

HARD REAL-TIME COMPUTING SYSTEMS

**Predictable Scheduling
Algorithms and Applications**

Giorgio C. Buttazzo



Kluwer Academic Publishers

HARD REAL-TIME COMPUTING SYSTEMS

*Predictable Scheduling
Algorithms and Applications*

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

REAL-TIME SYSTEMS

Consulting Editor

John A. Stankovic

REAL-TIME SYSTEMS: Design Principles for Distributed Embedded Applications, by Hermann Kopetz, ISBN: 0-7923-9894-7

REAL-TIME DATABASE SYSTEMS: Issues and Applications, edited by Azer Bestavros, Kwei-Jay Lin and Sang Hyuk Son, ISBN: 0-7923-9897-1

FAULT-TOLERANT REAL-TIME SYSTEMS: The Problem of Replica Determinism, by Stefan Poledna, ISBN: 0-7923-9657-X

RESPONSIVE COMPUTER SYSTEMS: Steps Toward Fault-Tolerant Real-Time Systems, by Donald Fussell and Miroslaw Malek, ISBN: 0-7923-9563-8

IMPRECISE AND APPROXIMATE COMPUTATION, by Swaminathan Natarajan, ISBN: 0-7923-9579-4

FOUNDATIONS OF DEPENDABLE COMPUTING: System Implementation, edited by Gary M. Koob and Clifford G. Lau, ISBN: 0-7923-9486-0

FOUNDATIONS OF DEPENDABLE COMPUTING: Paradigms for Dependable Applications, edited by Gary M. Koob and Clifford G. Lau, ISBN: 0-7923-9485-2

FOUNDATIONS OF DEPENDABLE COMPUTING: Models and Frameworks for Dependable Systems, edited by Gary M. Koob and Clifford G. Lau, ISBN: 0-7923-9484-4

THE TESTABILITY OF DISTRIBUTED REAL-TIME SYSTEMS, Werner Schütz; ISBN: 0-7923-9386-4

A PRACTITIONER'S HANDBOOK FOR REAL-TIME ANALYSIS: Guide to Rate Monotonic Analysis for Real-Time Systems, Carnegie Mellon University (Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, Michale González Harbour); ISBN: 0-7923-9361-9

FORMAL TECHNIQUES IN REAL-TIME FAULT-TOLERANT SYSTEMS, J. Vytopil; ISBN: 0-7923-9332-5

SYNCHRONOUS PROGRAMMING OF REACTIVE SYSTEMS, N. Halbwachs; ISBN: 0-7923-9311-2

REAL-TIME SYSTEMS ENGINEERING AND APPLICATIONS, M. Schiebe, S. Pferer; ISBN: 0-7923-9196-9

SYNCHRONIZATION IN REAL-TIME SYSTEMS: A Priority Inheritance Approach, R. Rajkumar, ISBN: 0-7923-9211-6

CONSTRUCTING PREDICTABLE REAL TIME SYSTEMS, W. A. Halang, A. D. Stoyenko; ISBN: 0-7923-9202-7

FOUNDATIONS OF REAL-TIME COMPUTING: Formal Specifications and Methods, A. M. van Tilborg, G. M. Koob; ISBN: 0-7923-9167-5

FOUNDATIONS OF REAL-TIME COMPUTING: Scheduling and Resource Management, A. M. van Tilborg, G. M. Koob; ISBN: 0-7923-9166-7

REAL-TIME UNIX SYSTEMS: Design and Application Guide, B. Furht, D. Grostic, D. Gluch, G. Rabbat, J. Parker, M. McRoberts, ISBN: 0-7923-9099-7

HARD REAL-TIME COMPUTING SYSTEMS

*Predictable Scheduling
Algorithms and Applications*

by

Giorgio C. Buttazzo
*Scuola Superiore S. Anna
Pisa, Italy*



KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London

Distributors for North America:

Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

Distributors for all other countries:

Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Copyright © 1997 by Kluwer Academic Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061

Printed on acid-free paper.

Printed in the United States of America

CONTENTS

Preface	ix
1 A GENERAL VIEW	1
1.1 Introduction	1
1.2 What does real time mean?	4
1.3 Achieving predictability	12
2 BASIC CONCEPTS	23
2.1 Introduction	23
2.2 Types of task constraints	25
2.3 Definition of scheduling problems	34
2.4 Scheduling anomalies	44
3 APERIODIC TASK SCHEDULING	51
3.1 Introduction	51
3.2 Jackson's algorithm	52
3.3 Horn's algorithm	56
3.4 Non-preemptive scheduling	61
3.5 Scheduling with precedence constraints	68
3.6 Summary	74
4 PERIODIC TASK SCHEDULING	77
4.1 Introduction	77
4.2 Rate Monotonic scheduling	82
4.3 Earliest Deadline First	93
4.4 Deadline Monotonic	96
4.5 EDF with deadlines less than periods	102
4.6 Summary	107

5 FIXED-PRIORITY SERVERS	109
5.1 Introduction	109
5.2 Background scheduling	110
5.3 Polling Server	111
5.4 Deferrable Server	116
5.5 Priority Exchange	125
5.6 Sporadic Server	132
5.7 Slack stealing	138
5.8 Non-existence of optimal servers	142
5.9 Performance evaluation	145
5.10 Summary	146
6 DYNAMIC PRIORITY SERVERS	149
6.1 Introduction	149
6.2 Dynamic Priority Exchange Server	150
6.3 Dynamic Sporadic Server	155
6.4 Total Bandwidth Server	159
6.5 Earliest Deadline Late Server	163
6.6 Improved Priority Exchange Server	167
6.7 Improving TBS	171
6.8 Performance evaluation	175
6.9 Summary	178
7 RESOURCE ACCESS PROTOCOLS	181
7.1 Introduction	181
7.2 The priority inversion phenomenon	182
7.3 Priority Inheritance Protocol	186
7.4 Priority Ceiling Protocol	201
7.5 Stack Resource Policy	208
7.6 Summary	221
8 HANDLING OVERLOAD CONDITIONS	225
8.1 Introduction	225
8.2 Load definitions	228
8.3 Performance metrics	230
8.4 Scheduling schemes for overload	243
8.5 Performance evaluation	249

9 KERNEL DESIGN ISSUES	253
9.1 Structure of a real-time kernel	253
9.2 Process states	256
9.3 Data structures	261
9.4 Miscellaneous	265
9.5 Kernel primitives	271
9.6 Intertask communication mechanisms	289
9.7 System overhead	296
10 APPLICATION DESIGN ISSUES	301
10.1 Introduction	302
10.2 Time constraints definition	306
10.3 Hierarchical design	313
10.4 A robot control example	318
11 EXAMPLES OF REAL-TIME SYSTEMS	323
11.1 Introduction	323
11.2 MARS	325
11.3 Spring	331
11.4 RK	336
11.5 ARTS	340
11.6 HARTIK	345
Glossary	353
REFERENCES	363
INDEX	373

PREFACE

Real-time computing plays a crucial role in our society since an increasing number of complex systems rely, in part or completely, on processor control. Examples of applications that require real-time computing include nuclear power plants, railway switching systems, automotive electronics, air traffic control, telecommunications, robotics, and military systems.

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 reinforced 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 and clearly illustrates the main characteristics that distinguish real-time processing from other types of computing.

Chapter 2 treats the general issue of scheduling tasks on a single processor system. Objectives, performance metrics, and hypotheses are clearly presented, and the scheduling problem is precisely formalized. The different algorithms proposed in the literature are then classified in a taxonomy, which provides a useful reference framework for understanding the different approaches. At the end of the chapter, a number of multiprocessor 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 Chapter 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 important 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. The latter two protocols also prevent deadlocks and chained blocking.

Chapter 8 deals with the problem of real-time scheduling during transient overload conditions; that is, those situations in which the total task 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 9 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 intertask 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 10 discusses some important issues related to the design of real-time applications. A robot control system is considered as a specific example for illustrating why control applications need real-time computing and how time constraints 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 Chapter 9 is used to illustrate a concrete programming example of real-time tasks for sensory processing and control activities.

Chapter 11 concludes the book by presenting a number of hard real-time operating systems proposed in the literature. The systems examined include MARS, Spring, RK, ARTS, and HARTIK. Each system is considered in terms of supported architecture, scheduling algorithm, communication mechanism, and interrupt handling.

Acknowledgments

This work is the result of seven years of research and teaching activity in the field of real-time control 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.

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 my students at the University of Pisa, who have directly and indirectly contributed to its readability and clarity.

A personal note of appreciation goes to Paolo Ancilotti, who gave me the opportunity to teach these topics. Moreover, I would like to acknowledge the contributions of Jack Stankovic, Krithi Ramamritham, Herman Kopetz, John Lehoczky, and Gerard Le Lann. Their input enhanced the overall quality of this work. I would also like to thank the Kluwer editorial staff, and especially Bob Holland, for the support I received during the preparation of the manuscript.

Special appreciation goes to Marco Spuri, 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, and Gerhard Fohler, for the interesting discussions on leading scheduling issues.

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

Finally, I express my appreciation to my wife, Maria Grazia, and my daughter, Rossella, for their patience and understanding during the preparation of this book.

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,
- Industrial automation,
- Robotics,
- Military systems,

- Space missions, and
- Virtual reality.

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 program 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 time 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 features 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 damage to the environment.

A high percentage of accidents that occur in nuclear power plants, in space missions, or in defensive 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 percent) 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 on-board 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 on-board 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 subtle software bug, the real-time clock of the on-board computer was accumulating a delay of about 57 microseconds per minute. The day of the accident, the computer had been working for about 100 hours (an exceptional condition that was never experienced before), thus accumulating a total delay of 343 milliseconds. This delay caused a prediction error in the verification phase of 687 meters! The bug was corrected on February 26, the day after the accident.

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.

¹ *L'Espresso*, Vol. XXXVIII, No. 14, 5 April 1992, p. 167.

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 time 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 U.S. 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 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

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 bomb-proof, not jinx-proof.

Troutman Postulates

1. *Any software bug will tend to maximize the damage.*
2. *The worst software bug will be discovered six 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.

Table 1.1 Murphy's laws on real-time systems.

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 fly-swatter, a mouse can be captured by a trap, or a cat can be run down by a speeding car. In these examples, the fly-swatter, 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 Figure 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 that it is strictly related to the environment in which the system operates. It does not make sense to design a real-time computing system for flight control without considering the timing characteristics of the aircraft. Hence, 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. 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].

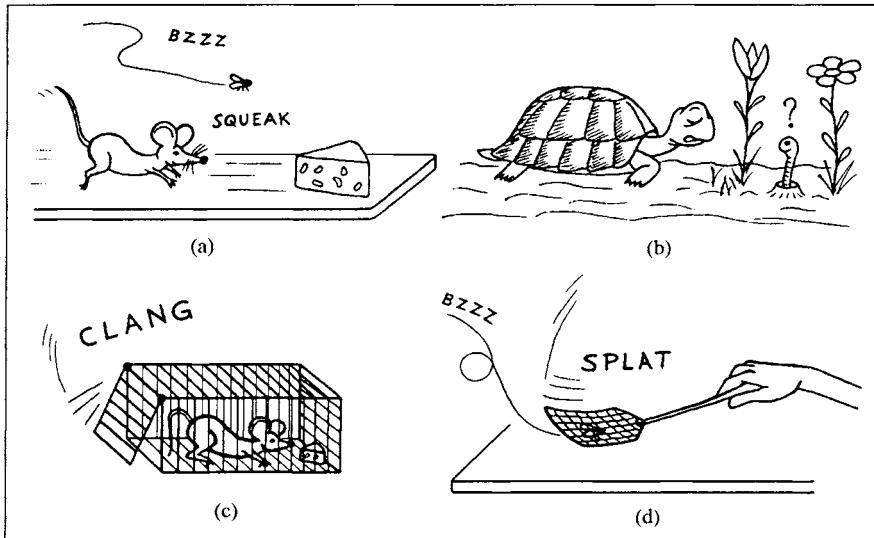


Figure 1.1 Both the mouse (a) and the turtle (b) behave in real time with respect to their natural habitat. Nevertheless, the survival of fast animals such as a mouse or a fly can be jeopardized by events (c and d) quicker than their reactive capabilities.

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 story²:

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.

²From John Stankovic's notes.

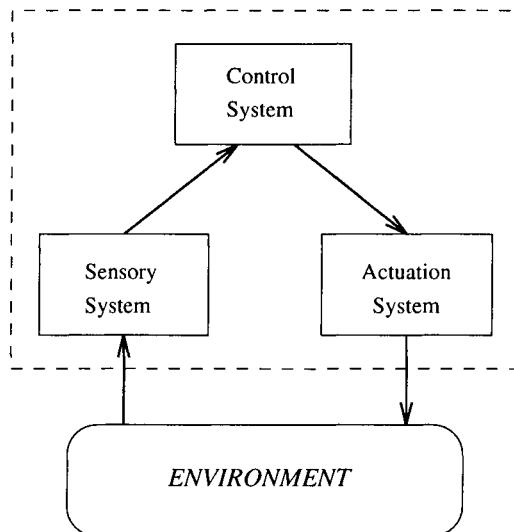


Figure 1.2 Block diagram of a generic real-time control system.

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, real-time tasks are usually distinguished in two classes, *hard* and *soft*:

- A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the environment under control.
- A real-time task is said to be *soft* if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.

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 and soft activities, and therefore a hard real-time system should be designed to handle both hard and soft tasks using two different strategies. In general, when an application consists of a hybrid task set, the objective of the operating system should be to guarantee the individual timing constraints of the hard tasks while minimizing the average response time of the soft activities.

Examples of hard activities that may be present in a control application include

- Sensory data acquisition,
- Detection of critical conditions,
- Actuator servoing,
- Low-level control of critical system components, and
- Planning sensory-motor actions that tightly interact with the environment.

Examples of soft activities 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 timesharing operating systems. As a consequence, they have the same basic features found in timesharing 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 timesharing 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 timesharing 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 time constraints and different criticalness.
- **Design for peak load.** Real-time systems must not collapse when they are subject to peak-load conditions, so they must be designed to manage all anticipated scenarios.
- **Predictability.** To guarantee a minimum level of performance, the system must be able to predict the consequences of any scheduling decision. 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 in time to cope with the event.
- **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 do 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 a 8 Kbyte cache, the data requested from a program are found in the cache 80 percent 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 percent of the time, it is also true that in the other 20 percent 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 percent of the memory accesses are for read operations, whereas only 10 percent 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. To perform worst-case analysis, in fact, we should assume a cache fault for each memory access. The consequence of this is that, to obtain a higher degree of predictability at the low level, it would be more efficient to have processors without cache or with the cache disabled. In other approaches, the influence of the cache on the task execution time is taken into account by a multiplicative factor, which depends on an estimated percentage of cache faults. A more

precise estimation of the cache behavior can be achieved by analyzing the code of the tasks and estimating the execution times by using a mathematical model of the cache.

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.

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].

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 little 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].

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 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 Figure 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.

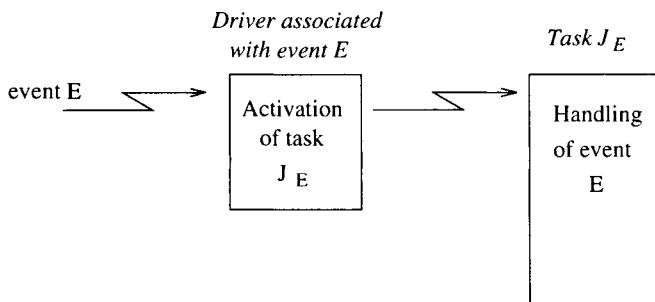


Figure 1.3 Activation of a device-handling task.

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 [BD93, 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 Chapter 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 run time. 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.

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, that 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 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

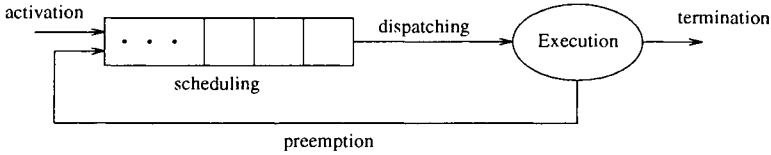


Figure 2.1 Queue of ready tasks waiting for execution.

a queue, called *ready queue*. Operating systems that handles 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, while the CPU is assigned to the most important ready task which 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 criticalness expressing task importance, preemption permits to anticipate the execution of the most critical activities.
- More efficient schedules can be produced to improve system responsiveness.

Given a set of tasks, $J = \{J_1, \dots, J_n\}$, 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, J_3 .

- At times t_1, t_2, t_3 , and t_4 , the processor performs a *context switch*.

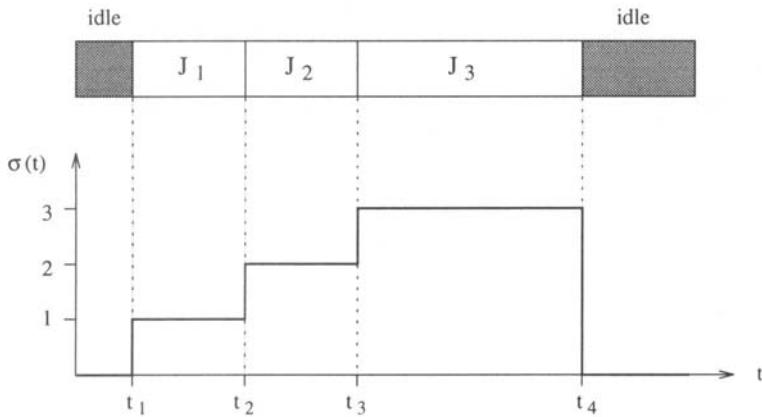


Figure 2.2 Schedule obtained by executing three tasks J_1 , J_2 , and J_3 .

- 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 intervals 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 Figure 2.3.

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.

