time of task τ_n is delayed by the interference of τ_i with higher priority. (**b**) The interference may increase advancing the release time of τ_i

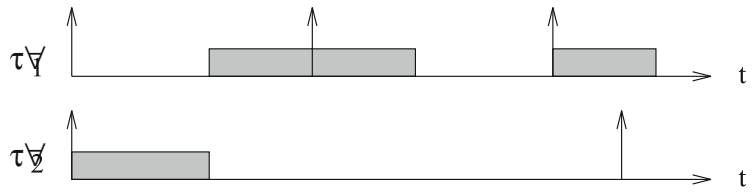


Fig. 4.6 Tasks scheduled by an algorithm different from RM

priority is not the highest, we can conclude that the worst response time of any task occurs when it is released simultaneously with all higher-priority tasks.

A first consequence of this result is that the schedulability of a task set can easily be checked when all the tasks are activated at their critical instants. Specifically, if all the tasks are guaranteed to be feasible at their critical instants, then the task set is also schedulable in any other phasing condition. Based on this result, the optimality of RM can be proved by showing that, when all tasks are activated at their critial instant, if a task set is schedulable by an arbitrary priority assignment, then it is also schedulable by RM.

Consider a set of two periodic tasks τ_1 and τ_2, with $T_1 < T_2$. If priorities are not assigned according to RM, then task T_2 will receive the highest priority. This situation is depicted in Fig. 4.6, from which it is easy to see that, at critical instants, the schedule is feasible if the following inequality is satisfied:

$$C_1 + C_2 \leq T_1. \tag{4.1}$$

On the other hand, if priorities are assigned according to RM, task T_1 will receive the highest priority. In this situation, illustrated in Fig. 4.7, in order to guarantee a feasible schedule, two cases must be considered. Let $F = \lfloor T_2/T_1 \rfloor$ be the number[1] of periods of τ_1 entirely contained in T_2.

Case 1. The computation time C_1 is short enough that all requests of τ_1 within the critical time zone of τ_2 are completed before the second request of τ_2. That is, $C_1 \leq T_2 - FT_1$.

In this case, from Fig. 4.7a we can see that the task set is schedulable if

$$(F + 1)C_1 + C_2 \leq T_2. \tag{4.2}$$

We now show that inequality (4.1) implies (4.2). In fact, by multiplying both sides of (4.1) by F, we obtain:

$$FC_1 + FC_2 \leq FT_1,$$

[1] $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x, whereas $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

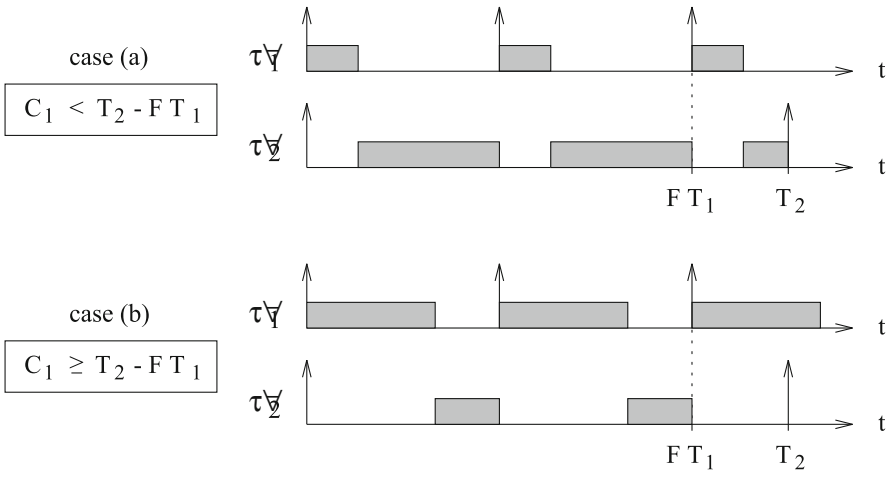


Fig. 4.7 Schedule produced by RM in two different conditions

and, since $F \geq 1$, we can write:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1.$$

Adding C_1 to each member, we get:

$$(F + 1)C_1 + C_2 \leq FT_1 + C_1.$$

Since we assumed that $C_1 \leq T_2 - FT_1$, we have:

$$(F + 1)C_1 + C_2 \leq FT_1 + C_1 \leq T_2,$$

which satisfies (4.2).

Case 2. The execution of the last request of τ_1 in the critical time zone of τ_2 overlaps the second request of τ_2. That is, $C_1 \geq T_2 - FT_1$.

In this case, from Fig. 4.7b we can see that the task set is schedulable if

$$FC_1 + C_2 \leq FT_1. \tag{4.3}$$

Again, inequality (4.1) implies (4.3). In fact, by multiplying both sides of (4.1) by F, we obtain:

$$FC_1 + FC_2 \leq FT_1,$$

and, since $F \geq 1$, we can write:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1,$$

which satisfies (4.3).

Basically, it has been shown that, given two periodic tasks τ_1 and τ_2, with $T_1 < T_2$, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM. That is, RM is optimal. This result can easily be extended to a set of n periodic tasks. We now show how to compute the least upper bound U_{lub} of the processor utilization factor for the RM algorithm. The bound is first determined for two tasks and then extended for an arbitrary number of tasks.

4.3.2 Calculation of U_{lub} for Two Tasks

Consider a set of two periodic tasks τ_1 and τ_2, with $T_1 < T_2$. In order to compute U_{lub} for RM, we have to:

- Assign priorities to tasks according to RM, so that τ_1 is the task with the highest priority;
- Compute the upper bound U_{ub} for the set by setting tasks' computation times to fully utilize the processor;
- Minimize the upper bound U_{ub} with respect to all other task parameters.

As before, let $F = \lfloor T_2/T_1 \rfloor$ be the number of periods of τ_1 entirely contained in T_2. Without loss of generality, the computation time C_2 is adjusted to fully utilize the processor. Again two cases must be considered.

Case 1. The computation time C_1 is short enough that all requests of τ_1 within the critical time zone of τ_2 are completed before the second request of τ_2. That is, $C_1 \leq T_2 - FT_1$.

In this situation, depicted in Fig. 4.8, the largest possible value of C_2 is

$$C_2 = T_2 - C_1(F + 1),$$

and the corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - C_1(F+1)}{T_2} = \\ &= 1 + \frac{C_1}{T_1} - \frac{C_1}{T_2}(F+1) = \\ &= 1 + \frac{C_1}{T_2}\left[\frac{T_2}{T_1} - (F+1)\right]. \end{aligned}$$

Since the quantity in square brackets is negative, U_{ub} is monotonically decreasing in C_1, and, being $C_1 \leq T_2 - FT_1$, the minimum of U_{ub} occurs for

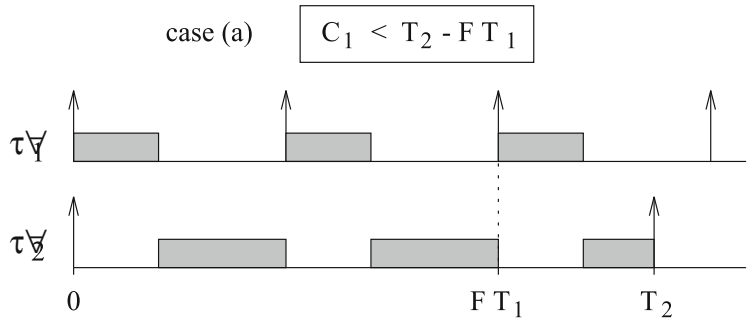


Fig. 4.8 The second request of τ_2 is released when τ_1 is idle

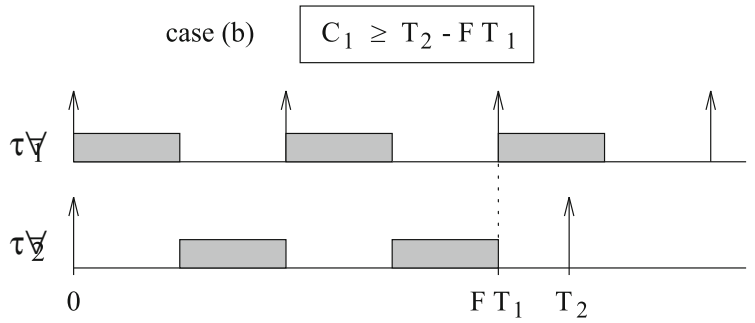


Fig. 4.9 The second request of τ_2 is released when τ_1 is active

$$C_1 = T_2 - FT_1.$$

Case 2. The execution of the last request of τ_1 in the critical time zone of τ_2 overlaps the second request of τ_2. That is, $C_1 \geq T_2 - FT_1$.

In this situation, depicted in Fig. 4.9, the largest possible value of C_2 is

$$C_2 = (T_1 - C_1)F,$$

and the corresponding upper bound U_{ub} is

$$\begin{aligned} U_{ub} &= \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{(T_1 - C_1)F}{T_2} = \\ &= \frac{T_1}{T_2}F + \frac{C_1}{T_1} - \frac{C_1}{T_2}F = \\ &= \frac{T_1}{T_2}F + \frac{C_1}{T_2}\left[\frac{T_2}{T_1} - F\right]. \end{aligned} \tag{4.4}$$

Since the quantity in square brackets is positive, U_{ub} is monotonically increasing in C_1 and, being $C_1 \geq T_2 - FT_1$, the minimum of U_{ub} occurs for

$$C_1 = T_2 - FT_1.$$

In both cases, the minimum value of U_{ub} occurs for

$$C_1 = T_2 - T_1 F.$$

Hence, using the minimum value of C_1, from Eq. (4.4), we have:

$$\begin{aligned} U &= \frac{T_1}{T_2}F + \frac{C_1}{T_2}\left(\frac{T_2}{T_1} - F\right) = \\ &= \frac{T_1}{T_2}F + \frac{(T_2 - T_1 F)}{T_2}\left(\frac{T_2}{T_1} - F\right) = \\ &= \frac{T_1}{T_2}\left[F + \left(\frac{T_2}{T_1} - F\right)\left(\frac{T_2}{T_1} - F\right)\right]. \end{aligned} \tag{4.5}$$

To simplify the notation, let $G = T_2/T_1 - F$. Thus,

$$\begin{aligned} U &= \frac{T_1}{T_2}(F + G^2) = \frac{(F + G^2)}{T_2/T_1} = \\ &= \frac{(F + G^2)}{(T_2/T_1 - F) + F} = \frac{F + G^2}{F + G} = \\ &= \frac{(F + G) - (G - G^2)}{F + G} = 1 - \frac{G(1 - G)}{F + G}. \end{aligned} \tag{4.6}$$

Since $0 \leq G < 1$, the term $G(1 - G)$ is nonnegative. Hence, U is monotonically increasing with F. As a consequence, the minimum of U occurs for the minimum value of F; namely, $F = 1$. Thus,

$$U = \frac{1 + G^2}{1 + G}. \tag{4.7}$$

Minimizing U over G, we have:

$$\begin{aligned} \frac{dU}{dG} &= \frac{2G(1 + G) - (1 + G^2)}{(1 + G)^2} = \\ &= \frac{G^2 + 2G - 1}{(1 + G)^2}, \end{aligned}$$

and $dU/dG = 0$ for $G^2 + 2G - 1 = 0$, which has two solutions:

$$\begin{cases} G_1 = -1 - \sqrt{2} \\ G_2 = -1 + \sqrt{2}. \end{cases}$$

Since $0 \leq G < 1$, the negative solution $G = G_1$ is discarded. Thus, from Eq. (4.7), the least upper bound of U is given for $G = G_2$:

$$U_{lub} = \frac{1 + (\sqrt{2} - 1)^2}{1 + (\sqrt{2} - 1)} = \frac{4 - 2\sqrt{2}}{\sqrt{2}} = 2(\sqrt{2} - 1).$$

That is,

$$U_{lub} = 2(2^{1/2} - 1) \simeq 0.83. \tag{4.8}$$

Notice that if T_2 is a multiple of T_1, $G = 0$ and the processor utilization factor becomes 1. In general, the utilization factor for two tasks can be computed as a function of the ratio $k = T_2/T_1$. For a given F, from Eq. (4.5), we can write:

$$U = \frac{F + (k - F)^2}{k} = k - 2F + \frac{F(F + 1)}{k}.$$

Minimizing U over k, we have:

$$\frac{dU}{dk} = 1 - \frac{F(F + 1)}{k^2},$$

and $dU/dk = 0$ for $k^* = \sqrt{F(F + 1)}$. Hence, for a given F, the minimum value of U is

$$U^* = 2(\sqrt{F(F + 1)} - F).$$

Table 4.1 shows some values of k^* and U^* as a function of F, whereas Fig. 4.10 shows the upper bound of U as a function of k.

Table 4.1 Values of k_i^* and U_i^* as a function of F

F	k^*	U^*
1	$\sqrt{2}$	0.828
2	$\sqrt{6}$	0.899
3	$\sqrt{12}$	0.928
4	$\sqrt{20}$	0.944
5	$\sqrt{30}$	0.954

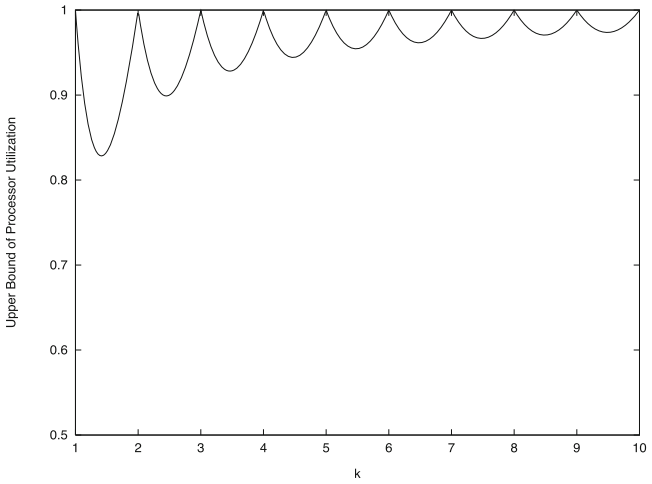


Fig. 4.10 Upper bound of the processor utilization factor as a function of the ratio $k = T_2/T_1$

4.3.3 Calculation of U_{lub} for N Tasks

From the previous computation, the conditions that allow to compute the least upper bound of the processor utilization factor are

$$\begin{cases} F = 1 \\ C_1 = T_2 - FT_1 \\ C_2 = (T_1 - C_1)F, \end{cases}$$

which can be rewritten as

$$\begin{cases} T_1 < T_2 < 2T_1 \\ C_1 = T_2 - T_1 \\ C_2 = 2T_1 - T_2. \end{cases}$$

Generalizing for an arbitrary set of n tasks, the worst conditions for the schedulability of a task set that fully utilizes the processor are

$$\begin{cases} T_1 < T_n < 2T_1 \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_1 - (C_1 + C_2 + \dots + C_{n-1}) = 2T_1 - T_n. \end{cases}$$

Thus, the processor utilization factor becomes:

$$U = \frac{T_2 - T_1}{T_1} + \frac{T_3 - T_2}{T_2} + \ldots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_1 - T_n}{T_n}.$$

Defining

$$R_i = \frac{T_{i+1}}{T_i}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = \sum_{i=1}^{n-1} R_i + \frac{2}{R_1 R_2 \ldots R_{n-1}} - n.$$

To minimize U over R_i, $i = 1, \ldots, n-1$, we have:

$$\frac{\partial U}{\partial R_k} = 1 - \frac{2}{R_i^2 (\prod_{i \neq k}^{n-1} R_i)}.$$

Thus, defining $P = R_1 R_2 \ldots R_{n-1}$, U is minimum when

$$\begin{cases} R_1 P = 2 \\ R_2 P = 2 \\ \ldots \\ R_{n-1} P = 2. \end{cases}$$

That is, when all R_i have the same value:

$$R_1 = R_2 = \ldots = R_{n-1} = 2^{1/n}.$$

Substituting this value in U, we obtain:

$$\begin{aligned} U_{lub} &= (n-1)2^{1/n} + \frac{2}{2^{(1-1/n)}} - n = \\ &= n2^{1/n} - 2^{1/n} + 2^{1/n} - n = \\ &= n(2^{1/n} - 1). \end{aligned}$$

Table 4.2 Values of U_{lub} as a function of n

n	U_{lub}
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

Therefore, for an arbitrary set of periodic tasks, the least upper bound of the processor utilization factor under the Rate-Monotonic scheduling algorithm is

$$U_{lub} = n(2^{1/n} - 1). \tag{4.9}$$

This bound decreases with n, and values for some n are shown in Table 4.2.

For high values of n, the least upper bound converges to

$$U_{lub} = \ln 2 \simeq 0.69.$$

In fact, with the substitution $y = (2^{1/n} - 1)$, we obtain: $n = \frac{\ln 2}{\ln(y+1)}$, and hence

$$\lim_{n\to\infty} n(2^{1/n} - 1) = (\ln 2) \lim_{y\to 0} \frac{y}{\ln(y+1)}$$

and since (by Hospital's rule)

$$\lim_{y\to 0} \frac{y}{\ln(y+1)} = \lim_{y\to 0} \frac{1}{1/(y+1)} = \lim_{y\to 0} (y+1) = 1,$$

we have that

$$\lim_{n\to\infty} U_{lub}(n) = \ln 2.$$

4.3.4 Hyperbolic Bound for RM

The feasibility analysis of the RM algorithm can also be performed using a different approach, called the hyperbolic bound [BBB01, BBB03]. The test has the same complexity as the original Liu and Layland bound, but it is less pessimistic, so allowing to accept task sets that would be rejected using the original approach. Instead of minimizing the processor utilization with respect to task periods, the feasibility condition can be manipulated in order to find a tighter sufficient schedulability test as a function of the individual task utilizations.

The following theorem provides a sufficient condition for testing the schedulability of a task set under the RM algorithm.

Theorem 4.1 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i. Then, Γ is schedulable with the RM algorithm if*

$$\prod_{i=1}^{n}(U_i + 1) \leq 2. \tag{4.10}$$

Proof Without loss of generality, we may assume that tasks are ordered by increasing periods, so that τ_1 is the task with the highest priority and τ_n is the task with the lowest priority. In [LL73], as well as in [DG00], it has been shown that the worst-case scenario for a set on n periodic tasks occurs when all the tasks start simultaneously (e.g., at time $t = 0$) and periods are such that

$$\forall i = 2, \ldots, n \quad T_1 < T_i < 2\,T_1.$$

Moreover, the total utilization factor is minimized when computation times have the following relations:

$$\begin{cases} C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \ldots \\ C_{n-1} = T_n - T_{n-1} \end{cases} \tag{4.11}$$

and the schedulability condition is given by

$$\sum_{i=1}^{n} C_i \leq T_1. \tag{4.12}$$

From Eqs. (4.11), the schedulability condition can also be written as

$$C_n \leq 2\,T_1 - T_n \tag{4.13}$$

Now, defining

$$R_i = \frac{T_{i+1}}{T_i} \quad \text{and} \quad U_i = \frac{C_i}{T_i}.$$

Eqs. (4.11) can be written as follows:

$$\begin{cases} U_1 = R_1 - 1 \\ U_2 = R_2 - 1 \\ \ldots \\ U_{n-1} = R_{n-1} - 1. \end{cases} \tag{4.14}$$

Now we notice that

$$\prod_{i=1}^{n-1} R_i = \frac{T_2}{T_1}\frac{T_3}{T_2}\cdots\frac{T_n}{T_{n-1}} = \frac{T_n}{T_1}.$$

If we divide both sides of the feasibility condition (4.13) by T_n, we get:

$$U_n \leq \frac{2T_1}{T_n} - 1.$$

Hence, the feasibility condition for a task set which fully utilizes the processor can be written as

$$U_n + 1 \leq \frac{2}{\prod_{i=1}^{n-1} R_i}.$$

Since $R_i = U_i + 1$ for all $i = 1, \ldots, n-1$, we have:

$$(U_n + 1)\prod_{i=1}^{n-1}(U_i + 1) \leq 2$$

and finally

$$\prod_{i=1}^{n}(U_i + 1) \leq 2,$$

which proves the theorem. □

The new test can be compared with the Liu and Layland one in the task utilization space, denoted as the U-space. Here, the Liu and Layland bound for RM is represented by a n-dimensional plane which intersects each axis in $U_{\text{lub}}(n) = n\,(2^{1/n} - 1)$. All points below the RM surface represent periodic task sets that are feasible by RM. The new bound expressed by Eq. (4.10) is represented by a n-dimensional hyperbolic surface tangent to the RM plane and intersecting the axes for $U_i = 1$ (this is the reason why it is referred to as the hyperbolic bound). Figure 4.11 illustrates such bounds for $n = 2$. Notice that the asymptotes of the hyperbole are at $U_i = -1$. From the plots, we can clearly see that the feasibility region below the H-bound is larger than that below the LL-bound, and the gain is given by the dark gray area.

It has been shown [BBB03] that the hyperbolic bound is tight, meaning that it is the best possible bound that can be found using the individual task utilization factors U_i as a task set knowledge.

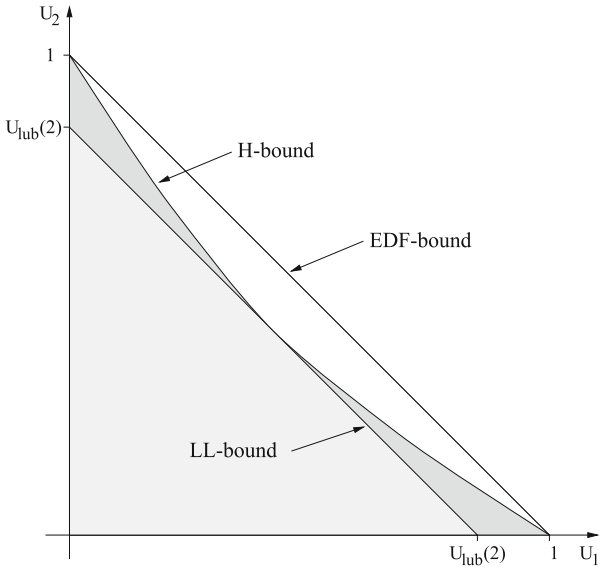


Fig. 4.11 Schedulability bounds for RM and EDF in the utilization space

Moreover, the gain (in terms of schedulability) achieved by the hyperbolic test over the classical Liu and Layland test increases as a function of the number of tasks, and tends to $\sqrt{2}$ for n tending to infinity.

4.4 Earliest Deadline First

The Earliest Deadline First (EDF) algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. Specifically, tasks with earlier deadlines will be executed at higher priorities. Since the absolute deadline of a periodic task depends on the current jth instance as

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i,$$

EDF is a dynamic priority assignment. Moreover, it is intrinsically preemptive: the currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

Notice that EDF does not make any specific assumption on the periodicity of the tasks; hence, it can be used for scheduling periodic as well as aperiodic tasks. For the same reason, the optimality of EDF, proved in Chap. 3 for aperiodic tasks, also holds for periodic tasks.

4.4.1 Schedulability Analysis

Under the assumptions A1, A2, A3, and A4, the schedulability of a periodic task set handled by EDF can be verified through the processor utilization factor. In this case, however, the least upper bound is one; therefore, tasks may utilize the processor up to 100% and still be schedulable. In particular, the following theorem holds [LL73, SBS95].

Theorem 4.2 *A set of periodic tasks is schedulable with EDF if and only if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

Proof *Only if*. We show that a task set cannot be scheduled if $U > 1$. In fact, by defining $T = T_1 T_2 \ldots T_n$, the total demand of computation time requested by all tasks in T can be calculated as

$$\sum_{i=1}^{n} \frac{T}{T_i} C_i = UT.$$

If $U > 1$—that is, if the total demand UT exceeds the available processor time T—there is clearly no feasible schedule for the task set.

If. We show the sufficiency by contradiction. Assume that the condition $U < 1$ is satisfied and yet the task set is not schedulable. Let t_2 be the instant at which the time-overflow occurs and let $[t_1, t_2]$ be the longest interval of continuous utilization, before the overflow, such that only instances with deadline less than or equal to t_2 are executed in $[t_1, t_2]$ (see Fig. 4.12 for explanation). Note that t_1 must be the release time of some periodic instance. Let $C_p(t_1, t_2)$ be the total computation time

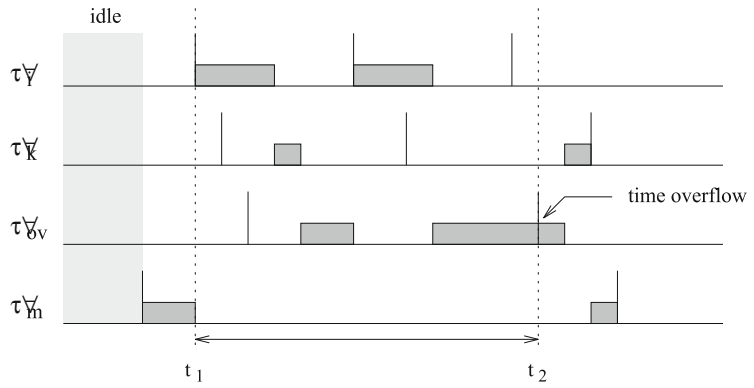


Fig. 4.12 Interval of continuous utilization in an EDF schedule before a time-overflow

demanded by periodic tasks in $[t_1, t_2]$, which can be computed as

$$C_p(t_1, t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i. \tag{4.15}$$

Now, observe that

$$C_p(t_1, t_2) = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U.$$

Since a deadline is missed at t_2, $C_p(t_1, t_2)$ must be greater than the available processor time $(t_2 - t_1)$; thus, we must have:

$$(t_2 - t_1) < C_p(t_1, t_2) \leq (t_2 - t_1)U.$$

That is, $U > 1$, which is a contradiction. □

4.4.2 An Example

Consider the periodic task set illustrated in Fig. 4.13, for which the processor utilization factor is

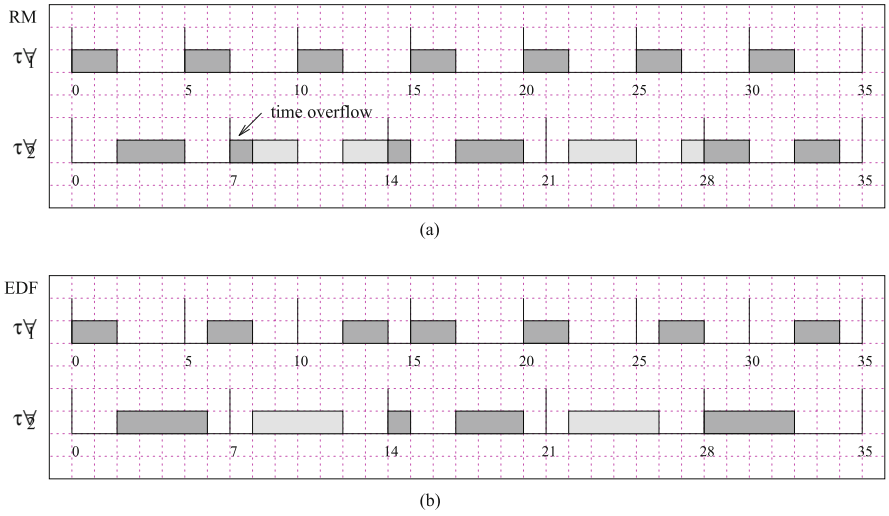


Fig. 4.13 Schedule produced by RM (**a**) and EDF (**b**) on the same set of periodic tasks

$$U = \frac{2}{5} + \frac{4}{7} = \frac{34}{35} \simeq 0.97.$$

This means that 97% of the processor time is used to execute the periodic tasks, whereas the CPU is idle in the remaining 3%. Being $U > \ln 2$, the schedulability of the task set cannot be guaranteed under RM, whereas it is guaranteed under EDF. Indeed, as shown in Fig. 4.13a, RM generates a time-overflow at time $t = 7$, whereas EDF completes all tasks within their deadlines (see Fig. 4.13b).

Another important difference between RM and EDF concerns the number of preemptions occurring in the schedule. As shown in Fig. 4.13, under RM every instance of task τ_2 is preempted, for a total number of five preemptions in the interval $T = T_1 T_2$. Under EDF, the same task is preempted only once in T. The small number of preemptions in EDF is a direct consequence of the dynamic priority assignment, which at any instant privileges the task with the earliest deadline, independently of tasks' periods.

4.5 Deadline Monotonic

The algorithms and the schedulability bounds illustrated in the previous sections rely on the assumptions A1, A2, A3, and A4 presented at the beginning of this chapter. In particular, assumption A3 imposes a relative deadline equal to the period, allowing an instance to be executed anywhere within its period. This condition could not always be desired in real-time applications. For example, relaxing assumption A3 would provide a more flexible process model, which could be adopted to handle tasks with jitter constraints or activities with short response times compared to their periods.

The Deadline Monotonic (DM) priority assignment weakens the "period equals deadline" constraint within a static priority scheduling scheme. This algorithm was first proposed in 1982 by Leung and Whitehead [LW82] as an extension of Rate Monotonic, where tasks can have relative deadlines less than or equal to their period (i.e., *constrained deadlines*). Specifically, each periodic task τ_i is characterized by four parameters:

- A phase Φ_i;
- A worst-case computation time C_i (constant for each instance);
- A relative deadline D_i (constant for each instance);
- A period T_i.

These parameters are illustrated in Fig. 4.14 and have the following relationships:

$$\begin{cases} C_i \leq D_i \leq T_i \\ r_{i,k} = \Phi_i + (k-1)T_i \\ d_{i,k} = r_{i,k} + D_i. \end{cases}$$

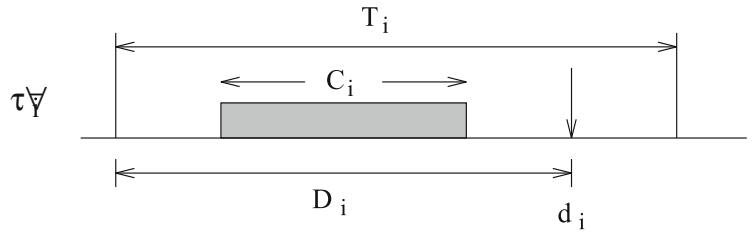


Fig. 4.14 Task parameters in Deadline-Monotonic scheduling

According to the DM algorithm, each task is assigned a fixed- priority P_i inversely proportional to its relative deadline D_i. Thus, at any instant, the task with the shortest relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As RM, DM is normally used in a fully preemptive mode; that is, the currently executing task is preempted by a newly arrived task with shorter relative deadline.

The Deadline-Monotonic priority assignment is optimal,[2] meaning that if a task set is schedulable by a some fixed-priority assignment, then it is also schedulable by DM.

4.5.1 Schedulability Analysis

The feasibility of a task set with constrained deadlines could be guaranteed using the utilization based test, by reducing tasks' periods to relative deadlines:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1).$$

However, such a test would not quite pessimistic, since the workload on the processor would be overestimated. A less pessimistic schedulability test can be derived by noting that:

- The worst-case processor demand occurs when all tasks are released simultaneously, that is, at their critical instants;
- For each task τ_i, the sum of its processing time and the interference (preemption) imposed by higher-priority tasks must be less than or equal to D_i.

Assuming that tasks are ordered by increasing relative deadlines, so that

$$i < j \iff D_i < D_j,$$

[2] The proof of DM optimality is similar to the one done for RM and it can be found in [LW82].

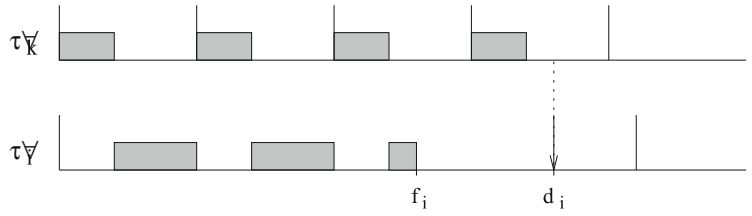


Fig. 4.15 More accurate calculation of the interference on τ_i by higher-priority tasks

such a test is given by

$$\forall i : 1 \leq i \leq n \quad C_i + I_i \leq D_i, \tag{4.16}$$

where I_i is a measure of the interference on τ_i, which can be computed as the sum of the processing times of all higher-priority tasks released before D_i:

$$I_i = \sum_{h=1}^{i-1} \left\lceil \frac{D_i}{T_h} \right\rceil C_h.$$

Notice that this test is sufficient but not necessary for guaranteeing the schedulability of the task set. This is due to the fact that I_i is calculated by assuming that each higher-priority task τ_h exactly interferes $\lceil \frac{D_i}{T_h} \rceil$ times on τ_i. However, as shown in Fig. 4.15, the actual interference can be smaller than I_i, since τ_i may terminate earlier.

To find a sufficient and necessary schedulability test for DM, the exact worst-case interference of higher-priority tasks must be evaluated for each task τ_i. Audsley et al. [ABRW92, ABR+93] proposed an efficient method for evaluating the exact interference on periodic tasks and derived a sufficient and necessary schedulability test for DM, called response time analysis (RTA). Note that similar methods have been independently proposed earlier in the literature, first by Jospeh and Pandia [JP86], and later by Lehoczky [Leh90]. In the following, the method proposed by Audsley et al. [ABR+93] is presented because it is formulated in a way that it is easier to understand.

4.5.2 Response Time Analysis

According to the method proposed by Audsley at al., the longest response time R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its computation time and the interference I_i of the higher-priority tasks:

$$R_i = C_i + I_i,$$

Table 4.3 A set of periodic tasks with deadlines less than periods

	C_i	T_i	D_i
τ_1	1	4	3
τ_2	1	5	4
τ_3	2	6	5
τ_4	1	11	10

where

$$I_i = \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h.$$

Hence,

$$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h. \tag{4.17}$$

Since R_i appears on both sides of Eq. (4.17) and cannot be grouped due to the ceiling operator, the solution requires an iterative approach, where at step $s = 0$ the response time of task τ_i is guessed to be equal to its computation time ($R_i^{(0)} = C_i$). At the generic step s, a new guess is computed by considering the interference in the interval $R_i^{(s-1)}$ given by the previous guess. If the response time at step s results to be equal to the one guessed at the previous step ($R_i^{(s)} = R_i^{(s-1)}$), then the guess was correct and we can stop the iteration; otherwise, it means that the interference is larger than what we guesses and we have to make another iterative step.

Thus, the worst-case response time of task τ_i is given by the smallest value of R_i that satisfies Eq. (4.17) and can be computed by the following iterative approach:

$$R_i^{(0)} = C_i \tag{4.18}$$

$$R_i^{(s)} = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h \tag{4.19}$$

where the iteration continues as long as $R_i^{(s)} > R_i^{(s-1)}$, and stops if $R_i^{(s)} = R_i^{(s-1)}$, in which case $R_i = R_i^{(s)}$.

Notice that the iteration can also be stopped if $R_i^{(s)} > D_i$, meaning that the task is not schedulable.

To clarify the schedulability test, consider the set of periodic tasks shown in Table 4.3, simultaneously activated at time $t = 0$. In order to guarantee τ_4, we have to calculate R_4 and verify that $R_4 \leq D_4$. The schedule produced by DM is illustrated in Fig. 4.16, and the iteration steps are shown below.

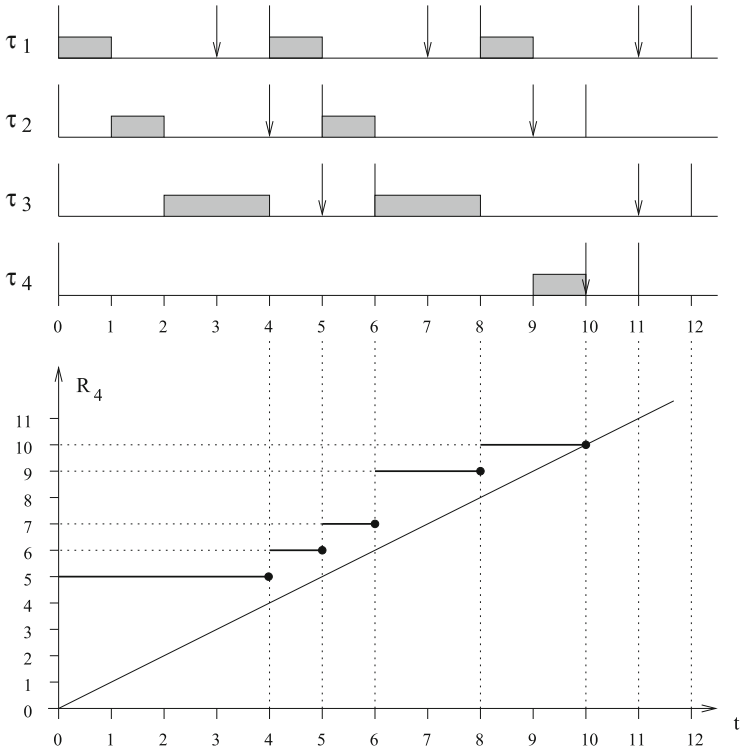


Fig. 4.16 Example of schedule produced by DM and response time experienced by τ_4 as a function of the considered interference interval

$$
\begin{aligned}
&\text{Step 0:} \quad R_4^{(0)} = C_4 = 1 \\
&\text{Step 1:} \quad R_4^{(1)} = 1 + \left\lceil \tfrac{1}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{1}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{1}{6} \right\rceil \cdot 2 = 5 \\
&\text{Step 2:} \quad R_4^{(2)} = 1 + \left\lceil \tfrac{5}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{5}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{5}{6} \right\rceil \cdot 2 = 6 \\
&\text{Step 3:} \quad R_4^{(3)} = 1 + \left\lceil \tfrac{6}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{6}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{6}{6} \right\rceil \cdot 2 = 7 \\
&\text{Step 4:} \quad R_4^{(4)} = 1 + \left\lceil \tfrac{7}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{7}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{7}{6} \right\rceil \cdot 2 = 9 \\
&\text{Step 5:} \quad R_4^{(5)} = 1 + \left\lceil \tfrac{9}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{9}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{9}{6} \right\rceil \cdot 2 = 10 \\
&\text{Step 6:} \quad R_4^{(6)} = 1 + \left\lceil \tfrac{10}{4} \right\rceil \cdot 1 + \left\lceil \tfrac{10}{5} \right\rceil \cdot 1 + \left\lceil \tfrac{10}{6} \right\rceil \cdot 2 = 10
\end{aligned}
$$

Since $R_4^{(6)} = R_4^{(5)} = 10$, we stop the iteration and conclude that the response time of task τ_4 is $R_4 = 10$. Also, since $R_4 \leq D_4$, τ_4 is schedulable within its deadline. If $R_i \leq D_i$ for all tasks, we conclude that the task set is schedulable by DM. Such a schedulability test can be performed by the algorithm illustrated in Fig. 4.17.

```
RTA_test (Γ)
{
    for (each τ_i ∈ Γ) {
        R_i = C_i;
        do {
            R_i^old = R_i;
            I_i = Σ_{h=1}^{i-1} ⌈R_old / T_h⌉ C_h;
            R_i = C_i + I_i;
            if (R_i > D_i) return(UNSCHEDULABLE);
        } while (R_i > R_i^old);
    }
    return(SCHEDULABLE);
}
```

Fig. 4.17 Algorithm for testing the schedulability of a periodic task set Γ with the response time analysis

Note that the algorithm in Fig. 4.17 has a pseudo-polynomial complexity. In fact, the guarantee of the entire task set requires $O(nN)$ steps, where n is the number of tasks and N is the number of iterations in the inner loop, which does not depend directly on n but on the period relations. In most cases, the RTA test converges very quickly and can also be used on-line for guaranteeing real-time applications with dynamic periodic task activations.

For those systems in which such a complexity is too high to execute the test for on-line guarantees, a sufficient approximated test can easily be derived by computing a response time upper bound, substituting the ceiling in Eq. (4.17) with the argument plus 1:

$$R_i = C_i + \sum_{h=1}^{i-1} \left(\frac{R_i}{T_h} + 1 \right) C_h$$

$$R_i = C_i + R_i \sum_{h=1}^{i-1} U_h + \sum_{h=1}^{i-1} C_h$$

$$R_i - R_i \sum_{h=1}^{i-1} U_h = \sum_{h=1}^{i} C_h.$$

From which the response time upper bound for task τ_i results to be

$$R_i^{ub} = \frac{\sum_{h=1}^{i} C_h}{1 - \sum_{h=1}^{i-1} U_h}. \tag{4.20}$$

Using the response time upper bound reported in Eq. (4.20), a set of real-time periodic tasks with relative deadlines less than or equal to periods is schedulable by Deadline Monotonic if

$$\forall i, \quad R_i^{ub} \leq D_i. \tag{4.21}$$

4.5.3 Improving RTA

The response time algorithm can be improved in several ways. The idea is to improve the initial guess $R_i^{(0)}$ used to estimate the response time of task τ_i. In fact, a higher guess can reduce the number of iteration steps required to converge. However, note that, since the response time of task τ_i is given by the smallest value of R_i that satisfies Eq. (4.17), the intial guess cannot be higher than the actual response time.

In the following, a number of methods [SH98] are considered to improve the initial guess for $R_i^{(0)}$.

METHOD 1: Sum of Computation Times of High-Priority Tasks
This method considers that each task τ_i experiences at least one interference from each higher-priority task. Hence, its worst-case response time cannot be smaller than the sum of the worst-case execution times of the higher-priority tasks plus its own computatio time. Therefore, we can safely set:

$$R_i^{(0)} = \sum_{h=1}^{i} C_h. \tag{4.22}$$

METHOD 2: Previous Response Time
This method considers that each task τ_i cannot start executing if all the tasks with higher priorities are not completed. Therefore, if response times are computed by following an increasing priority order, the response time of task τ_i cannot be smaller than the response time of task τ_{i-1} plus its own computation time. Therefore, we can safely set:

$$R_i^{(0)} = R_{i-1} + C_i. \tag{4.23}$$

METHOD 3: Response Time Lower Bound
This method computes a lower bound from Eq. (4.17) by removing the ceiling operator from the right-hand side term:

$$R_i \;=\; C_i + \sum_{h=1}^{i-1} \frac{R_i}{T_h} C_h \tag{4.24}$$

$$R_i - R_i \sum_{h=1}^{i-1} U_h \;=\; C_i. \tag{4.25}$$

From which the response time lower bound for task τ_i results to be

$$R_i^{lb} \;=\; \frac{C_i}{1 - \sum_{h=1}^{i-1} U_h}. \tag{4.26}$$

Therefore, we can safely set the initial guess for task τ_i as

$$R_i^{(0)} = \frac{C_i}{1 - \sum_{h=1}^{i-1} U_h}. \tag{4.27}$$

Comparing the Three Methods
Since the response time of a task τ_i cannot be smaller than the sum of the computation times of the higher-priority tasks plus its own computation time, we can say that

$$R_{i-1} \;\geq\; \sum_{h=1}^{i-1} C_h. \tag{4.28}$$

It follows that

$$R_{i-1} + C_i \;\geq\; \sum_{h=1}^{i} C_h. \tag{4.29}$$

This means that the guess provided by Method 2 is always higher than or equal to the guess given by Method 1.

Concerning the last two methods, no one dominates the other, since there are cases in which the guess provided by Method 2 is higher than that provided by Method 3, and vice versa. Hence, the best initial guess is to compute both and take the maximum:

$$R_i^{(0)} = \max \left\{ R_{i-1} + C_i,\; \frac{C_i}{1 - \sum_{h=1}^{i-1} U_h} \right\}. \tag{4.30}$$

4.5.4 Workload Analysis

Another necessary and sufficient test for checking the schedulability of fixed-priority systems with constrained deadlines was proposed by Lehoczky, Sha, and

Ding [LSD89]. The test is based on the concept of Level-i workload, defined as follows.

Definition 4.1 The Level-i workload $W_i(t)$ is the cumulative computation time requested in the interval $(0, t]$ by task τ_i and all the tasks with priority higher than P_i.

For a set of synchronous periodic tasks, the Level-i workload can be computed as follows:

$$W_i(t) = C_i + \sum_{h:P_h > P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h. \tag{4.31}$$

Then, the test can be expressed by the following theorem.

Theorem 4.3 (Lehoczky-Sha-Ding, 1989) *A set of fully preemptive periodic tasks can be scheduled by a fixed-priority algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in (0, D_i] : W_i(t) \leq t. \tag{4.32}$$

To verify the schedulability of task τ_i, condition $W_i(t) \leq t$ should be checked for $t = D_i$ and for all release times $r_{hk} < D_i$ of jobs with priority higher than or equal to P_i. More formally, the condition should be checked for all $t \in \mathcal{R}_i$, where

$$\mathcal{R}_i = \left\{ r_{hk} \mid r_{hk} = kT_h, h = 1, \ldots, i, k = 1, \ldots, \left\lfloor \frac{T_i}{T_h} \right\rfloor \right\} \cup \{D_i\}. \tag{4.33}$$

Figure 4.18 illustrates an example of workload function for a set of two periodic tasks that are feasible under DM. As clear from the graph, in the interval $[0, D_2]$, there exists a time $t < D_2$ (e.g., $t = 8$) at which $W_2(t) \leq t$, proving that τ_2 finishes before its deadline.

Later, Bini and Buttazzo [BB04] restricted the number of points in which condition (4.32) has to be checked to the following *Testing Set*:

$$\mathcal{TS}_i \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i) \tag{4.34}$$

where $\mathcal{P}_i(t)$ is defined by the following recurrent expression:

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t). \end{cases} \tag{4.35}$$

Thus, the schedulability test can be expressed by the following theorem.

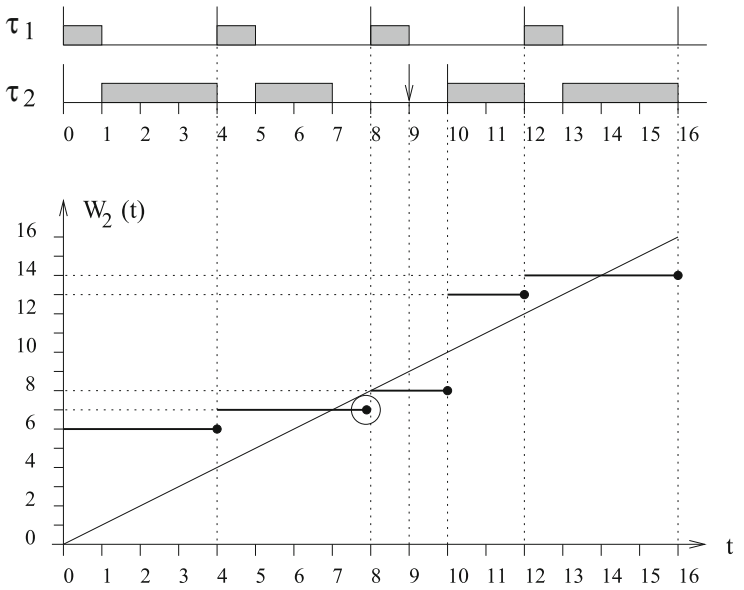


Fig. 4.18 Example of workload function for a task τ_2 that is feasible under DM. As clear from the graph, in the interval $[0, D_2]$, there exists a time ($t = 8$) in which $W_2(t) \leq t$

Theorem 4.4 (Bini and Buttazzo, 2004) *A set of fully preemptive periodic tasks can be scheduled by a fixed-priority algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in \mathcal{TS}_i : W_i(t) \leq t. \tag{4.36}$$

An advantage of Eq. (4.36) is that it can be formulated as the union of a set of simpler conditions, leading to a more efficient guarantee test, named the Hyperplane test [BB04]. The test has still a pseudo-polynomial complexity, but runs much quicker than the response time analysis in the average case. Moreover, a novel feature of this test is that it can be tuned using a parameter to balance acceptance ratio versus complexity. Such a tunability property is important in those cases in which the performance of a polynomial time test is not sufficient for achieving high processor utilization, and the overhead introduced by exact tests is too high for an on-line admission control.

Another advantage of this formulation is that Eq. (4.36) can be manipulated to describe the feasibility region of the task set in a desired space of design parameters, so enabling sensitivity analysis [BDNB08], which allows determining how to change task set parameters when the schedule is infeasible.

4.6 EDF with Deadlines Less Than Periods

Under EDF, the analysis of periodic tasks with deadlines less than periods can be performed using a *processor demand* criterion. This method has been described by Baruah, Rosier, and Howell in [BRH90] and later used by Jeffay and Stone [JS93] to account for interrupt handling costs under EDF. Here, we first illustrate this approach for the case of deadlines equal to periods and then extend it to more general task models.

4.6.1 The Processor Demand Approach

In general, the processor demand of a task τ_i in an interval $[t_1, t_2]$ is the amount of processing time $g_i(t_1, t_2)$ requested by those instances of τ_i activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. That is,

$$g_i(t_1, t_2) = \sum_{r_{i,k} \geq t_1, d_{i,k} \leq t_2} C_i.$$

For the whole task set, the processor demand in $[t_1, t_2]$ is given by

$$g(t_1, t_2) = \sum_{i=1}^{n} g_i(t_1, t_2).$$

Then, the feasibility of a task set is guaranteed if and only if *in any interval of time* the processor demand does not exceed the available time, that is, if and only if

$$\forall t_1, t_2 \quad g(t_1, t_2) \leq (t_2 - t_1).$$

Referring to Fig. 4.19, the number of instances of task τ_i that contribute to the demand in $[t_1, t_2]$ can be expressed as

$$\eta_i(t_1, t_2) = \max \left\{ 0, \left\lfloor \frac{t_2 + T_i - D_i - \Phi_i}{T_i} \right\rfloor - \left\lceil \frac{t_1 - \Phi_i}{T_i} \right\rceil \right\}$$

and the processor demand in $[t_1, t_2]$ can be computed as

$$g(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) C_i. \tag{4.37}$$

If relative deadlines are no larger than periods and periodic tasks are simultaneously activated at time $t = 0$ (i.e., $\Phi_i = 0$ for all the tasks), then the number of

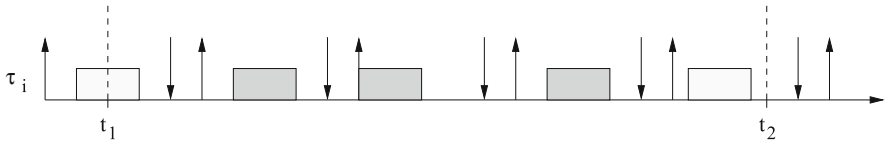


Fig. 4.19 The instances in dark gray are those contributing to the processor demand in $[t_1, t_2]$

instances contributing to the demand in an interval $[0, L]$ can be expressed as

$$\eta_i(0, L) = \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor.$$

Thus, the processor demand in $[0, L]$ can be computed as

$$g(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i.$$

Function $g(0, L)$ is also referred to as Demand-Bound Function:

$$\mathsf{dbf}(t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \tag{4.38}$$

Therefore, a synchronous set of periodic tasks with relative deadlines less than or equal to periods is schedulable by EDF if and only if

$$\forall t > 0 \quad \mathsf{dbf}(t) \leq t. \tag{4.39}$$

It is worth observing that, for the special case of tasks with relative deadlines equal to periods, the test based on the processor demand criterion is equivalent to the one based on the processor utilization. This result is formally expressed by the following theorem [JS93].

Theorem 4.5 (Jeffay and Stone, 1993) *A set of periodic tasks with relative deadlines equal to periods is schedulable by EDF if and only if*

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i \leq L. \tag{4.40}$$

Proof The theorem is proved by showing that Eq. (4.40) is equivalent to the classical Liu and Layland's condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{4.41}$$

(4.41) $\Rightarrow$ (4.40). If $U \leq 1$, then for all L, $L \geq 0$,

$$L \geq UL = \sum_{i=1}^{n} \left(\frac{L}{T_i} \right) C_i \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i.$$

To demonstrate (4.41) $\Leftarrow$ (4.40), we show that $\neg$(4.41) $\Rightarrow$ $\neg$(4.40). That is, we assume $U > 1$ and prove that there exist an $L \geq 0$ for which (4.40) does not hold. If $U > 1$, then for $L = H = lcm(T_1, \ldots, T_n)$:

$$H < HU = \sum_{i=1}^{n} \left(\frac{H}{T_i} \right) C_i = \sum_{i=1}^{n} \left\lfloor \frac{H}{T_i} \right\rfloor C_i.$$

□

4.6.2 Reducing Test Intervals

In this section we show that the feasibility test expressed by condition (4.39) can be simplified by reducing the number of intervals in which it has to be verified. We first observe that:

1. If tasks are periodic and are simultaneously activated at time $t = 0$, then the schedule repeats itself every hyperperiod H; thus, condition (4.39) needs to be verified only for values of L less than or equal to H.
2. $g(0, L)$ is a step function whose value increases when L crosses a deadline d_k and remains constant until the next deadline d_{k+1}. This means that if condition $g(0, L) < L$ holds for $L = d_k$, then it also holds for all L such that $d_k \leq L < d_{k+1}$. As a consequence, condition (4.39) needs to be verified only for values of L equal to absolute deadlines.

   The number of testing points can be reduced further by noting that

$$\left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor \leq \left(\frac{L + T_i - D_i}{T_i} \right).$$

And defining

$$G(0, L) = \sum_{i=1}^{n} \frac{L + T_i - D_i}{T_i} C_i = \sum_{i=1}^{n} \frac{T_i - D_i}{T_i} C_i + \frac{L}{T_i} C_i$$

we have that

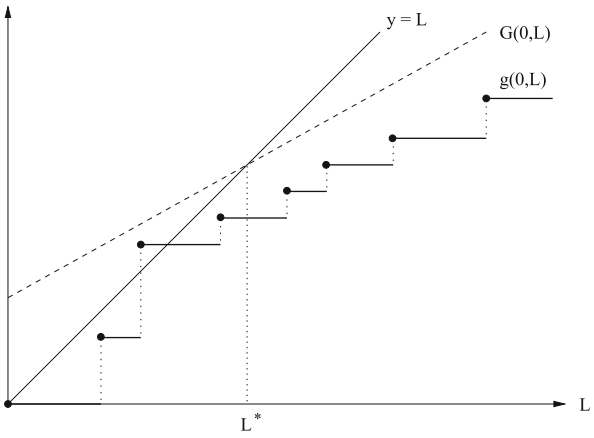


Fig. 4.20 Maximum value of L for which the processor demand test as to be verified

$$\forall L > 0, \quad g(0, L) \leq G(0, L),$$

where

$$G(0, L) = \sum_{i=1}^{n} (T_i - D_i)U_i + LU.$$

From Fig. 4.20, we can note that $G(0, L)$ is a function of L increasing as a straight line with slope U. Hence, if $U < 1$, there exists an $L = L^*$ for which $G(0, L) = L$. Clearly, for all $L \geq L^*$, we have that $g(0, L) \leq G(0, L) \leq L$, meaning that the schedulability of the task set is guaranteed. As a consequence, there is no need to verify condition (4.39) for values of $L \geq L^*$.

The value of L^* is the time at which $G(0, L^*) = L^*$, that is,

$$\sum_{i=1}^{n} (T_i - D_i)U_i + L^*U = L^*$$

which gives

$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i)U_i}{1 - U}.$$

Considering that the task set must be tested at least until the largest relative deadline D_{max}, the results of the previous observations can be combined in the following theorem.

Theorem 4.6 *A set of synchronous periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF if and only if* $U < 1$ *and*

Table 4.4 A task set with relative deadlines less than periods

	C_i	D_i	T_i
τ_1	2	4	6
τ_2	2	5	8
τ_3	3	7	9

Table 4.5 Testing intervals for the processor demand criterion

L	$g(0, L)$	Result
4	2	OK
5	4	OK
7	7	OK
10	9	OK
13	11	OK
16	16	OK
21	18	OK
22	20	OK

$$\forall t \in \mathcal{D} \quad \mathsf{dbf}(t) \leq t. \tag{4.42}$$

where

$$\mathcal{D} = \{d_k \mid d_k \leq \min[H, \max(D_{max}, L^*)]\}$$

and

$$L^* = \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U}.$$

4.6.2.1 Example

To illustrate the processor demand criterion, consider the task set shown in Table 4.4, where three periodic tasks with deadlines less than periods need to be guaranteed under EDF. From the specified parameters, it is easy to compute that

$$U = \frac{2}{6} + \frac{2}{8} + \frac{3}{9} = \frac{11}{12}$$
$$L^* = \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U} = 25$$
$$H = \text{lcm}(6, 8, 9) = 72.$$

Hence, condition (4.42) has to be tested for any deadline less than 25, and the set of checking points is given by $\mathcal{D} = \{4, 5, 7, 10, 13, 16, 21, 22\}$. Table 4.5 shows the results of the test, and Fig. 4.21 illustrates the schedule produced by EDF for the task set.

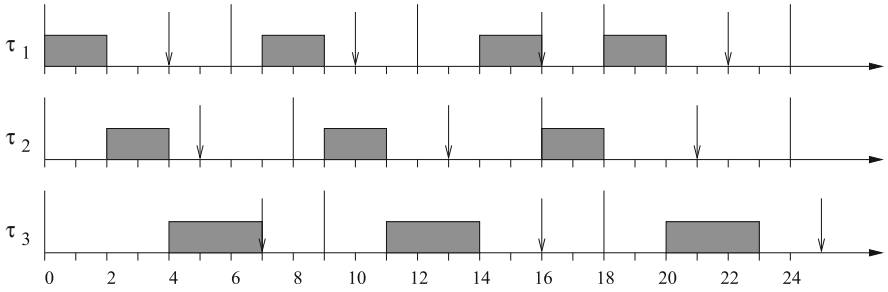


Fig. 4.21 Schedule produced by EDF for the task set shown in Table 4.4

4.7 Comparison Between RM and EDF

In conclusion, the problem of scheduling a set of independent and preemptable periodic tasks has been solved both under fixed- and dynamic-priority assignments.

The major advantage of the fixed-priority approach is that it is simpler to implement. In fact, if the ready queue is implemented as a multilevel queue with P priority levels (where P is the number of different priorities in the system), both task insertion and extraction can be achieved in $O(1)$. On the other hand, in a deadline-driven scheduler, the best solution for the ready queue is to implement it as a heap (i.e., a balanced binary tree), where task management requires an $O(logn)$ complexity.

Except for such an implementation issue, which becomes relevant only for very large task sets (consisting of hundreds of tasks), or for very slow processors, a dynamic-priority scheme has many advantages with respect to a fixed-priority algorithm. A detailed comparison between RM and EDF has been presented in [But03].

In terms of schedulability analysis, an exact guarantee test for RM requires a pseudo-polynomial complexity, even in the simple case of independent tasks with relative deadlines equal to periods, whereas it can be performed in $O(n)$ for EDF. In the general case in which deadlines can be less than or equal to periods, the schedulability analysis becomes pseudo-polynomial for both algorithms. Under fixed-priority assignments, the feasibility of the task set can be tested using the response time analysis, whereas under dynamic priority assignments, it can be tested using the processor demand criterion.

As for the processor utilization, EDF is able to exploit the full processor bandwidth, whereas the RM algorithm can only guarantee feasibility for task sets with utilization less than 69%, in the worst case. In the average case, a statistical study performed by Lehoczky, Sha, and Ding [LSD89] showed that for task sets with randomly generated parameters, the RM algorithm is able to feasibly schedule task sets with a processor utilization up to about 88%. However, this is only a statistical result and cannot be taken as an absolute bound for performing a precise guarantee test.

In spite of the extra computation needed by EDF for updating the absolute deadline at each job activation, EDF introduces less runtime overhead than RM, when context switches are taken into account. In fact, to enforce the fixed-priority order, the number of preemptions that typically occur under RM is much higher than under EDF.

An interesting property of EDF during permanent overloads is that it automatically performs a period rescaling, so tasks start behaving as they were executing at a lower rate. This property has been proved by Cervin in his PhD dissertation [Cer03], and it is formally stated in the following theorem.

Theorem 4.7 (Cervin) *Assume a set of n periodic tasks, where each task is described by a fixed period T_i, a fixed execution time C_i, a relative deadline D_i, and a release offset Φ_i. If $U > 1$ and tasks are scheduled by EDF, then, in stationarity, the average period $\overline{T_i}$ of each task τ_i is given by $\overline{T_i} = T_i U$.*

Notice that under fixed-priority scheduling, a permanent overload condition causes a complete blocking of the lower-priority tasks.

As it will be discussed later in the book, another major advantage of dynamic scheduling with respect to fixed-priority scheduling is a better responsiveness in handling aperiodic tasks. This property comes from the higher processor utilization bound of EDF. In fact, the lower schedulability bound of RM limits the maximum utilization ($U_s = C_s/T_s$) that can be assigned to a server for guaranteeing the feasibility of the periodic task set. As a consequence, the spare processor utilization that cannot be assigned to the server is wasted as a background execution. This problem does not occur under EDF, where, if U_p is the processor utilization of the periodic tasks, the full remaining fraction $1 - U_p$ can always be allocated to the server for aperiodic execution.

Exercises

4.1 Verify the schedulability and construct the schedule according to the RM algorithm for the following set of periodic tasks.

	C_i	T_i
τ_1	2	6
τ_2	2	8
τ_3	2	12

4.2 Verify the schedulability and construct the schedule according to the RM algorithm for the following set of periodic tasks.

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	1	10

4.3 Verify the schedulability and construct the schedule according to the RM algorithm for the following set of periodic tasks.

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

4.4 Verify the schedulability under RM of the following task set.

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

4.5 Verify the schedulability under EDF of the task set shown in Exercise 4.4.4, and then construct the corresponding schedule.

4.6 Verify the schedulability under EDF and construct the schedule of the following task set.

	C_i	D_i	T_i
τ_1	2	5	6
τ_2	2	4	8
τ_3	4	8	12

4.7 Verify the schedulability of the task set described in Exercise 4.4.6 using the Deadline-Monotonic algorithm. Then construct the schedule.

Chapter 5
Fixed-Priority Servers

5.1 Introduction

The scheduling algorithms treated in the previous chapters deal with homogeneous sets of tasks, where all computational activities are either aperiodic or periodic. Many real-time control applications, however, require both types of processes, which may also differ for their criticality. Typically, periodic tasks are time-driven and execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates. Aperiodic tasks are usually event-driven and may have hard, soft, or non-real-time requirements depending on the specific application.

When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities. Off-line guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment, that is, by assuming a maximum arrival rate for each critical event. This implies that aperiodic tasks associated with critical events are characterized by a minimum interarrival time between consecutive instances, which bounds the aperiodic load. Aperiodic tasks characterized by a minimum interarrival time are called *sporadic*. They are guaranteed under peak-load situations by assuming their maximum arrival rate.

If the maximum arrival rate of some event cannot be bounded a priori, the associated aperiodic task cannot be guaranteed off-line, although an online guarantee of individual aperiodic requests can still be done. Aperiodic tasks requiring online guarantee on individual instances are called *firm*. Whenever a firm aperiodic request enters the system, an acceptance test can be executed by the kernel to verify whether the request can be served within its deadline. If such a guarantee cannot be done, the request is rejected.

In the next sections, we present a number of scheduling algorithms for handling hybrid task sets consisting of a subset of hard periodic tasks and a subset of soft

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_5

aperiodic tasks. All algorithms presented in this chapter rely on the following assumptions:

- Periodic tasks are scheduled based on a fixed-priority assignment, namely, the Rate-Monotonic (RM) algorithm;
- All periodic tasks start simultaneously at time $t = 0$ and their relative deadlines are equal to their periods.
- Arrival times of aperiodic requests are unknown.
- When not explicitly specified, the minimum interarrival time of a sporadic task is assumed to be equal to its deadline.
- All tasks are fully preemptable.

Aperiodic scheduling under dynamic priority assignment is discussed in the next chapter.

5.2 Background Scheduling

The simplest method to handle a set of soft aperiodic activities in the presence of periodic tasks is to schedule them in background, that is, when there are not periodic instances ready to execute. The major problem with this technique is that, for high periodic loads, the response time of aperiodic requests can be too long for certain applications. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringent timing constraints and the periodic load is not high.

Figure 5.1 illustrates an example in which two periodic tasks are scheduled by RM, while two aperiodic tasks are executed in background. Since the processor utilization factor of the periodic task set ($U = 0.73$) is less than the least upper bound for two tasks ($U_{lub}(2) \simeq 0.83$), the periodic tasks are schedulable by RM. Notice that the guarantee test does not change in the presence of aperiodic requests, since background scheduling does not influence the execution of periodic tasks.

The major advantage of background scheduling is its simplicity. As shown in Fig. 5.2, two queues are needed to implement the scheduling mechanism: one (with

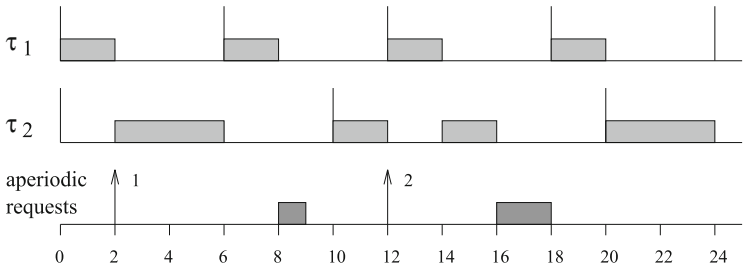


Fig. 5.1 Example of background scheduling of aperiodic requests under Rate Monotonic

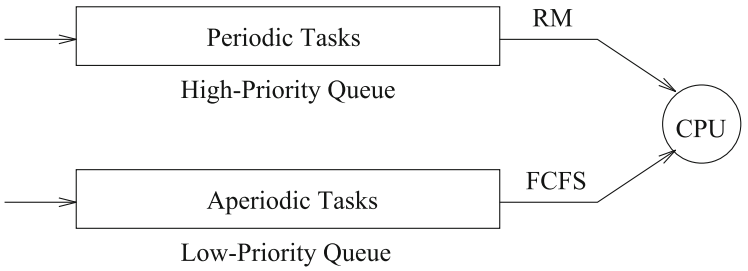


Fig. 5.2 Scheduling queues required for background scheduling

a higher priority) dedicated to periodic tasks and the other (with a lower priority) reserved for aperiodic requests. The two queueing strategies are independent and can be realized by different algorithms, such as RM for periodic tasks and First Come First Served (FCFS) for aperiodic requests. Tasks are taken from the aperiodic queue only when the periodic queue is empty. The activation of a new periodic instance causes any aperiodic tasks to be immediately preempted.

5.3 Polling Server

The average response time of aperiodic tasks can be improved with respect to background scheduling through the use of a *server*, that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a period T_s and a computation time C_s, called server *capacity*, or server *budget*. In general, the server is scheduled with the same algorithm used for the periodic tasks, and, once active, it serves the aperiodic requests within the limit of its budget. The ordering of aperiodic requests does not depend on the scheduling algorithm used for periodic tasks, and it can be done by arrival time, computation time, deadline, or any other parameter.

The *Polling Server* (PS) is an algorithm based on such an approach. At regular intervals equal to the period T_s, PS becomes active and serves the pending aperiodic requests within the limit of its capacity C_s. If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the budget originally allocated for aperiodic service is discharged and given periodic tasks [LSS87, SSL89]. Note that if an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next period, when the server capacity is replenished at its full value.

Figure 5.3 illustrates an example of aperiodic service obtained through a Polling Server scheduled by RM. The aperiodic requests are reported on the third row, whereas the fourth row shows the server capacity as a function of time. Numbers beside the arrows indicate the computation times associated with the requests.

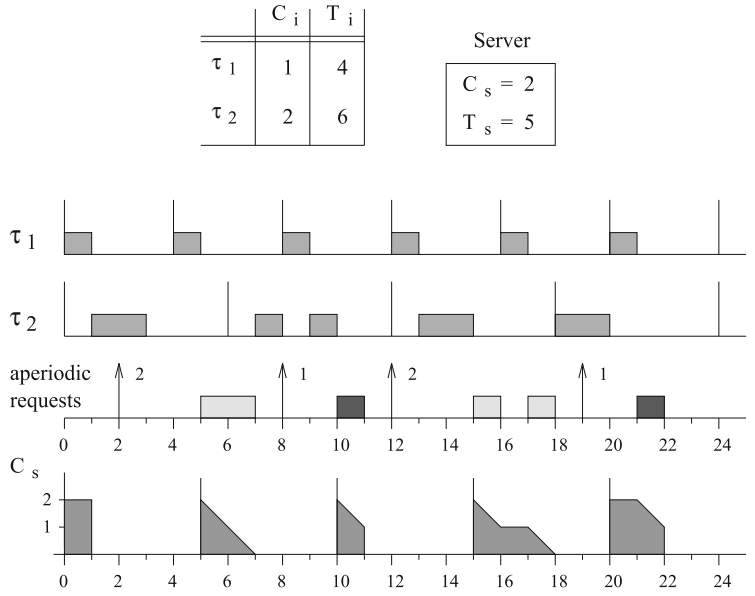


Fig. 5.3 Example of a Polling Server scheduled by RM

In the example shown in Fig. 5.3, the Polling Server has a period $T_s = 5$ and a capacity $C_s = 2$, so it runs with an intermediate priority with respect to the other periodic tasks. At time $t = 0$, the processor is assigned to task τ_1, which is the highest-priority task according to RM. At time $t = 1$, τ_1 completes its execution and the processor is assigned to PS. However, since no aperiodic requests are pending, the server suspends itself and its capacity is used by periodic tasks. As a consequence, the request arriving at time $t = 2$ cannot receive immediate service but must wait until the beginning of the second server period ($t = 5$). At this time, the capacity is replenished at its full value ($C_s = 2$) and used to serve the aperiodic task until completion. Note that, since the capacity has been totally consumed, no other aperiodic requests can be served in this period; thus, the server becomes idle.

The second aperiodic request receives the same treatment. However, note that since the second request only uses half of the server capacity, the remaining half is discarded because no other aperiodic tasks are pending. Also note that, at time $t = 16$, the third aperiodic request is preempted by task τ_1, and the server capacity is preserved.

5.3.1 Schedulability Analysis

We first consider the problem of guaranteeing a set of hard periodic tasks in the presence of soft aperiodic tasks handled by a Polling Server. Then, we show how to derive a schedulability test for firm aperiodic requests.

The schedulability of periodic tasks can be guaranteed by evaluating the interference introduced by the Polling Server on periodic execution. In the worst case, such an interference is the same as the one introduced by an equivalent periodic task having a period equal to T_s and a computation time equal to C_s. In fact, independently of the number of aperiodic tasks handled by the server, a maximum time equal to C_s is dedicated to aperiodic requests at each server period. As a consequence, the processor utilization factor of the Polling Server is $U_s = C_s/T_s$, and hence the schedulability of a periodic set with n tasks and utilization U_p can be guaranteed if

$$U_p + U_s \leq U_{lub}(n+1).$$

If periodic tasks (including the server) are scheduled by RM, the schedulability test becomes:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq (n+1)[2^{1/(n+1)} - 1].$$

Note that more Polling Servers can be created and execute concurrently on different aperiodic task sets. For example, a high-priority server could be reserved for a subset of important aperiodic tasks, whereas a lower-priority server could be used to handle less important requests. In general, in the presence of m servers, a set of n periodic tasks is schedulable by RM if

$$U_p + \sum_{j=1}^{m} U_{s_j} \leq U_{lub}(n+m).$$

A more precise schedulability test can be derived using the same technique adopted for the Liu and Layland bound, by assuming that PS is the highest-priority task in the system. To simplify the computation, the worst-case relations among the tasks are first determined, and then the lower bound is computed against the worst-case model.

Consider a set of n periodic tasks, $\tau_1, \ldots, \tau_n$, ordered by increasing periods, and a PS server with a highest priority. The worst-case scenario for a set of periodic tasks that fully utilize the processor is the one illustrated in Fig. 5.4, where tasks are characterized by the following parameters:

$$\begin{cases} C_s = T_1 - T_s \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \quad \ldots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = 2T_s - T_n. \end{cases}$$

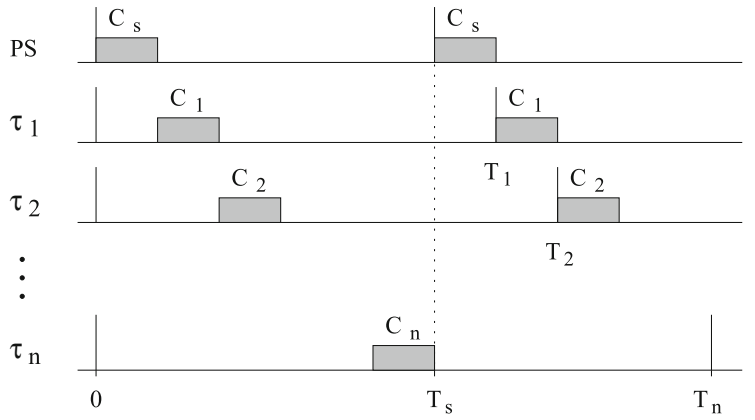


Fig. 5.4 Worst-case scenario for n periodic tasks and a Polling Server (PS) with the highest priority

The resulting utilization is then

$$\begin{aligned} U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} = \\ &= U_s + \frac{T_2 - T_1}{T_1} + \ldots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{2T_s - T_n}{T_n} = \\ &= U_s + \frac{T_2}{T_1} + \ldots + \frac{T_n}{T_{n-1}} + \left(\frac{2T_s}{T_1}\right)\frac{T_1}{T_n} - n. \end{aligned}$$

Defining

$$\begin{cases} R_s &= T_1/T_s \\ R_i &= T_{i+1}/T_i \\ K &= 2T_s/T_1 = 2/R_s \end{cases}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{R_1 R_2 \ldots R_{n-1}} - n.$$

Following the approach used for RM, we minimize U over R_i, $i = 1, \ldots, n-1$. Hence,

$$\frac{\partial U}{\partial R_i} = 1 - \frac{K}{R_i^2(\prod_{j \neq i}^{n-1} R_j)}.$$

Thus, defining $P = R_1 R_2 \ldots R_{n-1}$, U is minimum when

$$\begin{cases} R_1 P = K \\ R_2 P = K \\ \quad \ldots \\ R_{n-1} P = K \end{cases}$$

that is, when all R_i have the same value:

$$R_1 = R_2 = \ldots = R_{n-1} = K^{1/n}.$$

Substituting this value in U, we obtain:

$$\begin{aligned} U_{lub} - U_s &= (n-1)K^{1/n} + \frac{K}{K^{(1-1/n)}} - n = \\ &= nK^{1/n} - K^{1/n} + K^{1/n} - n = \\ &= n(K^{1/n} - 1) \end{aligned}$$

that is,

$$U_{lub} = U_s + n(K^{1/n} - 1). \tag{5.1}$$

Now, noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{T_s} = R_s - 1$$

we have:

$$R_s = (U_s + 1).$$

Thus, K can be rewritten as

$$K = \frac{2}{R_s} = \frac{2}{U_s + 1},$$

and finally

$$U_{lub} = U_s + n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]. \tag{5.2}$$

Taking the limit of Eq. (5.1) as $n \to \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n\to\infty} U_{lub} = U_s + \ln(K) = U_s + \ln\left(\frac{2}{U_s+1}\right). \tag{5.3}$$

Thus, given a set of n periodic tasks and a Polling Server with utilization factors U_p and U_s, respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_s + n\left(K^{1/n} - 1\right)$$

that is, if

$$U_p \leq n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]. \tag{5.4}$$

A plot of Eq. (5.3) as a function of U_s is shown in Fig. 5.5. For comparison, the RM bound is also reported in the plot. Notice that the schedulability test expressed in Eq. (5.4) is also valid for all servers that behave like a periodic task.

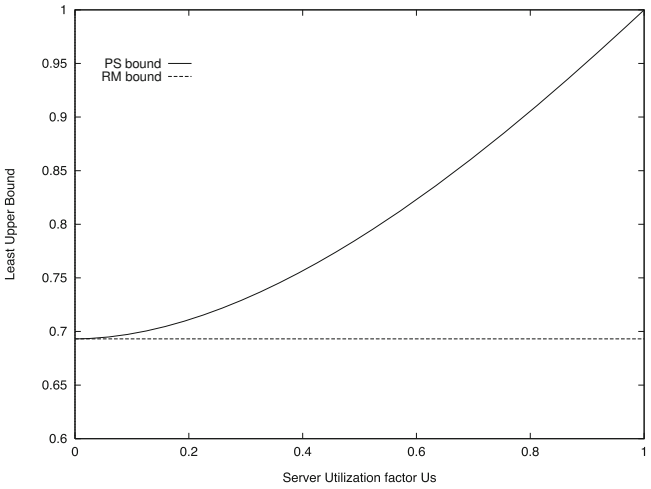


Fig. 5.5 Schedulability bound for periodic tasks and PS as a function of the server utilization factor U_s

Using the hyperbolic bound, the guarantee test for a task set in the presence of a Polling Server can be performed as follows:

$$\prod_{i=1}^{n}(U_i + 1) \leq \frac{2}{U_s + 1}. \tag{5.5}$$

Finally, the response time of a periodic task τ_i in the presence of a Polling Server at the highest priority can be found as the smallest integer satisfying the following recurrent relation:

$$R_i = C_i + \left\lceil \frac{R_i}{T_s} \right\rceil C_s + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{5.6}$$

5.3.2 Dimensioning a Polling Server

Given a set of periodic tasks, how can we compute the server parameters (C_s and T_s) that can guarantee a feasible schedule? First of all, we need to compute the maximum server utilization U_s^{max} that guarantees the feasibility of the task set. Since the response time is not easy to manipulate, due to the ceiling functions, we can derive U_s^{max} from the hyperbolic test of Eq. (5.5), which is tighter than the utilization test of Eq. (5.4). If we define

$$P \stackrel{\text{def}}{=} \prod_{i=1}^{n}(U_i + 1), \tag{5.7}$$

for the schedulability of the task set, from Eq. (5.5), it must be

$$P \leq \frac{2}{U_s + 1}$$

that is

$$U_s \leq \frac{2 - P}{P}.$$

Hence,

$$U_s^{max} = \frac{2 - P}{P}. \tag{5.8}$$

Thus, U_s must be set to be less than or equal to U_s^{max}. For a given U_s, however, there is an infinite number of pairs (C_s, T_s) leading to the same utilization, so how

can we select the pair that enhances aperiodic responsiveness? A simple solution is to assign the server the highest priority, that is, the smallest period, under Rate Monotonic. However, it is not useful to set $T_s < T_1$, since a smaller T_s implies a smaller C_s, which would cause higher fragmentation (i.e., higher runtime overhead) in aperiodic execution. Hence, assuming that priority ties between periodic tasks and the server are broken in favor of the server, then the highest priority of the server can be achieved by setting $T_s = T_1$, and then $C_s = U_s T_s$.

5.3.3 Aperiodic Guarantee

This section shows how to estimate the response time of an aperiodic job handled by a Polling Server, in order to possibly perform an online guarantee of firm aperiodic requests characterized by a deadline. To do that, consider the case of a single aperiodic job J_a, arrived at time r_a, with computation time C_a and deadline D_a. Since, in the worst case, the job can wait for at most one period before receiving service, if $C_a \leq C_s$, the request is certainly completed within two server periods. Thus, it is guaranteed if

$$2T_s \leq D_a.$$

For arbitrary computation times, the aperiodic request is certainly completed in $\lceil C_a/C_s \rceil$ server periods; hence, it is guaranteed if

$$T_s + \left\lceil \frac{C_a}{C_s} \right\rceil T_s \leq D_a.$$

This schedulability test is only sufficient because it does not consider when the server executes within its period.

A sufficient and necessary schedulability test can be derived for the case in which the PS has the highest priority among the periodic tasks, that is, the shortest period. In this case, in fact, it always executes at the beginning of its periods; thus, the finishing time of the aperiodic request can be estimated precisely. As shown in Fig. 5.6, by defining

$$F_a \stackrel{\text{def}}{=} \left\lceil \frac{C_a}{C_s} \right\rceil - 1$$

$$next(r_a) \stackrel{\text{def}}{=} \left\lceil \frac{r_a}{T_s} \right\rceil T_s$$

the initial delay of request J_a is given by $\Delta_a = next_a(r_a) - r_a$. Then, since $F_a C_s$ is the total budget consumed by J_a in F_a server periods, the residual execution to be done in the next server period is

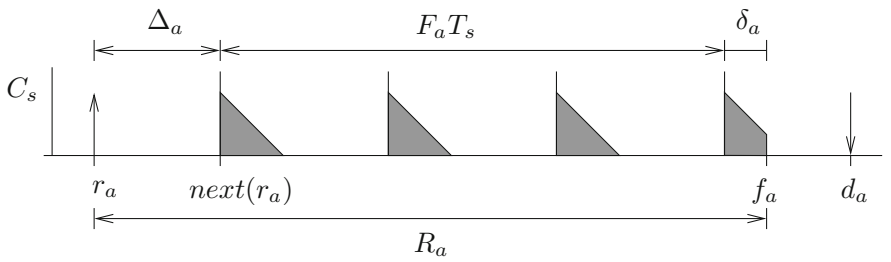


Fig. 5.6 Response time of an aperiodic job scheduled by a Polling Server with the highest priority

$$\delta_a = C_a - F_a C_s.$$

As a consequence, the response time R_a can be computed as

$$R_a = \Delta_a + F_a T_s + \delta_a,$$

which can be also written as

$$R_a = \Delta_a + C_a + F_a(T_s - C_s). \tag{5.9}$$

Note that the term $F_a(T_s - C_s)$ in Eq. (5.9) represents the delay introduced by the F_a inactive server intervals, each of size $(T_s - C_s)$.

Then, the schedulability of the aperiodic job can be guaranteed if and only if $R_a \leq D_a$.

5.4 Deferrable Server

The *Deferrable Server* (DS) algorithm is a service technique introduced by Lehoczky, Sha, and Strosnider in [LSS87, SLS95] to improve the average response time of aperiodic requests with respect to polling service. As the Polling Server, the DS algorithm creates a periodic task (usually having a high priority) for servicing aperiodic requests. However, unlike polling, DS preserves its capacity if no requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be serviced at the same server's priority at anytime, as long as the capacity has not been exhausted. At the beginning of any server period, the capacity is replenished at its full value.

The DS algorithm is illustrated in Fig. 5.7 using the same task set and the same server parameters ($C_s = 2$, $T_s = 5$) considered in Fig. 5.3. At time $t = 1$, when τ_1 is completed, no aperiodic requests are pending; hence, the processor is assigned to task τ_2. However, the DS capacity is not used for periodic tasks, but it is preserved

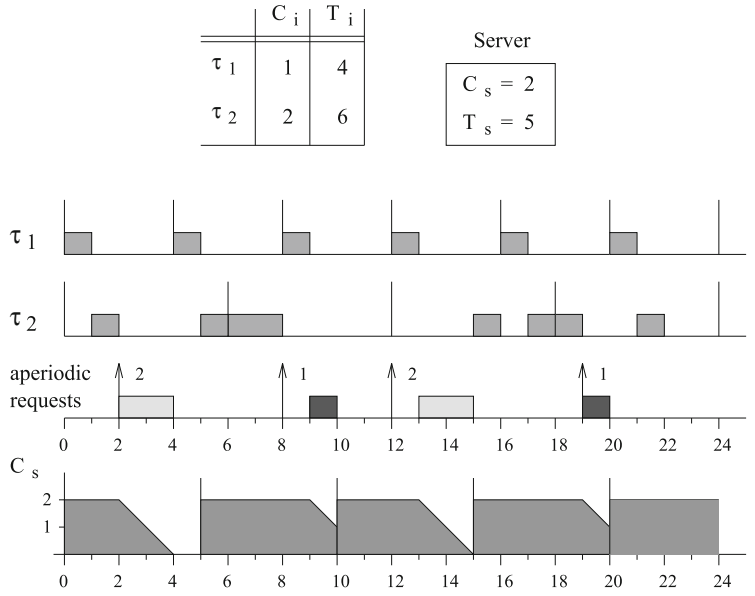


Fig. 5.7 Example of a Deferrable Server scheduled by RM

for future aperiodic arrivals. Thus, when the first aperiodic request arrives at time $t = 2$, it receives immediate service. Since the capacity of the server is exhausted at time $t = 4$, no other requests can be serviced before the next period. At time $t = 5$, C_s is replenished at its full value and preserved until the next arrival. The second request arrives at time $t = 8$, but it is not served immediately because τ_1 is active and has a higher priority.

Thus, DS provides much better aperiodic responsiveness than polling, since it preserves the capacity until it is needed. Shorter response times can be achieved by creating a Deferrable Server having the highest priority among the periodic tasks. An example of high-priority DS is illustrated in Fig. 5.8. Notice that the second aperiodic request preempts task τ_1, being $C_s > 0$ and $T_s < T_1$, and it entirely consumes the capacity at time $t = 10$. When the third request arrives at time $t = 11$, the capacity is zero; hence, its service is delayed until the beginning of the next server period. The fourth request receives the same treatment because it arrives at time $t = 16$, when C_s is exhausted.

5.4.1 Schedulability Analysis

Any schedulability analysis related to the Rate-Monotonic algorithm has been done on the implicit assumption that a periodic task cannot suspend itself, but must execute whenever it is the highest-priority task ready to run (assumption A5 in

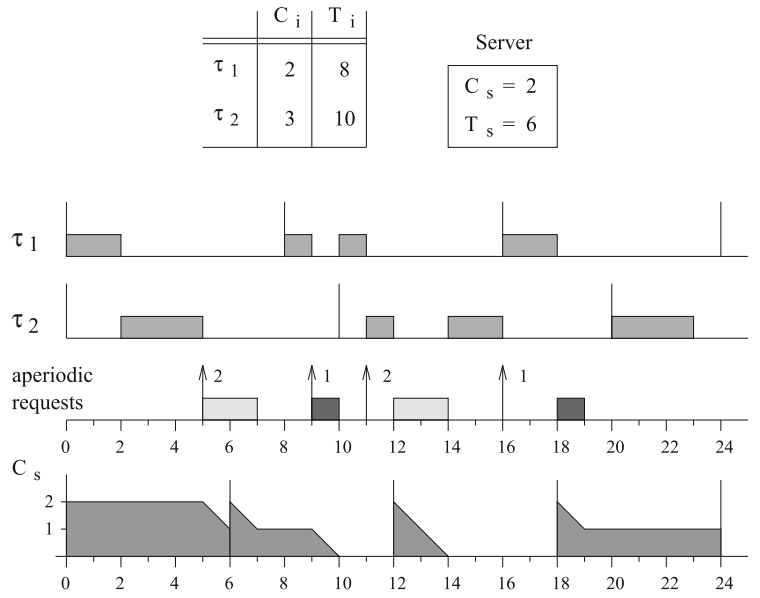


Fig. 5.8 Example of high-priority Deferrable Server

Sect. 4.1). It is easy to see that the Deferrable Server violates this basic assumption. In fact, the schedule illustrated in Fig. 5.8 shows that DS does not execute at time $t = 0$, although it is the highest-priority task ready to run, but it *defers* its execution until time $t = 5$, which is the arrival time of the first aperiodic request.

If a periodic task *defers* its execution when it could execute immediately, then a lower-priority task could miss its deadline even if the task set was schedulable. Figure 5.9 illustrates this phenomenon by comparing the execution of a periodic task to the one of a Deferrable Server with the same period and execution time.

The periodic task set considered in this example consists of two tasks, τ_1 and τ_2, having the same computation time ($C_1 = C_2 = 2$) and different periods ($T_1 = 4$, $T_2 = 5$). As shown in Fig. 5.9a, the two tasks are schedulable by RM. However, if τ_1 is replaced with a Deferrable Server having the same period and execution time, the low-priority task τ_2 can miss its deadline depending on the sequence of aperiodic arrivals. Figure 5.9b shows a particular sequence of aperiodic requests that cause τ_2 to miss its deadline at time $t = 15$. This happens because, at time $t = 8$, DS does not execute (as a normal periodic task would do) but preserves its capacity for future requests. This deferred execution, followed by the servicing of two consecutive aperiodic requests in the interval [10, 14], prevents task τ_2 from executing during this interval, causing its deadline to be missed.

Such an invasive behavior of the Deferrable Server results in a lower schedulability bound for the periodic task set. The calculation of the least upper bound of the processor utilization factor in the presence of Deferrable Server is shown in the next section.

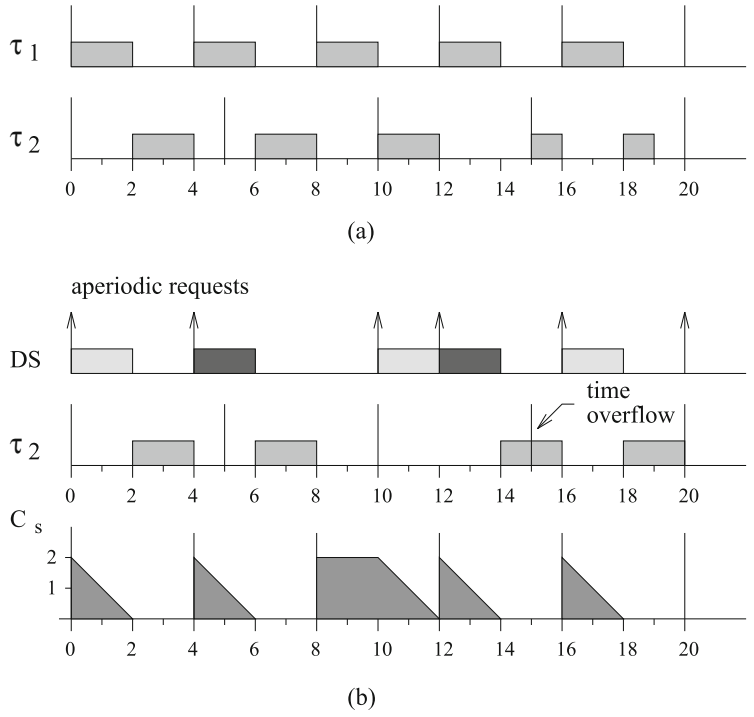


Fig. 5.9 DS is not equivalent to a periodic task. In fact, the periodic set $\{\tau_1, \tau_2\}$ is schedulable by RM (**a**); however, if we replace τ_1 with DS, τ_2 misses its deadline (**b**)

5.4.1.1 Calculation of U_{lub} for RM+DS

The schedulability bound for a set of periodic tasks with a Deferrable Server is derived under the same basic assumptions used in Chap. 4 to compute U_{lub} for RM. To simplify the computation of the bound for n tasks, we first determine the worst-case relations among the tasks, and then we derive the lower bound against the worst-case model [LSS87].

Consider a set of n periodic tasks, $\tau_1, \ldots, \tau_n$, ordered by increasing periods, and a Deferrable Server with a higher priority. The worst-case condition for the periodic tasks, as derived for the RM analysis, is such that $T_1 < T_n < 2T_1$. In the presence of a DS, however, the derivation of the worst-case is more complex and requires the analysis of three different cases, as discussed in [SLS95]. For the sake of clarity, here we analyze one case only, the most general, in which DS may execute three times within the period of the highest-priority periodic task. This happens when DS defers its service at the end of its period and also executes at the beginning of the next period. In this situation, depicted in Fig. 5.10, the full processor utilization is achieved by the following tasks' parameters:

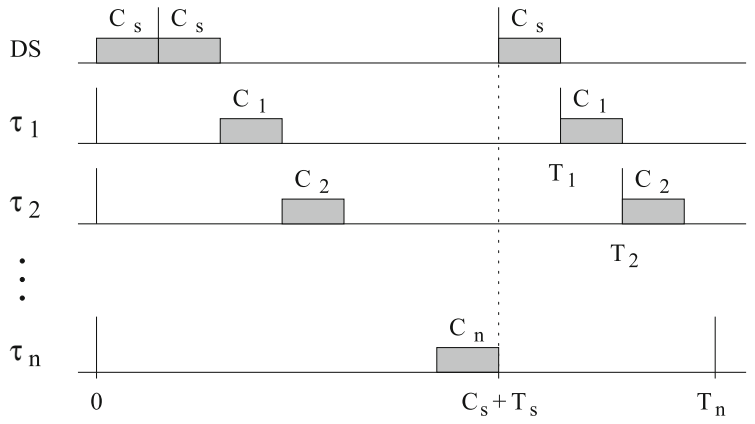


Fig. 5.10 Worst-case task relations for a Deferrable Server

$$\begin{cases} C_s &= T_1 - (T_s + C_s) = \frac{T_1 - T_s}{2} \\ C_1 &= T_2 - T_1 \\ C_2 &= T_3 - T_2 \\ &\dots \\ C_{n-1} &= T_n - T_{n-1} \\ C_n &= T_s - C_s - \sum_{i=1}^{n-1} C_i \;=\; \frac{3T_s + T_1 - 2T_n}{2}. \end{cases}$$

Hence, the resulting utilization is

$$\begin{aligned} U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} = \\ &= U_s + \frac{T_2 - T_1}{T_1} + \ldots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{3T_s + T_1 - 2T_n}{2T_n} = \\ &= U_s + \frac{T_2}{T_1} + \ldots + \frac{T_n}{T_{n-1}} + \left(\frac{3T_s}{2T_1} + \frac{1}{2}\right)\frac{T_1}{T_n} - n. \end{aligned}$$

Defining

$$\begin{cases} R_s &= T_1/T_s \\ R_i &= T_{i+1}/T_i \\ K &= \frac{1}{2}(3T_s/T_1 + 1) \end{cases}$$

and noting that

$$R_1 R_2 \ldots R_{n-1} = \frac{T_n}{T_1},$$

the utilization factor may be written as

$$U = U_s + \sum_{i=1}^{n-1} R_i + \frac{K}{R_1 R_2 \dots R_{n-1}} - n.$$

Following the approach used for RM, we minimize U over R_i, $i = 1, \dots, n-1$. Hence,

$$\frac{\partial U}{\partial R_i} = 1 - \frac{K}{R_i^2 (\prod_{j \neq i}^{n-1} R_j)}.$$

Thus, defining $P = R_1 R_2 \dots R_{n-1}$, U is minimum when

$$\begin{cases} R_1 P = K \\ R_2 P = K \\ \quad \dots \\ R_{n-1} P = K \end{cases}$$

that is, when all R_i have the same value:

$$R_1 = R_2 = \dots = R_{n-1} = K^{1/n}.$$

Substituting this value in U, we obtain:

$$\begin{aligned} U_{lub} - U_s &= (n-1)K^{1/n} + \frac{K}{K^{(1-1/n)}} - n = \\ &= nK^{1/n} - K^{1/n} + K^{1/n} - n = \\ &= n(K^{1/n} - 1) \end{aligned}$$

that is,

$$U_{lub} = U_s + n(K^{1/n} - 1). \tag{5.10}$$

Now, noting that

$$U_s = \frac{C_s}{T_s} = \frac{T_1 - T_s}{2T_s} = \frac{R_s - 1}{2}$$

we have:

$$R_s = (2U_s + 1).$$

Thus, K can be rewritten as

$$K = \left(\frac{3}{2R_s} + \frac{1}{2}\right) = \frac{U_s + 2}{2U_s + 1},$$

and finally

$$U_{lub} = U_s + n\left[\left(\frac{U_s + 2}{2U_s + 1}\right)^{1/n} - 1\right]. \tag{5.11}$$

Taking the limit as $n \to \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n\to\infty} U_{lub} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right). \tag{5.12}$$

A plot of Eq. (5.12) as a function of U_s is shown in Fig. 5.11. For comparison, the RM bound is also reported in the plot. Notice that for $U_s < 0.4$, the presence of DS worsens the RM bound, whereas for $U_s > 0.4$, the RM bound is improved.

Deriving Eq. (5.12) with respect to U_s, we can find the absolute minimum value of U_{lub}:

$$\frac{\partial U_{lub}}{\partial U_s} = 1 + \frac{(2U_s + 1)}{(U_s + 2)}\frac{(2U_s + 1) - 2(U_s + 2)}{(2U_s + 1)^2} = \frac{2U_s^2 + 5U_s - 1}{(U_s + 2)(2U_s + 1)}.$$

The value of U_s that minimizes the above expression is

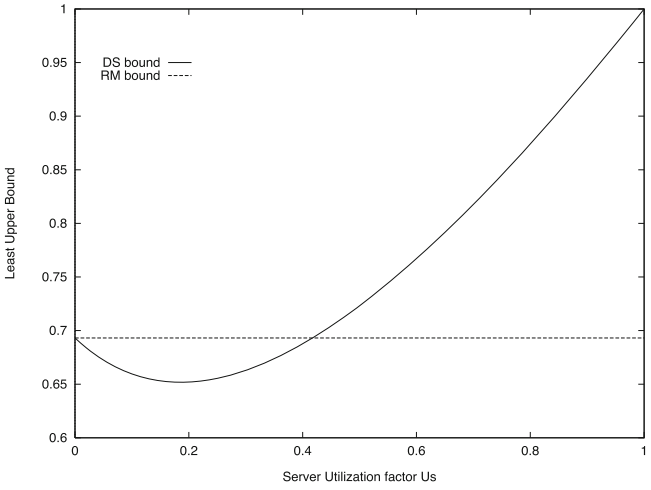


Fig. 5.11 Schedulability bound for periodic tasks and DS as a function of the server utilization factor U_s

$$U_s^* = \frac{\sqrt{33}-5}{4} \simeq 0.186,$$

so the minimum value of U_{lub} is $U_{lub}^* \simeq 0.652$.

In summary, given a set of n periodic tasks with utilization U_p and a Deferrable Server with utilization U_s, respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p \leq n\left(K^{1/n}-1\right). \tag{5.13}$$

where

$$K = \frac{U_s+2}{2U_s+1},$$

Using the hyperbolic bound, the guarantee test for a task set in the presence of a Deferrable Server can be performed as follows:

$$\prod_{i=1}^{n}(U_i+1) \leq \frac{U_s+2}{2U_s+1}. \tag{5.14}$$

5.4.2 Dimensioning a Deferrable Server

Following the same procedure described in Sect. 5.3.2, the maximum utilization U_s^{max} for a Deferrable Server can easily be computed from Eq. (5.14), which can be written, defining P as in Eq. (5.7), as

$$P \leq \frac{U_s+2}{2U_s+1}$$

that is

$$U_s \leq \frac{2-P}{2P-1}.$$

Hence,

$$U_s^{max} = \frac{2-P}{2P-1}. \tag{5.15}$$

Then, T_s can be set equal to the smallest period T_1, so that DS is executed by RM with the highest priority (assuming that priority ties are broken in favor of the server), and finally $C_s = U_s T_s$.

5.4.3 Aperiodic Guarantee

The online guarantee of a firm aperiodic job can be performed by estimating its worst-case response time in the case of a DS with the highest priority. Since DS preserves its execution time, let $c_s(t)$ be the value of its capacity at time t, and let J_a an aperiodic job with computation time C_a and relative deadline D_a, arriving at time $t = r_a$, when no other aperiodic requests are pending. Then, if $next(r_a) = \lceil r_a/T_s \rceil T_s$ is the next server activation after time r_a, the two cases illustrated in Fig. 5.12 can occur:

1. Case (a): $c_s(t) \leq next(r_a) - r_a$. In this case, the capacity is completely discharged within the current period, and a portion $C_0 = c_s(t)$ of J_a is executed in the current server period.
2. Case (b): $c_s(t) > next(r_a) - r_a$. In this case, the period ends before the server capacity is completely discharged; thus, a portion $C_0 = next(r_a) - r_a$ of J_a is executed in the current server period.

In general, the portion C_0 executed in the current server period is equal to

$$C_0 = \min\{c_s(t), \quad next(r_a) - r_a\}.$$

Using the same notation introduced for Polling Server, we define:

$$\begin{cases} \Delta_a = next(r_a) - r_a \\ F_a = \left\lceil \frac{C_a - C_0}{C_s} \right\rceil - 1 \\ \delta_a = C_a - C_0 - F_a C_s. \end{cases}$$

Hence, as depicted in Fig. 5.13, the response time R_a of job J_a can be computed as

$$R_a = \Delta_a + F_a T_s + \delta_a,$$

which can be also written as

$$R_a = \Delta_a + C_a - C_0 + F_a(T_s - C_s). \tag{5.16}$$

Note that the term $F_a(T_s - C_s)$ in Eq. (5.16) represents the delay introduced by the F_a inactive server intervals, each of size $(T_s - C_s)$.

Then, the schedulability of the aperiodic job can be guaranteed if and only if $R_a \leq D_a$.

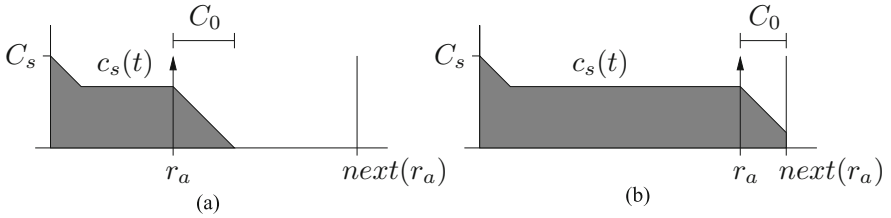


Fig. 5.12 Execution of J_a in the first server period

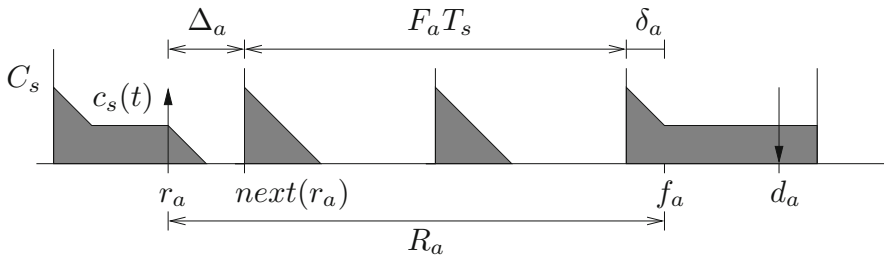


Fig. 5.13 Response time of an aperiodic job scheduled by a Deferrable Server with the highest priority

5.5 Priority Exchange

The *Priority Exchange* (PE) algorithm is a scheduling technique introduced by Lehoczky, Sha, and Strosnider in [LSS87] for servicing a set of soft aperiodic requests along with a set of hard periodic tasks. With respect to DS, PE has a slightly worse performance in terms of aperiodic responsiveness but provides a better schedulability bound for the periodic task set.

Like DS, the PE algorithm uses a periodic server (usually at a high priority) for servicing aperiodic requests. However, it differs from DS in the manner in which the capacity is preserved. Unlike DS, PE preserves its high-priority capacity by exchanging it for the execution time of a lower-priority periodic task.

At the beginning of each server period, the capacity is replenished at its full value. If aperiodic requests are pending and the server is the ready task with the highest priority, then the requests are serviced using the available capacity; otherwise C_s is exchanged for the execution time of the active periodic task with the highest priority.

When a priority exchange occurs between a periodic task and a PE server, the periodic task executes at the priority level of the server while the server accumulates a capacity at the priority level of the periodic task. Thus, the periodic task advances its execution, and the server capacity is not lost but preserved at a lower priority. If no aperiodic requests arrive to use the capacity, priority exchange continues with other lower-priority tasks until either the capacity is used for aperiodic service or it is degraded to the priority level of background processing. Since the objective of

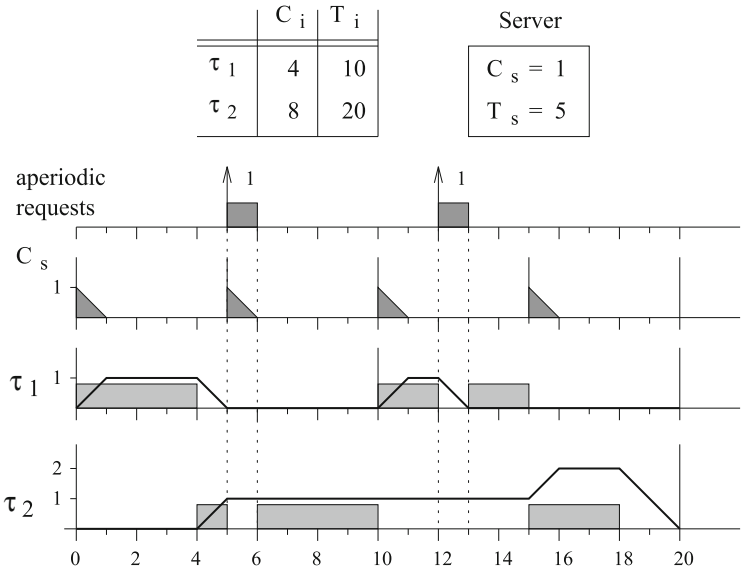


Fig. 5.14 Example of aperiodic service under a PE server

the PE algorithm is to provide high responsiveness to aperiodic requests, all priority ties are broken in favor of aperiodic tasks.

Figure 5.14 illustrates an example of aperiodic scheduling using the PE algorithm. In this example, the PE server is created with a period $T_s = 5$ and a capacity $C_s = 1$. Since the aperiodic time managed by the PE algorithm can be exchanged with all periodic tasks, the capacity accumulated at each priority level as a function of time is represented in overlapping with the schedule of the corresponding periodic task. In particular, the first timeline of Fig. 5.14 shows the aperiodic requests arriving in the system, the second timeline visualizes the capacity available at PE's priority, whereas the third and the fourth ones show the capacities accumulated at the corresponding priority levels as a consequence of the priority exchange mechanism.