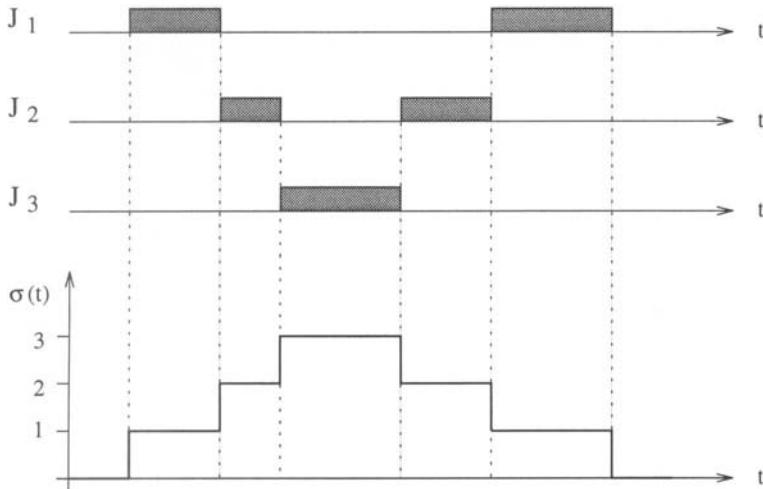


Figure 2.3 Example of a preemptive schedule.

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. Depending on the consequences of a missed deadline, real-time tasks are usually distinguished in two classes:

- **Hard.** A task is said to be hard if a completion after its deadline can cause catastrophic consequences on the system. In this case, any instance of the task should a priori be guaranteed in the worst-case scenario.
- **Soft.** A task is said to be soft if missing its deadline decreases the performance of the system but does not jeopardize its correct behavior.

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

- **Arrival time a_i :** is the time at which a task becomes ready for execution; it is also referred as *request time* or *release time* and indicated by r_i ;

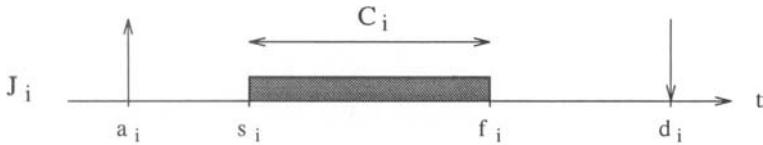


Figure 2.4 Typical parameters of a real-time task.

- **Computation time C_i :** is the time necessary to the processor for executing the task without interruption;
- **Deadline d_i :** is the time before which a task should be complete to avoid damage to the system;
- **Start time s_i :** is the time at which a task starts its execution;
- **Finishing time f_i :** is the time at which a task finishes its execution;
- **Criticalness:** is a parameter related to the consequences of missing the deadline (typically, it can be hard or soft);
- **Value v_i :** represents the relative importance of the task with respect to the other tasks in the system;
- **Lateness 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 E_i :** $E_i = \max(0, L_i)$ is the time a task stays active after its deadline;
- **Laxity or Slack time 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 Figure 2.4.

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*, that 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 job by J_i .

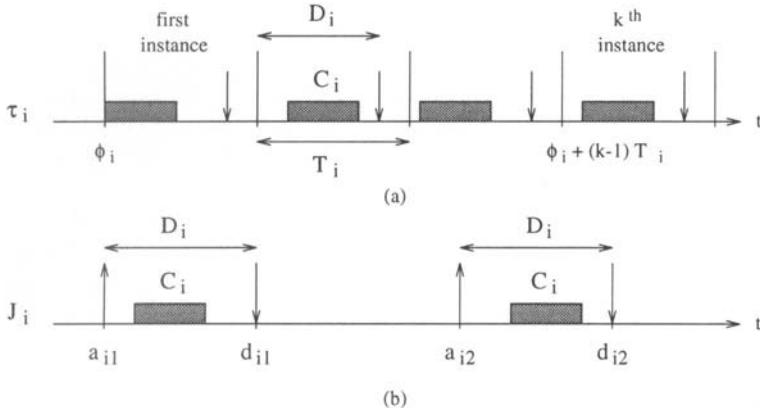


Figure 2.5 Sequence of instances for a periodic and an aperiodic task.

The activation time of the first periodic instance is called *phase*. If ϕ_i is the phase of the periodic task τ_i , the activation time of the k th instance is given by $\phi_i + (k - 1)T_i$, where T_i is called *period* of the task. In many practical cases, a periodic process can be completely characterized by its computation time C_i and its relative deadline D_i , which is often considered coincident to the end of the period. Moreover, the parameters C_i , T_i e D_i are considered to be constant for each instance. Aperiodic tasks also consist of an infinite sequence of identical activities (instances); however, their activations are not regular. Figure 2.5 shows an example of task instances for a periodic and for 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 .

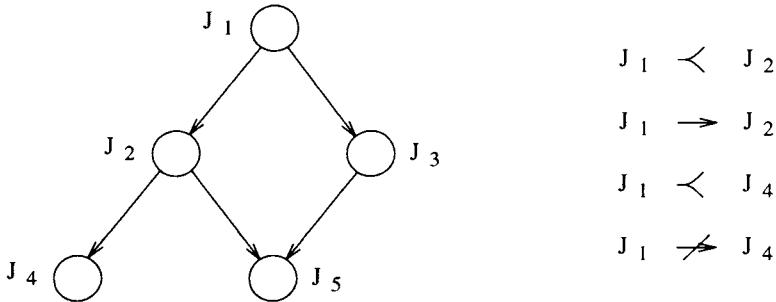


Figure 2.6 Precedence relations among five tasks.

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 Figure 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*);
- 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 as *shape*);

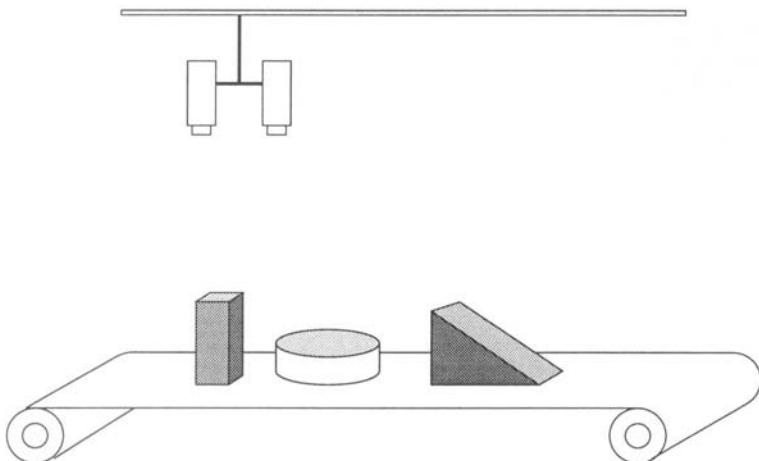


Figure 2.7 Industrial application which requires a visual recognition of objects on a conveyor belt.

- A task for computing the pixel disparities from the two images (it is referred as *disp*);
- A task for determining the object height from the results achieved by the *disp* task (it is referred 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 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 Figure 2.8.

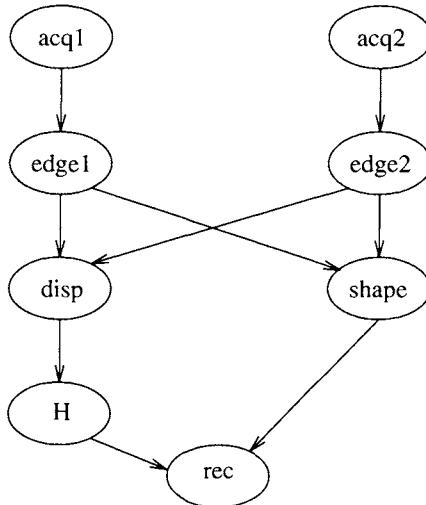


Figure 2.8 Precedence graph associated with the robotic application.

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 but require mutual exclusion among competing tasks. They are called *exclusive resources*. Let R be an exclusive resource shared by tasks J_a and J_b . If A is the operation performed on R by J_a , and B is the operation performed on R by J_b , then A and B must never be executed at the same time. A piece of code executed under mutual exclusion constraints is called a *critical section*.

To ensure sequential accesses to exclusive resources, operating systems usually provide a synchronization mechanism (such as semaphores) that can be used by tasks to create critical sections of code. Hence, when we say that two or more tasks have resource constraints, we mean that they have to be synchronized since they share exclusive resources.

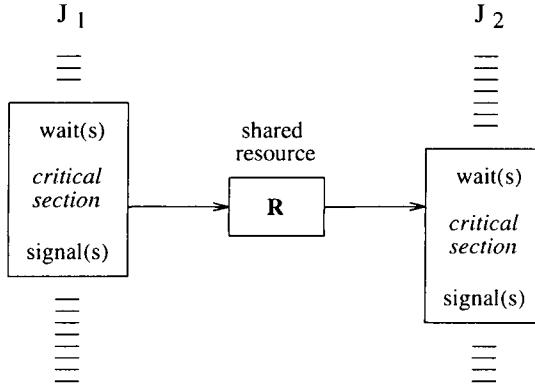


Figure 2.9 Structure of two tasks that share an exclusive resource.

Consider two tasks J_1 and J_2 that share an exclusive resource R (for instance, a list), on which two operations (such as *insert* and *remove*) are defined. The code implementing such operations is thus a critical section that must be executed in mutual exclusion. 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 (see Figure 2.9).

If preemption is allowed and J_1 has a higher priority than J_2 , then J_1 can block in the situation depicted in Figure 2.10. Here, task J_2 is activated first, and, after a while, it enters the critical section and locks the semaphore. While J_2 is executing the critical section, task J_1 arrives, and, since it has a higher priority, it preempts J_2 and starts executing. However, at time t_1 , when attempting to enter its critical section, it is blocked on the semaphore and J_2 is resumed. J_1 is blocked until time t_2 , when J_2 releases the critical section by executing the *signal(s)* primitive, which unlocks the semaphore.

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, which 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 the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Figure 2.11.

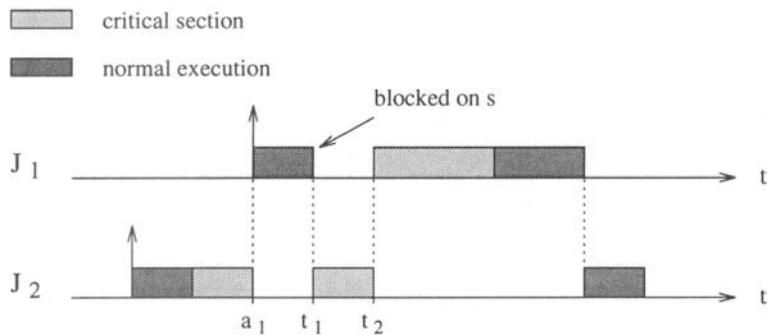


Figure 2.10 Example of blocking on an exclusive resource.

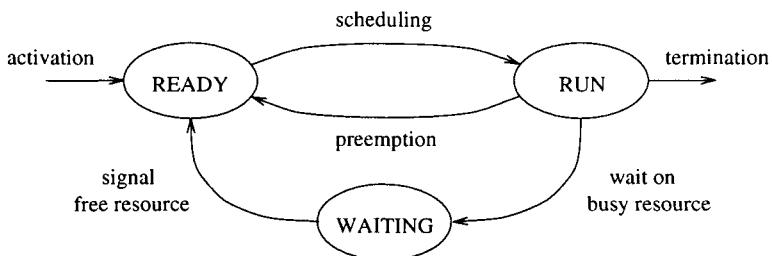


Figure 2.11 Waiting state caused by resource constraints.

2.3 DEFINITION OF SCHEDULING PROBLEMS

In general, to define a scheduling problem we need to specify three sets: a set of n tasks $J = \{J_1, J_2, \dots, J_n\}$, a set of m processors $P = \{P_1, P_2, \dots, P_m\}$ and a set of s types of resources $R = \{R_1, R_2, \dots, R_s\}$. 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 *nondeterministic* 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 $1 \mu s$. 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 $30 \mu s$, 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 (for example, 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.** With 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.
- **Non-preemptive.** With non-preemptive algorithms, a task, once started, is executed by the processor until completion. In this case, all scheduling decisions are taken as a task terminates its execution.
- **Static.** Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- **Dynamic.** Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.
- **Off-line.** We say that a scheduling algorithm is used off-line 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.
- **On-line.** 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.** 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 may fail to meet a deadline only if no other algorithms of the same class can meet it.
- **Heuristic.** An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

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.

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 precalculated off-line, and the entire schedule can be stored in a table that contains all guaranteed tasks arranged in the proper order. Then, at runtime, a simple 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 run-time 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, since new tasks can be activated at runtime, the guarantee must be done *on-line* every time a new task enters the system. A scheme of the guarantee mechanism typically adopted in dynamic real-time systems is illustrated in Figure 2.12.

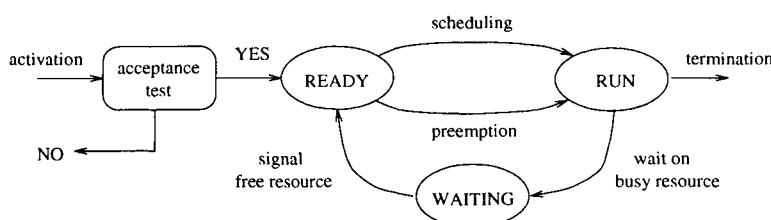


Figure 2.12 Scheme of the guarantee mechanism used in dynamic hard real-time systems.

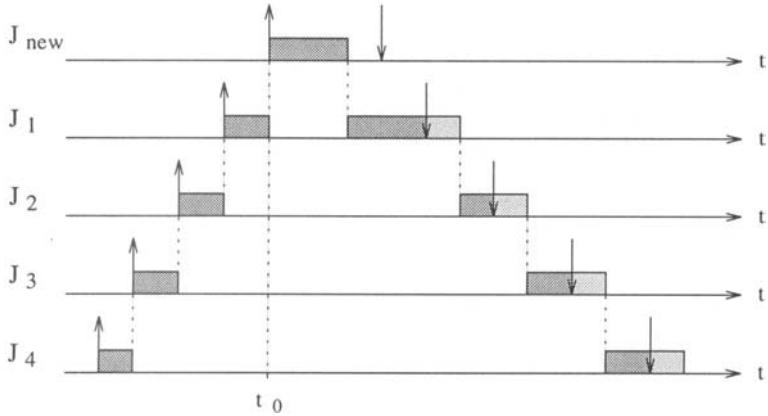


Figure 2.13 Example of domino effect.

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 to notice that, since the guarantee mechanism is based on worst-case assumptions, a task could unnecessarily be rejected. This means that the guarantee of hard tasks is achieved at the cost of reducing the average performance of the system. 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 Figure 2.13, where tasks are scheduled based on their absolute deadlines.

At time t_0 , if task J_{new} was 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.

Best-effort algorithms

In certain real-time applications, computational activities have soft timing constraints that should be met whenever possible to satisfy system requirements, however, no catastrophic events will occur if one or more tasks miss their deadlines. The only consequence associated with a timing fault is a performance degradation of the system.

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 queued 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.

Algorithms based on imprecise computation

The concept of imprecise and approximate computation has emerged as a new approach to increasing flexibility in dynamic scheduling by trading off computation accuracy with timing requirements [Nat95, LNL87, LLN87, LLS⁺91, L⁺94]. In dynamic situations, where the time and resources are not enough for computations to complete within the deadline, there may still be enough resources to produce approximate results that may at least prevent a catastrophe. The idea of using partial results when exact results cannot be produced within the deadline has been used for many years. Recently, however, this concept has been formalized, 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 J_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 J_i ; however, O_i becomes ready for execution when M_i is completed. If C_i is the computation time associated with J_i , 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 to notice 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 J_i (or simply the error of J_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 J_i is equal to

$$\epsilon_i = o_i - \sigma_i.$$

The *average error* $\bar{\epsilon}$ on the task set J is defined as

$$\bar{\epsilon} = \sum_{i=1}^n w_i \epsilon_i,$$

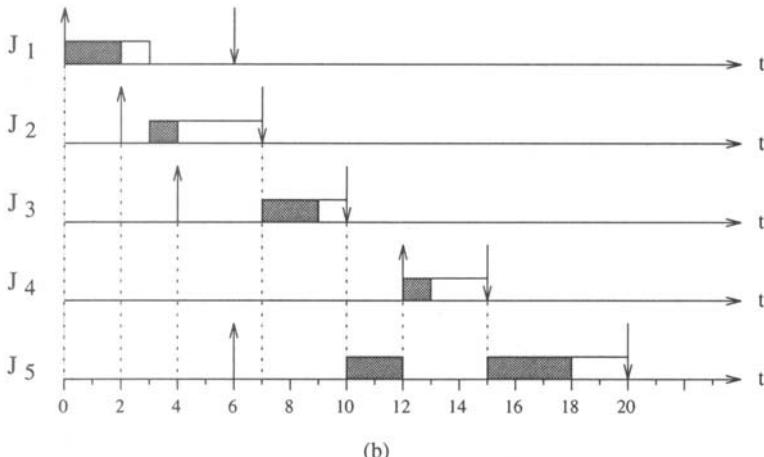
where w_i is the relative importance of J_i in the task set. An error $\epsilon_i > 0$ means that a portion of subtask O_i has been discarded in the schedule at the expense of the quality of the result produced by task J_i but for the benefit of other mandatory subtasks that can complete within their deadlines.

In this model, a schedule is said to be *feasible* if every mandatory subtask M_i is completed in the interval $[a_i, d_i]$. A schedule is said to be *precise* if the average error $\bar{\epsilon}$ on the task set is zero. In a precise schedule, all mandatory and optional subtasks are completed in the interval $[a_i, d_i]$.

As an illustrative example, let us consider the task set shown in Figure 2.14a. Notice that this task set cannot be precisely scheduled; however, a feasible schedule with an average error of $\bar{\epsilon} = 4$ can be found, and it is shown in Figure 2.14b. In fact, all mandatory subtasks finish within their deadlines,

	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)

Figure 2.14 An example of an imprecise schedule.

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 $\bar{e} = 4$.

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

Average response time:

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - a_i)$$

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

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

Table 2.1 Example of cost functions.

are considered, they are usually added as constraints, imposing that all tasks must meet their deadlines. 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 [SSDB95], 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 is not direct assessment of individual timing properties such as

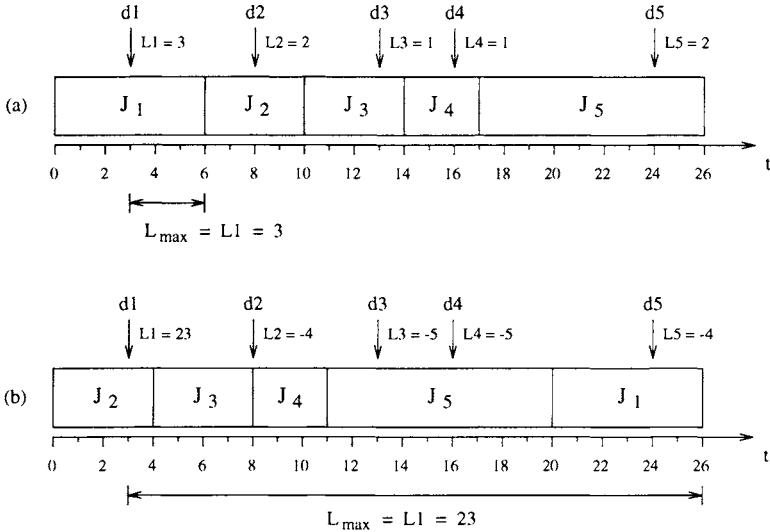


Figure 2.15 The schedule in a minimizes the maximum lateness, but all tasks miss their deadline. The schedule in b has a greater maximum lateness, but four tasks out of five complete before their deadline.

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 Figure 2.15. The schedule shown in Figure 2.15a minimizes the maximum lateness, but all tasks miss their deadline. On the other hand, the schedule shown in Figure 2.15b 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.

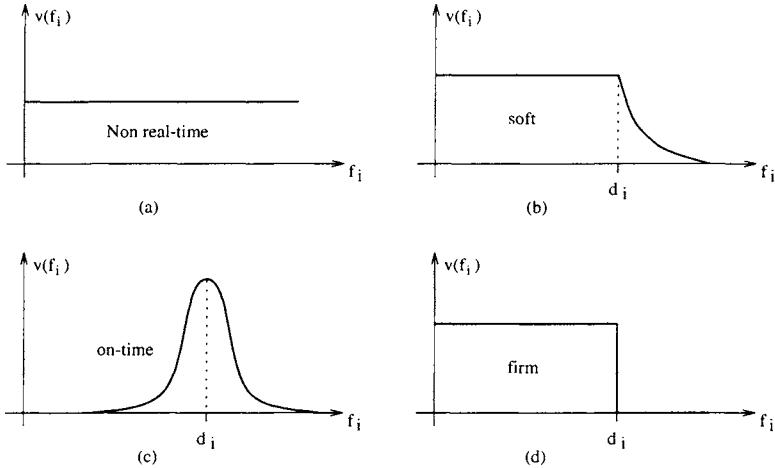


Figure 2.16 Example of cost functions for different types of tasks.

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 2.16 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 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. 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:

$$\text{Cumulative_value} = \sum_{i=1}^n v(f_i).$$

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 Chapter 8.

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, and an increase of computational power in the supporting hardware does not always cause an improvement on the performance of a task set. 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 so that they can avoid them. The most important anomalies are expressed by the following theorem [Gra76, SSDB95]:

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 (for example, 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 composed by nine tasks $J = \{J_1, J_2, \dots, J_9\}$, 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 Figure 2.17. Computation times are indicated in parentheses.

If the above set is executed on a parallel machine with three processors, we obtain the optimal schedule σ^* illustrated in Figure 2.18, 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.

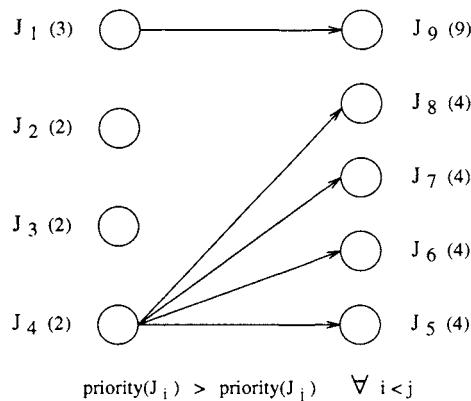


Figure 2.17 Precedence graph of the task set J ; numbers in parentheses indicate computation times.

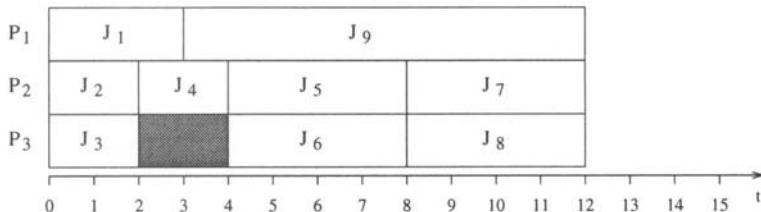


Figure 2.18 Optimal schedule of task set J on a three-processor machine.

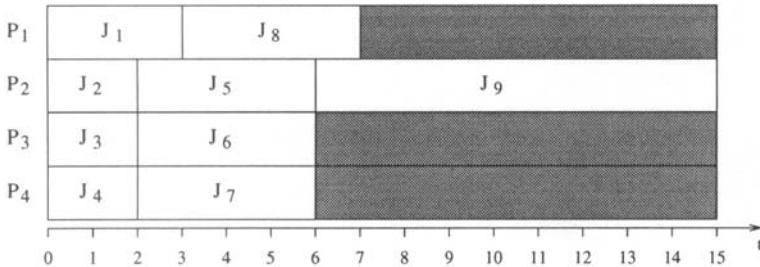


Figure 2.19 Schedule of task set J on a four-processor machine.

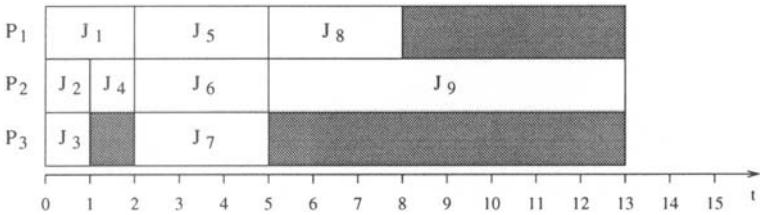


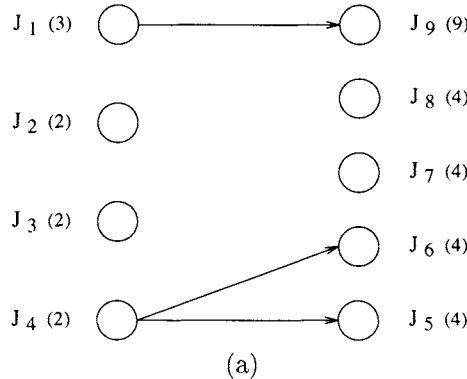
Figure 2.20 Schedule of task set J on three processors, with computation times reduced by one unit of time.

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 Figure 2.19, 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 Figure 2.20.



(a)

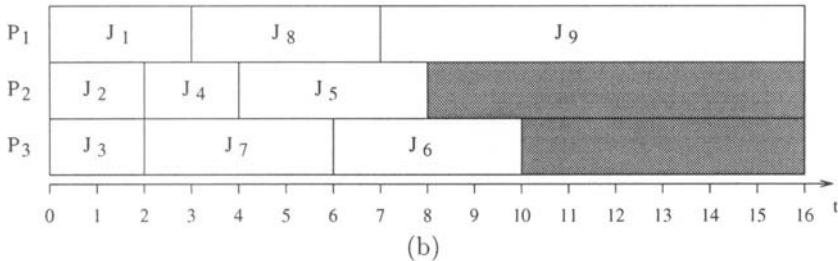


Figure 2.21 a. Precedence graph of task set J obtained by removing the constraints on tasks J_5 and J_6 . b. Schedule of task set J on three processors, with precedence constraints weakened.

Precedence constraints weakened

Scheduling anomalies can also arise if we remove precedence constraints from the directed acyclic graph depicted in Figure 2.17. For instance, if we remove the precedence relations between task J_4 and tasks J_5 and J_6 (see Figure 2.21a), we obtain the schedule shown in Figure 2.21b, which is characterized by a global completion time of $t_c = 16$ units of time.

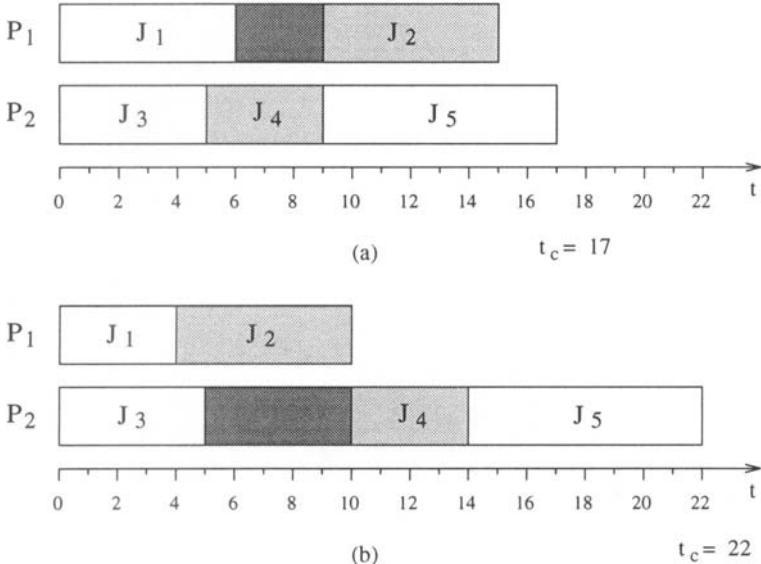


Figure 2.22 Example of anomaly under resource constraints. If J_2 and J_4 share the same resource in exclusive mode, the optimal schedule length (a) increases if the computation time of task J_1 is reduced (b). Task are statically allocated on the processors.

Anomalies under resource constraints

As a last example of scheduling anomalies, we will show how the schedule length of a task set can increase when reducing tasks' computation times in the presence of shared resources. Consider the case illustrated in Figure 2.22, 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. 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 Figure 2.22a, 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 Figure 2.22b. 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.

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], that tries to reclaim this idle time, while addressing the anomalies so that they will not occur.

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.
- 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 off-line, 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.

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 $\alpha | \beta | \gamma$, 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 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

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

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 Figure 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 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.

□

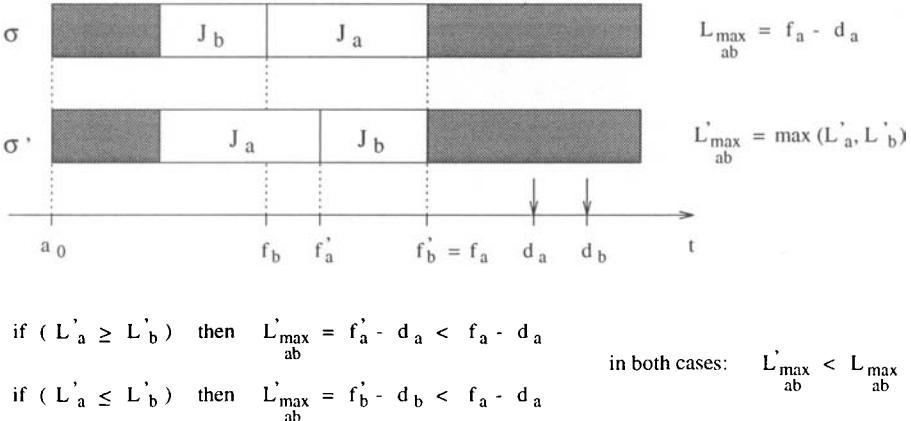


Figure 3.1 Optimality of Jackson's algorithm.

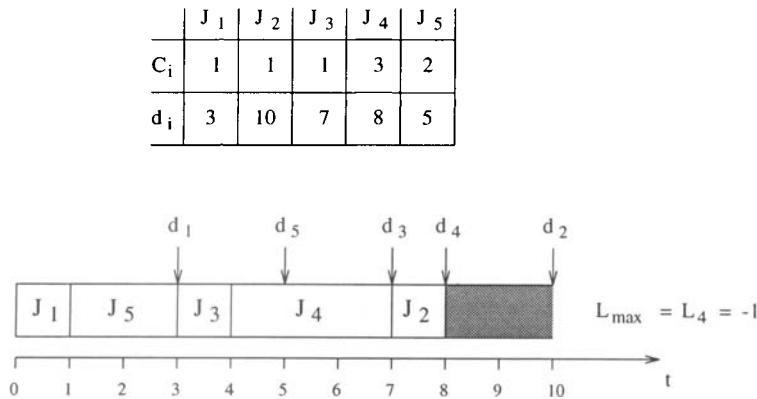
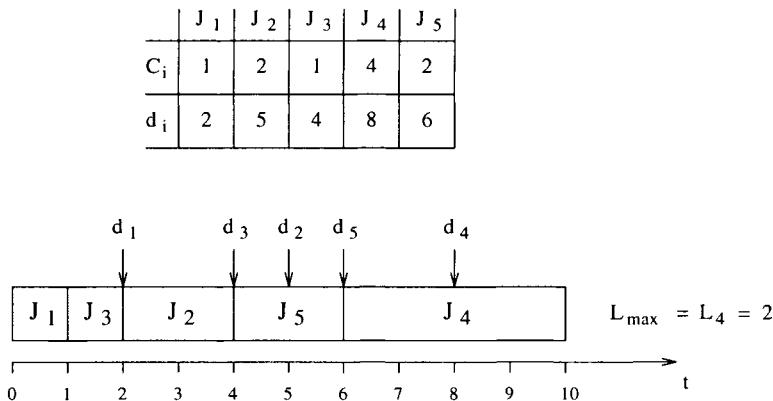
The complexity required by Jackson's algorithm to build the optimal schedule is due to the procedure that sorts the tasks by increasing deadlines. Hence, if the task set consists of n tasks, the complexity of the EDD algorithm is $O(n \log n)$.

3.2.1 Examples

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 Figure 3.2. The schedule of the tasks produced by the EDD algorithm is also depicted in Figure 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.

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.

**Figure 3.2** A feasible schedule produced by Jackson's algorithm.**Figure 3.3** An infeasible schedule produced by Jackson's algorithm.

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 deadline d_i :

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

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, \dots, J_n are listed by increasing deadlines, so that J_1 is the task with the earliest deadline. In this case, the worst-case finishing time of task J_i can be easily 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:

$$\forall i = 1, \dots, n \quad \sum_{k=1}^i C_k \leq d_i. \quad (3.1)$$

3.3 HORN'S ALGORITHM

If tasks are not synchronous but can have arbitrary arrival times (that is, 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 nonpreemptive counterpart. In a nonpreemptive 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 ($1 \mid \text{preem} \mid L_{\max}$).

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, since inserting the newly arrived task into an ordered queue (the ready queue) of n elements may require up to n steps. Hence, the overall complexity of EDF for the whole task set is $O(n^2)$.

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 Figure 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

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

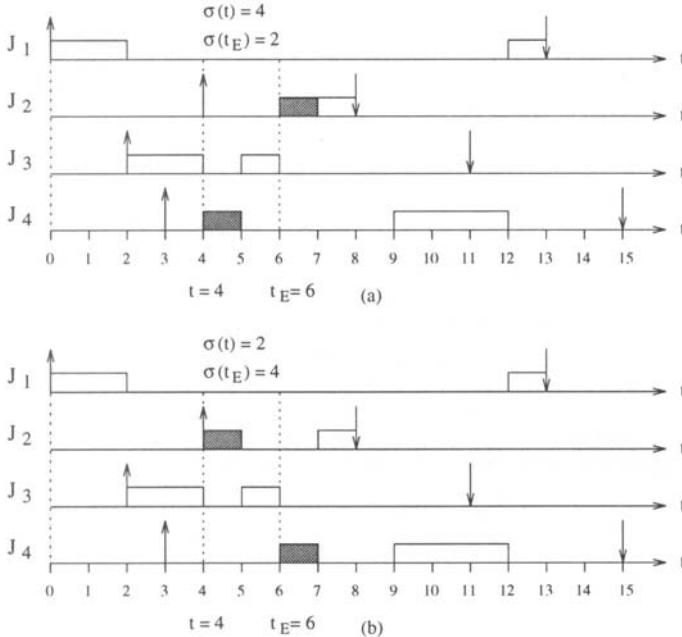


Figure 3.4 Proof of the optimality of the EDF algorithm. **a.** schedule σ at time $t = 4$. **b.** new schedule obtained after a transposition.

schedule σ starts at time $t = 0$ 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 Figure 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 Figure 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 Figure 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 $t_E = 6$. As a consequence, the two slices in gray are exchanged and the

```

Algorithm: interchange
{
    for ( $t=0$  to  $D-1$ ) {
        if ( $\sigma(t) \neq E(t)$ ) {
             $\sigma(t_E) = \sigma(t);$ 
             $\sigma(t) = E(t);$ 
        }
    }
}

```

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

resulting schedule is shown in Figure 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 Figure 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 $L_{max} = L_2 = 0$.

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

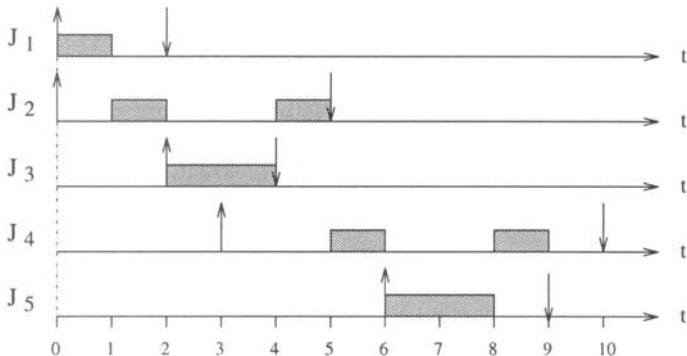


Figure 3.6 Example of EDF schedule.