At time $t = 0$, the PE server is brought at its full capacity, but no aperiodic requests are pending, so C_s is exchanged with the execution time of task τ_1. As a result, τ_1 advances its execution and the server accumulates one unit of time at the priority level of τ_1. At time $t = 4$, τ_1 completes and τ_2 begins to execute. Again, since no aperiodic tasks are pending, another exchange takes place between τ_1 and τ_2. At time $t = 5$, the capacity is replenished at the server priority, and it is used to execute the first aperiodic request. At time $t = 10$, C_s is replenished at the highest priority, but it is degraded to the priority level of τ_1 for lack of aperiodic tasks. At time $t = 12$, the capacity accumulated at the priority level of τ_1 is used to execute the second aperiodic request. At time $t = 15$, a new high-priority replenishment takes place, but the capacity is exchanged with the execution time of τ_2. Finally,

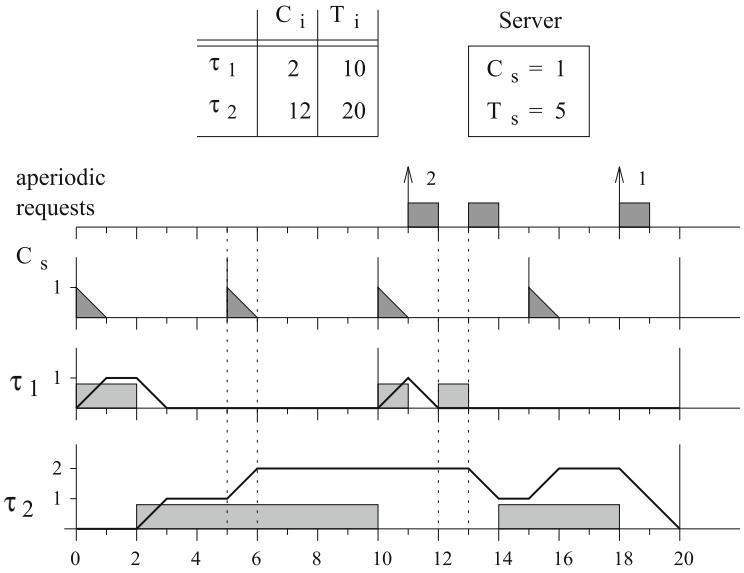


Fig. 5.15 Example of aperiodic service under a PE server

at time $t = 18$, the remaining capacity accumulated at the priority level of τ_2 is gradually discarded because no tasks are active.

Note that the capacity overlapped to the schedule of a periodic task indicates, at any instant, the amount of time by which the execution of that task is advanced with respect to the case of no exchange.

Another example of aperiodic scheduling under the PE algorithm is depicted in Fig. 5.15. Here, at time $t = 5$, the capacity of the server immediately degrades down to the lowest-priority level of τ_2, since no aperiodic requests are pending and τ_1 is idle. At time $t = 11$, when request J_1 arrives, it is interesting to observe that the first unit of computation time is immediately executed by using the capacity accumulated at the priority level of τ_1. Then, since the remaining capacity is available at the lowest-priority level and τ_1 is still active, J_1 is preempted by τ_1 and is resumed at time $t = 13$, when τ_1 completes.

5.5.1 Schedulability Analysis

Considering that, in the worst case, a PE server behaves as a periodic task, the schedulability bound for a set of periodic tasks running along with a Priority Exchange server is the same as the one derived for the Polling Server. Hence, assuming that PE is the highest-priority task in the system, we have that

$$U_{lub} = U_s + n\left(K^{1/n} - 1\right). \tag{5.17}$$

where

$$K = \frac{2}{U_s + 1}.$$

Thus, given a set of n periodic tasks and a Priority Exchange server with utilization factors U_p and U_s, respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p \leq n\left[\left(\frac{2}{U_s + 1}\right)^{1/n} - 1\right]. \tag{5.18}$$

5.5.2 PE Versus DS

The DS and the PE algorithms represent two alternative techniques for enhancing aperiodic responsiveness over traditional background and polling approaches. Here, these techniques are compared in terms of performance, schedulability bound, and implementation complexity, in order to help a system designer in selecting the most appropriate method for a particular real-time application.

The DS algorithm is much simpler to implement than the PE algorithm, because it always maintains its capacity at the original priority level and never exchanges its execution time with lower-priority tasks, as the PE algorithm does. The additional work required by PE to manage and track priority exchanges increases the overhead of PE with respect to DS, especially when the number of periodic tasks is large. On the other hand, DS does pay schedulability penalty for its simplicity in terms of a lower utilization bound. This means that, for a given periodic load U_p, the maximum size of a DS server that can still guarantee the periodic tasks is smaller than the maximum size of a PE server.

The maximum utilizations for DS and PE, as a function of task utilizations, have been derived in Eqs. (5.15) and (5.8), respectively (since PE, like PS, behaves as a periodic task in terms of utilization). Hence, we have:

$$U_{DS}^{max} = \frac{2 - P}{2P - 1} \tag{5.19}$$

$$U_{PE}^{max} = \frac{2 - P}{P} \tag{5.20}$$

where

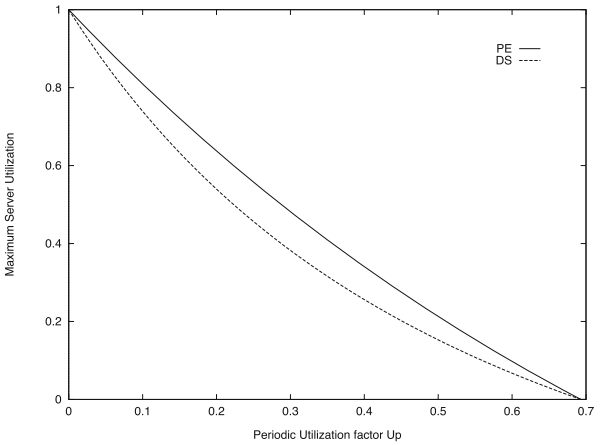


Fig. 5.16 Maximum server utilization as a function of the periodic load

$$P = \prod_{i=1}^{n} (U_i + 1).$$

Note that, if all the n periodic tasks have the same utilization $U_i = U_p/n$, the P factor can be expressed as a function of U_p as

$$P = \left(\frac{U_p}{n} + 1 \right)^n.$$

Note that assuming the same utilization for the periodic tasks is the worst-case situation for the task set, as clear from Fig. 4.11, since the hyperbole is tangent to the linear Liu and Layland bound.

A plot of the maximum server utilizations as a function of U_p (and for a large number of tasks) is shown in Fig. 5.16. Notice that, when $U_p = 0.6$, the maximum utilization for PE is 10%, whereas DS utilization cannot be greater than 7%. If instead $U_p = 0.3$, PE can have 48% utilization, while DS cannot go over 38%. The performance of the two algorithms in terms of average aperiodic response times is shown in Sect. 5.9.

As far as firm aperiodic tasks are concerned, the schedulability analysis under PE is much more complex than under DS. This is due to the fact that, in general, when an aperiodic request is handled by the PE algorithm, the server capacity can be distributed among $n + 1$ priority levels. Hence, calculating the finishing time of the request might require the construction of the schedule for all the periodic tasks up to the aperiodic deadline.

5.6 Sporadic Server

The *Sporadic Server* (SS) algorithm is another technique, proposed by Sprunt, Sha, and Lehoczky in [SSL89], which allows to enhance the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set.

The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way it replenishes its capacity. Whereas DS and PE periodically replenish their capacity to its full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

In order to simplify the description of the replenishment method used by SS, the following terms are defined:

P_{exe}	It denotes the priority level of the task which is currently executing.
P_s	It denotes the priority level associated with SS.
Active	SS is said to be *active* when $P_{exe} \geq P_s$.
Idle	SS is said to be *idle* when $P_{exe} < P_s$.
RT	It denotes the *replenishment time* at which the SS capacity will be replenished.
RA	It denotes the *replenishment amount* that will be added to the capacity at time RT.

Using this terminology, the capacity C_s consumed by aperiodic requests is replenished according to the following rule:

- The replenishment time RT is set as soon as SS becomes active and $C_s > 0$. Let t_A be such a time. The value of RT is set equal to t_A plus the server period ($RT = t_A + T_s$).
- The replenishment amount RA to be done at time RT is computed when SS becomes idle or C_s has been exhausted. Let t_I be such a time. The value of RA is set equal to the capacity consumed within the interval $[t_A, t_I]$.

An example of medium-priority SS is shown in Fig. 5.17. To facilitate the understanding of the replenishment rule, the intervals in which SS is active are also shown. At time $t = 0$, the highest-priority task τ_1 is scheduled, and SS becomes active. Since $C_s > 0$, a replenishment is set at time $RT_1 = t + T_s = 10$. At time $t = 1$, τ_1 completes, and, since no aperiodic requests are pending, SS becomes idle. Note that no replenishment takes place at time $RT_1 = 10$ ($RA_1 = 0$) because no capacity has been consumed in the interval [0, 1]. At time $t = 4$, the first aperiodic request J_1 arrives, and, since $C_s > 0$, SS becomes active and the request receives immediate service. As a consequence, a replenishment is set at $RT_2 = t + T_s = 14$. Then, J_1 is preempted by τ_1 at $t = 5$, is resumed at $t = 6$, and is completed at $t = 7$. At this time, the replenishment amount to be done at RT_2 is set equal to the capacity consumed in [4, 7]; that is, $RA_2 = 2$.

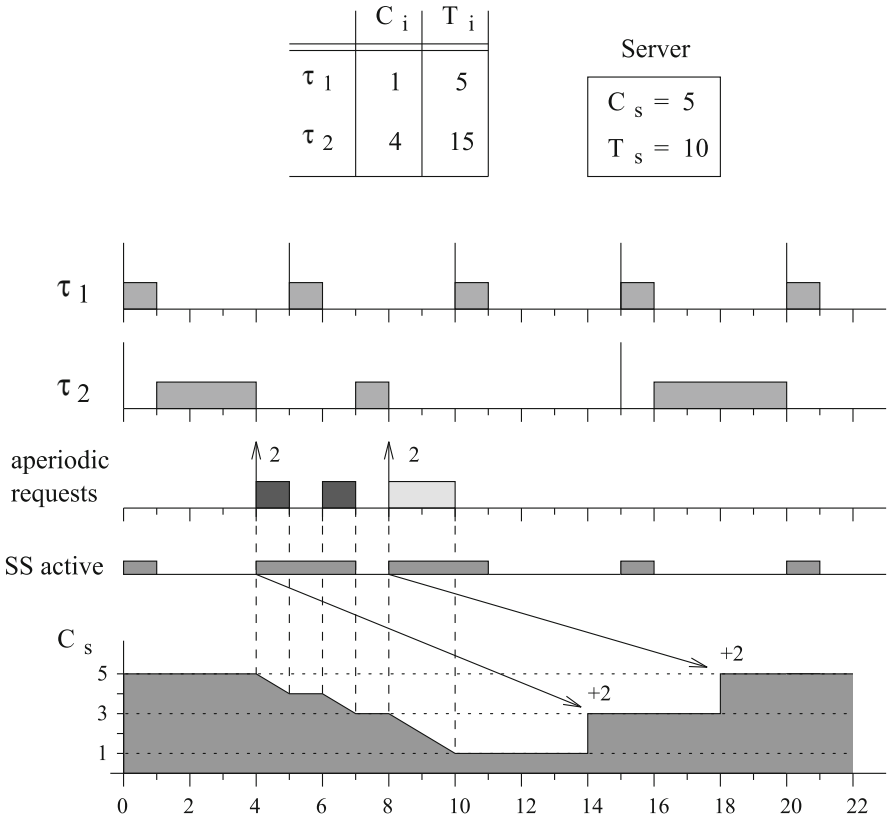


Fig. 5.17 Example of a medium-priority Sporadic Server

Notice that during preemption intervals, SS stays active. This allows to perform a single replenishment, even if SS provides a discontinuous service for aperiodic requests.

At time $t = 8$, SS becomes active again and a new replenishment is set at $RT_3 = t + T_s = 18$. At $t = 11$, SS becomes idle and the replenishment amount to be done at RT_3 is set to $RA_3 = 2$.

Figure 5.18 illustrates another example of aperiodic service in which SS is the highest-priority task. Here, the first aperiodic request arrives at time $t = 2$ and consumes the whole server capacity. Hence, a replenishment amount $RA_1 = 2$ is set at $RT_1 = 10$. The second request arrives when $C_s = 0$. In this case, the replenishment time RT_2 is set as soon as the capacity becomes greater than zero. Since this occurs at time $t = 10$, the next replenishment is set at time $RT_2 = 18$. The corresponding replenishment amount is established when J_2 completes and is equal to $RA_2 = 2$.

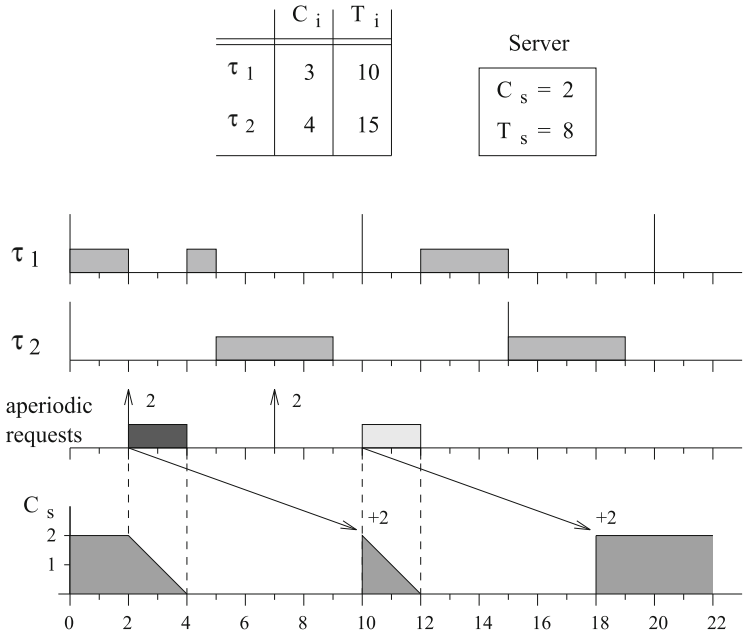


Fig. 5.18 Example of a high-priority Sporadic

5.6.1 Schedulability Analysis

The Sporadic Server violates one of the basic assumptions governing the execution of a standard periodic task. This assumption requires that once a periodic task is the highest-priority task that is ready to execute, it must execute. Like DS, in fact, SS defers its execution and preserves its capacity when no aperiodic requests are pending. However, we show that the replenishment rule used in SS compensates for any deferred execution and, from a scheduling point of view, SS can be treated as a normal periodic task with a period T_s and an execution time C_s. In particular, the following theorem holds [SSL89].

Theorem 5.1 (Sprunt-Sha-Lehoczky) *A periodic task set that is schedulable with a task τ_i is also schedulable if τ_i is replaced by a Sporadic Server with the same period and execution time.*

Proof The theorem is proved by showing that for any type of service, SS exhibits an execution behavior equivalent to one or more periodic tasks. Let t_A be the time at which C_s is full and SS becomes active, and let t_I be the time at which SS becomes idle, such that $[t_A, t_I]$ is a continuous interval during which SS remains active. The execution behavior of the server in the interval $[t_A, t_I]$ can be described by one of the following three cases (see Fig. 5.19):

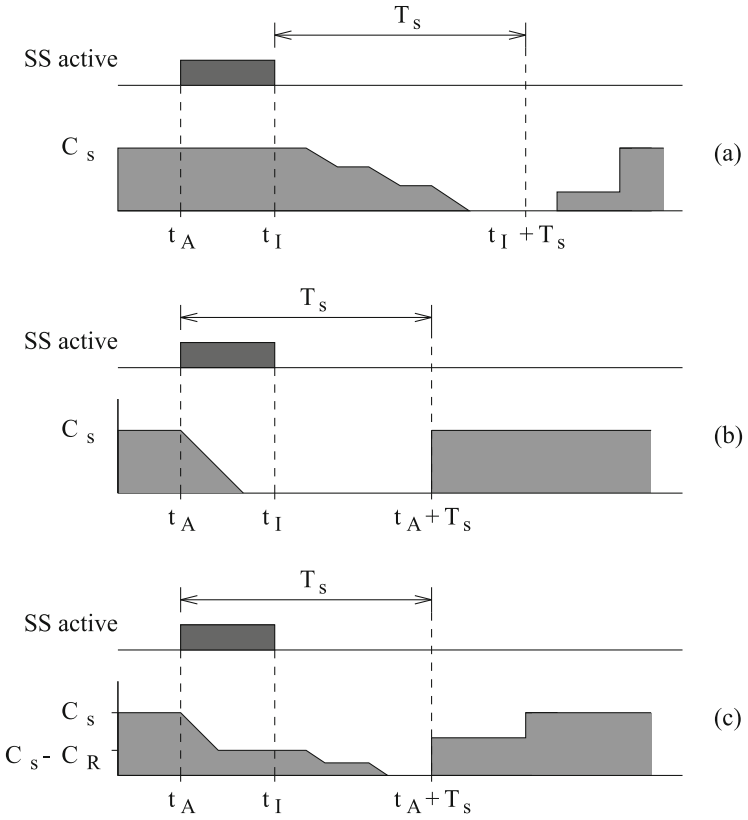


Fig. 5.19 Possible SS behavior during active intervals: (**a**) C_s is not consumed; (**b**) C_s is totally consumed; (**c**) C_s is partially consumed

1. No capacity is consumed.
2. The server capacity is totally consumed.
3. The server capacity is partially consumed.

Case 1. If no requests arrive in $[t_A, t_I]$, SS preserves its capacity and no replenishments can be performed before time $t_I + T_s$. This means that at most C_s units of aperiodic time can be executed in $[t_A, t_I + T_s]$. Hence, the SS behavior is identical to a periodic task $\tau_s(C_s, T_s)$ whose release time is delayed from t_A to t_I. As proved in Chap. 4 for RM, delaying the release of a periodic task cannot increase the response time of the other periodic tasks; therefore, this case does not jeopardize schedulability.

Case 2. If C_s is totally consumed in $[t_A, t_I]$, a replenishment of C_s units of time will occur at time $t_A + T_s$. Hence, SS behaves like a periodic task with period T_s and execution time C_s released at time t_A.

Case 3. If C_s is partially consumed in $[t_A, t_I]$, a replenishment will occur at time $t_A + T_s$, and the remaining capacity is preserved for future requests. Let C_R be the capacity consumed in $[t_A, t_I]$. In this case, the behavior of the server is equivalent to two periodic tasks, τ_x and τ_y, with periods $T_x = T_y = T_s$, and execution times $C_x = C_R$ and $C_y = C_s - C_R$, such that τ_x is released at t_A and τ_y is delayed until t_I. As in Case 1, the delay of τ_y has no schedulability effects.

Since in any servicing situation SS can be represented by one or more periodic tasks with period T_s and total execution time equal to C_s, the contribution of SS in terms of processor utilization is equal to $U_s = C_s/T_s$. Hence, from a schedulability point of view, SS can be replaced by a periodic task having the same utilization factor. □

Since SS behaves like a normal periodic task, the periodic task set can be guaranteed by the same schedulability test derived for the Polling Server. Hence, a set Γ of n periodic tasks with utilization factor U_p scheduled along with a Sporadic Server with utilization U_s is schedulable under RM if

$$U_p \leq n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \tag{5.21}$$

For large n, Γ is schedulable if

$$U_p \leq \ln \left(\frac{2}{U_s + 1} \right) \tag{5.22}$$

Using the hyperbolic bound, a periodic task set with utilization U_p is schedulable under RM+SS if

$$P \stackrel{\text{def}}{=} \prod_{i=1}^{n} (U_i + 1) \leq \left(\frac{2}{U_s + 1} \right). \tag{5.23}$$

And the maximum server size that preserves schedulability is

$$U_{SS}^{max} = \frac{2 - P}{P}. \tag{5.24}$$

As far as firm aperiodic tasks are concerned, the schedulability analysis under SS is not simple because, in general, the server capacity can be fragmented in a lot of small pieces of different size, available at different times according to the replenishment rule. As a consequence, calculating the finishing time of an aperiodic request requires to keep track of all the replenishments that will occur until the task deadline.

5.7 Slack Stealing

The *Slack-Stealing* algorithm is another aperiodic service technique, proposed by Lehoczky and Ramos-Thuel in [LRT92], which offers substantial improvements in response time over the previous service methods (PE, DS, and SS). Unlike these methods, the Slack-Stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the *Slack Stealer*, which attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. This is equivalent to stealing slack from the periodic tasks. We recall that, if $c_i(t)$ is the remaining computation time at time t, the slack of a task τ_i is

$$slack_i(t) = d_i - t - c_i(t).$$

The main idea behind slack stealing is that, typically, there is no benefit in early completion of the periodic tasks. Hence, when an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks and uses it to execute aperiodic requests as soon as possible. If no aperiodic requests are pending, periodic tasks are normally scheduled by RM. Similar algorithms based on slack stealing have been proposed by other authors [RTL93, DTB93, TLS95].

Figure 5.20 shows the behavior of the Slack Stealer on a set of two periodic tasks, τ_1 and τ_2, with periods $T_1 = 4$, $T_2 = 5$ and execution times $C_1 = 1$, $C_2 = 2$. In particular, Fig. 5.20a shows the schedule produced by RM when no aperiodic tasks are processed, whereas Fig. 5.20b illustrates the case in which an aperiodic request of three units arrives at time $t = 8$ and receives immediate service. In this case, a slack of three units is obtained by delaying the third instance of τ_1 and τ_2.

Notice that, in the example of Fig. 5.20, no other server algorithms (PS, DS, PE, and SS) can schedule the aperiodic requests at the highest priority and still guarantee the periodic tasks. For example, since $U_1 = 1/4$ and $U_2 = 2/5$, the P factor for the task set is $P = 7/4$; hence, the maximum server utilization, according to Eq. (5.24), is

$$U_{SS}^{max} = \frac{2}{P} - 1 = \frac{1}{7} \simeq 0.14.$$

This means that, even with $C_s = 1$, the shortest server period that can be set with this utilization factor is $T_s = \lceil C_s/U_s \rceil = 7$, which is greater than both task periods. Thus, the execution of the server would be equivalent to a background service, and the aperiodic request would be completed at time 15.

5.7.1 Schedulability Analysis

In order to schedule an aperiodic request $J_a(r_a, C_a)$ according to the Slack-Stealing algorithm, we need to determine the earliest time t such that at least C_a units of slack

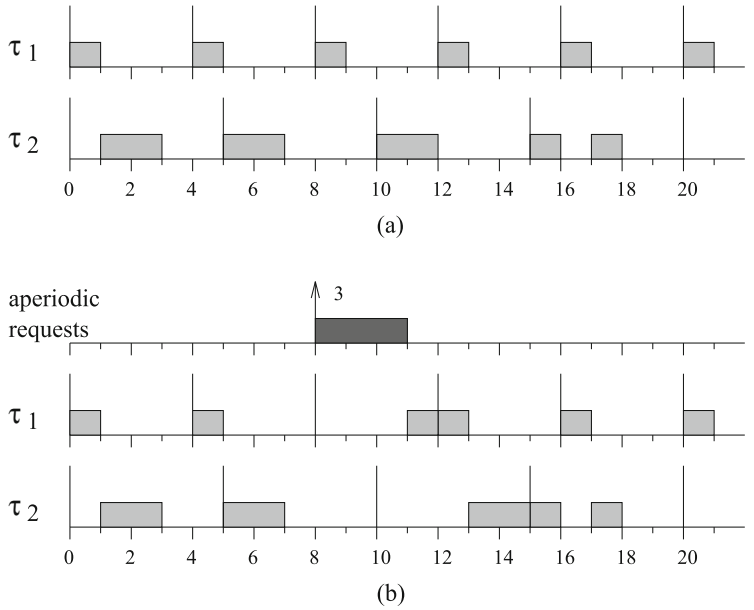


Fig. 5.20 Example of Slack Stealer behavior: (**a**) when no aperiodic requests are pending; (**b**) when an aperiodic request of three units arrives at time $t = 8$

are available in $[r_a, t]$. The computation of the slack is carried out through the use of a *slack function* $A(s, t)$, which returns the maximum amount of computation time that can be assigned to aperiodic requests in the interval $[s, t]$ without compromising the schedulability of periodic tasks.

Figure 5.21 shows the slack function at time $s = 0$ for the periodic task set considered in the previous example. For a given s, $A(s, t)$ is a nondecreasing step function defined over the hyperperiod, with jump points corresponding to the beginning of the intervals where the slack is available. As s varies, the slack function needs to be recomputed, and this requires a relatively large amount of calculation, especially for long hyperperiods. Figure 5.22 shows how the slack function $A(s, t)$ changes at time $s = 6$ for the same periodic task set.

According to the original algorithm proposed by Lehoczky and Ramos-Thuel [LRT92], the slack function at time $s = 0$ is precomputed and stored in a table. During runtime, the actual function $A(s, t)$ is then computed by updating $A(0, t)$ based on the periodic execution time, the aperiodic service time, and the idle time. The complexity for computing the current slack from the table is $O(n)$, where n is the number of periodic tasks; however, depending on the periods of the tasks, the size of the table can be too large for practical implementations.

A *dynamic* method of computing slack has been proposed by Davis, Tindell, and Burns in [DTB93]. According to this algorithm, the available slack is computed whenever an aperiodic requests enters the system. This method is more complex

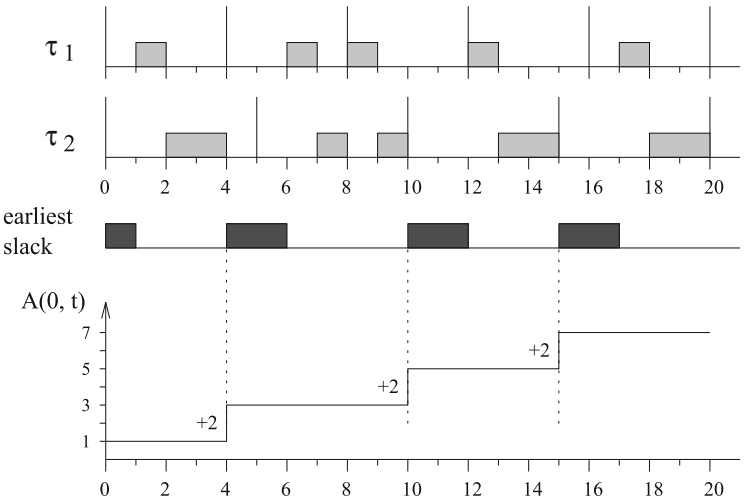


Fig. 5.21 Slack function at time $s = 0$ for the periodic task set considered in the previous example

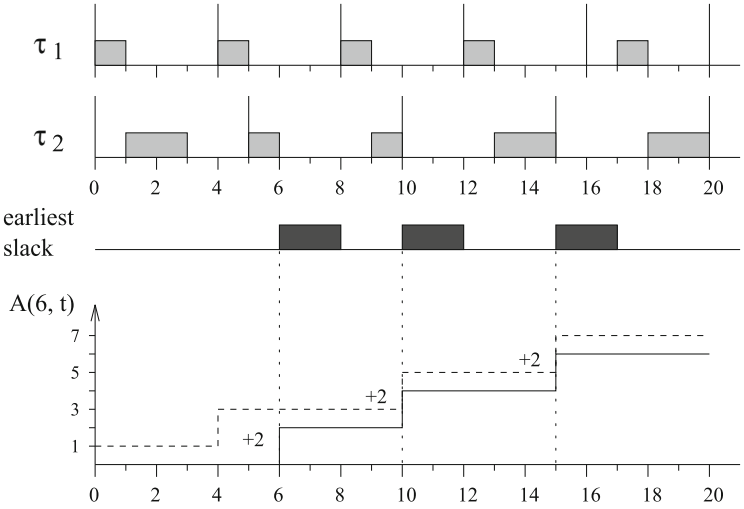


Fig. 5.22 Slack function at time $s = 6$ for the periodic task set considered in the previous example

than the previous *static* approach, but it requires much less memory and allows handling of periodic tasks with release jitter or synchronization requirements. Finally, a more efficient algorithm for computing the slack function has been proposed by Tia, Liu, and Shankar in [TLS95].

The Slack-Stealing algorithm has also been extended by Ramos-Thuel and Lehoczky [RTL93] to guarantee firm aperiodic tasks.

5.8 Non-existence of Optimal Servers

The Slack Stealer always advances all available slack as much as possible and uses it to execute the pending aperiodic tasks. For this reason, it originally was considered an optimal algorithm, that is, capable of minimizing the response time of every aperiodic request. Unfortunately, the Slack Stealer is not optimal because to minimize the response time of an aperiodic request, it is sometimes necessary to schedule it at a later time even if slack is available at the current time. Indeed, Tia, Liu, and Shankar [TLS95] proved that, if periodic tasks are scheduled using a fixed-priority assignment, no algorithm can minimize the response time of every aperiodic request and still guarantee the schedulability of the periodic tasks.

Theorem 5.2 (Tia-Liu-Shankar) *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any valid algorithm that minimizes the response time of every soft aperiodic request.*

Proof Consider a set of three periodic tasks with $C_1 = C_2 = C_3 = 1$ and $T_1 = 3$, $T_2 = 4$ and $T_3 = 6$, whose priorities are assigned based on the RM algorithm. Figure 5.23a shows the schedule of these tasks when no aperiodic requests are processed.

Now consider the case in which an aperiodic request J_1, with $C_{a_1} = 1$, arrives at time $t = 2$. At this point, any algorithm has two choices:

1. Do not schedule J_1 at time $t = 2$. In this case, the response time of J_1 will be greater than 1, and, thus, it will not be the minimum.
2. Schedule J_1 at time $t = 2$. In this case, assume that another request J_2, with $C_{a_2} = 1$, arrives at time $t = 3$. Since no slack time is available in the interval $[3, 6]$, J_2 can start only at $t = 6$ and finish at $t = 7$. This situation is shown in Fig. 5.23b.

However, the response time of J_2 achieved in case 2 is not the minimum possible. In fact, if J_1 were scheduled at time $t = 3$, another unit of slack would have been available at time $t = 4$; thus, J_2 would have been completed at time $t = 5$. This situation is illustrated in Fig. 5.23c.

The above example shows that it is not possible for any algorithm to minimize the response times of J_1 and J_2 simultaneously. If J_1 is scheduled immediately, then J_2 will not be minimized. On the other hand, if J_1 is delayed to minimize J_2, then J_1 will suffer. Hence, there is no optimal algorithm that can minimize the response time of any aperiodic request. □

Notice that Theorem 5.2 applies both to clairvoyant and online algorithms since the example is applicable regardless of whether the algorithm has a priori knowledge of the aperiodic requests. The same example can be used to prove another important result on the minimization of the average response time.

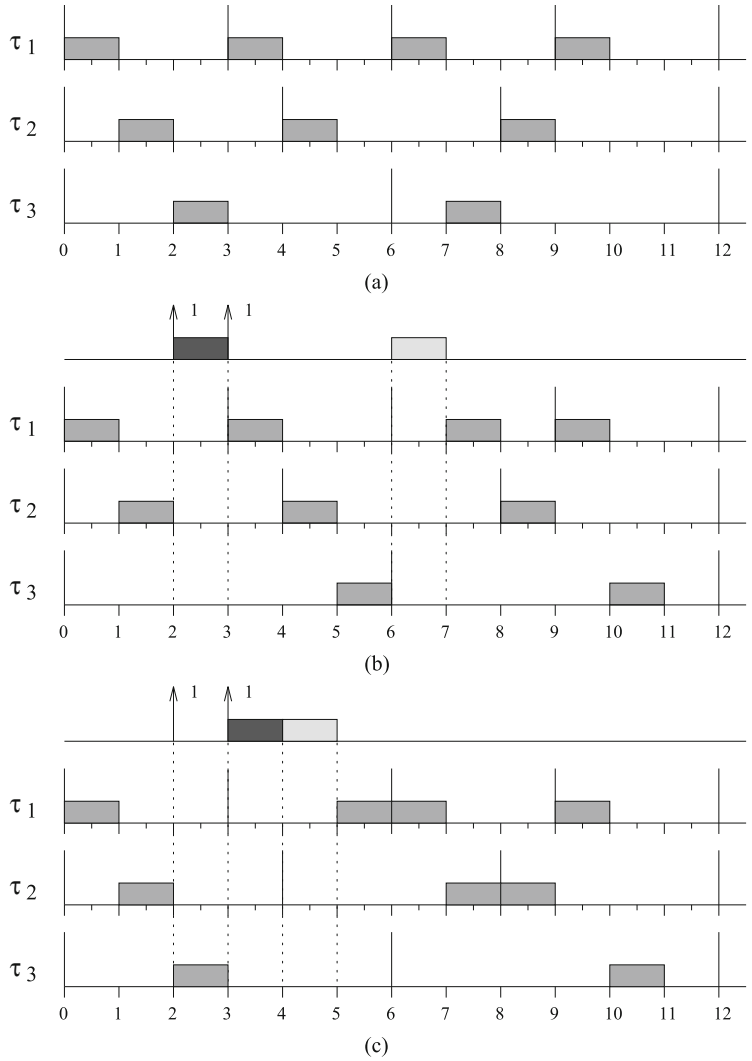


Fig. 5.23 No algorithm can minimize the response time of every aperiodic request. If J_1 is minimized, J_2 is not (**b**). On the other hand, if J_2 is minimized, J_1 is not (**c**)

Theorem 5.3 (Tia-Liu-Shankar) *For any set of periodic tasks ordered on a given fixed-priority scheme and aperiodic requests ordered according to a given aperiodic queueing discipline, there does not exist any online valid algorithm that minimizes the average response time of the soft aperiodic requests.*

Proof From the example illustrated in Fig. 5.23, it is easy to see that if there is only request J_1 in each hyperperiod, then scheduling J_1 as soon as possible will yield the minimum average response time. On the other hand, if J_1 and J_2 are present in

each hyperperiod, then scheduling each aperiodic request as soon as possible will not yield the minimum average response time. This means that, without a priori knowledge of the aperiodic requests' arrival, an online algorithm will not know when to schedule the requests. □

5.9 Performance Evaluation

The performance of the various algorithms described in this chapter has been compared in terms of average response times on soft aperiodic tasks. Simulation experiments have been conducted using a set of 10 periodic tasks with periods ranging from 54 to 1200 units of time and utilization factor $U_p = 0.69$. The aperiodic load was varied across the unused processor bandwidth. The interarrival times for the aperiodic tasks were modeled using a Poisson arrival pattern with average interarrival time of 18 units of time, whereas the computation times of aperiodic requests were modeled using an exponential distribution. Periods for PS, DS, PE, and SS were set to handle aperiodic requests at the highest priority in the system (priority ties were broken in favor of aperiodic tasks). Finally, the server capacities were set to the maximum value for which the periodic tasks were schedulable.

In the plots shown in Fig. 5.24, the average aperiodic response time of each algorithm is presented relative to the response time of background aperiodic service. This means that a value of 1.0 in the graph is equivalent to the average response time of background service, while an improvement over background service corresponds to a value less than 1.0. The lower the response time curve lies on the graph, the better the algorithm is for improving aperiodic responsiveness.

As can be seen from the graphs, DS, PE, and SS provide a substantial reduction in the average aperiodic response time compared to background and polling service. In particular, a better performance is achieved with short and frequent requests. This can be explained by considering that, in most of the cases, short tasks do not use the whole server capacity and can finish within the current server period. On the other hand, long tasks protract their completion because they consume the whole server capacity and have to wait for replenishments.

Notice that average response times achieved by SS are slightly higher than those obtained by DS and PE. This is mainly due to the different replenishment rule used by the algorithms. In DS and PE, the capacity is always replenished at its full value at the beginning of every server period, while in SS it is replenished T_s units of time after consumption. Thus, in the average, when the capacity is exhausted, waiting for replenishment in SS is longer than waiting in DS or in PE.

Figure 5.25 shows the performance of the Slack-Stealing algorithm with respect to background service, Polling, and SS. The performance of DS and PE is not shown because it is very similar to the one of SS. Unlike the previous figure, in this graph the average response times are not reported relative to background, but are directly expressed in time units. As we can see, the Slack-Stealing algorithm outperforms

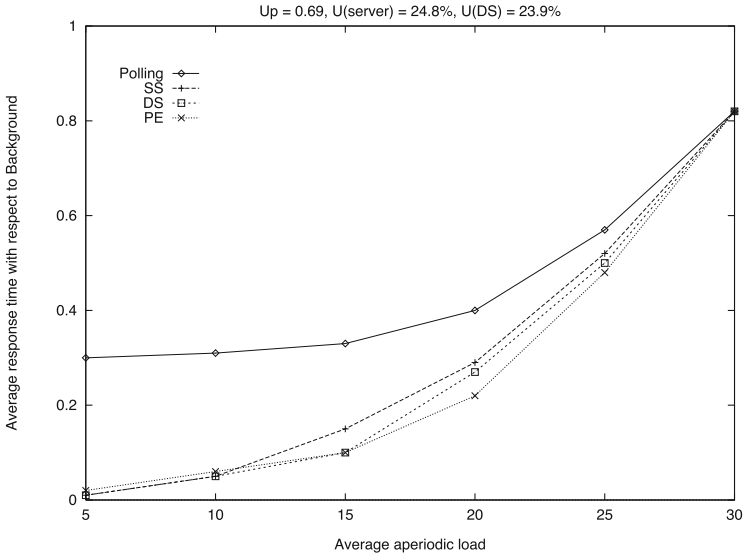


Fig. 5.24 Performance results of PS, DS, PE, and SS

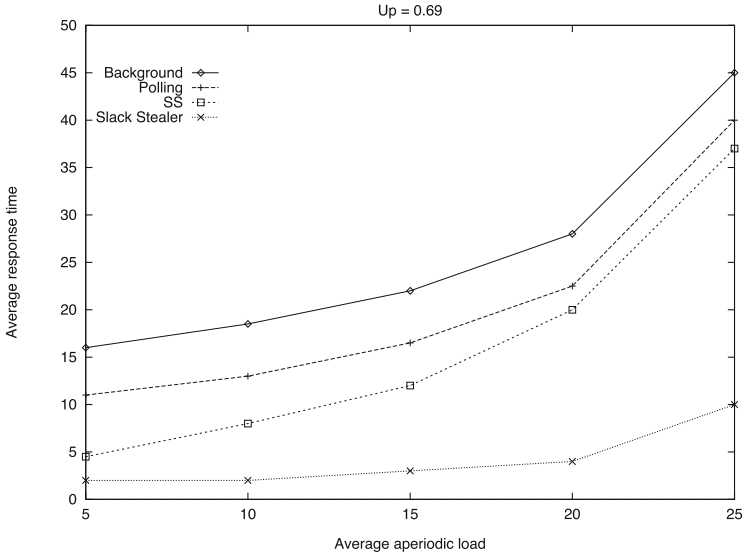


Fig. 5.25 Performance of the Slack Stealer with respect to background, PS, and SS

all the other scheduling algorithms over the entire range of aperiodic load. However, the largest performance gain of the Slack Stealer over the other algorithms occurs

at high aperiodic loads, when the system reaches the upper limit as imposed by the total resource utilization.

Other simulation results can be found in [LSS87] for Polling, PE, and DS, in [SSL89] for SS, and in [LRT92] for the Slack-Stealing algorithm.

5.10 Summary

The algorithms presented in this chapter can be compared not only in terms of performance but also in terms of computational complexity, memory requirement, and implementation complexity. In order to select the most appropriate service method for handling soft aperiodic requests in a hard real-time environment, all these factors should be considered. Figure 5.26 provides a qualitative evaluation of the algorithms presented in this chapter.

	performance	computational complexity	memory requirement	implementation complexity
Background Service				
Polling Server				
Deferrable Server				
Priority Exchange				
Sporadic Server				
Slack Stealer				

Fig. 5.26 Evaluation summary of fixed-priority servers

Exercises

5.1 Compute the best parameters that can be assigned to a Sporadic Server to guarantee the following periodic tasks under RM while enhancing aperiodic responsiveness as much as possible.

	C_i	T_i
τ_1	1	6
τ_2	2	7

5.2 Compute the best parameters that can be assigned to a Deferrable Server to guarantee the task set described in Exercise 5.5.1.

5.3 Consider two periodic tasks with computation times $C_1 = 1$, $C_2 = 2$ and periods $T_1 = 5$, $T_2 = 8$, handled by Rate Monotonic. Show the schedule produced by a Polling Server, having maximum utilization and intermediate priority, on the following aperiodic jobs.

	a_i	C_i
J_1	2	3
J_2	7	1
J_3	17	1

5.4 Solve the same scheduling problem described in Exercise 5.5.3, with a Sporadic Server having maximum utilization and intermediate priority.

5.5 Solve the same scheduling problem described in Exercise 5.5.3, with a Deferrable Server having maximum utilization and highest priority.

5.6 Using a Sporadic Server with capacity $C_s = 2$ and period $T_s = 5$, schedule the following tasks.

periodic tasks

	C_i	T_i
τ_1	1	4
τ_2	2	6

aperiodic tasks

	a_i	C_i
J_1	2	2
J_2	5	1
J_3	10	2

Chapter 6
Dynamic Priority Servers

6.1 Introduction

In this chapter[1] we discuss the problem of scheduling soft aperiodic tasks and hard periodic tasks under dynamic priority assignments . In particular, different service methods are introduced, whose objective is to reduce the average response time of aperiodic requests without compromising the schedulability of hard periodic tasks. Periodic tasks are scheduled by the Earliest Deadline First (EDF) algorithm.

With respect to fixed-priority assignments, dynamic scheduling algorithms are characterized by higher schedulability bounds, which allow the processor to be better utilized, increase the size of aperiodic servers, and enhance aperiodic responsiveness. Consider, for example, a set of two periodic tasks with the same utilization $U_1 = U_2 = 0.3$, so that $U_p = 0.6$. If priorities are assigned to periodic tasks based on RM and aperiodic requests are served by a Sporadic Server, the maximum server size that guarantees periodic schedulability is given by Eq. (5.24) and is $U_{SS}^{max} = 2/P - 1$, where $P = (U_1 + 1)(U_2 + 1) = 1.69$. Hence, we have $U_{SS}^{max} \simeq 0.18$. On the other hand, if periodic tasks are scheduled by EDF, the processor utilization bound goes up to 1.0, so the maximum server size can be increased up to $U_s = 1 - U_p = 0.4$.

For the sake of clarity, all properties of the algorithms presented in this chapter are proven under the following assumptions:

- All periodic tasks $\tau_i : i = 1, \ldots, n$ have hard deadlines and their schedulability must be guaranteed off-line.
- All aperiodic tasks $J_i : i = 1, \ldots, m$ do not have deadlines and must be scheduled as soon as possible, but without jeopardizing the schedulability of the periodic tasks.

[1] Part of this chapter is taken from the paper "Scheduling Aperiodic Tasks in Dynamic Priority Systems" by M. Spuri and G. Buttazzo, published in *Real-Time Systems*, 10(2), March 1996.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_6

- Each periodic task τ_i has a period T_i, a computation time C_i, and a relative deadline D_i equal to its period.
- All periodic tasks are simultaneously activated at time $t = 0$.
- Each aperiodic task has a known computation time but an unknown arrival time.

Some of the assumptions listed above can easily be relaxed to handle periodic tasks with arbitrary phasing and relative deadlines different from their periods. Shared resources can also be included in the model assuming an access protocol like the Stack Resource Policy [Bak91]. In this case, the schedulability analysis has to be consequently modified to take into account the blocking factors due to the mutually exclusive access to resources. For some algorithms, the possibility of handling firm aperiodic tasks is also discussed.

The rest of the chapter is organized as follows. In the next two sections, we discuss how two fixed-priority service algorithms—namely, the Priority Exchange and the Sporadic Server algorithms—can be extended to work under the EDF priority assignment. Then, we introduce three new aperiodic service algorithms, based on dynamic deadline assignments, that greatly improve the performance of the previous fixed-priority extensions. One of these algorithms, the EDL server, is shown to be optimal, in the sense that it minimizes the average response time of aperiodic requests.

6.2 Dynamic Priority Exchange Server

The *Dynamic Priority Exchange* (DPE) server is an aperiodic service technique proposed by Spuri and Buttazzo in [SB94, SB96] that can be viewed as an extension of the Priority Exchange server [LSS87], adapted to work with a deadline-based scheduling algorithm. The main idea of the algorithm is to let the server trade its runtime with the runtime of lower-priority periodic tasks (under EDF this means a longer deadline) in case there are no aperiodic requests pending. In this way, the server runtime 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.

The algorithm is defined as follows.

- The DPE server has a specified period T_s and a capacity C_s.
- At the beginning of each period, the server's *aperiodic* capacity is set to C_S^d, where d is the deadline of the current server period.
- Each deadline d associated with the instances (completed or not) of the ith periodic task has an aperiodic capacity, $C_{S_i}^d$, initially set to 0.
- Aperiodic capacities (those greater than 0) receive priorities according to their deadlines and the EDF algorithm, like all the periodic task instances (ties are broken in favor of capacities, i.e., aperiodic requests).
- Whenever the highest-priority entity in the system is an aperiodic capacity of C units of time the following happens:

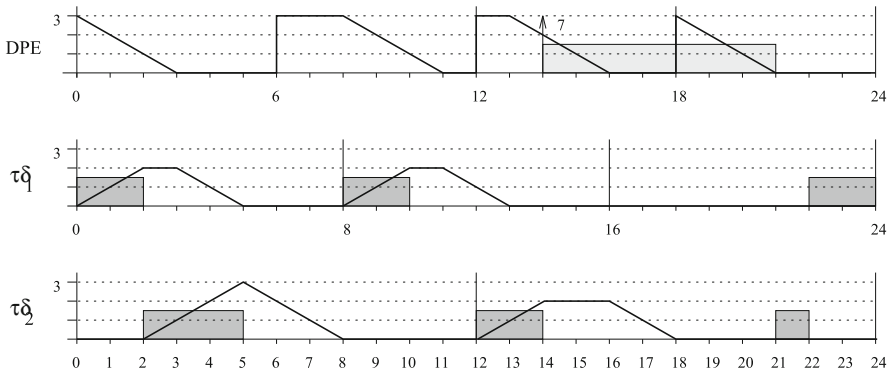


Fig. 6.1 Dynamic Priority Exchange server example

- if there are aperiodic requests in the system, these are served until they complete or the capacity is exhausted (each request consumes a capacity equal to its execution time);
- if there are no aperiodic requests pending, the periodic task having the shortest deadline is executed; a capacity equal to the length of the execution is added to the aperiodic capacity of the task deadline and is subtracted from C (i.e., the deadlines of the highest-priority capacity and the periodic task are exchanged);
- if neither aperiodic requests nor periodic task instances are pending, there is an idle time, and the capacity C is consumed until, at most, it is exhausted.

An example of schedule produced by the DPE algorithm is illustrated in Fig. 6.1. Two periodic tasks, τ_1 and τ_2, with periods $T_1 = 8$ and $T_2 = 12$ and worst-case execution times $C_1 = 2$ and $C_2 = 3$, and a DPE server with period $T_s = 6$ and capacity $C_s = 3$, are present in the system.

At time $t = 0$, the aperiodic capacities $C_{S_1}^8$ (with deadline 8) and $C_{S_2}^{12}$ (with deadline 12) are set to 0, while the server capacity (with deadline 6) is set to $C_s = C_S^6 = 3$. Since no aperiodic requests are pending, the two first periodic instances of τ_1 and τ_2 are executed, and C_s is consumed in the first three units of time. In the same interval, two units of time are accumulated in $C_{S_1}^8$ and one unit in $C_{S_2}^{12}$.

At time $t = 3$, $C_{S_1}^8$ is the highest-priority entity in the system. Again, since no aperiodic requests are pending, τ_2 keeps executing, and the two units of $C_{S_1}^8$ are consumed and accumulated in $C_{S_2}^{12}$. In the following three units of time, the processor is idle, and $C_{S_2}^{12}$ is completely consumed. Note that at time $t = 6$, the server capacity $C_s = C_S^{12}$ is set at value 3 and is preserved until time $t = 8$, when it becomes the highest-priority entity in the system (ties among aperiodic capacities are assumed to be broken in a FIFO order). At time $t = 8$, two units of C_S^{12} are exchanged with $C_{S_1}^{16}$, while the third unit of the server is consumed since the processor is idle.

At time $t = 14$, an aperiodic request, J_1, of seven units of time enters the system. Since $C_S^{18} = 2$, the first two units of J_1 are served with deadline 18, while the next two units are served with deadline 24, using the capacity $C_{S_2}^{24}$. Finally, the last three units are also served with deadline 24 because at time $t = 18$ the server capacity C_S^{24} is set to 3.

6.2.1 Schedulability Analysis

The schedulability condition for a set of periodic tasks scheduled together with a DPE server is now analyzed. Intuitively, the server behaves like any other periodic task. The difference is that it can trade its runtime with the runtime of lower-priority tasks. When a certain amount of time is traded, one or more lower-priority tasks are run at a higher-priority level, and their lower-priority time is preserved for possible aperiodic requests. This runtime exchange, however, does not affect schedulability; thus, the periodic task set can be guaranteed using the classical Liu and Layland condition

$$U_p + U_s \leq 1,$$

where U_p is the utilization factor of the periodic tasks and U_s is the utilization factor of the DPE server.

In order to prove this result, given a schedule σ produced using the DPE algorithm, consider a schedule σ' built in the following way:

- Replace the DPE server with a periodic task τ_s with period T_s and worst-case execution time C_s, so that in σ' τ_s executes whenever the server capacity is consumed in σ.
- The execution of periodic instances during deadline exchanges is postponed until the capacity decreases.
- All other executions of periodic instances are left as in σ.

Note that, from the definition of the DPE algorithm, at any time, at most one aperiodic capacity decreases in σ, so σ' is well defined. Also observe that, in each feasible schedule produced by the DPE algorithm, all the aperiodic capacities are exhausted before their respective deadlines.

Figure 6.2 shows the schedule σ' obtained from the schedule σ of Fig. 6.1. Note that all the periodic executions corresponding to increasing aperiodic capacities have been moved to the corresponding intervals in which the same capacities decrease. Also note that the schedule σ' does not depend on the aperiodic requests but depends only on the characteristics of the server and on the periodic task set. Based on this observation, the following theorem can be proved.

Theorem 6.1 (Spuri-Buttazzo) *Given a set of periodic tasks with processor utilization U_p and a DPE server with processor utilization U_s, the whole set is*

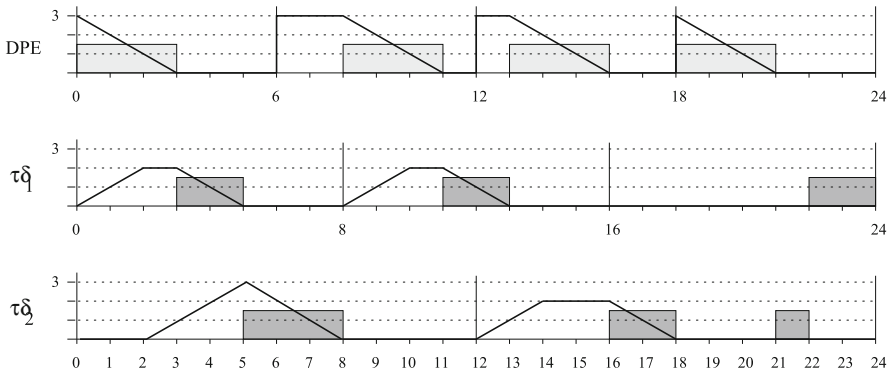


Fig. 6.2 DPE server schedulability

schedulable by EDF if and only if

$$U_p + U_s \leq 1.$$

Proof For any aperiodic load, all the schedules produced by the DPE algorithm have a unique corresponding EDF schedule σ', built according to the definition given above. Moreover, the task set in σ' is periodic with a processor utilization $U = U_p + U_s$. Hence, σ' is feasible if and only if $U_p + U_s \leq 1$. Now we show that σ is feasible if and only if σ' is feasible.

Observe that in each schedule σ, the completion time of a periodic instance is always less than or equal to the completion time of the corresponding instance in the schedule σ'. Hence, if σ' is feasible, then also σ is feasible; that is, the periodic task set is schedulable with the DPE algorithm. Vice versa, observing that σ' is a particular schedule produced by the DPE algorithm when there are enough aperiodic requests, if σ is feasible, then σ' will also be feasible; hence, the theorem holds. □

6.2.2 Reclaiming Spare Time

In hard real-time systems, the guarantee test of critical tasks is done by performing a worst-case schedulability analysis, that is, assuming the maximum execution time for all task instances. However, when such a peak load is not reached because the actual execution times are less than the worst-case values, it is not always obvious how to reclaim the spare time efficiently.

Using a DPE server, the spare time unused by periodic tasks can be easily reclaimed for servicing aperiodic requests. Whenever a periodic task completes, it is sufficient to add its spare time to the corresponding aperiodic capacity. An example of reclaiming mechanism is shown in Fig. 6.3.

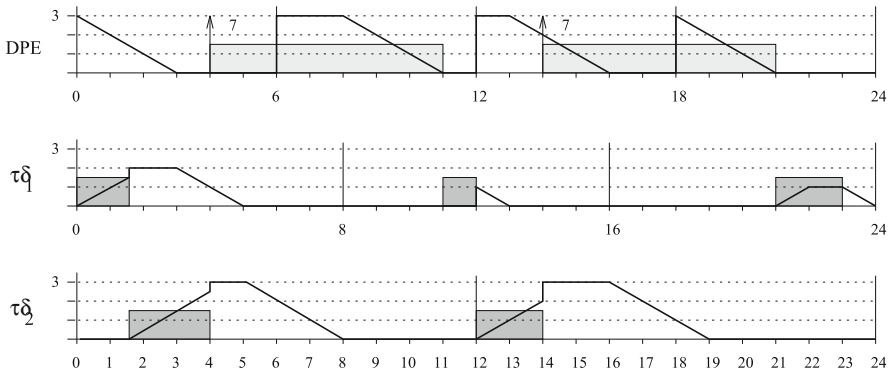


Fig. 6.3 DPE server resource reclaiming

As it can be seen from the capacity plot, at the completion time of the first two periodic instances, the corresponding aperiodic capacities ($C_{S_1}^{8}$ and $C_{S_2}^{12}$) are incremented by an amount equal to the spare time saved. Thanks to this reclaiming mechanism, the first aperiodic request can receive immediate service for all the seven units of time required, completing at time $t = 11$. Without reclaiming, the request would complete at time $t = 12$.

Note that reclaiming the spare time of periodic tasks as aperiodic capacities does not affect the schedulability of the system. In fact, any spare time is already "allocated" to a priority level corresponding to its deadline when the task set has been guaranteed. Hence, the spare time can be used safely if requested with the same deadline.

6.3 Dynamic Sporadic Server

The *Dynamic Sporadic Server*[2] (DSS) is an aperiodic service strategy proposed by Spuri and Buttazzo [SB94, SB96] that extends the Sporadic Server [SSL89] to work under a dynamic EDF scheduler. Similarly to other servers, DSS is characterized by a period T_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 main difference between the classical SS and its dynamic version consists in the way the priority is assigned to the server. Whereas SS has a fixed priority

[2] A similar algorithm called Deadline Sporadic Server has been independently developed by Ghazalie and Baker in [GB95].

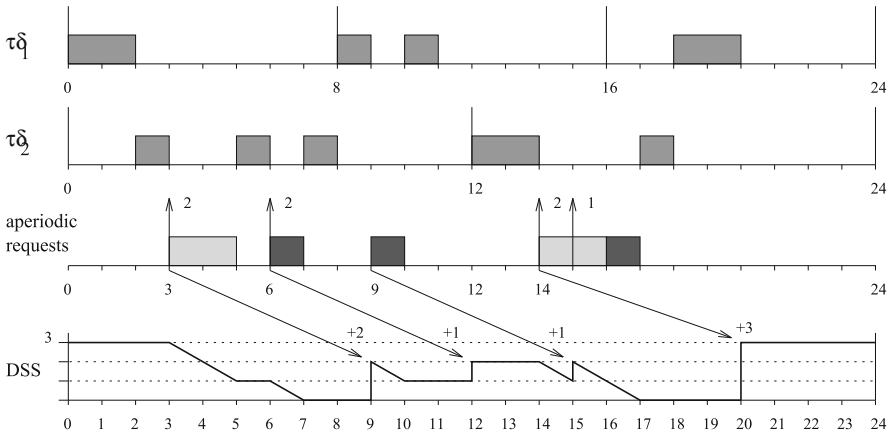


Fig. 6.4 Dynamic Sporadic Server example

chosen according to the RM algorithm (i.e., according to its period T_s), DSS has a dynamic priority assigned through a suitable deadline. The deadline assignment and the capacity replenishment are defined by the following rules:

- When the server is created, its capacity C_s is initialized at its maximum value.
- The next replenishment time RT and the current server deadline d_s are set as soon as $C_s > 0$, and there is an aperiodic request pending. If t_A is such a time, then $RT = d_s = t_A + T_s$.
- The replenishment amount RA to be done at time RT is computed when the last aperiodic request is completed or C_s has been exhausted. If t_I is such a time, then of RA is set equal to the capacity consumed within the interval $[t_A, t_I]$.

Figure 6.4 illustrates an EDF schedule obtained on a task set consisting of two periodic tasks with periods $T_1 = 8$, $T_2 = 12$ and execution times $C_1 = 2$, $C_2 = 3$, and a DSS with period $T_s = 6$ and capacity $C_s = 3$.

At time $t = 0$, the server capacity is initialized at its full value $C_s = 3$. Since there are no aperiodic requests pending, the processor is assigned to task τ_1, which has the earliest deadline. At time $t = 3$, an aperiodic request with execution time 2 arrives, and, since $C_s > 0$, the first replenishment time and the server deadline are set to $RT_1 = d_s = 3 + T_s = 9$. Being d_s the earliest deadline, DSS becomes the highest-priority task in the system, and the request is serviced until completion. At time $t = 5$, the request is completed, and no other aperiodic requests are pending; hence, a replenishment of two units of time is scheduled to occur at time $RT_1 = 9$.

At time $t = 6$, a second aperiodic requests arrives. Being $C_s > 0$, the next replenishment time and the new server deadline are set to $RT_2 = d_s = 6 + T_s = 12$. Again, the server becomes the highest-priority task in the system (we assume that ties among tasks are always resolved in favor of the server), and the request receives immediate service. This time, however, the capacity has only one unit of time available, and it gets exhausted at time $t = 7$. Consequently, a replenishment

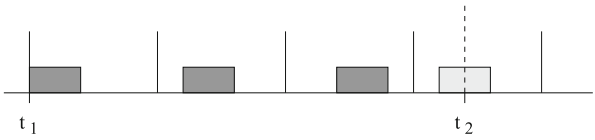


Fig. 6.5 Computational demand of a periodic task in $[t_1, t_2]$

of one unit of time is scheduled for $RT_2 = 12$, and the aperiodic request is delayed until $t = 9$, when C_s becomes again greater than zero. At time $t = 9$, the next replenishment time and the new deadline of the server are set to $RT_3 = d_s = 9 + T_s = 15$. As before, DSS becomes the highest-priority task; thus, the aperiodic request receives immediate service and finishes at time $t = 10$. A replenishment of one unit is then scheduled to occur at time $RT_3 = 15$.

Note that, as long as the server capacity is greater than zero, all pending aperiodic requests are executed with the same deadline. In Fig. 6.4 this happens at time $t = 14$, when the last two aperiodic requests are serviced with the same deadline $d_s = 20$.

6.3.1 Schedulability Analysis

To prove the schedulability bound for the Dynamic Sporadic Server, we first show that the server behaves like a periodic task with period T_s and execution time C_s.

Given a periodic task τ_i, we first note that, in any generic interval $[t_1, t_2]$ such that τ_i is released at t_1, the computation time scheduled by EDF with deadline less than or equal to t_2 is such that (see Fig. 6.5)

$$C_i(t_1, t_2) \leq \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i.$$

The following Lemma shows that the same property is true for DSS.

Lemma 6.1 *In each interval of time $[t_1, t_2]$, such that t_1 is the time at which DSS becomes ready (i.e., an aperiodic request arrives, and no other aperiodic requests are being served), the maximum aperiodic time executed by DSS in $[t_1, t_2]$ satisfies the following relation:*

$$C_{ape} \leq \left\lfloor \frac{t_2 - t_1}{T_s} \right\rfloor C_s.$$

Proof Since replenishments are always equal to the time consumed, the server capacity is at any time less than or equal to its initial value. Also, the replenishment policy establishes that the consumed capacity cannot be reclaimed before T_s units of time after the instant at which the server has become ready. This means that, from the time t_1 at which the server becomes ready, at most C_s time can be consumed in each subsequent interval of time of length T_s; hence, the thesis follows. □

Given that DSS behaves like a periodic task, the following theorem states that a full processor utilization is still achieved.

Theorem 6.2 (Spuri-Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and a Dynamic Sporadic Server with processor utilization U_s, the whole set is schedulable if and only if*

$$U_p + U_s \leq 1.$$

Proof *If.* Assume $U_p + U_s \leq 1$ and suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on ($t' < t$), only instances of tasks ready at t' or later and having deadlines less than or equal to t are run (the server may be one of these tasks). Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have $t - t' < C$. We also know that

$$\begin{aligned} C &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + C_{ape} \\ &\leq \sum_{i=1}^{n} \left\lfloor \frac{t-t'}{T_i} \right\rfloor C_i + \left\lfloor \frac{t-t'}{T_s} \right\rfloor C_s \\ &\leq \sum_{i=1}^{n} \frac{t-t'}{T_i} C_i + \frac{t-t'}{T_s} C_s \\ &\leq (t-t')(U_p + U_s). \end{aligned}$$

Thus, it follows that

$$U_p + U_s > 1,$$

a contradiction.

Only If. Since DSS behaves as a periodic task with period T_s and execution time C_s, the server utilization factor is $U_s = C_s/T_s$, and the total utilization factor of the processor is $U_p + U_s$. Hence, if the whole task set is schedulable, from the EDF schedulability bound [LL73], we can conclude that $U_p + U_s \leq 1$. □

6.4 Total Bandwidth Server

Looking at the characteristics of the Sporadic Server algorithm, it can be easily seen that, when the server has a long period, the execution of the aperiodic requests can be delayed significantly. This is due to the fact that when the period is long, the

server is always scheduled with a far deadline. And this is regardless of the aperiodic execution times.

There are two possible approaches to reduce the aperiodic response times. The first is, of course, to use a Sporadic Server with a shorter period. This solution, however, increases the runtime overhead of the algorithm because, to keep the server utilization constant, the capacity has to be reduced proportionally, but this causes more frequent replenishments and increases the number of context switches with the periodic tasks.

A second approach, less obvious, 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 the *Total Bandwidth Server* (TBS), a simple and efficient aperiodic service mechanism proposed by Spuri and Buttazzo in [SB94, SB96]. The name of the server comes from the fact that, each time an aperiodic request enters the system, the total bandwidth of the server is immediately assigned to it, whenever possible.

In particular, when the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s},$$

where C_k is the execution time of the request and U_s is the server utilization factor (i.e., its bandwidth). By definition $d_0 = 0$. Note that in the deadline assignment rule, the bandwidth allocated to previous aperiodic requests is considered through the deadline d_{k-1}.

Once the deadline is assigned, the request is inserted into the ready queue of the system and scheduled by EDF as any other periodic instance. As a consequence, the implementation overhead of this algorithm is practically negligible.

Figure 6.6 shows an example of EDF schedule produced by two periodic tasks with periods $T_1 = 6$, $T_2 = 8$ and execution times $C_1 = 3$, $C_2 = 2$, and a TBS with utilization $U_s = 1 - U_p = 0.25$. The first aperiodic request arrives at time $t = 3$ and is serviced with deadline $d_1 = r_1 + C_1/U_s = 3 + 1/0.25 = 7$. Being d_1 the earliest deadline in the system, the aperiodic request is executed immediately. Similarly, the second request, which arrives at time $t = 9$, receives a deadline $d_2 = r_2 + C_2/U_s = 17$, but it is not serviced immediately, since at time $t = 9$ there is an active periodic task, τ_2, with a shorter deadline, equal to 16. Finally, the third aperiodic request arrives at time $t = 14$ and gets a deadline $d_3 = \max(r_3, d_2) + C_3/U_s = 21$. It does not receive immediate service, since at time $t = 14$ task τ_1 is active and has an earlier deadline (18).

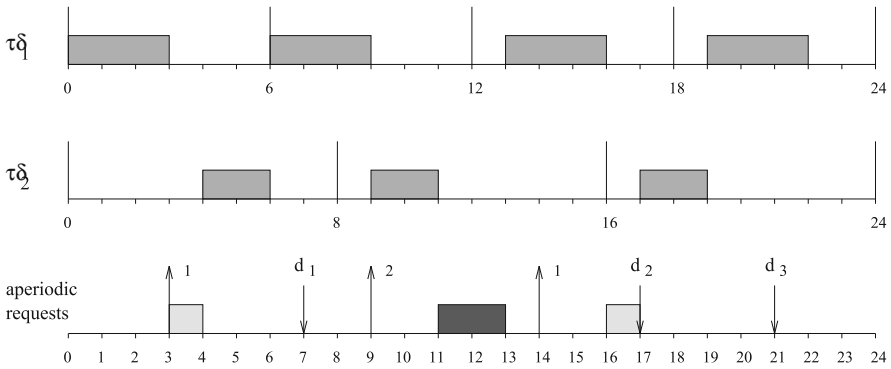


Fig. 6.6 Total Bandwidth Server example

6.4.1 Schedulability Analysis

In order to derive a schedulability test for a set of periodic tasks scheduled by EDF in the presence of a TBS, we first show that the aperiodic load executed by TBS cannot exceed the utilization factor U_s defined for the server.

Lemma 6.2 *In each interval of time* $[t_1, t_2]$*, if* C_{ape} *is the total execution time demanded by aperiodic requests arrived at* t_1 *or later and served with deadlines less than or equal to* t_2*, then*

$$C_{ape} \leq (t_2 - t_1)U_s.$$

Proof By definition

$$C_{ape} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k.$$

Given the deadline assignment rule of the TBS, there must exist two aperiodic requests with indexes k_1 and k_2 such that

$$\sum_{t_1 \leq r_k, d_k \leq t_2} C_k = \sum_{k=k_1}^{k_2} C_k.$$

It follows that

$$C_{ape} = \sum_{k=k_1}^{k_2} C_k$$

$$= \sum_{k=k_1}^{k_2} [d_k - \max(r_k, d_{k-1})] U_s$$
$$\leq [d_{k_2} - \max(r_{k_1}, d_{k_1-1})] U_s$$
$$\leq (t_2 - t_1) U_s.$$

□

The main result on TBS schedulability can now be proved.

Theorem 6.3 (Spuri-Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable by EDF if and only if*

$$U_p + U_s \leq 1.$$

Proof *If.* Assume $U_p + U_s \leq 1$ and suppose there is an overflow at time t. The overflow is preceded by a period of continuous utilization of the processor. Furthermore, from a certain point t' on ($t' < t$), only instances of tasks ready at t' or later and having deadlines less than or equal to t are run. Let C be the total execution time demanded by these instances. Since there is an overflow at time t, we must have

$$t - t' < C.$$

We also know that

$$C \leq \sum_{i=1}^{n} \left\lfloor \frac{t - t'}{T_i} \right\rfloor C_i + C_{ape}$$
$$\leq \sum_{i=1}^{n} \frac{t - t'}{T_i} C_i + (t - t') U_s$$
$$\leq (t - t')(U_p + U_s).$$

Thus, it follows that

$$U_p + U_s > 1,$$

a contradiction.

Only If. If an aperiodic request enters the system periodically, with period T_s and execution time $C_s = T_s U_s$, the server behaves exactly as a periodic task with period T_s and execution time C_s, and the total utilization factor of the processor is

$U_p + U_s$. Hence, if the whole task set is schedulable, from the EDF schedulability bound [LL73], we can conclude that $U_p + U_s \leq 1$. □

6.5 Earliest Deadline Late Server

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. For example, looking at the schedule in Fig. 6.6, we could argue that the second and the third aperiodic requests may be served as soon as they arrive, without compromising the schedulability of the system. This is possible because, when the requests arrive, the active periodic instances have enough slack time (laxity) 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 EDL server [SB94, SB96]. This aperiodic service algorithm can be viewed as a dynamic version of the Slack Stealing algorithm [LRT92].

The definition of the EDL server makes use of some results presented by Chetto and Chetto in [CC89]. In this paper, two complementary versions of EDF—namely, EDS and EDL—are proposed. Under EDS the active tasks are processed as soon as possible, whereas under EDL they are processed as late as possible. An important property of EDL is that in any interval $[0, t]$, it guarantees the maximum available idle time. In the original paper, this result is used to build an acceptance test for aperiodic tasks with hard deadlines, while here it is used to build an optimal server mechanism for soft aperiodic activities.

To simplify the description of the EDL server, $\omega_J^A(t)$ denotes the following *availability function*, defined for a scheduling algorithm A and a task set J:

$$\omega_J^A(t) = \begin{cases} 1 \text{ if the processor is idle at } t \\ 0 \text{ otherwise.} \end{cases}$$

The integral of $\omega_J^A(t)$ on an interval of time $[t_1, t_2]$ is denoted by $\Omega_J^A(t_1, t_2)$ and gives the total idle time in the specified interval. The function ω_J^{EDL} for the task set of Fig. 6.6 is depicted in Fig. 6.7.

The result of optimality addressed above is stated in Theorem 2 of [CC89], which we recall here.

Theorem 6.4 (Chetto and Chetto) *Let J be any aperiodic task set and A any preemptive scheduling algorithm. For any instant t,*

$$\Omega_J^{\text{EDL}}(0, t) \geq \Omega_J^A(0, t).$$

This result allows to develop an optimal server using the idle times of an EDL scheduler. In particular, given a periodic task set J, the function ω_J^A, which is

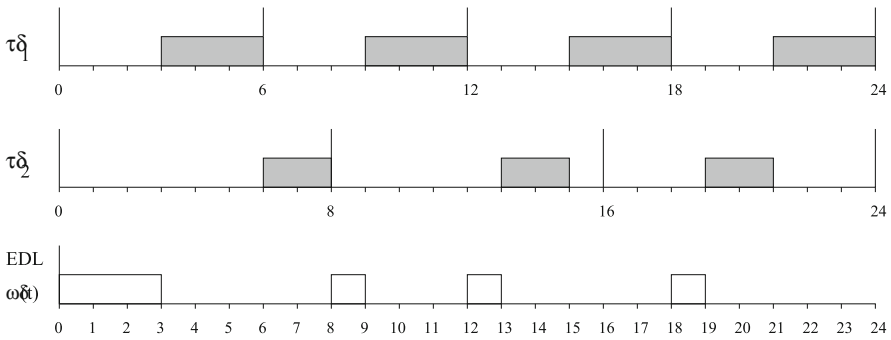


Fig. 6.7 Availability function under EDL

Table 6.1 Idle times under EDL

i	0	1	2	3
e_i	0	8	12	18
Δ_i	3	1	1	1

periodic with *hyperperiod* $H = \text{lcm}(T_1, \ldots, T_n)$, can be represented by means of two arrays. The first, $\mathcal{E} = (e_0, e_1, \ldots, e_p)$, represents the times at which idle times occur, while the second, $\mathcal{D} = (\Delta_0, \Delta_1, \ldots, \Delta_p)$, represents the lengths of these idle times. The two arrays for the example of Fig. 6.7 are shown in Table 6.1. Note that idle times occur only after the release of a periodic task instance.

The basic idea behind the EDL server is to use the idle times of an EDL schedule to execute aperiodic requests as soon as possible. When there are no aperiodic activities in the system, periodic tasks are scheduled according to the EDF algorithm. Whenever a new aperiodic request enters the system (and no previous aperiodic is still active), the idle times of an EDL scheduler applied to the current periodic task set are computed and then used to schedule the aperiodic requests pending. Figure 6.8 shows an example of the EDL service mechanism.

Here, an aperiodic request with an execution time of 4 units arrives at time $t =$. The idle times of an EDL schedule are recomputed using the current periodic tasks, as shown in Fig. 6.8a. The request is then scheduled according to the computed idle times (Fig. 6.8b). Notice that the server automatically allocates a bandwidth $1 - U_p$ to aperiodic requests. The response times achieved by this method are optimal, so they cannot be reduced further.

The procedure to compute the idle times of the EDL schedule is described in [CC89] and is not reported here. However, it is interesting to observe that not all the idle times have to be recomputed, but only those preceding the deadline of the current active periodic task with the longest deadline.

The worst-case complexity of the algorithm is $O(Nn)$, where n is the number of periodic tasks and N is the number of distinct periodic requests that occur in the hyperperiod. In the worst case, N can be very large and, hence, the algorithm may be of little practical interest. As for the Slack Stealer, the EDL server will be used to provide a lower bound to the aperiodic response times and to build a nearly optimal implementable algorithm, as described in the next section.

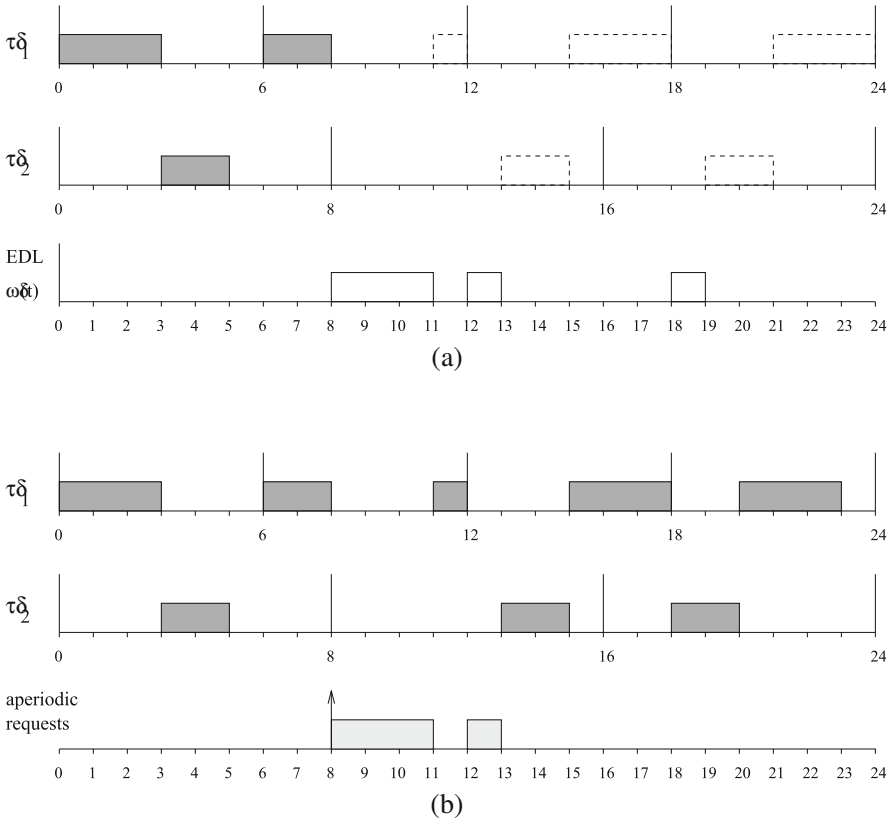


Fig. 6.8 (**a**) Idle times available at time $t = 8$ under EDL. (**b**) Schedule of the aperiodic request with the EDL server

6.5.1 EDL Server Properties

The schedulability analysis of the EDL server is quite straightforward. In fact, all aperiodic activities are executed using the idle times of a particular EDF schedule; thus, the feasibility of the periodic task set cannot be compromised. This is stated in the following theorem.

Theorem 6.5 (Spuri-Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and the corresponding EDL server (whose behavior strictly depends on the characteristics of the periodic task set), the whole set is schedulable if and only if*

$$U_p \leq 1.$$

Proof *If.* Since the condition ($U_p \leq 1$) is sufficient for guaranteeing the schedulability of a periodic task set under EDF, it is also sufficient under EDL, which is a particular implementation of EDF. The algorithm schedules the periodic tasks according to one or the other implementation, depending on the particular sequence of aperiodic requests. When aperiodic requests are pending, they are scheduled during precomputed idle times of the periodic tasks. In both cases, the timeliness of the periodic task set is unaffected, and no deadline is missed.

Only If. If a periodic task set is schedulable with an EDL server, it will be also schedulable without the EDL server, and hence ($U_p \leq 1$). □

We finally show that the EDL server is optimal; that is, the response times of the aperiodic requests under the EDL algorithm are the best achievable.

Lemma 6.3 *Let A be any online preemptive algorithm, τ a periodic task set, and J_i an aperiodic request. If $f^A_{\tau\cup\{J_i\}}(J_i)$ is the finishing time of J_i when $\tau \cup \{J_i\}$ is scheduled by A, then*

$$f^{EDL\ server}_{\tau\cup\{J_i\}}(J_i) \ \leq \ f^A_{\tau\cup\{J_i\}}(J_i).$$

Proof Suppose J_i arrives at time t, and let $\tau(t)$ be the set of the current active periodic instances (ready but not yet completed) and the future periodic instances. The new task J_i is scheduled together with the tasks in $\tau(t)$. In particular, consider the schedule σ of $\tau \cup \{J_i\}$ under A. Let A' be another algorithm that schedules the tasks in $\tau(t)$ at the same time as in σ, and σ' be the corresponding schedule. J_i is executed during some idle periods of σ'. Applying Theorem 6.4 with the origin of the time axis translated to t (this can be done since A is online), we know that for each $t' \geq t$

$$\Omega^{\mathrm{EDL}}_{\tau(t)}(t, t') \ \geq \ \Omega^{A'}_{\tau(t)}(t, t').$$

Recall now that, when there are aperiodic requests, the EDL server allocates the executions exactly during the idle times of EDL. Being

$$\Omega^{\mathrm{EDL}}_{\tau(t)}\left(t, f^{\mathrm{EDL\ server}}_{\tau\cup\{J_i\}}(J_i)\right) \ \geq \ \Omega^{A'}_{\tau(t)}\left(t, f^{\mathrm{EDL\ server}}_{\tau\cup\{J_i\}}(J_i)\right)$$

it follows that

$$f^{\mathrm{EDL}}_{\tau\cup\{J_i\}}(J_i) \ \leq \ f^A_{\tau\cup\{J_i\}}(J_i).$$

That is, under the EDL server, J_i is never completed later than under the A algorithm. □

6.6 Improved Priority Exchange Server

Although optimal, the EDL server has too much overhead to be considered practical. However, its main idea can be usefully adopted to develop a less complex algorithm that still maintains a nearly optimal behavior.

The heavy computation of the idle times can be avoided by using the mechanism of priority exchanges. With this mechanism, in fact, the system can easily keep track of the time advanced to periodic tasks and possibly reclaim it at the right priority level.

The idle times of the EDL algorithm can be precomputed 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 idle time advanced can be saved as aperiodic capacity at the priority levels of the periodic tasks executed.

The idea described above is used by the algorithm called *Improved Priority Exchange* (IPE) [SB94, SB96]. In particular, the DPE server is modified using the idle times of an EDL scheduler. There are two main advantages in 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. The IPE server is thus defined in the following way:

- The IPE server has an aperiodic capacity, initially set to 0.
- At each instant $t = e_i + kH$, with $0 \leq i \leq p$ and $k \geq 0$, a replenishment of Δ_i units of time is scheduled for the server capacity; that is, at time $t = e_0$ the server will receive Δ_0 units of time (the two arrays $\mathcal{E}$ and $\mathcal{D}$ have been defined in the previous section).
- The server priority is always the highest in the system, regardless of any other deadline.
- All other rules of IPE (aperiodic requests and periodic instances executions, exchange and consumption of capacities) are the same as for a DPE server.

The same task set of Fig. 6.8 is scheduled with an IPE server in Fig. 6.9.

Note that the server replenishments are set according to the function ω_τ^{EDL}, illustrated in Fig. 6.7.

When the aperiodic request arrives at time $t = 8$, one unit of time is immediately allocated to it by the server. However, other two units are available at the priority level corresponding to the deadline 12, due to previous deadline exchanges, and are allocated right after the first one. The last one is allocated later, at time $t = 12$, when the server receives a further unit of time. In this situation, the optimality of the response time is kept. As we will show later, there are only rare situations in which the optimal EDL server performs slightly better than IPE. That is, almost always IPE exhibits a nearly optimal behavior.

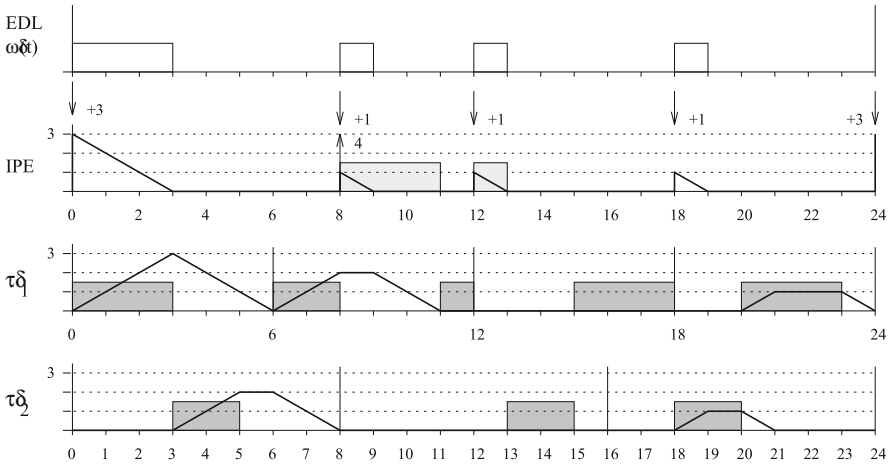


Fig. 6.9 Improved Priority Exchange server example

6.6.1 Schedulability Analysis

In order to analyze the schedulability of an IPE server, it is useful to define a transformation among schedules similar to that defined for the DPE server. In particular, given a schedule σ produced by the IPE algorithm, we build the schedule σ' in the following way:

- Each execution of periodic instances during deadline exchanges (i.e., increase in the corresponding aperiodic capacity) is postponed until the capacity decreases.
- All other executions of periodic instances are left as in σ.

In this case, the server is not substituted with another task. Again σ' is well defined and is invariant; that is, it does not depend on σ but only on the periodic task set. Moreover, σ' is the schedule produced by EDL applied to the periodic task set (compare Fig. 6.7 with Fig. 6.9). The optimal schedulability is stated by the following theorem.

Theorem 6.6 (Spuri-Buttazzo) *Given a set of n periodic tasks with processor utilization U_p and the corresponding IPE server (the parameters of the server depend on the periodic task set), the whole set is schedulable if and only if*

$$U_p \leq 1$$

(the server automatically allocates the bandwidth $1 - U_p$ to aperiodic requests).

Proof *If*. The condition is sufficient for the schedulability of the periodic task set under EDF, thus even under EDL, which is a particular implementation of EDF. Now, observe that in each schedule produced by the IPE algorithm, the completion

times of the periodic instances are never greater than the completion times of the corresponding instances in σ', which is the schedule of the periodic task set under EDL. That is, no periodic instance can miss its deadline. The thesis follows.

Only If. Trivial, since the condition is necessary even for the periodic task set only. □

6.6.2 Remarks

The reclaiming of unused periodic execution time can be done in the same way as for the DPE server. When a periodic task completes, its spare time is added to the corresponding aperiodic capacity. Again, this behavior does not affect the schedulability of the system. The reason is of course the same as for the DPE server.

To implement the IPE server, the two arrays $\mathcal{E}$ and $\mathcal{D}$ must be precomputed before the system is run. The replenishments of the server capacity are no longer periodic, but this does not change the complexity, which is comparable with that of DPE. What can change dramatically is the memory requirement. In fact, if the periods of periodic tasks are not harmonically related, we could have a huge *hyperperiod* $H = \text{lcm}(T_1, \ldots, T_n)$, which would require a great memory space to store the two arrays $\mathcal{E}$ and $\mathcal{D}$.

6.7 Improving TBS

The deadline assignment rule used by the TBS algorithm is a simple and efficient technique for servicing aperiodic requests in a hard real-time periodic environment. At the cost of a slightly higher complexity, such a rule can be modified to enhance aperiodic responsiveness. The key idea is to shorten the deadline assigned by the TBS as much as possible, still maintaining the periodic tasks schedulable [BS99].

If d_k is the deadline assigned to an aperiodic request by the TBS, a new deadline d'_k can be set at the estimated worst-case finishing time f_k of that request, scheduled by EDF with deadline d_k. The following lemma shows that setting the new deadline d'_k at the current estimated worst-case finishing time does not jeopardize schedulability.

Lemma 6.4 *Let σ be a feasible schedule of task set $\mathcal{T}$, in which an aperiodic job J_k is assigned a deadline d_k, and let f_k be the finishing time of J_k in σ. If d_k is substituted with $d'_k = f_k$, then the new schedule σ' produced by EDF is still feasible.*

Proof Since σ remains feasible after d_k is substituted with $d'_k = f_k$ and all other deadlines are unchanged, the optimality of EDF [Der74] guarantees that σ' is also feasible. □

The process of shortening the deadline can be applied recursively to each new deadline, until no further improvement is possible, given that the schedulability of the periodic task set must be preserved. If d_k^s is the deadline assigned to the aperiodic request J_k at step s and f_k^s is the corresponding finishing time in the current EDF schedule (achieved with d_k^s), the new deadline d_k^{s+1} is set at time f_k^s. At each step, the schedulability of the task set is guaranteed by Lemma 6.4.

The algorithm stops either when $d_k^s = d_k^{s-1}$ or after a maximum number of steps defined by the system designer for bounding the complexity. Notice that the exact evaluation of f_k^s would require the development of the entire schedule up to the finishing time of request J_k, scheduled with d_k^s. However, there is no need to evaluate the exact value of f_k^s to shorten the deadline. Rather, the following upper bound can be used:

$$\tilde{f}_k^s = t + C_k^a + I_p\left(t, d_k^s\right), \tag{6.1}$$

where t is the current time (corresponding to the release time r_k of request J_k or to the completion time of the previous request), C_k^a is the worst-case computation time required by J_k, and $I_p(t, d_k^s)$ is the interference on J_k due to the periodic instances in the interval $[t, d_k^s)$. $\tilde{f}_k^s$ is an upper bound for f_k^s because it identifies the time at which J_k and all the periodic instances with deadline less than d_k^s end to execute. Hence, $f_k^s \leq \tilde{f}_k^s$.

The periodic interference $I_p(t, d_k^s)$ in Eq. (6.1) can be expressed as the sum of two terms, $I_a(t, d_k^s)$ and $I_f(t, d_k^s)$, where $I_a(t, d_k^s)$ is the interference due to the currently active periodic instances with deadlines less than d_k^s and $I_f(t, d_k^s)$ is the future interference due to the periodic instances activated after time t with deadline before d_k^s. Hence,

$$I_a\left(t, d_k^s\right) = \sum_{\tau_i \ active,\ d_i < d_k^s} c_i(t) \tag{6.2}$$

and

$$I_f\left(t, d_k^s\right) = \sum_{i=1}^{n} \max\left(0, \left\lceil \frac{d_k^s - next\_r_i(t)}{T_i} \right\rceil - 1\right) C_i, \tag{6.3}$$

where $next\_r_i(t)$ identifies the time greater than t at which the next periodic instance of task τ_i will be activated. If periodic tasks are synchronously activated at time zero, then

$$next\_r_i(t) = \left\lceil \frac{t+1}{T_i} \right\rceil T_i. \tag{6.4}$$

In fact, when t is equal to the activation time of a periodic task τ_i (i.e., when t/T_i is an integer), the interference due to the activated job is already accounted in

I_a and should not be accounted in I_f. Using $(t+1)$ in Eq. (6.4), when t is equal to the activation time of a job of τ_i, $next\_r_i(t)$ correctly provides the activation time of the subsequent job.

Since I_a and I_f can be computed in $O(n)$, the overall complexity of the deadline assignment algorithm is $O(Nn)$, where N is the maximum number of steps performed by the algorithm to shorten the initial deadline assigned by the TB server. We now show that $\tilde{f}_k^s$ is the real worst-case finishing time if it coincides with the deadline d_k^s.

Lemma 6.5 *In any feasible schedule, $\tilde{f}_k^s = f_k^s$ only if $\tilde{f}_k^s = d_k^s$.*

Proof Assume that there exists a feasible schedule σ where $\tilde{f}_k^s = d_k^s$, but $\tilde{f}_k^s > f_k^s$. Since $\tilde{f}_k^s$ is the time at which J_k and all the periodic instances with deadline less than d_k^s end to execute, $\tilde{f}_k^s > f_k^s$ would imply that $\tilde{f}_k^s$ coincides with the end of a periodic instance having deadline less than $\tilde{f}_k^s = d_k^s$, meaning that this instance would miss its deadline. This is a contradiction; hence, the lemma follows. □

6.7.1 An Example

The following example illustrates the deadline approximation algorithm. The task set consists of two periodic tasks, τ_1 and τ_2, with periods 3 and 4, and computation times 1 and 2, respectively. A single aperiodic job J_k arrives at time $t = 2$, requiring two units of computation time. The periodic utilization factor is $U_p = 5/6$, leaving a bandwidth of $U_s = 1/6$ for the aperiodic tasks.

When the aperiodic request arrives at time $t = 2$, it receives a deadline $d_k^0 = r_k + C_k^a / U_s = 14$, according to the TBS algorithm. The schedule produced by EDF using this deadline assignment is shown in Fig. 6.10.

By applying Eqs. (6.2) and (6.3), we have

$$\begin{aligned} I_a(2, 14) = c_2(2) &= 1 \\ I_f(2, 14) = 3C_1 + 2C_2 &= 7, \end{aligned}$$

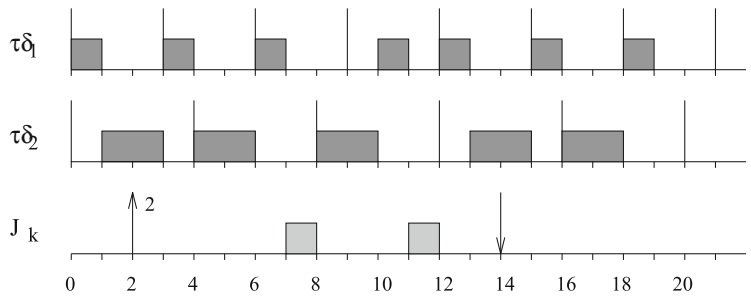


Fig. 6.10 Schedule produced by EDF with $d_k^0 = 14$

Table 6.2 Deadlines and finishing times computed by the algorithm

Step	d_k^s	f_k^s
0	14	12
1	12	9
2	9	8
3	8	6
4	6	5
5	5	5

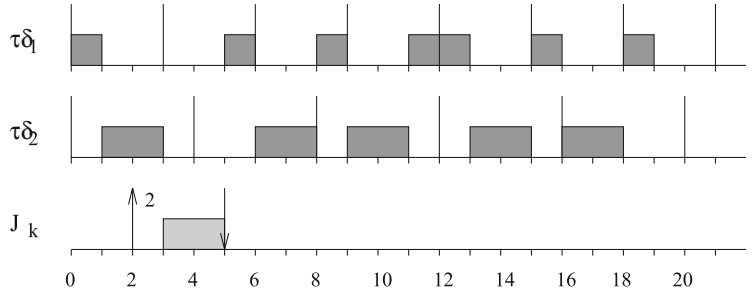


Fig. 6.11 Schedule produced by EDF with $d_k^* = 5$

and, by Eq. (6.1), we obtain

$$d_k^1 = \tilde{f}_k^0 = t + C_k^a + I_a + I_f = 12.$$

In this case, it can easily be verified that the aperiodic task actually terminates at $t = 12$. This happens because the periodic tasks do not leave any idle time to the aperiodic task, which is thus compelled to execute at the end. Table 6.2 shows the subsequent deadlines evaluated at each step of the algorithm. In this example, six steps are necessary to find the shortest deadline for the aperiodic request.

The schedule produced by EDF using the shortest deadline $d_k^* = d_k^5 = 5$ is shown in Fig. 6.11. Notice that at $t = 19$ the first idle time is reached, showing that the whole task set is schedulable.

6.7.2 Optimality

As far as the average case execution time of tasks is equal to the worst-case one, our deadline assignment method achieves optimality, yielding the minimum response time for each aperiodic task. Under this assumption, the following theorem holds.

Theorem 6.7 (Buttazzo-Sensini) *Let σ be a feasible schedule produced by EDF for a task set $\mathcal{T}$ and let f_k be the finishing time of an aperiodic request J_k, scheduled in σ with deadline d_k. If $f_k = d_k$, then $f_k = f_k^*$, where f_k^* is the minimum finishing time achievable by any other feasible schedule.*

Proof Assume $f_k = d_k$, and let r_0 be the earliest request such that interval $[r_0, d_k]$ is fully utilized by J_k and by tasks with deadline less than d_k. Hence, in σ, d_k represents the time at which J_k and all instances with deadline less than d_k end to execute.

We show that any schedule σ' in which J_k finishes at $f'_k < d_k$ is not feasible. In fact, since $[r_0, d_k]$ is fully utilized and $f'_k < d_k$, in σ' d_k must be the finishing time of some periodic instance[3] with deadline less than d_k. As a consequence, σ' is not feasible. Thus, the theorem follows. □

6.8 Performance Evaluation

The algorithms described in this chapter have been simulated on a synthetic workload in order to compare the average response times achieved on soft aperiodic activities. For completeness, a dynamic version of the Polling Server has also been compared with the other algorithms.

The plots shown in Fig. 6.12 have been obtained with a set of ten periodic tasks with periods ranging from 100 to 1000 units of time and utilization factor $U_p = 0.65$. The aperiodic load was varied across the range of processor utilization unused by the periodic tasks and in particular from 3% to 33%. The interarrival times for the aperiodic tasks were modeled using a Poisson arrival pattern, with average T_a, whereas the aperiodic computation times were modeled using an exponential distribution.

The processor utilization of the servers was set to all the utilization left by the periodic tasks, that is, $U_s = 1 - U_p$. The period of the periodic servers—namely Polling, DPE, and DSS—was set equal to the average aperiodic interarrival time (T_a), and, consequently, the capacity was set to $C_s = T_a U_s$.

In Fig. 6.12, the performance of the algorithms is shown as a function of the aperiodic load. The load was varied by changing the average aperiodic service time, while the average interarrival time was set at the value of $T_a = 100$. Note that the data plotted for each algorithm represent the ratio of the average aperiodic response time relative to the response time of background service. In this way, an average response time equivalent to background service has a value of 1.0 on the graph. A value less than 1.0 corresponds to an improvement in the average aperiodic response time over background service. The lower the response time curve lies on these graphs, the better the algorithm is for improving aperiodic responsiveness.

The EDL server is not reported in the graph since it has basically the same behavior as IPE for almost any load conditions. In particular, simulations showed that for small and medium periodic loads, the two algorithms do not have significant differences in their performance. However, even for a high periodic load, the

[3] Time d_k cannot be the finishing time of an aperiodic task, since we assume that aperiodic requests are served on a FCFS basis.

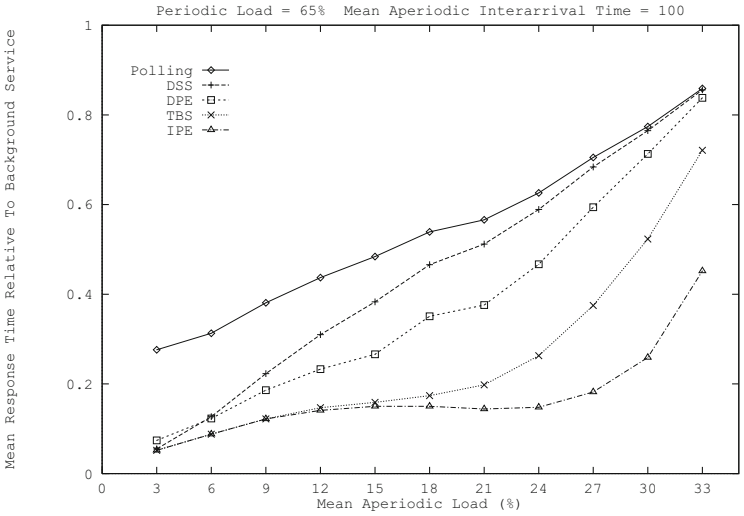


Fig. 6.12 Performance of dynamic server algorithms

difference is so small that it can be reasonably considered negligible for any practical application.

Although IPE and EDL have very similar performance, they differ significantly in their implementation complexity. As mentioned in previous sections, the EDL algorithm needs to recompute the server parameters quite frequently (namely, when an aperiodic request enters the system and all previous aperiodics have been completely serviced). This overhead can be too expensive in terms of cpu time to use the algorithm in practical applications. On the other hand, in the IPE algorithm, the parameters of the server can be computed off-line and used at runtime to replenish the server capacity.

As can be seen from the graph, the TBS and IPE algorithms can provide a significant reduction in average aperiodic response time compared to background or polling aperiodic service, whereas the performance of the DPE and DSS algorithms depends on the aperiodic load. For low aperiodic load, DPE and DSS perform as well as TBS and IPE, but as the aperiodic load increases, their performance tends to be similar to that one shown by the Polling Server.

Note that, in all graphs, TBS and IPE have about the same responsiveness when the aperiodic load is low, and they exhibit a slightly different behavior for heavy aperiodic loads.

All algorithms perform much better when the aperiodic load is generated by a large number of small tasks rather than a small number of long activities. Moreover, note that, as the interarrival time T_a increases, and the tasks' execution time becomes longer, the IPE algorithm shows its superiority with respect to the others, which tend to have about the same performance, instead.

The proposed algorithms have been compared with different periodic loads U_p as well. For very low periodic loads, all aperiodic service algorithms show a behavior similar to background service. As the periodic load increases, their performance improves substantially with respect to background service. In particular, DPE and DSS have a comparable performance, which tends to approach that of the Polling Server for high periodic loads. On the other hand, TBS and IPE outperform all other algorithms in all situations. The improvement is particularly significant with medium and high workloads. With a very high workload, TBS is no more able to achieve the same good performance of IPE, even though it is much better than the other algorithms. More extensive simulation results are reported in [SB94, SB96].

Simulations have also been conducted to test the performance of the different deadline assignment rules for the Total Bandwidth approach. In Fig. 6.13, TB* denotes the optimal algorithm, whereas TB(i) denotes the version of the algorithm which stops iteration after at most i steps from the TBS deadline. Thus, TB(0) coincides with the standard TBS algorithm. In order to show the improvement achieved by the algorithm for each deadline update, the performance of TB* is compared with the one of TB(0), TB(3), and TB(7), which were tested for different periodic and aperiodic loads. To provide a reference term, the response times for background service and for a M/M/1 model are also shown. The plots show the results obtained with a periodic load $U_p = 0.9$. The average response time is plotted

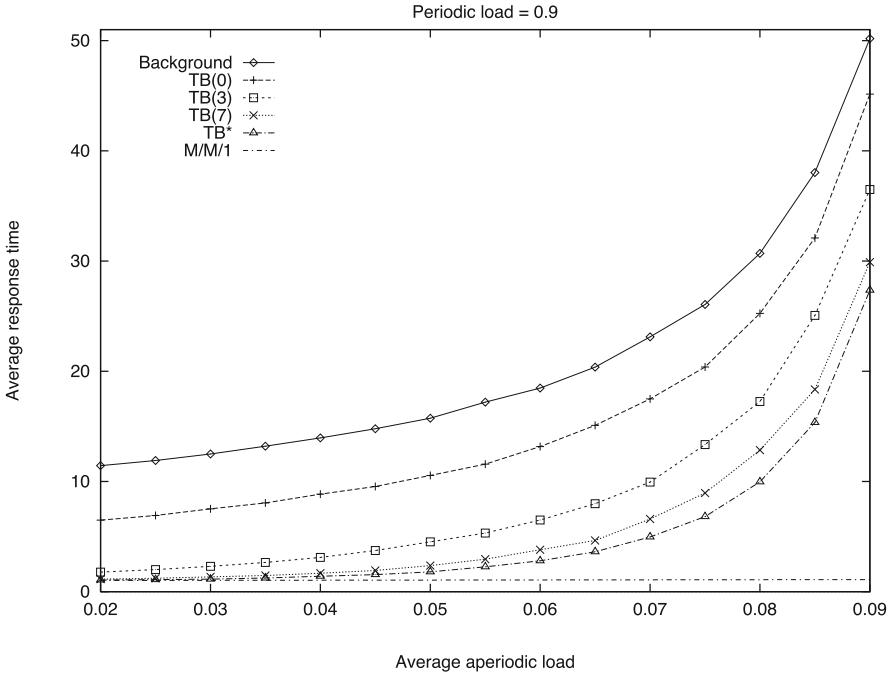


Fig. 6.13 Performance results for $U_p = 0.9$

with respect to the average task length. Thus, a value of 5 on the y-axis actually means an average response time five times longer than the task computation time.

6.9 The Constant Bandwidth Server

In this section, we present a novel service mechanism, called the Constant Bandwidth Server (CBS), which efficiently implements a bandwidth reservation strategy. As the DSS, the Constant Bandwidth Server guarantees that, if U_s is the fraction of processor time assigned to a server (i.e., its bandwidth), its contribution to the total utilization factor is no greater than U_s, even in the presence of overloads. Notice that this property is not valid for a TBS, whose actual contribution is limited to U_s only under the assumption that all the served jobs execute no more than the declared WCET. With respect to the DSS, however, the CBS shows a much better performance, comparable with the one achievable by a TBS.

The basic idea behind the CBS mechanism can be explained as follows: when a new job enters the system, it is assigned a suitable scheduling deadline (to keep its demand within the reserved bandwidth), and it is inserted in the EDF ready queue. If the job tries to execute more than expected, its deadline is postponed (i.e., its priority is decreased) to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work conserving algorithm, exploiting the available slack in an efficient (deadline-based) way, thus providing better responsiveness with respect to non-work-conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background, like [MST93, MST94a].

If a subset of tasks is handled by a single server, all the tasks in that subset will share the same bandwidth, so there is not isolation among them. Nevertheless, all the other tasks in the system are protected against overruns occurring in the subset.

In order not to miss any hard deadline, the deadline assignment rules adopted by the server must be carefully designed. The next section precisely defines the CBS algorithm and formally proves its correctness for any (know or unknown) execution request and arrival pattern.

6.9.1 Definition of CBS

The CBS can be defined as follows:

- A CBS is characterized by a budget c_s and by an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{s,k}$ is associated with the server. At the beginning $d_{s,0} = 0$.

- Each served job $J_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline $d_{s,k}$.
- Whenever a served job executes, the budget c_s is decreased by the same amount.
- When $c_s = 0$, the server budget is recharged at the maximum value Q_s, and a new server deadline is generated as $d_{s,k+1} = d_{s,k} + T_s$. Notice that there are no finite intervals of time in which the budget is equal to zero.
- A CBS is said to be active at time t if there are pending jobs (remember the budget c_s is always greater than 0), that is, if there exists a served job $J_{i,j}$ such that $r_{i,j} \leq t < f_{i,j}$. A CBS is said to be idle at time t if it is not active.
- When a job $J_{i,j}$ arrives and the server is active, the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline (e.g., FIFO).
- When a job $J_{i,j}$ arrives and the server is idle, if $c_s \geq (d_{s,k} - r_{i,j})U_s$ the server generates a new deadline $d_{s,k+1} = r_{i,j} + T_s$, and c_s is recharged at the maximum value Q_s; otherwise the job is served with the last server deadline $d_{s,k}$ using the current budget.
- When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.
- At any instant, a job is assigned the last deadline generated by the server.

6.9.2 Scheduling Example

Figure 6.14 illustrates an example in which a hard periodic task, τ_1, with computation time $C_1 = 4$ and period $T_1 = 7$, is scheduled together with a soft task, τ_2, served by a CBS having a budget $Q_s = 3$ and a period $T_s = 8$. The first job of τ_2 ($J_{2,1}$), requiring four units of execution time, arrives at time $r_1 = 3$, when the server is idle. Being $c_s \geq (d_0 - r_1)U_s$, the job is assigned a deadline $d_1 = r_1 + T_s = 11$ and c_s is recharged at $Q_s = 3$. At time $t = 7$, the budget is exhausted, so a new deadline $d_2 = d_1 + T_s = 19$ is generated, and c_s is replenished. Since the server

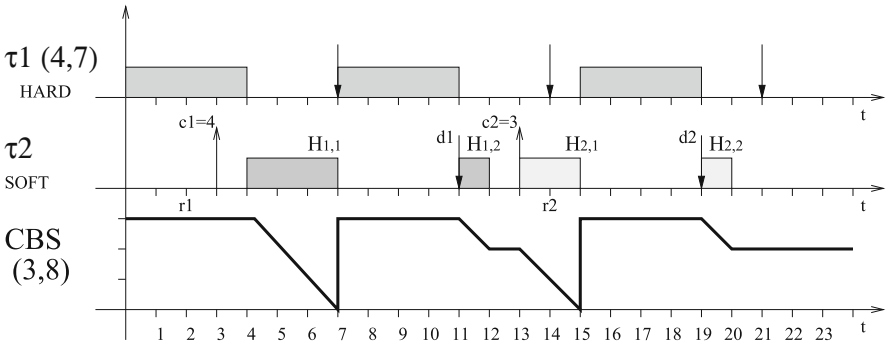


Fig. 6.14 An example of CBS scheduling

deadline is postponed, τ_1 becomes the task with the earliest deadline and executes until completion. Then, τ_2 resumes, and job $J_{2,1}$ (having deadline $d_2 = 19$) is finished at time $t = 12$, leaving a budget $c_s = 2$. The second job of task τ_2 arrives at time $r_2 = 13$ and requires three units of time. Since $c_s < (d_2 - r_2)U_s$, the last server deadline d_2 can be used to serve job $J_{2,2}$. At time $t = 15$, the server budget is exhausted, so a new server deadline $d_3 = d_2 + T_s = 27$ is generated, and c_s is replenished at Q_s. For this reason, τ_1 becomes the highest-priority task and executes until time $t = 19$, when job $J_{1,3}$ finishes and τ_2 can execute, finishing job $J_{2,2}$ at time $t = 20$ leaving a budget $c_s = 2$.