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 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

```

Algorithm: EDF-guarantee( $\mathcal{J}$ ,  $J_{new}$ )
{
     $\mathcal{J}' = \mathcal{J} \cup \{J_{new}\};$       /* ordered by deadline */
    t = current_time();
     $f_0 = 0;$ 
    for (each  $J_i \in \mathcal{J}'$ ) {
         $f_i = f_{i-1} + c_i(t);$ 
        if ( $f_i > d_i$ ) return(INFEASIBLE);
    }
    return(FEASIBLE);
}

```

Figure 3.7 EDF guarantee algorithm.

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 be easily computed as

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

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

$$\forall i = 1, \dots, n \quad \sum_{k=1}^i c_k(t) \leq d_i. \quad (3.2)$$

Noting that $f_i = f_{i-1} + c_i(t)$, the dynamic guarantee test can be performed in $O(n)$ by executing the algorithm shown in Figure 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 [LRKB77, LRK77, KIM78]. Figure 3.8 illustrates an example that shows that EDF is no longer optimal if tasks cannot be

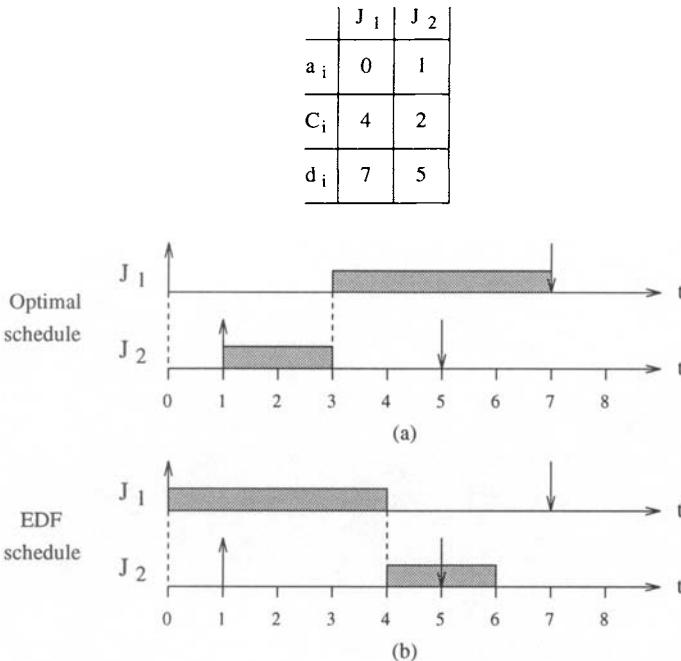


Figure 3.8 EDF is not optimal in a non-preemptive model. In fact, although there exists a feasible schedule (a), the schedule produced by EDF (b) is infeasible.

preempted during their execution. In fact, although a feasible schedule exists for that task set (see Figure 3.8a), EDF does not produce a feasible schedule (see Figure 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 Figure 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 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.

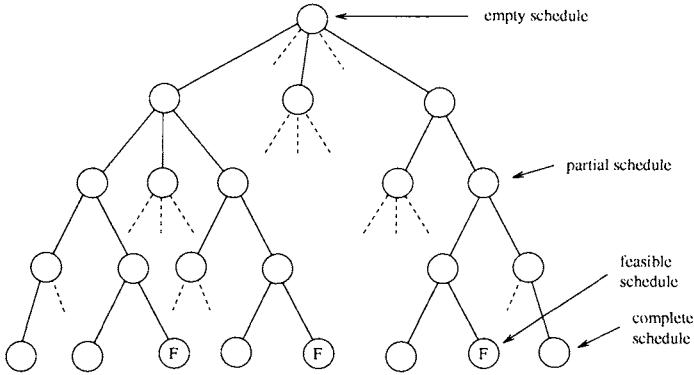


Figure 3.9 Search tree for producing a non-preemptive schedule.

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 Figure 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 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 $(1 \mid no_preem \mid feasible)$

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.

To better understand the pruning technique adopted by the algorithm, consider the task set shown in Figure 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 (\dagger) .

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 off-line 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.

	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^+ = task that misses its deadline

 = feasible schedule

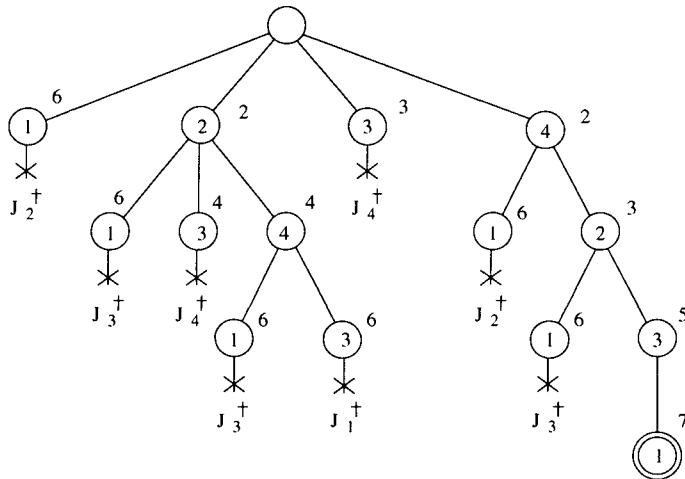


Figure 3.10 Example of search performed by Bratley's algorithm.

As in most off-line real-time systems, the resulting schedule produced by Bratley's algorithm can be stored in a data structure, called *task activation list*. Then, at run time, 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 Ramamirtham to support critical control applications in dynamic environments. The objective of the algorithm is to find a feasi-

ble 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 $R_i = [R_1(i), \dots, R_r(i)]$, 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

$$\begin{aligned} H &= d + W \cdot C \\ H &= d + W \cdot T_{est}, \end{aligned}$$

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.

$H = a$	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

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

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 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. Do to 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 run time, 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 Figure 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 $d_k \leq d_l$.

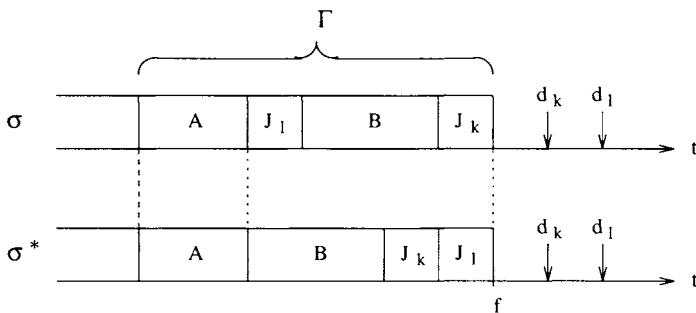


Figure 3.12 Proof of LDF optimality.

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

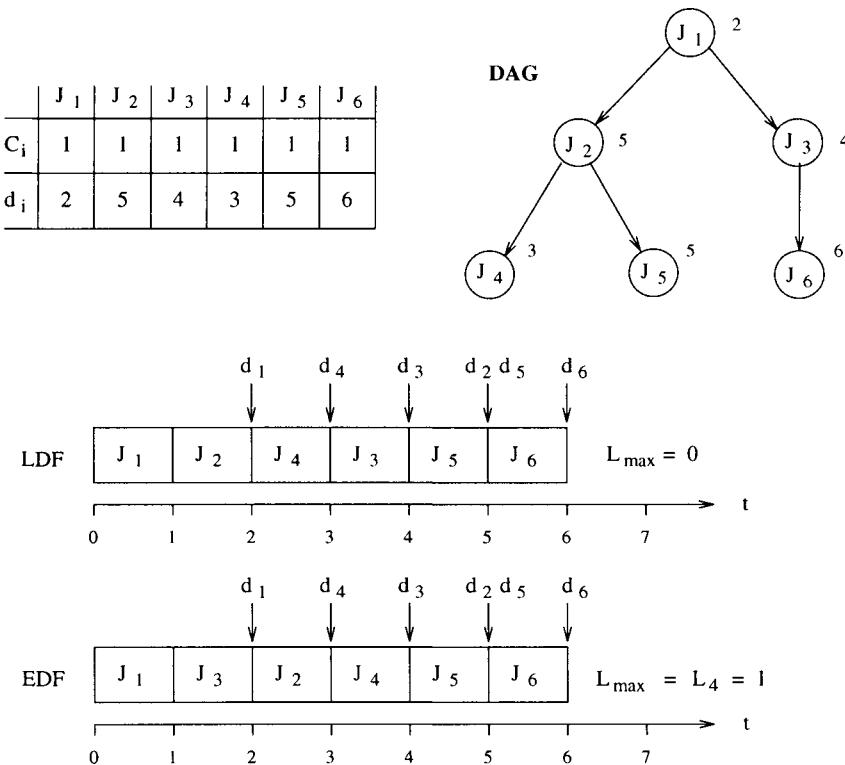


Figure 3.13 Comparison between schedules produced by LDF and EDF on a set of tasks with precedence constraints.

3.5.2 EDF with precedence constraints

$(1 \mid prec, preem \mid L_{max})$

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 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.

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 $J_a \rightarrow J_b$ (that is, 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 Figure 3.14):

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

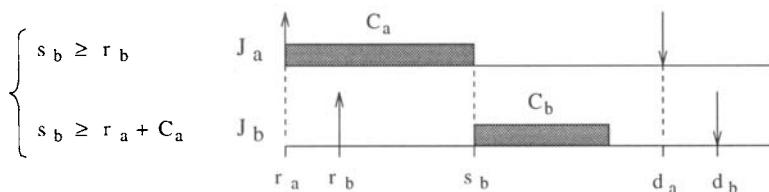


Figure 3.14 If $J_a \rightarrow J_b$, then the release time of J_b can be replaced by $\max(r_b, r_a + C_a)$.

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.

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 $J_a \rightarrow J_b$ (that is, 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 Figure 3.15):

$$\begin{aligned} f_a &\leq d_a & (\text{that is, } J_a \text{ must finish the execution within its deadline}); \\ f_a &\leq d_b - C_b & (\text{that is, } J_a \text{ must finish the execution not later than the maximum start time of } J_b). \end{aligned}$$

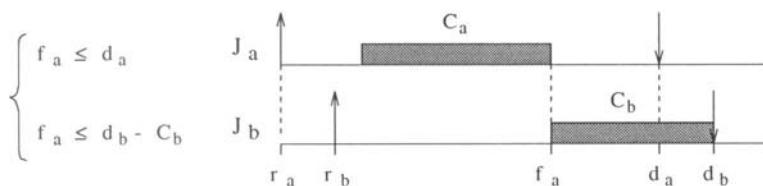


Figure 3.15 If $J_a \rightarrow J_b$, then the deadline of J_a can be replaced by $\min(d_a, d_b - C_b)$.

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.

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

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

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.

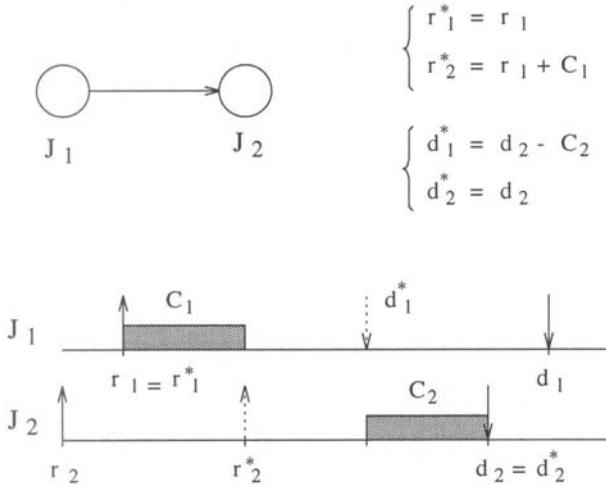


Figure 3.16 The transformation algorithm preserves the timing and the precedence constraints.

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) $O(n \log n)$ Optimal	EDF (Horn '74) $O(n^2)$ Optimal	Tree search (Bratley '71) $O(n \cdot n!)$ 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

Figure 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 the 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	15	4	7	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 $D = 20$, and computation times 2, 3, 3, 5, 1, 2, 5, respectively, modify their arrival times and deadlines to schedule them by EDF.

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 Chapter 10.

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 j th instance of task τ_i ;

$r_{i,j}$ denotes the release time of the j th 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 j th 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 j th instance of task τ_i ; that is, the time at which it starts executing.
- $f_{i,j}$ denotes the finishing time of the j th 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 Chapter 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

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

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

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

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

- **Response time** of an instance. It is the time (measured from the release time) at which the instance is terminated:

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

- **Critical instant** of a task. It is the time at which the release of a task will produce the largest 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 Release 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 Release 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 (that is, 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. 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:

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

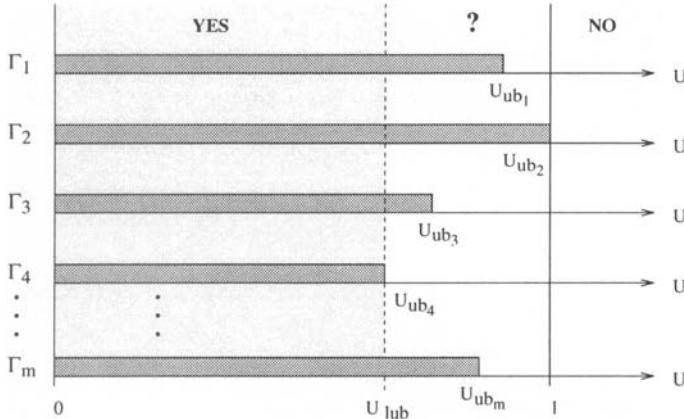


Figure 4.1 Meaning of the least upper bound of the processor utilization factor.

Figure 4.1 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 below this bound is schedulable by the algorithm. On the other hand, utilization above this bound can be achieved only if the periods of the tasks are suitably related.

If the utilization factor of a task set is greater than one, the task set cannot be scheduled by any algorithm. To show this result, let T be the product of all the periods: $T = T_1 T_2 \dots T_n$. If $U > 1$, we also have $UT > T$, which can be written as

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

The factor (T/T_i) represents the number of times that τ_i is executed in the interval T , whereas the quantity $(T/T_i)C_i$ is the total computation time requested by τ_i in the interval T . Hence, the sum on the left hand side represents the total demand of computation time requested by all tasks in T . Clearly, if the total demand exceeds the available processor time, there is no feasible schedule for the task set.

4.2 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 (that is, with shorter periods) will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: priorities are assigned to tasks before execution and do 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.

4.2.1 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 $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ 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 Figure 4.2a, the response time of task τ_n is delayed by the interference of τ_i with higher priority. Moreover, from Figure 4.2b it is clear that advancing the release time of τ_i may increase the completion time of τ_n . As a consequence, the response time of τ_n is largest when it is released simultaneously with τ_i . Repeating the argument for all τ_i , $i = 2, \dots, n - 1$, we prove

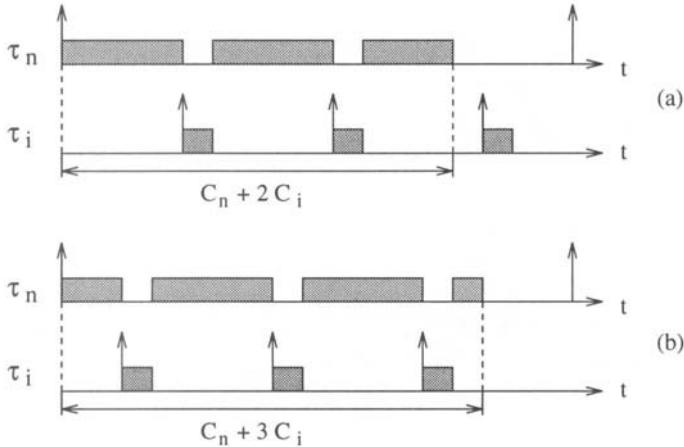


Figure 4.2 a. The response time of task τ_n is delayed by the interference of τ_i with higher priority. b. The interference may increase advancing the release time of τ_i .

that the worst response time of a task occurs when it is released simultaneously with all higher-priority tasks.

A first consequence of this result is that task schedulability can easily be checked at their critical instants. Specifically, if all tasks are feasible at their critical instants, then the task set is schedulable in any other condition. Based on this result, the optimality of RM is justified by showing that 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 Figure 4.3, from which it is easy to see that, at critical instants, the schedule is feasible if the following inequality is satisfied:

$$C_1 + C_2 \leq T_1. \quad (4.1)$$

On the other hand, if priorities are assigned according to RM, task T_1 will receive the highest priority. In this situation, illustrated in Figure 4.4, in order to guarantee a feasible schedule two cases must be considered. Let $F = \lfloor T_2/T_1 \rfloor$ be the number¹ of periods of τ_1 entirely contained in T_2 .

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 .

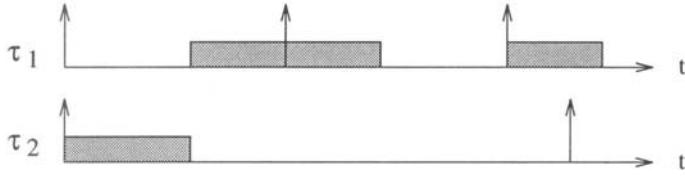


Figure 4.3 Tasks scheduled by an algorithm different from RM.