It is worth noting that under a CBS, a job J_j is assigned an absolute time-varying deadline d_j which can be postponed if the task requires more than the reserved bandwidth. Thus, each job J_j can be thought as consisting of a number of chunks $H_{j,k}$, each characterized by a release time $a_{j,k}$ and a fixed deadline $d_{j,k}$. An example of chunks produced by a CBS is shown in Fig. 6.14. To simplify the notation, we will indicate all the chunks generated by the server with an increasing index k (in the example of Fig. 6.14, $H_{1,1} = H_1$, $H_{1,2} = H_2$, $H_{2,1} = H_3$, and so on).

6.9.3 Formal Definition

In order to provide a formal definition of the CBS, let a_k and d_k be the release time and the deadline of the kth chunk generated by the server, and let c and n be the actual server budget and the number of pending requests in the server queue (including the request currently being served). These variables are initialized as follows:

$$d_0 = 0, \quad c = 0, \quad n = 0, \quad k = 0.$$

Using this notation, the server behavior can be described by the algorithm shown in Fig. 6.15.

6.9.4 CBS Properties

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting applications with highly variable computation times (e.g., continuous media applications). The most important one, the *isolation property*, is formally expressed by the following theorem and lemma. See the original work by Abeni and Buttazzo [AB98] for the proof.

Theorem 6.8 *The CPU utilization of a CBS S with parameters (Q_s, T_s) is $U_s = \frac{Q_s}{T_s}$, independently from the computation times and the arrival pattern of the served jobs.*

```
When job J_j arrives at time r_j
      enqueue the request in the server queue;
      n = n + 1;
      if (n == 1) /* (the server is idle) */
        if (r_j + (c / Q_s) * T_s >= d_k)
          /*---------------Rule 1---------------*/
          k = k + 1;
          a_k = r_j;
          d_k = a_k + T_s;
          c = Q_s;
        else
          /*---------------Rule 2---------------*/
          k = k + 1;
          a_k = r_j;
          d_k = d_{k-1};
          /* c remains unchanged */
When job J_j terminates
      dequeue J_j from the server queue;
      n = n - 1;
      if (n != 0) serve the next job in the queue with deadline d_k;
When job J_j executes for a time unit
      c = c - 1;
When (c == 0)
      /*---------------Rule 3---------------*/
      k = k + 1;
      a_k = actual time();
      d_k = d_{k-1} + T_s;
      c = Q_s;
```

Fig. 6.15 The CBS algorithm

The following lemma provides a simple guarantee test for verifying the feasibility of a task set consisting of hard and soft tasks.

Lemma 6.6 *Given a set of n periodic hard tasks with processor utilization U_p and a set of m CBSs with processor utilization $U_s = \sum_{i=1}^{m} U_{s_i}$, the whole set is schedulable by EDF if and only if*

$$U_p + U_s \leq 1.$$

The isolation property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee,

even in the case in which the execution times of the soft tasks are not known or the soft requests exceed the expected load.

In addition to the isolation property, the CBS has the following characteristics:

- The CBS behaves as a plain EDF algorithm if the served task τ_i has parameters (C_i, T_i) such that $C_i \leq Q_s$ and $T_i = T_s$. This is formally stated by the following lemma.

  Lemma 6.7 *A hard task τ_i with parameters (C_i, T_i) is schedulable by a CBS with parameters $Q_s \geq C_i$ and $T_s = T_i$ if and only if τ_i is schedulable with EDF.*

  Proof For any job of a hard task we have that $r_{i,j+1} - r_{i,j} \geq T_i$ and $c_{i,j} \leq Q_s$. Hence, by definition of the CBS, each hard job is assigned a deadline $d_{i,j} = r_{i,j} + T_i$, and it is scheduled with a budget $Q_s \geq C_i$. Moreover, since $c_{i,j} \leq Q_s$, each job finishes no later than the budget is exhausted; hence the deadline assigned to a job is never postponed and is exactly the same as the one used by EDF. □

- The CBS automatically reclaims any spare time caused by early completions. This is due to the fact that, whenever the budget is exhausted, it is always immediately replenished at its full value and the server deadline is postponed. In this way, the server remains eligible, and the budget can be exploited by the pending requests with the current deadline. *This is the main difference with respect to the processor capacity reserves* proposed by Mercer et al. [MST93, MST94a].
- Knowing the statistical distribution of the computation time of a task served by a CBS, it is possible to perform a QoS guarantee based on probabilistic deadlines (expressed in terms of probability for each served job to meet a deadline). Such a statistical analysis is presented in [AB98].

6.9.5 Simulation Results

This section shows how the CBS can be efficiently used as a service mechanism for improving responsiveness of soft aperiodic requests. Its performance has been tested against that of TBS and DSS, by measuring the mean tardiness experienced by soft tasks

$$E_{i,j} = max\{0, f_{i,j} - d_{i,j}\} \tag{6.5}$$

where $f_{i,j}$ is the finishing time of job $J_{i,j}$.

Such a metric was selected because in many soft real-time applications (e.g., multimedia), meeting all soft deadlines is either impossible or very inefficient; hence, the system should be designed to guarantee all the hard tasks and minimize the mean time that soft tasks execute after their deadlines.

All the simulations presented in this section have been conducted on a hybrid task set consisting of five periodic hard tasks with fixed parameters and five soft

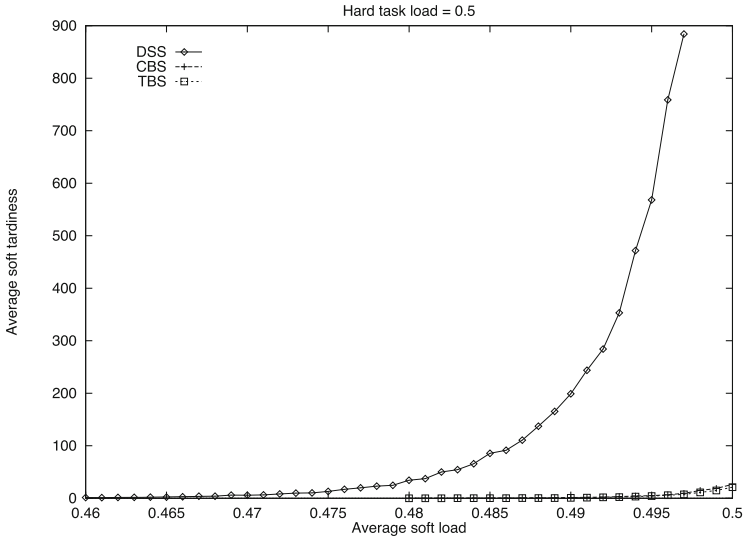


Fig. 6.16 First experiment: performance of TBS, CBS, and DSS

tasks with variable execution times and interarrival times. The execution times of the periodic hard tasks were randomly generated in order to achieve a desired processor utilization factor U_{hard}. The execution and interarrival times of the soft tasks were uniformly distributed in order to obtain a mean soft load $\overline{U_{soft}} = \sum_i \frac{\overline{c_{i,j}}}{r_{i,j+1} - r_{i,j}}$ with $\overline{U_{soft}}$ going from 0 to $1 - U_{hard}$.

The first experiment compares the mean tardiness experienced by soft tasks when they are served by a CBS, a TBS, and a DSS. In this test, the utilization factor of periodic hard tasks was $U_{hard} = 0.5$. The simulation results are illustrated in Fig. 6.16, which shows that the performance of the DSS is dramatically worse than the one achieved by the CBS and TBS. The main reason for such a different behavior between DSS and CBS is that, while the DSS becomes idle until the next replenishing time (that occurs at the server's deadline), the CBS remains eligible by increasing its deadline and replenishing the budget immediately. The TBS does not suffer from this problem; however *its correct behavior relies on the exact knowledge of WCETs*, so it cannot be used for supporting applications with highly variable computation times.

Figure 6.17 illustrates the results of a similar experiment repeated with $U_{hard} = 0.7$. As we can see, TBS slightly outperforms CBS, but does not protect hard tasks from transient overruns that may occur in the soft activities. Note that, since the CBS automatically reclaims any available idle time coming from early completions, for a fair comparison, an explicit reclaiming mechanism has also been added in the simulation of the TBS, as described in [SBS95].

The advantage of the CBS over the TBS can be appreciated when $WCET_i >> \overline{c_{i,j}}$. In this case, in fact, the TBS can cause an under-utilization of the processor,

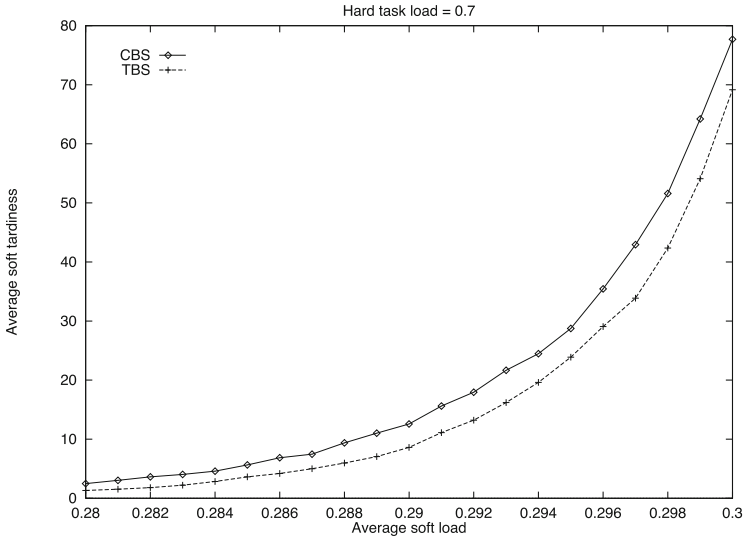


Fig. 6.17 Second experiment: CBS against TBS

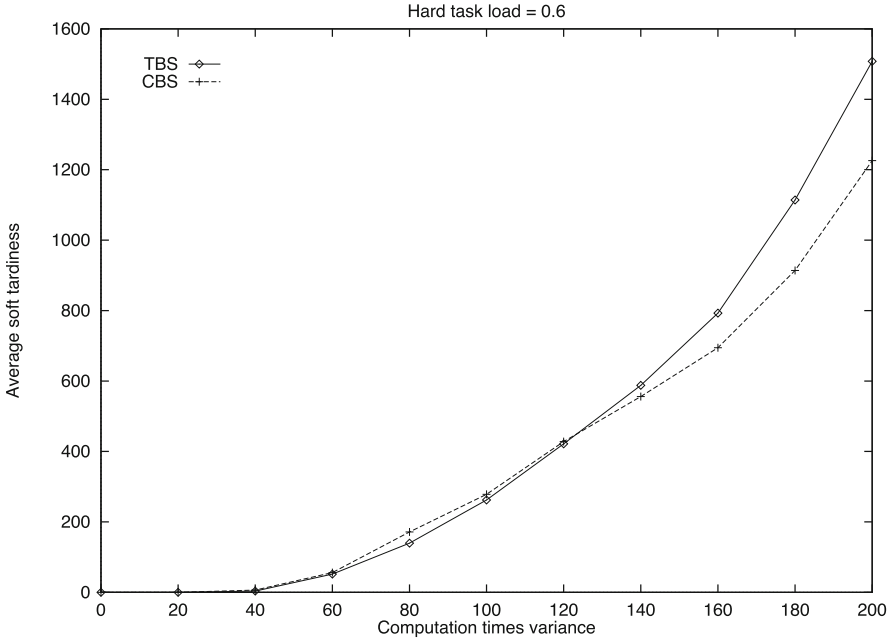


Fig. 6.18 Third experiment: CBS against TBS with variable execution times

due to its worst-case assumptions. This fact can be observed in Fig. 6.18, which shows the results of a fourth experiment, in which $U_{hard} = 0.6$, $\overline{U_{soft}} = 0.4$, the interarrival times are fixed, and the execution times of the soft tasks are uniformly

distributed with an increasing variance. As can be seen from the graph, the CBS performs better than the TBS when tasks' execution times have a high variance. Additional experiments on the CBS are presented in the original work by Abeni and Buttazzo [AB98].

6.9.6 Dimensioning CBS Parameters

This section presents a statistical study to evaluate the effects of the CBS parameters (Q_s, T_s) on task response times and proposes a technique to compute the parameters that minimize the average response time of the served tasks [BB06].

The worst-case response time R_i of a job with computation time C_i served by a CBS with bandwidth U_s is a function of the server budget Q_s. For the sake of clarity, R_i is first derived by neglecting the overhead and then modified to take the overhead into account.

From the CBS analysis, we know that, if the task set is feasible, that is, if the total processor utilization is less than 1, then the served job can never miss the current server deadline. Hence, the maximum response time R_i occurs when the other tasks in the system create the maximum interference on the server. If the computation time C_i of the served job is exactly a multiple of the server budget Q_s, then the job finishes at the server deadline, that is

$$R_i = \frac{C_i}{Q_s} T_s = \frac{C_i}{U_s}. \tag{6.6}$$

More generally, if the computation time C_i of the job is not multiple of the budget Q_s, the last portion of the job will not finish at the server deadline, but it will finish at most Δ_i units before the deadline, as shown in Fig. 6.19, where

$$\Delta_i = \left\lceil \frac{C_i}{Q_s} \right\rceil Q_s - C_i. \tag{6.7}$$

Hence, the response time of the job becomes

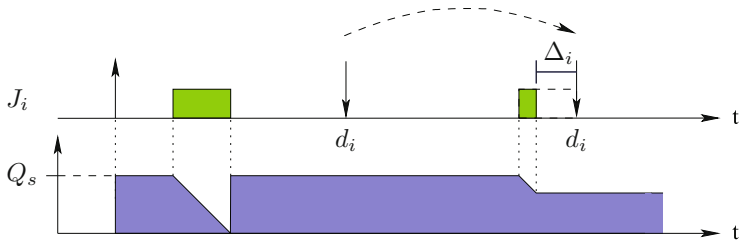


Fig. 6.19 Worst-case finishing time of a job served by a CBS

$$
\begin{aligned}
R_i &= \left\lceil \frac{C_i}{Q_s} \right\rceil T_s - \Delta_i \\
&= \left\lceil \frac{C_i}{Q_s} \right\rceil T_s - \left(\left\lceil \frac{C_i}{Q_s} \right\rceil Q_s - C_i \right) \\
&= C_i + \left\lceil \frac{C_i}{Q_s} \right\rceil (T_s - Q_s). \qquad (6.8)
\end{aligned}
$$

Figure 6.20 illustrates the worst-case response time of a CBS as a function of the budget.

From the graph shown in Fig. 6.20, it is clear that, for a given job with constant execution time C_i, the minimum worst-case response time is C_i/U_s and can be achieved when C_i is a perfect multiple of Q_s. In practice, however, task execution time varies, inducing response time fluctuations due to the bandwidth enforcement mechanism achieved through deadline postponements. From Fig. 6.20, it is also clear that such fluctuations would be reduced by making the budget very small compared to the average execution time, so that the server would approximate the ideal fluid server. Unfortunately, however, a small budget (which means a short server period) causes the job to be split in many small chunks, increasing the runtime overhead. As a consequence, to properly set the server granularity T_s, the runtime overhead must be taken into account in the analysis.

6.9.6.1 Taking Overheads into Account

Whenever the budget is exhausted, the server deadline is postponed, so the served job can be preempted by other tasks with earliest deadline. If ϵ denotes the time

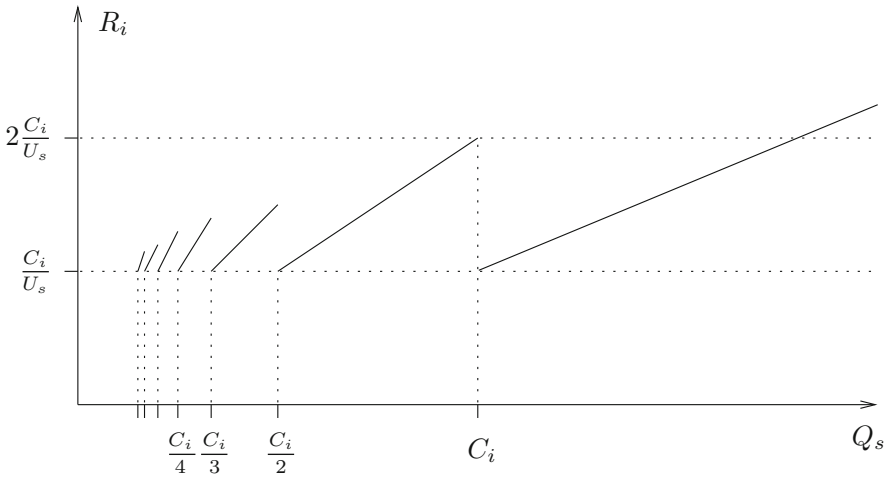


Fig. 6.20 Worst-case response time of a CBS as a function of the budget

needed for a context switch, then the overhead introduced by the CBS can be taken into account by subtracting such a time from the server budget. Hence, Eq. (6.8) can be modified as follows:

$$R_i = C_i + \left\lceil \frac{C_i}{Q_s - \epsilon} \right\rceil (T_s - Q_s + \epsilon)$$
$$= C_i + \left\lceil \frac{C_i}{T_s U_s - \epsilon} \right\rceil (T_s - T_s U_s + \epsilon). \quad (6.9)$$

Figure 6.21 illustrates the worst-case response time of a CBS as a function of the period, with and without overhead. Equation (6.9) has been plotted for $C_i = 10$, $U_s = 0.25$, and $\epsilon = 0.2$. As is clear from the plot, the overhead prevents using small values of the period; hence it is interesting to find the value of the server period that minimizes the response time. Note that, for computing the probability distribution function of the response time, we actually need to express the response time as a function of the job execution time C_i.

Figure 6.22 illustrates the worst-case response time of a CBS as a function of the job execution time. As it can be noticed from Fig. 6.22, the response time R_i can be upper bounded by the following linear function:

$$R_i^{\mathsf{ub}} = T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon} C_i \quad (6.10)$$

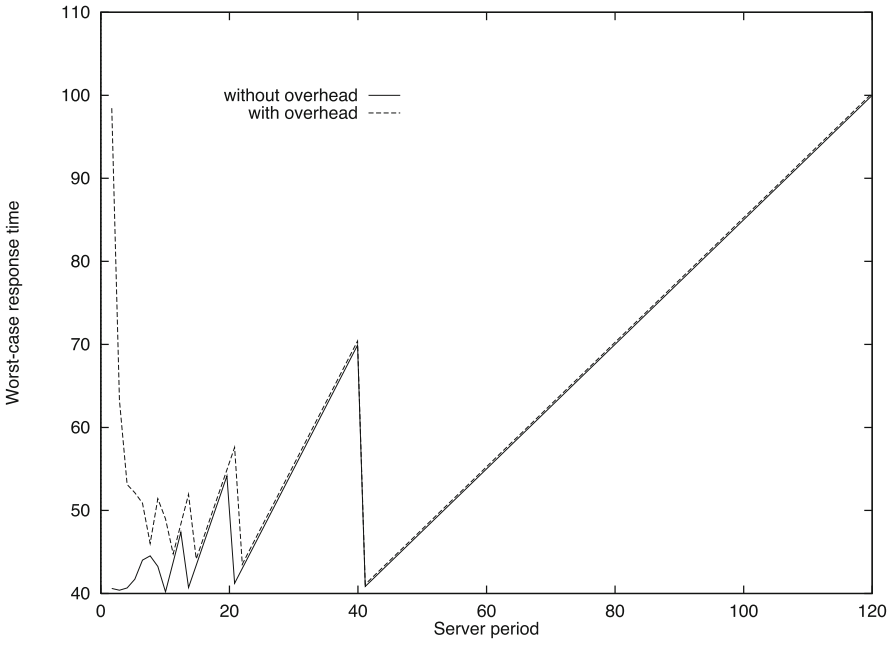


Fig. 6.21 Worst-case response time of a CBS as a function of the period

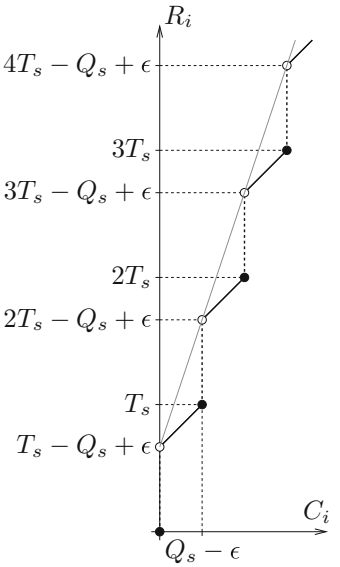


Fig. 6.22 Worst-case response time of a CBS as a function of the job execution time

and lower bounded by

$$R_i^{\mathsf{lb}} = \frac{T_s}{Q_s - \epsilon} C_i. \tag{6.11}$$

We now consider the problem of selecting the best CBS parameters, such that the average task response time R_i is minimized. For this purpose, we suppose to have the probability density function (p.d.f.) $f_C(c)$ of the task execution time and the respective cumulative distribution function (c.d.f.) $F_C(c)$, representing the probability that the execution time is smaller than or equal to c. That is,

$$F_C(c) = \int_0^c f_C(x)\mathrm{d}x. \tag{6.12}$$

Since the minimization of the average R_i can in general be too complex, we consider the problem of minimizing its linear upper bound R_i^{ub}. In this case, the average response time R_i^{avg} is computed as follows:

$$\begin{aligned} R_i^{\mathsf{avg}} &= \int_0^{+\infty} \left(T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon} x \right) f_C(x)\mathrm{d}x \\ &= T_s - Q_s + \epsilon + \frac{T_s}{Q_s - \epsilon} C^{\mathsf{avg}} \\ &= T_s(1 - U_s) + \epsilon + \frac{T_s}{T_s\, U_s - \epsilon} C^{\mathsf{avg}} \end{aligned} \tag{6.13}$$

Hence, the period T_s which minimizes the average response time R_i^{avg} can be computed by simple functional analysis. Thus, we have

$$\frac{\mathrm{d}R_i^{\mathsf{avg}}}{\mathrm{d}T_s} = 1 - U_s - \frac{\epsilon}{(T_s\,U_s - \epsilon)^2} C^{\mathsf{avg}} \tag{6.14}$$

which is equal to zero when

$$T_s = \frac{1}{U_s}\left(\epsilon + \sqrt{\frac{\epsilon\, C^{\mathsf{avg}}}{1 - U_s}}\right). \tag{6.15}$$

6.10 Summary

The experimental simulations have established that, from a performance point of view, IPE, EDL, and TB* show the best results for reducing aperiodic responsiveness. Although optimal, however, EDL is far from being reasonably practical, due to the overall complexity. On the other hand, IPE and TB* achieve a comparable performance with much less computational overhead. Moreover, both EDL and IPE may require significant memory space when task periods are not harmonically related.

The Total Bandwidth algorithm also shows a good performance, sometimes comparable to that of the nearly optimal IPE. Observing that its implementation complexity is among the simplest, the TBS algorithm could be a good candidate for practical systems. In addition, the TBS deadline assignment rule can be tuned to enhance aperiodic responsiveness up to the optimal TB* behavior. Compared to IPE and EDL, TB* does not require large memory space, and the optimal deadline can be computed in $O(Nn)$ complexity, being N the maximum number of steps that have to be done for each task to shorten its initial deadline (assigned by the TBS rule). As for the EDL server, this is a pseudo-polynomial complexity, since in the worst case N can be large.

One major problem of the TBS and TB* algorithms is that they do not use a server budget for controlling aperiodic execution, but rely on the knowledge of the worst-case computation time specified by each job at its arrival. When such a knowledge is not available, not reliable, or too pessimistic (due to highly variable execution times), then hard tasks are not protected from transient overruns occurring in the soft tasks and could miss their deadlines. The CBS algorithm can be efficiently used in these situations, since it has a performance comparable to the one of the TBS and also provides temporal isolation, by limiting the bandwidth requirements of the served tasks to the value U_s specified at design time.

Figure 6.23 provides a qualitative evaluation of the algorithms presented in this chapter in terms of performance, computational complexity, memory requirement, and implementation complexity.

	performance	computational complexity	memory requirement	implementation complexity
BKG				
DPE				
DSS				
TBS				
EDL				
IPE				
TB*				
CBS				

Fig. 6.23 Evaluation summary of dynamic-priority servers

Exercises

6.1 Compute the maximum processor utilization that can be assigned to a Dynamic Sporadic Server to guarantee the following periodic tasks, under EDF:

	C_i	T_i
τ_1	2	6
τ_2	3	9

6.2 Together with the periodic tasks illustrated in Exercise 6.1, schedule the following aperiodic tasks with a Dynamic Sporadic Server with $C_s = 2$ and $T_s = 6$.

	a_i	C_i
J_1	1	3
J_2	5	1
J_3	15	1

6.3 Solve the same scheduling problem described in Exercise 6.2 with a Total Bandwidth Server having utilization $U_s = 1/3$.

6.4 Solve the same scheduling problem described in Exercise 6.2 with a Constant Bandwidth Server with $C_s = 2$ and $T_s = 6$.

6.5 Solve the same scheduling problem described in Exercise 6.2 with an Improved Total Bandwidth Server with $U_s = 1/3$, which performs only one shortening step.

6.6 Solve the same scheduling problem described in Exercise 6.2 with the optimal Total Bandwidth Server (TB*).

6.7 Consider the following set of periodic tasks:

	C_i	T_i
τ_1	4	10
τ_2	4	12

After defining two Total Bandwidth Servers, TB_1 and TB_2, with utilization factors $U_{s1} = 1/10$ and $U_{s2} = 1/6$, construct the EDF schedule in the case in which two aperiodic requests $J_1(a_1 = 1, C_1 = 1)$ and $J_2(a_2 = 9, C_2 = 1)$ are served by TB_1 and two aperiodic requests $J_3(a_3 = 2, C_3 = 1)$ and $J_4(a_4 = 6, C_4 = 2)$ are served by TB_2.

6.8 A control application consists of two periodic tasks with computation times $C_1 = 8$, $C_2 = 6$ ms, and periods $T_1 = 20$, $T_2 = 30$ ms. Moreover, the system includes two interrupt handling routines, with computation times of 1.0 and 1.4 ms each. Considering a context switch cost of 20 µs, compute the CBS parameters that minimize the average response time of the interrupts.

Chapter 7
Resource Access Protocols

7.1 Introduction

A *resource* is any software structure that can be used by a process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*. A shared resource protected against concurrent accesses is called an *exclusive resource*.

To ensure consistency of the data structures in exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A piece of code executed under mutual exclusion constraints is called a *critical section*.

Any task that needs to enter a critical section must wait until no other task is holding the resource. A task waiting for an exclusive resource is said to be *blocked* on that resource; otherwise it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the resource associated with the critical section becomes *free*, and it can be allocated to another waiting task, if any.

Operating systems typically provide a general synchronization tool, called a *semaphore* [Dij68, BH73, PS85], that can be used by tasks to build critical sections. A semaphore is a kernel data structure that, apart from initialization, can be accessed only through two kernel primitives, usually called *wait* and *signal*. When using this tool, each exclusive resource R_k must be protected by a different semaphore S_k, and each critical section operating on a resource R_k must begin with a $wait(S_k)$ primitive and end with a $signal(S_k)$ primitive.

All tasks blocked on a resource are kept in a queue associated with the semaphore that protects the resource. When a running task executes a *wait* primitive on a locked semaphore, it enters a *waiting* state, until another task executes a *signal* primitive that unlocks the semaphore. When a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_7

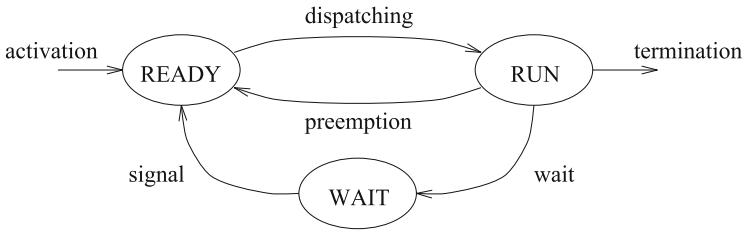


Fig. 7.1 Waiting state caused by resource constraints

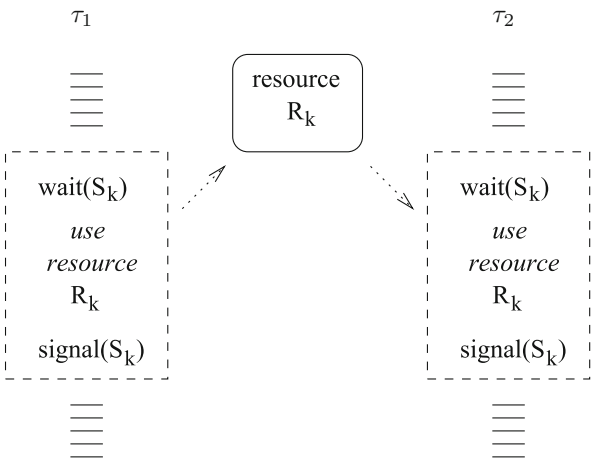


Fig. 7.2 Structure of two tasks that share an exclusive resource

the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Fig. 7.1.

In this chapter, we describe the main problems that may arise in a uniprocessor system when concurrent tasks use shared resources in exclusive mode, and we present some resource access protocols designed to avoid such problems and bound the maximum blocking time of each task. We then show how such blocking times can be used in the schedulability analysis to extend the guarantee tests derived for periodic task sets.

7.2 The Priority Inversion Phenomenon

Consider two tasks τ_1 and τ_2 that share an exclusive resource R_k (such as a list), on which two operations (such as *insert* and *remove*) are defined. To guarantee the mutual exclusion, both operations must be defined as critical sections. If a binary semaphore S_k is used for this purpose, then each critical section must begin with a *wait(S_k)* primitive and must end with a *signal(S_k)* primitive (see Fig. 7.2).

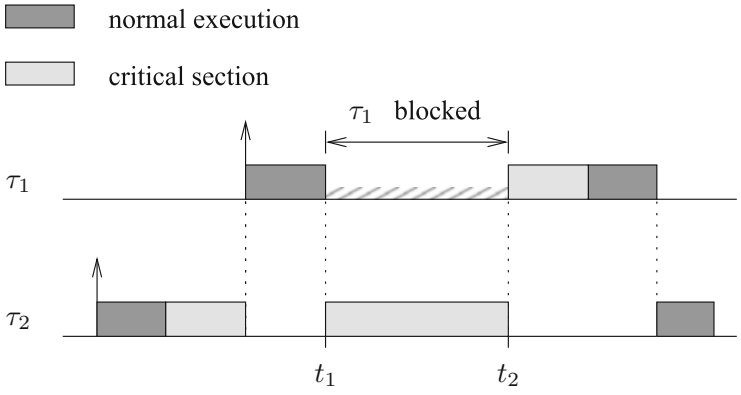


Fig. 7.3 Example of blocking on an exclusive resource

If preemption is allowed and τ_1 has a higher priority than τ_2, then τ_1 can be blocked in the situation depicted in Fig. 7.3. Here, task τ_2 is activated first, and, after a while, it enters the critical section and locks the semaphore. While τ_2 is executing the critical section, task τ_1 arrives, and, since it has a higher priority, it preempts τ_2 and starts executing. However, at time t_1, when attempting to enter its critical section, τ_1 is blocked on the semaphore, so τ_2 resumes. τ_1 has to wait until time t_2, when τ_2 releases the critical section by executing the *signal(S_k)* primitive, which unlocks the semaphore.

In this simple example, the maximum blocking time that τ_1 may experience is equal to the time needed by τ_2 to execute its critical section. Such a blocking cannot be avoided because it is a direct consequence of the mutual exclusion necessary to protect the shared resource against concurrent accesses of competing tasks.

Unfortunately, in the general case, the blocking time of a task on a busy resource cannot be bounded by the duration of the critical section executed by the lower-priority task. In fact, consider the example illustrated in Fig. 7.4. Here, three tasks τ_1, τ_2, and τ_3 have decreasing priorities, and τ_1 and τ_3 share an exclusive resource protected by a binary semaphore S.

If τ_3 starts at time t_0, it may happen that τ_1 arrives at time t_2 and preempts τ_3 inside its critical section. At time t_3, τ_1 attempts to use the resource, but it is blocked on the semaphore S; thus, τ_3 continues the execution inside its critical section. Now, if τ_2 arrives at time t_4, it preempts τ_3 (because it has a higher priority) and increases the blocking time of τ_1 by its entire duration. As a consequence, the maximum blocking time that τ_1 may experience does depend not only on the length of the critical section executed by τ_3 but also on the worst-case execution time of τ_2! This is a situation that, if it recurs with other medium-priority tasks, can lead to uncontrolled blocking and can cause critical deadlines to be missed. A *priority inversion* is said to occur in the interval $[t_3, t_6]$, since the highest-priority task τ_1 waits for the execution of lower-priority tasks (τ_2 and τ_3). In general, the duration

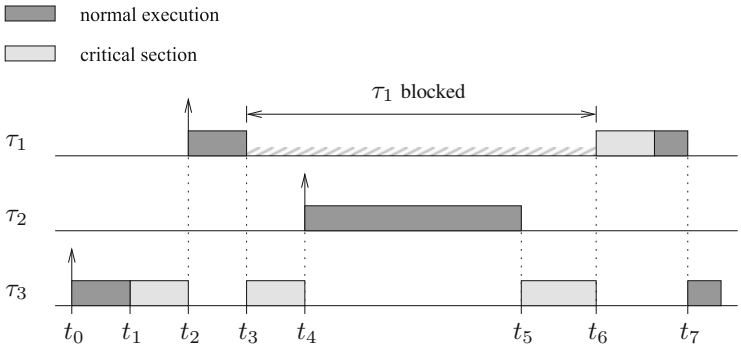


Fig. 7.4 An example of priority inversion

of priority inversion is unbounded, since any intermediate-priority task that can preempt τ_3 will indirectly block τ_1.

Several approaches have been defined to avoid priority inversion, both under fixed and dynamic priority scheduling.[1] All the methods developed in the context of fixed priority scheduling consist in raising the priority of a task when accessing a shared resource, according to a given protocol for entering and exiting critical sections.

Similarly, the methods developed under EDF consist in modifying a parameter based on the tasks' relative deadlines.

The rest of this chapter presents the following resource access protocols:

1. Non-Preemptive Protocol (NPP);
2. Highest Locker Priority (HLP), also called Immediate Priority Ceiling (IPC);
3. Priority Inheritance Protocol (PIP);
4. Priority Ceiling Protocol (PCP);
5. Stack Resource Policy (SRP);

Although the first four protocols have been developed under fixed priority assignments, some of them have been extended under EDF.[2] The Stack Resource Policy, instead, was natively designed to be applicable under both fixed and dynamic priority assignments.

7.3 Terminology and Assumptions

Throughout this chapter, we consider a set of n periodic tasks, $\tau_1, \tau_2, \ldots, \tau_n$, which cooperate through m shared resources, $R_1, R_2, \ldots, R_m$. Each task is characterized

[1] Under EDF, such a phenomenon is sometime referred to as *deadline inversion*.

[2] The Priority Inheritance Protocol has been extended for EDF by Spuri [Spu95], and the Priority Ceiling Protocol has been extended for EDF by Chen and Lin [CL90].

by a period T_i and a worst-case computation time C_i. Each resource R_k is guarded by a distinct semaphore S_k. Hence, all critical sections on resource R_k begin with a $wait(S_k)$ operation and end with a $signal(S_k)$ operation. Since a protocol modifies the task priority, each task is characterized by a fixed *nominal* priority P_i (assigned, e.g., by the Rate Monotonic algorithm) and an *active* priority p_i ($p_i \geq P_i$), which is dynamic and initially set to P_i. The following notation is adopted throughout the discussion:

- B_i denotes the maximum blocking time task τ_i can experience.
- $z_{i,k}$ denotes a generic critical section of task τ_i guarded by semaphore S_k.
- $Z_{i,k}$ denotes the longest critical section of task τ_i guarded by semaphore S_k.
- $\delta_{i,k}$ denotes the duration of $Z_{i,k}$.
- $z_{i,h} \subset z_{i,k}$ indicates that $z_{i,h}$ is entirely contained in $z_{i,k}$.
- σ_i denotes the set of semaphores used by τ_i.
- $\sigma_{i,j}$ denotes the set of semaphores that can block τ_i, used by the lower-priority task τ_j.
- $\gamma_{i,j}$ denotes the set of the longest critical sections that can block τ_i, accessed by the lower-priority task τ_j. That is,

$$\gamma_{i,j} = \{Z_{j,k} \mid (P_j < P_i) \text{ and } (S_k \in \sigma_{i,j})\} \tag{7.1}$$

- γ_i denotes the set of all the longest critical sections that can block τ_i. That is,

$$\gamma_i = \bigcup_{j:P_j<P_i} \gamma_{i,j} \tag{7.2}$$

Moreover, the properties of the protocols are valid under the following assumptions:

- Tasks $\tau_1, \tau_2, \ldots, \tau_n$ are assumed to have different priorities and are listed in descending order of nominal priority, with τ_1 having the highest nominal priority.
- Tasks do not suspend themselves on I/O operations or on explicit synchronization primitives (except on locked semaphores).
- The critical sections used by any task are *properly* nested; that is, given any pair $z_{i,h}$ and $z_{i,k}$, then either $z_{i,h} \subset z_{i,k}$, $z_{i,k} \subset z_{i,h}$, or $z_{i,h} \cap z_{i,k} = \emptyset$.
- Critical sections are guarded by binary semaphores, meaning that only one task at a time can be within a critical section.

7.4 Non-preemptive Protocol

A simple solution that avoids the unbounded priority inversion problem is to disallow preemption during the execution of any critical section. This method, also referred to as (NPP) can be implemented by raising the priority of a task to the highest-priority level whenever it enters a shared resource. In particular, as soon as

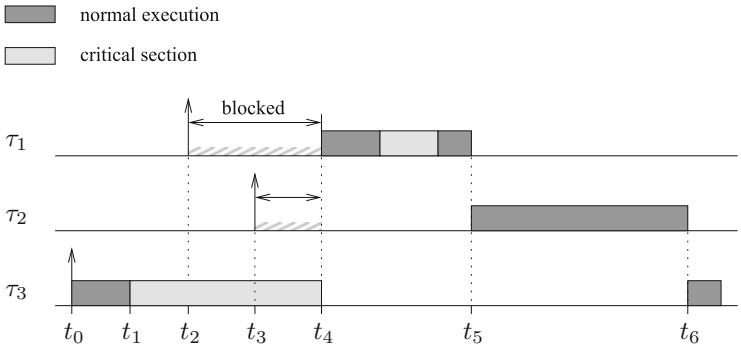


Fig. 7.5 Example of NPP preventing priority inversion

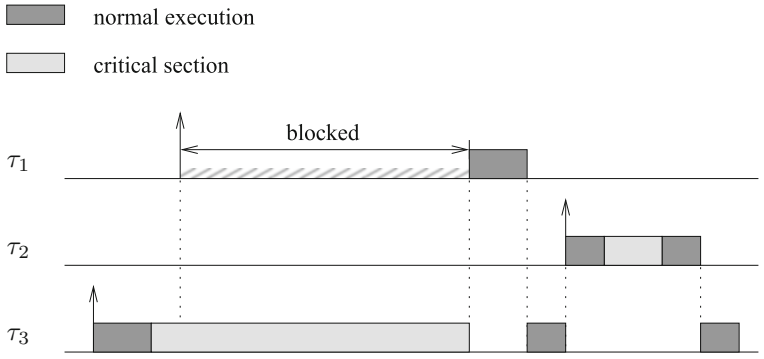


Fig. 7.6 Example in which NPP causes unnecessary blocking on τ_1

a task τ_i enters a resource R_k, its dynamic priority is raised to the level:

$$p_i(R_k) = \max_h \{P_h\}. \tag{7.3}$$

The dynamic priority is then reset to the nominal value P_i when the task exits the critical section. Figure 7.5 shows how NPP solves the priority inversion phenomenon. This method, however, is only appropriate when tasks use short critical sections, because it creates unnecessary blocking. Consider, for example, the case depicted in Fig. 7.6, where τ_1 is the highest-priority task that does not use any resource, whereas τ_2 and τ_3 are low-priority tasks that share an exclusive resource. If the low-priority task τ_3 enters a long critical section, τ_1 may unnecessarily be blocked for a long period of time.

7.4.1 Blocking Time Computation

Since a task τ_i cannot preempt a lower-priority task τ_j if τ_j is inside a critical section, then τ_i can potentially be blocked by any critical section belonging to a lower-priority task. Hence,

$$\gamma_i = \{Z_{j,k} \mid P_j < P_i,\ k = 1, \ldots, m\}$$

Moreover, since a task inside a resource R cannot be preempted, only one resource can be locked at any time t. Hence, a task τ_i can be blocked at most for the length of a single critical section belonging to lower-priority tasks. As a consequence, the maximum blocking time τ_i can suffer is given by the duration of the longest critical section among those belonging to lower-priority tasks. That is,

$$B_i = \max_{j,k}\{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\}. \tag{7.4}$$

The unnecessary blocking illustrated in Fig. 7.6 can easily be avoided by raising the priority inside a critical section at an opportune level that does not prevent a high-priority task to preempt when it is not sharing that resource.

7.5 Highest Locker Priority Protocol

The *Highest Locker Priority* (HLP) protocol improves NPP by raising the priority of a task that enters a resource R_k to the highest priority among the tasks sharing that resource. In particular, as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the level

$$p_i(R_k) = \max_h\{P_h \mid \tau_h \text{ uses } R_k\}. \tag{7.5}$$

The dynamic priority is then reset to the nominal value P_i when the task exits the critical section. The online computation of the priority level in Eq. (7.5) can be simplified by assigning each resource R_k a *priority ceiling* $C(R_k)$ (computed offline) equal to the maximum priority of the tasks sharing R_k, that is,

$$C(R_k) \stackrel{\text{def}}{=} \max_h\{P_h \mid \tau_h \text{ uses } R_k\}. \tag{7.6}$$

Then, as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the ceiling of the resource. For this reason, this protocol is also referred to as *Immediate Priority Ceiling*.

Note that the schedule produced under HLP for the example generating priority inversion is the same as the one shown in Fig. 7.5, since the ceiling of the shared

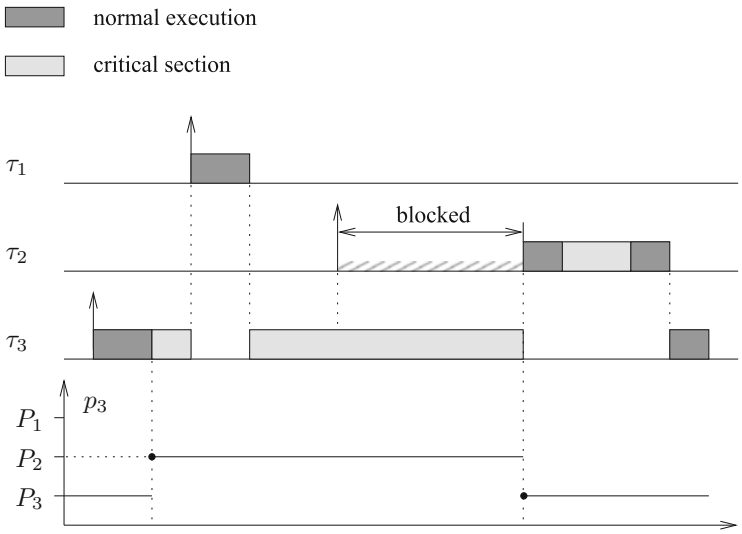


Fig. 7.7 Example of schedule under HLP, where p_3 is raised at the level $C(R) = P_2$ as soon as τ_3 starts using resource R

resource is P_1 (the highest priority). Figure 7.7 shows another example in which the ceiling of the shared resource is P_2; hence τ_1 can preempt τ_3 inside the critical section, whereas τ_2 is blocked until τ_3 exits its critical section. The figure also shows how the active priority of τ_3 is varied by the protocol during execution.

7.5.1 Blocking Time Computation

Under HLP, a task τ_i can only be blocked by critical sections belonging to lower-priority tasks with a resource ceiling higher than or equal to P_i. Hence,

$$\gamma_i = \{Z_{j,k} \mid (P_j < P_i) \text{ and } C(R_k) \geq P_i\}.$$

Moreover, a task can be blocked at most once, as formally stated in the following theorem.

Theorem 7.1 *Under HLP, a task τ_i can be blocked, at most, for the duration of a single critical section belonging to the set γ_i.*

Proof The theorem is proved by contradiction, assuming that τ_i is blocked by two critical sections, $z_{1,a}$ and $z_{2,b}$. For this to happen, both critical sections must belong to different tasks (τ_1 and τ_2) with priority lower than P_i, and both must have a ceiling higher than or equal to P_i. That is, by assumption, we must have

$$P_1 < P_i \leq C(R_a);$$
$$P_2 < P_i \leq C(R_b).$$

Now, τ_i can be blocked twice only if τ_1 and τ_2 are both inside the resources when τ_i arrives, and this can occur only if one of them (say τ_1) preempted the other inside the critical section. But, if τ_1 preempted τ_2 inside $z_{2,b}$, it means that $P_1 > C(R_b)$, which is a contradiction. Hence the theorem follows. □

Since a task τ_i can be blocked at most once, the maximum blocking time τ_i can suffer is given by the duration of the longest critical section among those that can block τ_i. That is,

$$B_i = \max_{j,k}\{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\}. \tag{7.7}$$

This method, although improving NPP, still contains a source of pessimism and could produce some unnecessary blocking. In fact, a task is blocked at the time it attempts to preempt, before it actually requires a resource. If a critical section is contained only in one branch of a conditional statement, then the task could be unnecessarily blocked, since during execution it could take the branch without the resource.

Such a source of pessimism is removed in the Priority Inheritance Protocol by postponing the blocking condition at the entrance of a critical section, rather than at the activation time.

7.6 Priority Inheritance Protocol

The *Priority Inheritance Protocol* (PIP), proposed by Sha, Rajkumar, and Lehoczky [SRL90], avoids unbounded priority inversion by modifying the priority of those tasks that cause blocking. In particular, when a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by the higher-priority tasks.

7.6.1 Protocol Definition

The Priority Inheritance Protocol can be defined as follows:

- Tasks are scheduled based on their active priorities. Tasks with the same priority are executed in a first-come first-served discipline.

- When task τ_i tries to enter a critical section $z_{i,k}$ and resource R_k is already held by a lower-priority task τ_j, then τ_i is blocked. τ_i is said to be blocked by the task τ_j that holds the resource. Otherwise, τ_i enters the critical section $z_{i,k}$.
- When a task τ_i is blocked on a semaphore, it transmits its active priority to the task, say τ_j, that holds that semaphore. Hence, τ_j resumes and executes the rest of its critical section with a priority $p_j = p_i$. Task τ_j is said to *inherit* the priority of τ_i. In general, a task inherits the highest priority of the tasks it blocks. That is, at every instant,

$$p_j(R_k) = \max \left\{ P_j, \max_h \{P_h | \tau_h \text{ is blocked on } R_k\} \right\}. \tag{7.8}$$

- When τ_j exits a critical section, it unlocks the semaphore, and the highest-priority task blocked on that semaphore, if any, is awakened. Moreover, the active priority of τ_j is updated as follows: if no other tasks are blocked by τ_j, p_j is set to its nominal priority P_j; otherwise it is set to the highest priority of the tasks blocked by τ_j, according to Eq. (7.8).
- Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2.

7.6.1.1 Examples

We first consider the same situation presented in Fig. 7.4 and show how the priority inversion phenomenon can be bounded by the Priority Inheritance Protocol. The modified schedule is illustrated in Fig. 7.8. Until time t_3, there is no variation in the schedule, since no priority inheritance takes place. At time t_3, τ_1 is blocked by τ_3; thus τ_3 inherits the priority of τ_1 and executes the remaining part of its critical section (from t_3 to t_5) at the highest priority. In this condition, at time t_4, τ_2 cannot preempt τ_3 and cannot create additional interference on τ_1. As τ_3 exits its critical section, τ_1 is awakened, and τ_3 resumes its original priority. At time t_5, the processor is assigned to τ_1, which is the highest-priority task ready to execute, and task τ_2 can only start at time t_6, when τ_1 has completed. The active priority of τ_3 as a function of time is also shown in Fig. 7.8 on the lowest timeline.

From this example, we can notice that a high-priority task can experience two kinds of blocking:

- **Direct blocking**. It occurs when a higher-priority task tries to acquire a resource already held by a lower-priority task. Direct blocking is necessary to ensure the consistency of the shared resources.
- **Push-through blocking**. It occurs when a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks. Push-through blocking is necessary to avoid unbounded priority inversion.

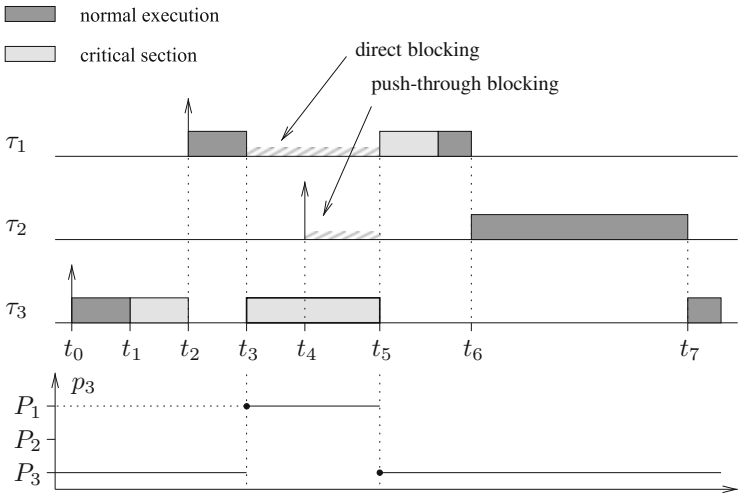


Fig. 7.8 Example of Priority Inheritance Protocol

Notice that, in most situations, when a task exits a critical section, it resumes the priority it had when it entered. However, this is not true in general. Consider the example illustrated in Fig. 7.9. Here, task τ_1 uses a resource R_a guarded by a semaphore S_a, task τ_2 uses a resource R_b guarded by a semaphore S_b, and task τ_3 uses both resources in a nested fashion (S_a is locked first). At time t_1, τ_2 preempts τ_3 within its nested critical section; hence, at time t_2, when τ_2 attempts to lock S_b, τ_3 inherits its priority, P_2. Similarly, at time t_3, τ_1 preempts τ_3 within the same critical section, and, at time t_4, when τ_1 attempts to lock S_a, τ_3 inherits the priority P_1. At time t_5, when τ_3 unlocks semaphore S_b, task τ_2 is awakened but τ_1 is still blocked; hence, τ_3 continues its execution at the priority of τ_1. At time t_6, τ_3 unlocks S_a, and, since no other tasks are blocked, τ_3 resumes its original priority P_3.

An example of transitive priority inheritance is shown in Fig. 7.10. Here, task τ_1 uses a resource R_a guarded by a semaphore S_a, task τ_3 uses a resource R_b guarded by a semaphore S_b, and task τ_2 uses both resources in a nested fashion (S_a protects the external critical section and S_b the internal one). At time t_1, τ_3 is preempted within its critical section by τ_2, which in turn enters its first critical section (the one guarded by S_a), and at time t_2 it is blocked on semaphore S_b. As a consequence, τ_3 resumes and inherits the priority P_2. At time t_3, τ_3 is preempted by τ_1, which at time t_4 tries to acquire R_a. Since S_a is locked by τ_2, τ_2 inherits P_1. However, τ_2 is blocked by τ_3; hence, for transitivity, τ_3 inherits the priority P_1 via τ_2. When τ_3 exits its critical section, no other tasks are blocked by it; thus it resumes its nominal priority P_3. Priority P_1 is now inherited by τ_2, which still blocks τ_1 until time t_6.

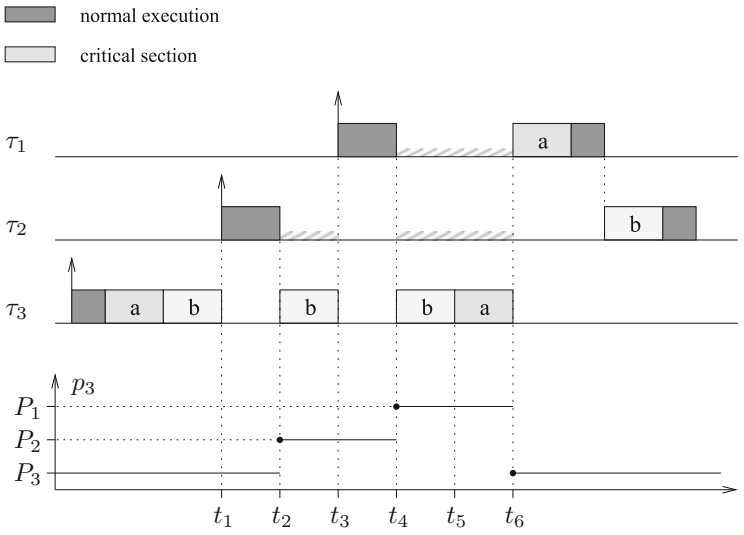


Fig. 7.9 Priority inheritance with nested critical sections

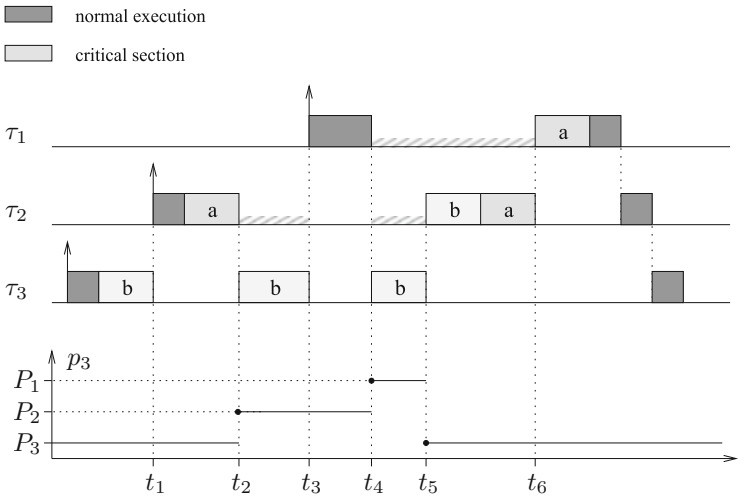


Fig. 7.10 Example of transitive priority inheritance

7.6.2 Properties of the Protocol

In this section, the main properties of the Priority Inheritance Protocol are presented. These properties are then used to analyze the schedulability of a periodic task set and compute the maximum blocking time that each task may experience.

Lemma 7.1 *A semaphore S_k can cause push-through blocking to task τ_i, only if S_k is accessed both by a task with priority lower than P_i and by a task with priority higher than P_i.*

Proof Suppose that semaphore S_k is accessed by a task τ_l with priority lower than P_i, but not by a task with priority higher than P_i. Then, τ_l cannot inherit a priority higher than P_i. Hence, τ_l will be preempted by τ_i and the lemma follows. □

Lemma 7.2 *Transitive priority inheritance can occur only in the presence of nested critical sections.*

Proof A transitive inheritance occurs when a high-priority task τ_H is blocked by a medium-priority task τ_M, which in turn is blocked by a low-priority task τ_L (see the example of Fig. 7.10). Since τ_H is blocked by τ_M, τ_M must hold a semaphore, say S_a. But τ_M is also blocked by τ_L on a different semaphore, say S_b. This means that τ_M attempted to lock S_b inside the critical section guarded by S_a. The lemma follows. □

Lemma 7.3 *If there are l_i lower-priority tasks that can block a task τ_i, then τ_i can be blocked for at most the duration of l_i critical sections (one for each of the l_i lower-priority tasks), regardless of the number of semaphores used by τ_i.*

Proof A task τ_i can be blocked by a lower-priority task τ_j only if τ_j has been preempted within a critical section, say $z_{j,k}$, that can block τ_i. Once τ_j exits $z_{j,k}$, it can be preempted by τ_i; thus, τ_i cannot be blocked by τ_j again. The same situation may happen for each of the l_i lower-priority tasks; therefore, τ_i can be blocked at most l_i times. □

Lemma 7.4 *If there are s_i distinct semaphores that can block a task τ_i, then τ_i can be blocked for at most the duration of s_i critical sections, one for each of the s_i semaphores, regardless of the number of critical sections used by τ_i.*

Proof Since semaphores are binary, only one of the lower-priority tasks, say τ_j, can be within a blocking critical section corresponding to a particular semaphore S_k. Once S_k is unlocked, τ_j can be preempted and no longer block τ_i again. If all s_i semaphores that can block τ_i are locked by the s_i lower-priority tasks, then τ_i can be blocked at most s_i times. □

Theorem 7.2 (Sha-Rajkumar-Lehoczky) *Under the Priority Inheritance Protocol, a task τ_i can be blocked for at most the duration of $\alpha_i = \min(l_i, s_i)$ critical sections, where l_i is the number of lower-priority tasks that can block τ_i and s_i is the number of distinct semaphores that can block τ_i.*

Proof It immediately follows from Lemmas 7.3 and 7.4. □

7.6.3 Blocking Time Computation

The evaluation of the maximum blocking time for each task can be computed based on the result of Theorem 7.2. However, a precise evaluation of the blocking factor B_i is quite complex because each critical section of the lower-priority tasks may interfere with τ_i via direct blocking, push-through blocking, or transitive inheritance. In this section, we present a simplified algorithm that can be used to compute the blocking factors of tasks that do not use nested critical sections. In this case, in fact, Lemma 7.2 guarantees that no transitive inheritance can occur; thus, the analysis of all possible blocking conditions is simplified.

Using the terminology introduced in Sect. 7.3, in the absence of transitive blocking, the set γ_i of critical sections that can block a task τ_i can be computed as follows:

1. The set of semaphores that can cause direct blocking to τ_i, shared by the lower-priority task τ_j, is
$$\sigma_{i,j}^{\text{dir}} = \sigma_i \cap \sigma_j.$$
2. The set of semaphores that can cause push-through blocking to τ_i, shared by the lower-priority task τ_j, is
$$\sigma_{i,j}^{\text{pt}} = \bigcup_{h:P_h > P_i} \sigma_h \cap \sigma_j.$$
3. The set of semaphores that can block τ_i (either directly or by push-through), shared by the lower-priority task τ_j, is
$$\sigma_{i,j} = \sigma_{i,j}^{\text{dir}} \cup \sigma_{i,j}^{\text{pt}} = \bigcup_{h:P_h \geq P_i} \sigma_h \cap \sigma_j.$$
4. The set of the longest critical sections used by τ_j that can block τ_i (either directly or by push-through) is then
$$\gamma_{i,j} = \{Z_{j,k} \mid (P_j < P_i) \text{ and } (R_k \in \sigma_{i,j}\}.$$
5. The set of all critical sections that can block τ_i (either directly or by push-through) is
$$\gamma_i = \bigcup_{j:P_j < P_i} \gamma_{i,j}.$$
6. B_i is then given by the largest sum of the lengths of the α_i critical sections in γ_i. Note that, since τ_i cannot be blocked twice by the same task or by the same

semaphore, the sum should contain only terms $\delta_{j,k}$ referring to different tasks and different semaphores.

Unfortunately, even without considering nested resources, the exact computation of each blocking factor requires a combinatorial search for finding the largest sum among all possible α_i durations. A simpler upper bound, however, can be computed according to the following algorithm.

1. According to Lemma 7.3, a task τ_i can be blocked at most once for each of the l_i lower-priority tasks. Hence, for each lower-priority task τ_j that can block τ_i, sum the duration of the longest critical section among those that can block τ_i; let B_i^l be this sum. That is,

$$B_i^l = \sum_{j:P_j<P_i} \max_k \{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\} \tag{7.9}$$

2. According to Lemma 7.4, a task τ_i can be blocked at most once for each of the s_i semaphores that can block τ_i. Hence, for each semaphore S_k that can block τ_i, sum the duration of the longest critical section among those that can block τ_i; let B_i^s be this sum. That is,

$$B_i^s = \sum_{k=1}^{m} \max_j \{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\} \tag{7.10}$$

3. According to Theorem 7.2, τ_i can be blocked at most for $\alpha_i = \min(l_i, s_i)$ critical sections. Hence, compute B_i as the minimum between B_i^l and B_i^s. That is,

$$B_i = \min\left(B_i^l, B_i^s\right) \tag{7.11}$$

The identification of the critical sections that can block a task can be greatly simplified if for each semaphore S_k we define a *ceiling* $C(S_k)$ equal to the highest-priority of the tasks that use S_k:

$$C(S_k) \stackrel{\text{def}}{=} \max_i \{P_i \mid S_k \in \sigma_i\}. \tag{7.12}$$

Then, the following lemma holds.

Lemma 7.5 *In the absence of nested critical sections, a critical section $z_{j,k}$ of τ_j guarded by S_k can block τ_i only if $P_j < P_i \leq C(S_k)$.*

Proof If $P_i \leq P_j$, then task τ_i cannot preempt τ_j; hence, it cannot be blocked by τ_j directly. On the other hand, if $C(S_k) < P_i$, by definition of $C(S_k)$, any task that uses S_k cannot have a priority higher than P_i. Hence, from Lemma 7.1, $z_{j,k}$ cannot cause push-through blocking on τ_i. Finally, since there are no nested

critical sections, Lemma 7.2 guarantees that $z_{j,k}$ cannot cause transitive blocking. The lemma follows. □

Using the result of Lemma 7.5, the blocking terms B_i^l and B_i^s can be determined as follows:

$$B_i^l = \sum_{j=i+1}^{n} \max_k \{\delta_{j,k} \mid C(S_k) \geq P_i\}$$

$$B_i^s = \sum_{k=1}^{m} \max_{j>i} \{\delta_{j,k} \mid C(S_k) \geq P_i\}.$$

This computation is performed by the algorithm shown in Fig. 7.11. The algorithm assumes that the task set consists of n periodic tasks that use m distinct binary semaphores. Tasks are ordered by decreasing priority, such that $P_i > P_j$ for all $i < j$. Critical sections are not nested. Notice that the blocking factor B_n is always zero, since there are no tasks with priority lower than P_n that can block τ_n. The complexity of the algorithm is $O(mn^2)$.

Note that the blocking factors derived by this algorithm are not tight. In fact, B_i^l may be computed by considering two or more critical sections guarded by the same semaphore, which for Lemma 7.4 cannot both block τ_i. Similarly, B_i^s may be computed by considering two or more critical sections belonging to the same task, which for Lemma 7.3 cannot both block τ_i. To exclude these cases, however, the complexity grows exponentially because the maximum blocking time has to be computed among all possible combinations of critical sections. An algorithm based on exhaustive search is presented in [Raj91]. It can find better bounds than those found by the algorithm presented in this section, but it has an exponential complexity.

7.6.3.1 Example

To illustrate the algorithm presented above against the exact method, consider the example shown in Table 7.1, where four tasks share three semaphores. For each task τ_i, the duration of the longest critical section among those that use the same semaphore S_k is denoted by $\delta_{i,k}$ and reported in the table. $\delta_{i,k} = 0$ means that task τ_i does not use semaphore S_k. Semaphore ceilings are indicated in parentheses.

According to the algorithm shown in Fig. 7.11, the blocking factors of the tasks are computed as follows:

Algorithm: Blocking Time Computation
Input: durations $\delta_{i,k}$ for each task τ_i and each semaphore S_k
Output: B_i for each task τ_i
// Assumes tasks are ordered by decreasing priorities

```
(1)  begin
(2)      for i := 1 to n − 1 do                  // for each task
(3)          B_i^l := 0;
(4)          for j := i + 1 to n do              // for each lower priority task
(5)              D_max := 0;
(6)              for k := 1 to m do              // for each semaphore
(7)                  if (C(S_k) ≥ P_i) and (δ_{j,k} > D_max) do
(8)                      D_max := δ_{j,k};
(9)                  end
(10)             end
(11)             B_i^l := B_i^l + D_max;
(12)         end
(13)         B_i^s := 0;
(14)         for k := 1 to m do                  // for each semaphore
(15)             D_max := 0;
(16)             for j := i + 1 to n do          // for each lower priority task
(17)                 if (C(S_k) ≥ P_i) and (δ_{j,k} > D_max) do
(18)                     D_max := δ_{j,k};
(19)                 end
(20)             end
(21)             B_i^s := B_i^s + D_max;
(22)         end
(23)         B_i := min(B_i^l, B_i^s);
(24)     end
(25)     B_n := 0;
(26) end
```

Fig. 7.11 Algorithm for computing the blocking factors

Table 7.1 Three semaphores shared by four tasks

	$S_a(P_1)$	$S_b(P_1)$	$S_c(P_2)$
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

$$B_1^l = 9 + 8 + 6 = 23$$
$$B_1^s = 8 + 9 = 17 \quad ==> \quad B_1 = 17$$

$$B_2^l = 8 + 6 = 14$$
$$B_2^s = 8 + 7 + 4 = 19 ==> \quad B_2 = 14$$

$$B_3^l = 6$$
$$B_3^s = 6 + 5 + 4 = 15 ==> \quad B_3 = 6$$

$$B_4^l = B_4^s = 0 \quad ==> \quad B_4 = 0$$

To understand why the algorithm is pessimistic, note that B_2^l is computed by adding the duration of two critical sections both guarded by semaphore S_1, which can never occur in practice.

7.6.3.2 Exact Method

To compute the worst-case blocking times using the exact method, the following steps need to be performed:

1. Identify for each task τ_i the set γ_i of critical sections that can block it, considering both direct and push-through blocking.
2. From set γ_i, compute the maximum number α_i of critical sections that can block τ_i. From Theorem 7.2, $\alpha_i = \min(l_i, s_i)$, where l_i is the number of lower-priority tasks that can block τ_i and s_i is the number of distinct semaphores that can block τ_i. Note that l_i is the number of different owners of critical sections in γ_i, whereas s_i is the number of different semaphores protecting the critical sections in γ_i.
3. From set γ_i, choose the α_i critical sections whose durations give the highest sum, avoiding considering critical sections belonging to the same task (for Lemma 7.3) and critical sections protected by the same semaphore (for Lemma 7.4). Note that, due to the constraints imposed by the two lemmas, the number of blocking critical sections could be less than α_i.
4. Finally, the blocking time B_i is computed by summing the durations of the selected critical sections in γ_i.

The results of each step of the exact method are summarized in Table 7.2. For task τ_1, the critical sections in γ_1 belong to three tasks ($l_1 = 3$) and are protected

Table 7.2 Results of the various steps of the exact method for the considered example

	γ_i	l_i	s_i	α_i	B_i
τ_1	$\{Z_{2,b}, Z_{3,a}, Z_{3,b}, Z_{4,a}, Z_{4,b}\}$	3	2	2	17
τ_2	$\{Z_{3,a}, Z_{3,b}, Z_{4,a}, Z_{4,b}, Z_{4,c}\}$	2	3	2	13
τ_3	$\{Z_{4,a}, Z_{4,b}, Z_{4,c}\}$	1	3	1	6
τ_4	$\{\}$	0	0	0	0

by only two semaphores ($s_1 = 2$), hence $\alpha_1 = min(3, 2) = 2$. From Table 7.1, we can see that the highest sum in γ_1 is given by taking $Z_{2,b}$ and $Z_{3,a}$. Hence, $B_1 = \delta_{2,b} + \delta_{3,a} = 9 + 8 = 17$. For task τ_2, the critical sections in γ_2 belong to two tasks ($l_2 = 2$) and are protected by three semaphores ($s_2 = 3$), hence $\alpha_2 = min(2, 3) = 2$. In this case, the highest sum in γ_2 is given by taking either $Z_{3,a}$ and $Z_{4,b}$, or $Z_{3,b}$ and $Z_{4,a}$. In both cases, $B_2 = 13$. For task τ_3, since $\alpha_3 = 1$, B_3 is due to the duration of a single critical section. The longest critical section in γ_3 is $Z_{4,a}$, with $\delta_{4,a} = 6$, hence, $B_3 = 6$. Finally, $B_4 = 0$.

7.6.4 Implementation Considerations

The implementation of the Priority Inheritance Protocol requires a slight modification of the kernel data structures associated with tasks and semaphores. First of all, each task must have a nominal priority and an active priority, which need to be stored in the task control block (TCB). Moreover, in order to speed up the inheritance mechanism, it is convenient that each semaphore keeps track of the task holding the lock on it. This can be done by adding in the semaphore data structure a specific field, say *holder*, for storing the identifier of the holder. In this way, a task that is blocked on a semaphore can immediately identify the task that holds its lock for transmitting its priority. Similarly, transitive inheritance can be simplified if each task keeps track of the semaphore on which it is blocked. In this case, this information has to be stored in a field, say *lock*, of the Task Control Block. Assuming that the kernel data structures are extended as described above, the primitives *pi_wait* and *pi_signal* for realizing the Priority Inheritance Protocol can be defined as follows.

pi_wait(s)

- If semaphore s is free, it becomes locked, and the name of the executing task is stored in the *holder* field of the semaphore data structure.
- If semaphore s is locked, the executing task is blocked on the s semaphore queue, the semaphore identifier is stored in the *lock* field of the TCB, and its priority is inherited by the task that holds s. If such a task is blocked on another semaphore, the transitivity rule is applied. Then, the ready task with the highest priority is assigned to the processor.

pi_signal(s)

- If the queue of semaphore s is empty (i.e., no tasks are blocked on s), s is unlocked.
- If the queue of semaphore s is not empty, the highest-priority task in the queue is awakened, its identifier is stored in $s.holder$, the active priority of the executing task is updated, and the ready task with the highest priority is assigned to the processor.

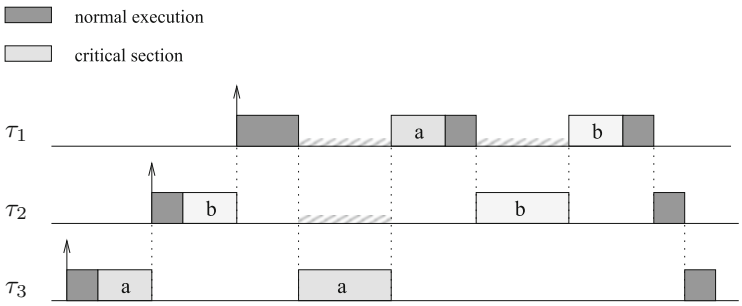


Fig. 7.12 Example of chained blocking

7.6.5 Unsolved Problems

Although the Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration for a task can still be substantial because a chain of blocking can be formed. Another problem is that the protocol does not prevent deadlocks.

7.6.5.1 Chained Blocking

Consider three tasks τ_1, τ_2, and τ_3 with decreasing priorities that share two semaphores S_a and S_b. Suppose that τ_1 needs to sequentially access S_a and S_b, τ_2 S_b, and τ_3 S_a. Also suppose that τ_3 locks S_a, and it is preempted by τ_2 within its critical section. Similarly, τ_2 locks S_b and it is preempted by τ_1 within its critical section. The example is shown in Fig. 7.12. In this situation, when attempting to use its resources, τ_1 is blocked for the duration of two critical sections, once to wait τ_3 to release S_a and then to wait τ_2 to release S_b. This is called a *chained blocking*. In the worst case, if τ_1 accesses n distinct semaphores that have been locked by n lower-priority tasks, τ_1 will be blocked for the duration of n critical sections.

7.6.5.2 Deadlock

Consider two tasks that use two semaphores in a nested fashion but in reverse order, as illustrated in Fig. 7.13. Now suppose that, at time t_1, τ_2 locks semaphore S_b and enters its critical section. At time t_2, τ_1 preempts τ_2 before it can lock S_a. At time t_3, τ_1 locks S_a, which is free, but then is blocked on S_b at time t_4. At this time, τ_2 resumes and continues the execution at the priority of τ_1. Priority inheritance does not prevent a deadlock, which occurs at time t_5, when τ_2 attempts to lock S_a. Notice, however, that the deadlock does not depend on the Priority Inheritance Protocol but

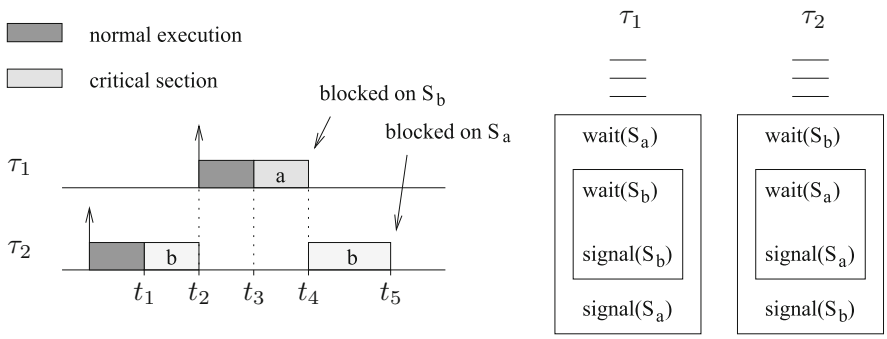


Fig. 7.13 Example of deadlock

is caused by an erroneous use of semaphores. In this case, the deadlock problem can be solved by imposing a total ordering on the semaphore accesses.

7.7 Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) was introduced by Sha, Rajkumar, and Lehoczky [SRL90] to bound the priority inversion phenomenon and prevent the formation of deadlocks and chained blocking.

The basic idea of this method is to extend the Priority Inheritance Protocol with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it. This means that, once a task enters its first critical section, it can never be blocked by lower-priority tasks until its completion.

In order to realize this idea, each semaphore is assigned a *priority ceiling* equal to the highest priority of the tasks that can lock it. Then, a task τ_i is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than τ_i.

7.7.1 Protocol Definition

The Priority Ceiling Protocol can be defined as follows:

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the highest priority of the tasks that can lock it. Note that $C(S_k)$ is a static value that can be computed off-line:

$$C(S_k) \stackrel{\text{def}}{=} \max_i \{P_i \mid S_k \in \sigma_i\}. \tag{7.13}$$

- Let τ_i be the task with the highest priority among all tasks ready to run; thus, τ_i is assigned the processor.
- Let S^* be the semaphore with the highest ceiling among all the semaphores currently locked by tasks other than τ_i and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k, τ_i must have a priority higher than $C(S^*)$. If $P_i \leq C(S^*)$, the lock on S_k is denied and τ_i is said to be blocked on semaphore S^* by the task that holds the lock on S^*.
- When a task τ_i is blocked on a semaphore, it transmits its priority to the task, say τ_j, that holds that semaphore. Hence, τ_j resumes and executes the rest of its critical section with the priority of τ_i. Task τ_j is said to *inherit* the priority of τ_i.
- In general, a task inherits the highest priority of the tasks blocked by it. That is, at every instant,

$$p_j(R_k) = \max\{P_j, \max_h \{P_h | \tau_h \text{ is blocked on } R_k\}\}. \tag{7.14}$$

- When τ_j exits a critical section, it unlocks the semaphore, and the highest-priority task, if any, blocked on that semaphore is awakened. Moreover, the active priority of τ_j is updated as follows: if no other tasks are blocked by τ_j, p_j is set to the nominal priority P_j; otherwise, it is set to the highest priority of the tasks blocked by τ_j, according to Eq. (7.14).
- Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2.

7.7.1.1 Example

In order to illustrate the Priority Ceiling Protocol, consider three tasks τ_1, τ_2, and τ_3 having decreasing priorities. The highest-priority task τ_1 sequentially accesses two critical sections guarded by semaphores S_A and S_B; task τ_2 accesses only a critical section guarded by semaphore S_C, whereas task τ_3 uses semaphore S_C and then makes a nested access to S_B. From tasks' resource requirements, all semaphores are assigned the following priority ceilings:

$$\begin{cases} C(S_A) = P_1 \\ C(S_B) = P_1 \\ C(S_C) = P_2. \end{cases}$$

Now suppose that events evolve as illustrated in Fig. 7.14.

- At time t_0, τ_3 is activated and, since it is the only task ready to run, it starts executing and later locks semaphore S_C.

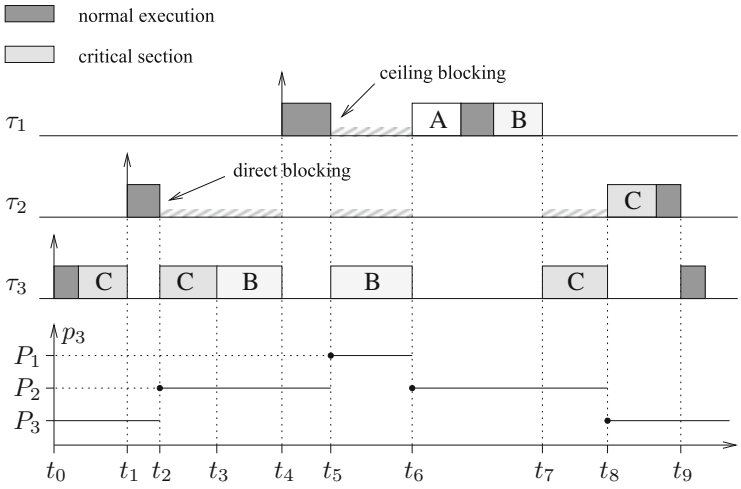


Fig. 7.14 Example of Priority Ceiling Protocol

- At time t_1, τ_2 becomes ready and preempts τ_3.
- At time t_2, τ_2 attempts to lock S_C, but it is blocked by the protocol because P_2 is not greater than $C(S_C)$. Then, τ_3 inherits the priority of τ_2 and resumes its execution.
- At time t_3, τ_3 successfully enters its nested critical section by locking S_B. Note that τ_3 is allowed to lock S_B because no semaphores are locked by other tasks.
- At time t_4, while τ_3 is executing at a priority $p_3 = P_2$, τ_1 becomes ready and preempts τ_3 because $P_1 > p_3$.
- At time t_5, τ_1 attempts to lock S_A, which is not locked by any task. However, τ_1 is blocked by the protocol because its priority is not higher than $C(S_B)$, which is the highest ceiling among all semaphores currently locked by the other tasks. Since S_B is locked by τ_3, τ_3 inherits the priority of τ_1 and resumes its execution.
- At time t_6, τ_3 exits its nested critical section, unlocks S_B, and, since τ_1 is awakened, τ_3 returns to priority $p_3 = P_2$. At this point, $P_1 > C(S_C)$; hence, τ_1 preempts τ_3 and executes until completion.
- At time t_7, τ_1 is completed, and τ_3 resumes its execution at a priority $p_3 = P_2$.
- At time t_8, τ_3 exits its outer critical section, unlocks S_C, and, since τ_2 is awakened, τ_3 returns to its nominal priority P_3. At this point, τ_2 preempts τ_3 and executes until completion.
- At time t_9, τ_2 is completed; thus, τ_3 resumes its execution.

Notice that the Priority Ceiling Protocol introduces a third form of blocking, called *ceiling blocking*, in addition to direct blocking and push-through blocking, caused by the Priority Inheritance Protocol. This is necessary for avoiding deadlock and chained blocking. In the previous example, a ceiling blocking is experienced by task τ_1 at time t_5.

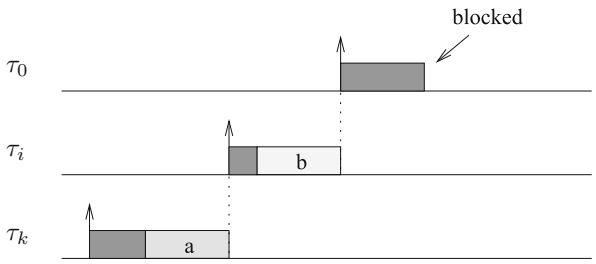


Fig. 7.15 An absurd situation that cannot occur under the Priority Ceiling Protocol

7.7.2 Properties of the Protocol

The main properties of the Priority Ceiling Protocol are presented in this section. They are used to analyze the schedulability and compute the maximum blocking time of each task.

Lemma 7.6 *If a task τ_k is preempted within a critical section Z_a by a task τ_i that enters a critical section Z_b, then, under the Priority Ceiling Protocol, τ_k cannot inherit a priority higher than or equal to that of task τ_i until τ_i completes.*

Proof If τ_k inherits a priority higher than or equal to that of task τ_i before τ_i completes, there must exist a task τ_0 blocked by τ_k, such that $P_0 \geq P_i$. This situation is shown in Fig. 7.15. However, this leads to the contradiction that τ_0 cannot be blocked by τ_k. In fact, since τ_i enters its critical section, its priority must be higher than the maximum ceiling C^* of the semaphores currently locked by all lower-priority tasks. Hence, $P_0 \geq P_i > C^*$. But since $P_0 > C^*$, τ_0 cannot be blocked by τ_k, and the lemma follows. □

Lemma 7.7 *The Priority Ceiling Protocol prevents transitive blocking.*

Proof Suppose that a transitive block occurs; that is, there exist three tasks τ_1, τ_2, and τ_3, with decreasing priorities, such that τ_3 blocks τ_2 and τ_2 blocks τ_1. By the transitivity of the protocol, τ_3 will inherit the priority of τ_1. However, this contradicts Lemma 7.6, which shows that τ_3 cannot inherit a priority higher than or equal to P_2. Thus, the lemma follows. □

Theorem 7.3 *The Priority Ceiling Protocol prevents deadlocks.*

Proof Assuming that a task cannot deadlock by itself, a deadlock can only be formed by a cycle of tasks waiting for each other, as shown in Fig. 7.16. In this situation, however, by the transitivity of the protocol, task τ_n would inherit the priority of τ_1, which is assumed to be higher than P_n. This contradicts Lemma 7.6, and hence the theorem follows. □

Theorem 7.4 (Sha-Rajkumar-Lehoczky) *Under the Priority Ceiling Protocol, a task τ_i can be blocked for at most the duration of one critical section.*

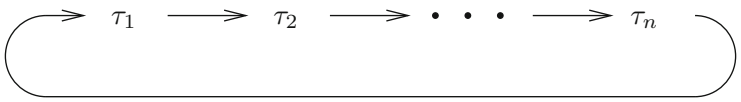


Fig. 7.16 Deadlock among n tasks

Proof Suppose that τ_i is blocked by two lower-priority tasks τ_1 and τ_2, where $P_2 < P_1 < P_i$. Let τ_2 enter its blocking critical section first, and let C_2^* be the highest-priority ceiling among all the semaphores locked by τ_2. In this situation, if task τ_1 enters its critical section, we must have that $P_1 > C_2^*$. Moreover, since we assumed that τ_i can be blocked by τ_2, we must have that $P_i \leq C_2^*$. This means that $P_i \leq C_2^* < P_1$. This contradicts the assumption that $P_i > P_1$. Thus, the theorem follows. □

7.7.3 Blocking Time Computation

The evaluation of the maximum blocking time for each task can be computed based on the result of Theorem 7.4. According to this theorem, a task τ_i can be blocked for at most the duration of the longest critical section among those that can block τ_i. The set of critical sections that can block a task τ_i is identified by the following lemma.

Lemma 7.8 *Under the Priority Ceiling Protocol, a critical section $z_{j,k}$ (belonging to task τ_j and guarded by semaphore S_k) can block a task τ_i only if $P_j < P_i$ and $C(S_k) \geq P_i$.*

Proof Clearly, if $P_j \geq P_i$, τ_i cannot preempt τ_j and hence cannot be blocked on $z_{j,k}$. Now assume $P_j < P_i$ and $C(S_k) < P_i$, and suppose that τ_i is blocked on $z_{j,k}$. We show that this assumption leads to a contradiction. In fact, if τ_i is blocked by τ_j, its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by tasks other than τ_i. Thus, we have that $C(S_k) < P_i \leq C^*$. On the other hand, since C^* is the maximum ceiling among all semaphores currently locked by tasks other than τ_i, we have that $C^* \geq C(S_k)$, which leads to a contradiction and proves the lemma. □

Using the result of Lemma 7.8, we can say that a task τ_i can only be blocked by critical sections belonging to lower-priority tasks with a resource ceiling higher than or equal to P_i. That is,

$$\gamma_i = \{Z_{j,k} \mid (P_j < P_i) \text{ and } C(R_k) \geq P_i\}. \tag{7.15}$$

And since τ_i can be blocked at most once, the maximum blocking time τ_i can suffer is given by the duration of the longest critical section among those that can block τ_i.

Table 7.3 Three semaphores shared by four tasks

	$S_a(P_1)$	$S_b(P_1)$	$S_c(P_2)$
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

That is,

$$B_i = \max_{j,k}\{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\}. \tag{7.16}$$

Consider the same example illustrated for the Priority Inheritance Protocol, reported in Table 7.3 for simplicity. According to Eq. (7.15), we have

$$\begin{aligned}
\gamma_1 &= \{Z_{2b}, Z_{3a}, Z_{3b}, Z_{4a}, Z_{4b}\} \\
\gamma_2 &= \{Z_{3a}, Z_{3b}, Z_{4a}, Z_{4b}, Z_{4c}\} \\
\gamma_3 &= \{Z_{4a}, Z_{4b}, Z_{4c}\} \\
\gamma_4 &= \{\}
\end{aligned}$$

Hence, based on Eq. (7.16), tasks blocking factors are computed as follows:

$$\begin{cases}
B_1 = \max(9, 8, 7, 6, 5) = 9 \\
B_2 = \max(8, 7, 6, 5, 4) = 8 \\
B_3 = \max(6, 5, 4) = 6 \\
B_4 = 0.
\end{cases}$$

7.7.4 Implementation Considerations

The major implication of the Priority Ceiling Protocol in the kernel data structures is that semaphore queues are no longer needed, since the tasks blocked by the protocol can be kept in the ready queue. In particular, whenever a task τ_i is blocked by the protocol on a semaphore S_k, the task τ_h that holds S_k inherits the priority of τ_i, and it is assigned to the processor, whereas τ_i returns to the ready queue. As soon as τ_h unlocks S_k, p_h is updated, and, if p_h becomes less than the priority of the first ready task, a context switch is performed.

To implement the Priority Ceiling Protocol, each semaphore S_k has to store the identifier of the task that holds the lock on S_k and the ceiling of S_k. Moreover, an additional field for storing the task active priority has to be reserved in the task control block. It is also convenient to have a field in the task control block for

storing the identifier of the semaphore on which the task is blocked. Finally, the implementation of the protocol can be simplified if the system also maintains a list of currently locked semaphores, order by decreasing priority ceilings. This list is useful for computing the maximum priority ceiling that a task has to overcome to enter a critical section and for updating the active priority of tasks at the end of a critical section.

If the kernel data structures are extended as described above, the primitives *pc_wait* and *pc_signal* for realizing the Priority Ceiling Protocol can be defined as follows:

pc_wait(s)

- Find the semaphore S^* having the maximum ceiling C^* among all the semaphores currently locked by tasks other than the one in execution (τ_{exe}).
- If $p_{exe} \leq C^*$, transfer P_{exe} to the task that holds S^*, insert τ_{exe} in the ready queue, and execute the ready task (other than τ_{exe}) with the highest priority.
- If $p_{exe} > C^*$, or whenever s is unlocked, lock semaphore s, add s in the list of currently locked semaphores, and store τ_{exe} in $s.holder$.

pc_signal(s)

- Extract s from the list of currently locked semaphores.
- If no other tasks are blocked by τ_{exe}, set $p_{exe} = P_{exe}$, else set p_{exe} to the highest priority of the tasks blocked by τ_{exe}.
- Let p^* be the highest priority among the ready tasks. If $p_{exe} < p^*$, insert τ_{exe} in the ready queue, and execute the ready task (other than τ_{exe}) with the highest priority.

7.8 Stack Resource Policy

The Stack Resource Policy (SRP) is a technique proposed by Baker [Bak91] for accessing shared resources. It extends the Priority Ceiling Protocol (PCP) in three essential points:

1. It allows the use of multi-unit resources.
2. It supports dynamic priority scheduling.
3. It allows the sharing of runtime stack-based resources.

From a scheduling point of view, the essential difference between the PCP and the SRP is on the time at which a task is blocked. Whereas under the PCP, a task is blocked at the time it makes its first resource request, under the SRP a task is blocked at the time it attempts to preempt. This early blocking slightly reduces concurrency but saves unnecessary context switches, simplifies the implementation of the protocol, and allows the sharing of runtime stack resources.

7.8.1 Definitions

Before presenting the formal description of the SRP, we introduce the following definitions.

7.8.1.1 Priority

Each task τ_i is assigned a priority p_i that indicates the importance (i.e., the urgency) of τ_i with respect to the other tasks in the system. Priorities can be assigned to tasks either statically or dynamically. At any time t, $p_a > p_b$ means that the execution of τ_a is more important than that of τ_b; hence, τ_b can be delayed in favor of τ_a. For example, priorities can be assigned to tasks based on Rate Monotonic (RM) or Earliest Deadline First (EDF).

7.8.1.2 Preemption Level

Besides a priority p_i, a task τ_i is also characterized by a *preemption level* π_i. The preemption level is a static parameter, assigned to a task at its creation time and associated with all instances of that task. The essential property of preemption levels is that a task τ_a can preempt another task τ_b only if $\pi_a > \pi_b$. This is also true for priorities. Hence, the reason for distinguishing preemption levels from priorities is that preemption levels are fixed values that can be used to predict potential blocking also in the presence of dynamic priority schemes. The general definition of preemption level used to prove all properties of the SRP requires that

> if τ_a arrives after τ_b and τ_a has higher priority than τ_b, then τ_a must have a higher preemption level than τ_b.

Under EDF scheduling, the previous condition is satisfied if preemption levels are ordered inversely with respect to the order of relative deadlines; that is,

$$\pi_i > \pi_j \iff D_i < D_j.$$

To better illustrate the difference between priorities and preemption levels, consider the example shown in Fig. 7.17. Here we have two tasks τ_1 and τ_2, with relative deadlines $D_1 = 10$ and $D_2 = 5$, respectively. Being $D_2 < D_1$, we define $\pi_1 = 1$ and $\pi_2 = 2$. Since $\pi_1 < \pi_2$, τ_1 can never preempt τ_2; however, τ_1 may have a priority higher than that of τ_2. In fact, under EDF, the priority of a task is dynamically assigned based on its absolute deadline. For example, in the case illustrated in Fig. 7.17a, the absolute deadlines are such that $d_2 < d_1$; hence, τ_2 will have higher priority than τ_1. On the other hand, as shown in Fig. 7.17b, if τ_2 arrives a time $r_1 + 6$, absolute deadlines are such that $d_2 > d_1$; hence, τ_1 will have higher priority than τ_2. Notice that, although τ_1 has a priority higher than τ_2, τ_2 cannot be

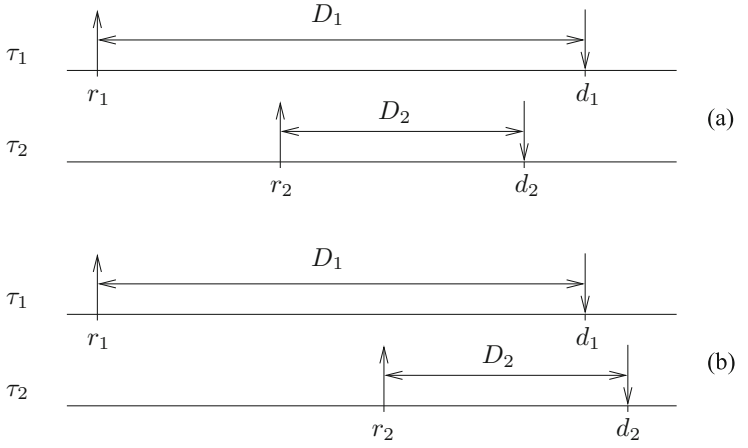


Fig. 7.17 Although $\pi_2 > \pi_1$, under EDF p_2 can be higher than p_1 (**a**) or lower than p_1 (**b**)

preempted. This happens because, when $d_1 < d_2$ and $D_1 > D_2$, τ_1 always starts before τ_2; thus, it does not need to preempt τ_2.

In the following, it is assumed that tasks are ordered by decreasing preemption levels, so that $i < j \iff \pi_i > \pi_j$. This also means that $D_1 < D_2 < \ldots < D_n$.

7.8.1.3 Resource Units

Each resource R_k is allowed to have N_k units that can be concurrently accessed by multiple tasks. This notion generalizes the classical single-unit resource, which can be accessed by one task at the time under mutual exclusion. A resource with N_k units can be concurrently accessed by N_k different tasks, if each task requires one unit. In general, n_k denotes the number of currently available units for R_k, meaning that $N_k - n_k$ units are locked. If $n_k = 0$, a task requiring three units of R_k is blocked until n_k becomes greater than or equal to 3.

7.8.1.4 Resource Requirements

When entering a critical section $z_{i,k}$ guarded by a multi-unit semaphore S_k, a task τ_i must specify the number of units it needs by calling a $wait(S_k, r)$, where r is the number of requested units. When exiting the critical section, τ_i must call a $signal(S_k)$, which releases all the r units.

The maximum number of units that can be simultaneously requested by τ_i to R_k is denoted by $\mu_i(R_k)$, which can be derived off-line by analyzing the task code. It is clear that, if τ_i requires more units than are available, that is, if $\mu_i(R_k) > n_k$, then τ_i blocks until n_k becomes greater than or equal to $\mu_i(R_k)$.

Table 7.4 Task parameters and resource requirements

	D_i	π_i	$\mu_i(R_1)$	$\mu_i(R_2)$	$\mu_i(R_3)$
τ_1	5	3	1	0	1
τ_2	10	2	2	1	3
τ_3	20	1	3	1	1

Table 7.5 Resource ceilings as a function of the number of available units. Dashes identify impossible cases

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_1	0	1	2	3
R_2	-	-	0	2
R_3	0	2	2	3

7.8.1.5 Resource Ceiling

Each resource R_k (or semaphore S_k) is assigned a *ceiling* $C_{Rk}(n_k)$ equal to the highest preemption level of the tasks that could be blocked on R_k if issuing their maximum request. Hence, the ceiling of a resource is a dynamic value, which is a function of the units of R_k that are currently available. That is,

$$C_{R_k}(n_k) = \max\{\pi_i \mid \mu_i(R_k) > n_k\}.$$

If all units of R_k are available, that is, if $n_k = N_k$, then $C_{R_k}(N_k) = 0$.

To better clarify this concept, consider the following example, where three tasks (τ_1, τ_2, τ_3) share three resources (R_1, R_2, R_3), consisting of three, one, and three units, respectively. All tasks parameters—relative deadlines, preemption levels, and resource requirements—are shown in Table 7.4.

Based on these requirements, the current ceilings of the resources as a function of the number n_k of available units are reported in Table 7.5 (dashes identify impossible cases).

Let us compute, for example, the ceiling of resource R_1 when only two units (out of three) are available. From Table 7.4, we see that the only task that could be blocked in this condition is τ_3 because it requires three units of R_1; hence, $C_{R1}(2) = \pi_3 = 1$. If only one unit of R_1 is available, the tasks that could be blocked are τ_2 and τ_3; hence, $C_{R1}(1) = \max(\pi_2, \pi_3) = 2$. Finally, if none of the units of R_1 is available, all three tasks could be blocked on R_1; hence, $C_{R1}(0) = \max(\pi_1, \pi_2, \pi_3) = 3$.

7.8.1.6 System Ceiling

The resource access protocol adopted in the SRP also requires a *system ceiling*, Π_s, defined as the maximum of the current ceilings of all the resources, that is,

$$\Pi_s = \max_k\{C_{R_k}\}.$$

Notice that Π_s is a dynamic parameter that can change every time a resource is accessed or released by a task.

7.8.2 Protocol Definition

The key idea of the SRP is that, when a task needs a resource that is not available, it is blocked at the time it attempts to preempt, rather than later. Moreover, to prevent multiple priority inversions, a task is not allowed to start until the resources currently available are sufficient to meet the maximum requirement of every task that could preempt it. Using the definitions introduced in the previous paragraph, this is achieved by the following preemption test:

> **SRP Preemption Test**: A task is not permitted to preempt until its priority is the highest among those of all the tasks ready to run, and its preemption level is higher than the system ceiling.

If the ready queue is ordered by decreasing priorities, the preemption test can be simply performed by comparing the preemption level $\pi(\tau)$ of the ready task with the highest priority (the one at the head of the queue) with the system ceiling. If $\pi(\tau) > \Pi_s$, task τ is executed; otherwise it is kept in the ready queue until Π_s becomes less than $\pi(\tau)$. The condition $\pi(\tau) > \Pi_s$ has to be tested every time Π_s may decrease; that is, every time a resource is released.

7.8.2.1 Observations

The implications that the use of the SRP has on tasks' execution can be better understood through the following observations:

- Passing the preemption test for task τ ensures that the resources that are currently available are sufficient to satisfy the maximum requirement of task τ and the maximum requirement of every task that could preempt τ. This means that, once τ starts executing, it will never be blocked for resource contention.
- Although the preemption test for a task τ is performed before τ starts executing, resources are not allocated at this time but only when requested.
- A task can be blocked by the preemption test even though it does not require any resource. This is needed to avoid unbounded priority inversion.
- Blocking at preemption time, rather than at access time, introduces more pessimism and could create unnecessary blocking (as under HLP), but it decreases the number of context switches, reduces the runtime overhead, and simplifies the implementation of the protocol.
- The preemption test has the effect of imposing priority inheritance; that is, an executing task that holds a resource modifies the system ceiling and resists preemption as though it inherits the priority of any tasks that might need that

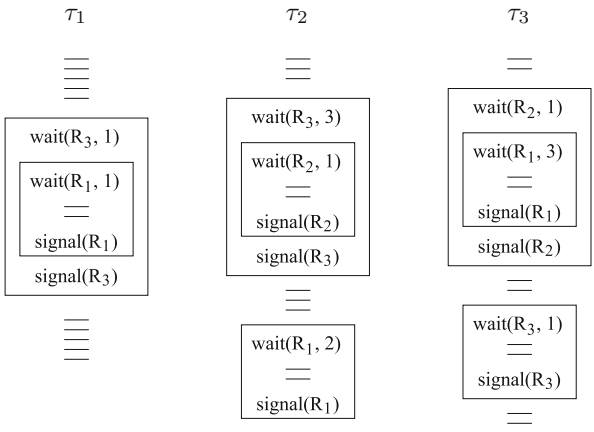


Fig. 7.18 Structure of the tasks in the SRP example

resource. Note that this effect is accomplished without modifying the priority of the task.

7.8.2.2 Example

In order to illustrate how the SRP works, consider the task set already described in Table 7.4. The structure of the tasks is shown in Fig. 7.18, where $wait(R_k, r)$ denotes the request of r units of resource R_k and $signal(R_k)$ denotes their release. The current ceilings of the resources have already been shown in Table 7.5, and a possible EDF schedule for this task set is depicted in Fig. 7.19. In this figure, the fourth timeline reports the variation of the system ceiling, whereas the numbers along the schedule denote resource indexes.

At time t_0, τ_3 starts executing and the system ceiling is zero because all resources are completely available. When τ_3 enters its first critical section, it takes the only unit of R_2; thus, the system ceiling is set to the highest preemption level among the tasks that could be blocked on R_2 (see Table 7.5); that is, $\Pi_s = \pi_2 = 2$. As a consequence, τ_2 is blocked by the preemption test and τ_3 continues to execute. Note that when τ_3 enters its nested critical section (taking all units of R_1), the system ceiling is raised to $\Pi_s = \pi_1 = 3$. This causes τ_1 to be blocked by the preemption test.

As τ_3 releases R_1 (at time t_2), the system ceiling becomes $\Pi_s = 2$; thus, τ_1 preempts τ_3 and starts executing. Note that, once τ_1 is started, it is never blocked during its execution, because the condition $\pi_1 > \Pi_s$ guarantees that all the resources needed by τ_1 are available. As τ_1 terminates, τ_3 resumes the execution and releases resource R_2. As R_2 is released, the system ceiling returns to zero and τ_2 can preempt τ_3. Again, once τ_2 is started, all the resources it needs are available; thus, τ_2 is never blocked.

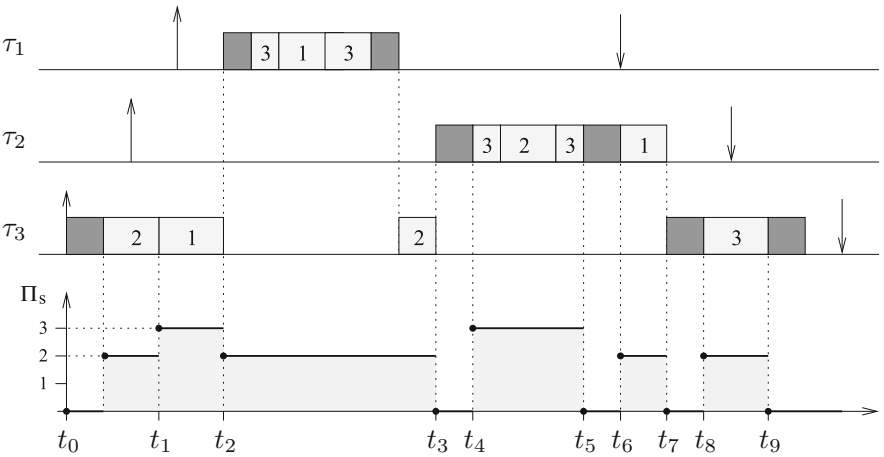


Fig. 7.19 Example of a schedule under EDF and SRP. Numbers on tasks execution denote the resource indexes

7.8.3 Properties of the Protocol

The main properties of the Stack Resource Policy are presented in this section. They will be used to analyze the schedulability and compute the maximum blocking time of each task.

Lemma 7.9 *If the preemption level of a task τ is greater than the current ceiling of a resource R, then there are sufficient units of R available to:*

1. *Meet the maximum requirement of τ and*
2. *Meet the maximum requirement of every task that can preempt τ.*

Proof Assume $\pi(\tau) > C_R$, but suppose that the maximum request of τ for R cannot be satisfied. Then, by definition of current ceiling of a resource, we have $C_R \geq \pi(\tau)$, which is a contradiction.

Assume $\pi(\tau) > C_R$, but suppose that there exists a task τ_H that can preempt τ such that the maximum request of τ_H for R cannot be satisfied. Since τ_H can preempt τ, it must be $\pi(\tau_H) > \pi(\tau)$. Moreover, since the maximum request of τ_H for R cannot be satisfied, by definition of current ceiling of a resource, we have $C_R \geq \pi(\tau_H) > \pi(\tau)$, which contradicts the assumption. □

Theorem 7.5 (Baker) *If no task τ is permitted to start until $\pi(\tau) > \Pi_s$, then no task can be blocked after it starts.*

Proof Let N be the number of tasks that can preempt a task τ and assume that no task is permitted to start until its preemption level is greater than Π_s. The thesis will be proved by induction on N.

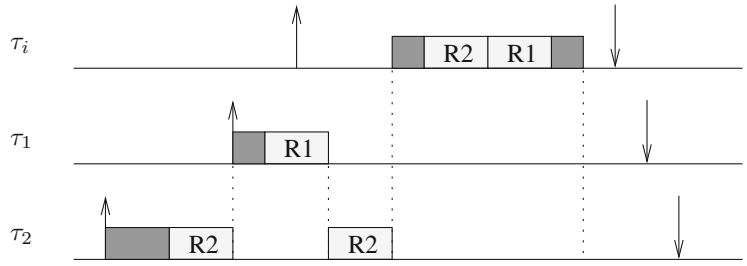


Fig. 7.20 An absurd situation that cannot occur under SRP

If $N = 0$, there are no tasks that can preempt τ. If τ is started when $\pi(\tau) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by τ are available when τ preempts; hence, τ will execute to completion without blocking.

If $N > 0$, suppose that τ is preempted by τ_H. If τ_H is started when $\pi(\tau_H) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by τ_H are available when τ_H preempts. Since any task that preempts τ_H also preempts τ, the induction hypothesis guarantees that τ_H executes to completion without blocking, as will any task that preempts τ_H, transitively. When all the tasks that preempted τ complete, τ can resume its execution without blocking, since the higher-priority tasks released all resources and when τ started the resources available were sufficient to meet the maximum request of τ. □

Theorem 7.6 (Baker) *Under the Stack Resource Policy, a task τ_i can be blocked for at most the duration of one critical section.*

Proof Suppose that τ_i is blocked for the duration of two critical sections shared with two lower-priority tasks, τ_1 and τ_2. Without loss of generality, assume $\pi_2 < \pi_1 < \pi_i$. This can happen only if τ_1 and τ_2 hold two different resources (such as R_1 and R_2) and τ_2 is preempted by τ_1 inside its critical section. This situation is depicted in Fig. 7.20. This immediately yields to a contradiction. In fact, since τ_1 is not blocked by the preemption test, we have $\pi_1 > \Pi_s$. On the other hand, since τ_i is blocked, we have $\pi_i \leq \Pi_s$. Hence, we obtain that $\pi_i < \pi_1$, which contradicts the assumption. □

Theorem 7.7 (Baker) *The Stack Resource Policy prevents deadlocks.*

Proof By Theorem 7.5, a task cannot be blocked after it starts. Since a task cannot be blocked while holding a resource, there can be no deadlock. □

7.8.4 Blocking Time Computation

The maximum blocking time that a task can experience with the SRP is the same as the one that can be experienced with the Priority Ceiling Protocol. Theorem 7.6, in fact, guarantees that under the SRP a task τ_i can be blocked for at most the duration of one critical section among those that can block τ_i. Lemma 7.8, proved for the PCP, can be easily extended to the SRP; thus a critical section $Z_{j,k}$ belonging to task τ_j and guarded by semaphore S_k can block a task τ_i only if $\pi_j < \pi_i$ and $\max(C_{S_k}) \geq \pi_i$. Notice that, under the SRP, the ceiling of a semaphore is a dynamic variable, so we have to consider its maximum value, that is, the one corresponding to a number of units currently available equal to zero.