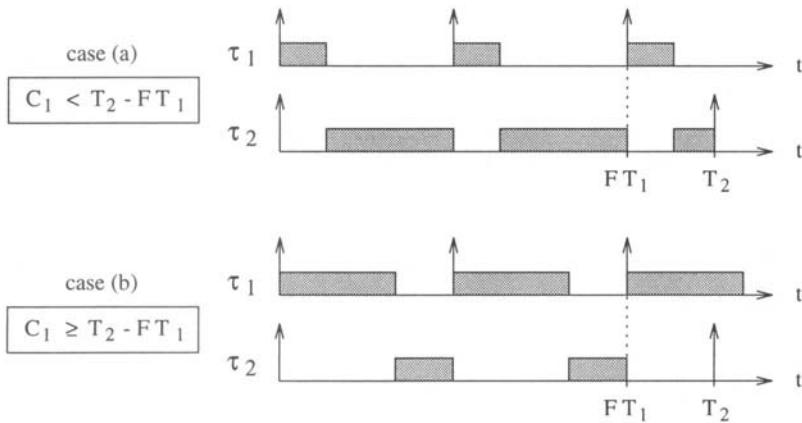


Figure 4.4 Schedule produced by RM in two different conditions.

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 Figure 4.4a we can see that the task set is schedulable if

$$(F + 1)C_1 + C_2 \leq T_2. \quad (4.2)$$

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

$$FC_1 + FC_2 \leq FT_1,$$

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 Figure 4.4b we can see that the task set is schedulable if

$$FC_1 + C_2 \leq FT_1. \quad (4.3)$$

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.

4.2.2 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 Figure 4.5, 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

$$C_1 = T_2 - FT_1.$$

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 Figure 4.6, the largest possible value of C_2 is

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

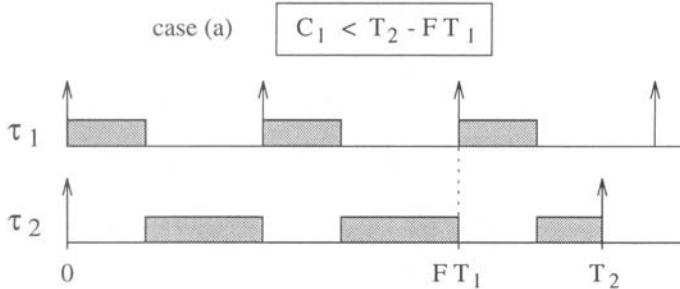


Figure 4.5 The second request of τ_2 is released when τ_1 is idle.

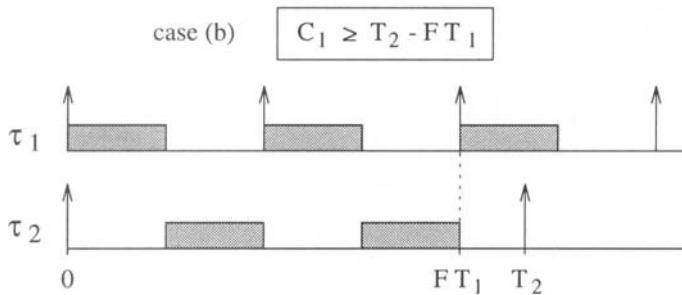


Figure 4.6 The second request of τ_2 is released when τ_1 is active.

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

$$C_1 = T_2 - T_1F.$$

Hence, using the minimum value of C_1 , from equation (4.4) we have

$$\begin{aligned} U &= \frac{T_1}{T_2}F + \frac{C_1}{T_2} \left(\frac{T_2}{T_1} - F \right) = \\ &= \frac{T_1}{T_2}F + \frac{(T_2 - T_1F)}{T_2} \left(\frac{T_2}{T_1} - F \right) = \\ &= \frac{T_1}{T_2} \left[F + \left(\frac{T_2}{T_1} - F \right) \left(\frac{T_2}{T_1} - F \right) \right]. \end{aligned} \quad (4.5)$$

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

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

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,

$$U = \frac{1 + G^2}{1 + G}. \quad (4.7)$$

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 equation (4.7), the least upper bound of U is given for $G = G_2$:

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

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

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

That is,

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

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 equation (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 Figure 4.7 shows the upper bound of U as a function of k .

4.2.3 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

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

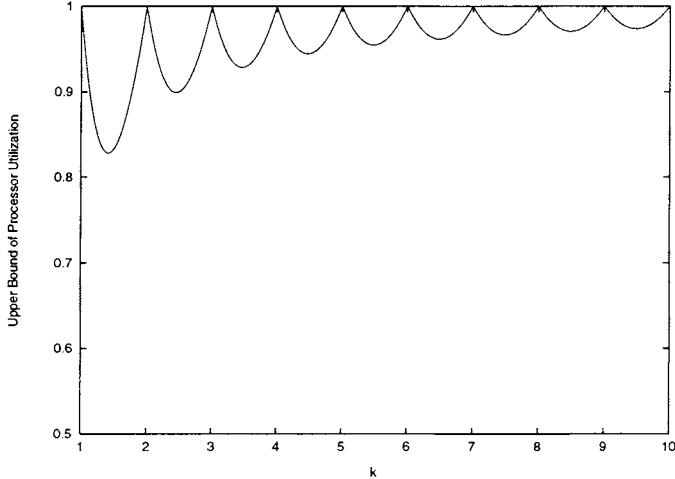


Figure 4.7 Upper bound of the processor utilization factor as a function of the ratio $k = T_2/T_1$.

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

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

Defining

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

and noting that

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

the utilization factor may be written as

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

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

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

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

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

That is, when all R_i have the same value:

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

Substituting this value in U we obtain

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

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

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

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

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

n	U_{lub}
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743

n	U_{lub}
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

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

and since (by the Hospital's rule)

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

we have that

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

4.2.4 Concluding remarks on RM

To summarize the most important results derived in this section, the Rate-Monotonic algorithm has been proved to be 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. Moreover, RM guarantees that an arbitrary set of periodic tasks is schedulable if the total processor utilization U does not exceed a value of 0.69.

Notice that this schedulability condition is sufficient to guarantee the feasibility of any task set, but it is not necessary. This means that, if a task set has an utilization factor greater than U_{lub} and less than one, nothing can be said on the feasibility of the set. A sufficient and necessary condition for the schedulability under RM has been derived by Audsley et al. [ABRW91] for the more general case of periodic tasks with relative deadlines less than periods, and it is presented in Section 4.4.

A simulation study carried out by Lehoczky, Sha, and Ding [LSD89] showed that for random task sets the processor utilization bound is approximately 0.88. However, since RM is optimal among all static assignments, an improvement of the processor utilization bound can be achieved only by using dynamic scheduling algorithms.

4.3 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 j th 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 Chapter 3 for aperiodic tasks, also holds for periodic tasks.

4.3.1 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 4.1 *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 $T = T_1 T_2 \dots T_n$, 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.

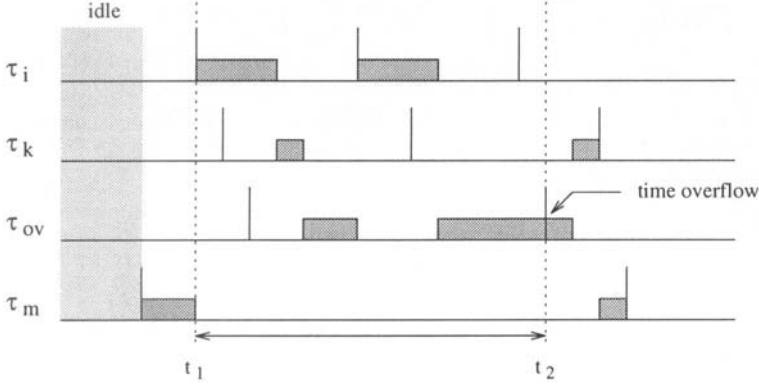


Figure 4.8 Interval of continuous utilization in an EDF schedule before a time-overflow.

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 Figure 4.8 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 demanded by periodic tasks in $[t_1, t_2]$, which can be computed as

$$C_p(t_1, t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k = \sum_{i=1}^n \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i. \quad (4.10)$$

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. \square

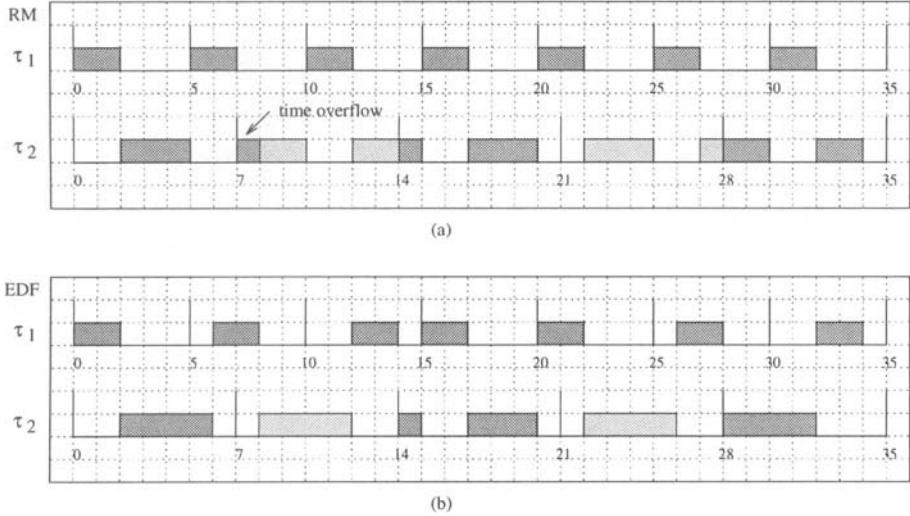


Figure 4.9 Schedule produced by RM (a) and EDF (b) on the same set of periodic tasks.

4.3.2 An example

Consider the periodic task set illustrated in Figure 4.9, for which the processor utilization factor is

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

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 Figure 4.9a, RM generates a time-overflow at time $t = 7$, whereas EDF completes all tasks within their deadlines (see Figure 4.9b). Another important difference between RM and EDF concerns the number of preemptions occurring in the schedule. As shown in Figure 4.9, 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.

4.4 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 a relative deadline less than their period. 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 Figure 4.10 and have the following relationships:

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

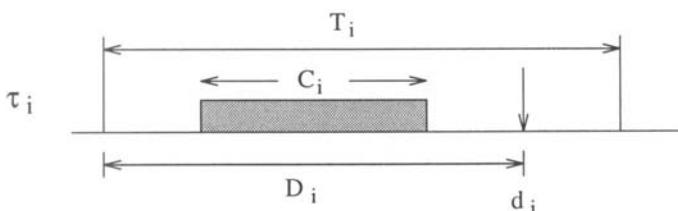


Figure 4.10 Task parameters in Deadline-Monotonic scheduling.

According to the DM algorithm, each task is assigned a priority inversely proportional to its relative deadline. 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 preemptive; 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 any static priority scheduling algorithm can schedule a set of tasks with deadlines unequal to their periods, then DM will also schedule that task set.

4.4.1 Schedulability analysis

The feasibility of a set of tasks with deadlines unequal to their periods could be guaranteed using the Rate-Monotonic schedulability 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 be optimal as 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 (pre-emption) 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

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

such a test is given by

$$\forall i : 1 \leq i \leq n \quad C_i + I_i \leq D_i, \tag{4.11}$$

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

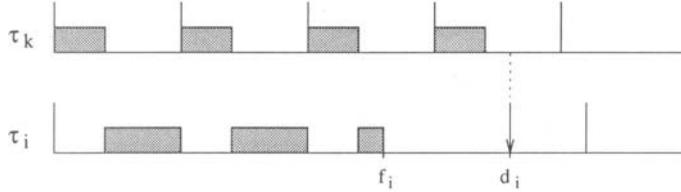


Figure 4.11 More accurate calculation of the interference on τ_i by higher priority tasks.

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 :

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j.$$

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_j exactly interferes $\lceil \frac{D_i}{T_j} \rceil$ times on τ_i . However, as shown in Figure 4.11, 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 interleaving of higher-priority tasks must be evaluated for each process. In general, this procedure is quite costly since, for each task τ_i , it requires the construction of the schedule until D_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.

4.4.2 Sufficient and necessary schedulability test

According to the method proposed by Audsley et 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 due to preemption by higher-priority tasks:

$$R_i = C_i + I_i,$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j.$$

Hence,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \quad (4.12)$$

No simple solution exists for this equation since R_i appears on both sides. Thus, the worst-case response time of task τ_i is given by the smallest value of R_i that satisfies equation (4.12). Notice, however, that only a subset of points in the interval $[0, D_i]$ need to be examined for feasibility. In fact, the interference on τ_i only increases when there is a release of a higher-priority task.

To simplify the notation, let R_i^k be the k th estimate of R_i and let I_i^k be the interference on task τ_i in the interval $[0, R_i^k]$:

$$I_i^k = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j. \quad (4.13)$$

Then the calculation of R_i is performed as follows:

1. Iteration starts with $R_i^0 = C_i$, which is the first point in time that τ_i could possibly complete.
2. The actual interference I_i^k in the interval $[0, R_i^k]$ is computed by equation (4.13).
3. If $I_i^k + C_i = R_i^k$, then R_i^k is the actual worst-case response time of task τ_i ; that is, $R_i = R_i^k$. Otherwise, the next estimate is given by

$$R_i^{k+1} = I_i^k + C_i,$$

and the iteration continues from step 2.

Once R_i is calculated, the feasibility of task τ_i is guaranteed if and only if $R_i \leq D_i$.

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 Figure 4.12, and the iteration steps are shown below.

	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

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

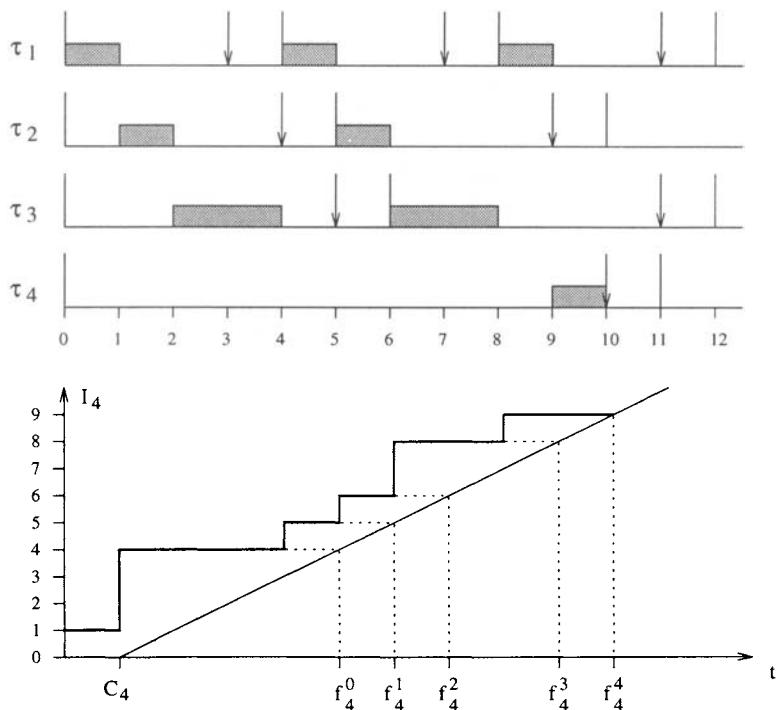


Figure 4.12 Example of schedule produced by DM.

- Step 0: $R_4^0 = C_4 = 1$, but $I_4^0 = 4$ and $I_4^0 + C_4 > R_4^0$, hence τ_4 does not finish at R_4^0 .
- Step 1: $R_4^1 = I_4^0 + C_4 = 5$, but $I_4^1 = 5$ and $I_4^1 + C_4 > R_4^1$, hence τ_4 does not finish at R_4^1 .
- Step 2: $R_4^2 = I_4^1 + C_4 = 6$, but $I_4^2 = 6$ and $I_4^2 + C_4 > R_4^2$, hence τ_4 does not finish at R_4^2 .
- Step 3: $R_4^3 = I_4^2 + C_4 = 7$, but $I_4^3 = 7$ and $I_4^3 + C_4 > R_4^3$, hence τ_4 does not finish at R_4^3 .
- Step 4: $R_4^4 = I_4^3 + C_4 = 9$, but $I_4^4 = 9$ and $I_4^4 + C_4 > R_4^4$, hence τ_4 does not finish at R_4^4 .
- Step 5: $R_4^5 = I_4^4 + C_4 = 10$, but $I_4^5 = 9$ and $I_4^5 + C_4 = R_5^4$ hence τ_4 finishes at $R_4 = 10$.

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 Figure 4.13.

```

DM_guarantee ( $\Gamma$ ) {
  for (each  $\tau_i \in \Gamma$ ) {
     $I = 0$ ;
    do {
       $R = I + C_i$ ;
      if ( $R > D_i$ ) return(UNSCHEDULABLE);
       $I = \sum_{j=1}^{i-1} \left\lceil \frac{R}{T_j} \right\rceil C_j$ ;
    } while ( $I + C_i > R$ );
  }
  return(SCHEDULABLE);
}

```

Figure 4.13 Algorithm for testing the schedulability of a periodic task set Γ under Deadline Monotonic.

4.5 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.

4.5.1 The processor demand approach

In general, the processor demand of a task τ_i in any interval $[t, t + L]$ is the amount of processing time required by τ_i in $[t, t + L]$ that has to complete at or before $t + L$. In a deadline-based system, it is the processing time required in $[t, t + L]$ that has to be executed with deadlines less than or equal to $t + L$.