Hence, a task τ_i can only be blocked by critical sections belonging to tasks with preemption level lower than π_i and guarded by semaphores with maximum ceiling higher than or equal to π_i. That is,

$$\gamma_i = \{Z_{j,k} \mid (\pi_j < \pi_i) \text{ and } C_{S_k}(0) \geq \pi_i\}. \tag{7.17}$$

And since τ_i can be blocked at most once, the maximum blocking time τ_i can suffer is given by the duration of the longest critical section among those that can block τ_i. That is,

$$B_i = \max_{j,k}\{\delta_{j,k} \mid Z_{j,k} \in \gamma_i\}. \tag{7.18}$$

7.8.5 Sharing Runtime Stack

Another interesting implication deriving from the use of the SRP is that it supports stack sharing among tasks. This is particularly convenient for those applications consisting of a large number of tasks, dedicated to acquisition, monitoring, and control activities. In conventional operating systems, each process must have a private stack space, sufficient to store its context (i.e., the content of the CPU registers) and its local variables. A problem with these systems is that, if the number of tasks is large, a great amount of memory may be required for the stacks of all the tasks.

For example, consider four tasks τ_1, τ_2, τ_3, and τ_4, with preemption levels 1, 2, 2, and 3, respectively (3 being the highest preemption level). Figure 7.21 illustrates a possible evolution of the stacks, assuming that each task is allocated its own stack space, equal to its maximum requirement. At time t_1, τ_1 starts executing; τ_2 preempts at time t_2 and completes at time t_3, allowing τ_1 to resume. At time t_4, τ_1 is preempted by τ_3, which in turn is preempted by τ_4 at time t_5. At time t_6, τ_4 completes and τ_3 resumes. At time t_7, τ_3 completes and τ_1 resumes.

Note that the top of each process stack varies during the process execution, while the storage region reserved for each stack remains constant and corresponds to the

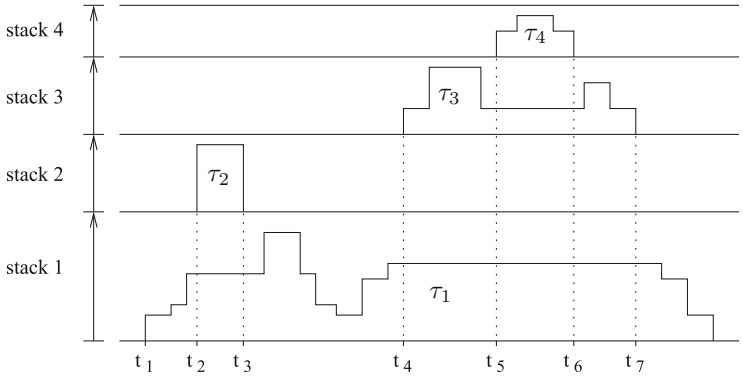


Fig. 7.21 Possible evolution with one stack per task

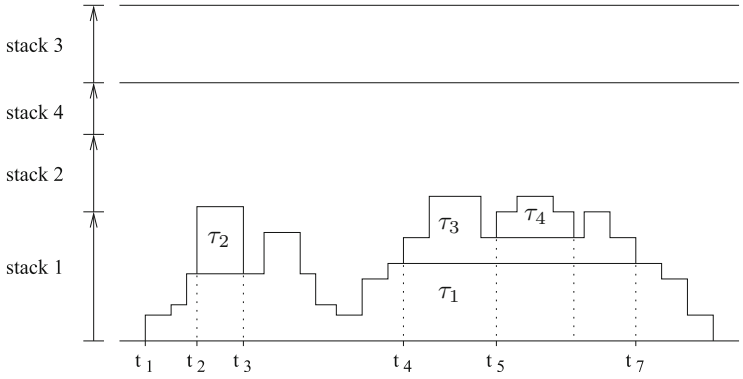


Fig. 7.22 Possible evolution with a single stack for all tasks

distance between two horizontal lines. In this case, the total storage area that must be reserved for the application is equal to the sum of the stack regions dedicated to each process.

However, if all tasks are independent or use the SRP to access shared resources, then they can share a single stack space. In this case, when a task τ is preempted by a task τ', τ maintains its stack, and the stack of τ' is allocated immediately above that of τ. Figure 7.22 shows a possible evolution of the previous task set when a single stack is allocated to all tasks.

Under the SRP, stack overlapping without interpenetration is a direct consequence of Theorem 7.5. In fact, since a task τ can never be blocked once started, its stack can never be penetrated by the ones belonging to tasks with lower preemption levels, which can resume only after τ is completed.

Note that the stack space between the two upper horizontal lines (which is equivalent to the minimum stack between τ_2 and τ_3) is no longer needed, since τ_2 and τ_3 have the same preemption level, so they can never occupy stack space at

the same time. In general, the higher the number of tasks with the same preemption level, the larger stack saving.

For example, consider an application consisting of 100 tasks distributed on 10 preemption levels, with 10 tasks for each level, and suppose that each task needs up to 10 Kbytes of stack space. Using a stack per task, 1000 Kbytes would be required. On the contrary, using a single stack, only 100 Kbytes would be sufficient, since no more than one task per preemption level could be active at one time. Hence, in this example, we would save 900 Kbytes, that is, 90%. In general, when tasks are distributed on k preemption levels, the space required for a single stack is equal to the sum of the largest request on each level.

7.8.6 Implementation Considerations

The implementation of the SRP is similar to that of the PCP, but the locking operations (srp_wait and srp_signal) are simpler, since a task can never be blocked when attempting to lock a semaphore. When there are no sufficient resources available to satisfy the maximum requirement of a task, the task is not permitted to preempt and is kept in the ready queue.

To simplify the preemption test, all the ceilings of the resources (for any number of available units) can be precomputed and stored in a table. Moreover, a stack can be used to keep track of the system ceiling. When a resource R is allocated, its current state, n_R, is updated and, if $C_R(n_R) > \Pi_s$, then Π_s is set to $C_R(n_R)$. The old values of n_R and Π_s are pushed onto the stack. When resource R is released, the values of Π_s and n_R are restored from the stack. If the restored system ceiling is lower than the previous value, the preemption test is executed on the ready task with the highest priority to check whether it can preempt. If the preemption test is passed, a context switch is performed; otherwise, the current task continues its execution.

7.9 Schedulability Analysis

This section explains how to verify the feasibility of a periodic task set in the presence of shared resources. All schedulability tests presented in Chap. 4 for independent tasks can be extended to include blocking terms, whose values depend on the specific concurrency control protocol adopted in the schedule.

In general, all the extended tests guarantee one task τ_i at the time, by inflating its computation time C_i by the blocking factor B_i. In addition, all the guarantee tests that were necessary and sufficient under preemptive scheduling become only sufficient in the presence of blocking factors, since blocking conditions are derived in worst-case scenarios that differ for each task and could never occur simultaneously.

Liu and Layland Test for Rate Monotonic A set of periodic tasks with blocking factors and relative deadlines equal to periods is schedulable by RM if

$$\forall i = 1, \ldots, n \quad \sum_{h:P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1). \tag{7.19}$$

Liu and Layland Test for EDF A set of periodic tasks with blocking factors and relative deadlines equal to periods is schedulable by EDF if

$$\forall i = 1, \ldots, n \quad \sum_{h:P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq 1. \tag{7.20}$$

Hyperbolic Test Using the hyperbolic bound, a task set with blocking factors and relative deadlines equal to periods is schedulable by RM if

$$\forall i = 1, \ldots, n \quad \prod_{h:P_h > P_i} \left(\frac{C_h}{T_h} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2. \tag{7.21}$$

Response Time Analysis Under blocking conditions, the response time of a generic task τ_i with a fixed priority can be computed by the following recurrent relation:

$$\begin{cases} R_i^{(0)} &= C_i + B_i \\ R_i^{(s)} &= C_i + B_i + \displaystyle\sum_{h:P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{7.22}$$

Workload Analysis Similarly, using the workload analysis, a task set with blocking factors is schedulable by a fixed-priority assignment if

$$\forall i = 1, \ldots, n \quad \exists\, t \in \mathcal{TS} : B_i + W_i(t) \leq t. \tag{7.23}$$

Processor Demand Criterion The Processor Demand Criterion in the presence of blocking terms has been extended by Baruah [Bar06], using the concept of *Blocking Function* $B(L)$, defined as the largest amount of time for which a task with relative deadline $\leq L$ may be blocked by a task with relative deadline $> L$.

If δ_{jh} denotes the maximum length of time for which τ_j holds a resource that is also needed by τ_h, the blocking function can be computed as follows:

$$B(L) = \max \left\{ \delta_{j,h} \mid (D_j > L) \text{ and } (D_h \leq L) \right\}. \tag{7.24}$$

Then, a task set can be scheduled by EDF if $U < 1$ and

$$\forall L \in \mathcal{D} \quad B(L) + g(0, L) \leq L. \tag{7.25}$$

Table 7.6 Evaluation summary of resource access protocols

	Priority	Num. of blocking	Pessimism	Blocking instant	Transparency	Deadlock prevention	Implementation
NPP	Any	1	High	On arrival	YES	YES	Easy
HLP	Fixed	1	Medium	On arrival	NO	YES	Easy
PIP	Fixed	α_i	Low	On access	YES	NO	Hard
PCP	Fixed	1	Medium	On access	NO	YES	Medium
SRP	Any	1	Medium	On arrival	NO	YES	Easy

where $\mathcal{D}$ is the set of all absolute deadlines no greater than a certain point in time, given by the minimum between the hyperperiod H and the following expression:

$$\max\left(D_{max}, \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1-U}\right).$$

where $D_{max} = \max\{D_1, \ldots, D_n\}$.

7.10 Summary

The concurrency control protocols presented in this chapter can be compared with respect to several characteristics. Table 7.6 provides a qualitative evaluation of the algorithms in terms of priority assignment, number of blocking, level of pessimism, instant of blocking, programming transparency, deadlock prevention, and implementation complexity. Here, transparency refers to the impact of the protocol on the programming interface. A transparent protocol (like NPP and PIP) can be implemented without modifying the task code, that is, exploiting the same primitives used by classical semaphores. This means that, legacy applications developed using classical semaphores (prone to priority inversion) can also be executed under a transparent protocol. This feature makes such protocols attractive for commercial operating systems (like VxWorks), where predictability can be improved without introducing new kernel primitives. On the contrary, protocols that use resource ceilings (as HLP, PCP, and SRP) need an additional system call to specify such values in the application.

Exercises

7.1 Verify whether the following task set is schedulable by the Rate Monotonic algorithm. Apply the processor utilization approach first and then the response time analysis:

	C_i	T_i	B_i
τ_1	4	10	5
τ_2	3	15	3
τ_3	4	20	0

7.2 Consider three periodic tasks τ_1, τ_2, and τ_3 (having decreasing priority), which share three resources, A, B, and C, accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration $D_i(R)$ for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C
τ_1	2	0	2
τ_2	2	3	0
τ_3	3	2	5

7.3 Solve the same problem described in Exercise 7.2 when the resources are accessed by the Priority Ceiling Protocol.

7.4 For the task set described in Exercise 7.2, illustrate the situation produced by RM + PIP in which task τ_2 experiences its maximum blocking time.

7.5 Consider four periodic tasks τ_1, τ_2, τ_3, and τ_4 (having decreasing priority), which share five resources, A, B, C, D, and E, accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration $\delta_{i,R}$ for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C	D	E
τ_1	2	5	9	0	6
τ_2	0	0	7	0	0
τ_3	0	3	0	7	13
τ_4	6	0	8	0	10

7.6 Solve the same problem described in Exercise 7.5 when the resources are accessed by the Priority Ceiling Protocol.

7.7 For the task set described in Exercise 7.5, illustrate the situation produced by RM + PIP in which task τ_2 experiences its maximum blocking time.

7.8 Consider three tasks τ_1, τ_2, and τ_3, which share three multi-unit resources, A, B, and C, accessed using the Stack Resource Policy. Resources A and B have three units, whereas C has two units. Compute the ceiling table for all the resources based on the following task characteristics:

	D_i	μ_A	μ_B	μ_C
τ_1	5	1	0	1
τ_2	10	2	1	2
τ_3	20	3	1	1

Chapter 8
Limited Preemptive Scheduling

8.1 Introduction

The question whether preemptive systems are better than non-preemptive systems has been debated for a long time, but only partial answers have been provided in the real-time literature, and still some issues remain open. In fact, each approach has advantages and disadvantages, and no one dominates the other when both predictability and efficiency have to be taken into account in the system design. This chapter presents and compares some existing approaches for reducing preemptions and describes an efficient method for minimizing preemption costs by removing unnecessary preemptions while preserving system schedulability.

Preemption is a key factor in real-time scheduling algorithms, since it allows the operating system to immediately allocate the processor to incoming tasks with higher priority. In fully preemptive systems, the running task can be interrupted at any time by another task with higher priority and be resumed to continue when all higher-priority tasks have completed. In other systems, preemption may be disabled for certain intervals of time during the execution of critical operations (e.g., interrupt service routines, critical sections, etc.). In other situations, preemption can be completely forbidden to avoid unpredictable interference among tasks and achieve a higher degree of predictability (although higher blocking times).

The question of enabling or disabling preemption during task execution has been investigated by many authors under several points of view, and it is not trivial to answer. A general disadvantage of the non-preemptive discipline is that it introduces an additional blocking factor in higher-priority tasks, so reducing schedulability. On the other hand, however, there are several advantages to be considered when adopting a non-preemptive scheduler.

In particular, the following issues have to be taken into account when comparing the two approaches:

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_8

- In many practical situations, such as I/O scheduling or communication in a shared medium, either preemption is impossible or prohibitively expensive.
- Preemption destroys program locality, increasing the runtime overhead due to cache misses and pre-fetch mechanisms. As a consequence, worst-case execution times (WCETs) are more difficult to characterize and predict [LHS+98, RM06, RM08, RM09].
- The mutual exclusion problem is trivial in non-preemptive scheduling, which naturally guarantees the exclusive access to shared resources. On the contrary, to avoid unbounded priority inversion, preemptive scheduling requires the implementation of specific concurrency control protocols for accessing shared resources, as those presented in Chap. 7, which introduce additional overhead and complexity.
- In control applications, the input-output delay and jitter are minimized for all tasks when using a non-preemptive scheduling discipline, since the interval between start time and finishing time is always equal to the task computation time [BC07]. This simplifies control techniques for delay compensation at design time.
- Non-preemptive execution allows using stack sharing techniques [Bak91] to save memory space in small embedded systems with stringent memory constraints [GAGB01].

In summary, arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability. In particular, at least four different types of costs need to be taken into account at each preemption:

1. *Scheduling cost*. It is the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task.
2. *Pipeline cost*. It accounts for the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.
3. *Cache-related cost*. It is the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [AG08, GA07]. Bui et al. [BCSM08] showed that on a PowerPC MPC7410 with 2 MByte two-way associative L2 cache the WCET increment due to cache interference can be as large as 33% of the WCET measured in non-preemptive mode.
4. *Bus-related cost*. It is the extra bus interference for accessing the RAM due to the additional cache misses caused by preemption.

The cumulative execution overhead due to the combination of these effects is referred to as *architecture-related cost*. Unfortunately, this cost is characterized by a high variance and depends on the specific point in the task code when preemption takes place [AG08, GA07, LDS07].

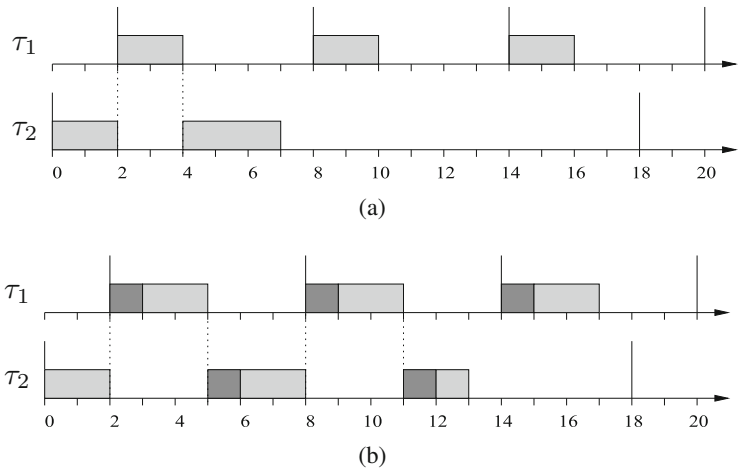


Fig. 8.1 Task τ_2 experiences a single preemption when preemption cost is neglected, and two preemptions when preemption cost is taken into account. (**a**) Schedule without preemption cost. (**b**) Schedule with preemption cost

The total increase of the worst-case execution time of a task τ_i is also a function of the total number of preemptions experienced by τ_i, which in turn depends on the task set parameters, on the activation pattern of higher-priority tasks, and on the specific scheduling algorithm. Such a circular dependency of WCET and number of preemptions makes the problem not easy to be solved. Figure 8.1a shows a simple example in which neglecting preemption cost task τ_2 experiences a single preemption. However, when taking preemption cost into account, τ_2's WCET becomes higher, and hence τ_2 experiences additional preemptions, which in turn increase its WCET. In Fig. 8.1b the architecture-related cost due to preemption is represented by dark gray areas.

Some methods for estimating the number of preemptions have been proposed [ERC95, YS07], but they are restricted to the fully preemptive case and do not consider such a circular dependency.

Often, preemption is considered a prerequisite to meet timing requirement in real-time system design; however, in most cases, a fully preemptive scheduler produces many unnecessary preemptions. Figure 8.2a illustrates an example in which, under fully preemptive scheduling, task τ_5 is preempted four times. As a consequence, the WCET of τ_5 is substantially inflated by the architecture-related cost (represented by dark gray areas), causing a response time equal to $R_5 = 18$. However, as shown in Fig. 8.2b, only one preemption is really necessary to guarantee the schedulability of the task set, reducing the WCET of τ_5 from 14 to 11 units of time and its response time from 18 to 13 units.

To reduce the runtime overhead due to preemptions and still preserve the schedulability of the task set, the following approaches have been proposed in the literature:

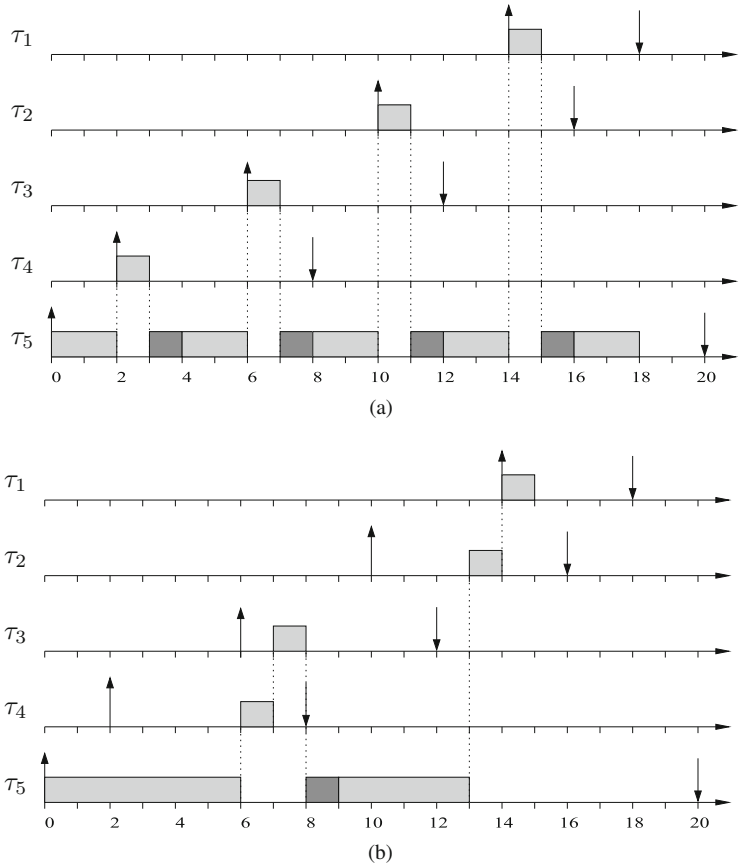


Fig. 8.2 Fully preemptive scheduling can generate several preemptions (**a**), although only a few of them are really necessary to guarantee the schedulability of the task set (**b**). (**a**) τ_5 is preempted 4 times. (**b**) Only one preemption is really necessary for τ_5

- *Preemption Thresholds*. According to this approach, proposed by Wang and Saksena [WS99], a task is allowed to disable preemption up to a specified priority level, which is called preemption threshold. Thus, each task is assigned a regular priority and a preemption threshold, and the preemption is allowed to take place only when the priority of the arriving task is higher than the threshold of the running task.
- *Deferred Preemptions*. According to this method, each task τ_i specifies the longest interval q_i that can be executed non-preemptively. Depending on how non-preemptive regions are implemented, this model can come in two slightly different flavors:

1. *Floating model*. In this model, non-preemptive regions are defined by the programmer by inserting specific primitives in the task code that disable and enable preemption. Since the start time of each region is not specified in the model, non-preemptive regions cannot be identified off-line, and, for the sake of the analysis, are considered to be "floating" in the code, with a duration $\delta_{i,k} \leq q_i$.
2. *Time-triggered model*. In this model, non-preemptive regions are triggered by the arrival of a higher-priority task and enforced by a timer to last for q_i units of time (unless the task finishes earlier), after which preemption is enabled. Once a timer is set at time t, additional activations arriving before the timeout $(t + q_i)$ do not postpone the preemption any further. After the timeout, a new high-priority arrival can trigger another non-preemptive region.

- *Task splitting*. According to this approach, investigated by Burns [Bur94], a task implicitly executes in non-preemptive mode, and preemption is allowed only at predefined locations inside the task code, called *preemption points*. In this way, a task is divided into a number of non-preemptive chunks (also called subjobs). If a higher-priority task arrives between two preemption points of the running task, preemption is postponed until the next preemption point. This approach is also referred to as *Cooperative scheduling*, because tasks cooperate to offer suitable preemption points to improve schedulability.

To better understand the different limited preemptive approaches, the task set reported in Table 8.1 will be used as a common example throughout this chapter.

Figure 8.3 illustrates the schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 8.1. Notice that the task set is not schedulable, since task τ_3 misses its deadline.

Table 8.1 Parameters of a sample task set with relative deadlines less than periods

	C_i	T_i	D_i
τ_1	1	6	4
τ_2	3	10	8
τ_3	6	18	12

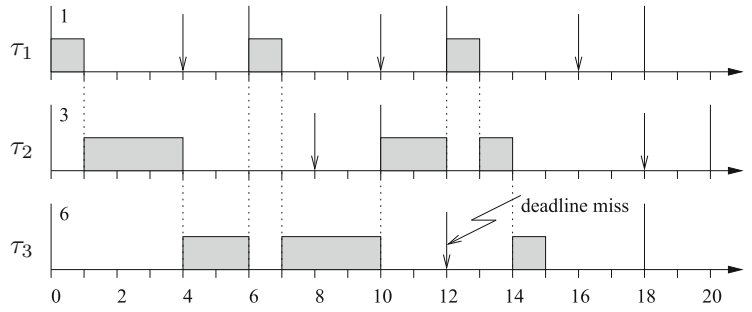


Fig. 8.3 Schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 8.1

8.1.1 Terminology and Notation

Throughout this chapter, a set of n periodic or sporadic real-time tasks will be considered to be scheduled on a single processor. Each task τ_i is characterized by a worst-case execution time (WCET) C_i, a relative deadline D_i, and a period (or minimum inter-arrival time) T_i. A constrained deadline model is adopted, so D_i is assumed to be less than or equal to T_i. For scheduling purposes, each task is assigned a fixed priority P_i, used to select the running task among those tasks ready to execute. A higher value of P_i corresponds to a higher priority. Notice that task activation times are not known a priori and the actual execution time of a task can be less than or equal to its worst-case value C_i. Tasks are indexed by decreasing priority, i.e., $\forall i \mid 1 \leq i < n : P_i > P_{i+1}$. Additional terminology will be introduced below for each specific method.

8.2 Non-preemptive Scheduling

The most effective way to reduce preemption cost is to disable preemptions completely. In this condition, however, each task τ_i can experience a blocking time B_i equal to the longest computation time among the tasks with lower priority. That is,

$$B_i = \max_{j:P_j<P_i} \{C_j - 1\} \tag{8.1}$$

where the maximum of an empty set is assumed to be zero. Notice that one unit of time is subtracted from the computation time of the blocking task to consider that, to block τ_i, it must start at least one unit before the critical instant. Such a blocking term introduces an additional delay before task execution, which could jeopardize schedulability. High-priority tasks are those that are most affected by such a blocking delay, since the maximum in Eq. (8.1) is computed over a larger set of tasks. Figure 8.4 illustrates the schedule generated by Deadline Monotonic on the task set of Table 8.1 when preemptions are disabled. With respect to the schedule shown in Fig. 8.3, notice that τ_3 is now able to complete before its deadline, but the task set is still not schedulable, since now τ_1 misses its deadline.

Unfortunately, under non-preemptive scheduling, the least upper bounds of both RM and EDF drop to zero! This means that there exist task sets with arbitrary low utilization that cannot be scheduled by RM and EDF when preemptions are disabled. For example, the task set illustrated in Fig. 8.5a is not feasible under non-preemptive Rate-Monotonic scheduling (as well as under non-preemptive EDF), since $C_2 > T_1$, but its utilization can be set arbitrarily low by reducing C_1 and increasing T_2. The same task set is clearly feasible when preemption is enabled, as shown in Fig. 8.5b.

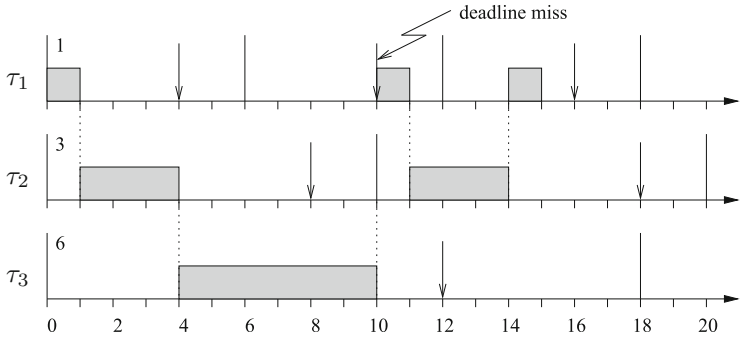


Fig. 8.4 Schedule produced by non-preemptive Deadline Monotonic on the task set of Table 8.1

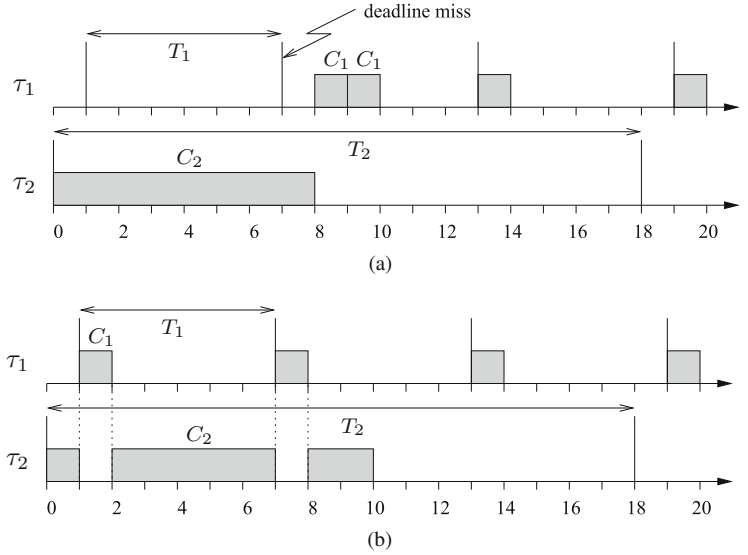


Fig. 8.5 A task set with low utilization that is unfeasible under non-preemptive Rate-Monotonic scheduling and feasible when preemption is enabled. (**a**) Non-preemptive case. (**b**) Preemptive case

8.2.1 Feasibility Analysis

The feasibility analysis of non-preemptive task sets is more complex than under fully preemptive scheduling. Bril et al. [BLV09] showed that in non-preemptive scheduling, the largest response time of a task does not necessarily occur in the first job, after the critical instant. An example of such a situation is illustrated in Fig. 8.6, where the worst-case response time of τ_3 occurs in its second instance. Such a scheduling anomaly, identified as *self-pushing phenomenon*, occurs because the high priority jobs activated during the non-preemptive execution of τ_i's first

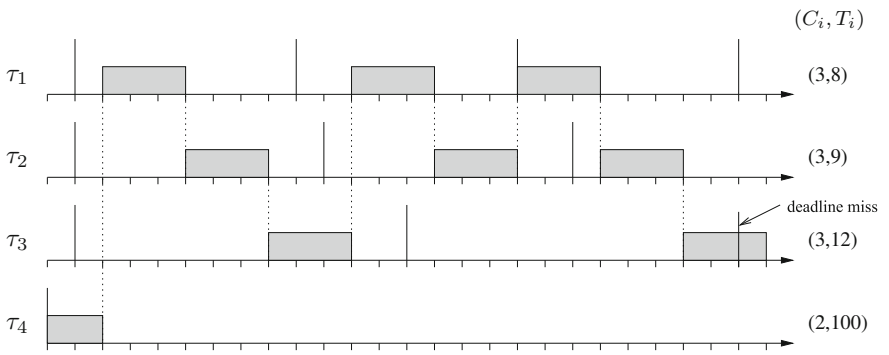


Fig. 8.6 An example of self-pushing phenomenon occurring on task τ_3

instance are pushed ahead to successive jobs, which then may experience a higher interference.

The presence of the self-pushing phenomenon in non-preemptive scheduling implies that the response time analysis for a task τ_i cannot be limited to its first job, activated at the critical instant, as done in preemptive scheduling, but it must be performed for multiple jobs, until the processor finishes executing tasks with priority higher than or equal to P_i. Hence, the response time of a task τ_i needs to be computed within the longest Level-i Active Period, defined as follows [BLV09].

Definition 8.1 The *Level-i pending workload* $W_i^p(t)$ at time t is the amount of processing that still needs to be performed at time t due to jobs with priority higher than or equal to P_i released strictly before t.

Definition 8.2 A *Level-i Active Period* L_i is an interval $[a, b)$ such that the Level-i pending workload $W_i^p(t)$ is positive for all $t \in (a, b)$ and null in a and b.

The longest Level-i Active Period can be computed by the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \displaystyle\sum_{h:P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{8.2}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$.

This means that the response time of task τ_i must be computed for all jobs $\tau_{i,k}$, with $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{8.3}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can then be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, and the interference of the tasks with priority higher than P_i.

Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} = B_i + \sum\limits_{h:P_h>P_i} C_h \\ s_{i,k}^{(\ell)} = B_i + (k-1)C_i + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.4}$$

Since, once started, the task cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + C_i. \tag{8.5}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{8.6}$$

Once the response time of each task is computed, the task set is feasible if and only if

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{8.7}$$

Yao, Buttazzo, and Bertogna [YBB10a] showed that the analysis of non-preemptive tasks can be reduced to a single job, under specific (but not too restrictive) conditions.

Theorem 8.1 (Yao, Buttazzo, and Bertogna, 2010) *The worst-case response time of a non-preemptive task occurs in the first job if the task is activated at its critical instant and the following two conditions are both satisfied:*

1. *the task set is feasible under preemptive scheduling;*
2. *relative deadlines are less than or equal to periods.*

Under these conditions, the longest relative start time S_i of task τ_i is equal to $s_{i,1}$ and can be computed from Eq. (8.4) for $k = 1$:

$$S_i = B_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h. \tag{8.8}$$

Hence, the response time R_i is simply

$$R_i = S_i + C_i. \tag{8.9}$$

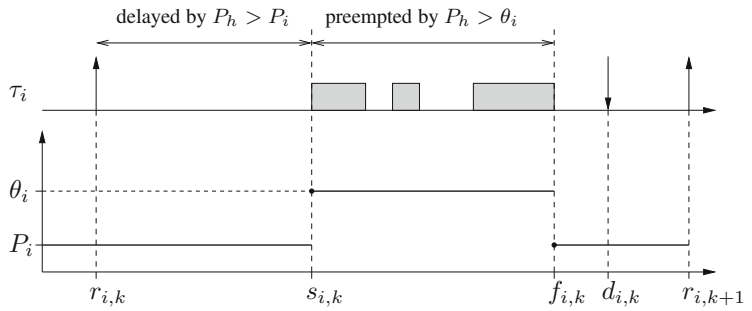


Fig. 8.7 Example of task executing under preemption threshold

8.3 Preemption Thresholds

According to this model, proposed by Wang and Saksena [WS99], each task τ_i is assigned a nominal priority P_i (used to enqueue the task into the ready queue and to preempt) and a *preemption threshold* $\theta_i \geq P_i$ (used for task execution). Then, τ_i can be preempted by τ_h only if $P_h > \theta_i$. Figure 8.7 illustrates how the threshold is used to raise the priority of a task τ_i during the execution of its k-th job. At the activation time $r_{i,k}$, the priority of τ_i is set to its nominal value P_i, so it can preempt all the tasks τ_j with threshold $\theta_j < P_i$. The nominal priority is maintained as long as the task is kept in the ready queue. During this interval, τ_i can be delayed by all tasks τ_h with priority $P_h > P_i$. When all such tasks complete (at time $s_{i,k}$), τ_i is dispatched for execution and its priority is raised at its threshold level θ_i until the task terminates (at time $f_{i,k}$). During this interval, τ_i can be preempted by all tasks τ_h with priority $P_h > \theta_i$. Notice that, when τ_i is preempted, its priority is kept to its threshold level.

Preemption threshold can be considered as a trade-off between fully preemptive and fully non-preemptive scheduling. Indeed, if each threshold priority is set equal to the task nominal priority, the scheduler behaves like the fully preemptive scheduler; whereas, if all thresholds are set to the maximum priority, the scheduler runs in non-preemptive fashion. Wang and Saksena also showed that, by appropriately setting the thresholds, the system can achieve a higher utilization efficiency, compared with fully preemptive and fully non-preemptive scheduling. For example, assigning the preemption thresholds shown in Table 8.2, the task set of Table 8.1 results to be schedulable by Deadline Monotonic, as illustrated in Fig. 8.8.[1]

Notice that, at $t = 6$, τ_1 can preempt τ_3 since $P_1 > \theta_3$. However, at $t = 10$, τ_2 cannot preempt τ_3, being $P_2 = \theta_3$. Similarly, at $t = 12$, τ_1 cannot preempt τ_2, being $P_1 = \theta_2$.

[1] Note that the task set is not schedulable under sporadic activations; in fact, τ_2 misses its deadline if τ_1 and τ_2 are activated one unit of time after τ_3.

Table 8.2 Preemption thresholds assigned to the tasks of Table 8.1

	P_i	θ_i
τ_1	3	3
τ_2	2	3
τ_3	1	2

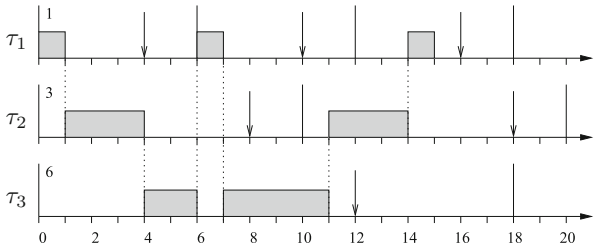


Fig. 8.8 Schedule produced by preemption thresholds for the task set in Table 8.1

8.3.1 Feasibility Analysis

Under fixed priorities, the feasibility analysis of a task set with preemption thresholds can be performed by the feasibility test derived by Wang and Saksena [WS99] and later refined by Regehr [Reg02]. First of all, a task τ_i can be blocked only by lower-priority tasks that cannot be preempted by it, that is, by tasks having a priority $P_j < P_i$ and a threshold $\theta_j \geq P_i$. Hence, a task τ_i can experience a blocking time equal to the longest computation time among the tasks with priority lower than P_i and threshold higher than or equal to P_i. That is,

$$B_i = \max_j \{C_j - 1 \mid P_j < P_i \leq \theta_j\} \tag{8.10}$$

where the maximum of an empty set is assumed to be zero. Then, the response time R_i of task τ_i is computed by considering the blocking time B_i, the interference before its start time (due to the tasks with priority higher than P_i), and the interference after its start time (due to tasks with priority higher than θ_i), as depicted in Fig. 8.7. The analysis must be carried out within the longest Level-i active period L_i, defined by the following recurrent relation:

$$L_i = B_i + \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i}{T_h} \right\rceil C_h. \tag{8.11}$$

This means that the response time of task τ_i must be computed for all the jobs $\tau_{i,k}$ ($k = 1, 2, \ldots$) within the longest Level-i active period. That is, for all $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{8.12}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} &= B_i + \sum\limits_{h:P_h>P_i} C_h \\ s_{i,k}^{(\ell)} &= B_i + (k-1)C_i + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.13}$$

For the same job $\tau_{i,k}$, the finishing time $f_{i,k}$ can be computed by summing to the start time $s_{i,k}$ the computation time of job $\tau_{i,k}$ and the interference of the tasks that can preempt $\tau_{i,k}$ (those with priority higher than θ_i). That is,

$$\begin{cases} f_{i,k}^{(0)} &= s_{i,k} + C_i \\ f_{i,k}^{(\ell)} &= s_{i,k} + C_i + \sum\limits_{h:P_h>\theta_i} \left(\left\lceil \dfrac{f_{i,k}^{(\ell-1)}}{T_h} \right\rceil - \left(\left\lfloor \dfrac{s_{i,k}}{T_h} \right\rfloor + 1 \right) \right) C_h. \end{cases} \tag{8.14}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{ f_{i,k} - (k-1)T_i \}. \tag{8.15}$$

Once the response time of each task is computed, the task set is feasible if

$$\forall i = 1, \ldots, n \quad R_i \le D_i. \tag{8.16}$$

The feasibility analysis under preemption thresholds can also be simplified under the conditions of Theorem 8.1. In this case, we have that the worst-case start time is

$$S_i = B_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h \tag{8.17}$$

and the worst-case response time of task τ_i can be computed as

$$R_i = S_i + C_i + \sum_{h:P_h>\theta_i} \left(\left\lceil \frac{R_i}{T_h} \right\rceil - \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) \right) C_h. \tag{8.18}$$

8.3.2 Selecting Preemption Thresholds

The example illustrated in Fig. 8.8 shows that a task set unfeasible under both preemptive and non-preemptive scheduling can be feasible under preemption

Algorithm: Assign Minimum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: Task set feasibility and $\theta_i, \forall \tau_i \in \mathcal{T}$
// Assumes tasks are ordered by decreasing priorities

```
(1)  begin
(2)      for (i := n to 1) do        // from the lowest priority task
(3)          θ_i := P_i;
(4)          Compute R_i by Equation (8.15);
(5)          while (R_i > D_i) do          // while not schedulable
(6)              θ_i := θ_i + 1;           // increase threshold
(7)              if (θ_i > P_1) then       // system infeasible
(8)                  return (INFEASIBLE);
(9)              end
(10)             Compute R_i by Equation (8.15);
(11)         end
(12)     end
(13)     return (FEASIBLE);
(14) end
```

Fig. 8.9 Algorithm for assigning the minimum preemption thresholds

thresholds, for a suitable setting of threshold levels. The algorithm presented in Fig. 8.9 was proposed by Wang and Saksena [WS99] and allows assigning a set of thresholds to achieve a feasible schedule, if there exists one. Threshold assignment is started from the lowest-priority task to the highest priority one, since the schedulability analysis only depends on the thresholds of tasks with lower priority than the current task. While searching the optimal preemption threshold for a specific task, the algorithm stops at the minimum preemption threshold that makes it schedulable. The algorithm assumes that tasks are ordered by decreasing priorities, being τ_1 the highest-priority task.

Notice that the algorithm is optimal in the sense that, if there exists a preemption threshold assignment that can make the system schedulable, the algorithm will always find an assignment that ensures schedulability.

Given a task set that is feasible under preemptive scheduling, another interesting problem is to determine the thresholds that limit preemption as much as possible, without jeopardizing the schedulability of the task set. The algorithm shown in Fig. 8.10, proposed by Saksena and Wang [SW00], tries to increase the threshold of each task up to the level after which the schedule would become infeasible. The algorithm considers one task at the time, starting from the highest-priority task.

Algorithm: Assign Maximum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: Thresholds $\theta_i, \forall \tau_i \in \mathcal{T}$
// Assumes that the task set is preemptively feasible

```
(1)  begin
(2)      for (i := 1 to n) do
(3)          θ_i = P_i;
(4)          k = i;                    // priority level k
(5)          schedulable := TRUE;
(6)          while ((schedulable := TRUE) and (k > 1)) do
(7)              k = k − 1;            // go to the higher priority level
(8)              θ_i = P_k;            // set threshold at that level
(9)              Compute R_k by Equation (8.15);
(10)             if (R_k > D_k) then   // system not schedulable
(11)                 schedulable := FALSE;
(12)                 θ_i = P_{k+1};    // assign the previous priority level
(13)             end
(14)         end
(15)     end
(16) end
```

Fig. 8.10 Algorithm for assigning the maximum preemption thresholds

8.4 Deferred Preemptions

According to this method, each task τ_i defines a maximum interval of time q_i in which it can execute non-preemptively. Depending on the specific implementation, the non-preemptive interval can start after the invocation of a system call inserted at the beginning of a non-preemptive region (floating model) or can be triggered by the arrival of a higher-priority task (time-triggered model).

Under the floating model, preemption is resumed by another system call, inserted at the end of the region (long at most q_i units), whereas, under the time-triggered model, preemption is enabled by a timer interrupt after exactly q_i units (unless the task completes earlier).

Since, in both cases, the start times of non-preemptive intervals are assumed to be unknown a priori, non-preemptive regions cannot be identified off-line and, for the sake of the analysis, they are considered to occur in the worst possible time (in the sense of schedulability).

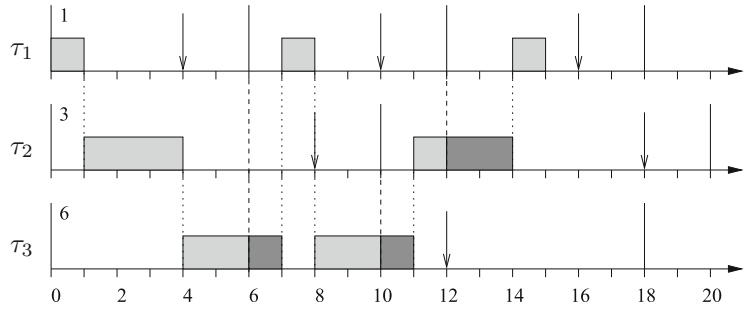


Fig. 8.11 Schedule produced by Deadline Monotonic with deferred preemptions for the task set reported in Table 8.1, with $q_2 = 2$ and $q_3 = 1$

For example, considering the same task set of Table 8.1, assigning $q_2 = 2$ and $q_3 = 1$, the schedule produced by Deadline Monotonic with deferred preemptions is feasible, as illustrated in Fig. 8.11. Dark regions represent intervals executed in non-preemptive mode, triggered by the arrival of higher-priority tasks.

8.4.1 Feasibility Analysis

In the presence of non-preemptive intervals, a task can be blocked when, at its arrival, a lower-priority task is running in non-preemptive mode. Since each task can be blocked at most once by a single lower-priority task, B_i is equal to the longest non-preemptive interval belonging to tasks with lower priority. Under a time-triggered implementation, we have

$$B_i = \max_{j:P_j<P_i} \{q_j\}. \tag{8.19}$$

Notice that there is no need to subtract one unit of time from q_j, since a non-preemptive interval is programmed to be exactly q_j. One unit should be subtracted from q_j under the floating model, to allow the non-preemptive region to start before τ_i. Then, schedulability can be checked through the response time analysis, by Eq. (7.22), or through the workload analysis, by Eq. (7.23). Note that, under the floating model, the analysis does not need to be carried out within the longest Level-i active period. In fact, since non-preemptive regions can have an arbitrary duration no greater than q_i, the worst-case interference on τ_i can actually occur in the first instance, assuming that τ_i can be preempted an epsilon before its completion.

On the other hand, the analysis is more pessimistic under the time-triggered model, where non-preemptive intervals are exactly equal to q_i units and can last until the end of the task. In this case, the analysis does not take advantage of the fact that τ_i cannot be preempted when higher periodic tasks arrive q_i units (o less) before

its completion. The advantage of such a pessimism, however, is that the analysis can be limited to the first job of each task.

8.4.2 Longest Non-preemptive Interval

When using the deferred preemption method, an interesting problem is to find the longest non-preemptive interval Q_i for each task τ_i that can still preserve the task set schedulability. More precisely, the problem can be stated as follows:

> Given a set of n periodic tasks that is feasible under preemptive scheduling, find the longest non-preemptive interval of length Q_i for each task τ_i, so that τ_i can continue to execute for Q_i units of time in non-preemptive mode, without violating the schedulability of the original task set.

This problem has been first solved under EDF by Baruah [Bar05] and then under fixed priorities by Yao et al. [YBB09]. The solution is based on the concept of *blocking tolerance* β_i, for a task τ_i, defined as follows.

Definition 8.3 The *blocking tolerance* β_i of a task τ_i is the maximum amount of blocking τ_i can tolerate without missing any of its deadlines.

A simple way to compute the blocking tolerance is from the Liu and Layland test, which, in the presence of blocking factors, becomes (see Eq. (7.20)

$$\forall i = 1, \ldots, n \quad \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} + \frac{B_i}{T_i} \leq U_{lub}(i)$$

where $U_{lub}(i) = i(2^{1/i} - 1)$. Isolating the blocking factor, the test can also be rewritten as

$$B_i \leq T_i \left[U_{lub}(i) - \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} \right].$$

Hence:

$$\beta_i = T_i \left[U_{lub}(i) - \sum_{h:P_h \geq P_i} \frac{C_h}{T_h} \right]. \tag{8.20}$$

A more precise bound for β_i can be achieved by using the schedulability test expressed by Eq. (7.23), which leads to the following result:

$$\exists t \in \mathcal{TS}(\tau_i) : B_i \leq \{t - W_i(t)\}.$$

$$B_i \leq \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i(t)\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i(t)\}. \tag{8.21}$$

Given the blocking tolerance, the feasible test can also be expressed as follows:

$$\forall i = 1, \ldots, n \quad B_i \leq \beta_i$$

and, by Eq. (8.19), we can write

$$\forall i = 1, \ldots, n \quad \max_{j: P_j < P_i} \{q_j\} \leq \beta_i.$$

This implies that, to achieve feasibility, we must have

$$\forall i = 1, \ldots, n \quad q_i \leq \min_{k: P_k > P_i} \{\beta_k\}$$

Hence, the longest non-preemptive interval Q_i that preserves feasibility for each task τ_i is

$$Q_i = \min_{k: P_k > P_i} \{\beta_k\}. \tag{8.22}$$

The Q_i terms can also be computed more efficiently, starting from the highest-priority task (τ_1) and proceeding with decreasing priority order, according to the following theorem.

Theorem 8.2 *The longest non-preemptive interval Q_i of task τ_i that preserves feasibility can be computed as*

$$Q_i = \begin{cases} \infty & \text{if } i = 1 \\ \min\{Q_{i-1}, \beta_{i-1}\} & \text{otherwise} \end{cases} \tag{8.23}$$

Proof The theorem can be proved by noting that

$$\min_{k: P_k > P_i} \{\beta_k\} = \min\{\min_{k: P_k > P_{i-1}} \{\beta_k\}, \beta_{i-1}\}$$

and since from Eq. (8.22)

$$Q_{i-1} = \min_{k: P_k > P_{i-1}} \{\beta_k\}$$

we have that

$$Q_i = \min\{Q_{i-1}, \beta_{i-1}\}$$

Algorithm: Compute the Longest Non-Preemptive Intervals
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: $Q_i, \forall \tau_i \in \mathcal{T}$
// Assumes $\mathcal{T}$ is preemptively feasible and $D_i \leq T_i$

(1) **begin**
(2) $\beta_1 = D_1 - C_1$;
(3) $Q_1 = \infty$;
(4) **for** $(i := 2$ **to** $n)$ **do**
(5) $Q_i = \min\{Q_{i-1}, \beta_{i-1}\}$;
(6) Compute β_i using Equation (8.20) or (8.21);
(7) **end**
(8) **end**

Fig. 8.12 Algorithm for computing the longest non-preemptive intervals

which proves the theorem. □

Note that, in order to apply Theorem 8.2, Q_i is not constrained to be less than or equal to C_i, but a value of Q_i greater than C_i means that τ_i can be fully executed in non-preemptive mode. The algorithm for computing the longest non-preemptive intervals is illustrated in Fig. 8.12.

8.5 Task Splitting

According to this model, each task τ_i is split into m_i non-preemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code. Thus, preemptions can only occur at the subjobs boundaries. All the jobs generated by one task have the same subjob division. The kth subjob has a worst-case execution time $q_{i,k}$, hence $C_i = \sum_{k=1}^{m_i} q_{i,k}$.

Among all the parameters describing the subjobs of a task, two values are of particular importance for achieving a tight schedulability result:

$$\begin{cases} q_i^{max} = \max\limits_{k \in [1, m_i]} \{q_{i,k}\} \\ q_i^{last} = q_{i,m_i} \end{cases} \tag{8.24}$$

In fact, the feasibility of a high-priority task τ_k is affected by the size q_j^{max} of the longest subjob of each task τ_j with priority $P_j < P_k$. Moreover, the length q_i^{last} of the final subjob of τ_i directly affects its response time. In fact, all higher-priority jobs arriving during the execution of τ_i's final subjob do not cause a preemption,

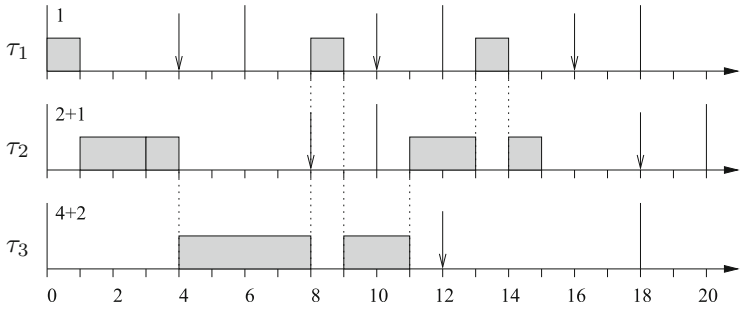


Fig. 8.13 Schedule produced by Deadline Monotonic for the task set reported in Table 8.1, when τ_2 is split in two subjobs of 2 and 1 unit, and τ_3 is split in two subjobs of 4 and 2 units, respectively

since their execution is postponed at the end of τ_i. Therefore, in this model, each task will be characterized by the following 5-tuple:

$$\left\{C_i, D_i, T_i, q_i^{max}, q_i^{last}\right\}.$$

For example, consider the same task set of Table 8.1, and suppose that τ_2 is split in two subjobs of 2 and 1 unit and τ_3 is split in two subjobs of 4 and 2 units. The schedule produced by Deadline Monotonic with such a splitting is feasible, and it is illustrated in Fig. 8.13.

8.5.1 Feasibility Analysis

Feasibility analysis for task splitting can be carried out in a very similar way as the non-preemptive case, with the following differences:

- The blocking factor B_i to be considered for each task τ_i is equal to the length of longest subjob (instead of the longest task) among those with lower priority:

$$B_i = \max_{j:P_j<P_i} \left\{q_j^{max} - 1\right\}. \tag{8.25}$$

- The last non-preemptive chunk of τ_i is equal to q_i^{last} (instead of C_i).

The response time analysis for a task τ_i has to consider all the jobs within the longest Level-i Active Period, which can be computed using the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \displaystyle\sum_{h:P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{8.26}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$. This means that the response time of τ_i must be computed for all jobs $\tau_{i,k}$ with $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{8.27}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ of the last subjob can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, those subjobs preceding the last one ($C_i - q_i^{last}$), and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} &= B_i + C_i - q_i^{last} + \sum\limits_{h:P_h>P_i} C_h \\ s_{i,k}^{(\ell)} &= B_i + kC_i - q_i^{last} + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.28}$$

Since, once started, the last subjob cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + q_i^{last}. \tag{8.29}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{8.30}$$

Once the response time of each task is computed, the task set is feasible if

$$\forall i = 1, \ldots, n \quad R_i \le D_i. \tag{8.31}$$

Assuming that the task set is preemptively feasible, the analysis can be simplified to the first job of each task, after the critical instant, as shown by Yao et al. [YBB10a]. Hence, the longest relative start time of τ_i can be computed as the smallest value satisfying the following recurrent relation:

$$\begin{cases} S_i^{(0)} &= B_i + C_i - q_i^{last} + \sum\limits_{h:P_h>P_i} C_h \\ S_i^{(\ell)} &= B_i + kC_i - q_i^{last} + \sum\limits_{h:P_h>P_i} \left(\left\lfloor \dfrac{S_i^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8.32}$$

Then, the response time R_i is simply

$$R_i = S_i + q_i^{last}. \tag{8.33}$$

8.5.2 *Longest Non-preemptive Interval*

As done in Sect. 8.4.2 under deferred preemptions, it is interesting to compute, also under task splitting, the longest non-preemptive interval Q_i for each task τ_i that can still preserve the schedulability. It is worth observing that splitting tasks into subjobs allows achieving a larger Q_i, because a task τ_i cannot be preempted during the execution of the last q_i^{last} units of time.

Tasks are assumed to be preemptively feasible, so that the analysis can be limited to the first job of each task. In this case, a bound on the blocking tolerance β_i can be achieved using the following schedulability condition [YBB10a]:

$$\exists t \in \mathcal{TS}^*(\tau_i) : B_i \leq \{t - W_i^*(t)\}, \tag{8.34}$$

where $W_i^*(t)$ and the testing set $\mathcal{TS}^*$ are defined as

$$W_i^*(t) = C_i - q_i^{last} + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h, \tag{8.35}$$

$$\mathcal{TS}^*(\tau_i) \stackrel{\text{def}}{=} \mathcal{P}_{i-1}(D_i - q_i^{last}) \tag{8.36}$$

being $\mathcal{P}_i(t)$ given by Eq. (4.35).

Rephrasing Eq. (8.34), we obtain

$$B_i \leq \max_{t \in \mathcal{TS}^*(\tau_i)} \left\{ t - W_i^*(t) \right\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}^*(\tau_i)} \left\{ t - W_i^*(t) \right\}. \tag{8.37}$$

The longest non-preemptive interval Q_i that preserves feasibility for each task τ_i can then be computed by Theorem 8.2, using the blocking tolerances given by Eq. (8.37). Applying the same substitutions, the algorithm in Fig. 8.12 can also be used under task splitting.

As previously mentioned, the maximum length of the non-preemptive chunk under task splitting is larger than in the case of deferred preemptions. It is worth pointing out that the value of Q_i for task τ_i only depends on the β_k of the higher-priority tasks, as expressed in Eq. (8.22), and the blocking tolerance β_i is a function of q_i^{last}, as clear from Eqs. (8.35) and (8.37).

8.6 Selecting Preemption Points

When a task set is not schedulable in non-preemptive mode, there can be several ways to split tasks into subtasks to generate a feasible schedule, if there exists one. Moreover, as already observed in Sect. 8.1, the runtime overhead introduced by the preemption mechanism depends on the specific point where the preemption takes place. Hence, it would be useful to identify the best locations for placing preemption points inside each task to achieve a feasible schedule while minimizing the overall preemption cost. This section illustrates an algorithm that achieves this goal.

Considering that there exist sections of code where preemption is not desirable (e.g., short loops, critical sections, I/O operations, etc.), each job of τ_i is assumed to consist of a sequence of N_i non-preemptive basic blocks (BBs), identified by the programmer based on the task structure. Preemption is allowed only at basic block boundaries; thus each task has $N_i - 1$ *Potential Preemption Points* (PPPs), one between any two consecutive BBs. Critical sections and conditional branches are assumed to be executed entirely within a basic block. In this way, there is no need for using shared resource protocols to access critical sections.

The goal of the algorithm is to identify a subset of PPPs that minimizes the overall preemption cost while achieving the schedulability of the task set. A PPP selected by the algorithm is referred to as an *effective preemption point* (EPP), whereas the other PPPs are disabled. Therefore, the sequence of basic blocks between any two consecutive EPPs forms a non-preemptive region (NPR). The following notation is used to describe the algorithm:

- N_i denotes the number of BBs of task τ_i, determined by the $N_i - 1$ PPPs defined by the programmer;
- p_i denotes the number of NPRs of task τ_i, determined by the $p_i - 1$ EPPs selected by the algorithm;
- $\delta_{i,k}$ denotes the k-th basic block of task τ_i;
- $b_{i,k}$ denotes the WCET of $\delta_{i,k}$ without preemption cost, that is, when τ_i is executed non-preemptively;
- $\xi_{i,k}$ denotes the worst-case preemption overhead introduced when τ_i is preempted at the k-th PPP (i.e., between δ_k and δ_{k+1});
- $q_{i,j}$ denotes the WCET of the j-th NPR of τ_i, including the preemption cost;
- $q_i^{\max}$ denotes the maximum NPR length for τ_i:

$$q_i^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}.$$

To simplify the notation, the task index is omitted from task parameters whenever the association with the related task is evident from the context. In the following, we implicitly refer to a generic task τ_i, with maximum allowed NPR length $Q_i = Q$. As shown in the previous sections, Q can be computed by the algorithm in Fig. 8.12. We say that an EPP selection is *feasible* if the length of each resulting NPR, including the initial preemption overhead, does not exceed Q.

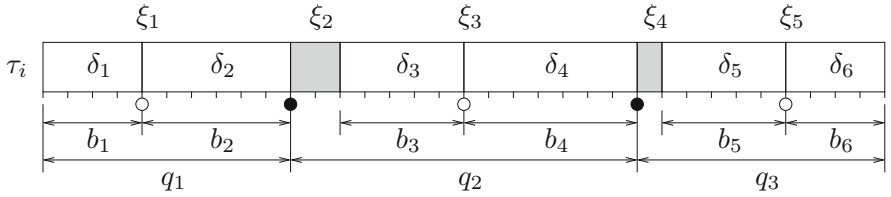


Fig. 8.14 Example of task with 6 BBs split into 3 NPRs. Preemption cost is reported for each PPPs, but accounted only for the EPPs

Figure 8.14 illustrates some of the defined parameters for a task with 6 basic blocks and 3 NPRs. PPPs are represented by dots between consecutive basic blocks: black dots are EPPs selected by the algorithm, while white dots are PPPs that are disabled. Above the task code, the figure also reports the preemption costs ξ_k for each PPP, although only the cost for the EPPs is accounted in the q_j of the corresponding NPR.

Using the notation introduced above, the non-preemptive WCET of τ_i can be expressed as follows:

$$C_i^{\mathrm{NP}} = \sum_{k=1}^{N_i} b_{i,k}.$$

The goal of the algorithm is to minimize the overall worst-case execution time C_i of each task τ_i, including the preemption overhead, by properly selecting the EPPs among all the PPPs specified in the code by the programmer, without compromising the schedulability of the task set. To compute the preemption overhead, we assume that each EPP leads to a preemption and that the cache is invalidated after each context switch. Therefore,

$$C_i = C_i^{\mathrm{NP}} + \sum_{k=1}^{N_i - 1} \mathsf{selected}(i,k) \cdot \xi_{i,k}$$

where $\mathsf{selected}(i,k) = 1$ if the k-th PPP of τ_i is selected by the algorithm to be an EPP, whereas $\mathsf{selected}(i,k) = 0$, otherwise.

8.6.1 Selection Algorithm

First of all, it is worth noting that minimizing the number of EPPs does not necessarily minimize the overall preemption overhead. In fact, there are cases in which inserting more preemption points, than the minimum number, could be more convenient to take advantage of points with smaller cost.

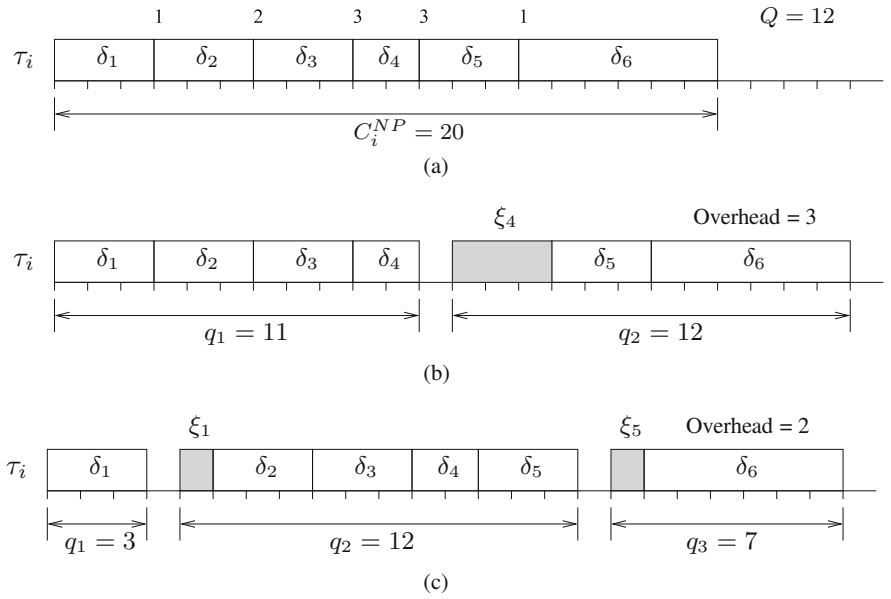


Fig. 8.15 Two solutions for selecting EPPs in a task with $Q = 12$: the first minimizes the number of EPPs, while the second minimizes the overall preemption cost. (**a**) Task with 6 basic blocks. (**b**) Task with a single preemption point. (**c**) Task with two preemption points

Consider, for instance, the task illustrated in Fig. 8.15, consisting of six basic blocks, whose total execution time in non-preemptive mode is equal to $C_i^{NP} = 20$. The numbers above each PPP in Fig. 8.15a denote the preemption cost, that is, the overhead that would be added to C_i^{NP} if a preemption occurred in that location. Assuming a maximum non-preemptive interval $Q = 12$, a feasible schedule could be achieved by inserting a single preemption point at the end of δ_4, as shown in Fig. 8.15b. In fact, $\sum_{k=1}^{4} b_k = 3 + 3 + 3 + 2 = 11 \leq Q$, and $\xi_4 + \sum_{k=5}^{6} b_k = 3 + 3 + 6 = 12 \leq Q$, leading to a feasible schedule. This solution is characterized by a total preemption overhead of 3 units of time (shown by the gray execution area). However, selecting two EPPs, one after δ_1 and another after δ_5, a feasible solution is achieved with a smaller total overhead $\xi_1 + \xi_5 = 1 + 1 = 2$, as shown in Fig. 8.15c. In general, for tasks with a large number of basic blocks with different preemption cost, finding the optimal solution is not trivial.

For a generic task, the worst-case execution time q of a NPR composed of the consecutive basic blocks $\delta_j, \delta_{j+1}, \ldots, \delta_k$ can be expressed as

$$q = \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell, \tag{8.38}$$

conventionally setting $\xi_0 = 0$. Note that the preemption overhead is included in q. Since any NPR of a feasible EPP selection has to meet the condition $q \leq Q$, we must have

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \leq Q. \tag{8.39}$$

Now, let $\hat{C}_k$ be the WCET of the chunk of code composed of the first k basic blocks—i.e., from the beginning of δ_1 until the end of δ_k—including the preemption overhead of the EPPs that are contained in the considered chunk. Then, we can express the following recursive expression:

$$\hat{C}_k = \hat{C}_{j-1} + q = \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell. \tag{8.40}$$

Note that since δ_N is the last BB, the worst-case execution time C_i of the whole task τ_i is equal to $\hat{C}_N$.

The algorithm for the optimal selection of preemption points is based on the equations presented above, and its pseudo-code is reported in Fig. 8.16. The algorithm evaluates all the BBs in increasing order, starting from the first one. For each BB δ_k, the feasible EPP selection that leads to the smallest possible $\hat{C}_k$ is computed as follows.

For the first BBs, the minimum $\hat{C}_k$ is given by the sum of the BB lengths $\sum_{\ell=1}^{k} b_\ell$ as long as this sum does not exceed Q. Note that if $b_1 > Q$, there is no feasible PPP selection, and the algorithm fails. For the following BBs, $\hat{C}_k$ needs to consider the cost of one or more preemptions as well. Let $Prev_k$ be the set of the preceding BBs $\delta_{j \leq k}$ that satisfy Condition (8.39), i.e., that might belong to the same NPR of δ_k. Again, if $\xi_{k-1} + b_k > Q$, there is no feasible PPP activation, and the algorithm fails. Otherwise, the minimum $\hat{C}_k$ is given by

$$\hat{C}_k = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{8.41}$$

Let $\delta^*(\delta_k)$ be the basic block for which the rightmost term of Expression (8.41) is minimum

$$\delta^*(\delta_k) = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{8.42}$$

If there are many possible BBs minimizing (8.41), the one with the smallest index is selected. Let $\delta_{Prev}(\delta_k)$ be the basic block preceding $\delta^*(\delta_k)$, if there exists one.

Algorithm: SelectEPP(τ_i, Q_i)
Input: $\{C_i, T_i, D_i, Q_i\}$ for task τ_i
Output: The set of EPPs for task τ_i

```
(1)  begin
(2)      Prev_k := {δ_0}; Ĉ_0 := 0;      // Initialize variables
(3)      for (k := 1 to N) do            // For all PPPs
(4)          Remove from Prev_k all δ_j violating (8.39);
(5)          if (Prev_k = ∅) then
(6)              return (INFEASIBLE);
(7)          end
(8)          Compute Ĉ_k using Equation (8.41);
(9)          Store δ_Prev(δ_k);
(10)         Prev_k := Prev_k ∪ {δ_k};
(11)     end
(12)     δ_j := δ_Prev(δ_N);
(13)     while (δ_j ≠ ∅) do
(14)         Select the PPP at the end of δ_Prev(δ_j);
(15)         δ_j ← δ_Prev(δ_j);
(16)     end
(17)     return (FEASIBLE);
(18) end
```

Fig. 8.16 Algorithm for selecting the optimal preemption points

The PPP at the end of $\delta_{Prev}(\delta_k)$—or, equivalently, at the beginning of $\delta^*(\delta_k)$—is meaningful for the analysis, since it represents the last PPP to activate for minimizing the preemption overhead of the first k basic blocks.

A feasible placement of EPPs for the whole task can then be derived with a recursive activation of PPPs, starting with the PPP at the end of $\delta_{Prev}(\delta_N)$, which will be the last EPP of the considered task. The penultimate EPP will be the one at the beginning of $\delta_{Prev}(\delta_{Prev}(\delta_N))$ and so on. If the result of this recursive lookup of function $\delta_{Prev}(k)$ is δ_1, the start of the program has been reached. A feasible placement of EPPs has therefore been detected, with a worst-case execution time, including preemption overhead, equal to $\hat{C}_N$. This is guaranteed to be the placement that minimizes the preemption overhead of the considered task, as proved in the next theorem.

Theorem 8.3 *The PPP activation pattern detected by procedure* **SelectEPP**(τ_i, Q_i) *minimizes the preemption overhead experienced by a task τ_i, without compromising the schedulability.*

Proof First, we prove that if procedure **SelectEPP**(τ_i, Q_i) fails, there is no other feasible EPP placement. For the procedure to fail, it is necessary that the condition at line (5) is satisfied for some δ_k. This means that δ_k violates Condition (8.39) for any $j \leq k$. That is,

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell > Q, \qquad \forall j \leq k.$$

This means that with any possible PPP activation pattern, the length of the NPR containing δ_k will be larger than Q, leading to a deadline miss.

We now consider the minimization of the preemption overhead. Let C_i be the WCET, including the preemption overhead, resulting from the EPP allocation given by the **SelectEPP**(τ_i, Q_i) procedure. Suppose there exists another feasible EPP allocation that results in a smaller $C_i' < C_i$. Then, there should be a BB $\delta_k, 1 \leq k \leq N$ for which $\hat{C}_k' < \hat{C}_k$. We prove by induction that this is not possible. The proof inducts over the index j of the basic blocks δ_j, proving that $\hat{C}_j$ is minimized for all $j, 1 \leq j \leq k$.

Base Case For $j = 1$, $\hat{C}_1 = b_1$ by definition. This is the minimum possible value of the WCET of the first BB, since it does not experience any preemption.

Inductive Step Assume that all $\hat{C}_\ell, \ \forall \ell < j$ are minimized by **SelectEPP**(τ_i, Q_i). We prove that $\hat{C}_j$ is also minimized. By Eq. (8.41), procedure **SelectEPP**(τ_i, Q_i) computes $\hat{C}_j$ as

$$\hat{C}_j = \min_{\delta_\ell \in Prev_j} \left\{ \hat{C}_{\ell-1} + \xi_{\ell-1} + \sum_{m=\ell}^{j} b_m \right\}.$$

Since, by induction hypothesis, all $\hat{C}_{\ell-1}$ terms are minimal, also $\hat{C}_j$ is minimized, proving the statement. Therefore, $\hat{C}_k$ is the minimum possible value, reaching a contradiction and proving the theorem. □

The feasibility of a given task set is maximized by applying the **SelectEPP**(τ_i, Q_i) procedure to each task τ_i, starting from τ_1 and proceeding in task order. Once the optimal allocation of EPPs has been computed for a task τ_i, the value of the overall WCET $C_i = \hat{C}_N$ can be used for the computation of the maximum allowed NPR Q_{i+1} of the next task τ_{i+1}, using the technique presented in Sect. 8.5. The procedure is repeated until a feasible PPP activation pattern has been produced for all tasks in the considered set. If the computed Q_{i+1} is too small to find a feasible EPP allocation, the only possibility to reach schedulability is by removing tasks from the system, as no other EPP allocation strategy would produce a feasible schedule.

8.7 Assessment of the Approaches

The limited preemption methods presented in this chapter can be compared under several aspects, such as:

- Implementation complexity.
- Predictability in estimating the preemption cost;
- Effectiveness in reducing the number of preemptions;

8.7.1 Implementation Issues

The preemption threshold mechanism can be implemented by raising the execution priority of the task, as soon as it starts running. The mechanism can be easily implemented at the application level by calling, at the beginning of the task, a system call that increases the priority of the task at its threshold level. The mechanism can also be fully implemented at the operating system level, without modifying the application tasks. To do that, the kernel has to increase the priority of a task at the level of its threshold when the task is scheduled for the first time. In this way, at its first activation, a task is inserted in the ready queue using its nominal priority. Then, when the task is scheduled for execution, its priority becomes equal to its threshold, until completion. Note that, if a task is preempted, its priority remains at its threshold level.

In deferred preemption (floating model), non-preemptive regions can be implemented by proper kernel primitives that disable and enable preemption at the beginning and at the end of the region, respectively. As an alternative, preemption can be disabled by increasing the priority of the task at its maximum value and can be enabled by restoring the nominal task priority. In the time-triggered mode, non-preemptive regions can be realized by setting a timer to enforce the maximum interval in which preemption is disabled. Initially, all tasks can start executing in non-preemptive mode. When τ_i is running and a task with priority higher than P_i is activated, a timer is set by the kernel (inside the activation primitive) to interrupt τ_i after q_i units of time. Until then, τ_i continues executing in non-preemptive mode. The interrupt handler associated with the timer must then call the scheduler to allow the higher-priority task to preempt τ_i. Notice that, once a timer has been set, other additional activations before the timeout will not prolong the timeout any further.

Finally, cooperative scheduling does not need special kernel support, but it requires the programmer to insert in each preemption point a primitive that calls the scheduler, so enabling pending high-priority tasks to preempt the running task.

8.7.2 Predictability

As observed in Sect. 8.1, the runtime overhead introduced by the preemption mechanism depends on the specific point where the preemption takes place. Therefore, a method that allows predicting where a task is going to be preempted simplifies the estimation of preemption costs, permitting a tighter estimation of task WCETs.

Unfortunately, under preemption thresholds, the specific preemption points depend on the actual execution of the running task and on the arrival time of high-priority tasks; hence, it is practically impossible to predict the exact location where a task is going to be preempted.

Under deferred preemptions (floating model), the position of non-preemptive regions is not specified in the model; thus they are considered to be unknown. In the time-triggered model, instead, the time at which the running task will be preempted is set q_i units of time after the arrival time of a higher-priority task. Hence, the preemption position depends on the actual execution of the running task and on the arrival time of the higher-priority task. Therefore, it can hardly be predicted off-line.

On the contrary, under cooperative scheduling, the locations where preemptions may occur are explicitly defined by the programmer at design time; hence, the corresponding overhead can be estimated more precisely by timing analysis tools. Moreover, through the algorithm presented in Sect. 8.6.1, it is also possible to select the best locations for placing the preemption points to minimize the overall preemption cost.

8.7.3 Effectiveness

Each of the presented methods can be used to limit preemption as much as desired, but the number of preemptions each task can experience depends of different parameters.

Under preemption thresholds, a task τ_i can only be preempted by tasks with priority greater than its threshold θ_i. Hence, if preemption cost is neglected, an upper bound ν_i on the number of preemptions τ_i can experience can be computed by counting the number of activations of tasks with priority higher than θ_i occurring in $[0, R_i]$, that is,

$$\nu_i = \sum_{h:P_h>\theta_i} \left\lceil \frac{R_i}{T_h} \right\rceil .$$

This is an upper bound because simultaneous activations are counted as they were different, although they cause a single preemption.

Under deferred preemption, the number of preemptions occurring on τ_i is easier to determine, because it directly depends on the non-preemptive interval q_i specified

for the task. If preemption cost is neglected, we simply have

$$\nu_i = \left\lceil \frac{C_i^{NP}}{q_i} \right\rceil - 1.$$

However, if preemption cost is not negligible, the estimation requires an iterative approach, since the task computation time also depends on the number of preemptions. Considering a fixed cost ξ_i for each preemption, then the number of preemptions can be upper bounded using the following recurrent relation:

$$\begin{cases} \nu_i^0 = \left\lceil \frac{C_i^{NP}}{q_i} \right\rceil - 1 \\ \nu_i^s = \left\lceil \frac{C_i^{NP} + \xi_i \nu_i^{s-1}}{q_i} \right\rceil - 1 \end{cases}$$

where the iteration process converges when $\nu_i^s = \nu_i^{s-1}$.

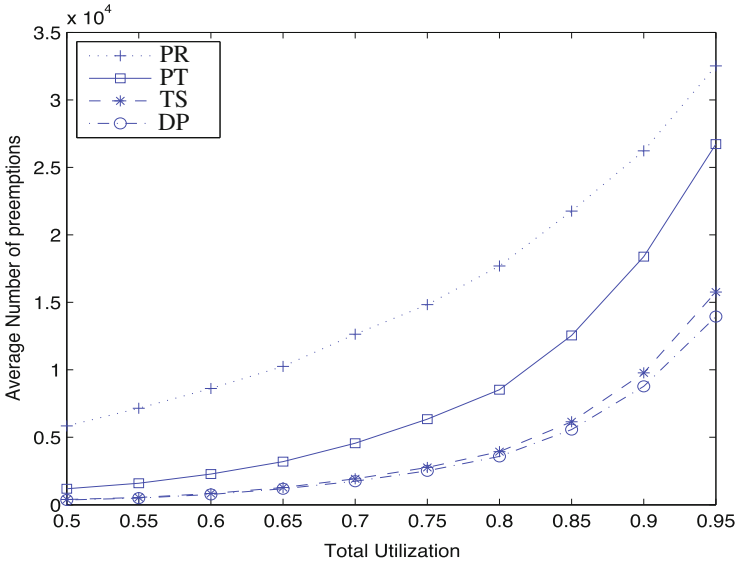
Finally, under cooperative scheduling, the number of preemptions can be simply upper bounded by the number of effective preemption points inserted in the task code.

Simulations experiments with randomly generated task sets have been carried out in [YBB10b] to better evaluate the effectiveness of the considered algorithms in reducing the number of preemptions. Figure 8.17a and b shows the simulation results obtained for a task set of 6 and 12 tasks, respectively, and reports the number of preemptions produced by each method as a function of the load.

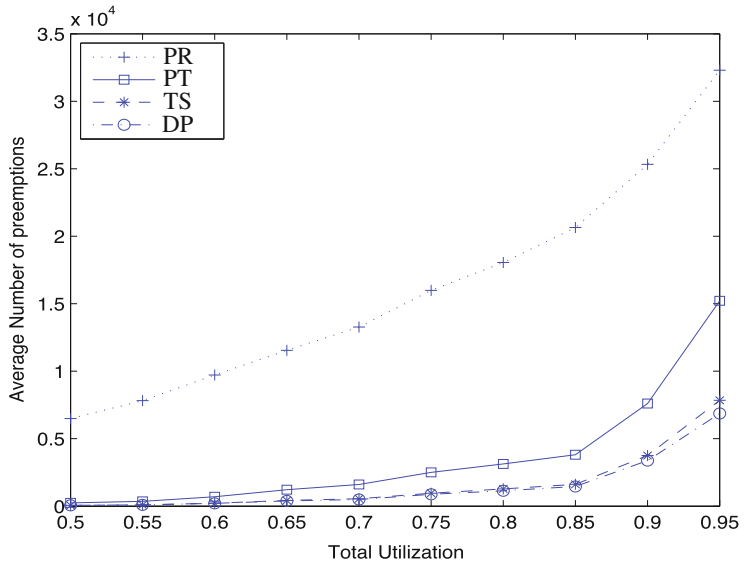
Each simulation run was performed on a set of n tasks with total utilization U varying from 0.5 to 0.95 with step 0.05. Individual utilizations U_i were uniformly distributed in [0,1], using the UUniFast algorithm [BB05]. Each computation time C_i was generated as a random integer uniformly distributed in [10, 50], and then T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$. The total simulation time was set to one million units of time. For each point in the graph, the result was computed by taking the average over 1000 runs.

All the task sets have been generated to be preemptive feasible, and the preemption cost was ignored, as also done in [WS99, YBB09]. Under preemption thresholds (PT), the algorithm proposed by Saksena and Wang [SW00] was used to find the maximum priority threshold that minimize the number of preemptions. Under deferred preemptions (DP) and task splitting (TS), the longest non-preemptive regions were computed according to the methods presented in Sects. 8.4.2 and 8.5.2, respectively. Finally, under task splitting, preemption points were inserted from the end of task code to the beginning.

As expected, fully preemptive scheduling (PR) generates the largest number of preemptions, while DP and TS are both able to achieve a higher reduction. PT has an intermediate behavior. Notice that DP can reduce slightly more preemptions than TS



(a)



(b)

Fig. 8.17 Average number of preemptions with different number of tasks. (**a**) Number of tasks: $n = 6$. (**b**) Number of tasks: $n = 12$

since, on the average, each preemption is deferred for a longer interval (always equal to Q_i, except when the preemption occur near the end of the task). However, it is important to consider that TS can achieve a lower and more predictable preemption

Table 8.3 Evaluation of limited preemption methods

	Implementation cost	Predictability	Effectiveness
Preemption thresholds	Low	Low	Medium
Deferred preemptions	Medium	Medium	High
Cooperative scheduling	Medium	High	High

cost, since preemption points can be suitably decided off-line with this purpose. As showed in the figures, PR produces a similar number of preemptions when the number of tasks increases, whereas all the other methods reduce the number of preemptions to an even higher degree. This is because, when n is larger, tasks have smaller individual utilization and thus can tolerate more blocking from lower priority tasks.

8.8 Conclusions

The results reported in this chapter can be summarized in Table 8.3, which compares the three presented methods in terms of the metrics presented above. As discussed in the previous section, the preemption threshold mechanism can reduce the overall number of preemptions with a low runtime overhead; however preemption cost cannot be easily estimated, since the position of each preemption, as well as the overall number of preemptions for each task, cannot be determined off-line. Using deferred preemptions, the number of preemptions for each task can be better estimated, but still the position of each preemption cannot be determined off line. Cooperative scheduling is the most predictable mechanism for estimating preemption costs, since both the number of preemptions and their positions are fixed and known from the task code. Its implementation, however, requires inserting explicit system calls in the source code that introduce additional overhead.

Exercises

8.1 Given the task set reported in the table, verify whether it is schedulable by the Rate-Monotonic algorithm in non-preemptive mode.

	C_i	T_i	D_i
τ_1	2	6	5
τ_2	2	8	6
τ_3	4	15	12

8.2 Given the task set reported in the table, verify whether it is schedulable by the Rate-Monotonic algorithm in non-preemptive mode.

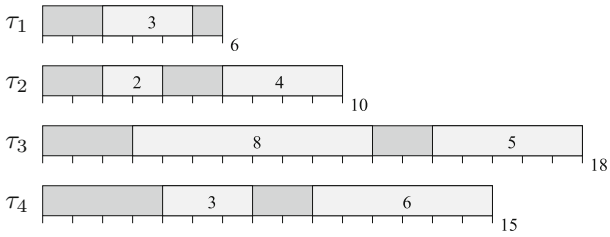
	C_i	T_i	D_i
τ_1	3	8	6
τ_2	3	9	8
τ_3	3	14	12
τ_4	2	80	80

8.3 Given the task set reported in the table, compute for each task τ_i the longest (floating) non-preemptive region Q_i that guarantees the schedulability under EDF.

	C_i	T_i
τ_1	2	10
τ_2	5	15
τ_3	5	30
τ_4	10	40
τ_5	2	60

8.4 For the same task set reported in Exercise 8.3, compute for each task τ_i the longest (floating) non-preemptive region Q_i that guarantees the schedulability under Rate Monotonic. Perform the computation using the Liu and Layland test.

8.5 Compute the worst-case response times produced by Rate Monotonic for the sporadic tasks illustrated below, where areas in light gray represent non-preemptive regions of code, whereas regions in dark gray are fully preemptable. The number inside a region denotes the worst-case execution time (WCET) of that portion of code, whereas the number on the right represents the WCET of the entire task.
Task periods are $T_1 = 24$, $T_2 = 30$, $T_3 = 120$, and $T_4 = 150$. Relative deadlines are equal to periods.



Chapter 9
Handling Overload Conditions

9.1 Introduction

This chapter deals with the problem of scheduling real-time tasks in overload conditions, that is, in those critical situations in which the computational demand requested by the task set exceeds the processor capacity, and hence, not all tasks can complete within their deadlines.

Overload conditions can occur for different causes, including:

- Bad system design. If a system is not designed or analyzed under pessimistic assumptions and worst-case load scenarios, it may work for most typical situations, but it can collapse in particular peak-load conditions, where too much computation is requested for the available computational resources.
- Simultaneous arrival of events. Even if the system is properly designed, the simultaneous arrival of several "unexpected" events could increase the load over the tolerated threshold.
- Malfunctioning of input devices. Sometimes, hardware defects in the acquisition boards or in some sensors could generate anomalous sequences of interrupts, saturating the processor bandwidth or delaying the application tasks after their deadlines.
- Unpredicted variations of the environmental conditions could generate a computational demand higher than that manageable by the processor under the specified timing requirements.
- Operating system exceptions. In some cases, anomalous combination of data could raise exceptions in the kernel, triggering the execution of high-priority handling routines that would delay the execution of application tasks.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_9

9.1.1 Load Definitions

In a real-time system, the definition of computational workload depends on the temporal characteristics of the computational activities. For non-real-time or soft real-time tasks, a commonly accepted definition of workload refers to the standard queueing theory, according to which a load ρ, also called *traffic intensity*, represents the expected number of job arrivals per mean service time. If $\overline{C}$ is the mean service time and λ is the average interarrival rate of the jobs, the average load can be computed as

$$\rho = \lambda \overline{C}.$$

Notice that this definition does not take deadlines into account; hence, it is not particularly useful to describe real-time workloads. In a hard real-time environment, a system is overloaded when, based on worst-case assumptions, there is no feasible schedule for the current task set, so one or more tasks will miss their deadline.

If the task set consists of n independent preemptable periodic tasks, whose relative deadlines are equal to their period, then the system load ρ is equivalent to the processor utilization factor:

$$\rho = U = \sum_{i=1}^{n} \frac{C_i}{T_i},$$

where C_i and T_i are the computation time and the period of task τ_i, respectively. In this case, a load $\rho > 1$ means that the total computation time requested by the periodic activities in their *hyperperiod* exceeds the available time on the processor; therefore, the task set cannot be scheduled by any algorithm.

For a generic set of real-time jobs that can be dynamically activated, the system load varies at each job activation and it is a function of the jobs' deadlines. In general, the load in a given interval $[t_a, t_b]$ can be defined as

$$\rho(t_a, t_b) = \max_{t_1, t_2 \in [t_a, t_b]} \frac{g(t_1, t_2)}{t_2 - t_1} \tag{9.1}$$

where $g(t_1, t_2)$ is the processor demand in the generic interval $[t_1, t_2]$. Such a definition, however, is of little practical use for load calculation, since the number of intervals in which the maximum has to be computed can be very high. Moreover, it is not clear how large the interval $[t_a, t_b]$ should be to estimate the overall system load.

A more practical definition that can be used to estimate the current load in dynamic real-time systems is the *instantaneous load* $\rho(t)$, proposed by Buttazzo and Stankovic in [BS95].

According to this method, the load is computed in all intervals from the current time t and each deadline (d_i) of the active jobs. Hence, the intervals that need to

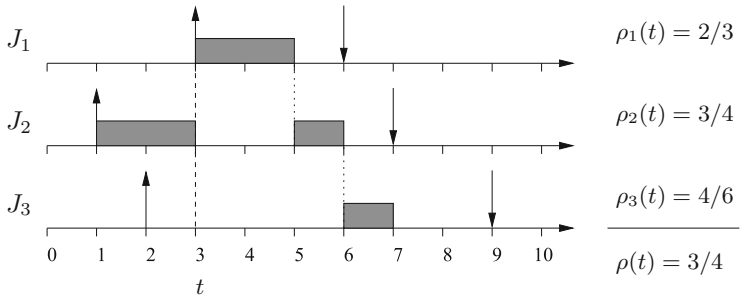


Fig. 9.1 Instantaneous load at time $t = 3$ for a set of three real-time jobs

be considered for the computation are $[t, d_1]$, $[t, d_2]$, ..., $[t, d_n]$. In each interval $[t, d_i]$, the partial load $\rho_i(t)$ due to the first i jobs is

$$\rho_i(t) = \frac{\sum_{d_k \le d_i} c_k(t)}{(d_i - t)}, \tag{9.2}$$

where $c_k(t)$ refers to the remaining execution time of job J_k with deadline less than or equal to d_i. Hence, the total load at time t is

$$\rho(t) = \max_i \rho_i(t). \tag{9.3}$$

Figure 9.1 shows an example of load calculation, at time $t = 3$, for a set of three real-time jobs. Then, Fig. 9.2 shows how the load varies as a function of time for the same set of jobs.

9.1.2 Terminology

When dealing with computational load, it is important to distinguish between *overload* and *overrun*.

Definition 9.1 A computing system is said to experience an **overload** when the computation time demanded by the task set in a certain interval of time exceeds the available processing time in the same interval.

Definition 9.2 A task (or a job) is said to experience an **overrun** when exceeding its expected utilization. An overrun may occur either because the next job is activated before its expected arrival time (*activation overrun*) or because the job computation time exceeds its expected value (*execution overrun*).

Note that, while the overload is a condition related to the processor, the overrun is a condition related to a task (or a single job). A task overrun does not necessarily

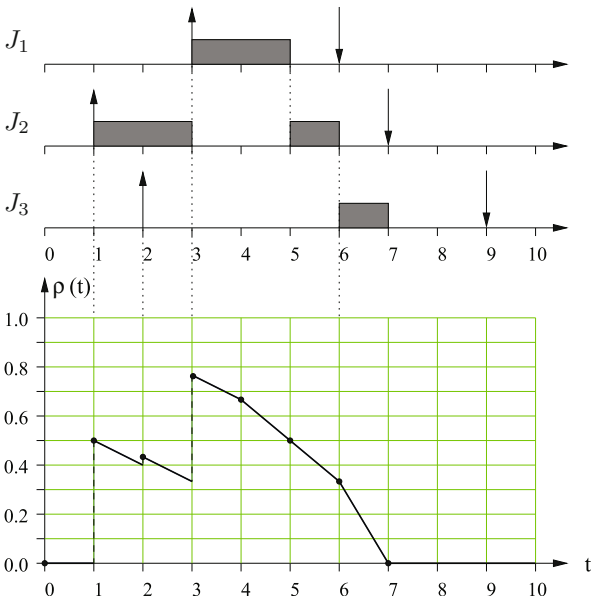


Fig. 9.2 Instantaneous load as a function of time for a set of three real-time jobs

cause an overload. However, a large unexpected overrun or a sequence of overruns can cause very unpredictable effects on the system, if not properly handled. In the following, we distinguish two types of overload conditions:

- **Transient overload**: It is an overload condition occurring for a limited duration, in a system in which the average load is less than or equal to one ($\overline{\rho} \leq 1$), but the maximum load is greater than one ($\rho^{max} > 1$).
- **Permanent overload**: It is an overload condition occurring for an unpredictable duration, in a system in which the average load is higher than one ($\overline{\rho} > 1$).

In a real-time computing system, a transient overload can be caused by a sequence of overruns, or by a bursty arrival of aperiodic requests, whereas a permanent overload condition typically occurs in periodic task systems when the total processor utilization exceeds one.

In the rest of this chapter, the following types of overload conditions will be analyzed:

- **Transient overloads due to aperiodic jobs**. This type of overload is typical of event-triggered systems consisting of many aperiodic jobs activated by external events. If the operating system is not designed to cope with excessive event arrivals, the effects of an overload can be unpredictable and cause serious problems on the controlled system. Experiments carried out by Locke [Loc86] have shown that EDF can rapidly degrade its performance during overload intervals, and there are cases in which the arrival of a new task can cause all the previous tasks to miss their deadlines. Such an undesirable phenomenon, called the *domino effect*, is depicted in Fig. 9.3. Figure 9.3a shows a feasible schedule

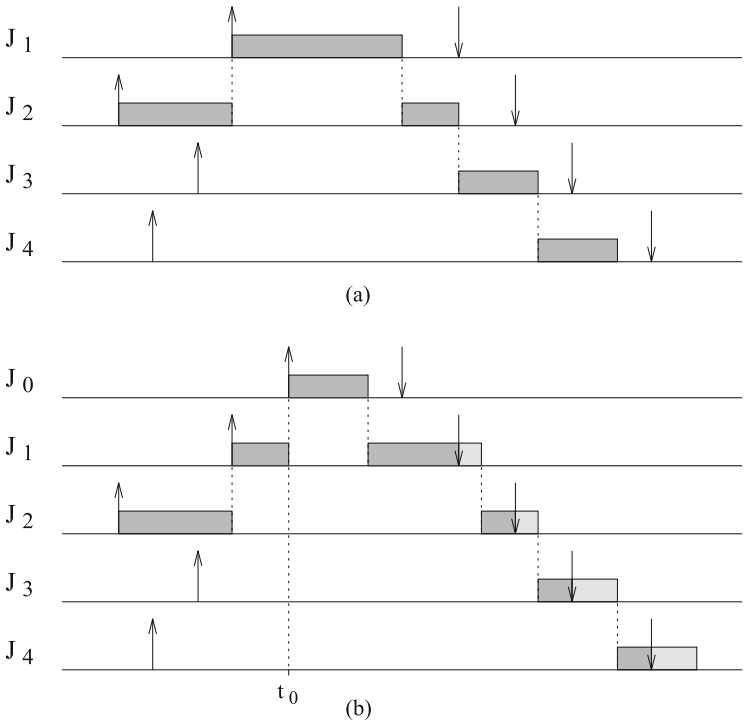
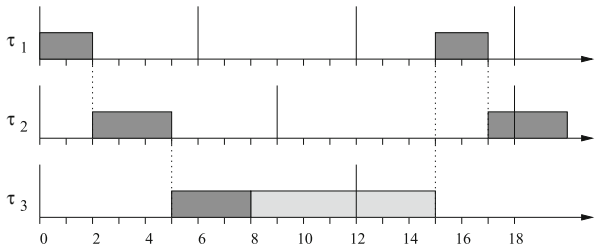


Fig. 9.3 Feasible schedule with earliest deadline first, in normal load condition (**a**). Overload with domino effect due to the arrival of task J_0 (**b**)

Fig. 9.4 Effect of an execution overrun in an EDF schedule



of a task set executed under EDF. However, if at time t_0 task J_0 is executed, all the previous tasks miss their deadlines (see Fig. 9.3b).

- **Transient overloads due to task overruns**. This type of overload can occur both in event-triggered and time-triggered systems, and it is due to periodic or aperiodic tasks that sporadically execute (or are activated) more than expected. Under rate monotonic, an overrun in a task τ_i does not affect tasks with higher priority, but any of the lower-priority task could miss its deadline. Under EDF, a task overrun can potentially affect all the other tasks in the system. Figure 9.4 shows an example of execution overrun in an EDF schedule.

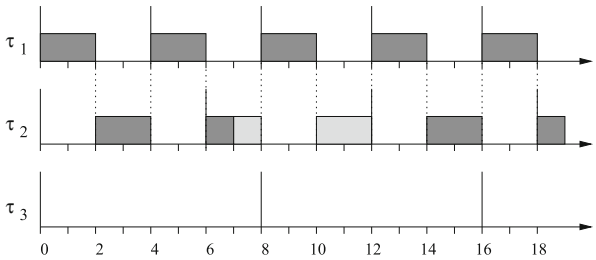


Fig. 9.5 Example of a permanent overload under rate monotonic: τ_2 misses its deadline and τ_3 can never execute

- **Permanent overloads in periodic task systems**. This type of overload occurs when the total utilization factor of the periodic task set is greater than one. This can happen either because the execution requirement of the task set was not correctly estimated, or because of some unexpected activation of new periodic tasks, or because some of the current tasks increased their activation rate to react to some change in the environment. In such a situation, computational activities start accumulating in the system's queues (which tend to become longer and longer, if the overload persists), and tasks' response times tend to increase indefinitely. Figure 9.5 shows the effect of a permanent overload condition in a rate monotonic schedule, where τ_2 misses its deadline and τ_3 can never execute.

9.2 Handling Aperiodic Overloads

In this section we consider event-driven systems where tasks arrive dynamically at unknown time instants. Each task consists of a single job, which is characterized by a fixed (known) computation time C_i and a relative deadline D_i. As a consequence, the overload in this systems can only be caused by the excessive number of tasks and can be detected at task activation times.

9.2.1 Performance Metrics

When tasks are activated dynamically and an overload occurs, there are no algorithms that can guarantee a feasible schedule of the task set. Since one or more tasks will miss their deadlines, it is preferable that late tasks be the less important ones in order to achieve graceful degradation. Hence, in overload conditions, distinguishing between time constraints and importance is crucial for the system. In general, the importance of a task is not related to its deadline or its period; thus, a task with a long deadline could be much more important than another one with an earlier deadline. For example, in a chemical process, monitoring the temperature every 10 s is certainly more important than updating the clock picture on the user console every second. This means that, during a transient overload, it is better to skip one

or more clock updates rather than miss the deadline of a temperature reading, since this could have a major impact on the controlled environment.

In order to specify importance, an additional parameter is usually associated with each task, its *value*, that can be used by the system to make scheduling decisions.

The value associated with a task reflects its importance with respect to the other tasks in the set. The specific assignment depends on the particular application. For instance, there are situations in which the value is set equal to the task computation time; in other cases, it is an arbitrary integer number in a given range; in other applications, it is set equal to the ratio of an arbitrary number (which reflects the importance of the task) and the task computation time; this ratio is referred to as the *value density*.

In a real-time system, however, the actual value of a task also depends on the time at which the task is completed; hence, the task importance can be better described by a utility function. For example, a non-real-time task, which has no time constraints, has a low constant value, since it always contributes to the system value, whenever it completes its execution. On the contrary, a hard task contributes to a value only if it completes within its deadline, and, since a deadline miss would jeopardize the behavior of the whole system, the value after its deadline can be considered minus infinity in many situations. A soft task can still give a value to the system if competed after its deadline, although this value may decrease with time. There are also real-time activities, so-called *firm*, that do not jeopardize the system, but give a negligible contribution if completed after their deadline. Figure 9.6 illustrates the utility functions of four different types of tasks.

Once the importance of each task has been defined, the performance of a scheduling algorithm can be measured by accumulating the values of the task utility functions computed at their completion time. Specifically, we define as *cumulative*

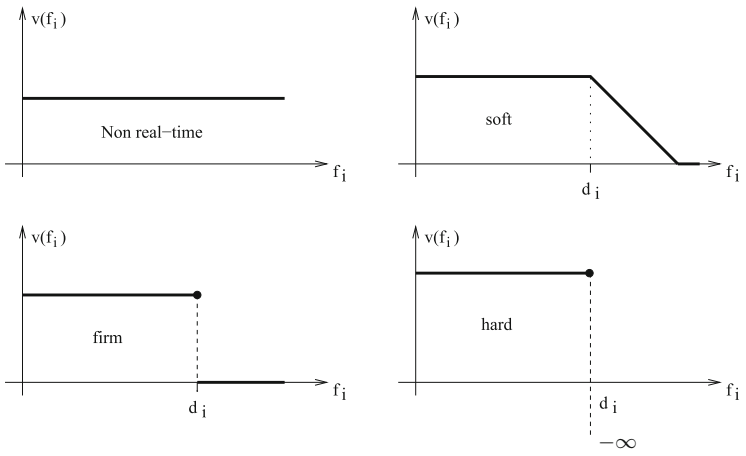


Fig. 9.6 Utility functions that can be associated to a task to describe its importance

value of a scheduling algorithm A the following quantity:

$$\Gamma_A = \sum_{i=1}^{n} v(f_i).$$

Given this metric, a scheduling algorithm is optimal if it maximizes the cumulative value achievable on a task set.

Notice that if a hard task misses its deadline, the cumulative value achieved by the algorithm is minus infinity, even though all other tasks completed before their deadlines. For this reason, all activities with hard timing constraints should be guaranteed a priori by assigning them dedicated resources (including processors). If all hard tasks are guaranteed a priori, the objective of a real-time scheduling algorithm should be to guarantee a feasible schedule in normal load conditions and maximize the cumulative value of soft and firm tasks during transient overloads.

Given a set of n jobs $J_i(C_i, D_i, V_i)$, where C_i is the worst-case computation time, D_i its relative deadline, and V_i the importance value gained by the system when the task completes within its deadline, the maximum cumulative value achievable on the task set is clearly equal to the sum of all values V_i; that is, $\Gamma_{max} = \sum_{i=1}^{n} V_i$. In overload conditions, this value cannot be achieved, since one or more tasks will miss their deadlines. Hence, if Γ^* is the maximum cumulative value that can be achieved by any algorithm on a task set in overload conditions, the performance of a scheduling algorithm A can be measured by comparing the cumulative value Γ_A obtained by A with the maximum achievable value Γ^*.

9.2.2 Online Versus Clairvoyant Scheduling

Since dynamic environments require online scheduling, it is important to analyze the properties and the performance of online scheduling algorithms in overload conditions. Although there exist optimal online algorithms in normal load conditions, it is easy to show that no optimal online algorithms exist in overload situations. Consider, for example, the task set shown in Fig. 9.7, consisting of three tasks $J_1(10, 11, 10)$, $J_2(6, 7, 6)$, $J_3(6, 7, 6)$.

Without loss of generality, we assume that the importance values associated to the tasks are proportional to their execution times ($V_i = C_i$) and that tasks are firm, so no value is accumulated if a task completes after its deadline. If J_1 and J_2 simultaneously arrive at time $t_0 = 0$, there is no way to maximize the cumulative value without knowing the arrival time of J_3. In fact, if J_3 arrives at time $t = 4$ or before, the maximum cumulative value is $\Gamma^* = 10$ and can be achieved by scheduling task J_1 (see Fig. 9.7a). However, if J_3 arrives between time $t = 5$ and time $t = 8$, the maximum cumulative value is $\Gamma^* = 12$, achieved by scheduling task J_2 and J_3 and discarding J_1 (see Fig. 9.7b). Notice that, if J_3 arrives at time $t = 9$ or later (see Fig. 9.7c), then the maximum cumulative value is $\Gamma^* = 16$ and can be

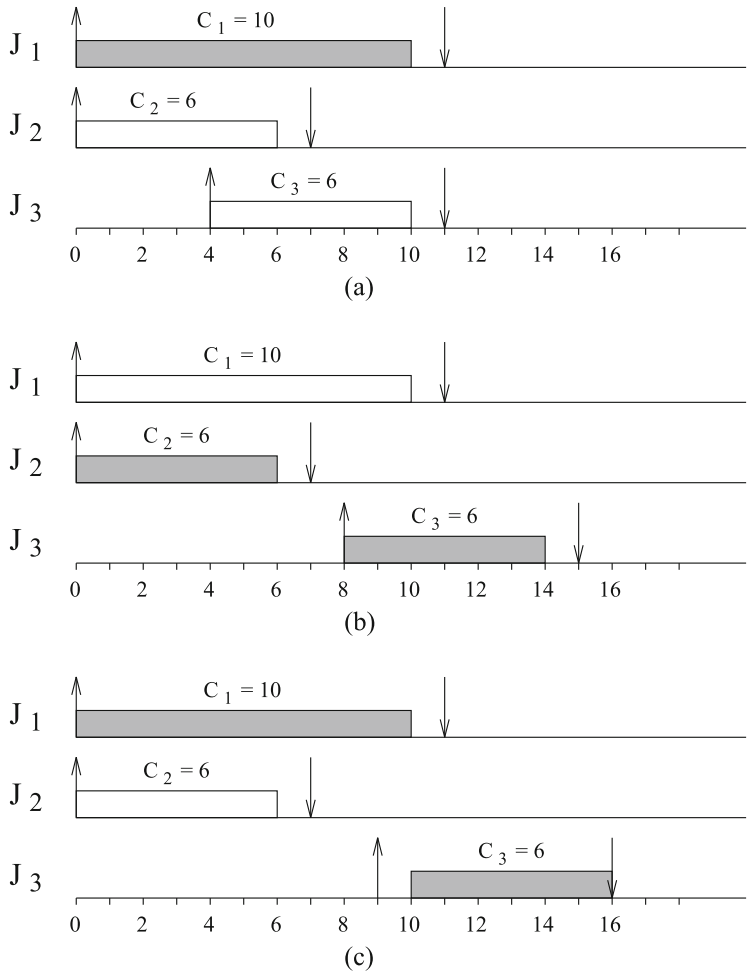


Fig. 9.7 No optimal online optimal algorithms exist in overload conditions, since the schedule that maximizes Γ depends on the knowledge of future arrivals: $\Gamma_{max} = 10$ in case (**a**), $\Gamma_{max} = 12$ in case (**b**), and $\Gamma_{max} = 16$ in case (**c**)

accumulated by scheduling tasks J_1 and J_3. Hence, at time $t = 0$, without knowing the arrival time of J_3, no online algorithm can decide which task to schedule for maximizing the cumulative value.

What this example shows is that, without an a priori knowledge of the task arrival times, no online algorithm can guarantee the maximum cumulative value Γ^*. This value can only be achieved by an ideal clairvoyant scheduling algorithm that knows the future arrival time of any task. Although the optimal clairvoyant scheduler is a pure theoretical abstraction, it can be used as a reference model to evaluate the performance of online scheduling algorithms in overload conditions.

9.2.3 Competitive Factor

The cumulative value obtained by a scheduling algorithm on a task set represents a measure of its performance for that particular task set. To characterize an algorithm with respect to worst-case conditions, however, the minimum cumulative value that can be achieved by the algorithm on any task set should be computed. A parameter that measures the worst-case performance of a scheduling algorithm is the *competitive factor*, introduced by Baruah et al. in [BKM+92].

Definition 9.3 A scheduling algorithm A has a *competitive factor* φ_A if and only if it can guarantee a cumulative value $\Gamma_A \geq \varphi_A \Gamma^*$, where Γ^* is the cumulative value achieved by the optimal clairvoyant scheduler.

From this definition, we note that the competitive factor is a real number $\varphi_A \in [0, 1]$. If an algorithm A has a competitive factor φ_A, it means that A can achieve a cumulative value Γ_A at least φ_A times the cumulative value achievable by the optimal clairvoyant scheduler on *any* task set.

If the overload has an infinite duration, then no online algorithm can guarantee a competitive factor greater than zero. In real situations, however, overloads are intermittent and usually have a short duration; hence, it is desirable to use scheduling algorithms with a high competitive factor.

Unfortunately, without any form of guarantee, the plain EDF algorithm has a zero competitive factor. To show these results it is sufficient to find an overload situation in which the cumulative value obtained by EDF can be arbitrarily small with respect to that one achieved by the clairvoyant scheduler. Consider the example shown in Fig. 9.8, where tasks have a value proportional to their computation time. This is an overload condition because both tasks cannot be completed within their deadlines.

When task J_2 arrives, EDF preempts J_1 in favor of J_2, which has an earlier deadline, so it gains a cumulative value of C_2. On the other hand, the clairvoyant scheduler always gains $C_1 > C_2$. Since the ratio C_2/C_1 can be made arbitrarily small, it follows that the competitive factor of EDF is zero.

An important theoretical result found in [BKM+92] is that there exists an upper bound on the competitive factor of any online algorithm. This is stated by the following theorem.

Theorem 9.1 (Baruah et al.) *In systems where the loading factor is greater than 2 ($\rho > 2$) and tasks' values are proportional to their computation times, no online algorithm can guarantee a competitive factor greater than* 0.25.

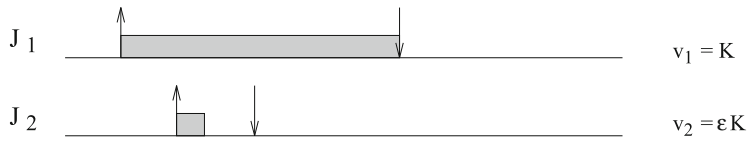


Fig. 9.8 Situation in which EDF has an arbitrarily small competitive factor

The proof of this theorem is done by using an adversary argument, in which the online scheduling algorithm is identified as a player and the clairvoyant scheduler as the adversary. In order to propose worst-case conditions, the adversary dynamically generates the sequence of tasks depending on the player decisions, to minimize the ratio Γ_A/Γ^*. At the end of the game, the adversary shows its schedule and the two cumulative values are computed. Since the player tries to do his best in worst-case conditions, the ratio of the cumulative values gives the upper bound of the competitive factor for any online algorithm.

9.2.3.1 Task Generation Strategy

To create an overload condition and force the hand of the player, the adversary creates two types of tasks: *major* tasks, of length C_i, and *associated* tasks, of length ϵ arbitrarily small. These tasks are generated according to the following strategy (see Fig. 9.9):

- All tasks have zero laxity; that is, the relative deadline of each task is exactly equal to its computation time.
- After releasing a major task J_i, the adversary releases the next major task J_{i+1} at time ϵ before the deadline of J_i; that is, $r_{i+1} = d_i - \epsilon$.
- For each major task J_i, the adversary may also create a sequence of associated tasks, in the interval $[r_i, d_i]$, such that each subsequent associated task is released at the deadline of the previous one in the sequence (see Fig. 9.9). Note that the resulting load is $\rho = 2$. Moreover, any algorithm that schedules any one of the associated tasks cannot schedule J_i within its deadline.
- If the player chooses to abandon J_i in favor of an associated task, the adversary stops the sequence of associated tasks.
- If the player chooses to schedule a major task J_i, the sequence of tasks terminates with the release of J_{i+1}.
- Since the overload must have a finite duration, the sequence continues until the release of J_m, where m is a positive finite integer.

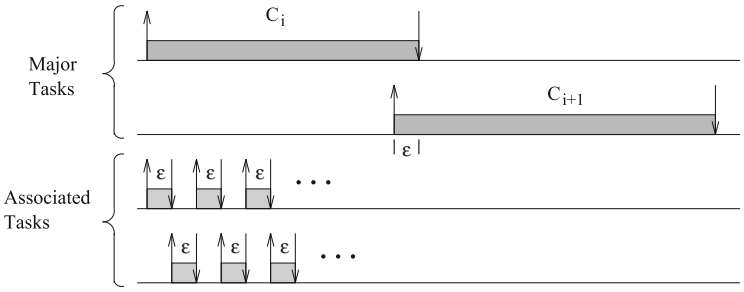


Fig. 9.9 Task sequence generated by the adversary

Notice that the sequence of tasks generated by the adversary is constructed in such a way that the player can schedule at most one task within its deadline (either a major task or an associated task). Clearly, since task values are equal to their computation times, the player never abandons a major task for an associated task, since it would accumulate a negligible value, that is, ϵ. On the other hand, the values of the major tasks (that is, their computation times) are chosen by the adversary to minimize the resulting competitive factor. To find the worst-case sequence of values for the major tasks, let

$$J_0, J_1, J_2, \ldots, J_i, \ldots, J_m$$

be the longest sequence of major tasks that can be generated by the adversary and, without loss of generality, assume that the first task has a computation time equal to $C_0 = 1$. Now, consider the following three cases.

Case 0. If the player decides to schedule J_0, the sequence terminates with J_1. In this case, the cumulative value gained by the player is C_0, whereas the one obtained by the adversary is $(C_0 + C_1 - \epsilon)$. Notice that this value can be accumulated by the adversary either by executing all the associated tasks or by executing J_0 and all associated tasks started after the release of J_1. Being ϵ arbitrarily small, it can be neglected in the cumulative value. Hence, the ratio among the two cumulative values is

$$\varphi_0 = \frac{C_0}{C_0 + C_1} = \frac{1}{1 + C_1} = \frac{1}{k}.$$

If $1/k$ is the value of this ratio ($k > 0$), then $C_1 = k - 1$.

Case 1. If the player decides to schedule J_1, the sequence terminates with J_2. In this case, the cumulative value gained by the player is C_1, whereas the one obtained by the adversary is $(C_0 + C_1 + C_2)$. Hence, the ratio among the two cumulative values is

$$\varphi_1 = \frac{C_1}{C_0 + C_1 + C_2} = \frac{k - 1}{k + C_2}.$$

In order not to lose with respect to the previous case, the adversary has to choose the value of C_2 so that $\varphi_1 \leq \varphi_0$; that is,

$$\frac{k - 1}{k + C_2} \leq \frac{1}{k},$$

which means

$$C_2 \geq k^2 - 2k.$$

However, observe that, if $\varphi_1 < \varphi_0$, the execution of J_0 would be more convenient for the player; thus, the adversary decides to make $\varphi_1 = \varphi_0$; that is,

$$C_2 = k^2 - 2k.$$

Case i. If the player decides to schedule J_i, the sequence terminates with J_{i+1}. In this case, the cumulative value gained by the player is C_i, whereas the one obtained by the adversary is $(C_0+C_1+\ldots+C_{i+1})$. Hence, the ratio among the two cumulative values is

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}}.$$

As in the previous case, to prevent any advantage to the player, the adversary will choose tasks' values so that

$$\varphi_i = \varphi_{i-1} = \ldots = \varphi_0 = \frac{1}{k}.$$

Thus,

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k},$$

and hence,

$$C_{i+1} = kC_i - \sum_{j=0}^{i} C_j.$$

Thus, the worst-case sequence for the player occurs when major tasks are generated with the following computation times:

$$\begin{cases} C_0 = 1 \\ C_{i+1} = kC_i - \sum_{j=0}^{i} C_j. \end{cases}$$

9.2.3.2 Proof of the Bound

Whenever the player chooses to schedule a task J_i, the sequence stops with J_{i+1} and the ratio of the cumulative values is

$$\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k}.$$

However, if the player chooses to schedule the last task J_m, the ratio of the cumulative values is

$$\varphi_m = \frac{C_m}{\sum_{j=0}^{m} C_j}.$$

Notice that if k and m can be chosen such that $\varphi_m \leq 1/k$, that is,

$$\frac{C_m}{\sum_{j=0}^{m} C_j} \leq \frac{1}{k}, \tag{9.4}$$

then we can conclude that, in the worst case, a player cannot achieve a cumulative value greater than $1/k$ times the adversary's value. Notice that

$$\frac{C_m}{\sum_{j=0}^{m} C_j} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + C_m} = \frac{C_m}{\sum_{j=0}^{m-1} C_j + kC_{m-1} - \sum_{j=0}^{m-1} C_j} = \frac{C_m}{kC_{m-1}}.$$

Hence, if there exists an m which satisfies Eq. (9.4), it also satisfies the following equation:

$$C_m \leq C_{m-1}. \tag{9.5}$$

Thus, (9.5) is satisfied if and only if (9.4) is satisfied.

From (9.4) we can also write

$$C_{i+2} = kC_{i+1} - \sum_{j=0}^{i+1} C_j$$

$$C_{i+1} = kC_i - \sum_{j=0}^{i} C_j,$$

and subtracting the second equation from the first one, we obtain

$$C_{i+2} - C_{i+1} = k(C_{i+1} - C_i) - C_{i+1},$$

that is,

$$C_{i+2} = k(C_{i+1} - C_i).$$

Hence, Eq. (9.4) is equivalent to

$$\begin{cases} C_0 = 1 \\ C_1 = k - 1 \\ C_{i+2} = k(C_{i+1} - C_i). \end{cases} \tag{9.6}$$

From this result, we can say that the tightest bound on the competitive factor of an online algorithm is given by the smallest ratio $1/k$ (equivalently, the largest k) such that (9.6) satisfies (9.5). Equation (9.6) is a recurrence relation that can be solved by standard techniques [Sha85]. The characteristic equation of (9.6) is

$$x^2 - kx + k = 0,$$

which has roots

$$x_1 = \frac{k + \sqrt{k^2 - 4k}}{2} \qquad \text{and} \qquad x_2 = \frac{k - \sqrt{k^2 - 4k}}{2}.$$

When $k = 4$, we have

$$C_i = d_1 i 2^i + d_2 2^i, \tag{9.7}$$

and when $k \neq 4$ we have

$$C_i = d_1 (x_1)^i + d_2 (x_2)^i, \tag{9.8}$$

where values for d_1 and d_2 can be found from the boundary conditions expressed in (9.6). We now show that for ($k = 4$) and ($k > 4$) C_i will diverge, so Eq. (9.5) will not be satisfied, whereas for ($k < 4$) C_i will satisfy (9.5).
Case ($k = 4$). In this case, $C_i = d_1 i 2^i + d_2 2^i$ and, from the boundary conditions, we find $d_1 = 0.5$ and $d_2 = 1$. Thus,

$$C_i = (\frac{i}{2} + 1) 2^i,$$

which clearly diverges. Hence, for $k = 4$, Eq. (9.5) cannot be satisfied.
Case ($k > 4$). In this case, $C_i = d_1 (x_1)^i + d_2 (x_2)^i$, where

$$x_1 = \frac{k + \sqrt{k^2 - 4k}}{2} \quad \text{and} \quad x_2 = \frac{k - \sqrt{k^2 - 4k}}{2}.$$

From the boundary conditions we find

$$\begin{cases} C_0 = d_1 + d_2 = 1 \\ C_1 = d_1 x_1 + d_2 x_2 = k - 1 \end{cases}$$

that is,

$$\begin{cases} d_1 = \frac{1}{2} + \frac{k-2}{2\sqrt{k^2-4k}} \\ d_2 = \frac{1}{2} - \frac{k-2}{2\sqrt{k^2-4k}}. \end{cases}$$

Since $(x_1 > x_2)$, $(x_1 > 2)$, and $(d_1 > 0)$, C_i will diverge, and hence, also for $k > 4$, Eq. (9.5) cannot be satisfied.

Case $(k < 4)$. In this case, since $(k^2 - 4k < 0)$, both the roots x_1, x_2 and the coefficients d_1, d_2 are complex conjugates, so they can be represented as follows:

$$\begin{cases} d_1 = se^{j\theta} \\ d_2 = se^{-j\theta} \end{cases} \qquad \begin{cases} x_1 = re^{j\omega} \\ x_2 = re^{-j\omega}, \end{cases}$$

where s and r are real numbers, $j = \sqrt{-1}$, and θ and ω are angles such that, $-\pi/2 < \theta < 0, 0 < \omega < \pi/2$. Equation (9.8) may therefore be rewritten as

$$\begin{aligned} C_i &= se^{j\theta} r^i e^{ji\omega} + se^{-j\theta} r^i e^{-ji\omega} = \\ &= sr^i [e^{j(\theta+i\omega)} + e^{-j(\theta+i\omega)}] = \\ &= sr^i [\cos(\theta + i\omega) + j \sin(\theta + i\omega) + \cos(\theta + i\omega) - j \sin(\theta + i\omega)] = \\ &= 2sr^i \cos(\theta + i\omega). \end{aligned}$$

Being $\omega \neq 0$, $\cos(\theta + i\omega)$ is negative for some $i \in \mathbf{N}$, which implies that there exists a finite m that satisfies (9.5).

Since (9.5) is satisfied for $k < 4$, the largest k that determines the competitive factor of an online algorithm is certainly less than 4. Therefore, we can conclude that $1/4$ is an upper bound on the competitive factor that can be achieved by any online scheduling algorithm in an overloaded environment. Hence, Theorem 9.1 follows.

9.2.3.3 Extensions

Theorem 9.1 establishes an upper bound on the competitive factor of online scheduling algorithms operating in heavy load conditions ($\rho > 2$). In lighter overload conditions ($1 < \rho \leq 2$), the bound is a little higher, and it is given by the following theorem [BR91].

Theorem 9.2 (Baruah et al.) *In systems with a loading factor* ρ, $1 < \rho \leq 2$, *and task values equal to computation times, no online algorithm can guarantee a competitive factor greater than* p, *where* p *satisfies*

$$4[1 - (\rho - 1)p]^3 = 27p^2. \tag{9.9}$$

Notice that, for $\rho = 1 + \epsilon$, Eq. (9.9) is satisfied for $p = \sqrt{4/27} \simeq 0.385$, whereas, for $\rho = 2$, the same equation is satisfied for $p = 0.25$.

In summary, whenever the system load does not exceed one, the upper bound of the competitive factor is obviously one. As the load exceeds one, the bound immediately falls to 0.385, and as the load increases from one to two, it falls from 0.385 to 0.25. For loads higher than two, the competitive factor limitation remains at 0.25. The bound on the competitive factor as a function of the load is shown in Fig. 9.10.

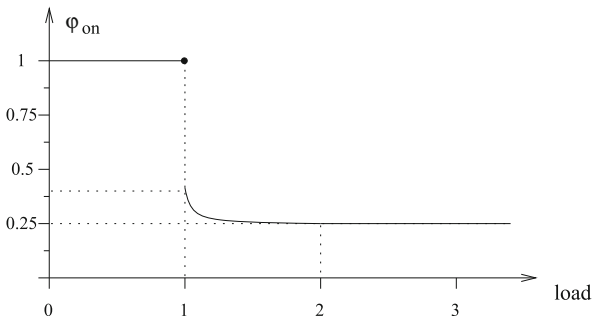


Fig. 9.10 Bound of the competitive factor of an online scheduling algorithm as a function of the load

Baruah et al. [BR91] also showed that, when using value density metrics (where the value density of a task is its value divided by its computation time), the best that an online algorithm can guarantee in environments with load $\rho > 2$ is

$$\frac{1}{(1+\sqrt{k})^2},$$

where k is the important ratio between the highest and the lowest value density task in the system.

In environments with a loading factor ρ, $1 < \rho \leq 2$, and an importance ratio k, two cases must be considered. Let $q = k(\rho - 1)$. If $q \geq 1$, then no online algorithm can achieve a competitive factor greater than

$$\frac{1}{(1+\sqrt{q})^2},$$

whereas, if $q < 1$, no online algorithm can achieve a competitive factor greater than p, where p satisfies

$$4(1-qp)^3 = 27p^2.$$

Before concluding the discussion on the competitive analysis, it is worth pointing out that all the above bounds are derived under very restrictive assumptions, such as all tasks have zero laxity, the overload can have an arbitrary (but finite) duration, and task's execution time can be arbitrarily small. In most real-world applications, however, task characteristics are much less restrictive; therefore, the 0.25 bound has only a theoretical validity, and more work is needed to derive other bounds based on more realistic assumptions on the actual load conditions. An analysis of online scheduling algorithms under different types of adversaries has been presented by Karp [Kar92].

9.2.4 *Typical Scheduling Schemes*

With respect to the strategy used to predict and handle overloads, most of the scheduling algorithms proposed in the literature can be divided into three main classes, illustrated in Fig. 9.11:

- **Best effort**. This class includes those algorithms with no prediction for overload conditions. At its arrival, a new task is always accepted into the ready queue, so the system performance can only be controlled through a proper priority assignment which takes task values into account.
- **With acceptance test**. This class includes those algorithms with admission control, performing a guarantee test at every job activation. Whenever a new task enters the system, a guarantee routine verifies the schedulability of the task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted in the system; otherwise, it is rejected.

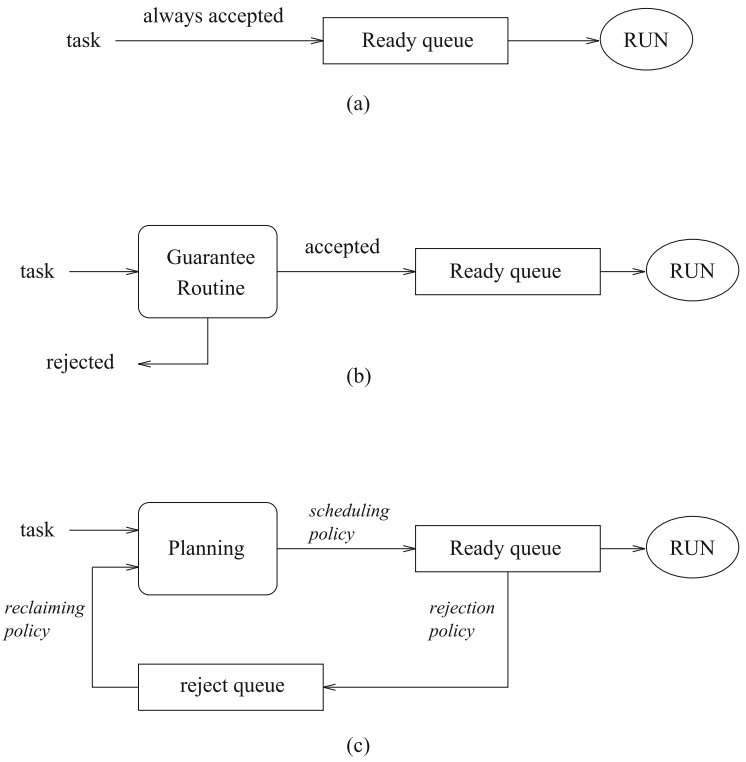


Fig. 9.11 Scheduling schemes for handling overload situations: best effort (**a**), with acceptance test (**b**), and robust (**c**)

- **Robust**. This class includes those algorithms that separate timing constraints and importance by considering two different policies: one for task acceptance and one for task rejection. Typically, whenever a new task enters the system, an acceptance test verifies the schedulability of the new task set based on worst-case assumptions. If the task set is found schedulable, the new task is accepted in the ready queue; otherwise, one or more tasks are rejected based on a different policy, aimed at maximizing the cumulative value of the feasible tasks.

In addition, an algorithm is said to be *competitive* if it has a competitive factor greater than zero.

Notice that the simple guarantee scheme is able to avoid domino effects by sacrificing the execution of the newly arrived task. Basically, the acceptance test acts as a filter that controls the load on the system and always keeps it less than one. Once a task is accepted, the algorithm guarantees that it will complete by its deadline (assuming that no task will exceed its estimated worst-case computation time). The acceptance test, however, does not take task importance into account and, during transient overloads, always rejects the newly arrived task, regardless of its value. In certain conditions (such as when tasks have very different importance levels), this scheduling strategy may exhibit poor performance in terms of cumulative value, whereas a robust algorithm can be much more effective.

When the load is controlled by job rejection, a reclaiming mechanism can be used to take advantage of those tasks that complete before their worst-case finishing time. To reclaim the spare time, rejected tasks will not be removed, but temporarily parked in a queue, from which they can be possibly recovered whenever a task completes before its worst-case finishing time.

In the real-time literature, several scheduling algorithms have been proposed to deal with transient overloads in event-triggered systems. Ramamritham and Stankovic [RS84] used EDF to dynamically guarantee incoming work via online planning. Locke [Loc86] proposed a best effort algorithm using EDF with a rejection policy based on tasks' value density. Biyabani et al. [BSR88] extended the work of Ramamritham and Stankovic to tasks with different values, and various policies were studied to decide which tasks should be dropped when a newly arriving task could not be guaranteed. Haritsa, Livny, and Carey [HLC91] presented the use of a feedback-based EDF algorithm for real-time database systems.

In real-time Mach [TWW87] tasks were ordered by EDF and overload was predicted using a statistical guess. If overload was predicted, tasks with least value were dropped.

Other general results on overload in real-time systems were also derived. For example, Sha [SLR88] showed that the rate monotonic algorithm has poor properties in overload. Thambidurai and Trivedi [TT89] studied transient overloads in fault-tolerant real-time systems, building and analyzing a stochastic model for such systems. Schwan and Zhou [SZ92] did online guarantees based on keeping a slot list and searching for free-time intervals between slots. Once schedulability is determined in this fashion, tasks are actually dispatched using EDF. If a new task cannot be guaranteed, it is discarded.

Zlokapa, Stankovic, and Ramamritham [Zlo93] proposed an approach called *well-time scheduling*, which focuses on reducing the guarantee overhead in heavily loaded systems by delaying the guarantee. Various properties of the approach were developed via queueing theoretic arguments, and the results were a multilevel queue (based on an analytical derivation), similar to that found in [HLC91] (based on simulation).

In the following sections we present two specific examples of scheduling algorithms for handling overload situations and then compare their performance for different peak load conditions.

9.2.5 The RED Algorithm

RED (robust earliest deadline) is a robust scheduling algorithm proposed by Buttazzo and Stankovic [BS93, BS95] for dealing with firm aperiodic tasks in overloaded environments. The algorithm synergistically combines many features including graceful degradation in overloads, deadline tolerance, and resource reclaiming. It operates in normal and overload conditions with excellent performance, and it is able to predict not only deadline misses but also the size of the overload, its duration, and its overall impact on the system.

In RED, each task $J_i(C_i, D_i, M_i, V_i)$ is characterized by four parameters: a worst-case execution time (C_i), a relative deadline (D_i), a deadline tolerance (M_i), and an importance value (V_i). The deadline tolerance is the amount of time by which a task is permitted to be late, that is, the amount of time that a task may execute after its deadline and still produce a valid result. This parameter can be useful in many real applications, such as robotics and multimedia systems, where the deadline timing semantics is more flexible than scheduling theory generally permits.

Deadline tolerances also provide a sort of compensation for the pessimistic evaluation of the worst-case execution time. For example, without tolerance, a task could be rejected although the system could be scheduled within the tolerance levels.

In RED, the primary deadline plus the deadline tolerance provides a sort of secondary deadline, used to run the acceptance test in overload conditions. Notice that having a tolerance greater than zero is different than having a longer deadline. In fact, tasks are scheduled based on their primary deadline but accepted based on their secondary deadline. In this framework, a schedule is said to be *strictly feasible* if all tasks complete before their primary deadline, whereas it is said to be *tolerant* if there exists some task that executes after its primary deadline but completes within its secondary deadline.

The guarantee test performed in RED is formulated in terms of residual laxity. The residual laxity L_i of a task is defined as the interval between its estimated finishing time f_i and its primary (absolute) deadline d_i. Each residual laxity can be efficiently computed using the result of the following lemma.

Lemma 9.1 *Given a set $J = \{J_1, J_2, \ldots, J_n\}$ of active aperiodic tasks ordered by increasing primary (absolute) deadline, the residual laxity L_i of each task J_i at time t can be computed as*

$$L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t), \tag{9.10}$$

where $L_0 = 0$, $d_0 = t$ (the current time), and $c_i(t)$ is the remaining worst-case computation time of task J_i at time t.

Proof By definition, a residual laxity is $L_i = d_i - f_i$. Since tasks are ordered by increasing deadlines, J_1 is executing at time t, and its estimated finishing time is given by the current time plus its remaining execution time ($f_1 = t + c_1$). As a consequence, L_1 is given by

$$L_1 = d_1 - f_1 = d_1 - t - c_1.$$

Any other task J_i, with $i > 1$, will start as soon as J_{i-1} completes and will finish c_i units of time after its start ($f_i = f_{i-1} + c_i$). Hence, we have

$$\begin{aligned} L_i = d_i - f_i &= d_i - f_{i-1} - c_i = d_i - (d_{i-1} - L_{i-1}) - c_i = \\ &= L_{i-1} + (d_i - d_{i-1}) - c_i, \end{aligned}$$

and the lemma follows. □

Notice that if the current task set J is schedulable and a new task J_a arrives at time t, the feasibility test for the new task set $J' = J \cup \{J_a\}$ requires to compute only the residual laxity of task J_a and that one of those tasks J_i such that $d_i > d_a$. This is because the execution of J_a does not influence those tasks having deadline less than or equal to d_a, which are scheduled before J_a. It follows that the acceptance test has $O(n)$ complexity in the worst case.

To simplify the description of the RED guarantee test, we define the *exceeding time* E_i as the time that task J_i executes after its secondary deadline:[1]

$$E_i = \max(0, -(L_i + M_i)). \tag{9.11}$$

We also define the *maximum exceeding time* E_{max} as the maximum among all E_i's in the task set; that is, $E_{max} = \max_i(E_i)$. Clearly, a schedule will be strictly feasible if and only if $L_i \geq 0$ for all tasks in the set, whereas it will be tolerant if and only if there exists some $L_i < 0$, but $E_{max} = 0$.

By this approach we can identify which tasks will miss their deadlines and compute the amount of processing time required above the capacity of the system—the maximum exceeding time. This global view allows to plan an action to recover

[1] If $M_i = 0$, the *exceeding time* is also called the *tardiness*.

Algorithm: RED Acceptance Test
Input: A task set $\mathcal{J}$ with $\{C_i, D_i, V_i, M_i\}, \forall J_i \in \mathcal{J}$
Output: A schedulable task set
// Assumes deadlines are ordered by decreasing values

```
(1)  begin
(2)      E = 0;                          // Maximum Exceeding Time
(3)      L_0 = 0;
(4)      d_0 = current_time();
(5)      J' = J ∪ {J_new};
(6)      k = <position of J_new in the task set J'>;
(7)      for (each task J'_i such that i ≥ k) do
(8)          L_i = L_{i-1} + (d_i − d_{i-1}) − c_i;
(9)          if (L_i + M_i < −E) then          // compute E_max
(10)             E = −(L_i + M_i);
(11)         end
(12)     end
(13)     if (E > 0) then
(14)         <select a set J* of least-value tasks to be rejected>;
(15)         <reject all task in J*>;
(16)     end
(17) end
```

Fig. 9.12 The RED acceptance test

from the overload condition. Many recovering strategies can be used to solve this problem. The simplest one is to reject the least-value task that can remove the overload situation. In general, we assume that, whenever an overload is detected, some rejection policy will search for a subset J^* of least-value tasks that will be rejected to maximize the cumulative value of the remaining subset. The RED acceptance test is shown in Fig. 9.12.

A simple rejection strategy consists in removing the task with the smallest value that resolves the overload. To quickly identify the task to be rejected, we can keep track of the *first exceeding task*, denoted as J_{FET}, which is the task with the earliest primary deadline that misses its secondary deadline. The FET index can be easily determined within the loop in which residual laxities are computed. Note that in order to resolve the overload, the task to be rejected must have a residual computation time greater than or equal to the maximum exceeding time and a primary deadline less than d_{FET}. Hence, the rejection strategy can be expressed as follows:

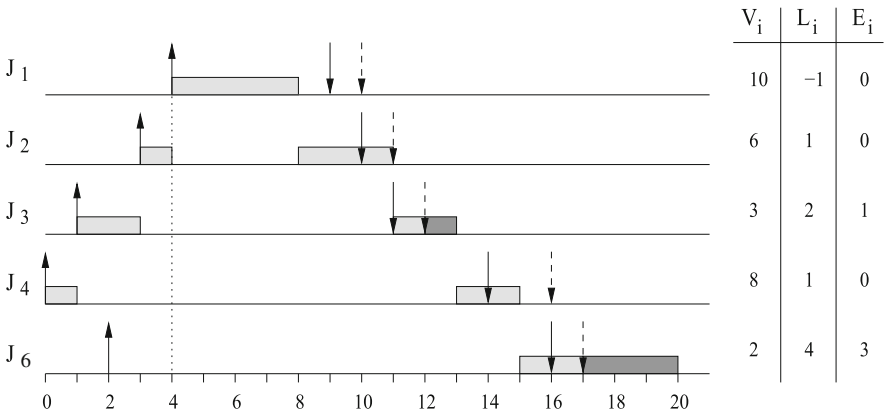


Fig. 9.13 Example of overload in a task set with deadline tolerances

Reject the task J_r with the least value, such that

$$(r \leq \text{FET}) \quad \text{and} \quad (c_r(t) \geq E_{max})$$

To better understand the rejection strategy, consider the example illustrated in Fig. 9.13, where secondary deadlines are drawn with dashed arrows. As can be easily verified, before the arrival of J_1 the task set $\{J_2, J_3, J_4, J_5\}$ is strictly feasible.

At time $t = 4$, when J_1 arrives, an overload occurs because both J_3 and J_5 would terminate after their secondary deadline. The least value task able to resolve the overload is J_2. In fact, J_5, which has the smallest value, cannot be rejected because, having a long primary deadline, it would not advance the execution of J_3. Also, rejecting J_3 would not solve the overload, since its residual computation time is not sufficient to advance the completion of J_5 before the deadline.

A more efficient rejection strategy could consider rejecting more than one task to minimize the cumulative value of the rejected tasks. For example, rejecting J_3 and J_5 is better than rejecting J_2. However, minimizing the value of the rejected tasks requires a combinatorial search that would be too expensive to be performed online for large task sets.

To take advantage of early completions and reduce the pessimism of the acceptance test, some algorithms use an online reclaiming mechanism that exploits the saved time to possibly recover previously rejected tasks. For example, in RED, a rejected task is not removed from the system, but it is temporarily parked in a *reject queue*, with the hope that it can be recovered due to some early completion. If δ is the time saved by the running task, then all the residual laxities will increase by δ, and some of the rejected tasks may be recovered based on their value.

9.2.6 D_{over}: A Competitive Algorithm

Koren and Shasha [KS92] found an online scheduling algorithm, called D^{over}, which has been proved to be optimal, in the sense that it gives the best competitive factor achievable by any online algorithm (that is, 0.25).

As long as no overload is detected, D^{over} behaves like EDF. An overload is detected when a ready task reaches its *latest start time (LST)*, that is, the time at which the task's remaining computation time is equal to the time remaining until its deadline. At this time, some task must be abandoned: either the task that reached its LST or some other task.

In D_{over}, the set of ready tasks is partitioned in two disjoint sets: *privileged* tasks and *waiting* tasks. Whenever a task is preempted, it becomes a *privileged* task. However, whenever some task is scheduled as the result of a LST, all the ready tasks (whether preempted or never executed) become *waiting* tasks.

When an overload is detected because a task J_z reaches its LST, then the value of J_z is compared against the total value V_{priv} of all the privileged tasks (including the value v_{curr} of the currently running task). If

$$v_z > (1 + \sqrt{k})(v_{curr} + V_{priv})$$

(where k is ratio of the highest value density and the lowest value density task in the system), then J_z is executed; otherwise, it is abandoned. If J_z is executed, all the privileged tasks become waiting tasks. Task J_z can in turn be abandoned in favor of another task J_x that reaches its LST, but only if

$$v_x > (1 + \sqrt{k})v_z.$$

It is worth to observe that having the best competitive factor among all online algorithms does not mean having the best performance in *any* load condition. In fact, in order to guarantee the best competitive factor, D^{over} may reject tasks with values higher than the current task but not higher than the threshold that guarantees optimality. In other words, to cope with worst-case sequences, D^{over} does not take advantage of lucky sequences and may reject more value than it is necessary. In Sect. 9.2.7, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

9.2.7 Performance Evaluation

In this section, the performance of the scheduling algorithms described above is tested through simulation using a synthetic workload. Each plot on the graphs represents the average of a set of 100 independent simulations, the duration of each is chosen to be 300,000 time units long. The algorithms are executed on

task sets consisting of 100 aperiodic tasks, whose parameters are generated as follows. The worst-case execution time C_i is chosen as a random variable with uniform distribution between 50 and 350 time units. The interarrival time T_i is modeled as a random variable with a Poisson distribution with average value equal to $T_i = NC_i/\rho$, where N is the total number of tasks and ρ is the average load. The laxity of a task is computed as a random value with uniform distribution from 150 and 1850 time units, and the relative deadline is computed as the sum of its worst-case execution time and its laxity. The task value is generated as a random variable with uniform distribution ranging from 150 to 1850 time units, as for the laxity.

The first experiment illustrates the effectiveness of the guaranteed (GED) and robust scheduling paradigm (RED) with respect to the best-effort scheme, under the EDF priority assignment. In particular, it shows how the pessimistic assumptions made in the guarantee test affect the performance of the algorithms and how much a reclaiming mechanism can compensate for this degradation. In order to test these effects, tasks were generated with actual execution times less than their worst-case values. The specific parameter varied in the simulations was the average *unused computation time ratio*, defined as

$$\beta = 1 - \frac{\text{Actual computation time}}{\text{Worst-case computation time}}.$$

Note that, if ρ_n is the *nominal* load estimated based on the worst-case computation times, the *actual* load ρ is given by

$$\rho = \rho_n(1 - \beta).$$

In the graphs reported in Fig. 9.14, the task set was generated with a nominal load $\rho_n = 3$, while β was varied from 0.125 to 0.875. As a consequence, the actual mean load changed from a value of 2.635 to a value of 0.375, thus ranging over very different actual load conditions. The performance was measured by computing the *hit value ratio (HVR)*, that is, the ratio of the cumulative value achieved by an algorithm and the total value of the task set. Hence, $HVR = 1$ means that all the tasks completed within their deadlines and no tasks were rejected.

For small values of β, that is, when tasks execute for almost their maximum computation time, the guaranteed (GED) and robust (RED) versions are able to obtain a significant improvement compared to the plain EDF scheme. Increasing the unused computation time, however, the actual load falls down and the plain EDF performs better and better, reaching the optimality in underload conditions. Notice that as the system becomes underloaded ($\beta \simeq 0.7$), GED becomes less effective than EDF. This is due to the fact that GED performs a worst-case analysis, thus rejecting tasks that still have some chance to execute within their deadline. This phenomenon does not appear on RED, because the reclaiming mechanism implemented in the robust scheme is able to recover the rejected tasks whenever possible.

In the second experiment, D_{over} is compared against two robust algorithms: RED (robust earliest deadline) and RHD (robust high density). In RHD, the task

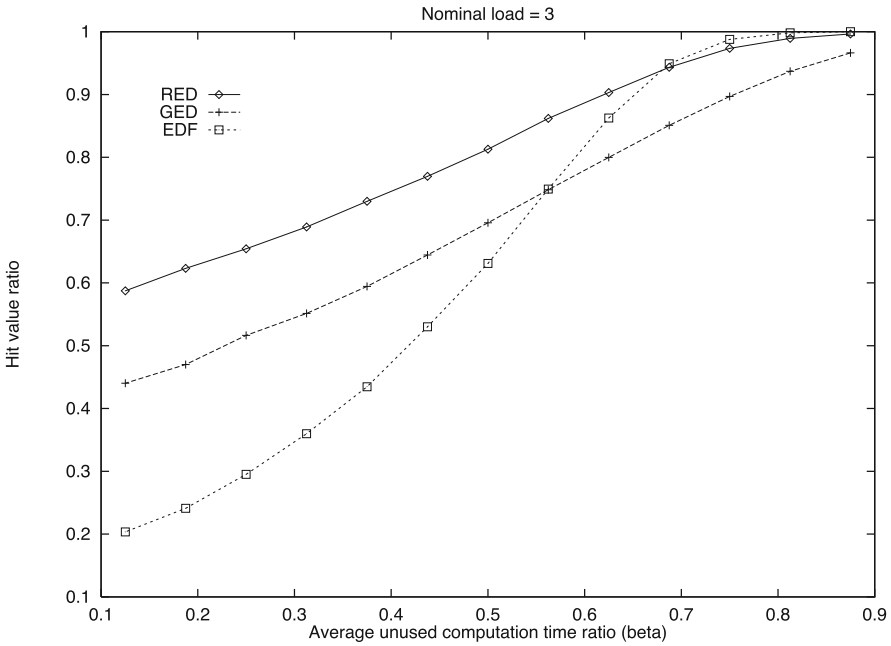


Fig. 9.14 Performance of various EDF scheduling schemes: best-effort (EDF), guaranteed (GED) and robust (RED)

with the highest value density (v_i/C_i) is scheduled first, regardless of its deadline. Performance results are shown in Fig. 9.15.

Notice that in underload conditions D_{over} and RED exhibit optimal behavior ($HVR = 1$), whereas RHD is not able to achieve the total cumulative value, since it does not take deadlines into account. However, for high load conditions ($\rho > 1.5$), RHD performs even better than RED and D_{over}.

In particular, for random task sets, D_{over} is less effective than RED and RHD for two reasons: First, it does not have a reclaiming mechanism for recovering rejected tasks in the case of early completions; second, the threshold value used in the rejection policy is set to reach the best competitive factor in a worst-case scenario. But this means that for random sequences D_{over} may reject tasks that could increase the cumulative value, if executed.

In conclusion, we can say that in overload conditions no online algorithm can achieve optimal performance in terms of cumulative value. Competitive algorithms are designed to guarantee a *minimum* performance in any load condition, but they cannot guarantee the best performance for all possible scenarios. For random task sets, robust scheduling schemes appear to be more appropriate.

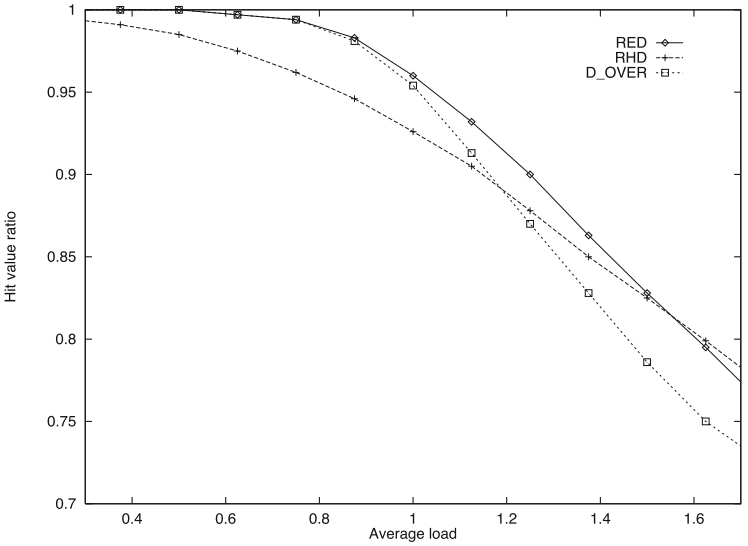


Fig. 9.15 Performance of D_{over} against RED and RHD

9.3 Handling Overruns

This section presents some methodology for handling transient overload conditions caused by tasks that execute more than expected or are activated more frequently than expected. This could happen either because some task parameter was wrongly estimated or because the system was intentionally designed under less pessimistic assumptions to achieve a higher average utilization.

If not properly handled, task overruns can cause serious problems in the real-time system, jeopardizing the guarantee performed for the critical tasks and causing an abrupt performance degradation. An example of negative effects of an execution overrun in EDF was already illustrated in Fig. 9.4.

To prevent an overrun to introduce unbounded delays on tasks' execution, the system could either decide to abort the current job experiencing the overrun or let it continue with a lower priority. The first solution is not safe, because the job could be in a critical section when aborted, thus leaving a shared resource with inconsistent data (very dangerous). The second solution is much more flexible, since the degree of interference caused by the overrun on the other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation. A general technique for implementing such a solution is the resource reservation approach.

9.3.1 Resource Reservation

Resource reservation is a general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times [MST93, MST94b, AB98, AB04]. According to this method, each task is assigned a fraction of the processor bandwidth, just enough to satisfy its timing constraints. The kernel, however, must prevent each task to consume more than the requested amount to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the behavior of the other tasks.

A simple and effective mechanism for implementing resource reservation in a real-time system is to reserve each task τ_i a specified amount of CPU time Q_i in every interval P_i. Such a general approach can also be applied to other resources different than the CPU, but in this context we will mainly focus on the CPU, because CPU scheduling is the topic of this book.

Some authors [RJMO98] tend to distinguish between *hard* and *soft* reservations. According to such a taxonomy, a hard reservation allows the reserved task to execute *at most* for Q_i units of time every P_i, whereas a soft reservation guarantees that the task executes *at least* for Q_i time units every P_i, allowing it to execute more if there is some idle time available.

A resource reservation technique for fixed priority scheduling was first presented in [MST94a]. According to this method, a task τ_i is first assigned a pair (Q_i, P_i) (denoted as a CPU *capacity reserve*) and then it is enabled to execute as a real-time task for Q_i units of time every P_i. When the task consumes its reserved quantum Q_i, it is blocked until the next period, if the reservation is hard, or it is scheduled in the background as a non-real-time task, if the reservation is soft. If the task is not finished, it is assigned another time quantum Q_i at the beginning of the next period and it is scheduled as a real-time task until the budget expires, and so on.

In this way, a task is *reshaped* so that it behaves like a periodic real-time task with known parameters (Q_i, P_i) and can be properly scheduled by a classical real-time scheduler.

Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent from a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

A simple kernel mechanism to enforce temporal protection under EDF scheduling is the constant bandwidth server (CBS) [AB98, AB04], described in Chap. 6. To properly implement temporal protection, however, each task τ_i with variable computation time should be handled by a dedicated CBS with bandwidth U_{s_i}, so

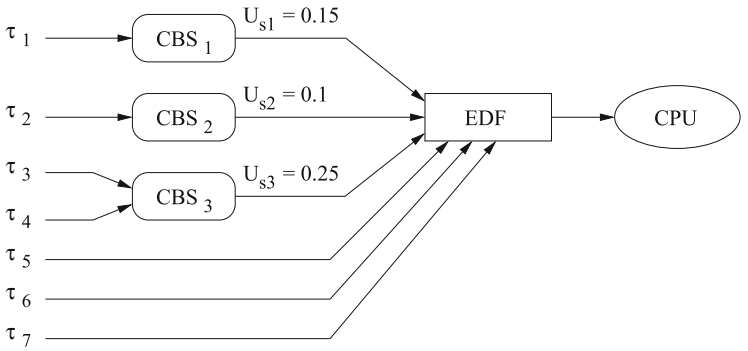
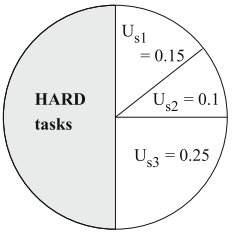


Fig. 9.16 Achieving temporal protection using the CBS mechanism

Fig. 9.17 Bandwidth allocation for a set of task



that it cannot interfere with the rest of the tasks for more than U_{s_i}. Figure 9.16 illustrates an example in which two tasks (τ_1 and τ_2) are served by two dedicated CBSs with bandwidth $U_{s_1} = 0.15$ and $U_{s_2} = 0.1$, a group of two tasks (τ_3, τ_4) is handled by a single CBS with bandwidth $U_{s_3} = 0.25$, and three hard periodic tasks (τ_5, τ_6, τ_7) are directly scheduled by EDF, without server intercession, since their execution times are not subject to large variations. In this example the total processor bandwidth is shared among the tasks as shown in Fig. 9.17.

The properties of the CBS guarantee that the set of hard periodic tasks (with utilization U_p) is schedulable by EDF if and only if

$$U_p + U_{s_1} + U_{s_2} + U_{s_3} \leq 1. \tag{9.12}$$

Note that, if condition (9.12) holds, the set of hard periodic tasks is always guaranteed to use 50% of the processor, independently of the execution times of the other tasks. Also observe that τ_3 and τ_4 are not isolated with respect to each other (i.e., one can steal processor time from the other), but they cannot interfere with the other tasks for more than 1/4 of the total processor bandwidth.

The CBS version presented in this book is meant for handling soft reservations. In fact, when the budget is exhausted, it is always replenished at its full value and the server deadline is postponed (i.e., the server is always active). As a consequence, a served task can execute more than Q_s in each period P_s, if there are no other tasks in

the system. However, the CBS can be easily modified to enforce hard reservations, just by postponing the budget replenishment to the server deadline.

9.3.2 Schedulability Analysis

Although a reservation is typically implemented using a server characterized by a budget Q_k and a period T_k, there are cases in which temporal isolation can be achieved by executing tasks in a static partition of disjoint time slots.

To characterize a bandwidth reservation independently on the specific implementation, Mok et al. [MFC01] introduced the concept of *bounded delay partition*, which describes a reservation by two parameters: a bandwidth α_k and a delay Δ_k. The bandwidth α_k measures the fraction of resource that is assigned to the served tasks, whereas the delay Δ_k represents the longest interval of time in which the resource is not available. In general, the minimum service provided by a resource can be precisely described by its *supply function* [LB03, SL03], representing the minimum amount of time the resource can provide in a given interval of time.

Definition 9.4 Given a reservation, the *supply function* $Z_k(t)$ is the minimum amount of time provided by the reservation in every time interval of length $t \geq 0$.

The supply function can be defined for many kinds of reservations, as static time partitions [MFC01, FM02], periodic servers [LB03, SL03], or periodic servers with arbitrary deadline [EAL07]. Consider, for example, that processing time is provided only in the intervals illustrated in Fig. 9.18, with a period of 12 units. In this case, the minimum service occurs when the resource is requested at the beginning of the longest idle interval; hence, the supply function is the one depicted in Fig. 9.19.

For this example we have $\alpha_k = 0.5$ and $\Delta_k = 3$. Once the bandwidth and the delay are computed, the supply function of a resource reservation can be lower bounded by the following *supply bound function*:

$$\mathsf{sbf}_k(t) \stackrel{\text{def}}{=} \max\{0,\ \alpha_k(t - \Delta_k)\}. \tag{9.13}$$

represented by the dashed line in Fig. 9.19. The advantage of using such a lower bound instead of the exact $Z_k(t)$ is that a reservation can be expressed with just two parameters.

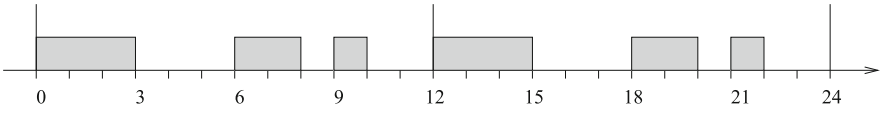


Fig. 9.18 A reservation implemented by a static partition of intervals

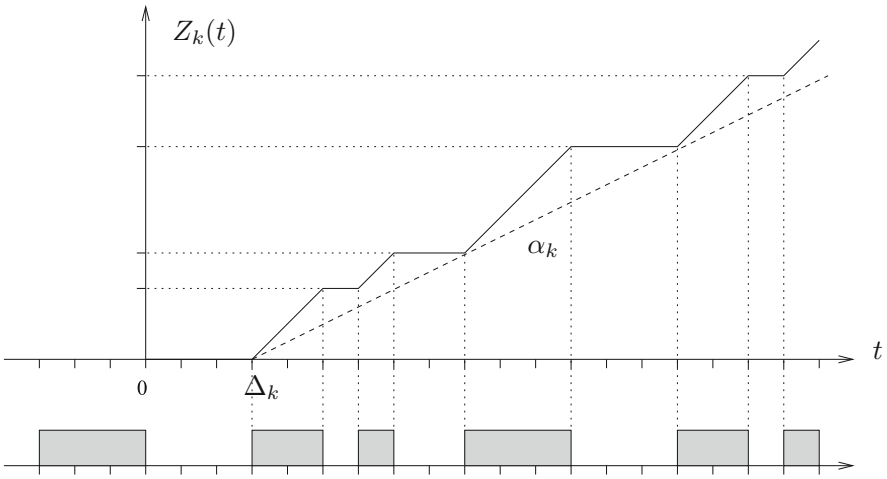


Fig. 9.19 A reservation implemented by a static partition of intervals

In general, for a given supply function $Z_k(t)$, the bandwidth α_k and the delay Δ_k can be formally defined as follows:

$$\alpha_k = \lim_{t \to \infty} \frac{Z_k(t)}{t} \tag{9.14}$$

$$\Delta_k = \sup_{t \geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}. \tag{9.15}$$

If a reservation is implemented using a periodic server with unspecified priority that allocates a budget Q_k every period T_k, then the supply function is the one illustrated in Fig. 9.20, where

$$\alpha_k = Q_k / T_k \tag{9.16}$$

$$\Delta_k = 2(T_k - Q_k). \tag{9.17}$$

It is worth observing that reservations with smaller delays are able to serve tasks with shorter deadlines, providing better responsiveness. However, small delays can only be achieved with servers with a small period, a condition for which the context switch overhead cannot be neglected. If σ is the runtime overhead due to a context switch (subtracted from the budget every period), then the effective bandwidth of reservation is

$$\alpha_k^{\text{eff}} = \frac{Q - \sigma}{T_k} = \alpha_k \left(1 - \frac{\sigma}{Q_k} \right).$$

Expressing Q_k and T_k as a function of α_k and Δ_k, we have

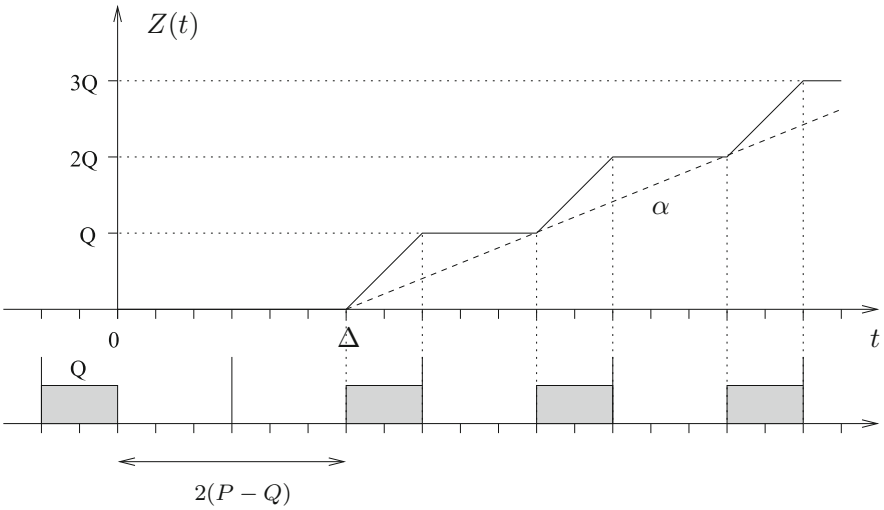


Fig. 9.20 A reservation implemented by a static partition of intervals

$$Q_k = \frac{\alpha_k \Delta_k}{2(1-\alpha_k)}$$
$$P_k = \frac{\Delta_k}{2(1-\alpha_k)}.$$

Hence,

$$\alpha_k^{\text{eff}} = \alpha_k + \frac{2\sigma(1-\alpha_k)}{\Delta_k}. \tag{9.18}$$

Within a reservation, the schedulability analysis of a task set under fixed priorities can be performed by extending Theorem 4.4 as follows [BBL09]:

Theorem 9.3 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by a fixed priority algorithm, under a reservation characterized by a supply function $Z_k(t)$, if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in \mathcal{TS}_i : W_i(t) \leq Z_k(t). \tag{9.19}$$

Similarly, the schedulability analysis of a task set under EDF can be performed by extending Theorem 4.6 as follows [BBL09]:

Theorem 9.4 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF, under a reservation characterized by*

a supply function $Z_k(t)$, if and only if $U < \alpha_k$ *and*

$$\forall t > 0 \quad \mathsf{dbf}(t) \leq Z_k(t). \tag{9.20}$$

In the specific case in which $Z_k(t)$ is lower bounded by the supply bound function, the test becomes only sufficient and the set of testing points can be better restricted as stated in the following theorem [BFB09]:

Theorem 9.5 *A set of preemptive periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF, under a reservation characterized by a supply function* $Z_k(t) = \max[0, \alpha_k(t - \Delta_k)]$, *if* $U < \alpha_k$ *and*

$$\forall t \in \mathcal{D} \quad \mathsf{dbf}(t) \leq \max[0, \alpha_k(t - \Delta_k)] \tag{9.21}$$

where

$$\mathcal{D} = \{d_k \mid d_k \leq \min[H, \max(D_{max}, L^*)]\}$$

and

$$L^* = \frac{\alpha_k \Delta_k + \sum_{i=1}^{n} (T_i - D_i) U_i}{\alpha_k - U}.$$

9.3.3 Handling Wrong Reservations

As already mentioned, under resource reservations, the system performance heavily depends on a correct bandwidth allocation to the various activities. In fact, if the bandwidth is under-allocated, the activities within that reservation will progress more slowly than expected, whereas on over-allocated bandwidth the available resources may be wasted. This problem can be solved by proper capacity sharing mechanisms that can transfer unused budgets to the reservations that need more bandwidth.

Capacity sharing algorithms have been developed both under fixed priority servers [BB02, BBB04] and dynamic priority servers [CBS00]. For example, the CASH algorithm [CBT05] extends CBS to include a slack reclamation. When a server becomes idle with residual budget, the slack is inserted in a queue of spare budgets (CASH queue) ordered by server deadlines. Whenever a new server is scheduled for execution, it first uses any CASH budget whose deadline is less than or equal to its own.

The bandwidth inheritance (BWI) algorithm [LLA01] applies the idea of priority inheritance to CPU resources in CBS, allowing a blocking low-priority process to steal resources from a blocked higher-priority process. IRIS [MLBC04] enhances CBS with fairer slack reclaiming, so slack is not reclaimed until all current jobs have

been serviced and the processor is idle. BACKSLASH [LB05] is another algorithm that enhances the efficiency of the reclaiming mechanism under EDF.

Wrong reservations can also be handled through feedback scheduling. If the operating system allows monitoring the actual execution time $e_{i,k}$ of each task instance, the actual maximum computation time of a task τ_i can be estimated (in a moving window) as

$$\hat{C}_i = \max_k \{e_{i,k}\}$$

and the actual requested bandwidth as $\hat{U}_i = \hat{C}_i / T_i$. Hence, $\hat{U}_i$ can be used as a reference value in a feedback loop to adapt the reservation bandwidth allocated to the task according to the actual needs. If more reservations are adapted online, we must ensure that the overall allocated bandwidth does not exceed the processor utilization; hence, a form of global feedback adaptation is required to prevent an overload condition. Similar approaches to achieve adaptive reservations have been proposed in [AB01] and in [PALW02].

9.3.4 Resource Sharing

When critical sections are used by tasks handled within a reservation server, an additional problem occurs when the server budget is exhausted inside a region. In this case, the served task cannot continue the execution to prevent other tasks to miss their deadlines; thus, an extra delay is added to the blocked tasks to wait until the next budget replenishment. Figure 9.21 illustrates a situation in which a high-priority task τ_1 shares a resource with another task τ_2 handled by a reservation server (e.g., a sporadic server) with budget $Q_k = 4$ and period $T_k = 10$. At time $t = 3$, τ_1 preempts τ_1 within its critical section, and at time $t = 4$ it blocks on the locked resource. When τ_2 resumes, however, the residual budget is not sufficient to finish the critical section, and τ_2 must be suspended until the budget will be replenished at time $t = 10$, introducing an extra delay [5,10] in the execution of τ_1. Two solutions have been proposed in the literature to prevent such an extra delay.

9.3.4.1 Solution 1: Budget Check

When a task wants to enter a critical section, the current server budget is checked before granting the access to the resource; if the budget is sufficient, the task enters the critical section; otherwise, the access to the resource is postponed until the next budget replenishment.

This mechanism is used in the SIRAP protocol, proposed by Behnam et al. [BSNN07, NSBS09] to share resources among reservations in fixed priority systems. An example of such a strategy is illustrated in Fig. 9.22. In the example,

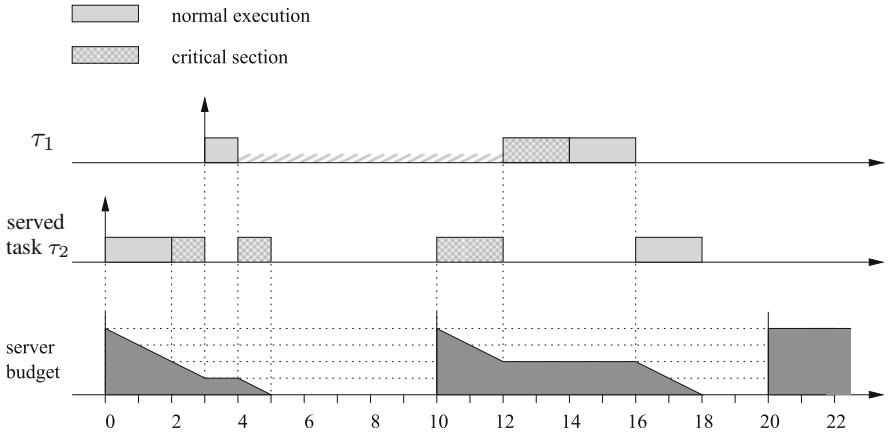


Fig. 9.21 Example of extra blocking introduced when the budget is exhausted inside a critical section

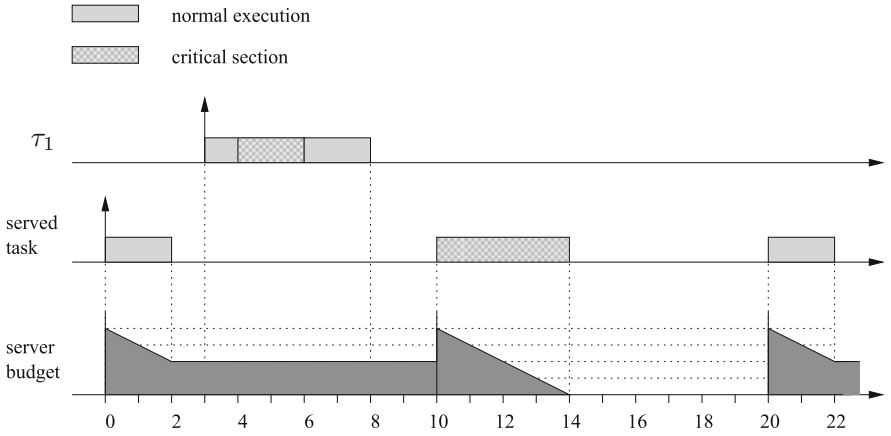


Fig. 9.22 Example of budget check to allow resource sharing within reservations

since at time $t = 2$ the budget is $Q_k = 2$ and the critical section is 4 units long, the resource is not granted and τ_2 is suspended until time $t = 10$, when the budget is recharged. In this case, τ_1 is able to execute immediately, while τ_2 experiences a longer delay with respect to the absence of protocol.

Under reservation servers scheduled by EDF, another more efficient protocol for accessing shared resources is BROE, proposed by Bertogna, Fisher, and Baruah [BFB09]. According to this method, a budget check is performed before entering a critical section: If the server budget is enough to complete the critical section, the access is granted; otherwise, the budget is recharged at its full value and the server deadline is proportionally postponed. This protocol has then be used to derive a

schedulability analysis of hierarchical real-time systems under shared resources [BBB16].

9.3.4.2 Solution 2: Budget Overrun

The second approach consists in entering a critical section without performing any budget check. When the budget is exhausted inside a resource, the server is allowed to consume some extra budget until the end of the critical section. In this case, the maximum extra budget must be estimated offline and taken into account in the schedulability analysis. An example of such a strategy is illustrated in Fig. 9.23. In this example, at time $t = 5$, when the budget is exhausted inside the resource, τ_2 is allowed to continue the execution until the end of the critical section, consuming 2 extra units of budget. In the worst case, the extra budget to be taken into account is equal to the longest critical section of the served task.

This approach was first proposed by Abeni and Buttazzo under EDF, using a constant bandwidth server (CBS) [AB04]. Then, it was analyzed under fixed priority systems by Davis and Burns [DB06] and later extended under EDF by Behnam et al. [BSNN08, BNSS10]. Davis and Burns proposed two versions of this mechanism:

1. *Overrun with payback*, where the server pays back in the next execution instant, in that the next budget replenishment is decreased by the overrun value
2. *Overrun without payback*, where no further action is taken after the overrun

Notice that the first solution (budget check) does not affect the execution of tasks in other reservations, but penalizes the response time of the served task. On the contrary, the second solution (budget overrun) does not increase the response time of the served task at the cost of a greater bandwidth requirement for the reservation.

Fig. 9.23 Example of budget overrun to allow resource sharing within reservations

9.4 Handling Permanent Overloads

This section presents some methodologies for handling permanent overload conditions occurring in periodic task systems when the total processor utilization exceeds one. Basically, there are three methods to reduce the load:

- **Job skipping.** This method reduces the total load by properly skipping (i.e., aborting) some job execution in the periodic tasks, in such a way that a minimum number of jobs per task is guaranteed to execute within their timing constraints.
- **Period adaptation.** According to this approach, the load is reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold.
- **Service adaptation.** According to this method, the load is reduced by decreasing the computational requirements of the tasks, trading predictability with quality of service.

9.4.1 Job Skipping

The computational load of a set of periodic tasks can be reduced by properly *skipping* a few jobs in the task set, in such a way that the remaining jobs can be scheduled within their deadlines. This approach is suitable for real-time applications characterized by soft or firm deadlines, such as those typically found in multimedia systems, where skipping a video frame once in a while is better than processing it with a long delay. Even in certain control applications, the sporadic skip of some jobs can be tolerated when the controlled systems is characterized by a high inertia.

To understand how job skipping can make an overloaded system schedulable, consider the following example, consisting of two tasks, with computation times $C_1 = 2$ and $C_2 = 8$ and periods $T_1 = 4$ and $T_2 = 12$. Since the processor utilization factor is $U_p = 14/12 > 1$, the system is under a permanent overload, and the tasks cannot be scheduled within their deadlines. Nevertheless, Fig. 9.24 shows that skipping a job every three in task τ_1 the overload can be resolved and all the remaining jobs can be scheduled within their deadlines.

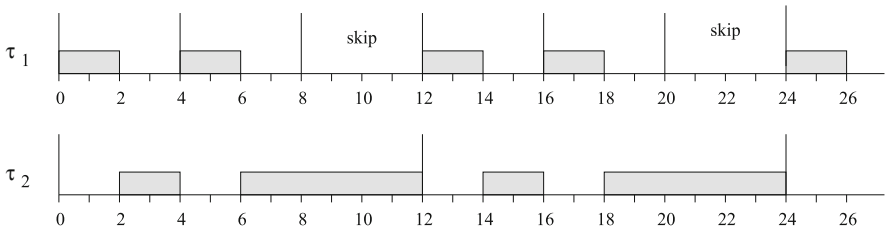


Fig. 9.24 The overload condition resolved by skipping one job every three in task τ_1

In order to control the overall system load, it is important to derive the relation between the number of skips (i.e., the number of aborted jobs per task) and the total computational demand. In 1995, Koren and Shasha [KS95] proposed a new task model (known as the *firm* periodic model) suited to be handled by this technique.

According to this model, each periodic task τ_i is characterized by the following parameters:

$$\tau_i(C_i, T_i, D_i, S_i)$$

where C_i is the worst-case computation time, T_i its period, D_i its relative deadline (assumed to be equal to the period), and S_i a skip parameter, $2 \leq S_i \leq \infty$, expressing the minimum distance between two consecutive skips. For example, if $S_i = 5$, the task can skip one instance every five. When $S_i = \infty$, no skips are allowed and τ_i is equivalent to a hard periodic task. The skip parameter can be viewed as a *quality of service* (QoS) metric (the higher S, the better the quality of service).

Using the terminology introduced by Koren and Shasha in [KS95], every job of a periodic task can be *red* or *blue*: A red job must be completed within its deadline, whereas a blue job can be aborted at any time. To meet the constraint imposed by the skip parameter S_i, each scheduling algorithm must have the following characteristics:

- If a blue job is skipped, then the next $S_i - 1$ jobs must be red.
- If a blue job completes successfully, the next job is also blue.

The authors showed that making optimal use of skips is NP-hard and presented two algorithms (one working under rate monotonic and one under EDF) that exploit skips to schedule slightly overloaded systems. In general, these algorithms are not optimal, but they become optimal under a particular condition, called the *deeply red* condition.

Definition 9.5 A system is *deeply red* if all tasks are synchronously activated and the first $S_i - 1$ instances of every task τ_i are red.

Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply red. For this reason, all the results shown in this section are proved under this condition. This means that, if a task set is schedulable under the deeply red condition, it is also schedulable in any other situation.

9.4.1.1 Schedulability Analysis

The feasibility analysis of a set of firm tasks can be performed through the processor demand criterion [BRH90] illustrated in Chap. 4, under the deeply red condition, and assuming that, in the worst case, all blue jobs are aborted. In this worst-case scenario, the processor demand of τ_i due to the red jobs in an interval $[0, L]$ can be obtained as the difference between the demand of all the jobs and the demand of the

blue jobs:

$$g_i^{skip}(0, L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i . \tag{9.22}$$

Hence, the feasibility of the task set can be verified through the following test:

Sufficient Condition

A set of firm periodic tasks is schedulable by EDF if

$$\forall L \geq 0 \qquad \sum_{i=1}^{n} \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i \leq L \tag{9.23}$$

A necessary condition can be easily derived by observing that a schedule is certainly infeasible when the utilization factor due to the red jobs is greater than one. That is,

Necessary Condition

Necessary condition for the schedulability of a set of firm periodic tasks is that

$$\sum_{i=1}^{n} \frac{C_i (S_i - 1)}{T_i S_i} \leq 1 \tag{9.24}$$

9.4.1.2 Example

To better clarify the concepts mentioned above, consider the task set shown in Fig. 9.25 and the corresponding feasible schedule, obtained by EDF. Notice that the processor utilization factor is greater than 1 ($U_p = 1.25$), but both conditions (9.23) and (9.24) are satisfied.

9.4.1.3 Skips and Bandwidth Saving

If skips are permitted in the periodic task set, the spare time saved by rejecting the blue instances can be reallocated for other purposes, for example, for scheduling slightly overloaded systems or for advancing the execution of soft aperiodic requests.

Task	Task1	Task2	Task3
Computation	1	2	5
Period	3	4	12
Skip Parameter	4	3	∞
U_p	1.25		

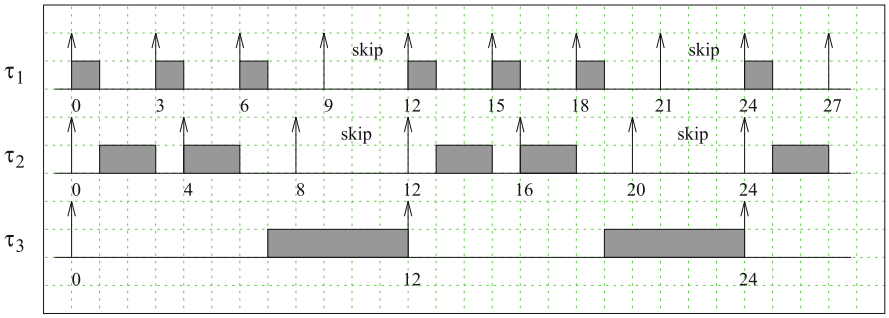


Fig. 9.25 A schedulable set of firm periodic tasks

Unfortunately, the spare time has a "granular" distribution and cannot be reclaimed at any time. Nevertheless, it can be shown that skipping blue instances still produces a bandwidth saving in the periodic schedule. Caccamo and Buttazzo [CB97] identified the amount of bandwidth saved by skips using a simple parameter, the *equivalent utilization factor* U_p^{skip}, which can be defined as follows:

$$U_p^{skip} = \max_{L \geq 0} \left\{ \frac{\sum_i g_i^{skip}(0, L)}{L} \right\} \tag{9.25}$$

where $g_i^{skip}(0, L)$ is given in Eq. (9.22).

Using this definition, the schedulability of a deeply red skippable task set can be also verified using the following theorem [CB97]:

Theorem 9.6 *A set* Γ *of deeply red skippable periodic tasks is schedulable by EDF if*

$$U_p^{skip} \leq 1.$$

Notice that the U_p^{skip} factor represents the net bandwidth really used by periodic tasks, under the deeply red condition. It is easy to show that $U_p^{skip} \leq U_p$. In fact, according to Eq. (9.25) (setting $S_i = \infty$), U_p can also be defined as

$$U_p = \max_{L \geq 0} \left\{ \frac{\sum_i \left\lfloor \frac{L}{T_i} \right\rfloor C_i}{L} \right\}$$

Thus, $U_p^{skip} \leq U_p$ because

$$\left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) \leq \left\lfloor \frac{L}{T_i} \right\rfloor.$$

The bandwidth saved by skips can easily be exploited by an aperiodic server to advance the execution of aperiodic tasks. The following theorem [CB97] provides a sufficient condition for guaranteeing a hybrid (periodic and aperiodic) task set.

Theorem 9.7 *Given a set of periodic tasks that allow skip with equivalent utilization U_p^{skip} and a set of soft aperiodic tasks handled by a server with utilization factor U_s, the hybrid set is schedulable by EDF if*

$$U_p^{skip} + U_s \leq 1. \tag{9.26}$$

The fact that condition of Theorem 9.7 is not necessary is a direct consequence of the "granular" distribution of the spare time produced by skips. In fact, a fraction of this spare time is uniformly distributed along the schedule and can be used as an additional free bandwidth ($U_p - U_p^{skip}$) available for aperiodic service. The remaining portion is discontinuous and creates a kind of "holes" in the schedule, which cannot be used any time, unfortunately. Whenever an aperiodic request falls into some hole, it can exploit a bandwidth greater than $1 - U_p^{skip}$. Indeed, it is easy to find examples of feasible task sets with a server bandwidth $U_s > 1 - U_p^{skip}$. The following theorem [CB97] gives a maximum bandwidth U_s^{max} above which the schedule is certainly not feasible.

Theorem 9.8 *Given a set Γ of n periodic tasks that allow skips and an aperiodic server with bandwidth U_s, a necessary condition for the feasibility of Γ is*

$$U_s \leq U_s^{max}$$

where

$$U_s^{max} = 1 - U_p + \sum_{i=1}^{n} \frac{C_i}{T_i S_i}. \tag{9.27}$$

9.4.1.4 Example

Consider the periodic task set shown in Table 9.1. The equivalent utilization factor of the periodic task set is $U_p^{skip} = 4/5$, while $U_s^{max} = 0.27$, leaving a bandwidth of $U_s = 1 - U_p^{skip} = 1/5$ for the aperiodic tasks. Three aperiodic jobs J_1, J_2, and J_3

Table 9.1 A schedulable task set

Task	Task1	Task2
Computation	2	2
Period	3	5
Skip parameter	2	∞
U_p	1.07	
U_p^{skip}	0.8	
$1 - U_p^{skip}$	0.2	
U_s^{max}	0.27	

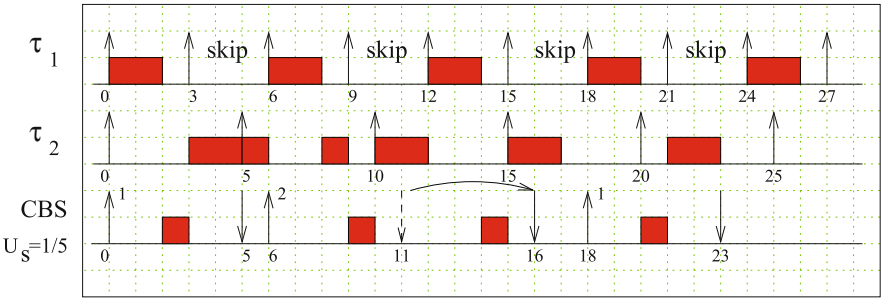


Fig. 9.26 Schedule produced by EDF+CBS for the task set shown in Table 9.1

are released at times $t_1 = 0$, $t_2 = 6$, and $t_3 = 18$; moreover, they have computation times $C_1^{ape} = 1$, $C_2^{ape} = 2$, and $C_3^{ape} = 1$, respectively.

Supposing aperiodic activities are scheduled by a CBS server with budget $Q^s = 1$ and period $T^s = 5$, Fig. 9.26 shows the resulting schedule under EDF+CBS. Notice that J_2 has a deadline postponement (according to CBS rules) at time $t = 10$ with new server deadline $d_{new}^s = d_{old}^s + T^s = 11 + 5 = 16$. According to the sufficient schedulability test provided by Theorem 9.7, the task set is schedulable when the CBS is assigned a bandwidth $U_s = 1 - U_p^{skip}$. However, this task set is also schedulable with a bandwidth $U_s = 0.25$, greater than $1 - U_p^{skip}$ but less than U_s^{max}, although this is not true in general.

9.4.2 Period Adaptation

There are several real-time applications in which timing constraints are not rigid, but depend on the system state. For example, in a flight control system, the sampling rate of the altimeters is a function of the current altitude of the aircraft: The lower the altitude, the higher the sampling frequency. A similar need arises in mobile robots operating in unknown environments, where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, to

maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle.

The possibility of varying tasks' rates also increases the flexibility of the system in handling overload conditions, providing a more general admission control mechanism. For example, whenever a new task cannot be guaranteed, instead of rejecting the task, the system can reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

In the real-time literature, several approaches exist for dealing with an overload through a period adaptation. For example, Kuo and Mok [KAK91] propose a load scaling technique to gracefully degrade the workload of a system by adjusting the periods of processes. In this work, tasks are assumed to be equally important and the objective is to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. Nakajima and Tezuka [NT94] show how a real-time system can be used to support an adaptive application: Whenever a deadline miss is detected, the period of the failed task is increased. Seto et al. [SLSS96] change tasks' periods within a specified range to minimize a performance index defined over the task set. This approach is effective at a design stage to optimize the performance of a discrete control system, but cannot be used for online load adjustment. Lee, Rajkumar, and Mercer [LRM96] propose a number of policies to dynamically adjust the tasks' rates in overload conditions. Abdelzaher, Atkins, and Shin [AAS97] present a model for QoS negotiation to meet both predictability and graceful degradation requirements during overloads. In this model, the QoS is specified as a set of negotiation options, in terms of rewards and rejection penalties. Nakajima [Nak98] shows how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational demand. However, it is not clear how the QoS can be increased when the system is underloaded. Beccari et al. [BCRZ99] propose several policies for handling overload through period adjustment. The authors, however, do not address the problem of increasing the task rates when the processor is not fully utilized.

Although these approaches may lead to interesting results in specific applications, a more general framework can be used to avoid a proliferation of policies and treat different applications in a uniform fashion.

The elastic model presented in this section (originally introduced in [BAL98] and later extended in [BLCA02]) provides a novel theoretical framework for flexible workload management in real-time applications.

9.4.2.1 Examples

To better understand the idea behind the elastic model, consider a set of three periodic tasks, with computation times $C_1 = 10$, $C_2 = 10$, and $C_3 = 15$ and periods $T_1 = 20$, $T_2 = 40$, and $T_3 = 70$. Clearly, the task set is schedulable by EDF because

Table 9.2 Period ranges for the task set considered in the example

	$T_{i_{min}}$	$T_{i_{max}}$
τ_1	20	25
τ_2	40	50
τ_3	35	80

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.964 < 1.$$

To allow a certain degree of flexibility, suppose that tasks are allowed to run with periods ranging within two values, reported in Table 9.2.

Now, suppose that a new task τ_4, with computation time $C_4 = 5$ and period $T_4 = 30$, enters the system at time t. The total processor utilization of the new task set is

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} + \frac{5}{30} = 1.131 > 1.$$

In a rigid scheduling framework, τ_4 should be rejected to preserve the timing behavior of the previously guaranteed tasks. However, τ_4 can be accepted if the periods of the other tasks can be increased in such a way that the total utilization is less than one. For example, if T_1 can be increased up to 23, the total utilization becomes $U_p = 0.989$, and hence, τ_4 can be accepted.

As another example, if tasks are allowed to change their frequency and τ_3 reduces its period to 50, no feasible schedule exists, since the utilization would be greater than 1:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{50} = 1.05 > 1.$$

Notice that a feasible schedule exists for $T_1 = 22$, $T_2 = 45$, and $T_3 = 50$. Hence, the system can accept the higher request rate of τ_3 by slightly decreasing the rates of τ_1 and τ_2. Task τ_3 can even run with a period $T_3 = 40$, since a feasible schedule exists with periods T_1 and T_2 within their range. In fact, when $T_1 = 24$, $T_2 = 50$, and $T_3 = 40$, $U_p = 0.992$. Finally, notice that if τ_3 requires to run at its minimum period ($T_3 = 35$), there is no feasible schedule with periods T_1 and T_2 within their range, hence the request of τ_3 to execute with a period $T_3 = 35$ must be rejected.

Clearly, for a given value of T_3, there can be many different period configurations which lead to a feasible schedule; thus, one of the possible feasible configurations must be selected. The elastic approach provides an efficient way for quickly selecting a feasible period configuration among all the possible solutions.

9.4.2.2 The Elastic Model

The basic idea behind the elastic model is to consider each task as flexible as a spring with a given rigidity coefficient and length constraints. In particular, the utilization of a task is treated as an elastic parameter, whose value can be modified by changing the period within a specified range.

Each task is characterized by four parameters: a computation time C_i, a nominal period T_{i_0} (considered as the minimum period), a maximum period $T_{i_{max}}$, and an elastic coefficient $E_i \geq 0$, which specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration. The greater E_i, the more elastic the task. Thus, an elastic task is denoted as

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i).$$

In the following, T_i will denote the actual period of task τ_i, which is constrained to be in the range $[T_{i_0}, T_{i_{max}}]$. Any task can vary its period according to its needs within the specified range. Any variation, however, is subject to an *elastic* guarantee and is accepted only if there exists a feasible schedule in which all the other periods are within their range.

Under the elastic model, given a set of n periodic tasks with utilization $U_p > U_{max}$, the objective of the guarantee is to compress tasks' utilization factors to achieve a new desired utilization $U_d \leq U_{max}$ such that all the periods are within their ranges.

The following definitions are also used in this section:

$$U_{i_{min}} = C_i / T_{i_{max}}$$

$$U_{min} = \sum_{i=1}^{n} U_{i_{min}}$$

$$U_{i_0} = C_i / T_{i_0}$$

$$U_0 = \sum_{i=1}^{n} U_{i_0}$$

Clearly, a solution can always be found if $U_{min} \leq U_d$, hence, this condition has to be verified a priori.

It is worth noting that the elastic model is more general than the classical Liu and Layland's task model, so it does not prevent a user from defining hard real-time tasks. In fact, a task having $T_{i_{max}} = T_{i_0}$ is equivalent to a hard real-time task with fixed period, independently of its elastic coefficient. A task with $E_i = 0$ can arbitrarily vary its period within its specified range, but it cannot be varied by the system during load reconfigurations.

To understand how an elastic guarantee is performed in this model, it is convenient to compare an elastic task τ_i with a linear spring S_i characterized by

a rigidity coefficient k_i, a nominal length x_{i_0}, and a minimum length $x_{i_{min}}$. In the following, x_i will denote the actual length of spring S_i, which is constrained to be greater than or equal to $x_{i_{min}}$.

In this comparison, the length x_i of the spring is equivalent to the task's utilization factor $U_i = C_i/T_i$, and the rigidity coefficient k_i is equivalent to the inverse of the task's elasticity ($k_i = 1/E_i$). Hence, a set of n periodic tasks with total utilization factor $U_p = \sum_{i=1}^{n} U_i$ can be viewed as a sequence of n springs with total length $L = \sum_{i=1}^{n} x_i$.

In the linear spring system, this is equivalent to compressing the springs so that the new total length L_d is less than or equal to a given maximum length L_{max}. More formally, in the spring system the problem can be stated as follows.

> Given a set of n springs with known rigidity and length constraints, if the total length $L_0 = \sum_{i=1}^{n} x_{i_0} > L_{max}$, find a set of new lengths x_i such that $x_i \geq x_{i_{min}}$ and $\sum_{i=1}^{n} x_i = L_d$, where L_d is any arbitrary desired length such that $L_d < L_{max}$.

For the sake of clarity, we first solve the problem for a spring system without length constraints (i.e., $x_{i_{min}} = 0$), then we show how the solution can be modified by introducing length constraints, and finally we show how the solution can be adapted to the case of a task set.

9.4.2.3 Springs with No Length Constraints

Consider a set Γ of n springs with nominal length x_{i_0} and rigidity coefficient k_i positioned one after the other, as depicted in Fig. 9.27. Let L_0 be the total length of the array, that is, the sum of the nominal lengths: $L_0 = \sum_{i=1}^{n} x_{i_0}$. If the array is compressed so that its total length is equal to a desired length L_d ($0 < L_d < L_0$), the first problem we want to solve is to find the new length x_i of each spring, assuming that for all i, $0 < x_i < x_{i_0}$ (i.e., $x_{i_{min}} = 0$).

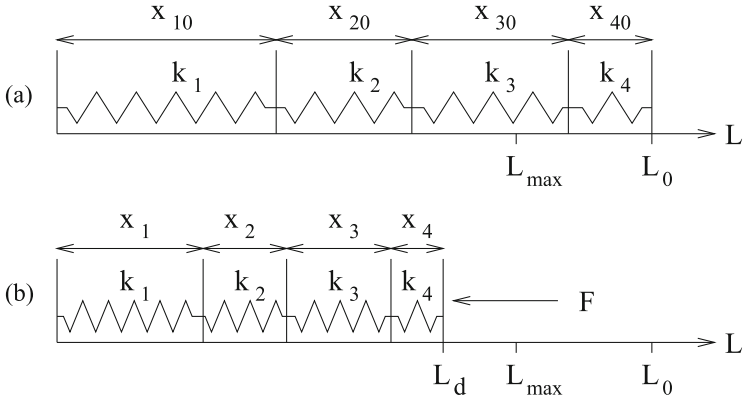


Fig. 9.27 A linear spring system: (**a**) the total length is L_0 when springs are uncompressed; (**b**) the total length is $L_d < L_0$ when springs are compressed by a force F

Being L_d the total length of the compressed array of springs, we have

$$L_d = \sum_{i=1}^{n} x_i. \tag{9.28}$$

If F is the force that keeps a spring in its compressed state, then, for the equilibrium of the system, it must be

$$\forall i \quad F = k_i(x_{i_0} - x_i),$$

from which we derive

$$\forall i \quad x_i = x_{i_0} - \frac{F}{k_i}. \tag{9.29}$$

By summing Eqs. (9.29) we have

$$L_d = L_0 - F \sum_{i=1}^{n} \frac{1}{k_i}.$$

Thus, force F can be expressed as

$$F = K_p(L_0 - L_d), \tag{9.30}$$

where

$$K_p = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}}. \tag{9.31}$$

Substituting expression (9.30) into Eqs. (9.29), we finally achieve

$$\forall i \quad x_i = x_{i_0} - (L_0 - L_d)\frac{K_p}{k_i}. \tag{9.32}$$

Equation (9.32) allows us to compute how each spring has to be compressed in order to have a desired total length L_d.

For a set of elastic tasks, Eq. (9.32) can be translated as follows:

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{E_i}{E_0}, \tag{9.33}$$

where $E_i = 1/k_i$ and $E_0 = \sum_{i=1}^{n} E_i$.

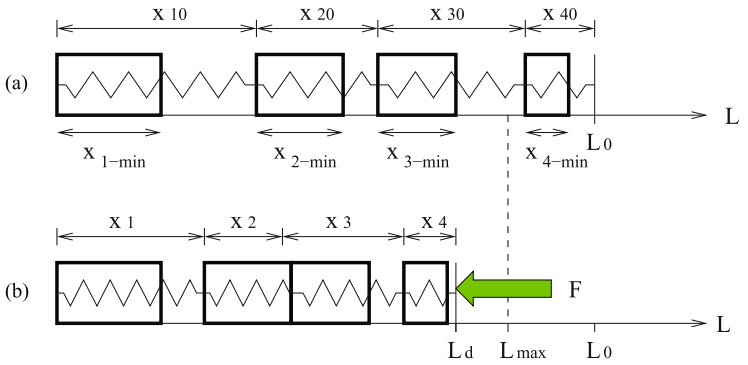


Fig. 9.28 Springs with minimum length constraints (**a**); during compression, spring S_2 reaches its minimum length and cannot be compressed any further (**b**)

9.4.2.4 Introducing Length Constraints

If each spring has a length constraint, in the sense that its length cannot be less than a minimum value $x_{i_{min}}$, then the problem of finding the values x_i requires an iterative solution. In fact, if during compression one or more springs reach their minimum length, the additional compression force will only deform the remaining springs. Such a situation is depicted in Fig. 9.28.

Thus, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed springs having minimum length and a set Γ_v of variable springs that can still be compressed. Applying Eqs. (9.32) to the set Γ_v of variable springs, we have

$$\forall S_i \in \Gamma_v \quad x_i = x_{i_0} - (L_{v_0} - L_d + L_f)\frac{K_v}{k_i} \tag{9.34}$$

where

$$L_{v_0} = \sum_{S_i \in \Gamma_v} x_{i_0} \tag{9.35}$$

$$L_f = \sum_{S_i \in \Gamma_f} x_{i_{min}} \tag{9.36}$$

$$K_v = \frac{1}{\sum_{S_i \in \Gamma_v} \frac{1}{k_i}}. \tag{9.37}$$

Whenever there exists some spring for which Eq. (9.34) gives $x_i < x_{i_{min}}$, the length of that spring has to be fixed at its minimum value, sets Γ_f and Γ_v must be updated, and Eqs. (9.34), (9.35), (9.36), and (9.37) must be recomputed for the new set Γ_v.

If there exists a feasible solution, that is, if $L_d \geq L_{min} = \sum_{i=1}^{n} x_{i_{min}}$, the iterative process ends when each value computed by Eqs. (9.34) is greater than or equal to its corresponding minimum $x_{i_{min}}$.

9.4.2.5 Compressing Tasks' Utilizations

When dealing with a set of elastic tasks, Eqs. (9.34), (9.35), (9.36), and (9.37) can be rewritten by substituting all length parameters with the corresponding utilization factors and the rigidity coefficients k_i and K_v with the corresponding elastic coefficients E_i and E_v. Similarly, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed tasks having minimum utilization and a set Γ_v of variable tasks that can still be compressed. Let $U_{i_0} = C_i/T_{i_0}$ be the nominal utilization of task τ_i, $U_0 = \sum_{i=1}^{n} U_{i_0}$ be the nominal utilization of the task set, U_{v_0} be the sum of the nominal utilizations of tasks in Γ_v, and U_f be the total utilization factor of tasks in Γ_f. Then, to achieve a desired utilization $U_d < U_0$, each task has to be compressed up to the following utilization:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v} \tag{9.38}$$

where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0} \tag{9.39}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{9.40}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{9.41}$$

If there exist tasks for which $U_i < U_{i_{min}}$, then the period of those tasks has to be fixed at its maximum value $T_{i_{max}}$ (so that $U_i = U_{i_{min}}$), sets Γ_f and Γ_v must be updated (hence, U_f and E_v recomputed), and Eq. (9.38) must be applied again to the tasks in Γ_v. If there exists a feasible solution, that is, if the desired utilization U_d is greater than or equal to the minimum possible utilization $U_{min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{max}}}$, the iterative process ends when each value computed by Eq. (9.38) is greater than or equal to its corresponding minimum $U_{i_{min}}$. The algorithm[2] for compressing a set Γ of n elastic tasks up to a desired utilization U_d is shown in Fig. 9.29. It has $O(n^2)$ complexity.

[2] The actual implementation of the algorithm contains more checks on tasks' variables, which are not shown here to simplify its description.

```
Algorithm: Elastic_compression(Γ, U_d)
Input: A task set Γ and a desired utilization U_d < 1
Output: A task set with modified periods such that U_p = U_d

begin

(1)  U_min := Σ_{i=1}^{n} C_i/T_{i_max};
(2)  if (U_d < U_min) return(INFEASIBLE);
(3)  for (i := 1 to n) U_{i_0} := C_i/T_{i_0};

(4)  do
(5)       U_f := 0; U_{v_0} := 0; E_v := 0;
(6)       for (i := 1 to n) do
(7)            if ((E_i == 0) or (T_i == T_{i_max})) then
(8)                 U_f := U_f + U_{i_min};
(9)            else
(10)                E_v := E_v + E_i;
(11)                U_{v_0} := U_{v_0} + U_{i_0};
(12)           end
(13)      end

(14)      ok := 1;
(15)      for (each τ_i ∈ Γ_v) do
(16)           if ((E_i > 0) and (T_i < T_{i_max})) then
(17)                U_i := U_{i_0} − (U_{v_0} − U_d + U_f)E_i/E_v;
(18)                T_i := C_i/U_i;
(19)                if (T_i > T_{i_max}) then
(20)                     T_i := T_{i_max};
(21)                     ok := 0;
(22)                end
(23)           end
(24)      end

(25) while (ok == 0);
(26) return(FEASIBLE);

end
```

Fig. 9.29 Algorithm for compressing a set of elastic tasks

9.4.2.6 Decompression

All tasks' utilizations that have been compressed to cope with an overload situation can return toward their nominal values when the overload is over. Let Γ_c be the subset of compressed tasks (that is, the set of tasks with $T_i > T_{i_0}$), let Γ_a be the set of remaining tasks in Γ (that is, the set of tasks with $T_i = T_{i_0}$), and let U_d be the current processor utilization of Γ. Whenever a task in Γ_a voluntarily increases

its period, all tasks in Γ_c can expand their utilizations according to their elastic coefficients, so that the processor utilization is kept at the value of U_d.

Now, let U_c be the total utilization of Γ_c, let U_a be the total utilization of Γ_a after some task has increased its period, and let U_{c_0} be the total utilization of tasks in Γ_c at their nominal periods. It can easily be seen that if $U_{c_0} + U_a \leq U_{lub}$, all tasks in Γ_c can return to their nominal periods. On the other hand, if $U_{c_0} + U_a > U_{lub}$, then the release operation of the tasks in Γ_c can be viewed as a compression, where $\Gamma_f = \Gamma_a$ and $\Gamma_v = \Gamma_c$. Hence, it can still be performed by using Eqs. (9.38), (9.40) and (9.41) and the algorithm presented in Fig. 9.29.

9.4.3 Implementation Issues

The elastic compression algorithm can be efficiently implemented on top of a real-time kernel as a routine (elastic manager) that is activated every time a new task is created, terminated, or there is a request for a period change. When activated, the elastic manager computes the new periods according to the compression algorithm and modify them atomically. Sudvarg, Chris, and Baruah [SGB21] presented an efficient implementation of the elastic algorithm in which initialization takes $O(n \log n)$ time and online admission control takes $O(n)$ time.

To avoid any deadline miss during the transition phase, it is crucial to ensure that all the periods are modified at opportune time instants, according to the following rule [BLCA02]:

> The period of a task τ_i can be increased at any time, but can only be reduced at the next job activation.

Figure 9.30 shows an example in which τ_1 misses its deadline when its period is reduced at time $t = 15$ (i.e., before its next activation time [$t = 20$]). Notice that the task set utilization is $U_p = 29/30$ before compression and $U'_p = 28/30$ after compression. This means that, although the system is schedulable by EDF in both steady-state conditions, some deadline can be missed if a period is reduced too early.

An earlier instant at which a period can be safely reduced without causing any deadline miss in the transition phase has been computed in [BLCA02] and later improved in [Gua09].

9.4.3.1 Period Rescaling

If the elastic coefficients are set equal to task nominal utilizations, elastic compression has the effect of a simple rescaling, where all the periods are increased by the same percentage. In order to work correctly, however, period rescaling must be

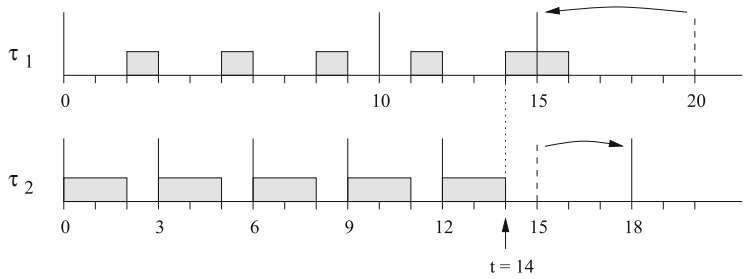


Fig. 9.30 A task can miss its deadline if a period is reduced at an arbitrary time instant

uniformly applied to all the tasks, without restrictions on the maximum period. This means having $U_f = 0$ and $U_{v_0} = U_0$. Under this assumption, by setting $E_i = U_{i_0}$, equations (9.38) become

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{U_{i_0}}{U_0} = \frac{U_{i_0}}{U_0}[U_0 - (U_0 - U_d)] = \frac{U_{i_0}}{U_0}U_d$$

from which we have that

$$T_i = T_{i_0}\frac{U_0}{U_d}. \tag{9.42}$$

This means that in overload situations ($U_0 > 1$) the compression algorithm causes all task periods to be increased by a common scale factor

$$\eta = \frac{U_0}{U_d}.$$

Notice that, after compression is performed, the total processor utilization becomes

$$U = \sum_{i=1}^{n} \frac{C_i}{\eta T_{i_0}} = \frac{1}{\eta}U_0 = \frac{U_d}{U_0}U_0 = U_d$$

as desired.

If a maximum period needs to be defined for some task, an online guarantee test can easily be performed before compression to check whether all the new periods are less than or equal to the maximum value. This can be done in $O(n)$ by testing whether

$$\forall i = 1, \ldots, n \quad \eta T_{i_0} \leq T_i^{max}.$$

By deciding to apply period rescaling, we lose the freedom of choosing the elastic coefficients, since they must be set equal to task nominal utilizations. However,

this technique has the advantage of leaving the task periods ordered as in the nominal configuration, which simplifies the compression algorithm in the presence of resource constraints and enables its usage in fixed priority systems, where priorities are typically assigned based on periods.

9.4.3.2 Concluding Remarks

The elastic model offers a flexible way to handle overload conditions. In fact, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request. As soon as a transient overload condition is over (because a task terminates or voluntarily increases its period), all the compressed tasks may expand up to their original utilization, eventually recovering their nominal periods.

The major advantage of the elastic method is that the policy for selecting a solution is implicitly encoded in the elastic coefficients provided by the user (e.g., based on task importance). Each task is varied based on its elastic status and a feasible configuration is found, if there exists one. This is very useful for supporting both multimedia systems and control applications, in which the execution rates of some computational activities have to be dynamically tuned as a function of the current system state. Furthermore, the elastic mechanism can easily be implemented on top of classical real-time kernels and can be used under fixed or dynamic priority scheduling algorithms.

It is worth observing that the elastic approach is not limited to task scheduling. Rather, it represents a general resource allocation methodology which can be applied whenever a resource has to be allocated to objects whose constraints allow a certain degree of flexibility. For example, in a distributed system, dynamic changes in node transmission rates over the network could be efficiently handled by assigning each channel an elastic bandwidth, which could be tuned based on the actual network traffic. An application of the elastic model to the network has been proposed in [PGBA02].

Another interesting application of the elastic approach is to automatically adapt task rates to the current load, without specifying worst-case execution times. If the system is able to monitor the actual execution time of each job, such data can be used to compute the actual processor utilization. If this is less than one, task rates can be increased according to elastic coefficients to fully utilize the processor. On the other hand, if the actual processor utilization is a little greater than one and some deadline misses are detected, task rates can be reduced to bring the processor utilization at a desired safe value.

The elastic model has also been extended in [BLCA02] to deal with resource constraints, thus allowing tasks to interact through shared memory buffers. In order to estimate maximum blocking times due to mutual exclusion and analyze task schedulability, critical sections are assumed to be accessed through the Stack Resource Policy [Bak91].

9.4.4 Service Adaptation

A third method for coping with a permanent overload condition is to reduce the load by decreasing the task computation times. This can be done only if the tasks have been originally designed to trade performance with computational requirements. When tasks use some incremental algorithm to produce approximated results, the precision of results is related to the number of iterations and thus with the computation time. In this case, an overload condition can be handled by reducing the quality of results, aborting the remaining computation if the quality of the current results is acceptable.

The concept of imprecise and approximate computation has emerged as a new approach to increasing flexibility in dynamic scheduling by trading computation accuracy with timing requirements. If processing time is not enough to produce high-quality results within the deadlines, there could be enough time for producing approximate results with a lower quality. This concept has been formalized by many authors [LNL87, LLN87, LLS+91, SLC91, LSL+94, Nat95] and specific techniques have been developed for designing programs that can produce partial results.

In a real-time system that supports imprecise computation, every task τ_i is decomposed into a *mandatory* subtask M_i and an *optional* subtask O_i. The mandatory subtask is the portion of the computation that must be done in order to produce a result of acceptable quality, whereas the optional subtask refines this result [SLCG89]. Both subtasks have the same arrival time a_i and the same deadline d_i as the original task τ_i; however, O_i becomes ready for execution when M_i is completed. If C_i is the worst-case computation time associated with the task, subtasks M_i and O_i have computation times m_i and o_i, such that $m_i + o_i = C_i$. In order to guarantee a minimum level of performance, M_i must be completed within its deadline, whereas O_i can be left incomplete, if necessary, at the expense of the quality of the result produced by the task.

It is worth noticing that the task model used in traditional real-time systems is a special case of the one adopted for imprecise computation. In fact, a hard task corresponds to a task with no optional part ($o_i = 0$), whereas a soft task is equivalent to a task with no mandatory part ($m_i = 0$).

In systems that support imprecise computation, the *error* ϵ_i in the result produced by τ_i (or simply the error of τ_i) is defined as the length of the portion of O_i discarded in the schedule. If σ_i is the total processor time assigned to O_i by the scheduler, the error of task τ_i is equal to

$$\epsilon_i = o_i - \sigma_i .$$

The *average error* $\overline{\epsilon}$ on the task set is defined as

$$\overline{\epsilon} = \sum_{i=1}^{n} w_i \epsilon_i ,$$

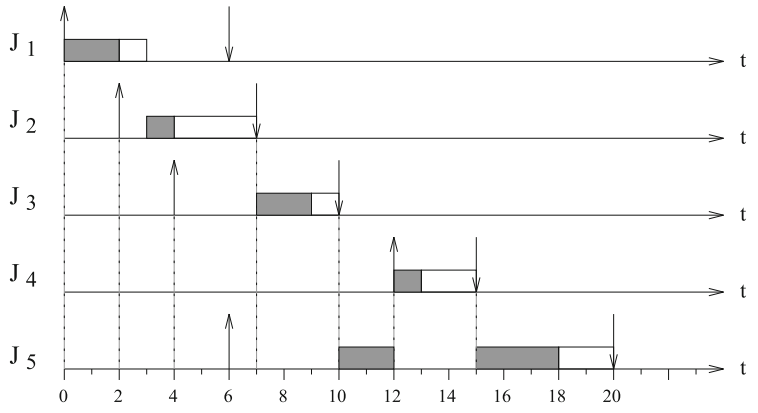
where w_i is the relative importance of τ_i in the task set. An error $\epsilon_i > 0$ means that a portion of subtask O_i has been discarded in the schedule at the expense of the quality of the result produced by task τ_i, but for the benefit of other mandatory subtasks that can complete within their deadlines.

In this model, a schedule is said to be *feasible* if every mandatory subtask M_i is completed within its deadline. A schedule is said to be *precise* if the average error $\overline{\epsilon}$ on the task set is zero. In a precise schedule, all mandatory and optional subtasks are completed within their deadlines.

As an illustrative example, let us consider a set of jobs $\{J_1, \ldots, J_n\}$ shown in Fig. 9.31a. Notice that this task set cannot be precisely scheduled; however, a feasible schedule with an average error of $\overline{\epsilon} = 4$ can be found, and it is shown in Fig. 9.31b. In fact, all mandatory subtasks finish within their deadlines, whereas not all optional subtasks are able to complete. In particular, a time unit of execution is subtracted from O_1, two units from O_3, and one unit from O_5. Hence, assuming that all tasks have an importance value equal to one ($w_i = 1$), the average error on the task set is $\overline{\epsilon} = 4$.

	a_i	d_i	C_i	m_i	o_i
J_1	0	6	4	2	2
J_2	2	7	4	1	3
J_3	4	10	5	2	3
J_4	12	15	3	1	2
J_5	6	20	8	5	3

(a)



(b)

Fig. 9.31 An example of an imprecise schedule

For a set of periodic tasks, the problem of deciding the best level of quality compatible with a given load condition can be solved by associating each optional part of a task a reward function $R_i(\sigma_i)$, which indicates the reward accrued by the task when it receives σ_i units of service beyond its mandatory portion. This problem has been addressed by Aydin et al. [AMMA01], who presented an optimal algorithm that maximizes the weighted average of the rewards over the task set.

Note that in the absence of a reward function, the problem can easily be solved by using a compression algorithm like the elastic approach. In fact, once, the new task utilizations U_i' are computed, the new computation times C_i' that lead to a given desired load can easily be computed from the periods as

$$C_i' = T_i U_i'.$$

Finally, if an algorithm cannot be executed in an incremental fashion or it cannot be aborted at any time, a task can be provided with multiple versions, each characterized by a different quality of performance and execution time. Then, the value C_i' can be used to select the task version having the computation time closer to, but smaller than C_i'.

9.4.4.1 Rate-Adaptive Tasks

This type of task is used to dynamically adapt the computational load in applications where the task activation rate depends on a physical system variable. For example, in automotive applications, engine control tasks are linked to the rotation of the crankshaft; thus, their activation is triggered by the hardware at specific rotation angles. Other computational activities may be linked to the rotation speed of other subsystems (e.g., wheels, gears, engine, cooling fan); thus, their activation rate is proportional to the angular velocity of a specific device. In avionic systems, altimeters are acquired more frequently at low altitudes and, similarly, in mobile robots, proximity sensors could be acquired more frequently when the robot gets closer to obstacles, so their activation rate results to be inversely proportional to the obstacle distance.

A potential problem that may occur with such tasks is that, for high activation rates, the system utilization could increase beyond a limit, generating an overload condition on the control processor. If not properly handled, an overload can have disruptive effects on the controlled system, introducing unbounded delays on the computational activities or even leading to a complete functionality loss. To prevent such a problem, a common practice adopted in automotive applications is to implement such tasks to automatically adapt their computational requirements as a function of the activation rate. For this reason, they are referred to as *rate-adaptive tasks* or *variable-rate tasks* [But12]. In engine control applications, this approach is also justified by considering that at higher speeds the system under control becomes inherently more stable, and therefore, some functions that must execute at lower

Table 9.3 Example of a task with a functionality decreasing with the rotation speed

Crankshaft rotation speed (rpm)	Functions to be executed in a given speed range
[0, 2000]	`f1(); f2(); f3(); f4();`
(2000, 4000]	`f1(); f2(); f3();`
(4000, 6000]	`f1(); f2();`
(6000, 8000]	`f1();`

Fig. 9.32 Implementation of a task with a functionality variable with the rotation speed

```
#define OMEGA1 2000
#define OMEGA2 4000
#define OMEGA3 6000
#define OMEGA4 8000

task  sample_task {
      ω = read_rotation_speed();
      if (ω ≤ OMEGA4) f1();
      if (ω ≤ OMEGA3) f2();
      if (ω ≤ OMEGA2) f3();
      if (ω ≤ OMEGA1) f4();
}
```

speeds do not need to run at higher speeds. This can be exploited to reduce the execution time of rotation-driven tasks at higher rotation speeds.

Table 9.3 illustrates an example of a rate-adaptive task with four levels of functionality, specified for different speed intervals, expressed in rotations per minute (rpm). The implementation of such a type of tasks is typically performed as a sequence of conditional if statements, each executing a specific subset of functions.

Figure 9.32 shows the pseudo code implementing the task illustrated in Table 9.3.

In the actual practice, rate-adaptive tasks are implemented as a set of modes, each operating within a specified range of speeds; in addition, a hysteresis is introduced to avoid frequent mode changes when the engine speed is around to a switching speed. Therefore, the computation time of an angular tasks can be described by a step function consisting of M_i execution modes, where each mode m ($m = 1, \ldots, M_i$) is defined by a computation time C_i^m and a speed range $[\omega_i^{m-}, \omega_i^{m+}]$.

Note that, when considering hysteresis in mode changes, the computation time depends not only on the value of ω but also on the current mode m, leading to two alternative mode-change behaviors, characterized by the following computation functions:

$$C_i^+(\omega) = C_i^m, \quad \forall \omega \in (\omega_i^{(m-1)+}, \omega_i^{m+}] \tag{9.43}$$

Table 9.4 Example of a rate-adaptive task with three modes with different functionality

Mode	C^m	$[\omega^{m-}, \omega^{m+}]$ (rpm)	Functionality
1	C^1	[500, 2000]	`f1()`
2	C^2	[1500, 4000]	`f2()`
3	C^3	[3500, 6000]	`f3()`

Fig. 9.33 Typical implementation of the task shown in Table 9.4

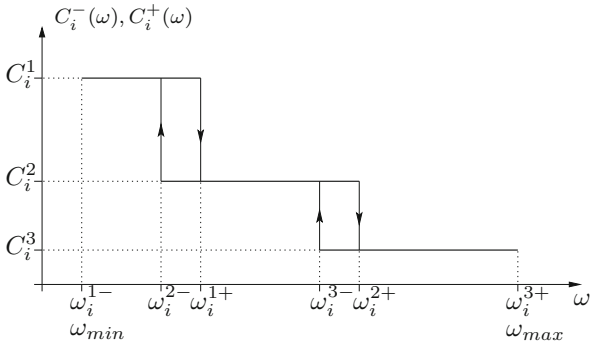
```
//Global variables
mode[M] = {f1(), f2(), f3()};
ω+[M] = {2000, 4000, 6000};
ω−[M] = {500, 1500, 3500};
m = 1;

task sample_angular_task {

    ω = read_rotation_speed();

    while (ω > ω+[m]) m = m + 1;
    while (ω < ω−[m]) m = m − 1;
    execute(mode[m]);
}
```

Fig. 9.34 Task WCET as a function of the rotation speed ω



$$C_i^-(\omega) = C_i^m, \quad \forall \omega \in [\omega_i^{m-}, \omega_i^{(m+1)-}). \tag{9.44}$$

Table 9.4 illustrates an example of a rate-adaptive task with three modes, specified for different (overlapping) speed intervals. Each mode m executes a different function characterized by a computation time C^m and mode transitions have a hysteresis of 500 rpm (for instance, the transition from mode 1 to mode 2 occurs at 2000 rpm, whereas the reverse transition occurs at 1500 rpm).

By defining an array of modes, a rate-adaptive task with hysteresis can be efficiently implemented as illustrated in Fig. 9.33. Figure 9.34 graphically illustrates functions $C_i^-(\omega)$ and $C_i^+(\omega)$ for the AVR task shown in Table 9.4.