For a set of periodic tasks (with deadlines equal to periods) invoked at time $t = 0$ the cumulative processor demand in any interval $[0, L]$ is the total amount of processing time $C_P(0, L)$ that has to be executed with deadlines less than or equal to L . That is,

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

Given this definition, the schedulability of a periodic task set is guaranteed if and only if the cumulative processor demand in any interval $[0, L]$ is less than the available time; that is, the interval length L . This is stated by the following theorem:

Theorem 4.2 (Jeffay and Stone) *A set of periodic tasks is schedulable by EDF if and only if for all L , $L \geq 0$,*

$$L \geq \sum_{i=1}^n \left\lfloor \frac{L}{T_i} \right\rfloor C_i. \quad (4.14)$$

Proof. The theorem is proved by showing that equation (4.14) is equivalent to the classical Liu and Layland's condition

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1. \quad (4.15)$$

(4.15) \Rightarrow (4.14). 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 (4.15) \Leftarrow (4.14) we show that $\neg(4.15) \Rightarrow \neg(4.14)$. That is, we assume $U > 1$ and prove that there exist an $L \geq 0$ for which (4.14) does not hold. If $U > 1$, then for $L = lcm(T_1, \dots, T_n)$,

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

□

Notice that to apply Theorem 4.2 it suffices to test equation (4.14) only for values of L equal to release times less than the hyperperiod H . In fact, if equation (4.14) holds for $L = r_k$, it will also hold for any $L \in [r_k, r_{k+1})$, since

$$\forall L \in [r_k, r_{k+1}), \quad \left\lfloor \frac{L}{T_i} \right\rfloor = \left\lfloor \frac{r_k}{T_i} \right\rfloor.$$

The values of L for which equation (4.14) has to be tested can still be reduced to the set of release times within the busy period. The *busy period* is the smallest interval $[0, L]$ in which the total processing time $W(L)$ requested in $[0, L]$ is completely executed. The quantity $W(L)$ can be computed as

$$W(L) = \sum_{i=1}^n \left\lfloor \frac{L}{T_i} \right\rfloor C_i. \quad (4.16)$$

Thus, the busy period B_p can be defined as

$$B_p = \min\{L \mid W(L) = L\}$$

and computed by the algorithm shown in Figure 4.14.

Notice that, when the system is overloaded, the processor is always busy and the busy period is equal to infinity. On the other hand, if the system is not

```

busy_period {
     $L = \sum_{i=1}^n C_i;$ 
     $L' = W(L);$ 
     $H = lcm(T_1, \dots, T_n);$ 
    while ( $L' \neq L$ ) and ( $L' \leq H$ ) {
         $L = L';$ 
         $L' = W(L);$ 
    }
    if ( $L' \leq H$ )  $B_p = L;$ 
    else  $B_p = \text{INFINITY};$ 
}

```

Figure 4.14 Algorithm for computing the busy period.

overloaded, the busy period coincides either with the beginning of an idle time (see Figure 4.15a) or with the release of a periodic instance (see Figure 4.15b).

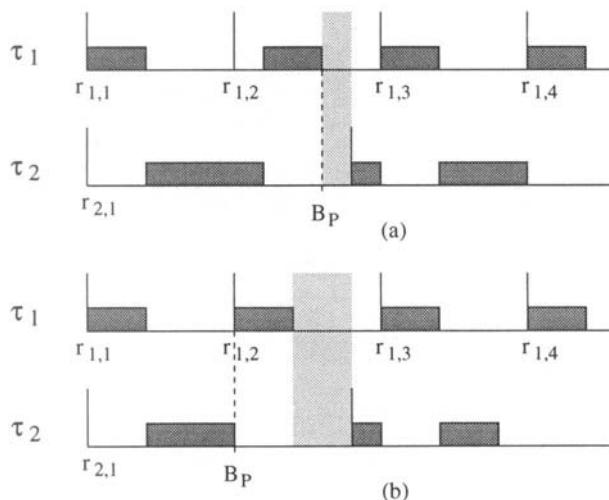


Figure 4.15 Examples of finite busy periods.

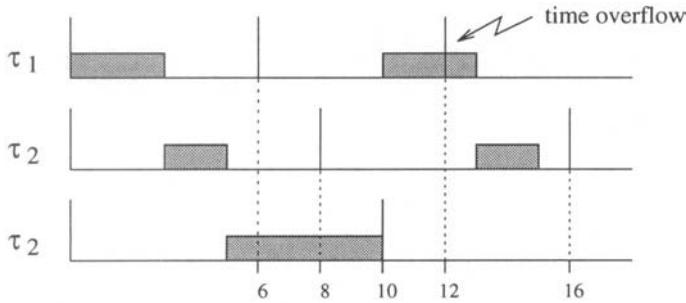


Figure 4.16 Examples of processor demand analysis.

L	$C_P(0, L)$	result
6	3	OK
8	5	OK
10	10	OK
12	13	NO

Table 4.4 Results of the processor demand criterion.

Based on the previous observations, to apply Theorem 4.2, equation (4.14) can be tested for all $L \in \mathcal{R}$, where

$$\mathcal{R} = \{r_{i,j} \mid r_{i,j} \leq \min(B_p, H), 1 \leq i \leq n, j \geq 1\}.$$

Example

To illustrate the processor demand criterion, consider the example shown in Figure 4.16, where three periodic tasks with periods 6, 8, 10, and processing times 3, 2, 5, respectively, are executed under EDF. In this case, the set checking points for equation (4.14) is given by $\mathcal{R} = \{6, 8, 10, 12, 16, \dots\}$. Applying Theorem 4.2 we have the results shown in Table 4.4.

4.5.2 Deadlines less than periods

The processor demand criterion can easily be extended to deal with tasks with deadlines different than periods. For example, consider the two tasks shown in Figure 4.17. In this case, the processor demands for tasks τ_1 and τ_2 in $[0, L]$ are clearly given by

$$\begin{cases} C_1(0, L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1 \\ C_2(0, L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2. \end{cases}$$

In general, we can write

$$C_i(0, L) = \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i. \quad (4.17)$$

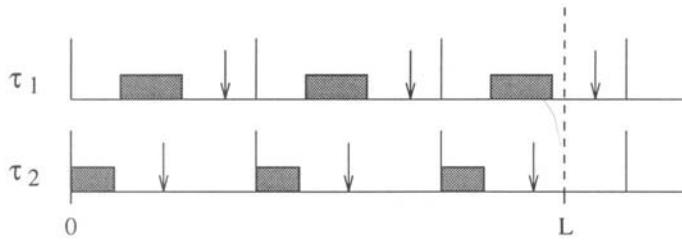


Figure 4.17 Processor demand when deadlines are less than periods.

In summary, the schedulability of a generic task set can be tested by the following theorem [BRH90], whose proof is very similar to the one shown for Theorem 4.2.

Theorem 4.3 *If $\mathcal{D} = \{d_{i,k} \mid d_{i,k} = kT_i + D_i, d_{i,k} \leq \min(B_p, H), 1 \leq i \leq n, k \geq 0\}$, then a set of periodic tasks with deadlines less than periods is schedulable by EDF if and only if*

$$\forall L \in \mathcal{D} \quad L \geq \sum_{i=1}^n \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i. \quad (4.18)$$

4.6 SUMMARY

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 Rate-Monotonic (RM) algorithm is optimal among all fixed-priority assignments, whereas the Earliest Deadline First (EDF) algorithm is optimal among all dynamic priority assignments. When deadlines are equal to periods, the guarantee test for both algorithms can be performed in $O(n)$ (being n the number of periodic tasks in the set), using the processor utilization approach. The test for RM, however, provides only a sufficient condition for guaranteeing the feasibility of the schedule.

In the general case in which deadlines can be less or equal to periods, the schedulability analysis becomes more complex and can be performed in pseudo-polynomial time [BRH90]. Under fixed-priority assignments, the feasibility of the task set can be tested using the response time approach, which uses a recurrent formula to calculate the worst-case finishing time of any task. Under dynamic priority assignments, the feasibility can be tested using the processor demand approach. In both cases the test provides a necessary and sufficient condition. The various methods are summarized in Figure 4.18.

	$D_i = T_i$	$D_i \leq T_i$
Static priority	RM Processor utilization approach $U \leq n(2^{\lfloor \frac{I}{T_i} \rfloor} - 1)$	DM Response time approach $\forall i \quad R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$
Dynamic priority	EDF Processor utilization approach $U \leq I$	EDF * Processor demand approach $\forall L > 0 \quad L \geq \sum_{i=1}^n \left(\left\lceil \frac{L - D_i}{T_i} \right\rceil + 1 \right) C_i$

Figure 4.18 Summary of guarantee tests for periodic tasks.

Exercises

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

	τ_1	τ_2
C_i	1	1
T_i	3	4

- 4.2 Given the following set of periodic tasks

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	10

verify the schedulability under RM using the processor utilization approach. Then, perform the worst-case response time analysis and construct the schedule.

- 4.3 Verify the schedulability under RM and construct the schedule of the following task set:

	τ_1	τ_2	τ_3
C_i	1	2	3
T_i	4	6	8

- 4.4 Verify the schedulability under EDF of the task set shown in Exercise 4.3, and then construct the corresponding schedule.

- 4.5 Compute the busy period for the task set described in Exercise 4.2.

- 4.6 Compute the busy period for the task set described in Exercise 4.3.

- 4.7 Verify the schedulability under EDF and construct the schedule of the following task set:

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

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

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 criticalness. 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 on-line guarantee of individual aperiodic requests can still be done. Aperiodic tasks requiring on-line 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 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.

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.

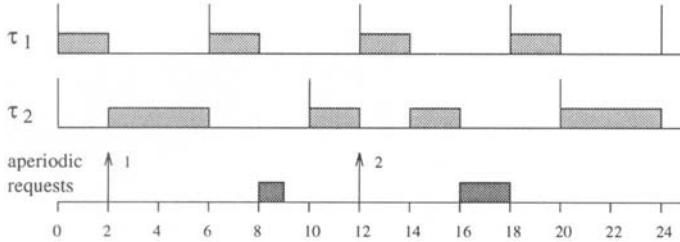


Figure 5.1 Example of background scheduling of aperiodic requests under Rate Monotonic.

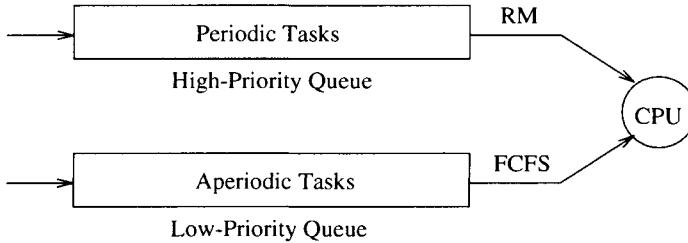


Figure 5.2 Scheduling queues required for background scheduling.

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

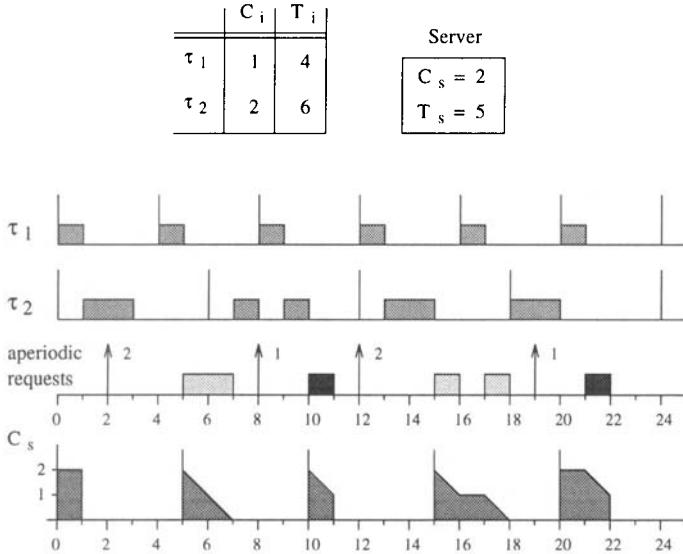


Figure 5.3 Example of a Polling Server scheduled by RM.

server *capacity*. 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 server capacity. 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 any 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 time originally allocated for aperiodic service is not preserved for aperiodic execution but is used by 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 polling 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.

In the example shown in Figure 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

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

A more precise schedulability test for aperiodic servers that behave like a periodic task will be derived in Section 5.5 for the Priority Exchange algorithm. 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 guaranteed if

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

5.3.2 Aperiodic guarantee

In order to analyze the schedulability of firm aperiodic activities under a Polling Server, consider the case of a single aperiodic request J_a , arrived at time r_a , with computation time C_a and deadline D_a . Since an aperiodic request 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 found for the case in which 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, so that the finishing time of the aperiodic request can be estimated precisely. As shown in Figure 5.4, by defining

$$\begin{aligned} F_a &= \left\lceil \frac{C_a}{C_s} \right\rceil \\ G_a &= \left\lceil \frac{r_a}{T_s} \right\rceil \end{aligned}$$

the initial delay of request J_a is given by $(G_a T_s - r_a)$. Then, since $F_a C_s$ is the total capacity consumed by J_a in F_a server periods, the residual execution to be done in the next server period is

$$R_a = C_a - F_a C_s.$$

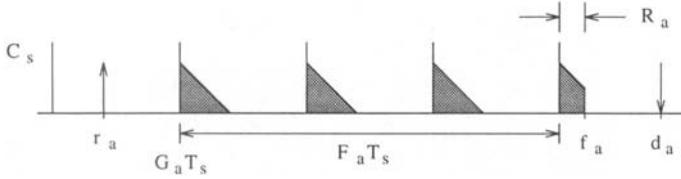


Figure 5.4 Calculation of the finishing time of an aperiodic request scheduled by a Polling Server having the highest priority.

As a consequence, the aperiodic finishing time can be computed as

$$f_a = G_a T_s + F_a T_s + R_a,$$

and its schedulability can be guaranteed if and only if $f_a \leq d_a$, being d_a the absolute deadline of the request ($d_a = r_a + D_a$). Thus, the resulting schedulability condition is

$$(F_a + G_a)T_s + R_a \leq d_a.$$

This result can be extended to a set of firm aperiodic requests ordered in a queue by increasing deadline. In this case, at any time t , the total aperiodic computation that has to be served in any interval $[t, d_k]$ is equal to the sum of the remaining processing times $c_i(t)$ of the tasks with deadline $d_i \leq d_k$; that is,

$$C_{ape}(t, d_k) = \sum_{i=1}^k c_i(t).$$

Note that, if $c_s(t)$ is the residual server capacity at time t and PS has the highest priority, a portion of C_{ape} equal to $c_s(t)$ is immediately executed in the current period. Hence, the finishing time of request J_k can be computed as

$$f_k = \begin{cases} t + C_{ape}(t, d_k) & \text{if } C_{ape}(t, d_k) \leq c_s(t) \\ (F_k + G_k)T_s + R_k & \text{otherwise,} \end{cases}$$

where

$$\begin{cases} F_k = \left\lfloor \frac{C_{ape}(t, d_k) - c_s(t)}{C_s} \right\rfloor \\ G_k = \left\lceil \frac{t}{T_s} \right\rceil \\ R_k = C_{ape}(t, d_k) - c_s(t) - F_k C_s. \end{cases}$$

Once all finishing times have been calculated, the set of firm aperiodic requests is guaranteed at time t if and only if

$$f_k \leq d_k \quad \forall k = 1, \dots, n.$$

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 Figure 5.5 using the same task set and the same server parameters ($C_s = 2$, $T_s = 5$) considered in Figure 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 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 Figure 5.6. 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 must execute whenever it is the highest-priority task ready to run. It is easy to see that the De-

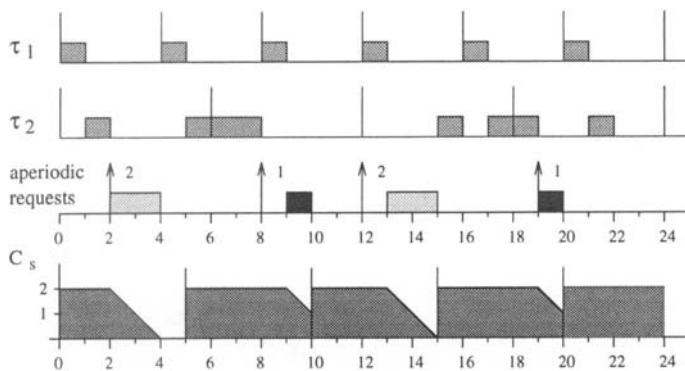


Figure 5.5 Example of a Deferrable Server scheduled by RM.

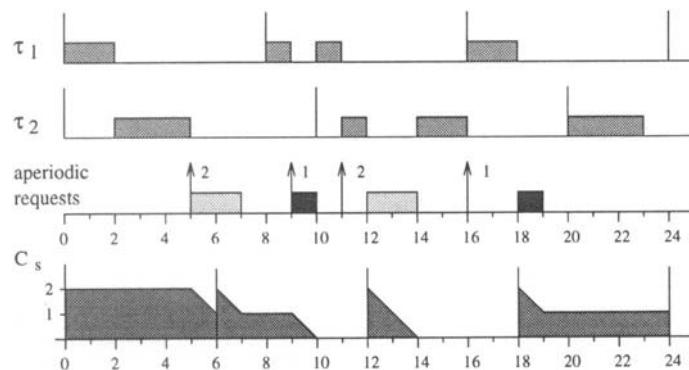


Figure 5.6 Example of high-priority Deferrable Server.

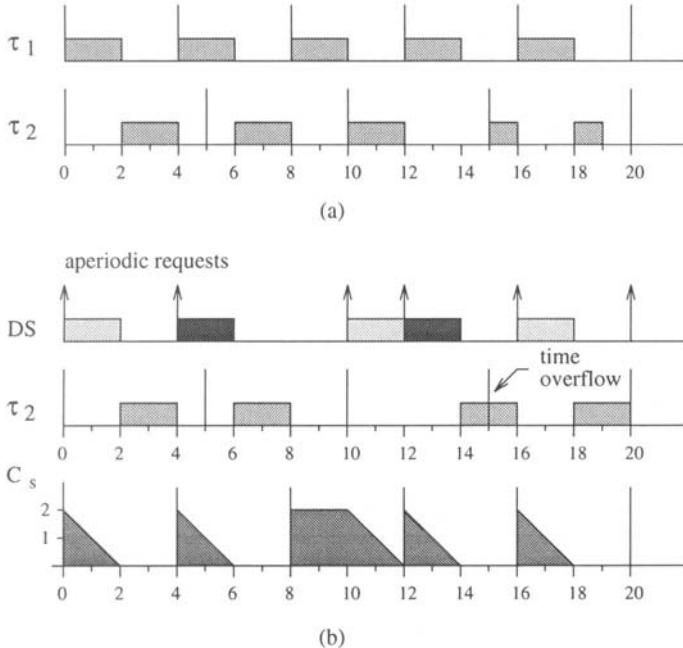


Figure 5.7 DS is not equivalent to a periodic task. In fact, the periodic set $\{\tau_1, \tau_2\}$ is schedulable by RM (a); however, if we replace τ_1 with DS, τ_2 misses its deadline (b).

ferrable Server violates this basic assumption. In fact, the schedule illustrated in Figure 5.6 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 5.7 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 Figure 5.7a, 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 5.7b 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.