The schedulability analysis of rate-adaptive tasks has been deeply investigated in the real-time literature and several results have been derived.

The first task model for rate-adaptive computations was presented by Buttazzo, Bini, and Buttle [BBB14], who also proposed a feasibility test for tasks under steady-state conditions (constant speed), as well as in dynamic conditions (constant acceleration). A design method was also discussed to determine the most suitable switching speeds for adapting the functionality of tasks without exceeding a desired utilization.

Pollex et al. [PFFSa+13] derived a sufficient schedulability test for fixed priorities, under the assumption of a constant engine speed. Davis et al. [DFPS14] analyzed the dynamic behavior of AVR tasks under fixed priority assignments and proposed a sufficient schedulability test using an integer linear programming formulation. This approach, however, is based on speed quantization, which adds more pessimism in the analysis.

A method for computing the exact interference of an rate-adaptive task under fixed priorities has been presented for the first time by Biondi et al. [BMM+14], who used a search-based approach in the speed domain and introduced the concept of dominant speeds to contain the complexity and avoid speed quantization. Then, the exact response time analysis of a mix set of rate-adaptive and periodic tasks was derived by Biondi et al. [BNB15, BNB18]. A feasibility analysis under EDF scheduling was presented by Biondi, Buttazzo, and Simoncelli [BBS15]. Finally, a complete survey of the schedulability analysis for rate-dependent tasks has been carried out by Feld et al. [FBD+18].

Exercises

9.1 For the set of two aperiodic jobs reported in the table, compute the instantaneous load for all instants in [0,8].

	r_i	C_i	D_i
J_1	3	3	5
J_2	0	5	10

9.2 Verify the schedulability under EDF of the set of skippable tasks illustrated in the table:

	C_i	T_i	S_i
τ_1	2	5	∞
τ_2	2	6	4
τ_3	4	8	5

9.3 A resource reservation mechanism achieves temporal isolation by providing service using the following periodic time partition, in every window of 10 units: {[0,2], [5,6], [8,9]}. This means that the service is only provided in the three intervals indicated in the set and repeats every 10 units. Illustrate the resulting supply function $Z(t)$ and compute the (α, Δ) parameters of the corresponding bounded delay function.

9.4 Consider the set of elastic tasks illustrated in the table, to be schedule by EDF:

	C_i	T_i^{min}	T_i^{max}	E_i
τ_1	9	15	30	1
τ_2	16	20	40	3

Since the task set is not feasible with the minimum periods, compute the new periods T_i' that make the task set feasible with a total utilization $U_d = 1$.

9.5 Considering the same periodic task set of the previous exercise, compute the new periods T_i' resulting by applying a period rescaling (hence, not using the elastic coefficients).

Chapter 10
Kernel Design Issues

In this chapter, we present some basic issues that should be considered during the design and the development of a hard real-time kernel for critical control applications. For didactical purposes, we illustrate the structure and the main components of a small real-time kernel, called DICK (*DI*dactic *C K*ernel), mostly written in C language, which is able to handle periodic and aperiodic tasks with explicit time constraints. The problem of time-predictable intertask communication is also discussed, and a particular communication mechanism for exchanging state messages among periodic tasks is illustrated. Finally, we show how the runtime overhead of the kernel can be evaluated and taken into account in the schedulability analysis.

10.1 Structure of a Real-Time Kernel

A kernel represents the innermost part of any operating system that is in direct connection with the hardware of the physical machine. A kernel usually provides the following basic activities:

- Process management
- Interrupt handling
- Process synchronization

Process management is the primary service that an operating system has to provide. It includes various supporting functions, such as process creation and termination, job scheduling, dispatching, context switching, and other related activities.

The objective of the interrupt handling mechanism is to provide service to the interrupt requests that may be generated by any peripheral device, such as the keyboard, serial ports, analog-to-digital converters, or any specific sensor interface. The service provided by the kernel to an interrupt request consists in the execution

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_10

of a dedicated routine (driver) that will transfer data from the device to the main memory (or vice versa). In classical operating systems, application tasks can always be preempted by drivers, at any time. In real-time systems, however, this approach may introduce unpredictable delays in the execution of critical tasks, causing some hard deadline to be missed. For this reason, in a real-time system, the interrupt handling mechanism has to be integrated with the scheduling mechanism, so that a driver can be scheduled as any other task in the system and a guarantee of feasibility can be achieved even in the presence of interrupt requests.

Another important role of the kernel is to provide a basic mechanism for supporting process synchronization and communication. In classical operating systems this is done by semaphores, which represent an efficient solution to the problem of synchronization, as well as to the one of mutual exclusion. As discussed in Chap. 7, however, semaphores are prone to priority inversion, which introduces unbounded blocking on tasks' execution and prevents a guarantee for hard real-time tasks. As a consequence, in order to achieve predictability, a real-time kernel has to provide special types of semaphores that support a resource access protocol (such as priority inheritance, priority ceiling, or stack resource policy) for avoiding unbounded priority inversion. Other kernel activities involve the initialization of internal data structures (such as queues, tables, task control blocks, global variables, semaphores, and so on) and specific services to higher levels of the operating system.

In the rest of this chapter, we describe the structure of a small real-time kernel, called DICK (*DI*dactic *C*l *K*ernel). Rather than showing all implementation details, we focus on the main features and mechanisms that are necessary to handle tasks with explicit time constraints.

DICK is designed under the assumption that all tasks are resident in main memory when it receives control of the processor. This is not a restrictive assumption, as this is the typical solution adopted in kernels for real-time embedded applications.

The various functions developed in DICK are organized according to the hierarchical structure illustrated in Fig. 10.1. Those low-level activities that directly

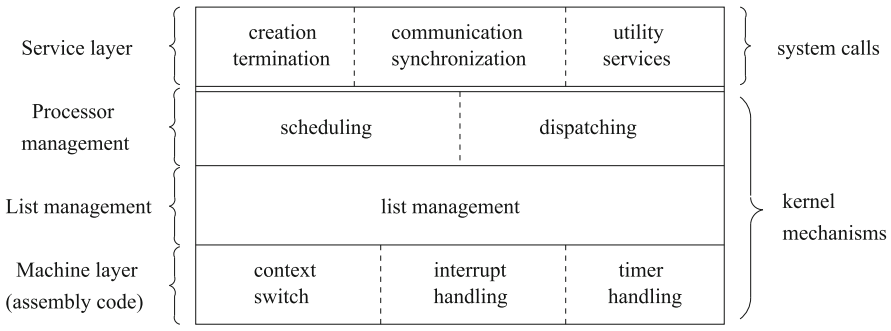


Fig. 10.1 Hierarchical structure of DICK

interact with the physical machine are realized in assembly language. Nevertheless, for the sake of clarity, all kernel activities are described in pseudo C.

The structure of DICK can be logically divided into four layers:

- **Machine layer**. This layer directly interacts with the hardware of the physical machine; hence, it is written in assembly language. The primitives realized at this level mainly deal with activities such as context switch , interrupt handling, and timer handling. These primitives are not visible at the user level.
- **List management layer**. To keep track of the status of the various tasks, the kernel has to manage a number of lists, where tasks having the same state are enqueued. This layer provides the basic primitives for inserting and removing a task to and from a list.
- **Processor management layer**. The mechanisms developed in this layer only concerns scheduling and dispatching operations.
- **Service layer**. This layer provides all services visible at the user level as a set of system calls. Typical services concern task creation, task abortion, suspension of periodic instances, activation and suspension of aperiodic instances, and system inquiry operations.

10.2 Process States

In this section, we describe the possible states in which a task can be during its execution and how a transition from a state to another can be performed.

In any kernel that supports the execution of concurrent activities on a single processor, where semaphores are used for synchronization and mutual exclusion, there are at least three states in which a task can enter:

- **Running**. A task enters this state as it starts executing on the processor.
- **Ready**. This is the state of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task. All tasks that are in this condition are maintained in a queue, called the *ready queue*.
- **Waiting**. A task enters this state when it executes a synchronization primitive to wait for an event. When using semaphores, this operation is a *wait* primitive on a locked semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head of this queue is resumed when the semaphore is unlocked by another task that executed a *signal* on that semaphore. When a task is resumed, it is inserted in the ready queue.

In a real-time kernel that supports the execution of periodic tasks, another state must be considered, the IDLE state. A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period. In order to be awakened by the timer, a periodic job must notify the end of its cycle by executing a specific system call, *end_cycle*, which puts the job in the IDLE state and assigns the processor to another ready job. At the right time, each periodic job

in the IDLE state will be awakened by the kernel and inserted in the ready queue. This operation is carried out by a routine activated by a timer, which verifies, at each tick, whether some job has to be awakened. The state transition diagram relative to the four states described above is shown in Fig. 10.2.

Additional states can be introduced by other kernel services. For example, a *delay* primitive, which suspends a job for a given interval of time, puts the job in a sleeping state (DELAY) , until it will be awakened by the timer after the elapsed interval.

Another state, found in many operating systems, is the RECEIVE state, introduced by the classical message passing mechanism. A job enters this state when it executes a *receive* primitive on an empty channel. The job exits this state when a *send* primitive is executed by another job on the same channel.

In real-time systems that support dynamic creation and termination of hard periodic tasks, a new state needs to be introduced for preserving the bandwidth assigned to the guaranteed tasks. This problem arises because, when a periodic task τ_k is aborted (for example, with a *kill* operation), its utilization factor U_k cannot be immediately subtracted from the total processor load, since the task could already have delayed the execution of other tasks. In order to keep the guarantee test consistent, the utilization factor U_k can be subtracted only at the end of the current period of τ_k.

For example, consider the set of three periodic tasks illustrated in Fig. 10.3, which are scheduled by the rate monotonic algorithm. Computation times are 1, 4, and 4, and periods are 4, 8, and 16, respectively. Since periods are harmonic and the total utilization factor is $U = 1$, the task set is schedulable by RM (remember that $U_{lub} = 1$ when periods are harmonic).

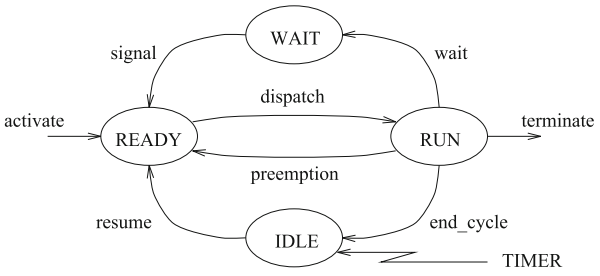


Fig. 10.2 Minimum state transition diagram of a real-time kernel

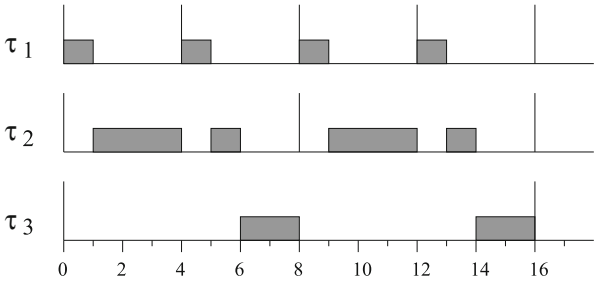


Fig. 10.3 Feasible schedule of three periodic tasks under RM

Now suppose that task τ_2 (with utilization factor $U_2 = 0.5$) is aborted at time $t = 4$ and that, at the same time, a new task τ_{new}, having the same characteristics of τ_2, is created. If the total load of the processor is decremented by 0.5 at time $t = 4$, task τ_{new} would be guaranteed, having the same utilization factor as τ_2. However, as shown in Fig. 10.4, τ_3 would miss its deadline. This happens because the effects of τ_2 execution on the schedule protract until the end of each period.

As a consequence, to keep the guarantee test consistent, the utilization factor of an aborted task can be subtracted from the total load only at the end of the current period. In the interval of time between the abort operation and the end of its period, τ_2 is said to be in a ZOMBIE state, since it does not exist in the system, but it continues to occupy processor bandwidth. Figure 10.5 shows that the task set is schedulable when the activation of τ_{new} is delayed until the end of the current period of τ_2.

A more complete state transition diagram including the states described above (DELAY, RECEIVE, and ZOMBIE) is illustrated in Fig. 10.6. Notice that, at the

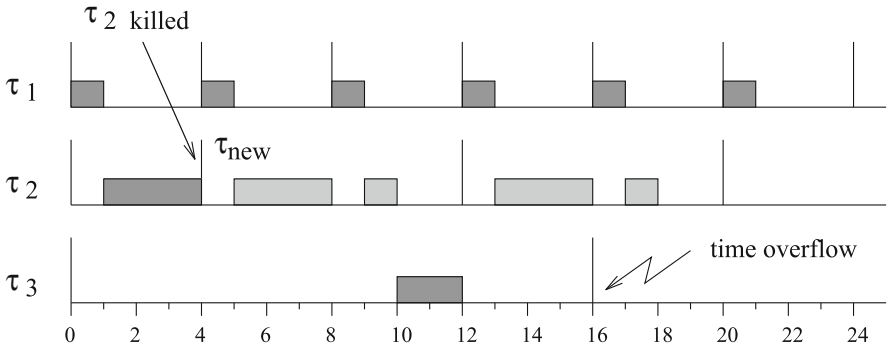


Fig. 10.4 The effects of τ_2 do not cancel at the time it is aborted, but protract till the end of its period

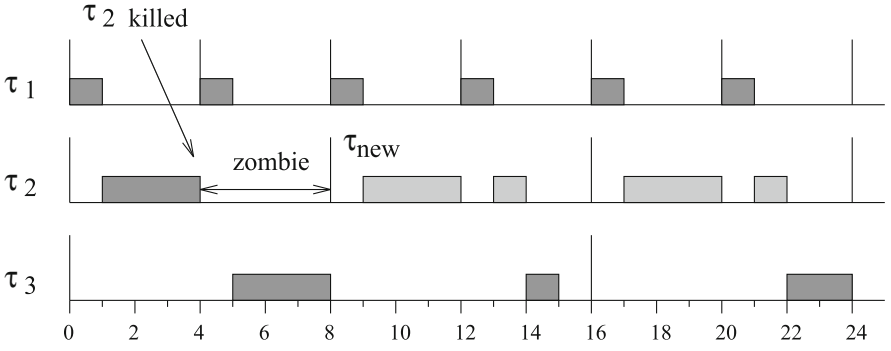


Fig. 10.5 The new task set is schedulable when τ_{new} is activated at the end of the period of τ_2

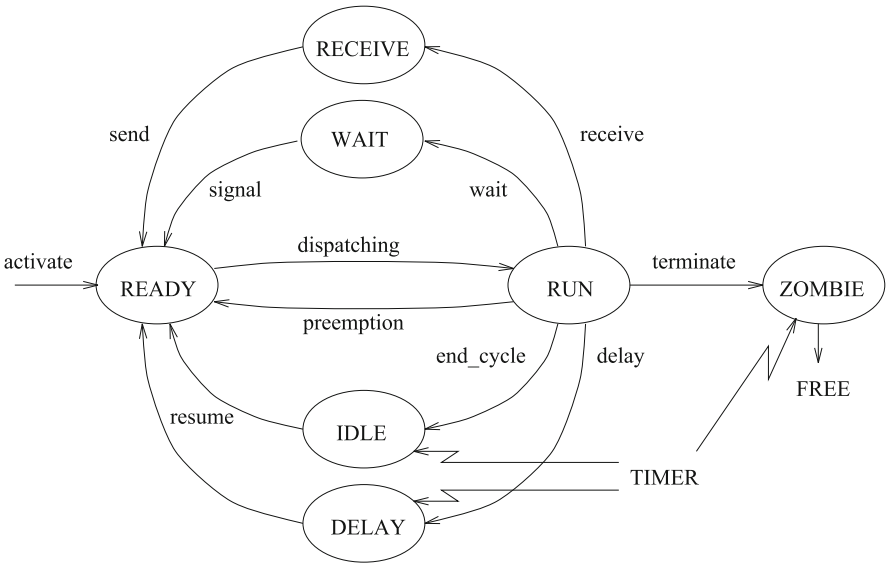


Fig. 10.6 State transition diagram including RECEIVE, DELAY, and ZOMBIE states

end of its last period, a periodic task (aborted or terminated) leaves the system completely and all its data structures are deallocated.

In order to simplify the description of DICK, in the rest of this chapter we describe only the essential functions of the kernel. In particular, the message passing mechanism and the delay primitive are not considered here; as a consequence, the states RECEIVE and DELAY are not present. However, these services can easily be developed on top of the kernel, as an additional layer of the operating system.

In DICK, activation and suspension of aperiodic tasks are handled by two primitives, *activate* and *sleep*, which introduce another state, called SLEEP. An aperiodic task enters the SLEEP state by executing the *sleep* primitive. A task exits the SLEEP state and goes to the READY state only when an explicit activation is performed by another task.

Task creation and activation are separated in DICK. The creation primitive (*create*) allocates and initializes all data structures needed by the kernel to handle the task; however, the task is not inserted in the ready queue, but it is left in the SLEEP state, until an explicit activation is performed. This is mainly done to reduce the runtime overhead of the activation primitive. The state transition diagram used in DICK is illustrated in Fig. 10.7.

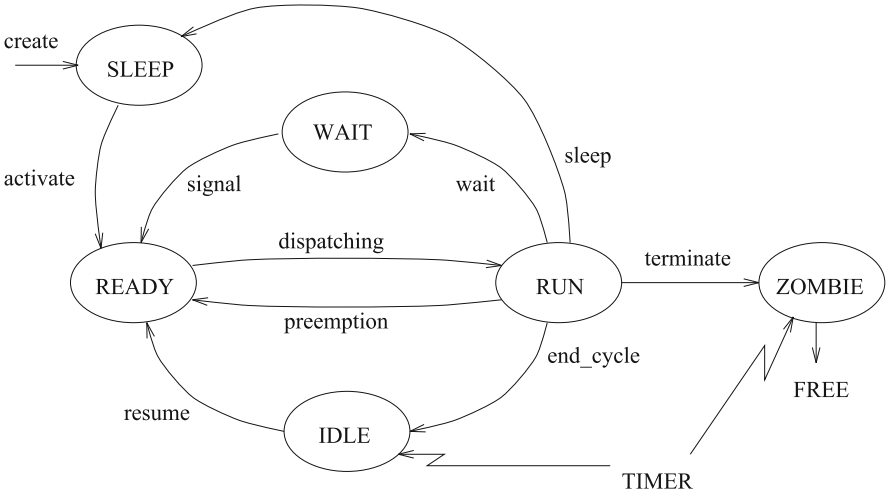


Fig. 10.7 State transition diagram in DICK

10.3 Data Structures

In any operating system, the information about a task are stored in a data structure, the *Task Control Block* (TCB). In particular, a TCB contains all the parameters specified by the programmer at creation time, plus other temporary information necessary to the kernel for managing the task. In a real-time system, the typical fields of a TCB are shown in Fig. 10.8 and contain the following information:

- An identifier, that is, a character string used by the system to refer the task in messages to the user
- The memory address corresponding to the first instruction of the task
- The task type (periodic, aperiodic, or sporadic)
- The task criticality (hard, soft, or non-real-time)
- The priority (or value), which represents the importance of the task with respect to the other tasks of the application
- The current state (ready, running, idle, waiting, and so on)
- The worst-case execution time
- The task period
- The relative deadline, specified by the user
- The absolute deadline, computed by the kernel at the arrival time
- The task utilization factor (only for periodic tasks)
- A pointer to the process stack, where the context is stored
- A pointer to a directed acyclic graph, if there are precedence constraints
- A pointer to a list of shared resources, if a resource access protocol is provided by the kernel

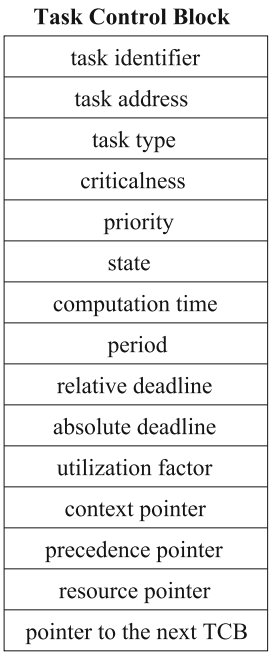


Fig. 10.8 Structure of the task control block

In addition, other fields can be necessary for specific features of the kernel. For example, if aperiodic tasks are handled by one or more server mechanisms, a field can be used to store the identifier of the server associated with the task, or, if the scheduling mechanism supports tolerant deadlines, a field can store the tolerance value for that task.

Finally, since a TCB has to be inserted in the lists handled by the kernel, an additional field has to be reserved for the pointer to the next element of the list.

In DICK, a TCB is an element of the `vdes[MAXPROC]` array, whose size is equal to the maximum number of tasks handled by the kernel. Using this approach, each TCB can be identified by a unique index, corresponding to its position in the `vdes` array. Hence, any queue of tasks can be accessed by an integer variable containing the index of the TCB at the head of the queue. Figure 10.9 shows a possible configuration of the ready queue within the `vdes` array.

Similarly, the information concerning a semaphore is stored in a semaphore control block (SCB), which contains at least the following three fields (see also Fig. 10.10):

- A counter, which represents the value of the semaphore
- A queue, for enqueueing the tasks blocked on the semaphore
- A pointer to the next SCB, to form a list of free semaphores

Each SCB is an element of the `vsem[MAXSEM]` array, whose size is equal to the maximum number of semaphores handled by the kernel. According to this

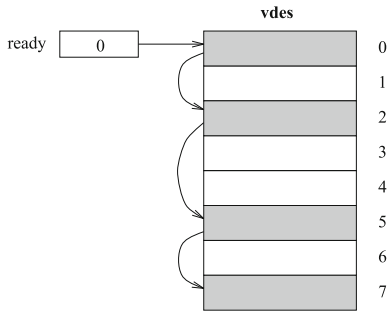


Fig. 10.9 Implementation of the ready queue as a list of task control blocks

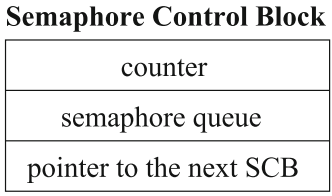


Fig. 10.10 Semaphore control block

approach, tasks, semaphores, and queues can be accessed by an integer number, which represents the index of the corresponding control block. For the sake of clarity, however, tasks, semaphores, and queues are defined as three different types.

```
typedef int    queue;          /* head index        */
typedef int    sem;            /* semaphore index   */
typedef int    proc;           /* process index     */
typedef int    cab;            /* cab buffer index  */
typedef char*  pointer;        /* memory pointer    */
```

```
struct tcb {
   char  name[MAXLEN+1]; /* task name              */
   proc  (*addr)();    /* first instruction address */
   int   type;         /* task type                 */
   int   state;        /* task state                */
   long  dline;        /* absolute deadline         */
   int   period;       /* task period               */
   int   prt;          /* task priority             */
   int   wcet;         /* worst-case execution time */
   float util;         /* task utilization factor   */
   int   *context;     /* pointer to the context    */
   proc  next;         /* pointer to the next tcb   */
   proc  prev;         /* pointer to previous tcb   */
};
```

```
struct scb {
    int    count;       /* semaphore counter          */
    queue  qsem;        /* semaphore queue            */
    sem    next;        /* pointer to the next        */
};
```

```
struct tcb    vdes[MAXPROC];          /*  tcb array  */
struct scb    vsem[MAXSEM];           /*  scb array  */
```

```
proc     pexe;          /* task in execution          */
queue    ready;         /* ready queue                */
queue    idle;          /* idle queue                 */
queue    zombie;        /* zombie queue               */
queue    freetcb;       /* queue of free tcb's        */
queue    freesem;       /* queue of free semaphores   */
float    util_fact;     /* utilization factor         */
```

10.4 Miscellaneous

10.4.1 Time Management

To generate a time reference, a timer circuit is programmed to interrupt the processor at a fixed rate, and the internal system time is represented by an integer variable, which is reset at system initialization and is incremented at each timer interrupt. The interval of time with which the timer is programmed to interrupt defines the unit of time in the system, that is, the minimum interval of time handled by the kernel (time resolution). The unit of time in the system is also called a system *tick*.

In DICK, the system time is represented by a long integer variable, called `sys_clock`, whereas the value of the tick is stored in a float variable called `time_unit`. At any time, `sys_clock` contains the number of interrupts generated by the timer since system initialization.

```
unsigned long    sys_clock;    /* system time          */
float            time_unit;    /* unit of time (ms)    */
```

If Q denotes the system tick and n is the value stored in `sys_clock`, the actual time elapsed since system initialization is $t = nQ$. The maximum time that can be represented in the kernel (the system *lifetime*) depends on the value of the system tick. Considering that `sys_clock` is an unsigned long represented on 32 bits, Table 10.1 shows the values of the system lifetime for some tick values.

The value to be assigned to the tick depends on the specific application. In general, small values of the tick improve system responsiveness and allow to handle

Table 10.1 System lifetime for some typical tick values

Tick	Lifetime
1 ms	50 days
5 ms	8 months
10 ms	16 months
50 ms	7 years

periodic activities with high activation rates. On the other hand, a very small tick causes a large runtime overhead due to the timer handling routine and reduces the system lifetime. Typical values used for the time resolution can vary from 1 to 50 ms.

To have a strict control on task deadlines and periodic activations, all time parameters specified on the tasks should be multiple of the system tick. If the tick can be selected by the user, the best possible tick value is equal to the greatest common divisor of all the task periods.

The timer interrupt handling routine has a crucial role in a real-time system. Other than updating the value of the internal time, it has to check for possible deadline misses on hard tasks, due to some incorrect prediction on the worst-case execution times. Other activities that can be carried out by the timer interrupt handling routine concern lifetime monitoring, activation of periodic tasks that are in idle state, awakening tasks suspended by a delay primitive, checking for deadlock conditions, and terminating tasks in zombie state.

In DICK, the timer interrupt handling routine increments the value of the `sys_clock` variable, checks the system lifetime, checks for possible deadline misses on hard tasks, awakes idle periodic tasks at the beginning of their next period, and, at their deadlines, deallocates all data structures of the tasks in zombie state. In particular, at each timer interrupt, the corresponding handling routine:

- Saves the context of the task in execution
- Increments the system time
- If the current time is greater than the system lifetime, generates a timing error
- If the current time is greater than some hard deadline, generates a time-overflow error
- Awakens those idle tasks, if any, that have to begin a new period
- If at least a task has been awakened, calls the scheduler
- Removes all zombie tasks for which their deadline is expired
- Loads the context of the current task
- Returns from interrupt

The runtime overhead introduced by the execution of the timer routine is proportional to its interrupt rate. In Sect. 10.7 we see how this overhead can be evaluated and taken into account in the schedulability analysis.

10.4.2 Task Classes and Scheduling Algorithm

Real-world control applications usually consist of computational activities having different characteristics. For example, tasks may be periodic, aperiodic, time-driven, and event-driven and may have different levels of criticality. To simplify the description of the kernel, only two classes of tasks are considered in DICK:

- HARD tasks, having a critical deadline
- Non-real-time (NRT) tasks, having a fixed priority

HARD tasks can be activated periodically or aperiodically depending on how an instance is terminated. If the instance is terminated with the primitive *end_cycle*, the task is put in the idle state and automatically activated by the timer at the beginning of its next period; if the instance is terminated with the primitive *end_aperiodic*, the task is put in the sleep state, from where it can be resumed only by explicit activation. HARD tasks are scheduled using the earliest deadline first (EDF) algorithm, whereas NRT tasks are executed in the background based on their priority.

In order to integrate the scheduling of these classes of tasks and avoid the use of two scheduling queues, priorities of NRT tasks are transformed into deadlines so that they are always greater than HARD deadlines. The rule for mapping NRT priorities into deadlines is shown in Fig. 10.11 and is such that

$$d_i^{NRT} = MAXDLINE - PRT\_LEV + P_i,$$

where MAXDLINE is the maximum value of the variable `sys_clock` ($2^{31} - 1$), PRT_LEV is the number of priority levels handled by the kernel, and P_i is the priority of the task, in the range [0, PRT_LEV-1] (0 being the highest priority). Such a priority mapping slightly reduces system lifetime but greatly simplifies task management and queue operations.

10.4.3 Global Constants

In order to clarify the description of the source code, a number of global constants are defined here. Typically, they define the maximum size of the main kernel

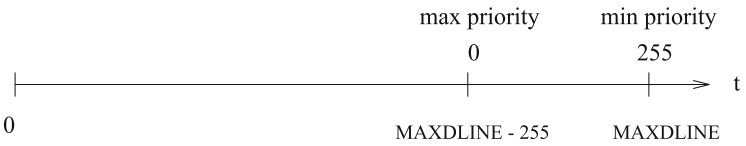


Fig. 10.11 Mapping NRT priorities into deadlines

data structures, such as the maximum number of processes and semaphores, the maximum length of a process name, the number of priority levels, the maximum deadline, and so on. Other global constants encode process classes, states, and error messages. They are listed below:

```
#define MAXLEN      12              /* max string length    */
#define MAXPROC     32              /* max number of tasks  */
#define MAXSEM      32              /* max No of semaphores */

#define MAXDLINE    0x7FFFFFFF      /* max deadline         */
#define PRT_LEV     255             /* priority levels      */
#define NIL         -1              /* null pointer         */
#define TRUE        1
#define FALSE       0

#define LIFETIME    MAXDLINE - PRT_LEV
```

```
/*-------------------------------------------------------------*/
/*                       Task types                            */
/*-------------------------------------------------------------*/
#define HARD        1               /* critical task       */
#define NRT         2               /* non real-time task  */
/*-------------------------------------------------------------*/
/*                       Task states                           */
/*-------------------------------------------------------------*/
#define FREE        0                /* TCB not allocated  */
#define READY       1                /* ready state        */
#define EXE         2                /* running state      */
#define SLEEP       3                /* sleep state        */
#define IDLE        4                /* idle state         */
#define WAIT        5                /* wait state         */
#define ZOMBIE      6                /* zombie state       */
```

```
/*-------------------------------------------------------------*/
/*                     Error messages                          */
/*-------------------------------------------------------------*/
#define OK               0          /* no error             */
#define TIME_OVERFLOW    -1         /* missed deadline      */
#define TIME_EXPIRED     -2         /* lifetime reached     */
#define NO_GUARANTEE     -3         /* task not schedulable */
#define NO_TCB           -4         /* too many tasks       */
#define NO_SEM           -5         /* too many semaphores  */
```

10.4.4 Initialization

The real-time environment supported by DICK starts when the *ini_system* primitive is executed within a sequential C program. After this function is executed, the main program becomes an NRT task in which new concurrent tasks can be created.

The most important activities performed by *ini_system* concern are as follows:

- Initializing all queues in the kernel
- Setting all interrupt vectors
- Preparing the TCB associated with the main process
- Setting the timer period to the system tick

```
void    ini_system(float tick)
{
proc    i;

    time_unit = tick;
    <enable the timer to interrupt every time_unit>
    <initialize the interrupt vector table>

    /* initialize the list of free TCBs and semaphores */

    for (i=0; i<MAXPROC-1; i++) vdes[i].next = i+1;
    vdes[MAXPROC-1].next = NIL;

    for (i=0; i<MAXSEM-1; i++) vsem[i].next = i+1;
    vsem[MAXSEM-1].next = NIL;

    ready = NIL;
    idle = NIL;
    zombie = NIL;

    freetcb = 0;
    freesem = 0;
    util_fact = 0;

    <initialize the TCB of the main process>

    pexe = <main index>;
}
```

10.5 Kernel Primitives

The structure of DICK is logically divided in a number of hierarchical layers, as illustrated in Fig. 10.1. The lowest layer includes all interrupt handling drivers and the routines for saving and loading a task context. The next layer contains the functions for list manipulation (insertion, extraction, and so on) and the basic mechanisms for task management (dispatching and scheduling). All kernel services visible from the user are implemented at a higher level. They concern task creation, activation, suspension, termination, synchronization, and status inquiry.

10.5.1 Low-Level Primitives

Basically, the low-level primitives implement the mechanism for saving and loading the context of a task, that is, the values of the processor registers.

```
/*-------------------------------------------------------------*/
/* save_context -- of the task in execution                    */
/*-------------------------------------------------------------*/

void    save_context(void)
{
int    *pc;                     /* pointer to pexe context  */

    <disable interrupts>
    pc = vdes[pexe].context;
    pc[0] = <register_0>        /* save register 0 */
    pc[1] = <register_1>        /* save register 1 */
    pc[2] = <register_2>        /* save register 2 */
        ...
    pc[n] = <register_n>        /* save register n */
}
```

```
/*-------------------------------------------------------------*/
/* load_context -- of the task to be executed                  */
/*------------------------------------------------------------ */

void    load_context}(void)
{
int    *pc;                      /* pointer to pexe context */

    pc = vdes[pexe].context;

    <register_0> = pc[0];       /* load register 0 */
    <register_1> = pc[1];       /* load register 1 */
        ...
    <register_n> = pc[n];       /* load register n */

    <enable interrupts>
    <return from interrupt>
}
```

10.5.2 List Management

Since tasks are scheduled based on EDF, all queues in the kernel are ordered by decreasing deadlines. In this way, the task with the earliest deadline can be simply extracted from the head of a queue, whereas an insertion operation requires to scan at most all elements of the list. All lists are implemented with bidirectional pointers

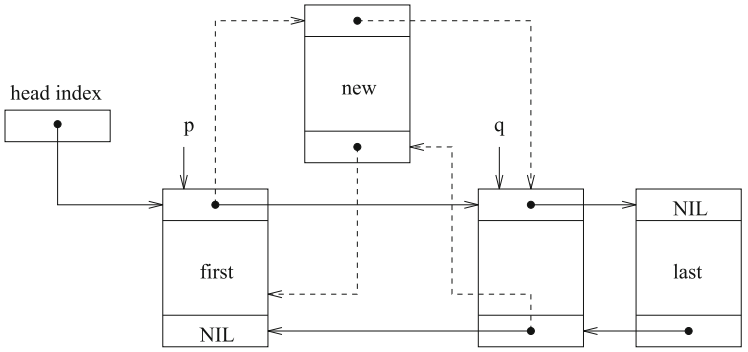


Fig. 10.12 Inserting a TCB in a queue

(next and prev). The *insert* function is called with two parameters: the index of the task to be inserted and the pointer of the queue. It uses two auxiliary pointers, p and q, whose meaning is illustrated in Fig. 10.12.

```
/*-------------------------------------------------------------*/
/* insert -- a task in a queue based on its deadline           */
/*-------------------------------------------------------------*/

void    insert(proc i, queue *que)
{
long   dl;          /* deadline of the task to be inserted  */
int    p;           /* pointer to the previous TCB          */
int    q;           /* pointer to the next TCB              */

    p = NIL;
    q = *que;
    dl = vdes[i].dline;

    /* find the element before the insertion point */

    while ((q != NIL) && (dl >= vdes[q].dline)) {
        p = q;
        q = vdes[q].next;
    }

    if (p != NIL) vdes[p].next = i;
    else *que = i;

    if (q != NIL) vdes[q].prev = i;
    vdes[i].next = q;
    vdes[i].prev = p;
}
```

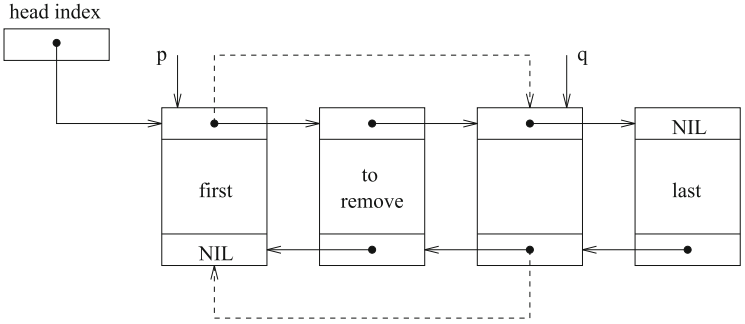


Fig. 10.13 Extracting a TCB from a queue

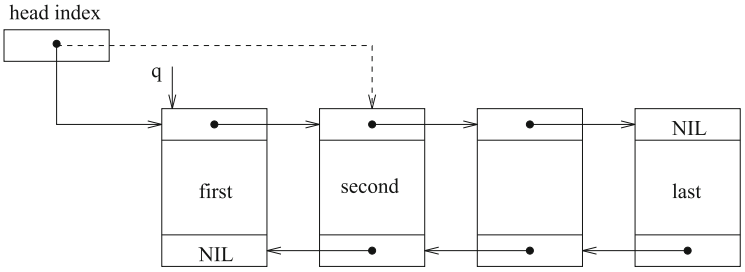


Fig. 10.14 Extracting the TCB at the head of a queue

The major advantage of using bidirectional pointers is in the implementation of the extraction operation, which can be realized in one step without scanning the whole queue. Figure 10.13 illustrates the extraction of a generic element, whereas Fig. 10.14 shows the extraction of the element at the head of the queue.

```
/*-------------------------------------------------------------*/
/* extract -- a task from a queue                              */
/*-------------------------------------------------------------*/

proc    extract(proc i, queue *que)
{
int    p, q;                          /* auxiliary pointers   */

    p = vdes[i].prev;
    q = vdes[i].next;

    if (p == NIL) *que = q;                /* first element    */
    else vdes[p].next = vdes[i].next;

    if (q != NIL) vdes[q].prev = vdes[i].prev;
    return(i);
}
```

```
/*-------------------------------------------------------------*/
/* getfirst -- extracts the task at the head of a queue        */
/*-------------------------------------------------------------*/

proc    getfirst(queue *que)
{
int    q;                  /* pointer to the first element  */

    q = *que;
    if (q == NIL) return(NIL);

    *que = vdes[q].next;
    vdes[*que].prev = NIL;
    return(q);
}
```

Finally, to simplify the code reading of the next levels, two more functions are defined: *firstdline* and *empty*. The former returns the deadline of the task at the head of the queue, while the latter returns TRUE if a queue is empty, FALSE otherwise.

```
/*-------------------------------------------------------------*/
/* firstdline -- returns the deadline of the first task        */
/*-------------------------------------------------------------*/

long    firstdline(queue *que)
{
    return(vdes[que].dline);
}
```

```
/*-------------------------------------------------------------*/
/* empty -- returns TRUE if a queue is empty                   */
/*-------------------------------------------------------------*/

int    empty(queue *que)
{
    if (que == NIL)
        return(TRUE);
    else
        return(FALSE);
}
```

10.5.3 Scheduling Mechanism

The scheduling mechanism in DICK is realized through the functions *schedule* and *dispatch*. The *schedule* primitive verifies whether the running task is the one with the earliest deadline. If so, no action is done; otherwise, the running task is inserted in the ready queue and the first ready task is dispatched. The *dispatch* primitive just assigns the processor to the first ready task.

```
/*-------------------------------------------------------*/
/* schedule -- selects the task with the earliest deadline */
/*-------------------------------------------------------*/

void   schedule(void)
{
       if (firstdline(ready) < vdes[pexe].dline) {
          vdes[pexe].state = READY;
          insert(pexe, &ready);
          dispatch();
       }
}
```

```
/*-------------------------------------------------------*/
/* dispatch -- assigns the cpu to the first ready task     */
/*-------------------------------------------------------*/

void    dispatch(void)
{
    pexe = getfirst(&ready);
    vdes[pexe].state = RUN;
}
```

The timer interrupt handling routine is called *wake_up* and performs the activities described in Sect. 10.4.1. In summary, it increments the *sys_clock* variable, checks for the system lifetime and possible deadline misses, removes those tasks in zombie state whose deadlines are expired, and, finally, resumes those periodic tasks in idle state at the beginning of their next period. Note that if at least a task has been resumed, the scheduler is invoked and a preemption takes place.

```
/*-------------------------------------------------------*/
/* wake_up -- timer interrupt handling routine           */
/*-------------------------------------------------------*/

void    wake_up(void)
{
proc    p;
int     count = 0;

    save_context();
    sys_clock++;
    if (sys_clock >= LIFETIME) abort(TIME_EXPIRED);

    if (vdes[pexe].type == HARD)
        if (sys_clock > vdes[pexe].dline)
            abort(TIME_OVERFLOW);

    while (!empty(zombie) &&
           (firstdline(zombie) <= sys_clock)) {
        p = getfirst(&zombie);
        util_fact = util_fact - vdes[p].util;
        vdes[p].state = FREE;
        insert(p, &freetcb);
    }

    while (!empty(idle) && (firstdline(idle) <= sys_clock)) {
        p = getfirst(&idle);
        vdes[p].dline += (long)vdes[p].period;
        vdes[p].state = READY;
        insert(p, &ready);
        count++;
    }

    if (count > 0) schedule();
    load_context();
}
```

10.5.4 Task Management

It concerns creation, activation, suspension, and termination of tasks. The *create* primitive allocates and initializes all data structures needed by a task and puts the task in SLEEP. A guarantee is performed for HARD tasks.

```
/*------------------------------------------------------------*/
/* create -- creates a task and puts it in sleep state      */
/*------------------------------------------------------------*/

proc    create(
        char    name[MAXLEN+1],      /* task name           */
        proc    (*addr)(),           /* task address        */
        int     type,                /* type (HARD, NRT)    */
        float   period,              /* period or priority */
        float   wcet)                /* execution time      */
{
proc    p;

    <disable cpu interrupts>
    p = getfirst(&freetcb);
    if (p == NIL) abort(NO_TCB);

    if (vdes[p].type == HARD)
        if (!guarantee(p)) return(NO_GUARANTEE);

    vdes[p].name = name;
    vdes[p].addr = addr;
    vdes[p].type = type;
    vdes[p].state = SLEEP;
    vdes[p].period = (int)(period / time_unit);
    vdes[p].wcet = (int)(wcet / time_unit);
    vdes[p].util = wcet / period;
    vdes[p].prt = (int)period;
    vdes[p].dline = MAX_LONG + (long)(period - PRT_LEV);

    <initialize process stack>
    <enable cpu interrupts>
    return(p);
}
```

```
/*------------------------------------------------------------*/
/* guarantee -- guarantees the feasibility of a hard task  */
/*------------------------------------------------------------*/

int   guarantee(proc p)
{
      util_fact = util_fact + vdes[p].util;
      if (util_fact > 1.0) {
          util_fact = util_fact - vdes[p].util;
          return(FALSE);
      }
      else return(TRUE);
}
```

The system call *activate* inserts a task in the ready queue, performing the transition SLEEP–READY. If the task is HARD, its absolute deadline is set equal to the current time plus its period. Then the scheduler is invoked to select the task with the earliest deadline.

```
/*------------------------------------------------------------*/
/* activate -- inserts a task in the ready queue              */
/*------------------------------------------------------------*/

int   activate(proc p)
{
      save_context();
      if (vdes[p].type == HARD)
          vdes[p].dline = sys_clock + (long)vdes[p].period;

      vdes[p].state = READY;
      insert(p, &ready);
      schedule();
      load_context();
}
```

The transition RUN–SLEEP is performed by the *sleep* system call. The running task is suspended in the sleep state, and the first ready task is dispatched for execution. Notice that this primitive acts on the calling task, which can be periodic or aperiodic. For example, the *sleep* primitive can be used at the end of a cycle to terminate an aperiodic instance.

```
/*------------------------------------------------------------*/
/* sleep -- suspends itself in a sleep state                  */
/*------------------------------------------------------------*/

void    sleep(void)
{
        save_context();
        vdes[pexe].state = SLEEP;
        dispatch();
        load_context();
}
```

The primitive for terminating a periodic instance is a bit more complex than its aperiodic counterpart, since the kernel has to be informed on the time at which the timer has to resume the job. This operation is performed by the primitive *end_cycle* , which puts the running task into the idle queue. Since it is assumed that deadlines are at the end of the periods, the next activation time of any idle periodic instance coincides with its current absolute deadline.

In the particular case in which a periodic job finishes exactly at the end of its period, the job is inserted not in the idle queue but directly in the ready queue, and its deadline is set to the end of the next period.

```
/*------------------------------------------------------*/
/* end_cycle -- inserts a task in the idle queue        */
/*------------------------------------------------------*/

void    end_cycle(void)
{
long    dl;

        save_context();
        dl = vdes[pexe].dline;

        if (sys_clock < dl) {
            vdes[pexe].state = IDLE;
            insert(pexe, &idle);
        }
        else {
            dl = dl + (long)vdes[pexe].period;
            vdes[pexe].dline = dl;
            vdes[pexe].state = READY;
            insert(pexe, &ready);
        }

        dispatch();
        load_context();
}
```

A typical example of periodic task has the following structure:

```
/*---------------------------------------*/
proc  cycle()
{
<local variables>

      <initialization>

      while (TRUE) {
          <periodic code>
          end_cycle();
      }
}
```

There are two primitives for terminating a process: The first, called *end_process*, directly operates on the calling task; the other one, called *kill*, terminates the task passed as a formal parameter. Notice that, if the task is HARD, it is not immediately removed from the system but put in ZOMBIE state. In this case, the complete removal will be done by the timer routine at the end of the current period:

```
/*-------------------------------------------------------------*/
/* end_process -- terminates the running task                  */
/*-------------------------------------------------------------*/

void    end_process(void)
{
        <disable cpu interrupts>

        if (vdes[pexe].type == HARD)
            insert(pexe, &zombie);
        else {
            vdes[pexe].state = FREE;
            insert(pexe, &freetcb);
        }

        dispatch();
        load_context();
}
```

```
/*-------------------------------------------------------------*/
/* kill -- terminates a task                                   */
/*-------------------------------------------------------------*/

void    kill(proc p)
{
        <disable cpu interrupts>

        if (pexe == p) {
            end_process();
            return;
        }

        if (vdes[p].state == READY) extract(p, &ready);
        if (vdes[p].state == IDLE)  extract(p, &idle);

        if (vdes[p].type == HARD)
            insert(p, &zombie);
        else {
            vdes[p].state = FREE;
            insert(p, &freetcb);
        }

        <enable cpu interrupts>
}
```

10.5.5 Semaphores

In DICK, synchronization and mutual exclusion are handled by semaphores. Four primitives are provided to the user to allocate a new semaphore (*newsem*), deallocate a semaphore (*delsem*), wait for an event (*wait*), and signal an event (*signal*).

The *newsem* primitive allocates a free semaphore control block and initializes the counter field to the value passed as a parameter. For example, `s1 = newsem(0)` defines a semaphore for synchronization, whereas `s2 = newsem(1)` defines a semaphore for mutual exclusion. The *delsem* primitive just deallocates the semaphore control block, inserting it in the list of free semaphores.

```
/*-------------------------------------------------------------*/
/* newsem -- allocates and initializes a semaphore             */
/*-------------------------------------------------------------*/

sem   newsem(int n)
{
sem   s;

      <disable cpu interrupts>

      s = freesem;           /* first free semaphore index */
      if (s == NIL) abort(NO_SEM);

      freesem = vsem[s].next;   /* update the freesem list */
      vsem[s].count = n;        /* initialize counter      */
      vsem[s].qsem = NIL;       /* initialize sem.~queue   */

      <enable cpu interrupts>
      return(s);
}
```

```
/*-------------------------------------------------------------*/
/* delsem -- deallocates a semaphore                           */
/*-------------------------------------------------------------*/

void  delsem(sem s)
{
      <disable cpu interrupts>

      vsem[s].next = freesem;    /* inserts s at the head  */
      freesem = s;               /* of the freesem list    */

      <enable cpu interrupts>
}
```

The *wait* primitive is used by a task to wait for an event associated to a semaphore. If the semaphore counter is positive, it is decremented, and the task continues its execution; if the counter is less than or equal to zero, the task is

blocked, and it is inserted in the semaphore queue. In this case, the first ready task is assigned to the processor by the *dispatch* primitive.

To ensure the consistency of the kernel data structures, all semaphore system calls are executed with cpu interrupts disabled. Notice that semaphore queues are ordered by decreasing absolute deadlines, so that, when more tasks are blocked, the first task awakened will be the one with the earliest deadline.

```
/*------------------------------------------------------------*/
/* wait -- waits for an event                                 */
/*------------------------------------------------------------*/

void    wait(sem s)
{
        <disable cpu interrupts>

        if (vsem[s].count > 0) vsem[s].count--;
        else {
            save_context();
            vdes[pexe].state = WAIT;
            insert(pexe, &vsem[s].qsem);
            dispatch();
            load_context();
        }

        <enable cpu interrupts>
}
```

The *signal* primitive is used by a task to signal an event associated with a semaphore. If no tasks are blocked on that semaphore (that is, if the semaphore queue is empty), the counter is incremented, and the task continues its execution. If there are blocked tasks, the task with the earliest deadline is extracted from the semaphore queue and is inserted in the ready queue. Since a task has been awakened, a context switch may occur; hence, the context of the running task is saved, a task is selected by the scheduler, and a new context is loaded.

```
/*------------------------------------------------------------*/
/* signal -- signals an event                                 */
/*------------------------------------------------------------*/

void    signal(sem s)
{
proc    p;

        <disable cpu interrupts>

        if (!empty(vsem[s].qsem)) {
            p = getfirst(&vsem[s].qsem);
            vdes[p].state = READY;
            insert(p, &ready);
            save_context();
            schedule();
            load_context();
        }
        else  vsem[s].count++;

        <enable cpu interrupts>
}
```

It is worth observing that classical semaphores are prone to the priority inversion phenomenon, which introduces unbounded delays during tasks' execution and prevents any form of guarantee on hard tasks (this problem is discussed in Chap. 7). As a consequence, this type of semaphores should be used only by non-real-time tasks, for which no guarantee is performed. Real-time tasks, instead, should rely on more predictable mechanisms, based on time-bounded resource access protocols (such as stack resource policy) or on asynchronous communication buffers. In DICK, the communication among hard tasks occurs through an asynchronous buffering mechanism, which is described in Sect. 10.6.

10.5.6 Status Inquiry

DICK also provides some primitives for inquiring the kernel about internal variables and task parameters. For example, the following primitives allow to get the system time, the state, the deadline, and the period of a desired task.

```
/*------------------------------------------------------------*/
/* get_time -- returns the system time in milliseconds        */
/*------------------------------------------------------------*/

float   get_time(void)
{
        return(time_unit *sys_clock);
}
```

```
/*-------------------------------------------------------------*/
/* get_state -- returns the state of a task                    */
/*-------------------------------------------------------------*/

int     get_state(proc p)
{
        return(vdes[p].state);
}
```

```
/*-------------------------------------------------------------*/
/* get_dline -- returns the deadline of a task                 */
/*-------------------------------------------------------------*/

long    get_dline(proc p)
{
        return(vdes[p].dline);
}
```

```
/*-------------------------------------------------------------*/
/* get_period -- returns the period of a task                  */
/*-------------------------------------------------------------*/

float   get_period(proc p)
{
        return(vdes[p].period);
}
```

10.6 Intertask Communication Mechanisms

Intertask communication is a critical issue in real-time systems, even in a uniprocessor environment. In fact, the use of shared resources for implementing message passing schemes may cause priority inversion and unbounded blocking on tasks' execution. This would prevent any guarantee on the task set and would lead to a highly unpredictable timing behavior.

In this section, we discuss problems and solutions related to a typical communication semantics used in operating systems: the synchronous and the asynchronous model.

In the pure synchronous communication model, whenever two tasks want to communicate, they must be synchronized for a message transfer to take place. This synchronization is called a *rendezvous*. Thus, if the sender starts first, it must wait until the recipient receives the message; on the other hand, if the recipient starts first, it must wait until the sender produces its message.

In a dynamic real-time system, synchronous communication schemes easily lead to unpredictable behavior, due to the difficulty of estimating the maximum blocking time for a process *rendezvous*. In a static real-time environment, the problem can be solved offline by transforming all synchronous interactions into precedence

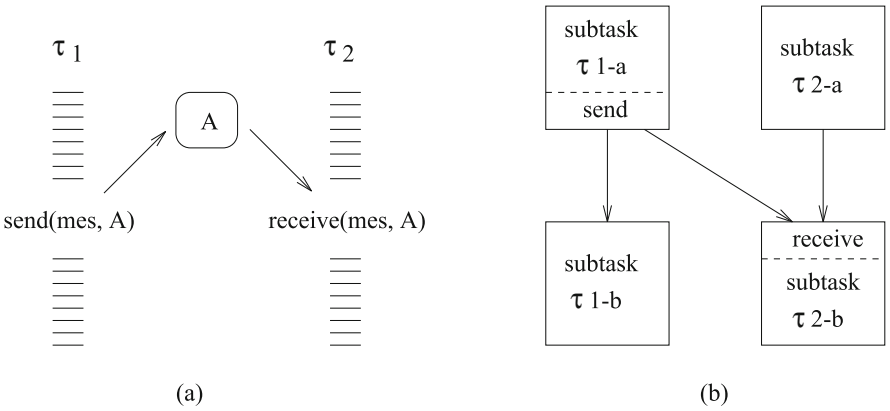


Fig. 10.15 Decomposition of communicating tasks (**a**) into subtasks with precedence constraints (**b**)

constraints. According to this approach, each task is decomposed into a number of subtasks that contain communication primitives not inside their code but only at their boundary. In particular, each subtask can receive messages only at the beginning of its execution and can send messages only at the end. Then a precedence relation is imposed between all adjacent subtasks deriving from the same father task and between all subtasks communicating through a send-receive pair. An example of such a task decomposition is illustrated in Fig. 10.15.

In a pure asynchronous scheme, communicating tasks do not have to wait for each other. The sender just deposits its message into a channel and continues its execution, independently of the recipient condition. Similarly, assuming that at least a message has been deposited into the channel, the receiver can directly access the message without synchronizing with the sender.

Asynchronous communication schemes are more suitable for dynamic real-time systems. In fact, if no unbounded delays are introduced during tasks' communication, timing constraints can easily be guaranteed without increasing the complexity of the system (for example, overconstraining the task set with additional precedence relations). Remember that having simple online guarantee tests (that is, with polynomial time complexity) is crucial for dynamic systems.

In most commercial real-time operating systems, the asynchronous communication scheme is implemented through a *mailbox* mechanism, illustrated in Fig. 10.16. A mailbox is a shared memory buffer capable of containing a fixed number of messages that are typically kept in a FIFO queue. The maximum number of messages that at any instant can be held in a mailbox represents its *capacity*.

Two basic operations are provided on a mailbox: *send* and *receive*. A *send(MX, mes)* operation causes the message *mes* to be inserted in the queue of mailbox *MX*. If at least a message is contained on mailbox *MX*, a *receive(MX, mes)* operation extracts the first message from its queue. Notice that, if the kernel provides the necessary support, more than two tasks can share a mailbox, and channels

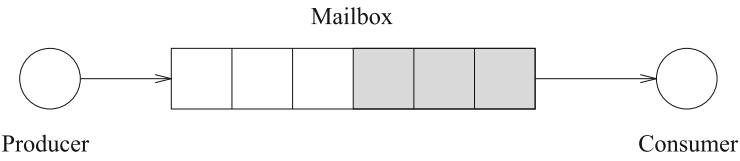


Fig. 10.16 The mailbox scheme

with multiple senders and/or multiple receivers can be realized. As long as it is guaranteed that a mailbox is never empty and never full, sender(s) and receiver(s) are never blocked.

Unfortunately, a mailbox provides only a partial solution to the problem of asynchronous communication, since it has a bounded capacity. Unless sender and receiver have particular arrival patterns, it is not possible to guarantee that the mailbox queue is never empty or never full. If the queue is full, the sender must be delayed until some message is received. If the queue is empty, the receiver must wait until some message is inserted.

For example, consider two periodic tasks, τ_1 and τ_2, with periods T_1 and T_2, that exchange messages through a mailbox having a capacity of n. Let τ_1 be the sender and τ_2 the receiver. If $T_1 < T_2$, the sender inserts in the mailbox more messages than the receiver can extract; thus, after a while the queue becomes full and the sender must be delayed. From this time on, the sender has to wait the receiver, so it synchronizes with its period (T_2). Vice versa, if $T_1 > T_2$, the receiver reads faster than the sender can write; thus, after a while the queue becomes empty and the receiver must wait. From this time on, the receiver synchronizes with the period of the sender (T_1). In conclusion, if $T_1 \neq T_2$, sooner or later both tasks will run at the lowest rate, and the task with the shortest period will miss its deadline.

An alternative approach to asynchronous communication is provided by acyclical asynchronous buffers, which are described in the next section.

10.6.1 Cyclical Asynchronous Buffers

Cyclical asynchronous buffers, or CABs, represent a particular mechanism purposely designed for the cooperation among periodic activities, such as control loops and sensory acquisition tasks. This approach was first proposed by Clark [Cla89] for implementing a robotic application based on hierarchical servo-loops, and it is used in the HARTIK system [But93, BDN93] as a basic communication support among periodic hard tasks.

A CAB provides a one-to-many communication channel, which at any instant contains the latest message or data inserted in it. A message is not consumed (that is, extracted) by a receiving process but is maintained into the CAB structure until a new message is overwritten. As a consequence, once the first message has been put

in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.

Notice that, using such a semantics, a message can be read more than once if the receiver is faster than the sender, while messages can be lost if the sender is faster than the receiver. However, this is not a problem in many control applications, where tasks are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition task.

CABs can be created and initialized by the *open_cab* primitive, which requires specifying the CAB name, the dimension of the message, and the number of messages that the CAB may contain simultaneously. The *delete_cab* primitive removes a CAB from the system and releases the memory space used by the buffers.

To insert a message in a CAB, a task must first reserve a buffer from the CAB memory space, then copy the message into the buffer, and finally put the buffer into the CAB structure, where it becomes the most recent message. This is done according to the following scheme:

```
buf_pointer = reserve(cab_id);
< copymessagein * buf_pointer >
putmes(buf_pointer, cab_id);
```

Similarly, to get a message from a CAB, a task has to get the pointer to the most recent message, use the data, and release the pointer. This is done according to the following scheme:

```
mes_pointer = getmes(cab_id);
< usemessage >
unget(mes_pointer, cab_id);
```

Notice that more tasks can simultaneously access the same buffer in a CAB for reading. On the other hand, if a task P reserves a CAB for writing while another task Q is using that CAB, a new buffer is created, so that P can write its message without interfering with Q. As P finishes writing, its message becomes the most recent one in that CAB. The maximum number of buffers that can be created in a CAB is specified as a parameter in the *open_cab* primitive. To avoid blocking, this number must be equal to the number of tasks that use the CAB plus one.

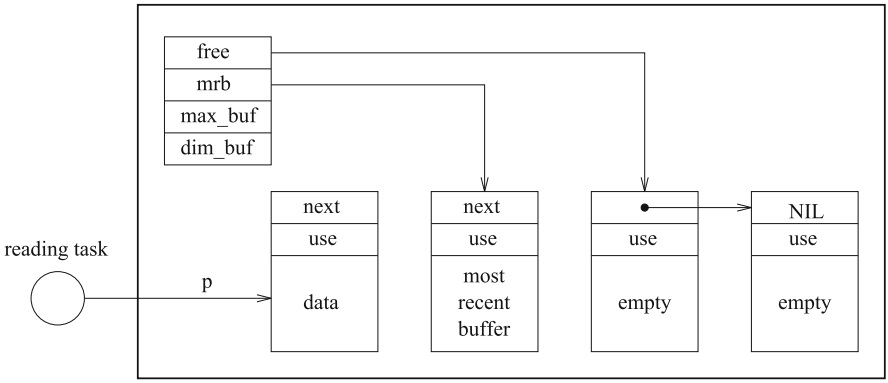


Fig. 10.17 CAB data structure

10.6.2 CAB Implementation

The data structure used to implement a CAB is shown in Fig. 10.17. A CAB control block must store the maximum number of buffers (*max_buf*), their dimension (*dim_buf*), a pointer to a list of free buffers (*free*), and a pointer to the most recent buffer (*mrb*). Each buffer in the CAB can be implemented as a data structure with three fields: a pointer (*next*) to maintain a list of free buffers, a counter (*use*) that stores the current number of tasks accessing that buffer, and a memory area (*data*) for storing the message.

The code of the four CAB primitives is shown below. Notice that the main purpose of the *putmes* primitive is to update the pointer to the most recent buffer (MRB). Before doing that, however, it deallocates the old MRB if no tasks are accessing that buffer. Similarly, the *unget* primitive decrements the number of tasks accessing that buffer and deallocates the buffer only if no task is accessing it and it is not the MRB.

```
/*-------------------------------------------------------------*/
/* reserve -- reserves a buffer in a CAB                       */
/*-------------------------------------------------------------*/

pointer  reserve(cab c)
{
pointer  p;

      <disable cpu interrupts>

      p = c.free;             /* get a free buffer    */
      c.free = p.next;        /* update the free list */
      return(p);

      <enable cpu interrupts>
}
```

```
/*-------------------------------------------------------------*/
/* putmes -- puts a message in a CAB                           */
/*-------------------------------------------------------------*/

void  putmes(cab c, pointer p)
{
      <disable cpu interrupts>

      if (c.mrb.use == 0) {         /* if not accessed,   */
          c.mrb.next = c.free;      /* deallocate the mrb */
          c.free = c.mrb;
      }
      c.mrb = p;                    /* update the mrb     */

      <enable cpu interrupts>
}
```

```
/*-------------------------------------------------------------*/
/* getmes -- gets a pointer to the most recent buffer          */
/*-------------------------------------------------------------*/

pointer  getmes(cab c)
{
pointer  p;

      <disable cpu interrupts>

      p = c.mrb;                  /* get the pointer to mrb */
      p.use = p.use + 1;          /* increment the counter  */
      return(p);

      <enable cpu interrupts>
}
```

```
/*-------------------------------------------------------------*/
/* unget -- deallocates a buffer only if it is not used        */
/*          and it is not the most recent buffer               */
/*-------------------------------------------------------------*/

void  unget(cab c, pointer p)
{
      <disable cpu interrupts>

      p.use = p.use - 1;
      if ((p.use == 0) && (p != c.mrb)) {
          p.next = c.free;
          c.free = p;
      }

      <enable cpu interrupts>
}
```

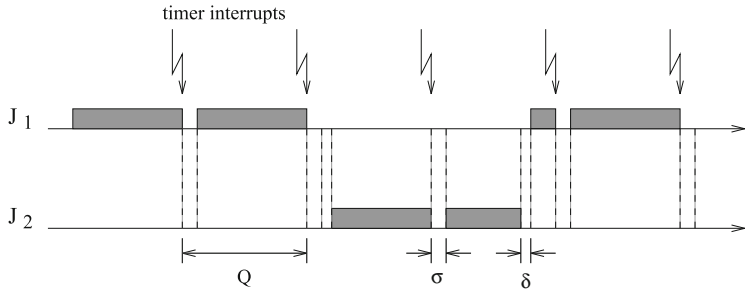


Fig. 10.18 Effects of the overhead on tasks' execution

10.7 System Overhead

The overhead of an operating system represents the time used by the processor for handling all kernel mechanisms, such as enqueueing tasks, performing context switches, updating the internal data structures, sending messages to communication channels, servicing the interrupt requests, and so on. The time required to perform these operations is usually much smaller than the execution times of the application tasks; hence, it can be neglected in the schedulability analysis and in the resulting guarantee test. In some cases, however, when application tasks have small execution times and tight timing constraints, the activities performed by the kernel may not be so negligible and may create a significant interference on tasks' execution. In these situations, predictability can be achieved only by considering the effects of the runtime overhead in the schedulability analysis.

The context switch time is one of the most significant overhead factors in any operating system. It is an intrinsic limit of the kernel that does not depend on the specific scheduling algorithm, nor on the structure of the application tasks. For a real-time system, another important overhead factor is the time needed by the processor to execute the timer interrupt handling routine. If Q is the system tick (that is, the period of the interrupt requests from the timer) and σ is the worst-case execution time of the corresponding driver, the timer overhead can be computed as the utilization factor U_t of an equivalent periodic task:

$$U_t = \frac{\sigma}{Q}.$$

Figure 10.18 illustrates the execution intervals (σ) due to the timer routine and the execution intervals (δ) necessary for a context switch.

The effects of the timer routine on the schedulability of a periodic task set can be taken into account by adding the factor U_t to the total utilization of the task set. This is the same as reducing the least upper bound of the utilization factor U_{lub} by U_t, so that the net bound becomes

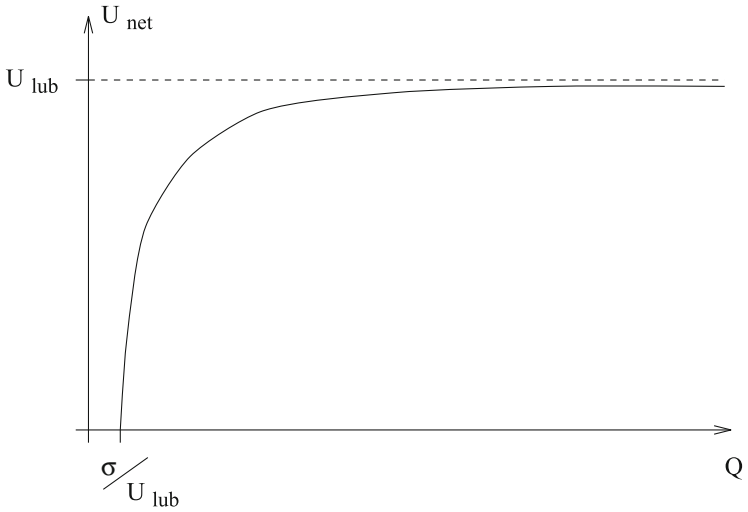


Fig. 10.19 Net utilization bound as a function of the tick value

$$U_{net} = U_{lub} - U_t = U_{lub} - \frac{\sigma}{Q} = U_{lub}\left(\frac{Q - \sigma/U_{lub}}{Q}\right).$$

From this result we can notice that, to have $U_{net} > 0$, the system tick Q must always be greater than (σ/U_{lub}). The plot of U_{net} as a function of Q is illustrated in Fig. 10.19. To have an idea of the degradation caused by the timer overhead, consider a system based on the EDF algorithm ($U_{lub} = 1$) and suppose that the timer interrupt handling routine has an execution time of $\sigma = 100$ μs. In this system, a 10-ms tick would cause a net utilization bound $U_{net} = 0.99$; a 1-ms tick would decrease the net utilization bound to $U_{net} = 0.9$, whereas a 200 μs tick would degrade the net bound to $U_{net} = 0.5$. This means that, if the greatest common divisor among the task periods is 200 μs, a task set with utilization factor $U = 0.6$ cannot be guaranteed under this system.

The overhead due to other kernel mechanisms can be taken into account as an additional term on tasks' execution times. In particular, the time needed for explicit context switches (that is, the ones triggered by system calls) can be considered in the execution time of the kernel primitives; thus, it will be charged to the worst-case execution time of the calling task. Similarly, the overhead associated with implicit context switches (that is, the ones triggered by the kernel) can be charged to the preempted tasks.

In this case, the schedulability analysis requires a correct estimation of the total number of preemptions that each task may experience. In general, for a given scheduling algorithm, this number can be estimated offline as a function of tasks' timing constraints. If N_i is the maximum number of preemptions that a periodic task τ_i may experience in each period, and δ is the time needed to perform a context

switch, the total utilization factor (overhead included) of a periodic task set can be computed as

$$U_{tot} = \sum_{i=1}^{n} \frac{C_i + \delta N_i}{T_i} + U_t = \sum_{i=1}^{n} \frac{C_i}{T_i} + \left(\delta \sum_{i=1}^{n} \frac{N_i}{T_i} + U_t \right).$$

Hence, we can write

$$U_{tot} = U_p + U_{ov},$$

where U_p is the utilization factor of the periodic task set and U_{ov} is a correction factor that considers the effects of the timer handling routine and the preemption overhead due to intrinsic context switches (explicit context switches are already considered in the C_i's terms):

$$U_{ov} = U_t + \delta \sum_{i=1}^{n} \frac{N_i}{T_i}.$$

Finally, notice that an upper bound for the number of preemptions N_i on a task τ_i can be computed as

$$N_i = \sum_{k=1}^{i-1} \left\lfloor \frac{T_i}{T_k} \right\rfloor.$$

However, this bound is too pessimistic, and better bounds can be found for particular scheduling algorithms.

10.7.1 Accounting for Interrupt

Two basic approaches can be used to handle interrupts coming from external devices. One method consists of associating an aperiodic or sporadic task to each source of interrupt. This task is responsible for handling the device and is subject to the scheduling algorithm as any other task in the system. With this method, the cost for handling the interrupt is automatically taken into account by the guarantee mechanism, but the task may not start immediately, due to the presence of higher-priority hard tasks. This method cannot be used for those devices that require immediate service for avoiding data loss.

Another approach allows interrupt handling routines to preempt the current task and execute immediately at the highest priority. This method minimizes the interrupt latency, but the interrupt handling cost has to be explicitly considered in the guarantee of the hard tasks.

Jeffay and Stone [JS93] found a schedulability condition for a set of n hard tasks and m interrupt handlers. In their work, the analysis is carried out by assuming a discrete time, with a resolution equal to a tick. As a consequence, every event in the system occurs at a time that is multiple of the tick. In their model, there is a set $\mathcal{I}$ of m handlers, characterized by a worst-case execution time C_i^H and a minimum separation time T_i^H, just as sporadic tasks. The difference is that interrupt handlers always have a priority higher than the application tasks.

The upper bound, $f(l)$, for the interrupt handling cost in any time interval of length l can be computed by the following recurrent relation [JS93]:

$$f(0) = 0$$
$$f(l) = \begin{cases} f(l-1) + 1 \text{ if } \sum_{i=1}^{m} \left\lceil \frac{l}{T_i^H} \right\rceil C_i^H > f(l-1) \\ f(l-1) \qquad \text{otherwise.} \end{cases} \tag{10.1}$$

In the particular case in which all the interrupt handlers start at time $t = 0$, function $f(l)$ is exactly equal to the amount of time spent by the processor in executing interrupt handlers in the interval $[0, l]$.

Theorem 10.1 (Jeffay-Stone) *A set $\mathcal{T}$ of n periodic or sporadic tasks and a set $\mathcal{I}$ of m interrupt handlers is schedulable by EDF if and only if for all L, $L \geq 0$,*

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor C_i \quad \leq \quad L - f(L). \tag{10.2}$$

The proof of Theorem 10.1 is very similar to the one presented for Theorem 4.5. The only difference is that, in any interval of length L, the amount of time that the processor can dedicate to the execution of application tasks is equal to $L - f(L)$.

It is worth to notice that Eq. (10.2) can be checked only for a set of points equal to release times less than the hyperperiod, and the complexity of the computation is pseudo-polynomial.

Chapter 11
Application Design Issues

This chapter discusses some crucial issues related to the design and the development of complex real-time applications requiring sensory acquisition, control, and actuation of mechanical components. The aim of this part is to give a precise characterization of control applications, so that theory developed for real-time computing and scheduling algorithms can be practically used in this field to make complex control systems more reliable. In fact, a precise observation of the timing constraints specified in the control loops and in the sensory acquisition processes is a necessary condition for guaranteeing a stable behavior of the controlled system, as well as a predictable performance.

As specific examples of control activities, we consider some typical robotic applications, in which a robot manipulator equipped with a set of sensors interacts with the environment to perform a control task according to stringent user requirements. In particular, we discuss when control applications really need real-time computing (and not just fast computing), and we show how time constraints, such as periods and deadlines, can be derived from the application requirements, even though they are not explicitly specified by the user.

Finally, the basic set of kernel primitives presented in Chap. 10 is used to illustrate some concrete programming examples of real-time tasks for sensory processing and control activities.

11.1 Introduction

All complex control applications that require the support of a computing system can be characterized by the following components:

1. The **system** to be controlled. It can be a plant, a car, a robot, or any physical device that has to exhibit a desired behavior.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_11

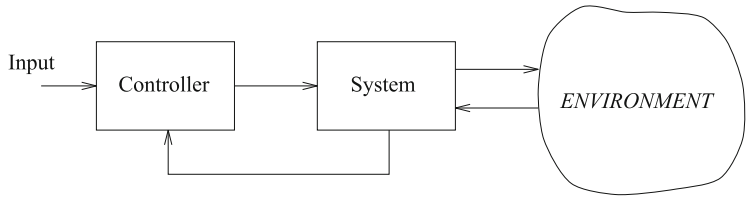


Fig. 11.1 Block diagram of a generic control system

2. The **controller**. For our purposes, it will be a computing system that has to provide proper inputs to the controlled system based on a desired control objective.
3. The **environment**. It is the external world in which the controlled system has to operate.

The interactions between the controlled system and the environment are, in general, bidirectional and occur by means of two peripheral subsystems (considered part of the controlled system): an *actuation* subsystem, which modifies the environment through a number of actuators (such as motors, pumps, engines, and so on), and a *sensory* subsystem, which acquires information from the environment through a number of sensing devices (such as microphones, cameras, transducers, and so on). A block diagram of the typical control system components is shown in Fig. 11.1. Depending on the interactions between the controlled system and the environment, three classes of control systems can be distinguished:

1. Monitoring systems,
2. Open-loop control systems, and
3. Feedback control systems.

Monitoring systems do not modify the environment but only use sensors to perceive its state, process sensory data, and display the results to the user. A block diagram of this type of system is shown in Fig. 11.2. Typical applications of these systems include radar tracking, air traffic control, environmental pollution monitoring, surveillance, and alarm systems. Many of these applications require periodic acquisitions of multiple sensors, and each sensor may need a different sampling rate. Moreover, if sensors are used to detect critical conditions, the sampling rate of each sensor has to be constant in order to perform a correct reconstruction of the external signals. In these cases, using a hard real-time kernel is a necessary condition for guaranteeing a predictable behavior of the system. If sensory acquisition is carried out by a set of concurrent periodic tasks (characterized by proper periods and deadlines), the task set can be analyzed off-line to verify the feasibility of the schedule within the imposed timing constraints.

Open-loop control systems are systems that interact with the environment. However, the actions performed by the actuators do not strictly depend on the current state of the environment. Sensors are used to plan actions, but there is no

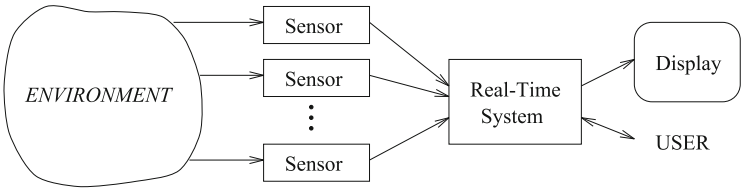


Fig. 11.2 General structure of a monitoring system

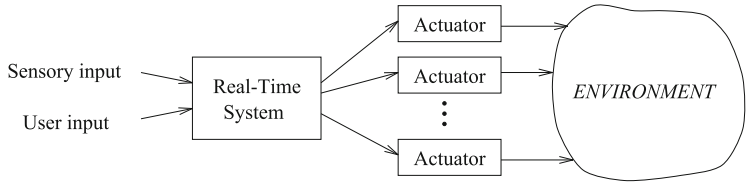


Fig. 11.3 General structure of an open-loop control system

feedback between sensors and actuators. This means that, once an action is planned, it can be executed independently of new sensory data (see Fig. 11.3).

As a typical example of an open-loop control system, consider a robot workstation equipped with a vision subsystem, whose task is to take a picture of an object, identify its location, and send the coordinates to the robot for triggering a pick and place operation. In this task, once the object location is identified and the arm trajectory is computed based on visual data, the robot motion does not need to be modified online; therefore, no real-time processing is required. Notice that real-time computing is not needed even though the pick and place operation has to be completed within a deadline. In fact, the correct fulfillment of the robot operation does not depend on the kernel but on other factors, such as the action planner, the processing speed of visual data, and the robot speed. For this control problem, fast computing and smart programming may suffice to meet the goal.

Feedback control systems (or closed-loop control systems) are systems that have frequent interactions with the environment in both directions; that is, the actions produced by the actuators strictly depend on the current sensory information. In these systems, sensing and control are tied together, and one or more feedback paths exist from the sensory subsystem to the controller. Sensors are often mounted on actuators and are used to probe the environment and continuously correct the actions based on actual data (see Fig. 11.4).

Human beings are perhaps the most sophisticated examples of feedback control systems. When we explore an unknown object, we do not just see it, but we look at it actively, and, in the course of looking, our pupils adjust to the level of illumination, our eyes bring the world into sharp focus, our eyes converge or diverge, we move our head or change our position to get a better view of it, and we use our hands to perceive and enhance tactile information.

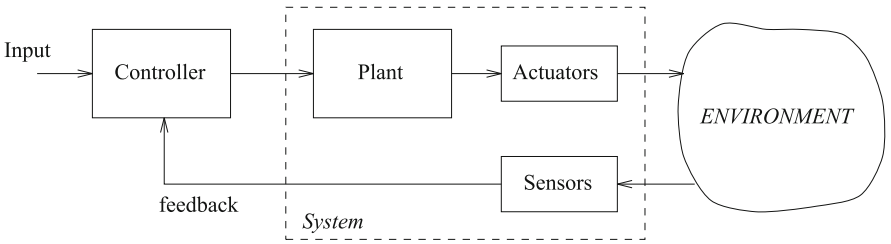


Fig. 11.4 General structure of a feedback control system

Modern "fly-by-wire" aircrafts are also good examples of feedback control systems. In these aircrafts, the basic maneuvering commands given by the pilot are converted into a series of inputs to a computer, which calculates how the physical flight controls shall be displaced to achieve a maneuver, in the context of the current flight conditions.

The robot workstation described above as an example of open-loop control system can also be a feedback control system if we close a loop with the camera and use the current visual data to update the robot trajectory online. For instance, visual feedback becomes necessary if the robot has to grasp a moving object whose trajectory is not known a priori.

In feedback control systems, the use of real-time computing is essential for guaranteeing a predictable behavior; in fact, the stability of these systems depends not only on the correctness of the control algorithms but also on the timing constraints imposed on the feedback loops. In general, when the actions of a system strictly depend on actual sensory data, wrong or late sensor readings may cause wrong or late actions on the environment, which may have negative effects on the whole system. In some case, the consequences of a late action can even be catastrophic. For example, in certain environmental conditions, under autopilot control, reading the altimeter too late could cause the aircraft to stall in a critical flight configuration that could prevent recovery. In delicate robot assembling operations, missing deadlines on force readings could cause the manipulator to exert too much force on the environment, generating an unstable behavior.

These examples show that, when developing critical real-time applications, the following issues should be considered in detail, in addition to the classical design issues:

1. Structuring the application in a number of concurrent tasks, related to the activities to be performed;
2. Assigning the proper timing constraints to tasks; and
3. Using a predictable operating environment that allows to guarantee that those timing constraints can be satisfied.

These and other issues are discussed in the following sections.

11.2 Time Constraints Definition

When we say that a system reacts in *real time* within a particular environment, we mean that its response to any event in that environment has to be effective, according to some control strategy, while the event is occurring. This means that, in order to be effective, a control task must produce its results within a specific deadline, which is defined based on the characteristics of the environment and the system itself.

If meeting a given deadline is critical for the system operation and may cause catastrophic consequences, the task must be treated as a *hard* task. If meeting time constraints is desirable, but missing a deadline does not cause any serious damage, the task can be treated as a *soft* task. In addition, activities that require regular activation should be handled as *periodic* tasks.

From the operating system point of view, a periodic task is a task whose activation is directly controlled by the kernel in a time-driven fashion, so that it is intrinsically guaranteed to be regular. Vice versa, an aperiodic task is a task that is activated by other application tasks or by external events. Hence, activation requests for an aperiodic task may come from the explicit execution of specific system calls or from the arrival of an interrupt associated with the task. Notice that, even though the external interrupts arrive at regular intervals, the associated task should still be handled as an aperiodic task by the kernel, unless precise upper bounds on the activation rate are guaranteed for that interrupt source.

If the interrupt source is well-known and interrupts are generated at a constant rate, or have a minimum interarrival time, then the aperiodic task associated with the corresponding event is said to be *sporadic* and its timing constraints can be guaranteed in worst-case assumptions—that is, assuming the maximum activation rate.

Once all application tasks have been identified and time constraints have been specified (including periodicity and criticality), the real-time operating system supporting the application is responsible for guaranteeing that all hard tasks complete within their deadlines. Soft and non-real-time tasks should be handled by using a best-effort strategy (or optimal, whenever possible) to reduce (or minimize) their average response times.

In the following, a few examples of control systems are illustrated to show how time constraints can be derived from the application requirements even when they are not explicitly defined by the user.

11.2.1 Automatic Braking System

Consider a wheel-vehicle equipped with range sensors that has to operate in a certain environment running within a maximum given speed. The vehicle could be a completely autonomous system, such as a robot mobile base, or a partially

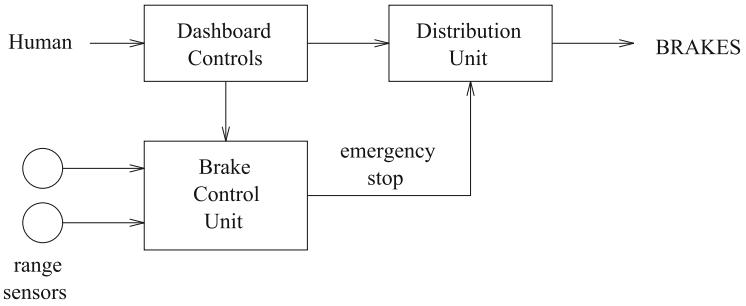


Fig. 11.5 Scheme of the automatic braking system

autonomous system driven by a human, such as a car or a train having an automatic braking system for stopping motion in emergency situations.

To simplify our discussion and reduce the number of controlled variables, let us consider a vehicle like a train, which moves along a straight line, and suppose that we have to design an automatic braking system able to detect obstacles in front of the vehicle and control the brakes to avoid collisions. A block diagram of the automatic braking system is illustrated in Fig. 11.5.

The brake control unit (BCU) is responsible for acquiring a pair of range sensors, computing the distance to the obstacle (if any), reading the state variables of the vehicle from instruments on the dashboard, and deciding whether an emergency stop has to be superimposed. Given the criticality of the braking action, this task has to be periodically executed on the BCU. Let T be its period.

In order to determine a safe value for T, several factors have to be considered. In particular, the system must ensure that the maximum latency from the time at which an obstacle appears and the time at which the vehicle reaches a complete stop is less than the time to impact. Equivalently, the distance D of the obstacle from the vehicle must always be greater than the minimum space L needed for a complete stop. To compute the length L, consider the plot illustrated in Fig. 11.6, which shows the velocity v of the vehicle as a function of time when an emergency stop is performed.

Three time intervals have to be taken in to account to compute the worst-case latency:

- The detection delay, from the time at which an obstacle appears on the vehicle trajectory and the time at which the obstacle is detected by the BCU. This interval is at most equal to the period T of the sensor acquisition task.
- The transmission delay, Δ_t, from the time at which the stop command is activated by the BCU and the time at which the command starts to be actuated by the brakes.
- The braking duration, Δ_b, needed for a complete stop.

If v is the actual velocity of the vehicle and μ_f is the wheel-road friction coefficient, the braking duration Δ_b is given by

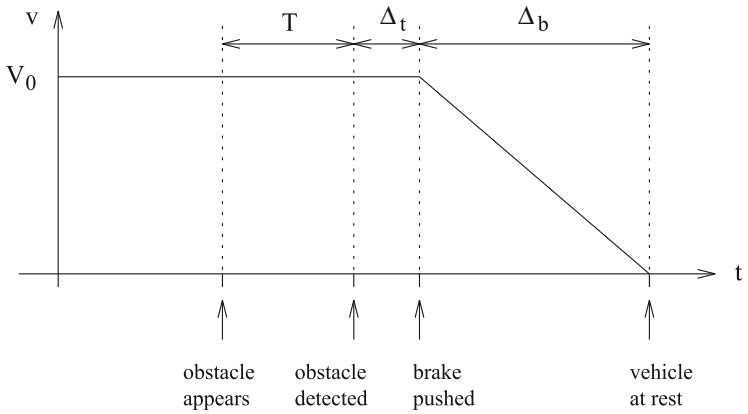


Fig. 11.6 Velocity during brake

$$\Delta_b = \frac{v}{\mu_f g},$$

where g is the acceleration of gravity ($g = 9.8\,\text{m/s}^2$). Thus, the resulting braking space x_b is

$$x_b = \frac{v^2}{2\mu_f g}.$$

Hence, the total length L needed for a complete stop is

$$L = v(T + \Delta_t) + x_b.$$

By imposing $D > L$, we obtain the relation that must be satisfied among the variables to avoid a collision:

$$D > \frac{v^2}{2\mu_f g} + v(T + \Delta_t). \tag{11.1}$$

If we assume that obstacles are fixed and are always detected at a distance D from the vehicle, Eq. (11.1) allows determining the maximum value that can be assigned to period T:

$$T < \frac{D}{v} - \frac{v}{2\mu_f g} - \Delta_t. \tag{11.2}$$

For example, if $D = 100$ m, $\mu_f = 0.5$, $\Delta_t = 250\,\text{ms}$, and $v_{max} = 30\,\text{m/s}$ (about 108 km/h), then the resulting sampling period T must be less than 22 ms.

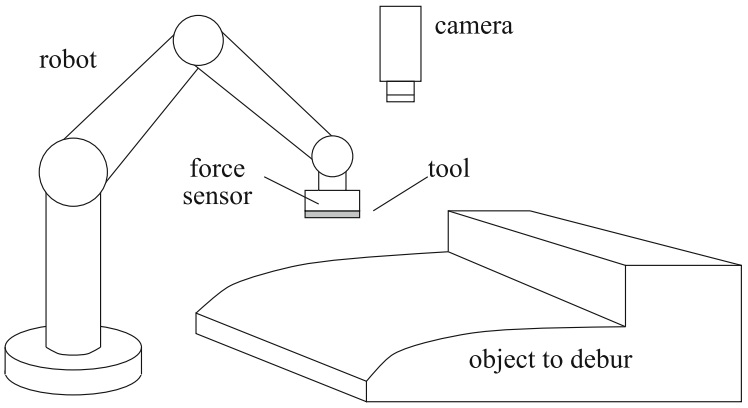


Fig. 11.7 Example of a robot deburring workstation

It is worth observing that this result can also be used to evaluate how long we can look away from the road while driving at a certain speed and visibility. For example, if $D = 50\,\text{m}$ (visibility under fog conditions), $\mu_f = 0.5$, $\Delta_t = 300\,\text{ms}$ (our typical reaction time), and $v = 60\,\text{km/h}$ (about 16.67 m/s or 37 mi/h), we can look away from the road for no more than one second!

11.2.2 Robot Deburring

Consider a robot arm that has to polish an object surface with a grinding tool mounted on its wrist, as shown in Fig. 11.7. This task can be specified as follows:

> Slide the grinding tool on the object surface with a constant tangential speed v while exerting a constant normal force F that must not exceed a maximum value equal to F_{max}.

In order to maintain a constant contact force against the object surface, the robot must be equipped with a force sensor, mounted between the wrist flange and the grinding tool. Moreover, to keep the normal force within the specified maximum value, the force sensor must be acquired periodically at a constant rate, which has to be determined based on the characteristics of the environment and the task requirements. At each cycle, the robot trajectory is corrected based on the current force readings.

As illustrated in Fig. 11.8, if T is the period of the control process and v is the tool tangential speed, the space covered by the robot end-effector within each period is $L_T = vT$. If an impact due to a contour variation occurs just after the force sensor has been read, the contact will be detected at the next period; thus, the robot keeps moving for a distance L_T against the object, exerting an increasing force that depends on the elastic coefficient of the robot-object interaction.

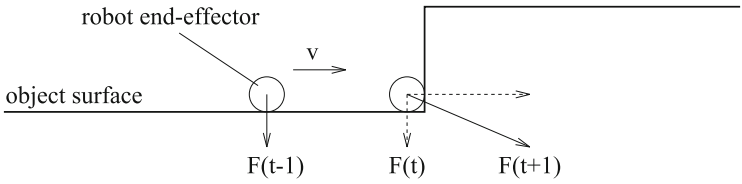


Fig. 11.8 Force on the robot tool during deburring

As the contact is detected, we also have to consider the braking space L_B covered by the tool from the time at which the stop command is delivered to the time at which the robot is at complete rest. This delay depends on the robot dynamic response and can be computed as follows. If we approximate the robot dynamic behavior with a transfer function having a dominant pole f_d (as typically done in most cases), then the braking space can be computed as $L_B = v\tau_d$, being $\tau_d = \frac{1}{2\pi f_d}$. Hence, the longest distance that can be covered by the robot after a collision is given by

$$L = L_T + L_B = v(T + \tau_d).$$

If K is the rigidity coefficient of the contact between the robot end-effector and the object, then the worst-case value of the normal force exerted on the surface is $F_h = KL = Kv(T + \tau_d)$. Since F_h has to be maintained below a maximum value F_{max}, we must impose that

$$Kv(T + \tau_d) < F_{max},$$

which means that

$$T < \left(\frac{F_{max}}{Kv} - \tau_d\right). \tag{11.3}$$

Notice that, in order to be feasible, the right side of condition (11.3) must not only be greater than zero but must also be greater than the system time resolution, fixed by the system tick Q, that is,

$$\frac{F_{max}}{Kv} - \tau_d > Q. \tag{11.4}$$

Equation (11.4) imposes an additional restriction on the application. For example, we may derive the maximum speed of the robot during the deburring operation as

$$v < \frac{F_{max}}{K(Q + \tau_d)}, \tag{11.5}$$

or, if v cannot be arbitrarily reduced, we may fix the tick resolution such that

$$Q \leq \left(\frac{F_{max}}{K v} - \tau_d \right).$$

Once the feasibility is achieved—that is, condition (11.4) is satisfied—the result expressed in Eq. (11.3) says that stiff environments and high robot velocities requires faster control loops to guarantee that force does not exceed the limit given by F_{max}.

11.2.3 Multilevel Feedback Control

In complex control applications characterized by nested servo loops, the frequencies of the control tasks are often chosen to separate the dynamics of the controllers. This greatly simplifies the analysis of the stability and the design of the control law.

Consider, for instance, the control architecture shown in Fig. 11.9. Each layer of this control hierarchy effectively decomposes an input task into simpler subtasks executed at lower levels. The top-level input command is the goal, which is successively decomposed into subgoals, or subtasks, at each hierarchical level, until at the lowest level, output signals drive the actuators. Sensory data enter this hierarchy at the bottom and are filtered through a series of sensory-processing and pattern-recognition modules arranged in a hierarchical structure. Each module processes the incoming sensory information, applying filtering techniques, extracting features, computing parameters, and recognizing patterns.

Sensory information that is relevant to control is extracted and sent as feedback to the control unit at the same level; the remaining partially processed data is then passed to the next higher level for further processing. As a result, feedback enters this hierarchy at every level. At the lowest level, the feedback is almost unprocessed and hence is fast-acting with very short delays, while at higher levels, feedback passes through more and more stages and hence is more sophisticated but slower. The implementation of such a hierarchical control structure has two main implications:

- Since the most recent data have to be used at each level of control, information can be sent through asynchronous communication primitives, using overwrite semantic and non-consumable messages. The use of asynchronous message passing mechanisms avoids blocking situations and allows the interaction among periodic tasks running at different frequencies.
- When the frequencies of hierarchical nested servo loops differ for about an order of magnitude, the analysis of the stability and the design of the control laws are significantly simplified.

For instance, if at the lowest level a joint position servo is carried out with a period of 1 ms, a force control loop closed at the middle level can be performed with a period of 10 ms, while a vision process running at the higher control level can be executed with a period of 100 ms.

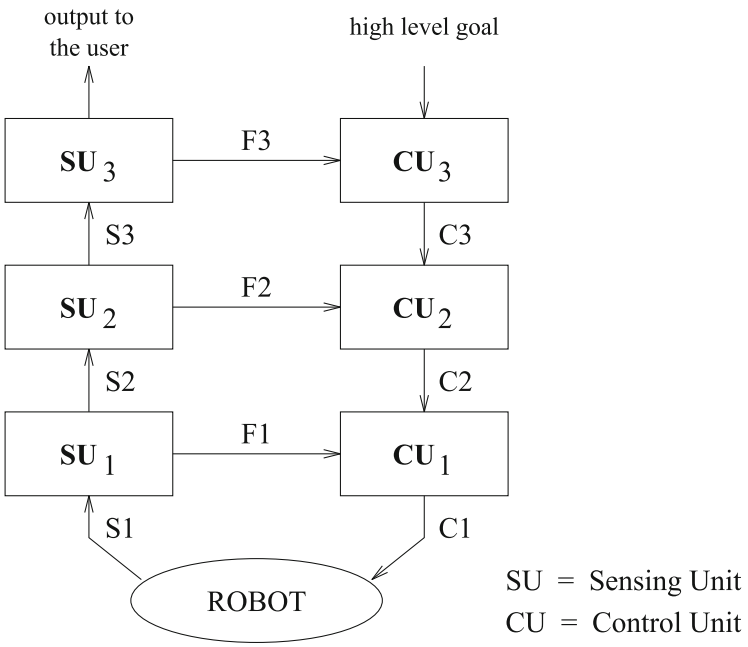


Fig. 11.9 Example of a hierarchical control system

11.3 Hierarchical Design

In this section, we present a hierarchical design approach that can be used to develop sophisticated control applications requiring sensory integration and multiple feedback loops. Such a design approach has been actually adopted and experimented on several robot control applications built on top of a hard real-time kernel [But91, BAF94, But96].

The main advantage of a hierarchical design approach is to simplify the implementation of complex tasks and provide a flexible programming interface, in which most of the low- and middle-level real-time control strategies are built in the system as part of the controller and hence can be viewed as basic capabilities of the system.

Figure 11.10 shows an example of a hierarchical programming environment for complex robot applications. Each layer provides the robot system with new functions and more sophisticated capabilities. The importance of this approach is not simply that one can divide the program into parts; rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a building block in defining other procedures.

The *Device Level* includes a set of modules specifically developed to manage all peripheral devices used for low-level I/O operations, such as sensor acquisition, joint servo, and output display. Each module provides a set of library functions, whose

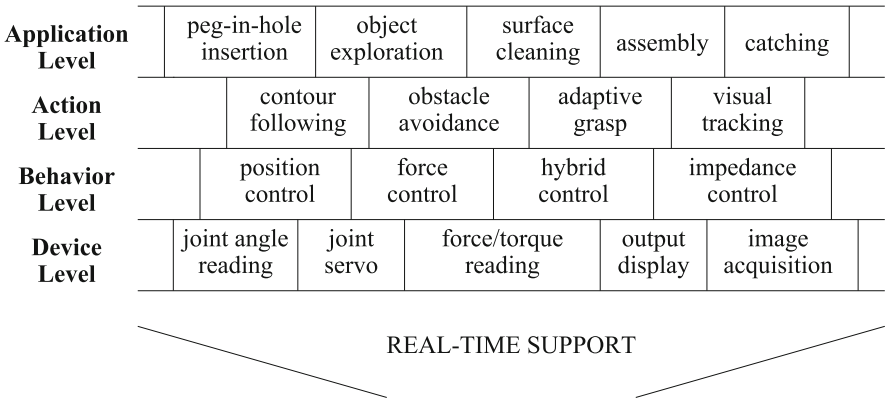


Fig. 11.10 Hierarchical software environment for programming complex robotic applications

purpose is to facilitate device handling and to encapsulate hardware details, so that higher-level software can be developed independently from the specific knowledge of the peripheral devices.

The *Behavior Level* is the level in which several sensor-based control strategies can be implemented to give the robot different kinds of behavior. The functions available at this level of the hierarchy allow the user to close real-time control loops, by which the robot can modify its trajectories based on sensory information, apply desired forces and torques on the environment, operate according to hybrid control schemes, or behave as a mechanical impedance.

These basic control strategies are essential for executing autonomous tasks in unknown conditions, and, in fact, they are used in the next level to implement more skilled actions.

Based on the control strategies developed in the Behavior Level, the *Action Level* enhances the robot capability by adding more sophisticated sensory-motor activities, which can be used at the higher level for carrying out complex tasks in unstructured environments. Some representative actions developed at this level include (1) the ability of the robot to follow an unknown object contour, maintaining the end-effector in contact with the explored surface; (2) the reflex to avoid obstacles, making use of visual sensors; (3) the ability to adapt the end-effector to the orientation of the object to be grasped, based on the reaction forces sensed on the wrist; and (4) visual tracking, to follow a moving object and keep it at the center of the visual field. Many other different actions can be easily implemented at this level by using the modules available at the Behavior Level or directly taking the suited sensory information from the functions at the Device Level.

Finally, the *Application Level* is the level at which the user defines the sequence of robot actions for accomplishing application tasks, such as assembling mechanical parts, exploring unknown objects, manipulating delicate materials, or catching moving targets. Notice that these tasks, although sophisticated in terms of control,

can be readily implemented thanks to the action primitives included in the lower levels of the hierarchical control architecture.

11.3.1 Examples of Real-Time Robotics Applications

In this section we describe a number of robot applications that have been implemented by using the control architecture presented above. In all the examples, the arm trajectory cannot be pre-computed off-line to accomplish the goal, but it must be continuously replanned based on the current sensory information. As a consequence, these applications require a predictable real-time support to guarantee a stable behavior of the robot and meet the specification requirements.

11.3.1.1 Assembly: Peg-in-Hole Insertion

Robot assembly is an active area of research since several years. Assembly tasks include inserting electronic components on circuit boards, placing armatures, bushings, and end housings on motors, pressing bearings on shafts, and inserting valves in cylinders.

Theoretical investigations of assembly have focused on the typical problem of inserting a peg into a hole, whose direction is known with some degree of uncertainty. This task is common to many assembly operations and requires the robot to be actively compliant during the insertion, as well as to be highly responsive to force changes, in order to continuously correct its motion and adapt to the hole constraints.

The peg-in-hole insertion task has typically been performed by using a hybrid position/force control scheme [Cut85, Whi85, AS88]. According to this method, the robot is controlled in position along the direction of the hole, whereas it is controlled in force along the other directions to reduce the reaction forces caused by the contact. Both position and force servo loops must be executed periodically at a proper frequency to ensure stability. If the force loop is closed around the position loop, as it usually happens, then the position loop frequency must be about an order of magnitude higher to avoid dynamics interference between the two controllers.

11.3.1.2 Surface Cleaning

Cleaning a flat and delicate surface, such as a window glass, implies large arm movements that must be controlled to keep the robot end-effector (such as a brush) within a plane parallel to the surface to be cleaned. In particular, to efficiently perform this task, the robot end-effector must be pressed against the glass with a desired constant force. Because of the high rigidity of the glass, a small misalignment of the robot with respect to the surface orientation could cause the

arm to exert large forces in some points of the glass surface or loose the contact in some other parts.

Since small misalignments are always possible in real working conditions, the robot is usually equipped with a force sensing device and is controlled in real time to exert a constant force on the glass surface. Moreover, the end-effector orientation must be continuously adjusted to be parallel to the glass plane.

The tasks for controlling the end-effector orientation, exerting a constant force on the surface, and controlling the position of the arm on the glass must proceed in parallel and must be coordinated by a global planner, according to the specified goal.

11.3.1.3 Object Tactile Exploration

When working in unknown environments, object exploration and recognition are essential capabilities for carrying out autonomous operations. If vision does not provide enough information or cannot be used because of insufficient light conditions, tactile and force sensors can be effectively employed to extract local geometric features from the explored objects, such as shape, contour, holes, edges, or protruding regions.

Like the other tasks described above, tactile exploration requires the robot to conform to a give geometry. More explicitly, the robot should be compliant in the direction normal to the object surface, so that unexpected variations in the contour do not produce large changes in the force that the robot applies against the object. In the directions parallel to the surface, however, the robot needs to maintain a desired trajectory and should therefore be position-controlled.

Strict time constraints for this task are necessary to guarantee robot stability during exploration. For example, periods of servo loops can be derived as a function of the robot speed, maximum applied forces, and rigidity coefficients, as we have shown in the example described in Sect. 11.2.2. Other issues involved in robot tactile exploration are discussed in [DB87, Baj88].

11.3.1.4 Catching Moving Objects

Catching a moving object with one hand is one of the most difficult tasks for humans, as well as for robot systems. In order to perform this task, several capabilities are required, such as smart sensing, visual tracking, motion prediction, trajectory planning, and fine sensory-motor coordination. If the moving target is an intelligent being, like a fast insect or a little mouse, the problem becomes more difficult to solve, since the *prey* may unexpectedly modify its trajectory, velocity, and acceleration. In this situation, sensing, planning, and control must be performed in real time—that is, while the target is moving—so that the trajectory of the arm can be modified in time to catch the prey.

Strict time constraints for the tasks described above derive from the maximum velocity and acceleration assumed for the moving object. An implementation of this task, using a six degrees of freedom robot manipulator and a vision system, is described in [BAF94].

11.4 A Robot Control Example

In order to illustrate a concrete real-time application, we show an implementation of a robot control system capable of exploring unknown objects by integrating visual and tactile information. To perform this task, the robot has to exert desired forces on the object surface and follow its contour by means of visual feedback. Such a robot system has been realized using a Puma 560 robot arm equipped with a wrist force/torque sensor and a CCD camera. The software control architecture is organized as two servo loops, as shown in Fig. 11.11, where processes are indicated by circles and CABs by rectangles. The inner loop is dedicated to image acquisition, force reading, and robot control, whereas the outer loop performs scene analysis and surface reconstruction. The application software consists of four processes:

- A sensory acquisition process periodically reads the force/torque sensor and puts data in a CAB named *force*. This process must have guaranteed execution time, since a missed deadline could cause an unstable behavior of the robot system. Hence, it is created as a hard task with a period of 20 ms.
- A visual process periodically reads the image memory filled by the camera frame grabber and computes the next exploring direction based on a user-defined strategy. Data are put in a CAB named *path*. This is a hard task with a period of 80 ms. A missed deadline for this task could cause the robot to follow a wrong direction on the object surface.
- Based on the contact condition given by the force/torque data and on the exploring direction suggested by the vision system, a robot control process computes the Cartesian set points for the Puma controller. A hybrid position/force control scheme [Whi85, KB86] is used to move the robot end-effector along a direction tangential to the object surface and to apply forces normal to the surface. The control process is a periodic hard task with a period of 28 ms (this rate is imposed by the communication protocol used by the robot controller). Missing a deadline for this task could cause the robot to react too late and exert too large forces on the explored surface that could break the object or the robot itself.
- A representation task reconstructs the object surface based on the current force/torque data and on the exploring direction. Since this is a graphics activity that does not affect robot motion, the representation process is created as a soft task with a period of 60 ms.

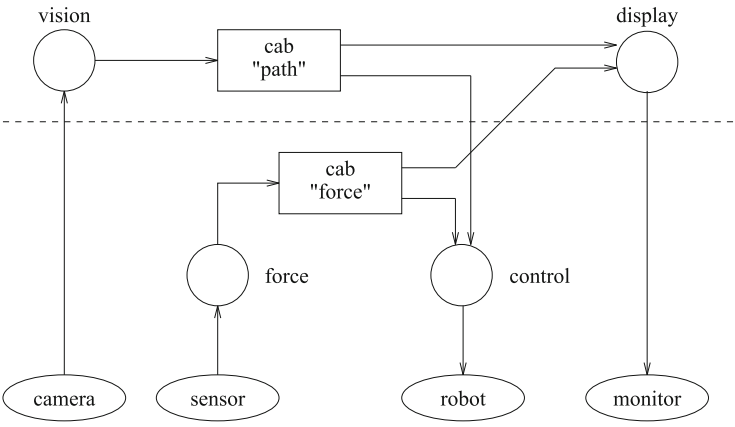


Fig. 11.11 Process structure for the surface exploration example

To better illustrate the application, we show the source code of the tasks. It is written in C language and includes the DICK kernel primitives described in Chap. 10.

```
/*-------------------------------------------------------------*/
/* Global constants                                            */
/*-------------------------------------------------------------*/

#include "dick.h"              /* DICK header file             */

#define TICK       1.0         /* system tick (1 ms)           */

#define T1         20.0        /* period for force   (20 ms) */
#define T2         80.0        /* period for vision  (80 ms) */
#define T3         28.0        /* period for control (28 ms) */
#define T4         60.0        /* period for display (60 ms) */

#define WCET1      0.300       /* exec-time for force   (ms) */
#define WCET2      4.780       /* exec-time for vision  (ms) */
#define WCET3      1.183       /* exec-time for control (ms) */
#define WCET4      2.230       /* exec-time for display (ms) */
```

```
/*-------------------------------------------------------------*/
/* Global variables                                            */
/*-------------------------------------------------------------*/
cab     fdata;                 /* CAB for force data           */
cab     angle;                 /* CAB for path angles          */

proc    force;                 /* force sensor acquisition     */
proc    vision;                /* camera acq.~and processing   */
proc    control;               /* robot control process        */
proc    display;               /* robot trajectory display     */
```

```
/*-------------------------------------------------------*/
/* main -- initializes the system and creates all tasks  */
/*-------------------------------------------------------*/

proc    main()
{
        ini_system(TICK);

        fdata = open_cab("force", 3*sizeof(float), 3);
        angle = open_cab("path", sizeof(float), 3);

        create(force,   HARD, PERIODIC, T1, WCET1);
        create(vision,  HARD, PERIODIC, T2, WCET2);
        create(control, HARD, PERIODIC, T3, WCET3);
        create(display, SOFT, PERIODIC, T4, WCET4);

        activate_all();
        while (sys_clock() < LIFETIME) /* do nothing */;
        end_system();
}
```

```
/*-------------------------------------------------------*/
/* force -- reads the force sensor and puts data in a cab */
/*-------------------------------------------------------*/

proc    force()
{
float   *fvect;                     /* pointer to cab data */

        while (1) {
              fvect = reserve(fdata);
              read_force_sensor(fvect);
              putmes(fvect, fdata);
              end_cycle();
        }
}
```

```
/*-------------------------------------------------------*/
/* control -- gets data and sends robot set points       */
/*-------------------------------------------------------*/

proc    control()
{
float   *fvect, *alfa;              /* pointers to cab data */
float   x[6];                       /* robot set-points     */

        while (1) {
              fvect = getmes(fdata);
              alfa = getmes(angle);
              control_law(fvect, alfa, x);
              send_robot(x);
              unget(fvect, fdata);
              unget(alfa, angle);
              end_cycle();
        }
}
```

```
/*-------------------------------------------------------*/
/* vision -- gets the image and computes the path angle  */
/*-------------------------------------------------------*/

proc    vision()
{
char    image[256][256];
float   *alfa;                       /* pointer to cab data */

        while (1) {
              get_frame(image);
              alfa = reserve(angle);
              *alfa = compute_angle(image);
              putmes(alfa, angle);
              end_cycle();
        }
}
```

```
/*-------------------------------------------------------*/
/* display -- represents robot trajectory on the screen  */
/*-------------------------------------------------------*/

proc    display()
{
float   *fvect, *alfa;          /* pointers to cab data    */
float   point[3];               /* 3D point on the surface */

        while (1) {
              fvect = getmes(fdata);
              alfa = getmes(angle);
              surface(fvect, *alfa, point);
              draw_pixel(point);
              unget(fvect, fdata);
              unget(alfa, angle);
              end_cycle();
        }
}
```

11.5 AI-Based Control Systems

The impressive performance achieved by artificial intelligence (AI) and deep neural networks (DNNs) in several application domains makes them very attractive to be integrated in robot systems to solve complex perception and control tasks. However, most of the AI methodologies have not been designed to work in safety-critical environments, and several issues need to be solved, at different architecture levels, to make them trustworthy. For this reason, this section presents some considerations and guidelines that could help improve the safety, security, and predictability of future AI-powered autonomous systems.