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 Chapter 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, τ_1, \dots, τ_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 Figure 5.8, the full processor utilization is achieved by the following tasks' parameters:

$$\left\{ \begin{array}{l} C_s = T_1 - (T_s + C_s) = \frac{T_1 - T_s}{2} \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = \frac{3T_s + T_1 - 2T_n}{2}. \end{array} \right.$$

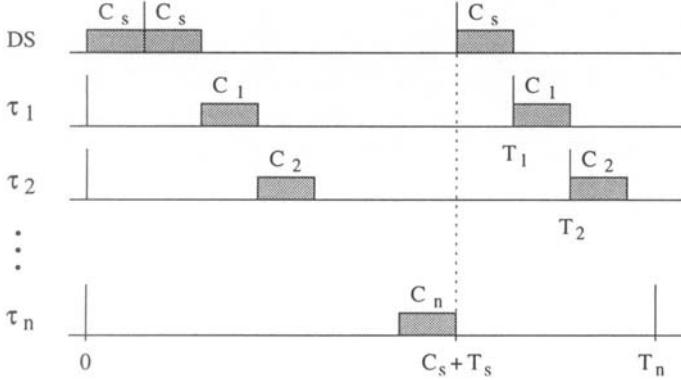


Figure 5.8 Worst-case task relations for a Deferrable Server.

Hence, the resulting utilization is

$$\begin{aligned}
 U &= \frac{C_s}{T_s} + \frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} = \\
 &= U_s + \frac{T_2 - T_1}{T_1} + \dots + \frac{T_n - T_{n-1}}{T_{n-1}} + \frac{3T_s + T_1 - 2T_n}{2T_n} = \\
 &= U_s + \frac{T_2}{T_1} + \dots + \frac{T_n}{T_{n-1}} + \left(\frac{3T_s}{2T_1} + \frac{1}{2}\right) \frac{T_1}{T_n} - n.
 \end{aligned}$$

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

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

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

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

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

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

that is,

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

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

$$U_{lub} = U_s + n \left[\left(\frac{U_s + 2}{2U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.2)$$

Taking the limit as $n \rightarrow \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n \rightarrow \infty} U_{lub} = U_s + \ln\left(\frac{U_s + 2}{2U_s + 1}\right). \quad (5.3)$$

Thus, given a set of n periodic tasks and a Deferrable Server with utilization factors U_p and U_s , respectively, the schedulability of the periodic task set is guaranteed under RM if

$$U_p + U_s \leq U_{lub}$$

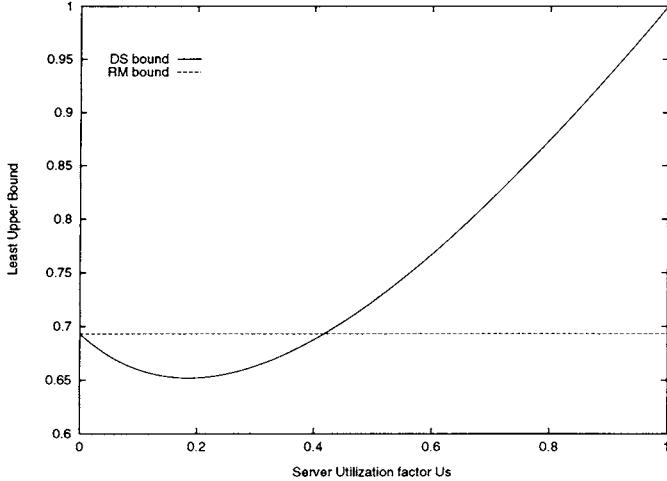


Figure 5.9 Schedulability bound for periodic tasks and DS as a function of the server utilization factor U_s .

that is, if

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right). \quad (5.4)$$

A plot of equation (5.3) as a function of U_s is shown in Figure 5.9. 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 equation (5.3) 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

$$U_s^* = \frac{\sqrt{33} - 5}{4} \approx 0.186,$$

so the minimum value of U_{lub} is $U_{lub}^* \approx 0.652$.

5.4.2 Aperiodic guarantee

The schedulability analysis for firm aperiodic tasks is derived by assuming a high-priority Deferrable Server. A guarantee test for a single request is first derived and then extended to a set of aperiodic tasks. Since DS preserves its execution time, let $c_s(t)$ be the value of its capacity at time t . Then, when a firm aperiodic request $J_a(C_a, D_a)$ enters the system at time $t = r_a$ (and no other requests are pending), three cases can occur:

1. $C_a \leq c_s(t)$. Hence, J_a completes at time $f_a = t + C_a$.
2. $C_a > c_s(t)$ and the capacity is completely discharged within the current period. In this case, a portion $\Delta_a = c_s(t)$ of J_a is executed in the current server period.
3. $C_a > c_s(t)$, but the period ends before the capacity is completely discharged. In this case, the portion of J_a executed in the current server period is $\Delta_a = G_a T_s - r_a$, where $G_a = \lceil t/T_s \rceil$.

In the last two cases, depicted in Figure 5.10, the portion of J_a executed in the current server period can be computed as

$$\Delta_a = \min[c_s(t), (G_a T_s - r_a)].$$

Using the same notation introduced during polling analysis, the finishing time f_a of request J_a can then be derived as follows (see Figure 5.11):

$$f_a = \begin{cases} r_a + C_a & \text{if } C_a \leq c_s(t) \\ (F_a + G_a)T_s + R_a & \text{otherwise,} \end{cases}$$

where

$$\begin{cases} F_a = \left\lfloor \frac{C_a - \Delta_a}{C_s} \right\rfloor \\ G_a = \left\lceil \frac{r_a}{T_s} \right\rceil \\ R_a = C_a - \Delta_a - F_a C_s. \end{cases}$$

Thus, the schedulability of a single aperiodic request is guaranteed if and only if $f_a \leq r_a + D_a$.

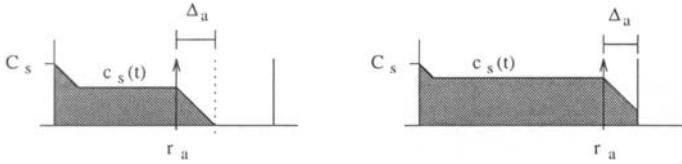


Figure 5.10 Execution of J_a in the first server period when $C_s > c_s(t)$.

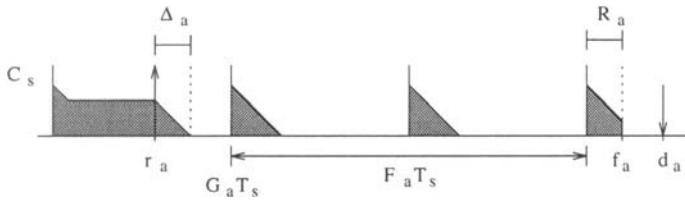


Figure 5.11 Calculation of the finishing time of J_a under DS.

To guarantee a set of firm aperiodic requests note that, at any time t , the total aperiodic computation that has to be served in any interval $[t, d_k]$ is equal to the sum of the remaining processing times $c_i(t)$ of the tasks with deadline $d_i \leq d_k$; that is,

$$C_{ape}(t, d_k) = \sum_{i=1}^k c_i(t).$$

And using the same approach adopted for the Polling Server, we define:

$$\begin{cases} G_k = \left\lceil \frac{t}{T_s} \right\rceil \\ \Delta_k = \min[c_s(t), (G_a T_s - r_a)] \\ F_k = \left\lfloor \frac{C_{ape}(t, d_k) - \Delta_k}{C_s} \right\rfloor \\ R_k = C_{ape}(t, d_k) - \Delta_k - F_k C_s. \end{cases}$$

Hence, the finishing time of the k th request is

$$f_k = \begin{cases} t + C_{ape} & \text{if } C_{ape} \leq c_s(t) \\ (F_k + G_k)T_s + R_k & \text{otherwise.} \end{cases}$$

Thus, a set of firm aperiodic requests is guaranteed at time t , if and only if

$$f_k \leq d_k \quad \forall k = 1, \dots, n.$$

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 the PE algorithm is to provide high responsiveness to aperiodic requests, all priority ties are broken in favor of aperiodic tasks.

Figure 5.12 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 Figure 5.12 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.

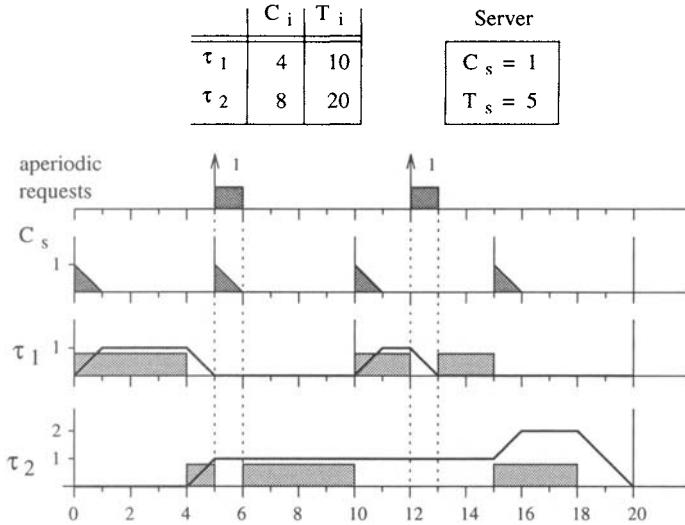


Figure 5.12 Example of aperiodic service under a PE server.

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, 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 Figure 5.13. Here, at time $t = 5$, the capacity of the server immediately

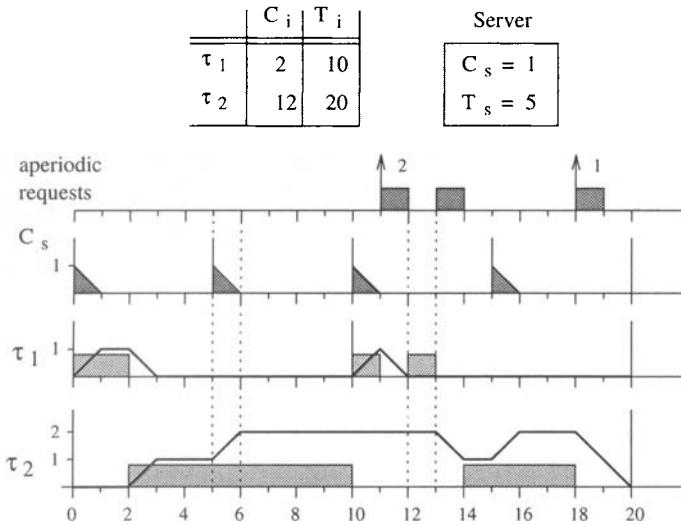


Figure 5.13 Example of aperiodic service under a PE server.

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

The schedulability bound for a set of periodic tasks running along with a Priority Exchange server is derived with the same technique used for the Deferrable Server. The least upper bound of the processor utilization factor in the presence of PE is calculated by assuming that PE is the highest-priority task in the system. To simplify the computation of the bound, the worst-case relations among the tasks is first determined, and then the lower bound is computed against the worst-case model [LSS87].

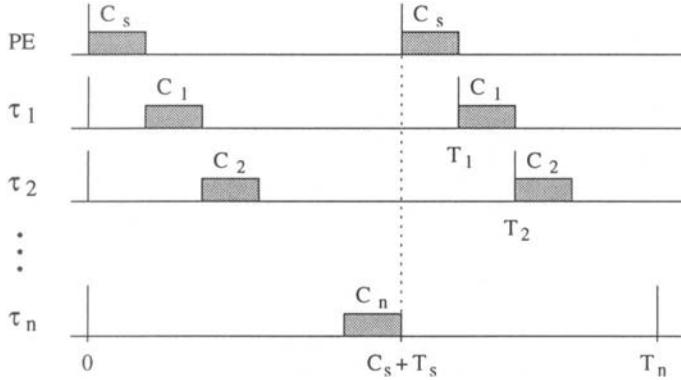


Figure 5.14 Worst-case tasks' relations under Priority Exchange.

Calculation of U_{lub} for PE+RM

Consider a set of n periodic tasks, τ_1, \dots, τ_n , ordered by increasing periods, and a PE server with a higher priority. The worst-case phasing and period relations for the periodic tasks are the same as the ones derived for the RM analysis; hence, $T_s < T_n < 2T_s$. The only difference with DS is that a PE server can execute at most two times within the period of the highest-priority periodic task. Hence, the worst-case situation for a set of periodic tasks that fully utilize the processor is the one illustrated in Figure 5.14, where tasks are characterized by the following parameters:

$$\left\{ \begin{array}{l} C_s = T_1 - T_s \\ C_1 = T_2 - T_1 \\ C_2 = T_3 - T_2 \\ \dots \\ C_{n-1} = T_n - T_{n-1} \\ C_n = T_s - C_s - \sum_{i=1}^{n-1} C_i = 2T_s - T_n. \end{array} \right.$$

The resulting utilization is then

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

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

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

the utilization factor may be written as

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

Since this is the same expression obtained for DS, the least upper bound is

$$U_{lub} = U_s + n(K^{1/n} - 1). \quad (5.5)$$

Equation (5.5) differs from equation (5.1) only for the value of K . And noting that

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

K can be rewritten as

$$K = \frac{2}{R_s} = \frac{2}{U_s + 1}.$$

Thus, finally

$$U_{lub} = U_s + n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.6)$$

Taking the limit as $n \rightarrow \infty$, we find the worst-case bound as a function of U_s to be given by

$$\lim_{n \rightarrow \infty} U_{lub} = U_s + \ln \left(\frac{2}{U_s + 1} \right). \quad (5.7)$$

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

$$U_p + U_s \leq U_s + \ln \left(\frac{2}{U_s + 1} \right)$$

that is, if

$$U_p \leq \ln \left(\frac{2}{U_s + 1} \right). \quad (5.8)$$

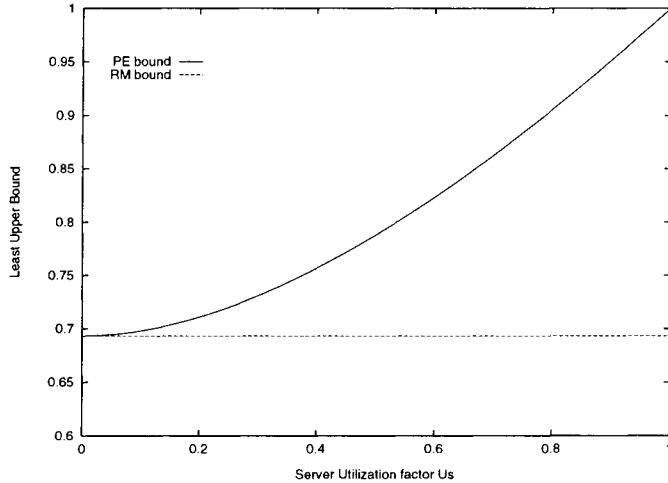


Figure 5.15 Schedulability bound for periodic tasks and PE as a function of the server utilization factor U_s .

A plot of equation (5.7) as a function of U_s is shown in Figure 5.15. For comparison, the RM bound is also reported in the plot. Notice that the schedulability test expressed in equation (5.8) is also valid for the Polling Server and, in general, for all servers that behave like a periodic task.

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

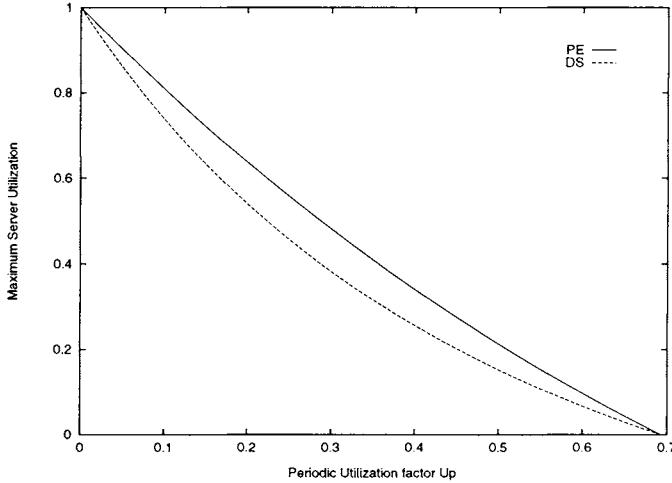


Figure 5.16 Maximum server utilization as a function of the periodic load.

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 size of a DS and a PE server as a function of U_p can easily be derived from the corresponding schedulability tests computed above. For example, from the DS schedulability test expressed in equation (5.4), the maximum utilization for DS turns out to be

$$U_{DS}^* = \frac{2 - e^{U_p}}{2e^{U_p} - 1}, \quad (5.9)$$

whereas, from equation (5.8), the maximum PE utilization is

$$U_{PE}^* = \frac{2 - e^{U_p}}{e^{U_p}}. \quad (5.10)$$

A plot of these two equations as a function of U_p is shown in Figure 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 Section 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 Figure 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, $RA_2 = 2$.

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 $RA_2 = 2$.