11.5.1 Predictability Issues

Machine learning algorithms are commonly developed, trained, and inferred by means of state-of-the-art frameworks (e.g., Tensorflow and PyTorch), which greatly simplify the implementation of new models. Unfortunately, however, none of the current frameworks is specifically optimized to be used in safety-critical environments, nor capable of providing bounded response times. This prevents their use in real-time applications like autonomous driving, where DNNs should have a highly predictable behavior, not only in the functional domain but also in the time domain, responding within predefined deadlines. This problem is exacerbated by the fact that such frameworks require a rich operating system that supports the acquisition

of complex input devices, as cameras and LiDARs, and are often executed under Linux, since it provides such a support for free.

Predictability issues are also introduced by the computing platforms normally used to accelerate the inference of machine learning models. To be used in real time, the inference of modern DNN models is typically performed on a general purpose graphics processing unit (GPU) or by exploiting programmable hardware, as a field programmable gate array (FPGA). Both devices, although greatly reducing the average inference times, can introduce large and variable latencies that are difficult to bound in general [But22].

11.5.1.1 GPU Issues

On the positive side, using a GPU to perform a DNN inference has two main advantages: (i) the average response time can be reduced by two orders of magnitude, and (ii) the development is supported by standard frameworks. On the negative side, however, GPUs are closed systems, and multiple tasks are scheduled in a non-preemptive fashion. This means that, if the system includes multiple neural networks with different complexity and periodicity requirements, DNNs with shorter periods will likely be delayed by those having lower priority, possible causing deadline misses.

Another negative behavior of GPU was reported by Cavicchioli et al. [CCB17], who observed significant and variable delays during GPU acceleration on heterogeneous embedded platforms due to the contention occurring on shared memory, especially for memory-intensive GPU tasks.

To solve this problem, Capodieci et al. [CCBP18], in collaboration with NVIDIA, proposed to modify the GPU internal scheduler with a preemptive scheduler based on Earliest Deadline First (EDF) [LL73], also providing bandwidth isolation by means of a Constant Bandwidth Server (CBS) [AB04]. Unfortunately, however, this solution is not yet available on commercial NVIDIA GPU platforms.

Other problems with GPU acceleration are due to the high power consumption and their significant weight and encumbrance, which prevent their usage in small embedded systems, as unmanned aerial vehicles (UAVs).

11.5.1.2 FPGA Issues

An alternative to GPUs for accelerating AI algorithms is provided by FPGAs. They are integrated circuits designed to be configured after manufacturing for implementing arbitrary logic functions in hardware. As such, they exhibit a highly predictable behavior in terms of execution times. In addition, they consume much less power with respect to GPUs, and existing commercial platforms are characterized by lower weight, encumbrance, and cost. Hence, they represent an ideal solution for being used on battery-operated embedded systems with size, weight, power, and cost (SWaP-C) constraints, as space robots, satellites, and UAVs.

Nevertheless, Restuccia et al. [RPB+19] identified some anomalous situations that can arise in an AXI bus arbiter in FPGA-based SoC and proposed a reservation mechanism to prevent this phenomenon and restore fairness during bus transactions. A timing analysis has also been proposed [RBMB20] to bound the execution of periodically invoked hardware accelerators in nominal conditions. This analysis can be used to configure a latency-free hardware module named AXI Stall Monitor (ASM) to detect and safely solve possible stalls during AXI bus transactions. Efforts have also been devoted to analytically bound the delay experienced by AXI bus transactions issued by hardware accelerators on FPGA [RPB+20].

Hardware acceleration typically involves memory-intensive computations; therefore, an accurate control of the memory traffic is crucial to achieve predictability in the execution of HW-tasks. Pagani et al. [PRB+19] proposed a bandwidth reservation mechanism for AXI-based transactions on FPGAs able to control the bus traffic generated by hardware accelerators. The mechanism, named Memory Budget and Protection Unit (MBPU), aims at "shielding" hardware accelerators from excessive or unpredictable memory interference. MBPUs are installed between AXI master ports and the interconnect, enforcing a given budget of memory transactions within a periodic interval of time. Budgets are recharged in a periodic fashion and are configurable from the CPU via memory-mapped registers. MBPUs also protect the system from unrestricted accesses to memory by HW-tasks: this is accomplished by masking the accesses that fall outside a set of configurable memory address spaces.

When used for DNN acceleration, FPGAs have other problems:

- No floating point unit (FPU) is available on the chip, unless it is explicitly programmed by the user, but consuming a significant fraction of the available fabric.
- Programming FPGAs is quite more difficult than programming CPUs or GPUs, and efficient coding requires a deep knowledge of low-level architecture details.
- The frameworks available today for developing AI applications on FPGA-based platforms are less rich and flexible than those available for GPUs, and the same is true for related libraries and tools.
- The overall FPGA area available in medium size SoCs could be insufficient to host more than one DNN or even a single large DNN.

To overcome the problems outlined above, a lot of research has been carried out in the recent years.

The absence of an FPU is overcome by performing a preliminary parameter quantization to convert floating point numbers into integers with n-bit precision. Several quantization methods have been proposed in the literature [Guo18], including symmetrical, asymmetrical, non-uniform, and statistical. An extreme quantization converts weights into binary numbers using the sign function. Courbariaux, Bengio, and David [CBD15] have shown that a binarized DNN can achieve 98.8% accuracy in classifying the handwritten digits of the MNIST dataset. Other optimization steps (e.g., network pruning and layer fusion) can also be performed, both on GPUs and

FPGAs, to reduce the computation time and the memory footprint of trained DNNs while minimizing the loss in accuracy.

To overcome the limitation on the FPGA area, Biondi et al. [BBP+16] proposed a programming framework, called FRED,[1] to support the design, development, and execution of predictable software on FPGAs. FRED exploits dynamic partial reconfiguration and recurrent execution to virtualize the FPGA area, thus enabling the user to allocate a larger number of hardware accelerators than those that could otherwise be fit into the physical fabric. FRED also integrates a tool for automated floorplanning [SBB19] and a set of runtime mechanisms to enhance predictability by scheduling hardware resources and regulating bus/memory contentions [RPB+19].

The FPGA virtualization is achieved through a timesharing mechanism that replaces inactive accelerators (i.e., those that finished their computation and are waiting for the next activation) with active ones. In this way, the total number of HW-tasks that can run on the FPGA can be much higher than the number of HW-tasks that would statically fit in the physical area available on the fabric. Hence, this mechanism virtualizes the FPGA by creating a virtual area much larger than the physical one. The resulting approach is similar to multitasking, where tasks continuously change their context, or a virtual memory mechanism, where memory pages are swapped between hard disk and dynamic memory. Thanks to a set of design choices and a proper scheduling infrastructure, resource contention delays experienced by tasks running under FRED are bounded and predictable, and hence they can be estimated to verify the system schedulability.

A full support for FRED has been developed under both FreeRTOS [PBB+17] and under Linux [PBM+ar]. The Linux support comes with a user-space daemon and a set of custom kernel drivers to handle the processor configuration port (PCAP) and the shared-memory communication buffers between CPUs and FPGA. A preemptable reconfiguration interface has also been developed by Rossi et al. [RDB+18] to achieve a finer control in scheduling the reconfiguration requests and a better control on the reconfiguration delays incurred by HW-tasks.

Finally, Restuccia and Biondi [RB21] proposed a set of techniques for accelerating DNNs on FPGA-based platforms with a highly predictable timing behavior under the Vitis AI frameworks by Xilinx. In Vitis AI, the execution of the DNN layers relies on the deep learning processing unit (DPU) core, a hardware accelerator optimized for the execution of convolutional DNNs. Based on an extensive profiling campaign conducted on the Xilinx Zynq Ultrascale+ platform, they proposed an execution model for the DPU employed to derive a response time analysis for guaranteeing real-time applications constraints.

[1] See details on http://fred.santannapisa.it.

11.5.2 Mixed Criticality Issues

Most robot applications that make use of AI algorithms consist of several components with different complexity and requirements. Consider, for example, a self-driving car. The functions responsible for steering, throttle modulation, braking, and engine control are highly critical and must satisfy stringent requirements in terms of safety, security, and real-time behavior. As such, they need to be managed by a real-time operating system that must be certified to guarantee the required safety integrity levels. On the other hand, high-level functions related to sensory perception, object tracking, and vehicle localization heavily rely on AI algorithms and need to be executed on a rich operating system (e.g., Linux) to exploit all the available device drivers, libraries, and development frameworks required for such complex computations. These software components are far from being certified and offer a large software surface for cyberattacks.

In 2015, two hackers, Charlie Miller and Chris Valasek, discovered a vulnerability in the Jeep Cherokee, which they exploited to remotely access the vehicle and gain physical control, including steering, braking, turning on the wipers, blasting the radio, and, finally, killing the engine to bring the vehicle to a complete stop [Erw21]. They wrote a long paper [MV15] where they explain how they accessed the CAN bus through the infotainment system, detailing the full attack chain.

A possible solution to manage multiple software components with different criticality requirements while isolating them from a reciprocal interference and cyberattacks would be to execute each component on a different hardware platform, under the most appropriate operating system. However, for several embedded applications, this approach is not efficient in terms of space, weight, power, and cost (SWaPC). For this reasons, a more appropriate solution is to host the execution of software components with mixed criticality requirements on the same hardware platform, isolating them into separate execution domains (i.e., virtual machines) through a hypervisor. Besides guaranteeing a spatial and temporal isolation between components, the hypervisor allows adopting different operating systems in different domains while providing security mechanisms for exchanging data between components.

Cittadini et al. [CMB+ar] proposed the architecture illustrated in Fig. 11.12 to manage two execution domains of different criticality on a drone that has to follow moving objects by a camera using a neural network for detection and tracking. The inference of the DNN model is executed in a general purpose domain under the Linux operating systems, whereas all low-level control tasks related to the drone are executed in a real-time domain under FreeRTOS.

The two domains are managed by the CLARE hypervisor,[2] which also provides a secure real-time communication channel for exchanging data between the two

[2] https://accelerat.eu/clare.

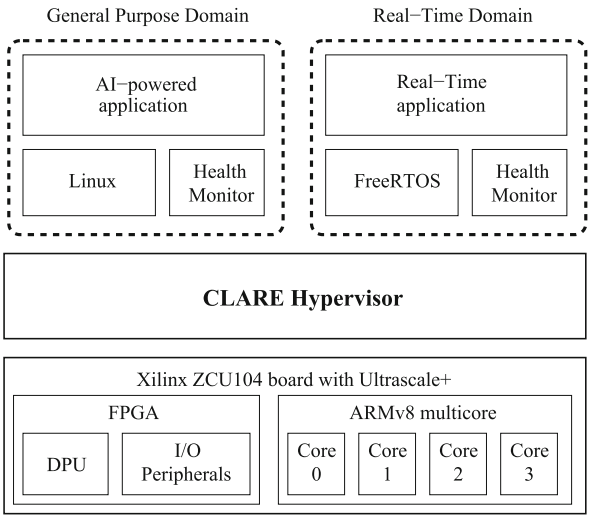


Fig. 11.12 Example of hypervisor architecture for managing two execution domains of different criticality

domains and multi-domain virtualization of the FPGA area, thus enabling strong isolation also for programmable logic components, such as hardware accelerators.

11.5.3 Safety and Security Issues

Current DNN models have been proved to suffer from serious safety and security issues that prevent them to be used in safety-critical systems, unless adopting specific countermeasures.

For instance, it has been shown that properly crafted imperceptible perturbations added to an image can fool a neural network in perceiving objects that are not present in the scene [SZS+14, Sim18, BR18]. Such perturbed inputs are called *adversarial examples* and represent a serious threat for the security of AI-based systems.

Adversarial attacks have also been extended to other types of data, as audio samples and text, and can even be performed in the real world through properly crafted patches that can be printed and placed on physical objects, without accessing the vision system. For example, an adversarial patch attached to a stop sign can make a DNN to misclassify it as a parking sign, threating the security of self-driving vehicles.

In addition to malicious attacks, the prediction of a neural model can also be compromised by natural inputs that are out of the typical distribution of the data samples used during training. For example, if a DNN has been trained to classify traffic signs taken from different views and lighting conditions, it is difficult to know in advance how the model will classify a sign that is partially covered by snow, if this type of situation was not present in the training set.

For the reasons mentioned above, effective methods for detecting or neutralizing adversarial attacks or out-of-distribution inputs will play a crucial role in the development of safety-critical systems that integrate machine learning models in their perception and control modules.

One method for detecting adversarial images relies on the fact that DNN models are usually robust to certain types of input transformations (e.g., translation, rotation, scaling, blurring, noise addition, etc.). This means that, if a genuine image is correctly recognized by a DNN, the prediction score does not reduce significantly when the same image is translated, rotated, or modified with the mentioned transformations. However, the same is not true for most adversarial examples, and it has been observed that they result to be more sensitive to input transformations, which cause a much higher degradation in the prediction score.

This property has been exploited by some authors [TYC18] to detect whether an input x is adversarial or genuine. If $y = f(x)$ is the top class score produced by the DNN on input x and $y_T = f(T(x))$ is the score produced on the transformed input $T(x)$, a simple detection method is to consider x to be adversarial if the difference $y - y_T$ is higher than a given threshold τ.

Unfortunately, it is possible to generate adversarial examples that are robust to input transformations. To cope with this case, Nesti et al. [NBB21] proposed a new method, called *defense perturbation*, capable of detecting adversarial images that are robust to input transformations. The defense perturbation is generated by a proper optimization process capable of making robust adversarial images sensitive again to input transformations. Furthermore, the paper also introduces multi-network adversarial examples that can fool multiple DNN models simultaneously, presenting a solution for detecting them.

A different approach for detecting adversarial attacks is based on a deeper analysis of the neuron activation values in the different DNN layers. In fact, in order to force a DNN model to classify an input with a desired wrong class, adversarial examples usually cause an overactivation of some neurons in different network layers. To identify such neurons, Rossolini et al. [RBB21] presented a new coverage analysis methodology capable of detecting both adversarial and out-of-distribution inputs. The approach works in two distinct phases: in a preliminary (off-line) phase, a trusted dataset is presented to the DNN, and the neuron outputs, in each layer and for each class, are analyzed and aggregated into a set of covered states, which all together represent a sort of *signature* describing how the model responds to the trusted samples for each given class. Then, at runtime, each new input is subject to an evaluation phase, in which the activation state produced by the input in each layer is compared with the corresponding signature for the class predicted by the network. The higher the number of activation values outside the range observed during the presentation of the trusted dataset, the higher the probability that the current input is not trustworthy.

Another effective method to improve the adversarial robustness of convolutional networks against physically realizable adversarial attacks for both semantic segmentation and object detection tasks has been proposed by Rossolini et al. [RNB+22]. Such a defense relies on specific Z-score analysis performed on the internal network

features to detect and mask the pixels corresponding to adversarial objects in the input image. Spatially contiguous activations are examined in shallow and deep layers to suggest potential adversarial regions. Such proposals are then aggregated through a multi- thresholding mechanism.

To exploit deep learning technology in safety-critical applications, it is crucial to integrate such detection and defense methods into the system architecture, so that if a malicious or unsafe input is detected, the system can ignore the output produced by the DNN model and switch to a simpler but safer backup module that can bring the system into a safe state. For instance, Biondi et al. [BNC+20] proposed a software architecture that allows embracing deep learning while guaranteeing safety, security, and predictability by integrating diverse technologies, as hypervisors, runtime monitoring, redundancy with diversity, predictive fault detection, fault recovery, and predictable resource management.

Chapter 12
Implementing Periodic Tasks in Linux

When learning real-time programming, one of the major difficulties is to face low-level details related to both the computing platform and the operating system, which require considerable programming effort, even for implementing a simple periodic task. This is because most operating systems do not provide any support for programming periodic tasks and managing deadlines. For example, the POSIX real-time standard only provides a low-level notion of thread; thus programmers need to develop additional code on top of the POSIX API to expose a higher-level interface easier to understand and manage.

This chapter presents how to develop a simple C library, named *Ptask* [BL13], that simplifies the implementation of (soft) real-time periodic tasks in Linux by hiding many of the low-level details that are visible when programming at the operating system level. The library is built on top of the POSIX thread functions (*Pthread* library[1]) and allows the user to create periodic tasks with given periods and deadlines, select the scheduler, manage the system time in a more practical way, and check for deadline misses. The Ptask library is adopted since many years to teach real-time programming in university courses and develop real-time control applications in simulated environments on a PC under Linux.

The rest of this chapter is organized as follows. Section 12.1 recalls some Linux internal features and the Pthread library. Section 12.2 illustrates the design choices made for the Ptask library and presents the interface exposed to the user. Section 12.3 explains how to implement the Ptask functions, showing how to map the proposed interface into the Pthread functions. Finally, Sect. 12.4 illustrates some examples of real-time tasks implemented with the Ptask library.

[1] https://en.wikipedia.org/wiki/Pthreads.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_12

12.1 Linux and the Pthread Library

This section briefly recalls the main features of the Linux operating system and the Pthread library upon which the Ptask functions are built. In particular, we will recall the way time is represented and managed in Linux, the available schedulers, the main Pthread functions for managing concurrent threads, and the available mutual exclusion and synchronization mechanisms.

Time Management
POSIX represents time using the `struct timespec` data type:

```
struct timespec {
    time_t  tv_sec;    // seconds
    long    tv_nsec;   // nanoseconds
};
```

However, it does not provide any function for manipulating such a data structure. Time is managed by two types of clocks:

- `CLOCK_REALTIME`: it maintains a value that is as close as possible to the absolute time. However, it may be discontinuous, because it can be adjusted by the system and by the user.
- `CLOCK_MONOTONIC`: it represents the elapsed time from an unspecified initial instant. It is not affected by adjustments; hence, it is the best solution for measuring the time elapsed between two events.

The current system time can be read by using the following function:

```
int clock_gettime(clockid_t clk_id, struct timespec *t);
```

which stores in `t` the value of the clock specified by `clk_id`. The task suspension at the end of the period can be implemented by the following function:

```
int  clock_nanosleep(clockid_t clock_id, int flags,
            const struct timespec *request,
            struct timespec *remain);
```

which suspends the execution of the calling thread until clock `clk_id` reaches the time specified by `t`. If `flag` is equal to zero, the time `t` is interpreted as relative to the current time; if `flag` is equal to `TIMER_ABSTIME`, the time `t` is interpreted as an absolute value. If the thread is awakened before the set time, the remaining time is stored in `rem`.

Linux Schedulers
The Linux kernel provides a flexible scheduling architecture that supports different *scheduling modules* managed as a prioritized hierarchy. When the kernel needs to select a task to be executed on a processor, it looks at each scheduling module in a priority order, until it finds a task to be executed. The official kernel includes at least two scheduling modules: the Completely Fair Scheduler (CFS) and the real-time scheduler, which has priority over CFS.

The CFS is a non-real-time, best effort fair scheduler that can be selected by specifying the constant SCHED_OTHER when creating processes and threads. It is the standard Linux time-sharing scheduler that is intended for all processes that do not require real-time service. It uses a peculiar priority aging mechanism to ensure fairness among tasks. Priorities are called *nice values* and range from 20 (very nice, favoring other threads) to −19 (no nice, less prone to leave the processor to other tasks).

The real-time scheduling module provides two scheduling policies, compliant with the POSIX RT standard: SCHED_FIFO and SCHED_RR. Both are priority-based schedulers with 99 priority levels, where level 1 denotes the lowest priority and level 99 the highest priority (note, however, that the POSIX standard requires to ensure only 32 levels). Each priority is associated with a queue, in which all threads with the same priority are enqueued. The thread at the head of the queue with the highest priority level is selected as the running task. The difference between the two schedulers is that in SCHED_FIFO tasks with the same priority are managed using a first-come first-served policy, while in SCHED_RR, tasks with the same priority are managed using a round robin policy.

SCHED_FIFO and SCHED_RR are referred to as real-time policies and can be used only from the superuser (root). A program using them should be compiled with the -lrt option. For example, a Rate Monotonic preemptive scheduler [LL73] can easily be implemented by assigning each periodic task a priority inversely proportional to its period. Similarly, a Deadline Monotonic preemptive scheduler [LW82] can be implemented by assigning each periodic task a priority inversely proportional to its relative deadline.

Recently, a new scheduling module, namely, SCHED_DEADLINE [LLFC11], has been proposed as a patch to the Linux kernel to provide the Earliest Deadline First (EDF) algorithm [LL73] along with a Constant Bandwidth Server (CBS) [AB98, AB04] to support resource reservation. The SCHED_DEADLINE module is supposed to run at the highest priority level in the sequence of scheduling modules.

Thread Management

A Linux process can include several concurrent threads, which can share the same process memory space, can execute the same code, but have distinct stack. To use the Pthread library, the program must include the header file pthread.h and must be compiled with the -lpthread option. A thread can be created by calling the following function:

```
int  pthread_create(thread_t *id, pthread_attr_t *attr,
                    void *(*body)(void *), void *arg);
```

which creates a thread and returns 0 on success, or an error code, otherwise. The function requires four arguments:

- id contains the identifier of the created thread (assigned by the function);
- attr is a pointer to a structure that contains the thread attributes (if NULL, default values are used);
- body is a pointer to the function defining the thread code;

```
pthread_attr_t     myatt;    // attribute structure
struct sched_param mypar;    // priority structure
pthread_t          tid;      // thread id

   pthread_attr_init(&myatt);

   pthread_attr_setinheritsched(&myatt, PTHREAD_EXPLICIT_SCHED);
   pthread_attr_setschedpolicy(&myatt, SCHED_FIFO);

   mypar.sched_priority = 23;
   pthread_attr_setschedparam(&myatt, &mypar);

   pthread_create(&tid, &myatt, task, NULL);
   pthread_join(tid, NULL);
   pthread_attr_destroy(&myatt);
```

Fig. 12.1 Example of code setting some thread attributes

- `arg` is a pointer to a single argument passed of the thread. More arguments can be passed using a pointer to a structure.

  The `attr` structure defines the following thread attributes:

- *State*. It can be *joinable*, if the thread termination is an event that can be waited for by some other thread, or *detached*, if this is not the case (the default value for the state is *joinable*).
- *Stack size*. It specifies the dimension of the stack (the default value is 8 MB).
- *Scheduler*. It specifies the scheduling algorithm to be used for scheduling the thread (the default value is SCHED_OTHER).
- *Priority*. It specifies the nice value (in $[-19, 20]$) to be used for `SCHED_OTHER` or the thread priority level (in [1, 99]) to be used for the real-time policies. The default value is set to 0.

Figure 12.1 illustrates an example of code that shows how to create a thread, named `task`, with priority level equal to 23, associated with the `SCHED_FIFO` scheduling policy. The thread state and stack size are left with the default values initialized by the `pthread_attr_init()` function. In the example, no parameter is passed to the created thread, since the last argument in the `pthread_create()` is `NULL`.

Figure 12.2 illustrates another example in which a thread named `task` is created with default attributes and receives a parameter as argument, whereas Fig. 12.3 shows how the thread can retrieve the parameter passed by the `pthread_create` function. Note that, since the passed argument can have any type, the passage has to be done by casting its type to a pointer to void. Similarly, the thread function has to perform the inverse casting to retrieve its value.

A thread can terminate for different causes:

```
pthread_t  tid;      // thread id
int        err;      // error code for pthread_create
int        a = 3;    // value to be passed to the thread

    err = pthread_create(&tid, NULL, task, (void *)&a);

    if (err == 0)
           printf("Thread successfully created\n");
    else   printf("ERROR: Thread not created\n");
```

Fig. 12.2 Example of code that creates a thread passing a parameter as argument

```
void  *task(void *p);
{
int   *pa;     // pointer to task argument
int   a;       // task argument

      pa = (int *)p;    // convert pointer
      a = *pa;          // retrieve argument

      printf("Retrieved value = %d\n", a);
      return NULL;
}
```

Fig. 12.3 Example of a thread that retrieves the passed parameter

1. When it executes the last instruction of the associated function (this is referred to as normal termination);
2. When it calls the `pthread_exit` function;
3. When another thread executes `pthread_cancel`.
4. When its father process (e.g., the `main()` function) terminates for a normal termination or for a call to the `exit()` function.

Finally, it is possible to wait for the termination of a thread through the function `pthread_join`

```
int  pthread_join(pthread_t th, void **retval);
```

which suspends the execution of the calling thread until thread `th` terminates its execution. If `*retval` is not equal to `NULL`, the return value of the terminated thread is copied into `*retval`. If the thread is already terminated, `*retval` gets the value `PTHREAD_CANCELED`. It returns 0 in case of success or an error code otherwise.

Synchronization and Mutual Exclusion

Linux POSIX semaphores can be used for implementing mutual exclusion and synchronization among threads. To use them, the header file `semaphore.h` has

```
int   x, y;                    // these are global variables
sem_t smux, ssyn;

      sem_init(&smux, 0, 1);   // this is a mutex sem
      sem_init(&ssyn, 0, 0);   // this is a sync sem

      sem_wait(&smux);         // this is a critical section
      x = 5;
      y = 8;
      sem_post(&smux);

      sem_wait(&ssyn);         // it waits for an event
```

Fig. 12.4 An example of usage of POSIX semaphores

to be included in the program. A semaphore must be declared as a variable of type **sem_t** and initialized using the following function:

```
int sem_init(sem_t *sem, int pshared, unsigned int v);
```

where `sem` is a pointer to the semaphore. The value `pshared` specifies, when equal to zero, that `sem` is shared among threads (hence it must be declared as global); if different than 0, it specifies that `sem` is shared among processes. When a single process is used, this value can be set to 0. The variable `v` sets the initial value of the semaphore: it must be equal to 0 if `sem` is used for synchronization, equal to 1 if `sem` is used to mutually exclude the access to single-unit resources, and greater than 1 if `sem` is used for mutual exclusion in multi-unit resources. Figure 12.4 illustrates a simple example that shows how to use POSIX semaphores.

The Pthread library provides another type of semaphores specific for mutual exclusion (they cannot be used for synchronization). These semaphores must be declared as variables of type **pthread_mutex_t** and have an attribute structure that is called **pthread_mutexattr_t**:

```
pthread_mutex_t      mux;    // define a mutex
pthread_mutexattr_t  matt;   // define its attributes
```

Mutex semaphores can be initialized in three different ways, shown in Fig. 12.5.

The first way is to initialize a mutex when it is declared, by setting its value equal to a predefined constant. The second way is to call the `pthread_mutex_init()` passing the address of the semaphore and the pointer to its attribute structure (a NULL pointer indicates that defaults values should be used). The third method is used when we need to initialize a mutex with special attributes. In this case, we first call `pthread_mutexattr_init()` to initialize the attributes with the default values, and then we modify the attributes and finally call `pthread_mutex_init()` passing the address of the mutex and the address of

```
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER;  // way 1
pthread_mutex_init(&mux, NULL);    // way 2 (default values)
pthread_mutexattr_init(&matt);     // way 3 (special values)
<modify the mutex attributes>
pthread_mutex_init(&mux, &matt);
```

Fig. 12.5 Three examples of how to initialize mutex semaphores

```
int   x, y;       // these are global variables
pthread_mutex_t mux = PTHREAD_MUTEX_INITIALIZER;

      pthread_mutex_lock(&mux);  // critical section
      x = 5;
      y = 8;
      pthread_mutex_unlock(&mux);
```

Fig. 12.6 An example of usage of mutex semaphores

the modified attribute structure. The functions to protect a critical section are called `pthread_mutex_lock()` and `pthread_mutex_unlock()`. Figure 12.6 illustrates a simple example that shows how to use mutex semaphores.

The main advantage of using Pthread mutexes with respect to POSIX semaphores is that they support two protocols for preventing priority inversion, which are described in the next section.

Resource Access Protocols

Pthread mutex semaphores support two resource access protocols for preventing priority inversion: Priority Inheritance [SRL90] and Higher Locker Priority, both explained in Chap. 7. They can be set using

```
pthread_mutexattr_setprotocol(&myatt, protocol),
```

where `protocol` can have one of the following values:

- `PTHREAD_PRIO_NONE`. This is the classical mutual exclusion mechanism that uses binary semaphores for accessing critical sections. It suffers from priority inversion phenomena that may block the execution of high-priority tasks for long time.
- `PTHREAD_PRIO_INHERIT`. This enables the Priority Inheritance Protocol [SRL90], which prevents priority inversion by increasing the priority of a task holding a resource to the maximum priority among those tasks blocked on the same resource.
- `PTHREAD_PRIO_PROTECT`. This enables the Highest Locker Priority protocol (or Immediate Priority Ceiling), according to which each resource R_k is assigned

```
#define _GNU_SOURCE

pthread_mutex_t     mux;
pthread_mutexattr_t myatt;

  pthread_mutexattr_init(&myatt);
  pthread_mutexattr_setprotocol(&matt, PTHREAD_PRIO_PROTECT);
  pthread_mutexattr_setprioceiling(&myatt, ceiling);
  pthread_mutex_init(&mux, &myatt);
  pthread_mutexattr_destroy(&myatt);
```

Fig. 12.7 Example that shows how to enable the immediate priority ceiling protocol

a ceiling $C(R_k)$ equal to the highest priority among the tasks using R_k. Then, a task entering a critical section related to R_k executes at a priority level equal to the ceiling $C(R_k)$.

When using Immediate Priority Ceiling, a *ceiling* value must be associated with each semaphore that must be equal to the highest priority among the threads using it. Also note that when using such protocols, the `_GNU_SOURCE` constant must be defined in the file. Figure 12.7 illustrates an example that shows how to enable the Immediate Priority Ceiling protocol.

12.2 Ptask Design Choices

From the examples presented above, it is clear that creating and managing concurrent activities using the Pthread library is cumbersome and not easy to read. In addition, the library does not provide any support for manipulating time variables, creating period tasks, and managing deadlines. For this reasons, this section explains how to develop an abstraction layer (the Ptask library) that can greatly simplify the programming interface for manipulating time and deadlines and defining periodic real-time tasks in a more natural way.

One of the main objectives of the Ptask library is to support the creation of periodic real-time tasks; hence the first step is to define the model of the periodic task that should be exported to the user.

A real-time periodic task, here denoted by τ_i, is a portion of code cyclically executed several times on different data. Each execution instance, identified as a *job* and denoted by $\tau_{i,j}$ ($j = 1, 2, \ldots$), is triggered at a precise time instant, referred to as the *job activation time*. The activation times of consecutive jobs of a periodic task τ_i are exactly separated by the same interval, called the *task period*. In general, the following parameters are typically defined on a periodic task:

- *Worst-case execution time* (WCET) C_i of task τ_i: it is the longest possible duration of the task on the considered hardware platform.
- *Job activation time* $a_{i,j}$: it is the absolute time at which job $\tau_{i,j}$ becomes active (i.e., ready to execute).
- *Period* T_i: it is the separation interval between any two consecutive job activation times of task τ_i.
- *Relative deadline* D_i: it is the maximum time (relative to the job activation time) within which any job of task τ_i should complete its execution.
- *Absolute deadline* $d_{i,j}$: it is the maximum absolute time within which job $\tau_{i,j}$ should complete its execution. It is computed as

$$d_{i,j} = a_{i,j} + D_i.$$

- *Priority* P_i: it is a number specifying the relative importance of task τ_i with respect to the others and used by the scheduler to select the task to be executed among the set of active tasks ready to run.
- *Phase* Φ_i: it is the activation time of the first job of task τ_i ($\Phi_i = a_{i,1}$). Note that, all the subsequent jobs of a periodic task are activated at precise time instants given by

$$a_{i,j} = \Phi_i + (j-1)T_i.$$

Among the task parameters defined above, the ones defined by the user are the period T_i, the relative deadline D_i, and the priority P_i; hence they should be exposed to the user through the programming interface when creating a periodic task. In the presented implementation, it is assumed that the phase Φ_i is not explicitly specified by the user, since tasks can be dynamically created at any time. In addition, for the sake of simplicity, period and relative deadlines will be expressed in milliseconds and will be represented by integers variables.

In the proposed implementation, other three arguments are exposed to the user when creating a periodic task:

- *task function*: it is the name of the function that defines the periodic task. It must contain a loop, whose body represents the code executed by each job.
- *task index*: it is a integer that univocally identifies a periodic task. Note that the user could create multiple tasks sharing the same code (hence, linked to same function) executed on different data. For this reason, we need to associate a unique identifier with each created task. In the proposed implementation, the task identifier is an integer specified by the user at creation time (also called *task argument*) that can be used in the task code to differentiate the behavior of tasks linked to the same function.
- *activation flag*: it is a flag that allows the user to specify whether a newly created task should be activated immediately or suspended until an explicit activation call is executed by another task.

Hence, the function for creating a periodic task will have the following prototype:

```
int  task_create(
     void* (*task)(void *), // task function
     int    index,          // task index
     int    period,         // task period
     int    deadline,       // relative deadline
     int    priority,       // task priority
     int    act_flag);      // activation flag
```

All the task parameters will be stored in a task control block represented by the following data structure:

```
struct  task_par {
        int       arg;          // task index
        long      wcet;         // worst-case exec time
        int       period;       // task period
        int       deadline;     // relative deadline
        int       priority;     // task priority
        int       act_flag;     // activation flag
        struct    timespec at;  // next activation time
        struct    timespec dl;  // absolute deadline
        int       dmiss;        // deadline miss counter
        sem_t     asem;         // activation semaphore
        pthread   tid;          // thread identifier
}
```

Note that, among the various elements defined in the task data structure, there is a deadline miss counter (`dmiss`) to keep track of the total number of deadline misses experienced by a task, a synchronization semaphore (`asem`) (to be initialized to 0) to start the execution of a task by an explicit activation function, and a final field (`tid`) to store the thread identifier returned by the `pthread_create` associated with the task. To create a maximum number of tasks defined by `MAX_TASKS`, an array of such structures has to be defined, here named `tp`:

```
struct task_par tp[MAX_TASKS];
```

According to the proposed implementation, the code of a periodic task must contain a cycle (e.g., a `while` loop with an ending condition) whose body represents the code executed by each job. In order to start the execution upon an explicit activation (here achieved by calling `task_activate(task_index)`), a periodic task must call a synchronization function (`wait_for_activation (task_index)`) before entering the loop. If the activation flag specified in the `task_create` indicates an explicit activation, such a function should block the execution of the task until a `task_activate` is invoked. Once unblocked, the `wait_for_activation` function should read the current system time and set

```
void*  periodic_task(void *arg)
{
<local variables>
int i;

    i = get_task_index(arg);
    wait_for_activation(i);

    while (!ending_condition) {

        <task body>

        if (deadline_miss(i)) <do action>
        wait_for_period(i);
    }
}
```

Fig. 12.8 Structure of a periodic task

the next activation time one period ahead. The task absolute deadline should also be set as the current time plus the relative deadline.

Before ending the loop, a periodic task should suspend itself until the next period to respect its activation rate and allow other lower-priority tasks to execute. Such a function is named `wait_for_period(task_index)` and suspends the calling task until the next activation time previously set.

When the suspended task resumes at the beginning of the next period, this function should also update the next activation time and the absolute deadline by setting them one period ahead. If needed, a check for a possible deadline miss has to be done at the end of the cycle, before suspending the task. This can be done by calling the `deadline_miss(task_index)` function.

Finally, each task is defined with a single argument (defined as a generic pointer to a `void`), used to pass the address of the `task_par` structure corresponding to the created thread. In this way, the task can recover its index (using the function `task_get_index(arg)`) and all its parameters.

Figure 12.8 illustrates the structure of a periodic task, which makes use of the auxiliary functions defined above.

Overall, the functions that are implemented in the Ptask library to simplify the management of periodic tasks are listed Fig. 12.9. Most of these functions are just syntactic sugar to make the final code easier to read and understand. Some of them will be analyzed in detail in Sect. 12.3.

```
ptask_init(policy);
task_create(id, func, period, deadline, priority, act flag);
task_activate(id);
wait_for_activation(id);
wait_for_period(id);
wait_for_task_end(id);
deadline_miss(id);
get_task_index(arg);
task_get_period(id);
task_get_deadline(id);
task_get_atime(id, &at);
task_get_adline(id, &dl);
task_set_period(id, period);
task_set_deadline(id, deadline);
```

Fig. 12.9 List of functions of the Ptask library

12.3 Implementing Ptask Functions

This section first illustrates how to implement some function to manipulate POSIX time variables and read the current time (since the application start time) using a desired time unit. Then, it shows how to implement some of the Ptask functions listed in Fig. 12.9.

Time Management

To manage time more easily, when starting the real-time application, while executing `ptask_init`, the Ptask library saves the current system time into a global variable (`ptask_t0`) and measures all the subsequent times with respect to it. In this way, such a relative time can be represented by a long integer expressed in a desired time unit, which is much easier to manage. Ptask also provides a number of auxiliary functions to copy, compare, and add time variables in the POSIX representation.

The function shown in Fig. 12.10 copies a source time structure `ts` into a destination time structure `td`. The function shown in Fig. 12.11 compares two time structures `t1` and `t2` and returns `0` if they are equal, `1` if `t1 > t2` and `-1` if `t1 < t2`.

The function reported in Fig. 12.12 adds a value expressed in milliseconds to a given time structure.

The function illustrated in Fig. 12.13 returns the current system time (relative to the application start time `ptask_t0`) in a user-defined time unit.

```
void time_copy(struct timespec *td, struct timespec ts)
{
      td->tv_sec = ts.tv_sec;
      td->tv_nsec = ts.tv_nsec;
}
```

Fig. 12.10 Function for copying a time structure into another one

```
int  time_cmp(struct timespec t1, struct timespec t2)
{
      if (t1.tv_sec > t2.tv_sec) return 1;
      if (t1.tv_sec < t2.tv_sec) return -1;
      if (t1.tv_nsec > t2.tv_nsec) return 1;
      if (t1.tv_nsec < t2.tv_nsec) return -1;
      return 0;
}
```

Fig. 12.11 Function for comparing two time structures

```
void time_add_ms(struct timespec *t, int ms)
{
      t->tv_sec += ms/1000;
      t->tv_nsec += (ms%1000)*1000000;

      if (t->tv_nsec > 1000000000) {
          t->tv_nsec -= 1000000000;
          t->tv_sec += 1;
      }
}
```

Fig. 12.12 Function to add a value expressed in milliseconds to a given time structure

Task-Related Functions

This section presents the implementation details of some Ptask functions, leaving the most trivial ones to the reader.

First of all, a `ptask_init()` function is needed to initialize the application start time, the scheduler used for all the tasks, and the synchronization semaphores used for task activations. This function receives a single integer argument specifying the scheduling policy that should be used for all the tasks, among those provided by Linux (`SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, etc.).

```
long get_systime(int unit)
{
struct timespec     t;
long tu, mul, div;

      switch (unit) {
          case MICRO: mul = 1000000; div = 1000;
                      break;
          case MILLI: mul = 1000; div = 1000000;
                      break;
          default:    mul = 1000; div = 1000000;
                      break;
      }

      clock_gettime(CLOCK_MONOTONIC, &t);
      tu = (t.tv_sec - ptask_t0.tv_sec)*mul;
      tu += (t.tv_nsec - ptask_t0.tv_nsec)/div;
      return tu;
}
```

Fig. 12.13 Function that returns the current system time in a desired time unit

```
void ptask_init(int policy)
{
int  i;

      ptask_policy = policy;
      clock_gettime(CLOCK_MONOTONIC, &ptask_t0);

      for (i=0; i<MAX_TASKS; i++)
           sem_init(&tp[i].asem, 0, 0);
}
```

Fig. 12.14 Function that initializes some Ptask variables

The function code is shown in Fig. 12.14. Note that `ptask_policy` and `ptask_t0` are global variables used to store the scheduling policy common to all the tasks and the application start time, respectively.

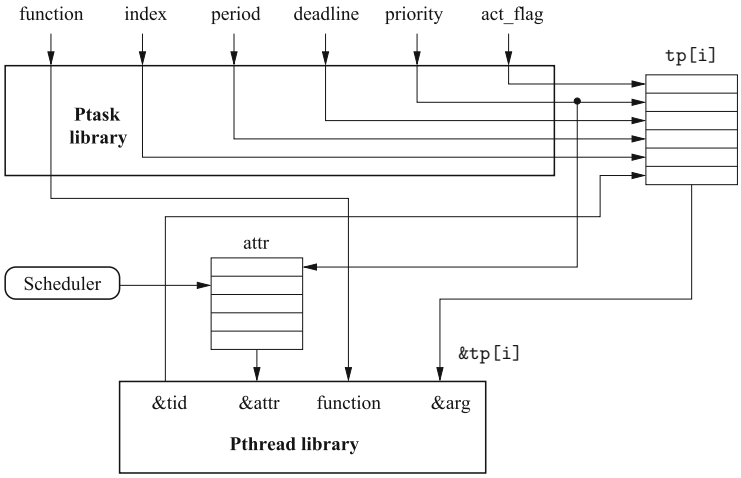


Fig. 12.15 Mapping the task parameters provided to the `task_create` function into the `pthread_create` arguments

The `task_create` function has to store the values of the task parameters into the element of the `tp` structure corresponding to the task being created (this element is accessed through the task index). Then, it has to call the `pthread_create` function with the required arguments. Figure 12.15 graphically illustrates where the various parameters are stored and how they are mapped to the `pthread_create` arguments. The function code is reported in Fig. 12.16.

Note that, if the activation flag is set to `ACT`, the `task_create` function executes a `task_activate(i)` to prevent task `i` to be blocked by the `wait_for_activation`. If instead the activation flag is set to `NO_ACT`, task `i` can be blocked by calling the `wait_for_activation` until a `task_activate(i)` is invoked by some other task.

The function `get_task_index` is illustrated in Fig. 12.17. It extracts the task index from the passed argument (`&tp[i]`) and returns it to the calling task. Note that, since the thread argument is passed as a pointer to `void`, such a pointer must first be converted to a pointer to `struct task_par`.

The function `wait_for_activation` is illustrated in Fig. 12.18. It first calls the `sem_wait` on the activation semaphore (`asem`) associated with the task to wait for the `task_activate` call to be invoked by some other task. Then, it reads the current time and set the next activation time and the absolute deadline of the calling task.

The `task_activate` function is shown in Fig. 12.19. Its purpose is simply to call a `sem_post` on the activation semaphore (`asem`) associated with the task whose index `i` is passed as a parameter. It this task is waiting on `wait_for_activation`, it will be unblocked. If instead `task_activate(i)` is executed before task `i` performs the `wait_for_activation`, then task `i` will not be blocked and will enter its cycle.

```
int task_create(
    void* (*task)(void *), // task function
    int   i,               // task index
    int   period,          // task period
    int   deadline,        // relative deadline
    int   priority,        // task priority
    int   act_flag)        // activation flag
{
pthread_attr_t   myatt;
struct     sched_param mypar;
int        tret;

    if (i >= MAX_TASKS) return -1;

    tp[i].arg = i;
    tp[i].period = period;
    tp[i].deadline = deadline;
    tp[i].priority = priority;
    tp[i].dmiss = 0;

    pthread_attr_init(&myatt);
    pthread_attr_setinheritsched(&myatt,
          PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&myatt, ptask_policy);

    mypar.sched_priority = tp[i].priority;
    pthread_attr_setschedparam(&myatt, &mypar);

    tret = pthread_create(&tp[i].tid, &myatt, task,
          (void*)(&tp[i]));

    if (aflag == ACT) task_activate(i);
    return tret;
}
```

Fig. 12.16 Ptask function to create a periodic task

Fig. 12.17 Function that returns the task index

```
int  get_task_index(void* arg)
{
struct task_par *tpar;

     tpar = (struct task_par *)arg;
     return tpar->arg;
}
```

```
void wait_for_activation(int i)
{
struct timespec t;

      sem_wait(&tp[i].asem);

      clock_gettime(CLOCK_MONOTONIC, &t);
      time_copy(&(tp[i].at), t);
      time_copy(&(tp[i].dl), t);
      time_add_ms(&(tp[i].at), tp[i].period);
      time_add_ms(&(tp[i].dl), tp[i].deadline);
}
```

Fig. 12.18 Function to wait for the task activation

Fig. 12.19 Function to activate a task

```
void task_activate(int i)
{
      sem_post(&tp[i].asem);
}
```

```
int  deadline_miss(int i)
{
struct timespec now;

      clock_gettime(CLOCK_MONOTONIC, &now);

      if (time_cmp(now, tp[i].dl) > 0) {
          tp[i].dmiss++;
          return 1;
      }
      return 0;
}
```

Fig. 12.20 Function to check for a deadline miss

The `deadline_miss` function is shown in Fig. 12.20. It reads the current time (`now`) and compares it with the task absolute deadline using the `time_cmp` function previously defined. If the current time is greater than the absolute deadline, it increments the deadline miss counter and returns `1`, otherwise returns `0`.

The `wait_for_period` function is shown in Fig. 12.21. It suspends the calling thread until the next activation and, after the task is awaken, updates both the next activation time and the absolute deadline by adding a period.

The other Ptask functions listed in Fig. 12.9, as, for example, `task_get_period` or `task_set_period`, are trivial to implement, since they just read or write the value of the corresponding element of the task parameter structure. For this reason, they are not reported here, and its implementation is left to the reader.

```
int  wait_for_period(int i)
{
     clock_nanosleep(CLOCK_MONOTONIC,
                     TIMER_ABSTIME, &(tp[i].at), NULL);
     time_add_ms(&(tp[i].at), tp[i].period);
     time_add_ms(&(tp[i].dl), tp[i].period);
}
```

Fig. 12.21 Function to suspend a periodic task until the next period

```
I am Task 1 running every 10 milliseconds

I am Task 2 running

I am Task 3 r
```

Fig. 12.22 Sample output produced by three periodic tasks with periods equal to 10, 20, and 30 ms

12.4 An Example

To illustrate how to use the Ptask library in practice, this section presents a simple application consisting of a number of periodic tasks dynamically created by the user by pressing a key. Each task writes a string on the screen in a different row, printing one character every period. The string is a local variable containing a simple hello message, the task index, and the task period. It is initialized by the task after getting its index and before entering the loop. All the tasks share the same code, and data are differentiated based on the task index.

To manage the screen in a more flexible way, the application exploits the graphic features of the Allegro library,[2] which allows the user to draw lines and geometric figures, load bitmaps, and display colored text in arbitrary positions of the screen. Figure 12.22 shows an example of the output expected when activating three periodic tasks, with period equal to 10, 20, and 30 ms, respectively.

Note that, since each task prints a character of the message every period, when task 1 displays the entire message, task 2 displays half message and task 3 one third of the message (if they start simultaneously).

Figure 12.23 shows the global constants defined for this application.

The following global variable, when set to 1 by the `main` function, is used to exit all periodic loops and terminate the tasks:

```
int  end = 0;
```

[2] https://github.com/dmitrysmagin/allegro.

```
#define   XWIN    640      // window x resolution
#define   YWIN    480      // window y resolution
#define   XBASE   40       // X start for the message
#define   YBASE   50       // Y level for the first task
#define   YINC    30       // Y increment for other tasks
#define   BKG     0        // background color
#define   MAXT    10       // max number of tasks
#define   LEN     80       // max message length
#define   PER     30       // base period
#define   PINC    20       // period increment
```

Fig. 12.23 Global constants defined in the sample application

```
int  main(void)
{
int  p, i = 0;
char scan;   // scan code

     init();

     do {
         if (keypressed()) scan = readkey() >> 8;

         if (scan == KEY_SPACE && i < MAXT) {
             p = PER + i*PINC;
             task_create(i, hello, p, p, 50, ACT);
             i++;
         }

     } while (scan != KEY_ESC);

     end = 1;
     for (i=0; i<=MAXT; i++) wait_for_task_end(i);
     allegro_exit();
     return 0;
}
```

Fig. 12.24 Code of the `main` function

Figure 12.24 shows the code of the `main` function, which calls the `init()` function to initialize the Allegro environment and the Ptask library. The code of the `init` function is shown in Fig. 12.25. Finally, the code of the `hello` task is reported in Fig. 12.26.

```
void init(void)
{
char s[SLEN]];      // string to print the prompt

      allegro_init();
      set_gfx mode(GFX_AUTODETECT_WINDOWED, XWIN, YWIN, 0, 0);
      clear_to_color(screen, BKG);
      install_keyboard();
      ptask_init(SCHED_FIFO);

      sprintf(s, "Press SPACE to create a task");
      textout_ex(screen, font, s, 10, 10, 14, BKG);
}
```

Fig. 12.25 Code of the `init` function

```
void* hello(void* arg)
{
int   i, k = 0;        // task and character index
int   x, y;            // text coordinates
char  buf[2];          // buffer for printing a char
char  msg[LEN];        // buffer for the message

      i = get_task_index(arg);
      sprintf(mes[i], "I am task %d running every
              %d milliseconds", i, task_get_period(i));

      wait_for_activation(i);

      while (!end) {

          x = XBASE + k*8;
          y = YBASE + i*YINC;
          sprintf(buf, "%c", msg[k]);
          textout_ex(screen, font, buf, x, y, 2+i, BKG);

          k = k + 1;
          if (mes[i][k] == '\0') {
              k = 0;
              textout_ex(screen, font, msg, XBASE, y, BKG, BKG);
          }

          wait_for_period(i);
      }
}
```

Fig. 12.26 Code of the `hello` task

Chapter 13
Real-Time Operating Systems and Standards

This chapter presents a brief overview of the state of art of real-time systems and standards. We first discuss the most common operating systems standard interfaces that play a major role for developing portable real-time applications. Then, we give a brief description of the most used commercial and open source real-time kernels available today, including some research kernels developed within the academia to experiment novel features and lead the future development. We finally present a set of development tools that can be used to speed up system analysis and implementation.

13.1 Standards for Real-Time Operating Systems

The role of standards in operating systems is very important as it provides portability of applications from one platform to another. In addition, standards give the possibility of having several kernel providers for a single application, so promoting competition among vendors and increasing quality. Current operating system standards mostly specify portability at the source code level, requiring the application developer to recompile the application for every different platform. There are four main operating system standards available today:

- POSIX, the main general-purpose operating system standard, with real-time extensions (RT-POSIX);
- OSEK, for the automotive industry;
- APEX, for avionics systems;
- μITRON, for small embedded systems.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_13

13.1.1 RT-POSIX

The goal of the POSIX standard (Portable Operating System Interface based on UNIX operating systems) is the portability of applications at the source code level. Its real-time extension (RT-POSIX) is one of the most successful standards in the area of real-time systems, adopted by all major kernel vendors.

The standard specifies a set of system calls for facilitating concurrent programming. Services include mutual exclusion synchronization with priority inheritance, wait and signal synchronization via condition variables, shared memory objects for data sharing, and prioritized message queues for intertask communication. It also specifies services for achieving predictable timing behavior, such as fixed-priority preemptive scheduling, sporadic server scheduling, time management with high resolution, sleep operations, multipurpose timers, execution-time budgeting for measuring and limiting task execution times, and virtual memory management, including the ability to disconnect virtual memory for specific real-time tasks. Since the POSIX standard is so large, subsets are defined to enable implementations for small systems. The following four real-time profiles are defined by POSIX.13 [POS03]:

- **Minimal Real-Time System Profile (PSE51)**. This profile is intended for small embedded systems, so most of the complexity of a general purpose operating system is eliminated. The unit of concurrency is the thread (processes are not supported). Input and output is possible through predefined device files, but there is not a complete file system. PSE51 systems can be implemented with a few thousand lines of code and with memory footprints in the tens of kilobytes range.
- **Real-Time Controller Profile (PSE52)**. It is similar to the PSE51 profile, with the addition of a file system in which regular files can be created, read, or written. It is intended for systems like a robot controller, which may need support for a simplified file system.
- **Dedicated Real-Time System Profile (PSE53)**. It is intended for large embedded systems (e.g., avionics) and extends the PSE52 profile with the support for multiple processes that operate with protection boundaries.
- **Multi-purpose Real-Time System Profile (PSE54)**. It is intended for general-purpose computing systems running applications with real-time and non-real-time requirements. It requires most of the POSIX functionality for general purpose systems and, in addition, most of the real-time services.

In summary, the RT-POSIX standard enables portability of real-time applications and specifies real-time services for the development of fixed-priority real-time systems with high degree of predictability. In the future, it is expected that RT-POSIX evolves toward more flexible scheduling schemes.

13.1.2 OSEK/VDX

OSEK/VDX[1] is a joint project of many automotive industries that aims at the definition of an industrial standard for an open-ended architecture for distributed control units in vehicles [OSE03]. The term OSEK means *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug* (open systems and the corresponding interfaces for automotive electronics); the term VDX means *Vehicle Distributed eXecutive*.

The objective of the standard is to describe an environment that supports efficient utilization of resources for automotive application software. This standard can be viewed as an application program interface (API) for real-time operating systems integrated on a network management system (VDX), which describes the characteristics of a distributed environment that can be used for developing automotive applications.

The typical applications considered by the standard are control applications with tight real-time constraints, high criticality, and large production volumes. To save on production costs, there is a strong push toward code optimization, by reducing the memory footprint to a minimum and enhancing the OS performance as much as possible. A typical OSEK system has the following characteristics:

- **Scalability**. The operating system is intended to be used on a wide range of hardware platforms (from 8 bit microcontrollers, to more powerful processors). To support such a wide range of systems, the standard defines four conformance classes with increasing complexity. Memory protection is not supported at all.
- **Software portability**. An ISO/ANSI-C interface between the application and the operating system is adopted to simplify software portability. However, due to the wide variety of hardware platforms, the standard does not specify any interface to the I/O subsystem. This reduces portability of the application source code, since the I/O system strongly impacts on the software architecture.
- **Configurability**. Appropriate configuration tools proposed by the OSEK standard help the designer in tuning the system services and the system footprint. Objects that need to be instantiated in the application can be specified through a language called OIL (OSEK Implementation Language).
- **Static allocation of software components**. All the kernel objects and the application components are statically allocated. The number of tasks, their code, the required resources, and services are defined at compile time. This approach simplifies the internal structure of the kernel and makes it easier to deploy the kernel and the application code on a ROM.
- **Kernel awareness**. The OSEK standard supports a standard OSEK Run Time Interface (ORTI) used to instruct a debugger about the meaning of specific data structures, so that specific information can be visualized on the screen in the proper way. For example, when tracing the running task, it is possible to visualize

[1] OSEK/VDX: http://www.osek-vdx.org.

the corresponding task identifier, as well as the time at which context switches take place.
- **Support for time-triggered architectures**. The OSEK standard provides the specification of OSEKTime OS, a time-triggered operating system that can be fully integrated in the OSEK/VDX framework.

13.1.2.1 Details on the OSEK/VDX Standard

The first feature that distinguishes an OSEK kernel from other operating systems is that all kernel objects are *statically* defined at compile time. In particular, most of these systems do not support dynamic memory allocation and dynamic tasks creation. To help the user in configuring the system, the OSEK/VDX standard defines an OSEK Implementation Language (OIL) to specify the objects that must be instantiated in the application. When the application is compiled, the OIL compiler generates the operating system data structures, allocating the exact amount of memory needed by the application, to be put in flash memory (which is less expensive than RAM memory on most microcontrollers).

The second feature distinguishing an OSEK/VDX system is the support for *Stack Sharing*. The reason for providing stack sharing is to save RAM memory, which is very expensive on small microcontrollers. The possibility of implementing a stack sharing system is related to how the task code is written. In traditional real-time systems, a periodic task is structured according to the following scheme:

```
Task(x) {
    int local;
    initialization();
    for (;;) {
        do_instance();
        end_instance();
    }
}
```

Such a scheme is characterized by a forever loop containing an instance of the periodic task that terminates with a blocking primitive (`end_instance()`), which has the effect of suspending the task until its next activation. When following such a programming scheme (called *extended task* in OSEK/VDX), the task is always present in the stack, even when waiting for its next activation. In this case, the stack cannot be shared, and a separate stack space must be allocated for each task.

To enable stack sharing, OSEK/VDX provides support for *basic tasks*, which are special tasks that are implemented in a way more similar to functions, according to the following scheme:

```
int local;

Task x() {
    do_instance();
}

system_initialization() {
    initialization();
    ...
}
```

With respect to extended tasks, in basic tasks, the persistent state that must be maintained between different instances is not stored in the stack, but in global variables. Also, the initialization part is moved at system initialization, because tasks are not dynamically created, but they exist since the beginning. No synchronization primitive is needed to block the task until its next period, because the task is activated every time a new instance starts. Finally, the task cannot call any blocking primitive; therefore it can either be preempted by higher priority tasks or execute until completion. In this way, the task behaves like a function, which allocates a frame on the stack, runs, and then cleans the stack frame. For this reason, such tasks do not occupy stack space between two executions, allowing the stack to be shared among them.

Concerning task management, OSEK/VDX kernels provide support for fixed-priority scheduling with Immediate Priority Ceiling (see Sect. 7.5) to avoid the priority inversion problem. The usage of Immediate Priority Ceiling is supported through the specification of the resource usage of each task in the OIL configuration file. The OIL compiler computes the resource ceiling of each task based on the resource usage declared by each task in the OIL file.

OSEK/VDX systems also support non-preemptive scheduling and preemption thresholds (see Sect. 8.3) to limit the overall stack usage. The main idea is that limiting the preemption between tasks reduces the number of tasks allocated on the system stack at the same time, further reducing the overall amount of required RAM. Note that reducing preemptions may degrade the schedulability of the tasks set; hence the degree of preemption must be traded with the system schedulability and the overall RAM memory used in the system.

Another requirement for operating systems designed for small microcontrollers is *scalability*, which implies the possibility of supporting reduced versions of the API for smaller footprint implementations. In mass production systems, in fact, the memory footprint has a significant impact on the overall system cost. In this context, scalability is provided through the concept of *conformance classes*, which define specific subsets of the operating system API. Conformance classes are also accompanied by an upgrade path between them, with the final objective of supporting partial implementation of the standard with reduced footprint. The conformance classes supported by the OSEK/VDX standard are:

BCC1: This is the smallest conformance class, supporting a minimum of eight tasks with different priority and a single shared resource.

BCC2: Compared to BCC1, this conformance class adds the possibility to have more tasks at the same priority. Each task can have pending activations, that is, the operating system records the number of instances that have been activated but not yet executed.

ECC1: Compared to BCC1, this conformance class adds the possibility to have extended tasks that can wait for an event to occur.

ECC2: This conformance class adds both multiple activations and extended tasks.

Another interesting feature of OSEK/VDX systems is that the system provides an API for controlling interrupts. This is a major difference when compared to POSIX-like systems, where the interrupts are exclusive domain of the operating system and are not exported to the operating system API. The rationale for this is that on small microcontrollers, users often want to directly control interrupt priorities; hence it is important to provide a standard way to deal with interrupt disabling/enabling.

The OSEK/VDX standard specifies two types of interrupt service routines (ISR):

ISR1: simpler and faster, does not implement a call to the scheduler at the end of the interrupt handler;

ISR2: this mechanism can call primitives that change the scheduling behavior. The end of the ISR is a rescheduling point. ISR1 has always higher priority than ISR2.

An important feature of OSEK/VDX kernels is the possibility to fine-tune the footprint by removing error checking code from the production versions, as well as to define hooks that will be called by the system when specific events occur. These features allow the programmer to fine-tuning the application footprint, that will be larger (and safer) when debugging and smaller in production, when most bugs are found and removed from the code.

To support a better debugging experience, the OSEK/VDX standard defines a textual language, named ORTI, which describes where the various objects of the operating system are allocated. The ORTI file is typically generated by the OIL compiler and is used by debuggers to print detailed information about operating system objects defined in the system (e.g., the debugger could print the list of the task in an application with their current status).

13.1.2.2 AUTOSAR OS

Starting from the OSEK/VDX specification, AUTOSAR[2] (AUTomotive Open System ARchitecture) defines a standard for automotive software architecture, jointly developed by automobile manufacturers, suppliers, and tool developers, to minimize the current barriers between functional domains in future vehicle applications.

[2] AUTOSAR: http://www.autosar.org.

Within the AUTOSAR standard, AUTOSAR OS provides the specification at the operating system level, extending the OSEK/VDX standard functionality in various directions, including memory protection, operating system applications, deadline monitoring, execution time monitoring, multicore support, and scheduling tables (for implementing time-triggered activities).

The AUTOSAR OS specification extends and defines those behaviors that were left unspecified as "implementation dependent" in the OSEK/VDX specification and proposes a configuration system described using AUTOSAR XML files.

13.1.3 ARINC-APEX

ARINC 653 (Avionics Application Standard Software Interface) is a software specification for avionics real-time systems that specifies how to host multiple applications on the same hardware. To decouple the operating system from the application software, ARINC 653 defines an API called APEX (APplication/EXecutive). The goal of APEX is to allow analyzable safety critical real-time applications to be implemented, certified and executed. Several critical real-time systems have been successfully built and certified using APEX, including some critical components for the Boeing 777 aircraft.

Traditionally, avionics computer systems have followed a federated approach, where separate functions are allocated to dedicated (often physically disjoint) computing "black-boxes." In recent years there has been a considerable effort by ARINC to define standards for Integrated Modular Avionics (IMA) [ARI91] that allow saving physical resources. IMA defines a standard operating system interface for distributed multiprocessor applications with shared memory and network communications, called the Avionics Application Software Standard Interface [ARI96]. The standard provides some indication about the kernel services expressed as pseudo-code.

Physical memory is subdivided into partitions, and software subsystems occupy distinct partitions at runtime. An offline cyclic schedule is used to schedule partitions. Each partition is temporally isolated from the others and cannot consume more processing time than that allocated to it in the cyclic schedule. Each partition contains one or more application processes, having attributes such as period, time capacity, priority, and running state. Processes within a partition are scheduled on a fixed priority basis. Under APEX, a missed deadline is detected when a rescheduling operation occurs; thus deadlines expiring outside the partition time-slice are only recognized at the start of the next time-slice for that partition.

Communication between processes in different partitions occurs via message passing over logical ports and physical channels. Currently, APEX restricts such messages to be from a single sender to a single receiver. Physical channels are established at initialization time, and many ports may be mapped to a single channel. Two types of messages are supported: *sampling messages*, where the arrival of a new message overwrites the previous one and messages are read non consumable, and

queuing messages, where messages are enqueued in FIFO order and read operation is destructive. A sender blocks when the buffer is full, and a receiver blocks when the buffer is empty. Processes within a partition can communicate using a variety of facilities, including conventional buffers, semaphores, and events, but none of these mechanisms are visible outside the partition.

13.1.4 Micro-ITRON

The ITRON (Industrial TRON—The Real-time Operating system Nucleus) project started in 1984, in Japan. ITRON is an architecture for real-time operating systems used to build embedded systems. The ITRON project has developed a series of de facto standards for real-time kernels, the previous of which was the Micro-ITRON 3.0 specification [Sak98], released in 1993. It included connection functions that allow a single embedded system to be implemented over a network. There are approximately 50 ITRON real-time kernel products for 35 processors registered with the TRON association, almost exclusively in Japan.

The ITRON standards primarily aim at small systems (8–16 and 32 bits). ITRON specification kernels have been applied over a large range of embedded application domains: audio/visual equipment (TVs, VCRs, digital cameras, STBs, audio components), home appliances (microwave ovens, rice cookers, air-conditioners, washing machines), personal information appliances (PDAs, personal organizers, car navigation systems), entertainment (game gear, electronic musical instruments), PC peripherals (printers, scanners, disk drives, CD-ROM drives), office equipment (copiers, FAX machines, word processors), communication equipment (phone answering machines, ISDN telephones, cellular phones, PCS terminals, ATM switches, broadcasting equipment, wireless systems, satellites), transportation (automobiles), industrial control (plant control, industrial robots), and others (elevators, vending machines, medical equipment, data terminals).

The Micro-ITRON 4.0 specification [Tak02] combines the loose standardization that is typical for ITRON standards with a Standard Profile that supports the strict standardization needed for portability. In defining the Standard Profile, an effort has been made to maximize software portability while maintaining scalability. As an example, a mechanism has been introduced for improving the portability of interrupt handlers while keeping overhead small.

The Standard Profile assumes the following system image: high-end 16-32 bit processor, kernel size from 10 to 20 KB, whole system linked in one module, and kernel object statically generated. There is no protection mechanism. The Standard Profile supports task priorities, semaphores, message queues, and mutual exclusion primitives with priority inheritance and priority ceiling protocols.

13.2 Commercial Real-Time Systems

At the present time, there are more than a hundred commercial products that can be categorized as real-time operating systems, from very small kernels with a memory footprint of a few kilobytes to large multipurpose systems for complex real-time applications. Most of them provide support for concurrency through processes and/or threads. Processes usually provide protection through separate address spaces, while threads can cooperate more efficiently by sharing the same address space, but with no protection. Scheduling is typically preemptive, because it leads to smaller latencies and a higher degree of resource utilization, and it is based on fixed priorities. At the moment, there are only a few systems providing deadline-driven priority scheduling. The most advanced kernels implement some form of priority inheritance to prevent priority inversion while accessing mutually exclusive resources. Note that this also requires the use of priority queues instead of regular FIFO queues.

Many operating systems also provide a set of tools for facilitating the development of real-time applications. Besides the general programming tools, such as editors, compilers, and debuggers, there are a number of tools specifically made for real-time systems. Advanced tools include memory analyzers, performance profilers, real-time monitors (to view variables while the program is running), and execution tracers (to monitor and display kernel events in a graphical form). Another useful tool for real-time systems is the schedulability analyzer, which enables designers to verify the feasibility of the task set against various design scenarios. There are also code analyzers to determine worst-case execution times of tasks on specific architectures.

Some major players in this field are:

- VxWorks (Wind River);
- OSE (OSE Systems);
- Windows CE (Microsoft);
- QNX;
- Integrity (Green Hills).

Some of these kernels are described below.

13.2.1 VxWorks

This real-time operating system is produced by Wind River Systems [VxW95], and it is marketed as the runtime component of the Tornado development platform. The kernel uses priority-based preemptive scheduling as a default algorithm, but round-robin scheduling can be also selected as well. It provides 256 priority levels, and a task can change its priority while executing.

Different mechanisms are supplied for intertask communication, including shared memory, semaphores for basic mutual exclusion and synchronization, message queues and pipes for message passing within a CPU, sockets and remote procedure calls for network-transparent communication, and signals for exception handling. Priority inheritance can be enabled on mutual exclusion semaphores to prevent priority inversion.

The kernel can be scaled, so that additional features can be included during development to speed up the work (such as the networking facilities) and then excluded to save resources in the final version.

A performance evaluation tool kit is available, which includes an execution timer for timing a routine or group of routines and some utilities to show the CPU utilization percentage by tasks. An integrated simulator, VxSim, simulates a VxWorks target for use as a prototyping and testing environment.

VxWorks 5.x conforms to the real-time POSIX 1003.1b standard. Graphics, multiprocessing support, memory management unit, connectivity, Java support, and file systems are available as separate services. All major CPU platforms for embedded systems are supported.

Another version, called VxWorks AE, conforms to POSIX and APEX standards. The key new concept in AE is the "protection domain," which corresponds to the partition in ARINC. All memory-based resources, such as tasks, queues, and semaphores, are local to the protected domain, which also provides the basis for automated resource reclamation. An optional Arinc-653 compatible protection domain scheduler (Arinc scheduler for short) extends the protection to the temporal domain. Such a two-level scheduler provides a guaranteed CPU time window for a protection domain in which tasks are able to run with temporal isolation. Priority-based preemptive scheduling is used within a protection domain, not between protection domains. VxWorks 5.x applications can run in an AE protected domain without modifications. VxWorks AE is available for a limited set of CPUs.

13.2.2 OSE

OSE is a real-time operating system produced by ENEA [OSE04]. It comes in three flavors: OSE, OSEck, and Epsilon. OSE is the portable kernel written mostly in C, OSEck is the compact kernel version aimed at digital signal processors (DSPs), and Epsilon is a set of highly optimized assembly kernels. The different kernels implement the OSE API in different levels, from A to D: A is the smallest set of features that is guaranteed to exist on all OSE supported platforms, while D is the full set of features including virtual memory, memory protection, and concept of users. OSE operating systems are widely used in the automotive industry and the communications industry.

OSE processes can either be static or dynamic that is created at compile-time or at runtime. Five different types of processes are supported: interrupt process, timer interrupt process, prioritized process, background process, and phantom process.

There are different scheduling principles for different processes: priority-based, cyclic, and round robin. The interrupt processes and the prioritized processes are scheduled according to their priority, while timer interrupt processes are triggered cyclically. The background processes are scheduled in a round-robin fashion. The phantom processes are not scheduled at all and are used as signal redirectors.

Processes can be grouped into blocks, and each block may be treated as a single process, e.g., one can start and stop a whole block at once. Moreover, one can associate each block a memory pool that specifies the amount of memory available for that block. Pools can be grouped into segments that can feature hardware memory protection if available. There is a special pool for the system.

OSE processes use messages (called signals) as their primary communication method. Signals are sent from one process to another and do not use the concept of mailboxes. Each process has a single input queue from which it can read signals. It is possible to use filters to read only specific types of messages. A process may also have a redirection table that forwards certain types of messages to other processes. By the use of "link handlers," signals may be sent between OSE systems over various communication channels (network, serial lines, etc.).

There is an API functionality to get information about processes on other OSE systems, so the right receiver can be determined. Sending signals to a higher-priority process transfers the execution to the receiver. Sending to lower processes does not. Signals going between processes inside the same memory area do not undergo copying. Only when it is necessary from memory point of view, the signal buffer is copied. Semaphores also exist in more than one flavor, but the use of those is discouraged due to priority inversion problems.

An application can be written across several CPUs by using signal IPC and link handlers. One can mix any kernel type with any other, links are monitored for hardware failures, and alternate routes are automatically attempted to be established upon a link failure. Processes are notified upon link failure events.

Errors in system calls are not indicated by a traditional return code, but as a call to an error handler. The error handlers exist on several levels: process, block, and system level. If an error handler on one level cannot handle the error, it is propagated to the next level, until it reaches the system level.

13.2.3 QNX Neutrino

QNX Neutrino [Hil92] is a real-time operating systems used for mission-critical applications, from medical instruments and Internet routers to telematics devices, process control applications, and air traffic control systems.

The QNX Neutrino microkernel implements only the most fundamental services in the kernel, such as signals, timers, and scheduling. All other components file systems, drivers, protocol stacks, applications run outside the kernel, in a memory-protected user space. As a result, a faulty component can be automatically restarted without affecting other components or the kernel.

Some of the real-time features include distributed priority inheritance to eliminate priority inversion and nested interrupts to allow priority-driven interrupt handling. All components communicate via message passing, which forms a virtual "software bus" that allows the user to dynamically plug in, or plug out, any component on the fly. It provides support for transparent distributed processing using standard messages to access hardware and software resources on remote nodes.

QNX complies with the POSIX 1003.1-2001 standard, providing real-time extensions and threads. It includes a power management framework to enable control over the power consumption of system components. This allows the application developer to determine power management policies.

Finally, QNX Neutrino also provides advanced graphics features, using layering techniques, to create visual applications for markets such as automotive, medical, industrial automation, and interactive gaming. A support for accelerated 3D graphics rendering (based on the Mesa implementation of the OpenGL standard) allows to create sophisticated displays with little impact on CPU performance.

13.3 Linux-Related Real-Time Kernels

Linux is a general-purpose operating system originally designed to be used in server or desktop environments. For this reason, not much attention has been dedicated to real-time issues. As a result, the latency experienced by real-time activities can be as large as hundreds of milliseconds. This makes common Linux distributions not suitable for hard real-time applications with tight timing constraints. On the other hand, making Linux a real-time operating system would allow using the full-power of a real operating system for real-time applications, including a broad range of open source drivers and development tools. For this reason, a considerable amount of work has been done during the last years for providing Linux with real-time features. Two different approaches have been followed:

- The first approach is called *interrupt abstraction*, and it is based on a small real-time executive having full control of interrupts and processor key features, which executes Linux as a thread. To achieve real-time behavior, the whole Linux kernel is treated by the real-time scheduler as the idle task, so Linux only executes when there are no real-time tasks to run. The Linux task can never block interrupts or prevent itself from being preempted. This approach has been successfully implemented in several existing frameworks, the most notable examples of which are RTLinux, RTAI, and Xenomai. It is an efficient solution, as it allows obtaining low latencies, but is also invasive, and, often, not all standard Linux facilities are available to tasks running with real-time privileges.
- The second approach consists in directly modifying the Linux internals to reduce latency and add real-time features. This is the approach followed by the PREEMPT_RT, SCHED_EDF, and Linux/RK projects.

13.3.1 RTLinux

RTLinux has been the first real-time extension for Linux, created by Victor Yodaiken. Currently, two versions of RTLinux are available: an Open-Source version, called RTLinuxFree,[3] and a commercial version covered by patent, distributed by Wind River as Wind River Real-Time Core for Linux.[4] RTLinux works as a small executive with a real-time scheduler that runs Linux as its lowest-priority thread. The Linux thread is made preemptable so that real-time threads and interrupt handlers do not experience long blocking delays by non-real-time operations. The executive modifies the standard Linux interrupt handler routine and the interrupt enabling and disabling macros.

When an interrupt is raised, the micro-kernel interrupt routine is executed. If the interrupt is related to a real-time activity, a real-time thread is notified, and the real-time kernel executes its own scheduler. If the interrupt is not related to a real-time activity, then it is "flagged." When no real-time threads are active, Linux is resumed and executes its own code. Moreover, any pending interrupt related to Linux is served.

In this way, Linux is executed as a background activity in the real-time executive. This approach has the advantage of separating as much as possible the interactions between the Linux kernel, which is very complex and difficult to modify, and the real-time executive. It also allows obtaining a very low average latency for real-time activities and, at the same time, the full power of Linux on the same machine. However, there are several drawbacks. Real-time tasks execute in the same address space as the Linux kernel; therefore, a fault in a user task may crash the kernel. When working with the real-time threads, it is not possible to use the standard Linux device driver mechanism; as a result it is often necessary to rewrite the device drivers for the real-time application. For example, for using the network in real time, it is necessary to use another device driver expressly designed for RTLinux. The real-time scheduler is a simple fixed-priority scheduler, which is POSIX compliant. There is no direct support for resource management facilities. Some Linux drivers directly disable interrupts during some portions of their execution. During this time, no real-time activities can be executed, and thus the worst-case latency of real-time activities is increased.

[3] RTLinuxFree: http://www.rtlinuxfree.com.

[4] RTLinux Wind River: http://www.windriver.com/products/linux/.

13.3.2 RTAI

RTAI[5] (Real-Time Application Interface) started as a modification of RTLinux made by Paolo Mantegazza at the Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Italy. In the course of the years, the original idea of RTLinux has been considerably changed and enhanced. RTAI is now a community project, and the source code is released as open source. Because of some uncertainty about the legal repercussion of the RTLinux patent on RTAI, the RTAI community has developed the *Adaptive Domain Environment for Operating Systems* (Adeos) nano-kernel as an alternative for RTAI's core to get rid of the old kernel patch and exploit a more structured and flexible way to add a real-time environment to Linux. The purpose of Adeos is not limited to be RTAI's core, but it is to provide a flexible environment for sharing hardware resources among multiple operating systems. It also makes it easier to plug in additional features, such as debuggers, analyzers, and standard open middleware layers, serving all operating systems running on top of it almost without any intrusion. A comparison between latency with and without RTAI is available on [LS06].

RTAI offers the same services of the Linux kernel core, adding the features of an industrial real-time operating system. It basically consists of an interrupt dispatcher that traps the peripherals interrupts and, if necessary, reroutes them to Linux. It is not an intrusive modification of the kernel: it uses the concept of hardware abstraction layer (HAL) to get information from Linux and trap some fundamental functions.

RTAI allows to uniformly mix hard and soft real-time activities by symmetrically integrating the scheduling of RTAI tasks, Linux kernel threads, and user-space tasks. By using Linux schedulable objects, RTAI benefits from threads protection at the price of a slight increase in latencies. RTAI offers also a native, dynamically extensible, light middleware layer based on the remote procedure call concept, which allows using the whole API in a distributed way.

One of the major limits of RTAI is that the project mainly focuses on x86 architectures; thus there is a very limited support for embedded platforms. For this reason, developers working in the embedded area often prefer the usage of Xenomai.

13.3.3 Xenomai

Started as a branch of the RTAI project, Xenomai[6] is the evolution of the Fusion project, an effort to execute real-time RTAI tasks in user space. Xenomai brings the concept of virtualization one step further: like RTAI, it uses the Adeos nano-kernel to provide the interrupt virtualization, but it allows a real-time task to

[5] RTAI: http://www.rtai.org.

[6] Xenomai: http://www.xenomai.org.

execute also in user space. This is done by introducing the concept of *domains*: in the *primary domain*, the task is controlled by the RTOS, while in the *secondary domain*, it is controlled by the Linux scheduler. Normally, a real-time task starts in the primary domain, where it remains as long as it invokes only the RTOS API. It is automatically migrated to the secondary domain as soon as it invokes a standard Linux system call or it needs to handle events such as exceptions or Linux signals. However, the task keeps its real-time priority, being scheduled with the SCHED_FIFO or SCHED_RR Linux policies.

While running in the secondary mode, the task can experience some delay and latency, due to the fact that it is scheduled by Linux. However, at any time after the function call has been completed, it can go back to the primary mode by explicitly calling a function. In this way, real-time applications can use the full power of Linux at the cost of some limited unpredictability.

Xenomai also provides a set of "skins" that implement various APIs offered by common RTOSs (e.g., VxWorks), as well as a POSIX API and a "native" API. This feature increases portability of application code from other platforms to Xenomai.

With respect to RTAI, Xenomai offers the support for a whole set of embedded architectures, including ARM, Blackfin, Altera Nios2, and PowerPC. Xenomai developers are currently putting a considerable effort for integrating its code with the PREEMPT_RT patch, in order to have the benefits of both patches on the same system.

13.3.4 PREEMPT_RT

PREEMPT_RT[7] is a kernel patch to make a Linux system more predictable and deterministic. This is done through several optimizations. The patch makes almost all kernel code preemptable, except for the most critical kernel routines, so reducing the maximum latency experienced by a task (at the cost of a slightly higher average latency). This feature is achieved by replacing almost every spinlock in the kernel code with preemptable mutexes with Priority Inheritance (see Sect. 7.6). A numerical evaluation of the PREEMPT_RT patch is available in [LS06].

The patch also provides "threaded interrupts" by converting interrupt handlers into preemptable kernel threads that are managed in "process context" by the regular Linux scheduler. Whenever some parts of the PREEMPT_RT patch are considered to be stable enough (e.g., the priority inheritance algorithm), they are merged in the official Linux kernel distribution.

[7] PREEMPT_RT: http://rt.wiki.kernel.org.

13.3.5 SCHED_DEADLINE

SCHED_DEADLINE[8] is a Linux kernel patch developed by Evidence s.r.l. in the context of the ACTORS European project.[9] It adds a deadline-based scheduler with resource reservations in the standard Linux kernel. The implemented algorithm is a global Constant Bandwidth Server (see Sect. 6.9), implemented as a partitioned algorithm with migration. Migration can be disabled; thus, it can also work as a partitioned algorithm.

The patch adds a new scheduling class to the Linux scheduler, so normal tasks can still behave as when the patch is not applied. The patch also adds one further system call to set budget and period for any SCHED_DEADLINE task.

The patch is platform-independent; therefore it supports any embedded platform supported by the standard Linux kernel. It also natively supports multicore architectures. Several options are available at runtime, like bandwidth reclaiming, soft or hard reservations, deadline inheritance, etc. The development is still ongoing, and new versions are often released on the Linux Kernel Mailing List (LKML). Eventually, this patch might be merged inside the official Linux kernel.

13.3.6 Linux/RK

In Linux/RK, the Linux kernel has been directly modified [Raj98, Raj00] to introduce real-time features. The kernel is supported by TimeSys Inc. RK stands for "resource kernel," because the kernel provides resource reservations directly to user processes. The use of this mechanism is transparent; thus it is possible to assign a reservation to a legacy Linux application. Moreover, it is possible to access a specific API to take advantage of the reservations and of the quality of service management.

A reserve represents a share of a single computing resource, which can be CPU time, physical memory pages, a network bandwidth, or a disk bandwidth. The kernel keeps track of the use of a reserve and enforces its utilization, when necessary. A reserve can be time-multiplexed or dedicated. Temporal resources like CPU cycles, network bandwidth, and disk bandwidth are time-multiplexed, whereas spatial resources, like memory pages, are dedicated. A time-multiplexed resource is characterized by three parameters, C, D, and T, where T represents a recurrence period, C represents the processing time required within T, and D is the deadline within which the C units of processing time must be available within T.

Within Linux/RK an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is available

[8] SCHED_DEADLINE: http://gitorious.org/sched_deadline/pages/Home.

[9] ACTORS: http://www.actors-project.eu.

to the application. Such a guarantee of resource allocation gives an application the knowledge of the amount of its currently available resources. A QoS manager or an application itself can then optimize the system behavior by computing the best QoS obtained from the available resources.

To simplify portability, the modifications to the original Linux kernel were limited as much as possible, and the RK layer has been developed as an independent module with several callback hooks.

13.4 Open-Source Real-Time Research Kernels

The main characteristics that distinguish this type of operating systems include:

- The ability to treat tasks with explicit timing constraints, such periods and deadlines;
- The presence of guarantee mechanisms that allow to verify in advance whether the application constraints can be met during execution;
- The possibility to characterize tasks with additional parameters, which are used to analyze the dynamic performance of the system;
- The use of specific resource access protocols that avoid priority inversion and limit the blocking time on mutually exclusive resources.

Some of the first operating systems that have been developed according to these principles are CHAOS [SGB87], MARS [KDK+89], Spring [SR91], ARTS [TM89], RK [LKP88], TIMIX [LK88], MARUTI [LTCA89], HARTOS [KKS89], YARTOS [JSP92], and HARTIK [But93]. Most of these kernels never evolved to a commercial product, but they were useful for experimenting novel mechanisms, some of which are to be integrated in next-generation operating systems.

The main differences among the kernels mentioned above concern the supporting architecture on which they have been developed, the static or dynamic approach adopted for scheduling shared resources, the types of tasks handled by the kernel, the scheduling algorithm, the type of analysis performed for verifying the schedulability of tasks, and the presence of fault-tolerance techniques.

The rest of this section presents three examples of such kernels:

- Erika Enterprise, an OSEK kernel for small embedded platforms;
- Shark, a POSIX-like kernel for PC platforms;
- Marte OS, a POSIX-like kernel for PC platforms supporting C++ and Ada 2005.

13.4.1 Erika Enterprise

Erika Enterprise[10] is a real-time operating system that proposes a free of charge open-source implementation of the OSEK/VDX API. Erika Enterprise is supported by RT-Druid, a set of Eclipse plugins implementing an OSEK OIL compiler, able to produce an OSEK ORTI file compatible with Lauterbach Trace32 debuggers. The kernel is characterized by a minimal memory footprint of 1–4 Kb flash and an easy-to-use API. ERIKA Enterprise also supports stack sharing, allowing all basic tasks in the system to share a single stack, so reducing the overall RAM memory used for this purpose. Several examples are available and maintained by the online community of developers.

In addition to the standard OSEK/VDX conformance classes, Erika Enterprise provides other customized conformance classes implementing EDF and soft-real-time resource reservations with the IRIS algorithm [MLBC04]. It also supports a subset of the AUTOSAR Scalability Class SC4, including memory protection support and OS Applications. Erika Enterprise supports a number of third-party libraries, including TCP/IP, 802.15.4, CMOS cameras, analog and digital sensors, encoders, DACs, ADCs, 3-axis accelerometers, DC motors, and many others. It also provides direct support for code generation from ScicosLab[11] models, useful for implementing model-driven data-flow control loops by the CAL[12] open dataflow language. Thanks to its compliancy to the OSEK/VDX standard, ERIKA Enterprise is now used by various companies for automotive applications and white goods systems (e.g., washing machines, air conditioners, refrigerators, and stoves).

13.4.1.1 Efficient EDF Implementation

ERIKA Enterprise contains an efficient implementation of the EDF scheduler, implemented as an extension to the OSEK/VDX specification.

There are two problems that need to be solved to implement an EDF scheduler on an OSEK system. The first is to provide a lightweight internal timing reference, since OSEK only exports an alarm interface, leaving the counter objects unspecified and implementation dependent. ERIKA Enterprise resolved the issue by adding an internal timing support that is implicitly used by the kernel primitives that follow the EDF specification. The second problem is to provide a timing reference with both high resolution (to handle short and fast activities with deadlines in the order of tens/hundreds of microseconds) and long lifetime (to handle absolute deadlines).

In POSIX systems, such timing properties have been resolved by using a `struct timespec` data structure, containing a 32 bit integer representing

[10] Erika Enterprise: http://erika.tuxfamily.org.

[11] ScicosLab: http://www.scicos.org.

[12] CAL: http://embedded.eecs.berkeley.edu/caltrop/language.html.

nanoseconds (thus providing high resolution) and a 32 bit integer representing seconds (thus giving a lifetime of around 136 years, more than enough for practical applications). Unfortunately, this approach cannot be implemented on small microcontrollers, where handling a 64 bit data structure imposes a significant overhead in the scheduler implementation. To overcome this problem, Erika adopts a circular timer [CB03], which replaces the absolute timing reference with a timing reference relative to the current time t, implemented by a free running timer.

In this way, it is possible to handle absolute deadlines that can range between $t - P/2$ and $t + P/2$, where P is the timer lifetime, defined as the minimum interval of time between two non-simultaneous events characterized by the same time representation. For example, in a Microchip dsPIC microcontroller with a timer clock running at 2 MHz, the 32 bit hardware timer can achieve a lifetime of 1073 s (enough for many practical applications).

If $t(e_i)$ is the absolute time at which event e_i occurs, the circular time method can be expressed follows.

If events are represented by n-bit unsigned integers, such that

$$\forall t \ \forall e_i, e_j \in E(t) \ |t(e_i) - t(e_j)| < \frac{P}{2} \tag{13.1}$$

then $\forall t \ \forall e_i, e_j \in E(t)$ we have:

1. $t(e_i) > t(e_j) \iff (e_i \ominus e_j) < \frac{P}{2}\ ,\ (e_i \ominus e_j) \neq 0$
2. $t(e_i) < t(e_j) \iff (e_i \ominus e_j) > \frac{P}{2}$
3. $t(e_i) = t(e_j) \iff (e_i \ominus e_j) = 0$

where $\ominus$ denotes a subtraction between n-bit integers, evaluated as an unsigned n-bit integer. It is worth observing that for 8/16/32-bit integers such a subtraction operation does not require a special support since it is implemented in all CPUs.

Figure 13.1 shows a set of events which satisfies condition (13.1).
The main property of the $\ominus$ operator is that

$$\forall a, b \in [0, 2^n - 1] \ \ unsigned(b \ominus a) = dist(a, b)$$

where

- $dist(x, y)$ is the distance from x to y evaluated on the time circle in the direction of increasing time values. Notice that $dist(x, y) = d$ means that if $t = x$ then, after a delay of d, we have $t = y$, independently from the fact that x and y belong to two different cycles.
- $unsigned(x)$ is the value of x, interpreted as an n-bit unsigned value. We recall that according to the 2's complement representation,

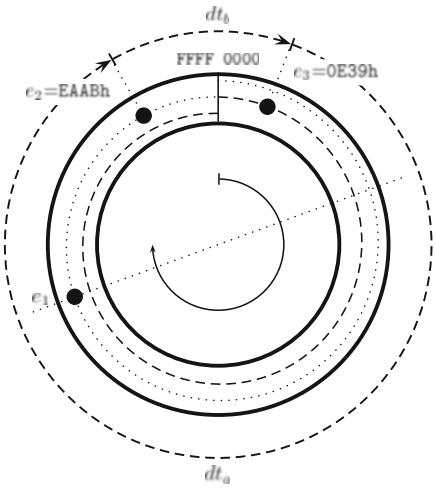


Fig. 13.1 Circular timer implementation in ERIKA Enterprise

$$unsigned(x) = \begin{cases} x & \text{if } x \geq 0 \\ 2^n + x & \text{otherwise} \end{cases}$$

For example, when evaluating the two events e_2 and e_3 in Fig. 13.1, we have that $dt_a = (e_2 - e_3) = DC72H > 8000H = P/2$. Hence, we conclude that e_2 must precede e_3 and that the actual time difference between the events is $dt_b = (e_3 - e_2) = 238EH < P/2$.

The implementation of EDF is done in a new conformance class, called "`EDF`." The EDF conformance class retains the same static approach of the OSEK/VDX standard, with a slightly simplified API (basically, the API does not include the `TerminateTask` primitive). The configuration of the EDF scheduler is then specified in the OIL file used by ERIKA Enterprise. For example, an EDF conformance class with a timer tick of 25 ns is specified as follows:

```
KERNEL_TYPE = EDF { TICK_TIME = "25 ns ";};
```

A relative deadline of 10 ms can be specified (in the OIL file) as

```
TASK myTask1 {
    REL_DEADLINE = "10 ms ";
};
```

The OIL compiler converts the milliseconds in the proper number of ticks, using the tick duration specified in the `TICK_TIME` variable. The rest of the OSEK/VDX kernel primitives are basically the same, since all system calls do not have periodicity and priorities as parameters.

Finally, based on the EDF scheduler, ERIKA Enterprise also supports resource reservation (see Sect. 9.3). The implementation was carried out within the FRES-

COR European Project[13] and consisted in the development of the IRIS scheduler [MLBC04] on top of the EDF scheduler. As a result, ERIKA Enterprise is able to support resource reservation under EDF with reclaiming of unused computation time and with a footprint of about 10 Kb Flash.

13.4.1.2 Multicore Support

Another feature that distinguishes ERIKA Enterprise from other real-time kernels is the support for multicore systems, that is, the possibility for an application to be statically partitioned on the various cores available on modern architectures.

Under ERIKA Enterprise, each concurrent task is statically linked at build time to a given CPU. Each CPU includes a separate copy of the operating system and of the device drivers that are present on the particular CPU. The partitioned approach is typically forced by modern multicore system-on-a-chip, since it is often not efficient to migrate the execution of a thread from one CPU to another.

Ideally, to allow the designer to write partitioning-independent code, tasks partitioning should be done at the end of the application design, without changing the application source code. In practice, partitioning can be limited by the fact that some hardware peripherals may not be available on all cores. The minimal partitioning item in a system is the source file; hence the code of tasks allocated to different CPUs must be on different files, to allow an easy partitioning in different CPU in a later stage of the development.

In a heterogeneous system, each core has usually different peripherals, and the memory spaces shared between cores may vary. Cache coherency is often not present due to cost reasons, and proper cache disabling techniques have to be used to avoid data corruption. One of the CPUs usually works as the master CPU, which plays the role of initializing the global data structures. That master CPU is also responsible for the implementation of a startup barrier, which is used to synchronize all cores at startup to have a coherent operating system startup.

The current implementation of ERIKA Enterprise (available open source) supports Altera Nios II multicore designs on FPGA, and Freescale PPC 5668G FADO, which is a dual core hosting a PPC z6 and a PPC z0 (the second processor is typically used to handle high-speed peripherals).

13.4.2 Shark

SHARK[14] (Soft and HArd Real-time Kernel) is a dynamic configurable real-time operating system developed at the Scuola Superiore S. Anna of Pisa [GAGB01]

[13] FRESCOR Project: http://www.frescor.org.

[14] SHARK: http://shark.sssup.it/.

to support the development and testing of new scheduling algorithms, aperiodic servers, and resource management protocols. The kernel is designed to be modular, so that applications can be developed independently of a particular system configuration and schedulers can be implemented independently of other kernel mechanisms.

The kernel is compliant with the POSIX 1003.13 PSE52 specifications and currently runs on Intel x86 architectures. SHARK is currently used for education in several real-time systems courses all over the world. It is available for free under GPL license at http://shark.sssup.it.

13.4.2.1 Kernel Architecture

To make scheduling algorithms independent of applications and other internal mechanisms, SHARK is based on the notion of a *Generic Kernel*, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the *scheduling modules*. In a similar fashion, the access to shared resources is coordinated by *resource modules*. A simplified scheme of the kernel architecture is depicted in Fig. 13.2.

The Generic Kernel provides the mechanisms used by the modules to perform scheduling and resource management, thus allowing the system to abstract from the algorithms that can be implemented. The Generic Kernel simply provides the primitives without specifying any algorithm, whose implementation resides in external modules, configured at runtime with the support of the *Model Mapper*.

Each module consists of a set of data and functions used for implementing a specific algorithm, whose implementation is independent from the other modules in the system. In this way, many different module configurations are possible. For example, a Polling Server can either work with Rate Monotonic or EDF without any modification. Moreover, scheduling modules can be composed into priority layers, as in a multilevel scheduling approach. Several scheduling modules are

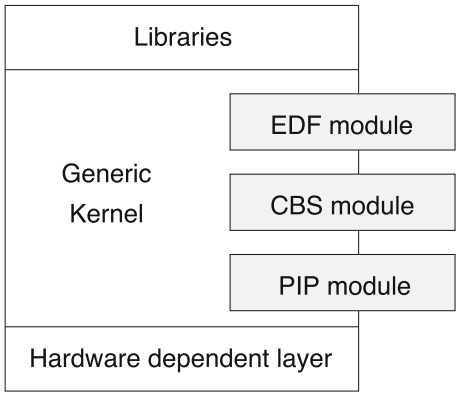


Fig. 13.2 The SHARK architecture

already available, such as Rate Monotonic, EDF, Polling Server, Deferrable Server, Slot Shifting, Total Bandwidth Server, Constant Bandwidth Server, and Elastic Scheduling, as well as a number of resource protocols, such as Priority Inheritance, Priority Ceiling, and Stack Resource Policy.

Another important component of the Generic Kernel is the Job Execution Time (JET) estimator, which monitors the computation time actually consumed by each job. This mechanism is used for statistical measurements, resource accounting, and temporal protection.

The API is exported through a set of *Libraries*, which use the Generic Kernel to support some common hardware devices (i.e., keyboard, sound cards, network cards, and graphic cards). They provide a compatibility layer with the POSIX interface (POSIX 1003.13 PSE52) to simplify porting of applications developed for other POSIX compliant kernels.

Independence between applications and scheduling algorithms is achieved by introducing the notion of *task model*. Two kinds of models are provided: task models and resource models. A task model expresses the QoS requirements of a task for the CPU scheduling. A resource model is used to define the QoS parameters relative to a set of shared resources used by a task. For example, the resource model can be used to specify the semaphore protocol to be used for protecting critical sections (e.g., Priority Inheritance, Priority Ceiling, or SRP). Requirements are specified through a set of parameters that depend on the specific models. Models are used by the generic kernel to assign a task to a specific module.

Task creation works as follows (see Fig. 13.3): when an application issues a request to the kernel for creating a new task, it also sends the model describing the requested QoS. A kernel component, namely, the *model mapper*, passes the model to a module, selected according to an internal policy, and the module checks whether it can provide the requested QoS; if the selected module cannot serve the task, the model mapper selects a different module. When a module accepts to manage the task described by the specified model, it converts the model's QOS parameters into the appropriate scheduling parameters. Such a conversion is performed by a module component, called the *QoS Mapper*.

13.4.2.2 Scheduling Modules

Scheduling modules are used by the Generic Kernel to schedule tasks or serve aperiodic requests using an aperiodic server. In general, the implementation of a scheduling algorithm should possibly be independent of resource access protocols and handle only the scheduling behavior. Nevertheless, the implementation of an aperiodic server relies on the presence of another scheduling module, called the Host Module (e.g., a Deferrable Server can be used if the base scheduling algorithm is RM or EDF, but not round robin). Such a design choice reflects the traditional approach followed in the literature, where most aperiodic servers insert their tasks directly into the scheduling queues of the base scheduling algorithm. Again, the modularity of the architecture hides this mechanism with the task models: an

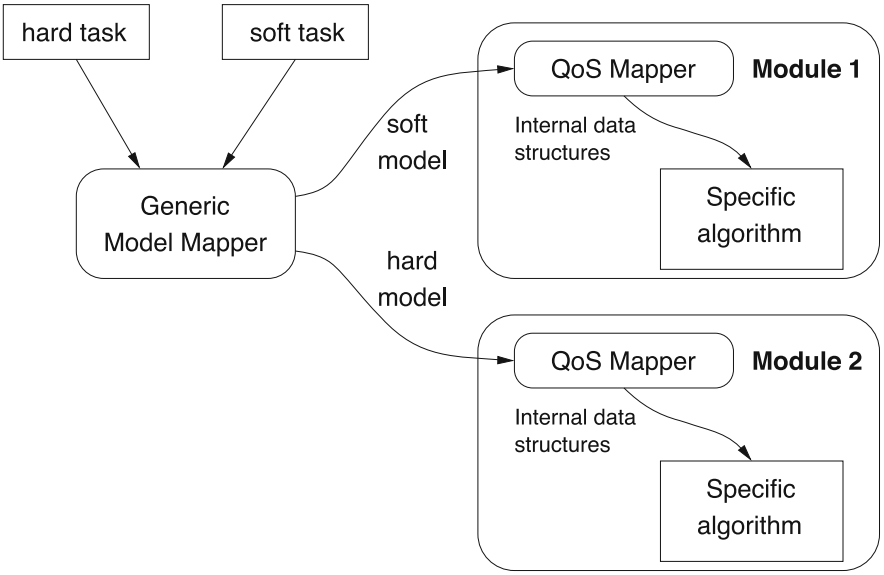


Fig. 13.3 The interaction between the model mapper and the QOS mapper

aperiodic server must use a task model to insert his tasks into the Host Module. In this way, the Guest Module have not to rely on the implementation of the Host Module.

The Model Mapper distributes the tasks to the registered modules according to the task models the set of modules can handle. For this reason, the task descriptor includes an additional field (`task_level`), which points to the module that is handling the task.

When the Generic Kernel has to perform a scheduling decision, it asks the modules for the task to schedule, according to fixed priorities: first, it invokes a scheduling decision to the highest priority module, then (if the module does not manage any task ready to run), it asks the next high-priority module, and so on. In this way, each module manages its private ready task list, and the Generic Kernel schedules the first task of the highest-priority non-empty module's queue.

The interface functions provided by a scheduling module can be grouped in three classes: Level Calls, Task Calls, and Guest Calls.

13.4.2.3 Shared Resource Access Protocols

As for scheduling, SHARK achieves modularity also in the implementation of shared resource access protocols. Resource modules are used to make resource protocols modular and almost independent from the scheduling policy and from the others resource protocols. Each resource module exports a common interface,

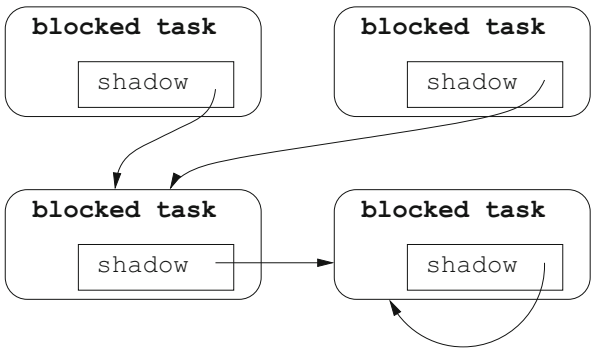


Fig. 13.4 The shadow task mechanism

similar to the one provided by POSIX for mutexes, and implements a specific resource access protocol. A task may also require to use a specified protocol through a resource model.

To make the protocol independent of a particular scheduling algorithm, SHARK supports a generic priority inheritance mechanism independent of the scheduling modules. Such a mechanism is based on the concept of *shadow tasks*. A shadow task is a task that is scheduled in place of another task chosen by the scheduler. When a task is blocked by the protocol, it is kept in the ready queue, and a shadow task is associated with it; when the blocked task becomes the first task in the ready queue, its associated shadow task is scheduled instead. In this way, the shadow task "inherits" the priority of the blocked task.

To implement this solution, a new field `shadow` is added to the generic part of the task descriptor. This field points to the shadow task. Initially, the shadow field is equal to the task ID (no substitution). When the task blocks, the shadow field is set to the task ID of the blocking task or to the task that must inherit the blocked task priority. In general, a graph can grow from a blocking task (see Fig. 13.4). In this way, when the blocked task is scheduled, the blocking (shadow) task is scheduled, thus allowing the schedulers to abstract from the resource protocols. This approach has also the benefit of allowing a classical deadlock detection strategy, by simply searching for cycles in the shadow graph.

13.4.2.4 Device Management

In SHARK, device management is performed outside the kernel, so that the device manager will not steal execution time to the application code. To allow precise resource accounting, SHARK distinguishes between *device drivers* and *device managers*. Device drivers are the hardware-dependent part of the device management code, implementing the routines necessary to perform low-level accesses to the devices. Drivers can easily be inherited from other free operating systems (e.g., Linux) that support most of the current PC hardware. The driver code is compiled

using some glue code, which remap the other system calls to the Generic Kernel interface.

The device manager is hardware independent and only performs device scheduling, taking device management costs into account to provide some form of guarantee on hardware accesses. For this purpose, the manager can be implemented as a dedicated thread or as an application code. If the thread is handled by server implementing temporal protection, the first solution ensures that the device management will not influence the other system's activities. The second solution, however, allows a better precision in accounting the CPU time used by the device manager to the application using the hardware resource.

13.4.3 Marte OS

Marte OS[15] (Minimal Real-Time Operating System for Embedded Applications) is a real-time kernel for embedded applications that follows the Minimal Real-Time System Profile (PSE51) defined in the POSIX.13 standard [POS03]. The services provided by the kernel have a time-bounded response, so hard real-time requirements can be supported. It allows executing concurrent real-time applications on a bare PC and, with some limitations, in a Linux box. MaRTE OS is distributed under a modified-GPL free-software license.

Most of the internal code of MaRTE OS is written in Ada with some C and assembler parts. Nonetheless, APIs for different programming languages are provided, allowing for the development of concurrent real-time applications written in Ada, C, and C++. It is even possible to mix different languages in the same application, for instance, with coexisting (and cooperating) C threads and Ada tasks running under a coherent real-time scheduling policy.

MaRTE OS was initially designed to support embedded applications running on a bare computer. Currently, the supported architecture is a bare PC using an 80386 processor or higher. A basic hardware abstraction layer (HAL) is defined to facilitate porting to other architectures. Such a layer can be implemented using the services of another operating system that acts as a virtual processor. An implementation of MaRTE OS is also available on the Linux operating system, which is useful for testing, development, and teaching purposes.

The development environment is based on the GNU compilers GNAT and `gcc`, as well as on their associated utilities, such as the `gdb` debugger. When developing embedded applications, a cross development environment is used with the development tools hosted in a Linux system. The executable images can be uploaded to the target via an Ethernet link, and cross debugging is possible through a serial line. It is also possible to write the executable image to a bootable device such as a flash memory, for isolated execution in the target. MaRTE OS has been used to

[15] Marte OS: http://marte.unican.es/.

develop industrial embedded systems and is also an excellent tool for educational activities related to real-time embedded systems programming.

13.4.3.1 Supported Functionality

MaRTE OS is an implementation of the POSIX.13 minimal real-time system profile, and as such it provides to C/C++ applications the services defined in the standard, which can be grouped as:

- Concurrency services supporting the management of threads.
- Scheduling with real-time policies based on preemptive fixed priorities and supporting variants such as FIFO within priorities, round robin within priorities, or the Sporadic Server.
- Synchronization through counting semaphores, mutexes, and condition variables. Mutexes have support for real-time mutual exclusion through the priority inheritance or priority ceiling protocols.
- Signals, as an asynchronous notification mechanism.
- Time management though clocks and timers. A monotonic clock that cannot have backward jumps is provided for real-time applications. Timers can be created to measure the passage of an absolute or relative time and will generate a signal to notify the application about their expiration.
- Execution-time clocks and timers are used to monitor tasks execution time and enforce time bounds under resource reservations.
- Dynamic memory management. MaRTE OS uses the TLSF[16] algorithm, which is a fast time-bounded dynamic memory allocator with low fragmentation.
- Device I/O through a simplified device name space and the standard operations, like open/close/read/write/ioctl.

In addition to the POSIX services, MaRTE OS also provides extensions that are useful to develop advanced real-time applications:

- Timed handlers, as a lightweight mechanism to define small handlers that are executed in interrupt context at the expiration of a timer. These handlers can also be used in conjunction with execution-time clocks.
- Earliest Deadline First (EDF) scheduling. This is a preemptive dynamic-priority thread dispatching policy that can be used to maximize the resource utilization in real-time applications. It requires an implementation of the Stack Resource Protocol [Bak91] for avoiding unbounded blocking effects when accessing protected objects.
- Thread sets or groups that can be used to create clocks for measuring the execution time of a group of threads. These clocks can in turn be used to create

[16] TLSF: Memory Allocator for Real-Time: http://rtportal.upv.es/rtmalloc/.

timers and timed handlers to implement advanced scheduling policies or to detect and bound the effects of timing violations by a group of threads.
- Interrupt management with the ability to install interrupt handlers and manage interrupt masking. In MaRTE OS there is separate accounting for the execution time of interrupt handlers.
- Application-defined scheduling [RH01, RH04], which is a group of services intended to allow an application to install its own scheduler for the OS threads. This is particularly interesting to implement advanced scheduling policies being defined by the real-time research community.

13.4.3.2 Application-Defined Scheduling

MARTE OS allows Ada and C++ applications to define its own scheduling algorithm [RH01, RH04] in a way compatible with the scheduler interface defined in POSIX and in the Ada 2005 Real-Time Systems Annex. Several application-defined schedulers, implemented as special user tasks, can coexist in the system in a predictable way.

Each application scheduler is responsible for scheduling a set of tasks that have previously been attached to it. The scheduler is typically a task that executes a loop where it waits for scheduling events and then determines the application task(s) to be activated or suspended. A scheduling event is generated every time a task requests attachment to the scheduler or terminates, gets ready or blocks, invokes a yield operation, changes its scheduling parameters, inherits or uninherits a priority, or executes any operation on an application-scheduled mutex.

The application scheduler can make use of the regular operating system services, including the high-resolution timers to program future events and the execution time clock and timers to impose execution-time budgets on the different threads and to simplify the implementation of scheduling algorithms such as the Sporadic Server or the Constant Bandwidth Server.

Because mutexes may cause priority inversion, it is necessary that the scheduler task knows about the use of mutexes to establish its own protocols. Two types of mutexes are considered:

- System-scheduled mutexes, created with the current POSIX protocols and scheduled by the system.
- Application-scheduled mutexes, whose protocol will be defined by the application scheduler.

13.4.3.3 Interrupt Management at Application Level

MARTE offers to Ada and C programmers an application program interface (API) that allows dealing with hardware interrupts in an easy way. Basically, this API offers operations to:

- Enable and disable hardware interrupts;
- Install interrupts procedures (several can be installed for the same interrupt number);
- Synchronize threads with the hardware interrupts.

Semaphores can be used as an alternative synchronization mechanism between interrupt handlers and tasks.

13.4.3.4 Drivers Framework

MARTE includes a driver framework that simplifies and standardizes installation and use of drivers, allowing programmers to share their drivers with other people in a simple way. This framework also facilitates the adaptation of drivers written for open operating systems (such as Linux).

The implementation model is similar to what is used in most UNIX-like operating systems. Applications access devices through "device files" using standard file operations (open, close, write, read, ioctl). In this case the device files are stored as a set of data tables inside the kernel, with pointers to the driver primitive operations.

13.4.3.5 Ada Support

The runtime of the GNAT Ada compiler has been adapted to run on the POSIX interface provided by MaRTE OS. Ada applications running on top of MaRTE OS can use the full basic Ada language functionality. MaRTE OS also supports most of the services defined in the Ada 2005 Reference Manual real-time annex:

- Timing events.
- Execution-time clocks and timers.
- Group execution-time budgets.
- Dynamic priorities for protected objects.
- EDF and round-robin scheduling policies.
- Priority-specific dispatching.

13.5 Development Tools

The implementation of complex real-time applications requires the use of specific tools for analyzing and verifying the behavior of a system. In addition to the general programming tools, such as editors, compilers, source code browsers, debuggers, and version control systems, there are a number of tools specifically aimed at cross development and runtime analysis. In particular:

- *Memory analyzers* show memory usage and reveal memory leaks before they cause a system failure.
- *Performance profilers* reveal code performance bottlenecks and show where a CPU is spending its cycles, providing a detailed function-by-function analysis.
- *Real-time monitors* allow the programmer to store or view any set of variables, while the program is running.
- *Execution tracers* display the function calls and function calling parameters of a running program, as well as return values and actual execution time.
- *Event analyzers* allow the programmer to view and track application events in a graphical viewer with stretchable time scale, showing tasks, context switches, semaphores, message queues, signals, timers, etc.
- *Timing analysis tools* perform a static analysis of the task code and derive a set of data used to verify the timing behavior of the application.
- *Schedulability analyzers* verify the feasibility of the schedule produced by a scheduling algorithm on a specific task set.
- *Scheduling simulators* simulate the execution behavior of specific scheduling algorithms on synthetic task sets with randomly generated parameters and desired workload. They are useful for testing the timing behavior of the system in several circumstances, to detect possible deadline misses, blocking conditions and task response times.

In the following, we give a brief overview of tools for timing analysis, schedulability analysis, and scheduling simulations.

13.5.1 Timing Analysis Tools

Obtaining accurate information about the longest execution time a software component can take to run on a specific platform is a key feature for ensuring that an embedded real-time system will operate correctly.

These tools perform a static analysis of the task code (some at the source level, some at the executable level) to determine a set of data that are essential to verify the timing behavior of a real-time application. Examples of such data include task worst-case execution times (WCETs), cache-related preemption delays (CRPDs), and stack usage profiles. The following are two commercial tools in this category:

- **RapiTime**[17] is a timing analysis tool, developed by Rapita Systems Ltd, targeted at real-time embedded applications. RapiTime collects execution traces and derives execution time measurement statistics to help the programmer in estimating tasks' worst-case execution times.

[17] RapiTime: http://www.rapitasystems.com/rapitime.

- **aiT**[18] is a WCET analyzer that statically computes tight bounds for the task WCETs in real-time systems. It directly analyzes binary executables taking the intrinsic cache and pipeline behavior into account. It was developed by Absint in the DAEDALUS European project, according to the requirements of Airbus France for validating the timing behavior of critical avionics software, including the flight control software of the A380 aircraft. A graphical user interface supports the visualization of the worst-case program path and the interactive inspection of all pipeline and cache states at arbitrary program points.

13.5.2 Schedulability Analysis

This type of tool allows designers to test software models against various design scenarios and evaluate how different implementations might optimize the performance of the system, isolating and identifying potential scheduling bottlenecks. Some of commercial schedulability analysis tools are reported below.

- **RTDruid**[19] is the development environment for ERIKA Enterprise, helping the programmer to write, compile, and analyze real-time applications in a comfortable environment. RT-Druid is composed by a set of plugins for the Eclipse framework, including the code generator (implementing the OIL language compiler) and a schedulability analyzer for verifying the feasibility of real-time applications and estimating worst-case response times of real-time tasks under different scheduling algorithms. RT-Druid also includes importers/exporters for AUTOSAR XML specifications produced by ARTOP[20] and dSpace SystemDesk. It also includes integration with tools like Papyrus UML, Absint aiT, and Lauterbach Trace32.
- **TimeWiz**[21] is an integrated design environment for building predictable embedded, real-time systems. It allows representing, analyzing, and simulating real-time systems. It works for the simplest micro-controller with a mini-executive up to a large distributed system with tens or even hundreds of processors.
- **symTA/S**[22] is a scheduling analysis tool suite used for budgeting, scheduling verification and optimization for processors, electronic control units (ECUs), communication buses, networks, and complete integrated systems. It enables end-to-end timing analysis, visualization, and optimization for distributed systems. In automotive electronics, SymTA/S supports standards such as OSEK, AUTOSAR-OS, CAN, and FlexRay.

[18] aiT: http://www.absint.com/ait/.

[19] RTDruid: http://erika.tuxfamily.org/.

[20] ARTOP: http://www.artop.org.

[21] TimeWiz: http://www.embeddedtechnology.com/product.mvc/TimeWiz-0001.

[22] symTA/S: http://www.symtavision.com/symtas.html.

- **chronVAL**[23] is a real-time analysis tool to analyze, optimize, and validate single and distributed embedded systems with regard to worst-case scenarios. The tool enables designers to analyze the dynamic behavior of embedded software and bus communication, including multiprocessor configurations. Key features include mathematical analysis of the real-time behavior, verification of application deadlines, validation of maximum message latencies, evaluation of system performance, and graphical visualization of results.

13.5.3 Scheduling Simulators

These tools allow generating the schedule produced by given scheduling algorithm on synthetic task sets. Some of the existing tools are reported below.

- **RTSim**[24] (Real-Time system SIMulator) is a collection of programming libraries written in C++ for simulating real-time control systems. RTSim has been developed at Retis Lab of the Scuola Superiore Sant'Anna of Pisa (Italy) as an internal project. It has been primarily used for testing the performance of new scheduling algorithms under different workload conditions. For these reasons, it contains, already implemented, most of the real-time scheduling algorithms developed in the real-time community. The tool is released as open source to let other researchers play with the simulator and build a shared simulation platform for comparing the performance of new scheduling algorithms. RTSim is currently compatible with many systems, including several distributions of Linux, recent FreeBSD, Mac OS X, and Cygwin.
- **TrueTime**[25] is a Matlab/Simulink-based simulator developed at the University of Lund (Sweden) for testing the behavior of real-time distributed control systems. TrueTime facilitates co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics. Features of the tool include the simulation of external interrupts, or the possibility to call Simulink block diagrams, including network blocks (Ethernet, CAN, TDMA, FDMA, round robin, Switched Ethernet, FlexRay, and PROFINET), wireless network blocks (802.11b WLAN and 802.15.4 ZigBee), and battery-powered devices using dynamic voltage scaling.
- **chronSIM**[26] is a tool that allows engineers to perform real-time simulation, analysis, and forecast of embedded software dynamic performance. It allows creating the task structure, interrupt service routines scheme and scheduling procedure to maximize data throughput, and comply with all specified response

[23] chronVAL: http://www.inchron.com/.

[24] RTSim: http://rtsim.sssup.it/.

[25] TrueTime: http://www3.control.lth.se/truetime/.

[26] chronSIM: http://www.inchron.com/chronsim.html.

times. chronSIM uncovers and visualizes hidden dynamic operating sequences using state-of-the-art UML-based diagrams (sequence, task, state, stack, processor load, function nesting, etc.) It allows carrying out Monte Carlo simulations and stress tests through random variations of the event times.

Chapter 14
Solutions to the Exercises

14.1 Solutions for Chap. 1

1.1 Fast computing tends to minimize the average response time of computation activities, whereas real-time computing is required to guarantee the timing constraints of each task.

1.2 The main limitations of the current real-time kernels are mainly due to the fact that they are developed to minimize runtime overhead (hence functionality) rather than offering support for a predictable execution. For example, short interrupt latency is good for servicing I/O devices, but introduces unpredictable delays in task execution for the high priority given to the interrupt handlers. Scheduling is mostly based on fixed priority and explicit timing constraints cannot be specified on tasks. No specific support is usually provided for periodic tasks and no aperiodic service mechanism is available for handling event-driven activities. Access to shared resources is often realized through classical semaphores, which are efficient, but prone to priority inversion if no protocol is implemented for entering critical sections. Finally, no temporal protection or resource reservation mechanism is usually available in current real-time kernels for coping with transient overload conditions, so a task executing too much may introduce unbounded delays on the other tasks.

1.3 A real-time kernel should allow the user to specify explicit timing constraints on application tasks and support a predictable execution of real-time activities with specific real-time mechanisms, including scheduling, resource management, synchronization, communication, and interrupt handling. In critical real-time systems, predictability is more important than high performance, and often an increased functionality can only be reached at the expense of a higher runtime overhead. Other important features that a real-time system should have include maintainability, fault tolerance, and overload management.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3_14

1.4 Three approaches can be used. The first one is to disable all external interrupts, letting application tasks to access peripheral devices through polling. This solution gives great programming flexibility and reduces unbounded delays caused by the driver execution, but it is characterized by a low processor efficiency on I/O operations, due to the busy wait.
A second solution is to disable interrupts and handle I/O devices by polling through a dedicated periodic kernel routine, whose load can be taken into account through a specific utilization factor. As in the previous solution, the major problem of this approach is due to the busy wait, but the advantage is that all hardware details can be encapsulated into a kernel routine and do not need to be known to the application tasks. An additional overhead is due to the extra communication required among application tasks and the kernel routine for exchanging I/O data.
A third approach is to enable interrupts but limit the execution of interrupt handlers as much as possible. In this solution, the interrupt handler activates a device handler, which is a dedicated task that is scheduled (and guaranteed) by the kernel as any other application task. This solution is efficient and minimizes the interference caused by interrupts.

1.5 The restrictions that should be used in a programming language to permit the analysis of real-time applications should limit the variability of execution times. Hence, a programmer should avoid using dynamic data structures, recursion, and all high-level constructs that make execution time unpredictable. Possible language extensions should be aimed at facilitating the estimation of worst-case execution times. For example, a language could allow the programmer to specify the maximum number of iterations in each loop construct and the probability of taking a branch in conditional statements.

14.2 Solutions for Chap. 2

2.1 A schedule is formally defined as a step function $\sigma : \mathbf{R}^+ \rightarrow \mathbf{N}$ such that $\forall t \in \mathbf{R}^+, \exists t_1, t_2$ such that $t \in [t_1, t_2)$ and $\forall t' \in [t_1, t_2)\ \sigma(t) = \sigma(t')$. For any $k > 0$, $\sigma(t) = k$ means that task J_k is executing at time t, while $\sigma(t) = 0$ means that the CPU is idle. A schedule is said to be *preemptive* if the running task can be arbitrarily suspended at any time to assign the CPU to another task according to a predefined scheduling policy. In a preemptive schedule, tasks may be executed in disjointed interval of times. In a non-preemptive schedule, a running task cannot be interrupted, and therefore, it proceeds until completion.

2.2 A periodic task consists of an infinite sequence of identical jobs that are regularly activated at a constant rate. If ϕ_i is the activation time of the first job

of task τ_i, the activation time of the kth job is given by $\phi_i + (k-1)T_i$, where T_i is the task period. Aperiodic tasks also consist of an infinite sequence of identical jobs; however, their activations are not regular. An aperiodic task where consecutive jobs are separated by a minimum interarrival time is called a sporadic task. The most important timing parameters defined for a real-time task are as follows:

- The *arrival time* (or *release time*), that is, the time at which a task becomes ready for execution
- The *computation time*, that is, the time needed by the processor for executing the task without interruption
- The *absolute deadline*, that is, the time before which a task should be completed to avoid damage to the system
- The *finishing time*, that is, the time at which a task finishes its execution
- The *response time*, that is, the difference between the finishing time and the release time: $R_i = f_i - r_i$

2.3 A real-time application consisting of tasks with precedence relations is shown in Sect. 2.2.2.

2.4 A static scheduler is one in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation. In a dynamic scheduler, scheduling decisions are based on dynamic parameters that may change during system evolution. A scheduler is said to be *offline* if it is precomputed (before task activation) and stored in a table. In an online scheduler, scheduling decisions are taken at runtime when a new task enters the system or when a running task terminates. An algorithm is said to be *optimal* if it minimizes some given cost function defined over the task set. A common optimality criterion for a real-time system is related to feasibility. Then, a scheduler is optimal whenever it can find a feasible schedule, if there exists one. Heuristic schedulers use a heuristic function to search for a feasible schedule; hence, it is not guaranteed that a feasible solution is found.

2.5 An example of domino effect is shown in Fig. 2.15.

14.3 Solutions for Chap. 3

3.1 To check whether the EDD algorithm produces a feasible schedule, tasks must be ordered with increasing deadlines, as shown in Table 14.1:
Then applying Eq. (3.1) we have

Table 14.1 Task set ordered by deadline

	J'_1	J'_2	J'_3	J'_4
C'_i	2	4	3	5
D'_i	5	9	10	16

$$f_1' = C_1' = 2$$
$$f_2' = f_1' + C_2' = 6$$
$$f_3' = f_2' + C_3' = 9$$
$$f_4' = f_3' + C_4' = 14$$

Since each finishing time is less than the corresponding deadline, the task set is schedulable by EDD.

3.2 The algorithm for finding the maximum lateness of a task set scheduled by the EDD algorithm is shown in Fig. 14.1.

3.3 The scheduling tree constructed by Bratley's algorithm for the following set of non-preemptive tasks is illustrated in Fig. 14.2.

3.4 The schedule found by the Spring algorithm on the scheduling tree developed in the previous exercise with the heuristic function $H = a + C + D$ is $\{J_2, J_4, J_3, J_1\}$ which is unfeasible, since J_3 and J_4 miss their deadlines. Noticed that the feasible solution is found with $H = a + d$.

Fig. 14.1 Code of the `init` function

```
Algorithm: EDD_Lmax(J)
{
        L_max = -D_n;
        f_0 = 0;
        for (each J_i ∈ J) {
            f_i = f_{i-1} + C_i;
            L_i = f_i + D_i;
            if (L_i > L_max) L_max = L_i;
        }
        return(L_max);
}
```

Fig. 14.2 Scheduling tree constructed by Bratley's algorithm for the task set shown in Table 14.2

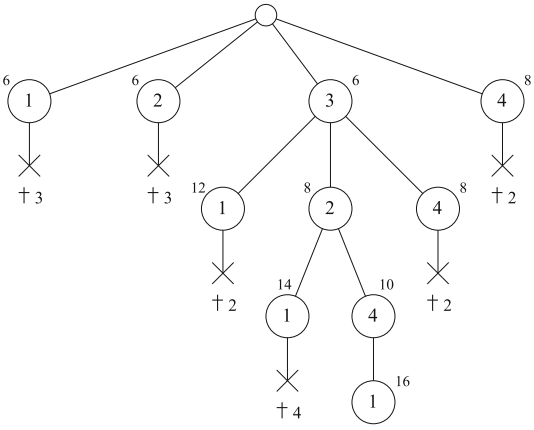


Table 14.2 Task set parameters for Bratley's algorithm

	J_1	J_2	J_3	J_4
a_i	0	4	2	6
C_i	6	2	4	2
D_i	18	8	9	10

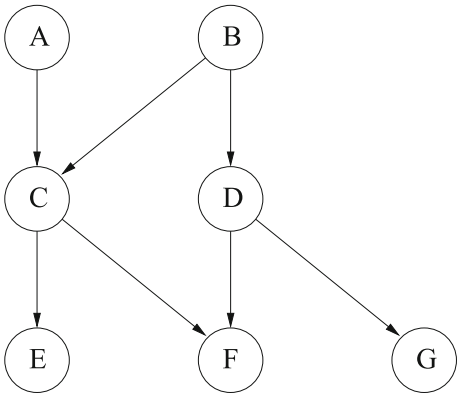


Fig. 14.3 Precedence graph for Exercise 3.3

Table 14.3 Task set parameters modified by Chetto and Chetto's algorithm

	C_i	r_i	$r*_i$	d_i	$d*_i$
A	2	0	0	25	20
B	3	0	0	25	15
C	3	0	3	25	23
D	5	0	3	25	20
E	1	0	6	25	25
F	2	0	8	25	25
G	5	0	8	25	25

3.5 The precedence graph is shown in Fig. 14.3.
Applying the transformation algorithm by Chetto and Chetto we get the parameters shown in Table 14.3.
So the schedule produced by EDF will be $\{B, A, D, C, E, F, G\}$.

14.4 Solutions for Chap. 4

4.1 The processor utilization factor of the task set is

$$U = \frac{2}{6} + \frac{2}{8} + \frac{2}{12} = 0.75$$

and considering that for three tasks the utilization least upper bound is

$$U_{lub}(3) = 3(2^{1/3} - 1) \simeq 0.78$$

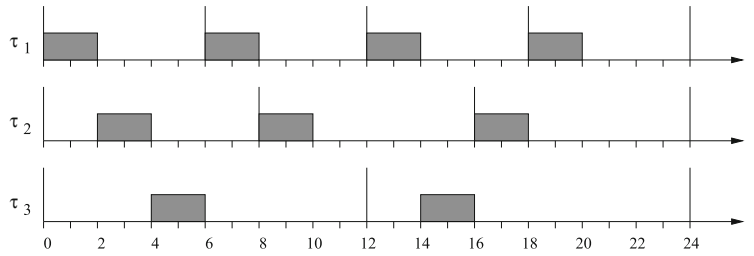


Fig. 14.4 Schedule produced by rate monotonic for the task set of Exercise 4.1

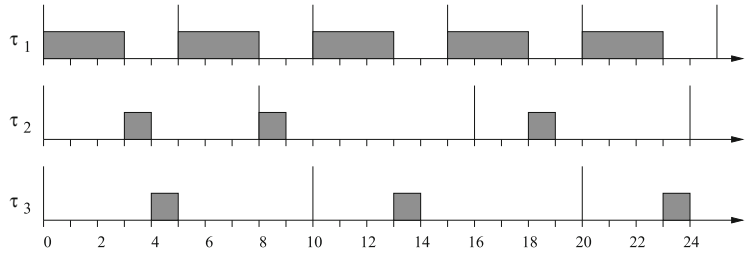


Fig. 14.5 Schedule produced by rate monotonic for the task set of Exercise 4.2

from the Liu and Layland test, since $U \leq U_{lub}$, we can conclude that the task set is schedulable by RM, as shown in Fig. 14.4.

4.2 The processor utilization factor of the task set is

$$U = \frac{3}{5} + \frac{1}{8} + \frac{1}{10} = 0.825$$

which is greater than $U_{lub}(3)$. Hence, we cannot verify the feasibility with the Liu and Layland test. Using the hyperbolic bound, we have that

$$\prod_{i=1}^{n}(U_i + 1) = 1.98$$

which is less than 2. Hence, we can conclude that the task set is schedulable by RM, as shown in Fig. 14.5.

4.3 Applying the Liu and Layland test we have that

$$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{10} = 0.88 > 0.78$$

so we cannot say anything. With the hyperbolic bound we have that

$$\prod_{i=1}^{n}(U_i + 1) = 2.16 > 2$$

so we cannot say anything. Applying the response time analysis we have to compute the response times and verify that they are less than or equal to the relative deadlines (which in this case are equal to periods). Hence, we have

$$R_1 = C_1 = 1$$

So τ_1 does not miss its deadline. For τ_2 we have

$$R_2^{(0)} = \sum_{j=1}^{2} C_j = C_1 + C_2 = 3$$

$$R_2^{(1)} = C_2 + \left\lceil \frac{R_2^{(0)}}{T_1} \right\rceil C_1 = 2 + \left\lceil \frac{3}{4} \right\rceil 1 = 3$$

So $R_2 = 3$, meaning that τ_2 does not miss its deadline. For τ_3 we have

$$R_3^{(0)} = \sum_{j=1}^{3} C_j = C_1 + C_2 + C_3 = 6$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{R_3^{(0)}}{T_1} \right\rceil C_1 + \left\lceil \frac{R_3^{(0)}}{T_2} \right\rceil C_2 = 2 + \left\lceil \frac{6}{4} \right\rceil 1 + \left\lceil \frac{6}{6} \right\rceil 2 = 7$$

$$R_3^{(2)} = 2 + \left\lceil \frac{7}{4} \right\rceil 1 + \left\lceil \frac{7}{6} \right\rceil 2 = 9$$

$$R_3^{(3)} = 2 + \left\lceil \frac{9}{4} \right\rceil 1 + \left\lceil \frac{9}{6} \right\rceil 2 = 10$$

$$R_3^{(4)} = 2 + \left\lceil \frac{10}{4} \right\rceil 1 + \left\lceil \frac{10}{6} \right\rceil 2 = 10$$

So $R_3 = 10$, meaning that τ_3 does not miss its deadline. Hence, we can conclude that the task set is schedulable by RM, as shown in Fig. 14.6.

4.4 Applying the response time analysis, we can easily verify that $R_3 = 10$ (see the solution of the previous exercise); hence, the task set is not schedulable by RM.

4.5 Since

$$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.96 < 1$$

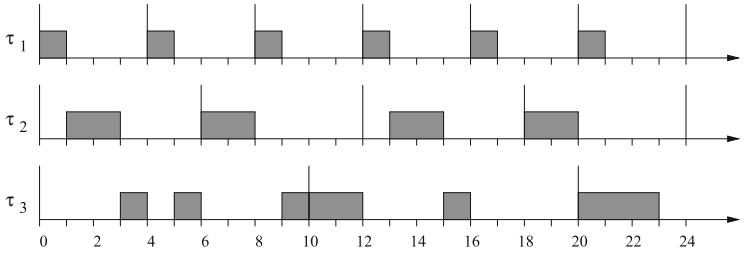


Fig. 14.6 Schedule produced by rate monotonic for the task set of Exercise 4.3

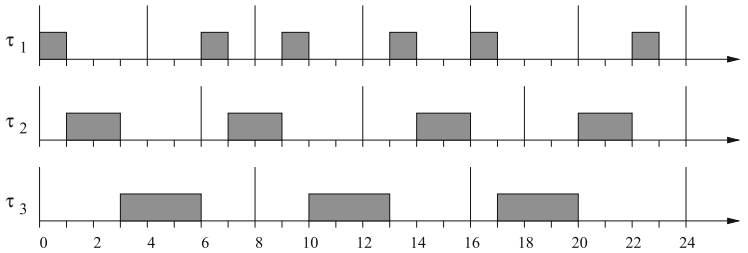


Fig. 14.7 Schedule produced by EDF for the task set of Exercise 4.5

the task set is schedulable by EDF, as shown in Fig. 14.7.

4.6 Applying the processor demand criterion, we have to verify that

$$\forall L \in \mathcal{D} \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \leq L.$$

where

$$\mathcal{D} = \{d_k \mid d_k \leq \min(L^*, H)\}.$$

For the specific example, we have

$$U = \frac{2}{6} + \frac{2}{8} + \frac{4}{12} = \frac{11}{12}$$
$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} = 32$$
$$H = \text{lcm}(6, 8, 12) = 24.$$

Hence, the set of checking points is given by $\mathcal{D} = \{4, 5, 8, 11, 12, 17, 20, 23\}$. Since the demand in these intervals is $\{2, 4, 8, 10, 12, 14, 20, 22\}$, we can

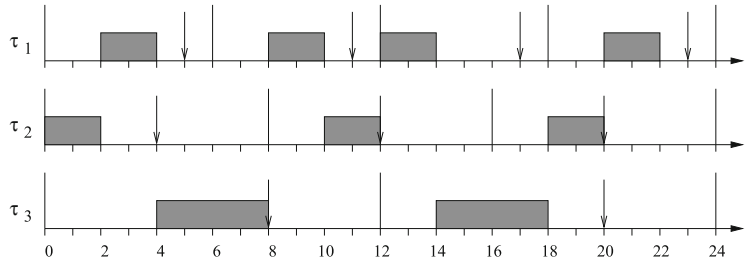


Fig. 14.8 Schedule produced by EDF for the task set of Exercise 4.6

conclude that the task set is schedulable by EDF. The resulting schedule is shown in Fig. 14.8.

4.7 Applying the response time analysis, we have to start by computing the response time of task τ_2, which is the one with the shortest relative deadline and hence the highest priority:

$$R_2 = C_2 = 2.$$

So τ_2 does not miss its deadline. For τ_1 we have

$$R_1^{(0)} = \sum_{j=1}^{2} C_j = C_1 + C_2 = 4$$

$$R_1^{(1)} = C_1 + \left\lceil \frac{R_1^{(0)}}{T_2} \right\rceil C_2 = 2 + \left\lceil \frac{4}{8} \right\rceil 2 = 4$$

So $R_1 = 4$, meaning that τ_1 does not miss its deadline. For τ_3 we have

$$R_3^{(0)} = \sum_{j=1}^{3} C_j = C_1 + C_2 + C_3 = 8$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{R_3^{(0)}}{T_2} \right\rceil C_2 + \left\lceil \frac{R_3^{(0)}}{T_1} \right\rceil C_1 = 4 + \left\lceil \frac{8}{8} \right\rceil 2 + \left\lceil \frac{8}{6} \right\rceil 2 = 10$$

And since $R_3^{(1)} > D_3$, we can conclude that the task set is not schedulable by DM. The resulting schedule is shown in Fig. 14.9.

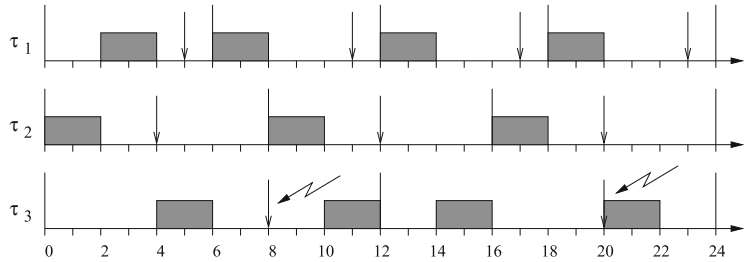


Fig. 14.9 Schedule produced by deadline monotonic for the task set of Exercise 4.7.

14.5 Solutions for Chap. 5

5.1 The maximum sporadic server size can be computed by Eq. (5.24):

$$U_{SS}^{max} = \frac{2 - P}{P}$$

where

$$P = \prod_{i=1}^{n} (U_i + 1) = \frac{7}{6} \cdot \frac{9}{7} = \frac{3}{2}.$$

Hence, substituting the value of P into Eq. (5.24) we have

$$U_{SS}^{max} = \frac{1}{3}.$$

To enhance aperiodic responsiveness, the server must run at the highest priority, and this can be achieved by setting its period to $T_s = T_1 = 6$. Then, assuming $U_s = U_{SS}^{max}$, its capacity will be $C_s = U_s T_s = 2$.

5.2 The maximum deferrable server size can be computed by Eq. (5.15). Hence,

$$U_{DS}^{max} = \frac{2 - P}{2P - 1}.$$

And substituting the value of $P = 3/2$ into Eq. (5.15) we have

$$U_{DS}^{max} = \frac{1}{4}.$$

Hence, by setting $U_s = U_{DS}^{max}$ and $T_s = T_1 = 6$, the capacity will be $C_s = U_s T_s = 6/4 = 1.5$.

5.3 Following the same steps reported in Exercise 5.5.1, we know that the maximum utilization that can be assigned to a polling server to guarantee

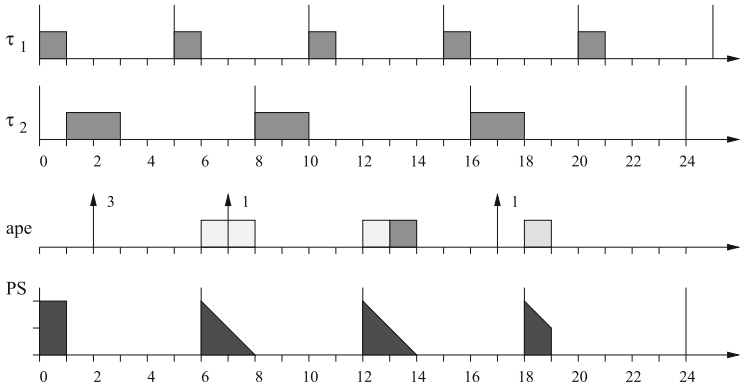


Fig. 14.10 Schedule produced by rate monotonic and polling server for the task set of Exercise 5.3

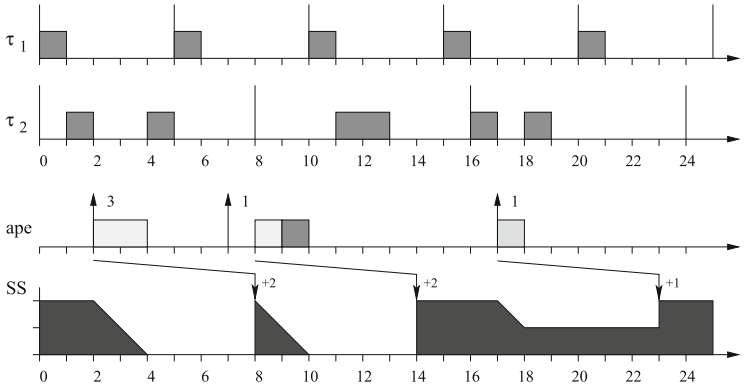


Fig. 14.11 Schedule produced by rate monotonic and sporadic server for the task set of Exercise 5.4

the periodic task set is

$$U_{PS}^{max} = \frac{2-P}{P} = \frac{1}{3}.$$

So, by setting $T_s = 6$ (intermediate priority) and $C_s = 2$, we satisfy the constraints. The resulting schedule is illustrated in Fig. 14.10.

5.4 A sporadic server can be guaranteed with the same method used for the polling server. So, using the same parameters computed before ($C_s = 2$ and $T_s = 6$), we have the schedule shown in Fig. 14.11.

5.5 Applying Eq. (5.15) to the considered task set, we see that the maximum utilization that can be assigned to a deferrable server to guarantee the periodic task set is $U_{s_{max}} = 1/4$. So, by setting $T_s = 4$ (maximum priority)

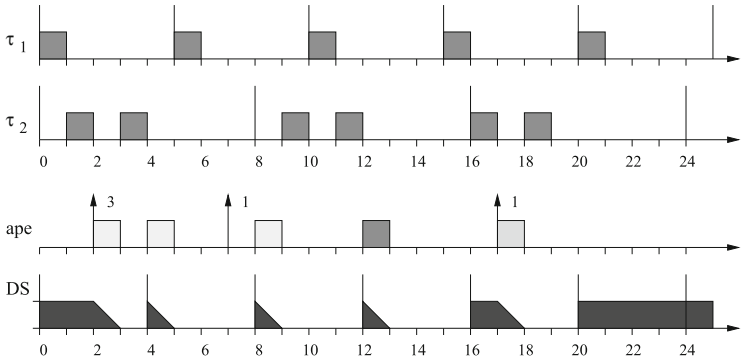


Fig. 14.12 Schedule produced by rate monotonic and deferrable server for the task set of Exercise 5.5

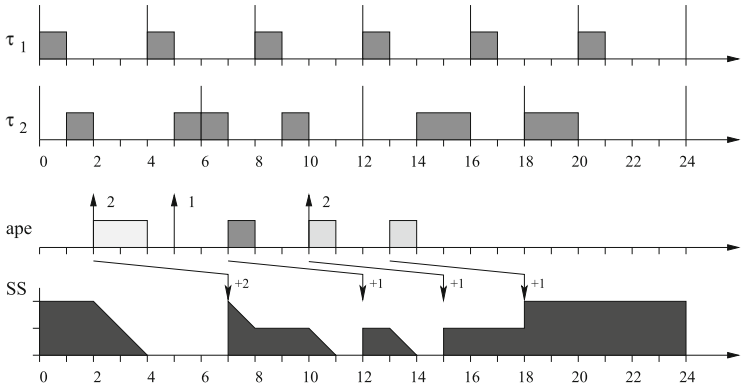


Fig. 14.13 Schedule produced by rate monotonic and sporadic server for the task set of Exercise 5.6

and $C_s = 1$, we satisfy the constraints. The resulting schedule is illustrated in Fig. 14.12.

5.6 The resulting schedule is illustrated in Fig. 14.13.

14.6 Solutions for Chap. 6

6.1 For any dynamic server we must have $U_p + U_s \leq 1$; hence, considering that $U_p = 2/3$, the maximum server utilization that can be assigned to a dynamic sporadic server is

$$U_s = 1 - U_p = 1/3.$$

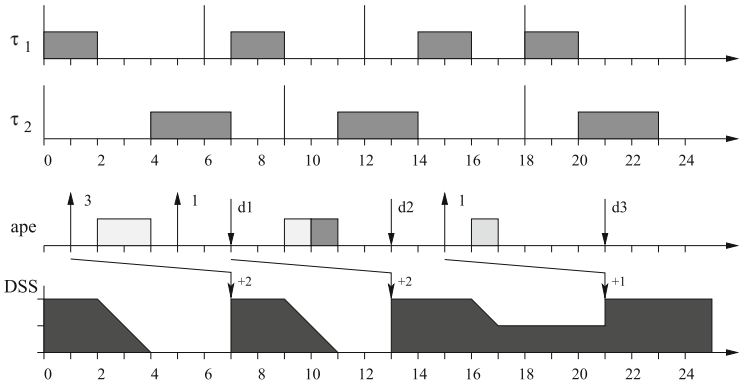


Fig. 14.14 Schedule produced by EDF + DDS for the task set of Exercise 6.2

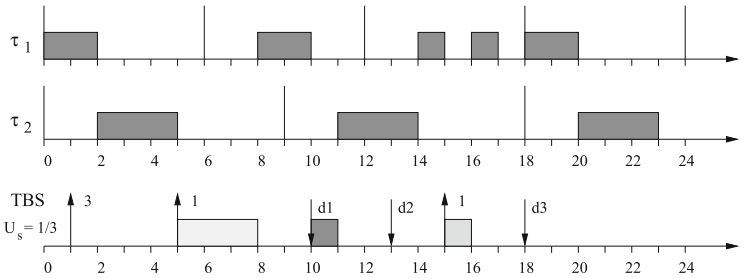


Fig. 14.15 Schedule produced by EDF + TBS for the task set of Exercise 6.3

6.2 The deadlines computed by the server for the aperiodic jobs result: $d_1 = a_1 + T_s = 7$, $d_2 = d_1 + T_s = 13$, and $d_3 = a_3 + T_s = 21$. The resulting schedule produced by EDF + DSS is illustrated in Fig. 14.14.

6.3 The deadlines computed by the server for the aperiodic jobs are as follows: $d_1 = a_1 + C_1/U_s = 10$, $d_2 = d_1 + C_2/U_s = 13$, and $d_3 = a_3 + C_3/U_s = 18$. The resulting schedule produced by EDF + TBS is illustrated in Fig. 14.15.

6.4 The events handled by the CBS are as follows:

Time	Event	Action
$t = 1$	Arrival	$c_s = Q_s, d_s = a_1 + T_s = 7$
$t = 4$	$c_s = 0$	$c_s = Q_s, d_s = d_s + T_s = 13$
$t = 5$	Arrival	Enqueue request
$t = 11$	$c_s = 0$	$c_s = Q_s, d_s = d_s + T_s = 19$
$t = 15$	Arrival	$c_s = Q_s, d_s = a_3 + T_s = 21$

The resulting schedule produced by EDF + CBS is illustrated in Fig. 14.16.

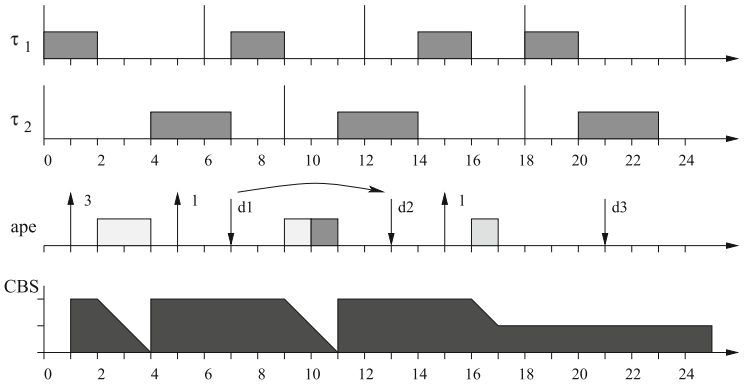


Fig. 14.16 Schedule produced by EDF + CBS for the task set of Exercise 6.4

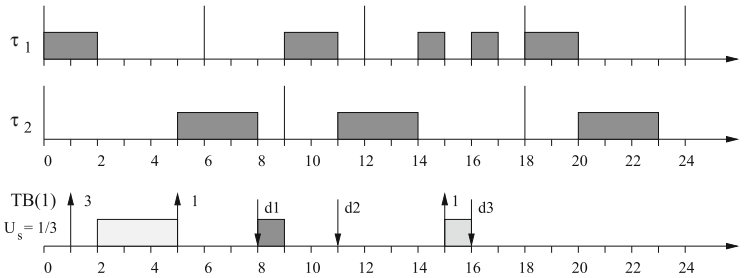


Fig. 14.17 Schedule produced by EDF + TB(1) for the task set of Exercise 6.5

6.5 The deadlines computed by the server are

$$d_1^{(0)} = a_1 + C_1/U_s = 10$$
$$d_1^{(1)} = f_1^{(0)} = 8$$

$$d_2^{(0)} = d_2^{(0)} + C_2/U_s = 13$$
$$d_2^{(1)} = f_2^{(0)} = 11$$

$$d_3^{(0)} = a_3 + C_3/U_s = 18$$
$$d_3^{(1)} = f_3^{(0)} = 16$$

The resulting schedule produced by EDF + TB(1) is illustrated in Fig. 14.17.

6.6 The deadlines computed by the server are

$$d_1^{(0)} = a_1 + C_1/U_s = 10$$

$$d_1^{(1)} = f_1^{(0)} = 8$$
$$d_1^{(2)} = f_1^{(1)} = 5$$
$$d_1^{(3)} = f_1^{(2)} = 4$$

$$d_2^{(0)} = d_2^{(0)} + C_2/U_s = 13$$
$$d_2^{(1)} = f_2^{(0)} = 11$$
$$d_2^{(2)} = f_2^{(1)} = 9$$
$$d_2^{(3)} = f_2^{(2)} = 6$$

$$d_3^{(0)} = a_3 + C_3/U_s = 18$$
$$d_3^{(1)} = f_3^{(0)} = 16$$

The resulting schedule produced by EDF + TB* is illustrated in Fig. 14.18.

6.7 The resulting schedule is illustrated in Fig. 14.19.

6.8 First of all, the utilization of the periodic task set is

$$U_p = \frac{8}{20} + \frac{6}{30} = \frac{3}{5} = 0.6$$

hence, the largest utilization that can be assigned to a CBS is $U_s = 1 - U_p = 0.4$. Then, converting all times in microseconds and substituting the values in Eq. (6.15) we obtain

$$C^{\text{avg}} = 1200\ \mu s$$
$$T_s = \frac{10}{4}\left(20 + \sqrt{\frac{20 \cdot 1200}{0.6}}\right) = 550\,\mu s$$
$$Q_s = T_s U_s = 220\,\mu s.$$

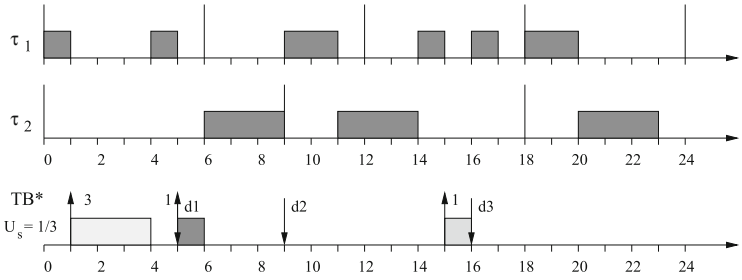


Fig. 14.18 Schedule produced by EDF + TB* for the task set of Exercise 6.6

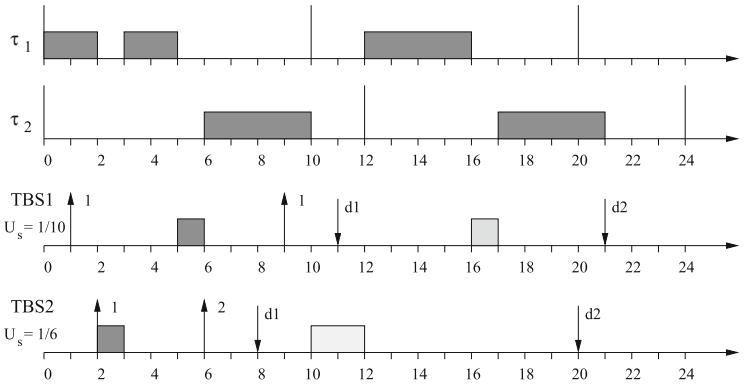


Fig. 14.19 Schedule produced by EDF + $TB_1 + TB_2$ for the task set of Exercise 6.7

14.7 Solutions for Chap. 7

7.1 Applying Eq. (7.19), we verify that

$$\forall i, \quad 1 \leq i \leq n, \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1).$$

So we have

$$\frac{C_1 + B_1}{T_1} = \frac{9}{10} < 1$$

$$\frac{C_1}{T_1} + \frac{C_2 + B_2}{T_2} = \frac{4}{10} + \frac{6}{15} = 0.8 < 0.83$$

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = \frac{4}{10} + \frac{3}{15} + \frac{4}{20} = 0.8 > 0.78$$

So we cannot say anything about feasibility. By applying the response time analysis, we have to verify that

$$\forall i, \quad 1 \leq i \leq n, \quad R_i \leq D_i$$

where

$$R_i = C_i + B_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

So we have

$$R_1 = C_1 + B_1 \;=\; 9 < 10$$

$$R_2^{(0)} = C_1 + C_2 + B_2 = 10$$

$$R_2^{(1)} = C_2 + B_2 + \left\lceil \frac{10}{10} \right\rceil 4 = 10 \;<\; 15$$

$$R_3^{(0)} = C_1 + C_2 + C_3 = 11$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{11}{10} \right\rceil 4 + \left\lceil \frac{11}{15} \right\rceil 3 = 15$$

$$R_3^{(2)} = C_3 + \left\lceil \frac{15}{10} \right\rceil 4 + \left\lceil \frac{15}{15} \right\rceil 3 = 15 \;<\; 20$$

Hence, we can conclude that the task set is schedulable by RM.

7.2 Using the Priority Inheritance Protocol, a task τ_i can be blocked at most for one critical section by each lower-priority task. Moreover, a critical section can block τ_i only if it belongs to a task with lower priority and it is shared with τ_i (direct blocking) or with higher-priority tasks (push-through blocking). Finally, we have to consider that two critical sections cannot block a task if they are protected by the same semaphore or they belong to the same task. Hence, if X_i denotes the critical section X belonging to task τ_i we have that

- Task τ_1 can only experience direct blocking (since there are no tasks with higher priority) and the set of critical sections that can potentially block it is $\{A_2, A_3, C_3\}$. It can be blocked at most for the duration of two critical sections in this set. Thus, the maximum blocking time is given by the sum of the two longest critical sections in this set. In this case, however, note that the longest critical sections are A_3 and C_4, which belong to the same task; hence, they cannot be selected together. Hence, the maximum blocking time for τ_1 is $B_0 = d_{(A_2)} + d_{(I_3)} = 7$.
- Task τ_2 can experience direct blocking on A_3 and B_3 and push-through blocking on A_3 and C_3. Hence, the set of critical sections that can potentially block τ_2 is $\{A_3, B_3, C_4\}$. It can be blocked at most for the duration of one critical section in this set. Thus, the maximum blocking time is given by the longest critical section in this set, that is, C_3. Hence, we have $B_2 = d_{(C_3)} = 5$.
- Task τ_3 cannot be blocked, because it is the task with the lowest priority (it can only be preempted by higher-priority tasks). Hence, we have $B_3 = 3$.

7.3 Using the Priority Ceiling Protocol, a task τ_i can be blocked at most for one critical section during its execution. The set of critical sections that can potentially block τ_i is the same as that computed for the Priority Inheritance Protocol. Hence, if X_i denotes the critical section X belonging to task τ_i we have that

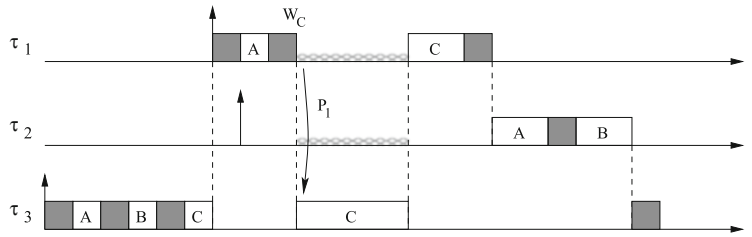


Fig. 14.20 Schedule produced by RM + PIP for the task set of Exercise 7.1

- The set of critical sections that can potentially block τ_1 is $\{A_2, A_3, C_3\}$. Hence, the maximum blocking time for τ_1 is $B_1 = d_{(C_3)} = 5$.
- The set of critical sections that can potentially block τ_2 is $\{A_3, B_3, C_3\}$. Hence, the maximum blocking time for τ_2 is $B_2 = d_{(C_3)} = 5$.
- Task τ_3 cannot be blocked, because it is the task with the lowest priority (it can only be preempted by higher-priority tasks). Hence, we have $B_3 = 0$.

7.4 The maximum blocking time for τ_2 is given by a push-through blocking on C_3. This means that, for this to happen, τ_3 must start first and must enter its critical section C_3. Then, τ_1 must preempt τ_3, so that τ_3 can inherit the highest priority to prevent τ_2 to execute. The situation is illustrated in Fig. 14.20.

7.5 To compute the maximum blocking time under the Priority Inheritance Protocol we reason as follows:

- The set of critical sections that can potentially block τ_1 is $\{C_2, B_3, E_3, A_4, C_4, E_4\}$. Among these, we have to select the three longest ones, one for each lower-priority task. Note that if we select C_2 and E_3, we cannot select E_4 (which is the longest of τ_4) because E has been already selected for τ_3, and we cannot select C_4 for the same reason. So, we have to select A_4. Hence, the maximum blocking time for τ_1 is $B_1 = d_{(C_2)} + d_{(E_3)} + d_{(A_4)} = 26$.
- Task τ_2 can experience direct blocking on C_4 and push-through blocking on B_3, E_3, A_4, C_4, and E_4. Hence, the set of critical sections that can potentially block τ_2 is $\{B_3, E_3, A_4, C_4, E_4\}$. It can be blocked at most for the duration of two critical sections in this set. Thus, we have $B_2 = d_{(E_3)} + d_{(C_4)} = 21$. Note that E_3 and E_4 cannot block τ_2 together.
- Task τ_3 can experience direct blocking on E_4 and push-through blocking on A_4, C_4, and E_4. Hence, the set of critical sections that can potentially block τ_3 is $\{A_4, C_4, E_4\}$. It can be blocked at most for the duration of one critical section in this set. Thus, we have $B_3 = d_{(E_4)} = 10$.
- Task τ_4 cannot be blocked, because it is the task with the lowest priority (it can only be preempted by higher priority tasks). Hence, we have $B_4 = 0$.

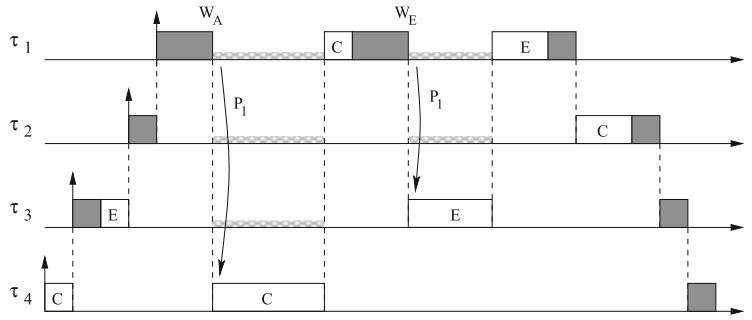


Fig. 14.21 Schedule produced by RM + PIP for the task set of Exercise 7.7

Table 14.4 SRP resource ceilings resulting for Exercise 7.8

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
A	0	1	2	3
B	0	0	0	2
C	–	0	2	3

7.6 The sets of critical sections that can cause blocking under the Priority Ceiling Protocol are the same as those derived in the previous exercise for the Priority Inheritance Protocol. The only difference is that under the Priority Ceiling Protocol each task can only be blocked for the duration of a single critical section. Hence, we have:

- The set of critical sections that can potentially block τ_1 is $\{C_2, B_3, D_3, E_3, A_4, C_4, E_4\}$. Hence, the maximum blocking time for τ_1 is $B_1 = d_{(E_3)} = 13$.
- The set of critical sections that can potentially block τ_2 is $\{B_3, E_3, A_4, C_4, E_4\}$. Hence, the maximum blocking time for τ_2 is $B_2 = d_{(E_3)} = 13$.
- The set of critical sections that can potentially block τ_3 is $\{A_4, C_4, E_4\}$. Hence, the maximum blocking time for τ_3 is $B_3 = d_{(E_4)} = 10$.
- Task τ_4 cannot be blocked, because it is the task with the lowest priority (it can only be preempted by higher-priority tasks). Hence, we have $B_4 = 0$.

7.7 The maximum blocking time for τ_2 is given by a push-through blocking on C_4 and E_3. This means that, for this to happen, τ_4 must start first and must enter its critical section C_4. Then, τ_3 must preempt τ_4, entering E_3. Now, when τ_1 arrives, it experiences a chained blocking when entering C_1 and E_1, which are both locked. The situation is illustrated in Fig. 14.21.

7.8 If tasks are assigned decreasing preemption levels as $\pi_1 = 3$, $\pi_2 = 2$, and $\pi_3 = 1$, the resource ceilings have the values shown in Table 14.4.

14.8 Solutions for Chap. 8

8.1 We first note that the task set is feasible in fully preemptive mode, in fact:

$$R_1 = C_1 = 2 \ \leq \ D_1$$

$$R_2^{(0)} = C_1 + C_2 = 4$$
$$R_2^{(1)} = C_2 + \left\lceil \frac{4}{T_1} \right\rceil C_1 = 4 \ \leq \ D_2$$

$$R_3^{(0)} = C_1 + C_2 + C_3 = 8$$
$$R_3^{(1)} = C_3 + \left\lceil \frac{8}{T_1} \right\rceil C_1 + \left\lceil \frac{8}{T_2} \right\rceil C_2 = 10$$
$$R_3^{(2)} = C_3 + \left\lceil \frac{10}{T_1} \right\rceil C_1 + \left\lceil \frac{10}{T_2} \right\rceil C_2 = 12$$
$$R_3^{(3)} = C_3 + \left\lceil \frac{12}{T_1} \right\rceil C_1 + \left\lceil \frac{12}{T_2} \right\rceil C_2 = 12 \ \leq \ D_2$$

Hence, by the result of Theorem 8.1, the feasibility of the task set in non-preemptive mode can be verified by just checking the first job of each task, when activated at its critical instant. The critical instant for task τ_i occurs when τ_i is activated together with all higher-priority tasks and one unit after the longest lower-priority task.
Using Eq. (8.1), the blocking times result to be $B_1 = 3$, $B_2 = 3$, $B_3 = 0$, and tasks' response times can be computed as $R_i = S_i + C_i$, where S_i is given by Eq. (8.8). So we have

$$S_1 = B_1 = 3$$
$$R_1 = S_1 + C_1 = 3 + 2 = 5 \ \leq \ D_1$$

$$S_2^{(0)} = B_2 + C_1 = 5$$
$$S_2^{(1)} = B_2 + \left(\left\lfloor \frac{5}{T_1} \right\rfloor + 1 \right) C_1 = 5$$
$$R_2 = S_2 + C_2 = 7 \ > \ D_2$$

Hence, the task set is not schedulable by non-preemptive RM, since τ_2 misses its deadline.

8.2 We first note that the task set is not feasible in fully preemptive mode, since

$$R_3^{(0)} = C_1 + C_2 + C_3 = 9$$

$$R_3^{(1)} = C_3 + \left\lceil \frac{9}{T_1} \right\rceil C_1 + \left\lceil \frac{9}{T_2} \right\rceil C_2 = 12$$

$$R_3^{(2)} = C_3 + \left\lceil \frac{12}{T_1} \right\rceil C_1 + \left\lceil \frac{12}{T_2} \right\rceil C_2 = 15 \; > \; D_3$$

Therefore, the response time of a task τ_i cannot be restricted to its first job, but has to be extended up to job $K_i = \lceil \frac{L_i}{T_i} \rceil$, where L_i is the longest Level-i Active Period. Using Eqs. (8.1) and (8.2), we get the following results:

	B_i	L_i	K_i
τ_1	2	5	1
τ_2	2	8	1
τ_3	1	37	3
τ_4	0	38	1

For task τ_1 we have

$$s_{1,1} = B_1 = 2$$
$$f_{1,1} = s_{1,1} + C_1 = 2 + 3 = 5$$
$$R_1 = f_{1,1} = 5 \leq D_1$$

For task τ_2 we have

$$s_{2,1}^{(0)} = B_2 + C_1 = 5$$
$$s_{2,1}^{(1)} = B_2 + \left(\left\lfloor \frac{5}{T_1} \right\rfloor + 1 \right) C_1 = 5$$
$$f_{2,1} = s_{2,1} + C_2 = 5 + 2 = 7$$
$$R_2 = f_{2,1} = 7 \; \leq \; D_2$$

For task τ_3, the response time must be checked in the first three jobs.
For $k = 1$

$$s_{3,1}^{(0)} = B_3 + C_2 + C_1 = 7$$
$$s_{3,1}^{(1)} = B_3 + \left(\left\lfloor \frac{7}{T_1} \right\rfloor + 1 \right) C_1 + \left(\left\lfloor \frac{7}{T_2} \right\rfloor + 1 \right) C_2 = 7$$
$$f_{3,1} = s_{3,1} + C_3 = 7 + 3 = 10$$
$$R_{3,1} = f_{3,1} = 10.$$

For $k = 2$

$$s_{3,2}^{(0)} = B_3 + C_3 + C_1 + C_2 = 10$$
$$s_{3,2}^{(1)} = B_3 + C_3 + \left(\left\lfloor \frac{10}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{10}{T_2} \right\rfloor + 1\right) C_2 = 16$$
$$s_{3,2}^{(2)} = B_3 + C_3 + \left(\left\lfloor \frac{16}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{16}{T_2} \right\rfloor + 1\right) C_2 = 19$$
$$s_{3,2}^{(3)} = B_3 + C_3 + \left(\left\lfloor \frac{19}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{19}{T_2} \right\rfloor + 1\right) C_2 = 22$$
$$s_{3,2}^{(4)} = B_3 + C_3 + \left(\left\lfloor \frac{22}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{22}{T_2} \right\rfloor + 1\right) C_2 = 22$$
$$f_{3,2} = s_{3,2} + C_3 = 22 + 3 = 25$$
$$R_{3,2} = f_{3,2} - T_3 = 25 - 14 = 11.$$

For $k = 3$

$$s_{3,3}^{(0)} = B_3 + 2C_3 + C_1 + C_2 = 13$$
$$s_{3,3}^{(1)} = B_3 + 2C_3 + \left(\left\lfloor \frac{13}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{13}{T_2} \right\rfloor + 1\right) C_2 = 19$$
$$s_{3,3}^{(2)} = B_3 + 2C_3 + \left(\left\lfloor \frac{19}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{19}{T_2} \right\rfloor + 1\right) C_2 = 25$$
$$s_{3,3}^{(3)} = B_3 + 2C_3 + \left(\left\lfloor \frac{25}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{25}{T_2} \right\rfloor + 1\right) C_2 = 28$$
$$s_{3,3}^{(4)} = B_3 + 2C_3 + \left(\left\lfloor \frac{28}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{28}{T_2} \right\rfloor + 1\right) C_2 = 31$$
$$s_{3,3}^{(5)} = B_3 + 2C_3 + \left(\left\lfloor \frac{31}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{31}{T_2} \right\rfloor + 1\right) C_2 = 31$$
$$f_{3,3} = s_{3,3} + C_3 = 31 + 3 = 34$$
$$R_{3,3} = f_{3,3} - 2T_3 = 34 - 28 = 6.$$

Hence, for τ_3 we have that $R_3 = \max\{R_{3,1}, R_{3,2}, R_{3,3}\} = 11 \leq D_3$. For τ_4, it can be easily verified that $R_4 = L_4 = 38 \leq D_4$. Hence, we conclude that the task set is schedulable by non-preemptive RM.

8.3 Using the Liu and Layland test, the blocking tolerance of each task can be computed by Eq. (8.20), where $U_{lub} = 1$, since tasks are scheduled by EDF:

$$\beta_i = T_i \left(1 - \sum_{h=1}^{i} \frac{C_h}{T_h}\right).$$

and, according to Eq. (8.2), Q_i results to be

$$Q_i = \begin{cases} \infty & \text{if } i = 1 \\ \min\{Q_{i-1}, \beta_{i-1}\} & \text{otherwise} \end{cases}$$

Hence, we have

	U_i	$\sum_{h=1}^{i} U_h$	β_i	Q_i
τ_1	1/5	1/5	8	∞
τ_2	1/3	8/15	7	8
τ_3	1/6	21/30	9	7
τ_4	1/4	57/60	2	7
τ_5	1/30	59/60	1	2

8.4 Under rate monotonic, using the Liu and Layland test, the blocking tolerance of each task can be computed by Eq. (8.20):

$$\beta_i = T_i \left(U_{lub}(i) - \sum_{h=1}^{i} \frac{C_h}{T_h}\right).$$

and, according to Eq. (8.2), Q_i results to be

$$Q_i = \begin{cases} \infty & \text{if } i = 1 \\ \min\{Q_{i-1}, \beta_{i-1}\} & \text{otherwise} \end{cases}$$

Hence, we have

	$U_{lub}(i)$	$\sum_{h=1}^{i} U_h$	β_i	Q_i
τ_1	1.0	1/5	8.00	∞
τ_2	0.828	8/15	4.42	8.00
τ_3	0.780	21/30	2.40	4.42
τ_4	0.757	57/60	<0	2.40
τ_5	0.743	59/60	<0	0

8.5 First of all, from the task structures, the following parameters can be derived: We note that, since the total utilization is $U = 0.75$, the task set is schedulable under rate monotonic in fully preemptive mode (in fact $U_{lub}(4) = 0.757$.

	C_i	T_i	U_i	q_i^{max}	q_i^{last}	B_i
τ_1	6	24	1/4	3	0	7
τ_2	10	30	1/4	4	4	7
τ_3	18	120	3/20	8	5	5
τ_4	15	150	1/10	6	6	0

Hence, the worst-case response time of each task can be computed considering the first job under the critical instant, using Eqs. (8.32) and (8.33).
For task τ_1 we have

$$R_1 = B_1 + C_1 = 7 + 6 = 13$$

For task τ_2 we have

$$S_2^{(0)} = B_2 + C_1 + C_2 - q_2^{last} = 7 + 6 + 10 - 4 = 19$$
$$S_2^{(1)} = B_2 + C_2 - q_2^{last} + \left(\left\lfloor \frac{19}{T_1} \right\rfloor + 1\right) C_1 = 19$$
$$R_2 = S_2 + q_2^{last} = 19 + 4 = 23$$

For task τ_3 we have

$$S_3^{(0)} = B_3 + C_1 + C_2 + C_3 - q_3^{last} = 5 + 6 + 10 + 18 - 5 = 34$$
$$S_3^{(1)} = B_3 + C_3 - q_3^{last} + \left(\left\lfloor \frac{34}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{34}{T_2} \right\rfloor + 1\right) C_2 = 50$$
$$S_3^{(2)} = B_3 + C_3 - q_3^{last} + \left(\left\lfloor \frac{50}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{50}{T_2} \right\rfloor + 1\right) C_2 = 56$$
$$S_3^{(3)} = B_3 + C_3 - q_3^{last} + \left(\left\lfloor \frac{56}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{56}{T_2} \right\rfloor + 1\right) C_2 = 56$$
$$R_3 = S_3 + q_3^{last} = 56 + 5 = 61$$

For task τ_4 we have

$$S_4^{(0)} = B_4 + C_1 + C_2 + C_3 + C_4 - q_4^{last} = 6 + 10 + 18 + 15 - 6 = 43$$
$$S_4^{(1)} = B_4 + C_4 - q_4^{last} + \left(\left\lfloor \frac{43}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{43}{T_2} \right\rfloor + 1\right) C_2 + \left(\left\lfloor \frac{43}{T_3} \right\rfloor + 1\right)$$
$$C_3 = 65$$

$$S_4^{(2)} = B_4 + C_4 - q_4^{last} + \left(\left\lfloor \frac{65}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{65}{T_2} \right\rfloor + 1\right) C_2 + \left(\left\lfloor \frac{65}{T_3} \right\rfloor + 1\right)$$

$$C_3 = 81$$

$$S_4^{(3)} = B_4 + C_4 - q_4^{last} + \left(\left\lfloor \frac{81}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{81}{T_2} \right\rfloor + 1\right) C_2 + \left(\left\lfloor \frac{81}{T_3} \right\rfloor + 1\right)$$

$$C_3 = 87$$

$$S_4^{(4)} = B_4 + C_4 - q_4^{last} + \left(\left\lfloor \frac{87}{T_1} \right\rfloor + 1\right) C_1 + \left(\left\lfloor \frac{87}{T_2} \right\rfloor + 1\right) C_2 + \left(\left\lfloor \frac{87}{T_3} \right\rfloor + 1\right)$$

$$C_3 = 87$$

$$R_4 = S_4 + q_4^{last} = 87 + 5 = 92$$

14.9 Solutions for Chap. 9

9.1 Applying the definition of instantaneous load, we have

Time	$\rho_1(t)$	$\rho_2(t)$	$\rho(t)$
$t = 0$	0	$5/10 = 0.5$	0.5
$t = 1$	0	$4/9 = 0.444$	0.444
$t = 2$	0	$3/8 = 0.375$	0.375
$t = 3$	$3/5 = 0.6$	$(3 + 2)/7 = 0.714$	0.714
$t = 4$	$2/4 = 0.5$	$(2 + 2)/6 = 0.667$	0.667
$t = 5$	$1/3 = 0.333$	$(1 + 2)/5 = 0.6$	0.6
$t = 6$	0	$2/4 = 0.5$	0.5
$t = 7$	0	$1/3 = 0.333$	0.333
$t = 8$	0	0	0

9.2 Checking condition (9.24), necessary for the schedulability of the task set, we have

$$\sum_{i=1}^{n} \frac{C_i(S_i - 1)}{T_i S_i} = \frac{2}{5} + \frac{2 \cdot 3}{6 \cdot 4} + \frac{4 \cdot 4}{8 \cdot 5} = \frac{21}{20} > 1.$$

Hence, we conclude that the task set is not schedulable by EDF.

9.3 From the service intervals provided by the server, it is clear that the longest service delay occurs when a task is ready at time $t = 2$, since it has to wait for 3 units of time. Then, the service will be provided according to the supply function illustrated in Fig. 14.22.
From the graph, it is easy to see that the associated bounded delay function has parameters $\alpha = 0.4$ and $\Delta = 3.5$.

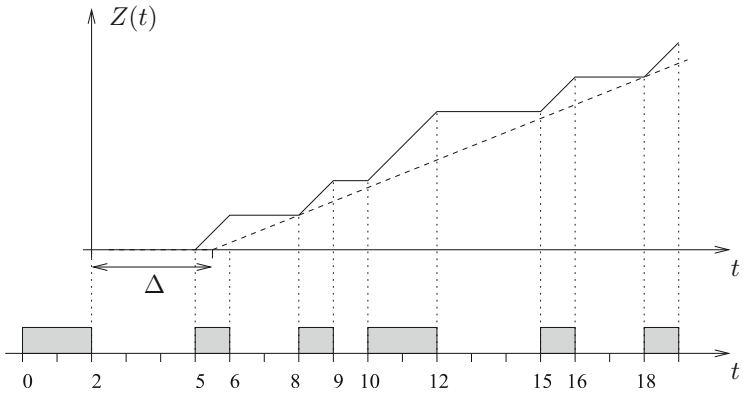


Fig. 14.22 Supply function and bounded delay function of the server

9.4 By applying Eq. (9.33) to the tasks we have

$$U_1 = U_{1_0} - (U_0 - U_d)\frac{E_1}{E_0} = 0.6 - (1.4 - 1.0)\frac{1}{4} = 0.5$$
$$U_2 = U_{2_0} - (U_0 - U_d)\frac{E_2}{E_0} = 0.8 - (1.4 - 1.0)\frac{3}{4} = 0.5$$

Hence,

$$T_1' = \frac{C_1}{U_1} = \frac{9}{0.5} = 18$$
$$T_2' = \frac{C_2}{U_2} = \frac{16}{0.5} = 32.$$

9.5 By applying Eq. (9.42) to the tasks we have

$$T_1' = T_{1_0}\frac{U_0}{U_d} = 15 \cdot 1.4 = 21$$
$$T_2' = T_{2_0}\frac{U_0}{U_d} = 20 \cdot 1.4 = 28.$$

Glossary

Absolute jitter The difference between the maximum and the minimum start time (relative to the request time) of all instances of a periodic task.

Acceptance test A schedulability test performed at the arrival time of a new task, whose result determines whether the task can be accepted into the system or rejected.

Access protocol A programming scheme that has to be followed by a set of tasks that want to use a shared resource.

Activation A kernel operation that moves a task from a sleeping state to an active state, from where it can be scheduled for execution.

Aperiodic task A type of task that consists of a sequence of identical jobs (instances), activated at irregular intervals.

Arrival rate The average number of jobs requested per unit of time.

Arrival time The time instant at which a job or a task enters the ready queue. It is also called *request time*.

Background scheduling Task-management policy used to execute low-priority tasks in the presence of high-priority tasks. Lower-priority tasks are executed only when no high-priority tasks are active.

Blocking A job is said to be blocked when it has to wait for a job having a lower priority.

Buffer A memory area shared by two or more tasks for exchanging data.

Capacity The maximum amount of time dedicated by a periodic server, in each period, to the execution of a service.

Ceiling Priority level associated with a semaphore or a resource according to an access protocol.

Ceiling blocking A special form of blocking introduced by the Priority Ceiling Protocol.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3

Channel A logical link through which two or more tasks exchange information by a message-passing mechanism.

Chained blocking A sequence of blocking experienced by a task while attempting to access a set of shared resources.

Clairvoyance An ideal property of a scheduling algorithm that implies the future knowledge of the arrival times of all the tasks that are to be scheduled.

Competitive factor A scheduling algorithm A is said to have a competitive factor φ_A if and only if it can guarantee a cumulative value at least φ_A times the cumulative value achieved by the optimal clairvoyant scheduler.

Completion time The time at which a job ends to execute. It is also called *finishing time*.

Computation time The amount of time required by the processor to execute a job without interruption. It is also called *service time* or *processing time*.

Concurrent processes Processes that overlap in time.

Context A set of data that describes the state of the processor at a particular time, during the execution of a task. Typically the context of a task is the set of values taken by the processor registers at a particular instant.

Context switch A kernel operation consisting in the suspension of the currently executing job for assigning the processor to another ready job (typically the one with the highest priority).

Creation A kernel operation that allocates and initializes all data structures necessary for the management of the object being created (such as task, resource, communication channel, and so on).

Critical instant The time at which the release of a job produces the largest response time.

Critical section A code segment subject to a mutual exclusion.

Critical zone The interval between a critical instant of a job and its corresponding finishing time.

Cumulative value The sum of the task values gained by a scheduling algorithm after executing a task set.

Deadline The time within which a real-time task should complete its execution.

Deadlock A situation in which two or more processes are waiting indefinitely for events that will never occur.

Direct blocking A form of blocking due to the attempt of accessing an exclusive resource, held by another task.

Dispatching A kernel operation consisting in the assignment of the processor to the task having highest priority.

Domino effect A phenomenon in which the arrival of a new task causes all previously guaranteed tasks to miss their deadlines.

Dynamic scheduling A scheduling method in which all active jobs are reordered every time a new job enters the system or a new event occurs.

Event An occurrence that requires a system reaction.

Exceeding time The interval of time in which a job stays active after its deadline. It is also called *tardiness*.

Exclusive resource A shared resource that cannot be accessed by more than one task at a time.

Feasible schedule A schedule in which all real-time tasks are executed within their deadlines and all the other constraints, if any, are met.

Finishing time The time at which a job ends to execute. It is also called *completion time*.

Firm task A task in which each instance must be either guaranteed to complete within its deadline or entirely rejected.

Guarantee A schedulability test that allows to verify whether a task or a set of tasks can complete within the specified timing constraints.

Hard task A task whose instances must be a priori guaranteed to complete within their deadlines.

Hyperperiod The minimum time interval after which the schedule repeats itself. For a set of periodic tasks, it is equal to the least common multiple of all the periods.

Idle state The state in which a task is not active and waits to be activated.

Idle time Time in which the processor does not execute any task.

Instance A particular execution of a task. A single job belonging to the sequence of jobs that characterize a periodic or an aperiodic task.

Interarrival time The time interval between the activation of two consecutive instances of the same task.

Interrupt A timing signal that causes the processor to suspend the execution of its current process and start another process.

Jitter The difference between the start times (relative to the request times) of two or more instances of a periodic task. See also *absolute jitter* and *relative jitter*.

Job A computation in which the operations, in the absence of other activities, are sequentially executed by the processor until completion.

Kernel An operating environment that enables a set of tasks to execute concurrently on a single processor.

Lateness The difference between the finishing time of a task and its deadline ($L = f - d$). Notice that a negative lateness means that a task is completed before its deadline.

Laxity The maximum delay that a job can experience after its activation and still complete within its deadline. At the arrival time, the laxity is equal to the relative deadline minus the computation time ($D - C$). It is also called *slack time*.

Lifetime The maximum time that can be represented inside the kernel.

Load Computation time demanded by a task set in an interval, divided by the length of the interval.

Mailbox A communication buffer characterized by a message queue shared between two or more jobs.

Message A set of data, organized in a predetermined format for exchanging information among tasks.

Mutual exclusion A kernel mechanism that allows to serialize the execution of concurrent tasks on critical sections of code.

Non-preemptive scheduling A form of scheduling in which jobs, once started, can continuously execute on the processor without interruption.

Optimal algorithm A scheduling algorithm that minimizes some cost function defined over the task set.

Overhead The time required by the processor to manage all internal mechanisms of the operating system, such as queuing jobs and messages, updating kernel data structures, performing context switches, activating interrupt handlers, and so on.

Overload Exceptional load condition on the processor, such that the computation time demanded by the tasks in a certain interval exceeds the available processor time in the same interval.

Overrun Exceptional condition in which a task exceeds its expected utilization. It can be caused by an earlier job activation (activation overrun) or by a longer execution time (execution overrun).

Period The interval of time between the activation of two consecutive instances of a periodic task.

Periodic task A type of task that consists of a sequence of identical jobs (instances), activated at regular intervals.

Phase The time instant at which a periodic task is activated for the first time, measured with respect to some reference time.

Polling A service technique in which the server periodically examines the requests of its clients.

Port A general intertask communication mechanism based on a message passing scheme.

Precedence graph A directed acyclic graph that describes the precedence relations in a group of tasks.

Precedence constraint Dependency relation between two or more tasks that specifies that a task cannot start executing before the completion of one or more tasks (called *predecessors*).

Predictability An important property of a real-time system that allows to anticipate the consequence of any scheduling decision.

Preemption An operation of the kernel that interrupts the currently executing job and assigns the processor to a more urgent job ready to execute.

Preemptive scheduling A form of scheduling in which jobs can be interrupted at any time and the processor assigned to more urgent jobs ready to execute.

Priority A number associated with a task and used by the kernel to establish an order of precedence among tasks competing for a common resource.

Priority inversion A phenomenon for which a task is blocked by a lower-priority task for an unbounded amount of time.

Process A computation in which the operations are executed by the processor one at a time. A process may consist of a sequence of identical jobs, also called instances. The words *process* and *task* are often used as synonyms.

Processing time The amount of time required by the processor to execute a job without interruption. It is also called *computation time* or *service time*.

Program A description of a computation in a formal language, called a programming language.

Push-through blocking A form of blocking introduced by the Priority Inheritance and by the Priority Ceiling Protocols.

Queue A set of jobs waiting for a given type of resource and ordered according to some parameter.

Relative jitter The maximum difference between the start times (relative to the request times) of two consecutive instances of a periodic task.

Request time The time instant at which a job or a task requests a service to the processor. It is also called *arrival time*.

Resource Any entity (processor, memory, program, data, and so on) that can be used by tasks to carry on their computation.

Resource constraint Dependency relation among tasks that share a common resource used in exclusive mode.

Response time The time interval between the request time and the finishing time of a job.

Schedulable task set A task set for which there exists a feasible schedule.

Schedule An assignment of tasks to the processor, so that each task is executed until completion.

Scheduling An activity of the kernel that determines the order in which concurrent jobs are executed on a processor.

Semaphore A kernel data structure used to synchronize the execution of concurrent jobs.

Server A kernel process dedicated to the management of a shared resource.

Service time The amount of time required by the processor to execute a job without interruption. It is also called *computation time* or *processing time*.

Shared resource A resource that is accessible by two or more processes.

Slack time The maximum delay that a job can experience after its activation and still complete within its deadline. At the arrival time, the slack is equal to the relative deadline minus the computation time ($D - C$). It is also called *laxity*.

Soft task A task whose instances should be possibly completed within their deadlines, but no serious consequences occur if a deadline is missed.

Sporadic task An aperiodic task characterized by a minimum interarrival time between consecutive instances.

Start time The time at which a job starts executing for the first time.

Starvation A phenomenon for which an active job waits for the processor for an unbounded amount of time.

Static scheduling A method in which all scheduling decisions are precomputed offline, and jobs are executed in a predetermined fashion, according to a time-driven approach.

Synchronization Any constraint that imposes an order to the operations carried out by two or more concurrent jobs. A synchronization is typically imposed for satisfying precedence or resource constraints.

Tardiness The interval of time in which a job stays active after its deadline. It is also called *exceeding time*.

Task A computation in which the operations are executed by the processor one at a time. A task may consist of a sequence of identical jobs, also called instances. The words *process* and *task* are often used as synonyms.

Task control block A kernel data structure associated with each task containing all the information necessary for task management.

Tick The minimum interval of time that is handled by the kernel. It defines the time resolution and the time unit of the system.

Timeout The time limit specified by a programmer for the completion of an action.

Time overflow Deadline miss. A situation in which the execution of a job continues after its deadline.

Timesharing A kernel mechanism in which the available time of the processor is divided among all active jobs in time slices of the same length.

Time slice A continuous interval of time in which a job is executed on the processor without interruption.

Utilization factor The fraction of the processor time utilized by a periodic computation. The utilization factor of a periodic task τ_i is the ratio of its computation time and its period ($U_i = C_i / T_i$). The utilization factor of a periodic task set is the sum of the individual task utilizations ($U = \sum_i U_i$).

Utility function A curve that describes the value of a task as a function of its finishing time.

Value A task parameter that describes the relative importance of a task with respect to the other tasks in the system.

Value density The ratio between the value of a task and its computation time.

References

[AAS97] T.F. Abdelzaher, E.M. Atkins, and K.G. Shin. QoS negotiation in real-time systems and its applications to automated flight control. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 1997.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, Spain, December 2–3, 1998.

[AB01] Luca Abeni and Giorgio Buttazzo. Hierarchical QoS management for time sensitive applications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, Taipei, Taiwan, May 30–June 1, 2001.

[AB04] Luca Abeni and Giorgio Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[ABR+93] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[ABRW92] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *IEEE Workshop on Real-Time Operating Systems*, 1992.

[AG08] S. Altmeyer and G. Gebhard. Wcet analysis for preemptive scheduling. In *Proc. of the 8th Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 105–112, Prague, Czech Republic, July 2008.

[AL86] L. Alger and J. Lala. Real-time operating system for a nuclear power plant computer. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1986.

[AMMA01] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 50(2):111–130, February 2001.

[ARI91] ARINC. *ARINC 651: Design Guidance for Integrated Modular Avionics*. Airlines Electronic Engineering Committee (AEEC), November 1991.

[ARI96] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), June 1996.

[AS88] R. J. Anderson and M. W. Spong. Hybrid impedance control of robotic manipulators. *IEEE Journal of Robotics and Automation*, 4(5), October 1988.

[B+93] J. Blazewicz et al. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3

[Bab97] Robert L. Baber. The ariane 5 explosion: a software engineer's view. *Risks Digest*, 18(89), March 1997.

[BAF94] G. C. Buttazzo, B. Allotta, and F. Fanizza. Mousebuster: a robot for catching fast objects. *IEEE Control Systems Magazine*, 14(1):49–56, February 1994.

[Baj88] R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8):996–1005, August 1988.

[Bak91] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[BAL98] Giorgio Buttazzo, Luca Abeni, and Giuseppe Lipari. Elastic task model for adaptive rate control. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[Bar05] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems (ECRTS'05)*, pages 137–144, Palma de Mallorca, Balearic Islands, Spain, July 6–8, 2005.

[Bar06] Sanjoy Baruah. Resource sharing in EDF-scheduled systems: a closer look. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, December 5–8, 2006.

[BB02] Guillem Bernat and Alan Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22:49–75, January 2002.

[BB04] Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[BB05] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1–2):129–154, 2005.

[BB06] Giorgio Buttazzo and Enrico Bini. Optimal dimensioning of a constant bandwidth server. In *Proc. of the IEEE 27th Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brasil, December 6–8, 2006.

[BBB01] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, pages 59–66, June 2001.

[BBB03] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[BBB04] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS'04)*, Lisbon, Portugal, December 5–8, 2004.

[BBB14] Giorgio Buttazzo, Enrico Bini, and Darren Buttle. Rate-adaptive tasks: Model, analysis, and design issues. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2014.

[BBB16] Alessandro Biondi, Giorgio Buttazzo, and Marko Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Transactions on Computers*, 65(5):1593–1605, 2016.

[BBL09] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transactions on Embedded Computing Systems*, 8(4):31:1–31:23, July 2009.

[BBP+16] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016)*, Porto, Portugal, November 29–December 2, 2016.

[BBS15] Alessandro Biondi, Giorgio Buttazzo, and Stefano Simoncelli. Feasibility analysis of engine control tasks under EDF scheduling. In *27th Euromicro Conference on Real-Time Systems*, Lund, Sweden, 2015.

[BC07] Giorgio Buttazzo and Anton Cervin. Comparative assessment and evaluation of jitter control methods. In *Proceedings of the 15th Int. Conf. on Real-Time and Network Systems (RTNS'07)*, pages 137–144, Nancy, France, March 29–30, 2007.

[BCRZ99] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli. Rate modulation of soft real-time tasks in autonomous robot control systems. In *IEEE Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[BCSM08] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *IEEE Proceedings of the 14th Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, Kaohsiung, Taiwan, August 2008.

[BDN93] G.C. Buttazzo and M. Di Natale. Hartik: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution. In *Proceedings of IEEE International Conference on Robotics and Automation*, May 1993.

[BDNB08] Enrico Bini, Marco Di Natale, and Giorgio C. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39(1–3):5–30, August 2008.

[BFB09] Marko Bertogna, Nathan Fisher, and Sanjoy Baruah. Resource-sharing servers for open environments. *IEEE Transactions on Industrial Informatics*, 5(3):202–220, August 2009.

[BFR71] P. Bratley, M. Florian, and P. Robillard. Scheduling with earliest start and due date constraints. *Naval Research Quarterly*, 18(4), 1971.

[BH73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, 1973.

[BKM+92] S. Baruah, G. Koren, D. Mao, A. Raghunathan B. Mishra, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Journal of Real-Time Systems*, 4, 1992.

[BL13] Giorgio Buttazzo and Giuseppe Lipari. Ptask: An educational C library for programming real-time systems on linux. In *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, Cagliari, Italy, 2013.

[BLCA02] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[Blo77] Arthur Bloch. *Murphy's Law*. Price/Stern/Sloan Publishers, Los Angeles, California, 1977.

[Blo80] Arthur Bloch. *Murphy's Law Book Two*. Price/Stern/Sloan Publishers, Los Angeles, California, 1980.

[Blo88] Arthur Bloch. *Murphy's Law Book Three*. Price/Stern/Sloan Publishers, Los Angeles, California, 1988.

[BLV09] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time System*, 42(1–3):63–119, 2009.

[BMM+14] Alessandro Biondi, Alessandra Melani, Mauro Marinoni, Marco Di Natale, and Giorgio Buttazzo. Exact interference of adaptive variable-rate tasks under fixed-priority scheduling. In *26th Euromicro Conference on Real-Time Systems*, Madrid, Spain, 2014.

[BNB15] Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. Response-time analysis for real-time tasks in engine control applications. In *6th International Conference on Cyber-Physical Systems (ICCPS 2015)*, Seattle, Washington, USA, 2015.

[BNB18] Alessandro Biondi, Marco Di Natale, and Giorgio Buttazzo. Response-time analysis of engine control applications under fixed-priority scheduling. *IEEE Transactions on Computers*, 67(5):687–703, 2018.

[BNC+20] Alessandro Biondi, Federico Nesti, Giorgiomaria Cicero, Daniel Casini, and Giorgio Buttazzo. A safe, secure, and predictable software architecture for deep learning in safety-critical systems. *IEEE Embedded Systems Letters*, 12(3):78–82, 2020.

[BNSS10] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[BR91] S. Baruah and L.E. Rosier. Limitations concerning on-line scheduling algorithms for overloaded real-time systems. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[BR18] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, December 2018.

[BRH90] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2, 1990.

[BS93] G.C. Buttazzo and J. Stankovic. Red: A robust earliest deadline scheduling algorithm. In *Proceedings of Third International Workshop on Responsive Computing Systems*, 1993.

[BS95] G.C. Buttazzo and J. Stankovic. Adding robustness in dynamic preemptive scheduling. In D.S. Fussel and M. Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*. Kluwer Academic Publishers, 1995.

[BS99] Giorgio Buttazzo and Fabrizio Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Transactions on Computers*, 48(10):1035–1052, October 1999.

[BSNN07] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, October 2007.

[BSNN08] Moris Behnam, Insik Shin, Thomas Nolte, and Mikael Nolin. Scheduling of semi-independent real-time components: Overrun methods and resource holding times. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*, pages 575–582, September 2008.

[BSR88] S. Biyabani, J. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1988.

[Bur94] Alan Burns. Preemptive priority based scheduling: An appropriate engineering approach. *S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[But91] G.C. Buttazzo. Harems: Hierarchical architecture for robotics experiments with multiple sensors. In *IEEE Proceedings of the Fifth International Conference on Advanced Robotics ('91 ICAR)*, June 1991.

[But93] G.C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1993.

[But96] G. C. Buttazzo. Real-time issues in advanced robotics applications. In *Proceedings of the 8th IEEE Euromicro Workshop on Real-Time Systems*, pages 77–82, June 1996.

[But03] Giorgio C. Buttazzo. Rate monotonic vs. EDF: Judgment day. In *Proceedings of the* 3rd *International Conference on Embedded Software (EMSOFT 2003*, pages 67–83, Philadelphia, PA, October 2003.

[But06] Giorgio C. Buttazzo. Achieving scalability in real-time systems. *IEEE Computer*, 39(5):54–59, May 2006.

[But12] Darren Buttle. Real-time in the prime-time. Keynote speech given at the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012), Pisa, Italy, July 12th, 2012.

[But22] Giorgio C. Buttazzo. Can we trust AI-powered real-time embedded systems? (invited paper). In Marko Bertogna, Federico Terraneo, and Federico Reghenzani, editors, *Third Workshop on Next Generation Real-Time Embedded Systems, NG-RES@HiPEAC 2022, June 22, 2022, Budapest, Hungary*, volume 98 of *OASIcs*, pages 1:1–1:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[CB97] M. Caccamo and G.C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *IEEE Real-Time Systems Symposium*, pages 330–339, San Francisco, California, USA, 1997.

[CB03] Alessio Carlini and Giorgio Buttazzo. An efficient time representation for real-time embedded systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2003)*, Melbourne, Florida, USA, 2003.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Proc. of the 29th Conference on Neural Information Processing Systems (NIPS 2015)*, Montreal, Canada, December 7–10, 2015.

[CBS00] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, 2000.

[CBT05] M. Caccamo, G.C. Buttazzo, and D.C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, February 2005.

[CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10), 1989.

[CCB17] R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory Interference Characterization Between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms. In *Proc. of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2017)*, Limassol, Cyprus, September 12–15, 2017.

[CCBP18] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. Deadline-Based Scheduling for GPU with Preemption Support. In *Proc. of the 39th IEEE Real-Time Systems Symposium (RTSS 2018)*, Nashville, Tennessee, USA, December 11–14, 2018.

[Cer03] Anton Cervin. Integrated control and real-time scheduling. Doctoral dissertation (page 94), ISRN LUTFD2/TFRT-1065-SE, Department of Automatic Control, University of Lund, Sweden, April 2003.

[CL90] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2, 1990.

[Cla89] D. Clark. HIC: An operating system for hierarchies of servo loops. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1989.

[CMB+ar] Edoardo Cittadini, Mauro Marinoni, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Supporting AI-powered Real-Time Cyber-Physical Systems on Heterogeneous Platforms via Hypervisor Technology. *Real-Time Systems*, 2023. https://doi.org/10.1007/s11241-023-09402-4

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems*, 2, 1990.

[Cut85] M. R. Cutkosky. *Robot Grasping and Fine Manipulation*. Kluwer Academic Publishers, 1985.

[DB87] P. Dario and G. C. Buttazzo. An anthropomorphic robot finger for investigating artificial tactile perception. *International Journal of Robotics Research*, 6(3):25–48, Fall 1987.

[DB06] R.I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the* 27th *IEEE International Real-Time Systems Symposium*, pages 257–270, Rio de Janeiro, Brazil, December 5–8, 2006.

[Der74] M.L. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 74, 1974.

[DFPS14] Robert I. Davis, Timo Feld, Victor Pollex, and Frank Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Berlin, Germany, 2014.

[DG00] R. Devillers and J. Goossens. Liu and layland's schedulability test revisited. *Information Processing Letters*, 73(5):157–161, March 2000.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, New York, 1968.

[DRSK89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of mars. *Operating System Review*, 23(3):141–157, July 1989.

[DTB93] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1993.

[EAL07] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using EDP resource models. In *Proceedings of the* 28th *IEEE International Real-Time Systems Symposium*, pages 129–138, Tucson, AZ, USA, 2007.

[ERC95] J. Echague, I. Ripoll, and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. In *Proc. of the 7th Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 14–16, 1995.

[Erw21] Blane Erwin. The Groundbreaking 2015 Jeep Hack Changed Automotive Cybersecurity, 2021.

[FBD+18] Timo Feld, Alessandro Biondi, Robert I. Davis, Giorgio Buttazzo, and Frank Slomka. A survey of schedulability analysis techniques for rate-dependent tasks. *Journal of Systems and Software*, 138:100–107, 2018.

[FM02] Xiang Feng and Aloysius K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX, USA, December 2002.

[GA07] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *Proc. of the 7th ACM-IEEE Int. Conf. on Embedded Software (EMSOFT 07)*, pages 166–171, Salzburg, Austria, 2007.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[GB95] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time Systems*, 9, 1995.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[GLLK79] R. Graham, E. Lawler, J.K. Lenstra, and A.H.G. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[GR91] N. Gehani and K. Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *Journal of Real-Time Systems*, 3, 1991.

[Gra76] R.L. Graham. Bounds on the performance of scheduling algorithms. In *Computer and Job Scheduling Theory*, pages 165–227. John Wiley and Sons, 1976.

[Gua09] Qian Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3):181–194, 2009.

[Guo18] Yunhui Guo. A Survey on Methods and Theories of Quantized Neural Networks. *ArXiv*, abs/1808.04752, 2018.

[HHPD87] V.P. Holmes, D. Harris, K. Piorkowski, and G. Davidson. Hawk: An operating system kernel for a real–time embedded multiprocessor. Technical report, Sandia National Laboratories, 1987.

[Hil92] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on MicroKernels and Other Kernel Architectures*, April 1992.

[HLC91] J.R. Haritsa, M. Livny, and M.J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1991.

[Hor74] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21, 1974.

[Jac55] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Management Science Research Project 43, University of California, Los Angeles, 1955.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(2):129–130, January 1997.

[JP86] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, 1986.

[JS93] K. Jeffay and D.L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 212–221, December 1993.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks with varying execution priority. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[JSP92] K. Jeffay, D.L. Stone, and D. Poirier. Yartos: Kernel support for efficient, predictable real-time systems. In W. Halang and K. Ramamritham, editors, *Real-Time Programming*, pages 7–12. Pergamon Press, 1992.

[KAK91] T.-W. Kuo and Mok A. K. Load adjustment in adaptive real-time systems. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[Kar92] R. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? *Information Processing*, 92(1), 1992.

[KB86] O. Khatib and J. Burdick. Motion and force control of robot manipulators. In *Proceedings of IEEE Conference on Robotics and Automation*, 1986.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabla, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1), February 1989.

[KIM78] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one machine scheduling problem with ready and due times. *Operations Research*, 26(1):121–126, 1978.

[KKS89] D.D. Kandlur, D.L. Kiskis, and K.G. Shin. Hartos: A distributed real-time operating system. *Operating System Review*, 23(3), July 1989.

[KS86] E. Kligerman and A. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9), September 1986.

[KS92] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[KS95] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995.

[Law73] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Managements Science*, 19, 1973.

[LB03] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proceedings of the* 15th *Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 151–158, Porto, Portugal, July 2003.

[LB05] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack management. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS 2005)*, Miami, Florida, USA, December 5–8, 2005.

[LDS07] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of ACM Workshop on Experimental Computer Science (ExpCS'07)*, San Diego, California, June 13–14, 2007.

[Leh90] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, Lake Buena Vista, Florida, USA, 1990.

[LHS+98] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[LK88] I. Lee and R. King. Timix: A distributed real-time kernel for multi-sensor robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, 1988.

[LKP88] I. Lee, R. King, and R. Paul. RK: A real-time kernel for a distributed system with predictable response. MS-CIS-88-78, GRASP LAB 155 78, Department of Computer Science, University of Pennsylvania, Philadelphia, PA, October 1988.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[LLA01] G. Lamastra, G. Lipari, and L. Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *IEEE Proceedings of the 22nd Real-Time Systems Symposium (RTSS'01)*, London, UK, December 3–6, 2001.

[LLFC11] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011)*, Porto, Portugal, 2011.

[LLN87] J.W.S. Liu, K.J. Lin, and S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. In *Proceedings of the IEEE Real-Time System Symposium*, December 1987.

[LLS+91] J.W.S. Liu, K. Lin, W. Shih, A. Yu, C. Chung, J. Yao, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[LNL87] K.J. Lin, S. Natarajan, and J.W.S. Liu. Concord: a system of imprecise computation. In *Proceedings of the 1987 IEEE Compsac*, October 1987.

[Loc86] C.D. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, 1986.

[LRKB77] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 7(1):343–362, 1977.

[LRM96] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic QoS in real-time mach. In *Proceedings of Multimedia Japan 96*, April 1996.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992.

[LS06] Giuseppe Lipari and Claudio Scordino. Linux and real-time: Current approaches and future opportunities. In *Proceedings of the 50th Int. Congress of ANIPLA on Methodologies for Emerging Technologies in Automation (ANIPLA 2006)*, Rome, Italy, November 2006.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS'89)*, pages 166–171, Santa Monica, CA, USA, December 5–7, 1989.

[LSL+94] J.W.S. Liu, W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, January 1994.

[LSS87] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[LTCA89] S.-T. Levi, S.K. Tripathi, S.D. Carson, and A.K. Agrawala. The maruti hard real-time operating system. *Operating System Review*, 23(3), July 1989.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[MFC01] Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pages 75–84, Taipei, Taiwan, May 2001.

[MLBC04] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proc. of the IEEE Real-*

Time and Embedded Technology and Applications Symposium, Toronto, Canada, May 2004.

[MST93] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[MST94a] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of IEEE international conference on Multimedia Computing and System*, May 1994.

[MST94b] C. W. Mercer, S. Savage, and H. Tokuda. Temporal protection in real-time operating systems. In *Proceedings of th 11th IEEE workshop on Real-Time Operating System and Software*, pages 79–83. IEEE, May 1994.

[MV15] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle, August 10, 2015.

[Nak98] T. Nakajima. Resource reservation for adaptive QoS mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 1998.

[Nat95] Swaminathan Natarajan, editor. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[NBB21] Federico Nesti, Alessandro Biondi, and Giorgio Buttazzo. Detecting Adversarial Examples by Input Transformations, Defense Perturbations, and Voting. *IEEE Transactions on Neural Networks and Learning Systems*, 34(3):1329–1341, March 2023.

[NSBS09] Thomas Nolte, Insik Shin, Moris Behnam, and Mikael Sjödin. A synchronization protocol for temporal isolation of software components in vehicular systems. *IEEE Transactions on Industrial Informatics*, 5(4):375–387, November 2009.

[NT94] Tatsuo Nakajima and Hiroshi Tezuka. A continuous media application supporting dynamic QoS control on real-time mach. In *ACM Multimedia*, 1994.

[OSE03] OSEK. *OSEK/VDX Operating System Specification 2.2.1*. OSEK Group, http://www.osek-vdx.org, 2003.

[OSE04] OSE. *OSE Real-Time Operating System*. ENEA Embedded Technology, http://www.ose.com, 2004.

[PALW02] Luigi Palopoli, Luca Abeni, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin-Texas, December 3–5, 2002.

[PBB+17] Marco Pagani, Alessio Balsini, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. A Linux-based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration. In *Proceedings of the 30th IEEE International System-on-Chip Conference (SOCC 2017)*, Munich, Germany, September 5–8, 2017.

[PBM+ar] M. Pagani, A. Biondi, M. Marinoni, L. Molinari, G. Lipari, and G. Buttazzo. A Linux-Based Support for Developing Real-Time Applications on Heterogeneous Platforms with Dynamic FPGA Reconfiguration. *Future Generation Computer Systems*, 129:125–140, April 2022.

[PFFSa+13] Victor Pollex, Timo Feld, Ulrich Margull Frank Slomka and, Ralph Mader, and Gerhard Wirrer. Sufficient real-time analysis for an engine control unit with constant angular velocities. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, 2013.

[PGBA02] P. Pedreiras, Paolo Gai, Giorgio Buttazzo, and Luis Almeida. FTT-Ethernet: A platform to implement the elastic task model over message streams. In *Fourth IEEE Workshop on Factory Communication Systems (WFCS 2002)*, pages 225–232, August 2002.

[POS03] POSIX. *IEEE Standard 1003.13-2003, Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime and Embedded Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 2003.

[PRB+19] Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo. A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Stuttgart, Germany, July 9–12, 2019.

[PS85] J. Peterson and A. Silberschatz. *Operating Systems Concepts*. Addison-Wesley, 1985.

[Raj91] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[RB21] Francesco Restuccia and Alessandro Biondi. Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms. In *Proc. of the 42nd IEEE Real-Time Systems Symposium (RTSS 2021)*, Online event, December 7–10, 2021.

[RBB21] Giulio Rossolini, Alessandro Biondi, and Giorgio Buttazzo. Increasing the Confidence of Deep Neural Networks by Coverage Analysis. In *arXiv:2101.12100 [cs.LG]*, January 2021.

[RBMB20] Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Safely preventing unbounded delays during bus transactions in FPGA-based SoC. In *Proceedings of the 28th Annual Int. Symposium on Field-Programmable Custom Computing Machines (FCCM 2020)*, Fayetteville, Arkansas, USA, May 3–6, 2020.

[RDB+18] Enrico Rossi, Marvin Damschen, Lars Bauer, Giorgio Buttazzo, and Jörg Henkel. Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing with FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 11(2):10:1–10:24, November 2018.

[Rea86] J. Ready. VRTX: A real-time operating system for embedded microprocessor applications. *IEEE Micro*, August 1986.

[Reg02] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 315–326, Austin, TX, USA, December 3–5, 2002.

[RH01] Mario Aldea Rivas and Michael González Harbour. MaRTE OS: An ada kernel for real-time embedded applications. In *Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, LNCS*, May 2001.

[RH04] Mario Aldea Rivas and Michael González Harbour. A new generalized approach to application-defined scheduling. In *Proceedings of 16th Euromicro Conference on Real-Time Systems (WiP)*, Catania, (Italy), June 30–July 2 2004.

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[RM06] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemption points. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 212–224, Rio de Janeiro, Brazil, December 5–8, 2006.

[RM08] Harini Ramaprasad and Frank Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 58–67, St. Louis, MO, USA, April 22–24, 2008.

[RM09] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 10(2), Dec 2010.

[RNB+22] Giulio Rossolini, Federico Nesti, Fabio Brau, Alessandro Biondi, and Giorgio Buttazzo. Defending from physically-realizable adversarial attacks through internal over-activation analysis. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, Washington, DC, USA, February 7–14, 2022.

[RPB+19] Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs. *ACM Transactions on Embedded Computing Systems*, 18(5–51):1–22, October 2019.

[RPB+20] Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Modeling and analysis of bus contention for hardware accelerators in FPGA SoCs. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, Online event, July 7–10, 2020.

[RS84] K. Ramamritham and J.A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), July 1984.

[RTL93] S. Ramos-Thuel and J.P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1993.

[Sak98] Ken Sakamura. *micro-ITRON: An Open and Portable Real-Time Operating System for Embedded Systems: Concept and Specification*. IEEE Computer Society, April 1998.

[SB94] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994.

[SB96] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2), 1996.

[SBB19] Biruk Seyoum, Alessandro Biondi, and Giorgio Buttazzo. FLORA: FLoorplan Optimizer for Reconfigurable Areas in FPGAs. *ACM Transactions on Embedded Computing Systems*, 18(5–73):1–20, October 2019.

[SBG86] K. Schwan, W. Bo, and P. Gopinath. A high performance, object-based operating system for real-time robotics application. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1986.

[SBS95] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1995.

[SGB87] K. Schwan, P. Gopinath, and W. Bo. Chaos–kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8), August 1987.

[SGB21] Marion Sudvarg, Chris Gill, and Sanjoy Baruah. Linear-time admission control for elastic scheduling. *Real-Time Systems*, 57(4):485–490, 2021.

[SH98] Mikael Sjodin and Hans Hansson. Improved response-time analysis calculations. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 399–408, 1998.

[Sha85] S. Shani. *Concepts in Discrete Mathematics*. Camelot Publishing Company, 1985.

[Sim18] Tom Simonite. AI Has a Hallucination Problem That's Proving Tough to Fix, March 12, 2018.

[SL03] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the* 24th *Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, December 2003.

[SLC91] W. Shih, W.S. Liu, and J. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal of Computing*, 20(3):537–552, July 1991.

[SLCG89] W. Shih, W.S. Liu, J. Chung, and D.W. Gillies. Scheduling tasks with ready times and deadlines to minimize average error. *Operating System Review*, 23(3), July 1989.

[SLR88] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1988.

[SLS95] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 4(1), January 1995.

[SLSS96] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control system. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[Spu95] M. Spuri. *Earliest Deadline Scheduling in Real-Time Systems*. PhD thesis, Scuola Superiore S. Anna, Pisa, Italy, 1995.

[SR87] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[SR88] J. Stankovic and K. Ramamritham, editors. *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[SR90] J.A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Journal of Real-Time Systems*, 2, 1990.

[SR91] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[SRS93] C. Shen, K. Ramamritham, and J. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Computing*, 4(4):382–397, April 1993.

[SSDNB95] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6), June 1995.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[Sta88] J.A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10), October 1988.

[SW00] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, Orlando, Florida, USA, November 27–30, 2000.

[SZ92] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[SZS+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proc. of the 2nd International Conference on Learning Representations (ICLR 2014)*, Banff, AB, Canada, April 14–16, 2014.

[Tak02] Hiroaki Takada. *micro-ITRON 4.0 specification (version 4.00.00)*. TRON Association, Japan, 2002, http://www.ertl.jp/ITRON/SPEC/home-e.html, 2002.

[TK88] H. Tokuda and M. Kotera. A real-time tool set for the arts kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1988.

[TLS95] T.S. Tia, J.W.S. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*, 1995.

[TM89] H. Tokuda and C.W. Mercer. Arts: A distributed real-time kernel. *Operating System Review*, 23(3), July 1989.

[TT89] P. Thambidurai and K.S. Trivedi. Transient overloads in fault-tolerant real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.

[TWW87] H. Tokuda, J. Wendorf, and H. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[TYC18] Shixin Tian, Guolei Yang, and Ying Cai. Detecting Adversarial Examples through Image Transformation. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18)*, New Orleans, Louisiana, USA, February 2–7, 2018.

[VxW95] VxWorks. *VxWorks Programmer's Guide: Algorithms for Real-Time Scheduling Problems*. Wind River Systems, Inc., Alameda, CA (USA), 1995.

[Whi85] D. E. Whitney. Historical perspective and state of the art in robot force control. In *Proceedings of IEEE Conference on Robotics and Automation*, 1985.

[WR91] E. Walden and C.V. Ravishankar. Algorithms for real-time scheduling problems. Technical report, University of Michigan, Department of Electrical Engineering and Computer Science, Michigan (USA), April 1991.

[WS99] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of the 6th IEEE Int. Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 328–335, Hong Kong, China, December 13–15, 1999.

[YBB09] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. of the 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 351–360, Beijing, China, August 24–26, 2009.

[YBB10a] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Proc. of the 16th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'10)*, pages 71–80, Macau, SAR, China, August 23–25, 2010.

[YBB10b] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Comparative evaluation of limited preemptive methods. In *Proc. of the 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2010)*, Bilbao, Spain, September 13–16, 2010.

[YS07] Patrick Meumeu Yomsi and Yves Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Proc. of the 19th EuroMicro Conf. on Real-Time Systems (ECRTS'07)*, Pisa, Italy, July 4–6, 2007.

[Zlo93] G. Zlokapa. Real-time systems: Well-timed scheduling and scheduling with precedence constraints. Ph.D. thesis, CS-TR 93 51, Department of Computer Science, University of Massachusetts, Amherst, MA, February 1993.

Index

G. Buttazzo, *Hard Real-Time Computing Systems*,
https://doi.org/10.1007/978-3-031-45410-3