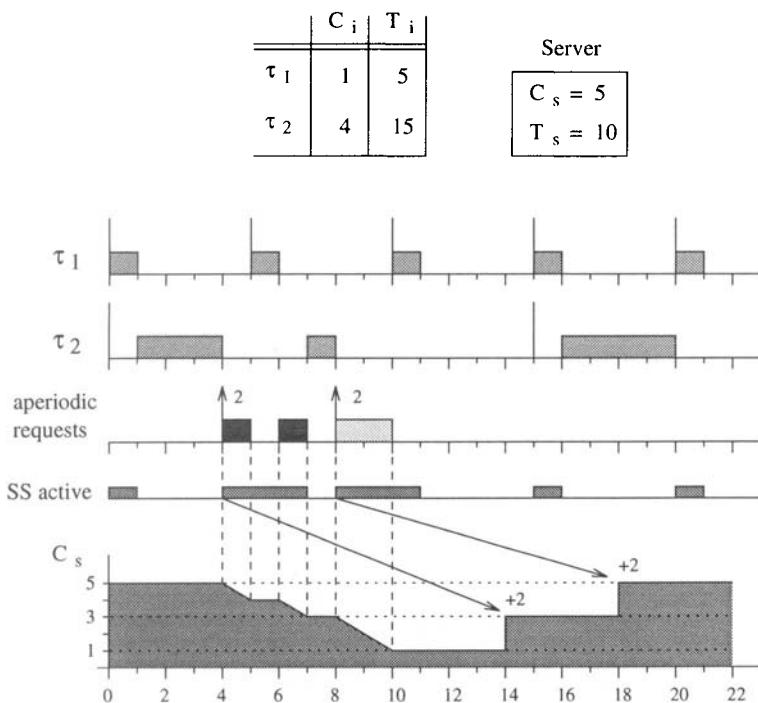


Figure 5.17 Example of a medium-priority Sporadic Server.

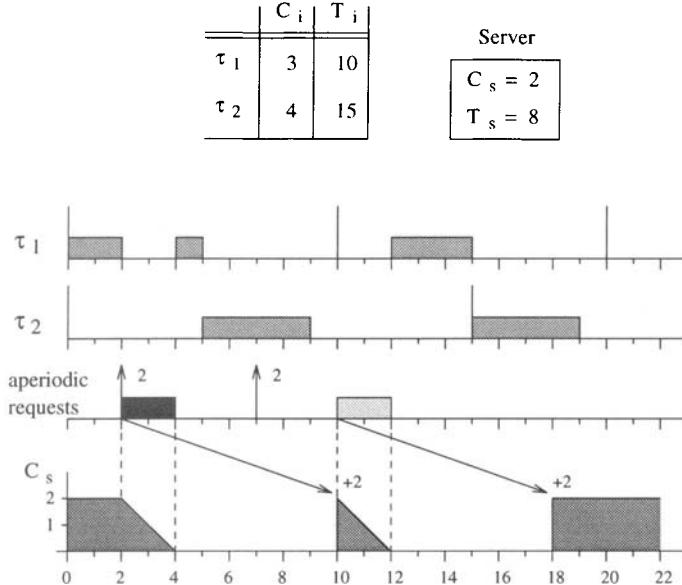


Figure 5.18 Example of a high-priority Sporadic Server.

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 Figure 5.19):

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 Chapter 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 PE. Hence, a set Γ of n

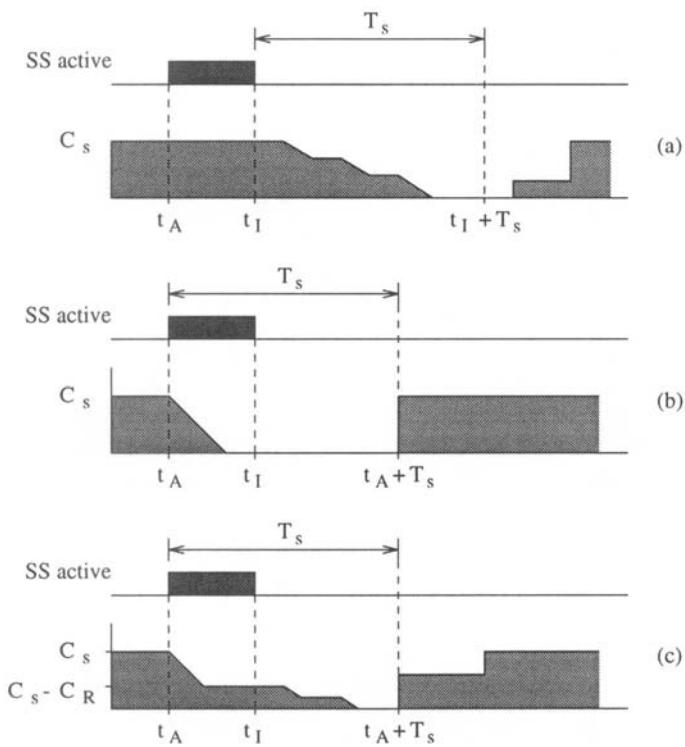


Figure 5.19 Possible SS behavior during active intervals: **a.** C_s is not consumed; **b.** C_s is totally consumed; **c.** C_s is partially consumed.

periodic tasks with utilization factor U_p scheduled along with a Sporadic Server with utilization U_s are schedulable under RM if

$$U_p \leq n \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right]. \quad (5.11)$$

For large n , Γ is schedulable if

$$U_p \leq \ln \left(\frac{2}{U_s + 1} \right) \quad (5.12)$$

For a given U_p , the maximum server size that guarantees the schedulability of the periodic tasks is

$$U_{SS}^* = 2 \left(\frac{U_p}{n} + 1 \right)^{-n} - 1, \quad (5.13)$$

and for large n it becomes

$$U_{SS}^* = \frac{2}{e^{U_p}} - 1. \quad (5.14)$$

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

$$\text{slack}_i(t) = d_i - t - c_i(t).$$

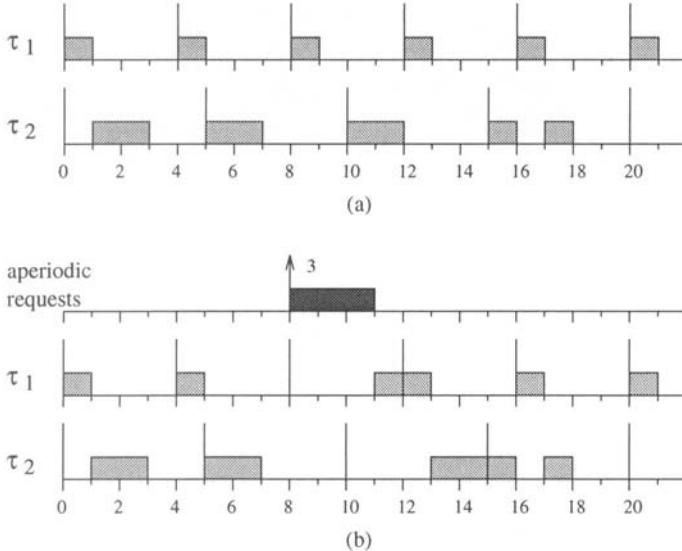


Figure 5.20 Example of Slack Stealer behavior: **a.** when no aperiodic requests are pending; **b.** when an aperiodic request of three units arrives at time $t = 8$.

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, Figure 5.20a shows the schedule produced by RM when no aperiodic tasks are processed, whereas Figure 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 Figure 5.20, no other server algorithms (DS, PE, or SS) can schedule the aperiodic requests at the highest priority and still guarantee the periodic tasks. For example, since $U_p = 1/4 + 2/5 = 0.65$, the

maximum utilization factor for a sporadic server to guarantee the schedulability of the periodic task set is (see equation (5.13)):

$$U_{SS}^* = 2 \left(\frac{U_p}{2} + 1 \right)^{-2} - 1 \simeq 0.14.$$

This means that, even with $C_s = 1$, the shortest server period that can be set with this utilization factor is $T_s = \lceil C_s/U_s \rceil = 8$, 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 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 non-decreasing 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 request enters the system. This method is more complex than the previous *static* approach, but it requires much less memory

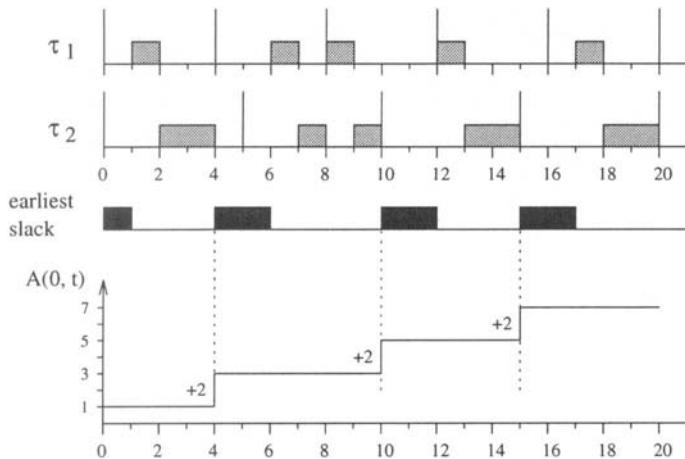


Figure 5.21 Slack function at time $s = 0$ for the periodic task set considered in the previous example.

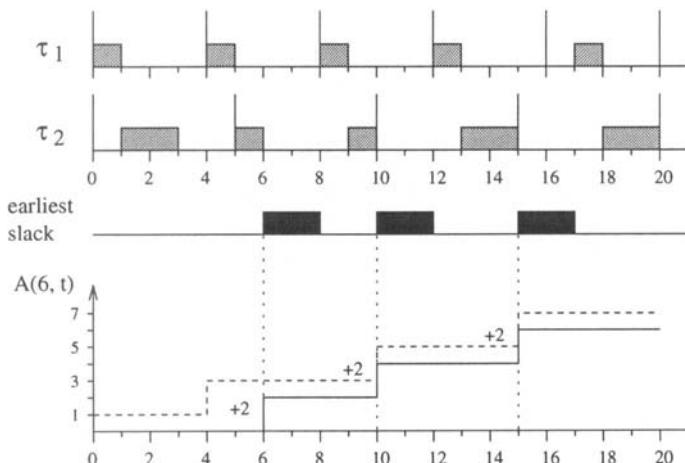


Figure 5.22 Slack function at time $s = 6$ for the periodic task set considered in the previous example.

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 Figure 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 Figure 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. \square

Notice that Theorem 5.2 applies both to clairvoyant and on-line 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.

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 on-line valid algorithm that minimizes the average response time of the soft aperiodic requests.*

Proof. From the example illustrated in Figure 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 on-line algorithm will not know when to schedule the requests. \square

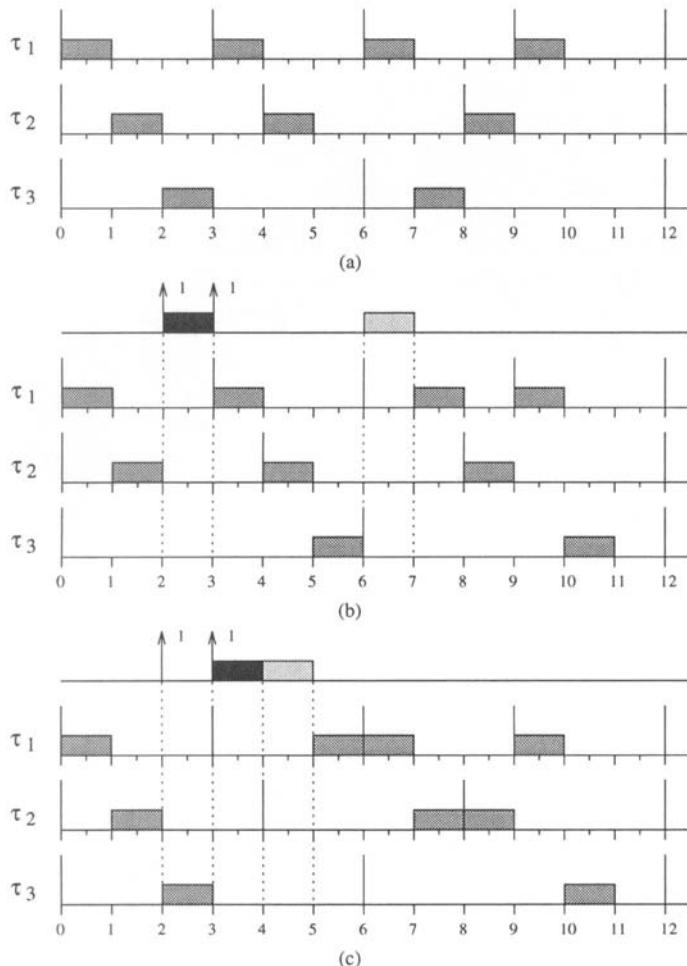


Figure 5.23 No algorithm can minimize the response time of every aperiodic request. If J_1 is minimized, J_2 is not (b). On the other hand, if J_2 is minimized, J_1 is not (c).

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 ten 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 Figure 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

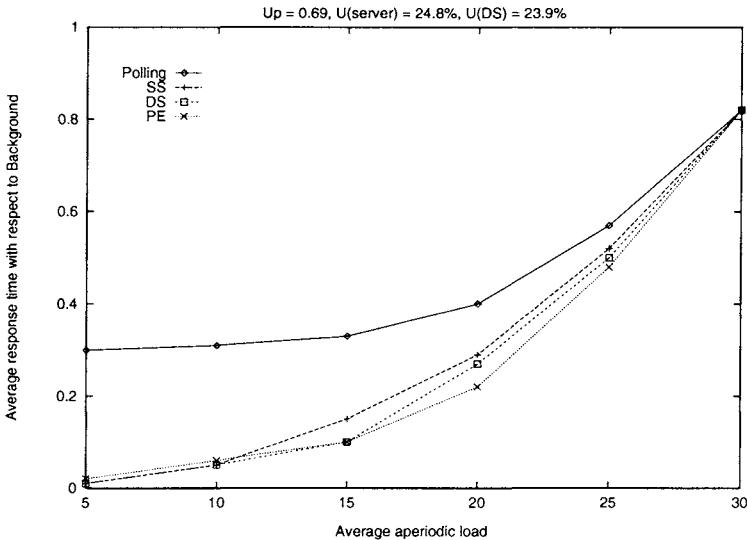


Figure 5.24 Performance results of PS, DS, PE, and SS.

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 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.

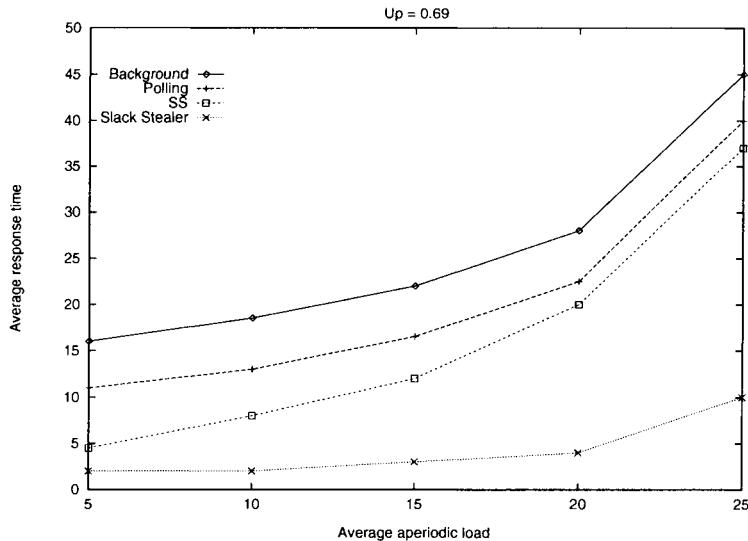


Figure 5.25 Performance of the Slack Stealer with respect to background, PS, and SS.



	performance	computational complexity	memory requirement	implementation complexity
Background Service	poor	good	good	good
Polling Server	poor	good	good	good
Deferrable Server	good	good	good	good
Priority Exchange	good	good	poor	good
Sporadic Server	good	good	poor	poor
Slack Stealer	good	poor	poor	poor

Figure 5.26 Evaluation summary of fixed-priority servers.

Exercises

- 5.1 Compute the maximum processor utilization that can be assigned to a Sporadic Server to guarantee the following periodic tasks under RM:

	τ_1	τ_2
C_i	1	2
T_i	5	8

- 5.2 Compute the maximum processor utilization that can be assigned to a Deferrable Server to guarantee the task set illustrated in Exercise 5.1.
- 5.3 Together with the periodic tasks illustrated in Exercise 5.1, schedule the following aperiodic tasks with a Polling Server having maximum utilization and intermediate priority.

	J_1	J_2	J_3
a_i	2	7	9
C_i	3	2	1

- 5.4 Solve the same scheduling problem described in Exercise 5.3, with a Sporadic Server having maximum utilization and intermediate priority.
- 5.5 Solve the same scheduling problem described in Exercise 5.3, with a Deferrable Server having maximum utilization and highest priority.
- 5.6 Solve the same scheduling problem described in Exercise 5.3, with a Priority Exchange Server having maximum utilization and highest priority.
- 5.7 Using a Sporadic Server with capacity $C_s = 2$ and period $T_s = 5$, schedule the following tasks:

periodic tasks		
	τ_1	τ_2
C_i	1	2
T_i	4	6

aperiodic tasks			
	J_1	J_2	J_3
a_i	2	5	10
C_i	2	1	2

- 5.8 Given the same tasks described in Exercise 5.7, compute the maximum capacity that can be assigned to a Sporadic Server with a period $T_s = 4$. Then, schedule the tasks using such a capacity.

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_i must be protected by

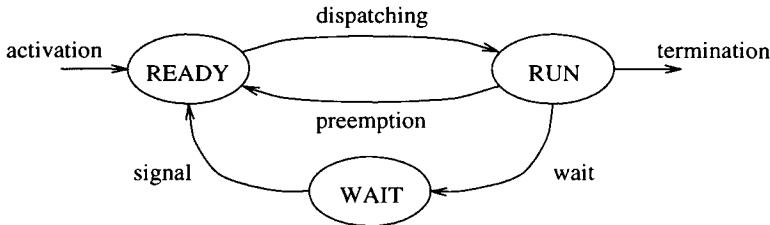


Figure 7.1 Waiting state caused by resource constraints.

a different semaphore S_i and each critical section operating on a resource R_i must begin with a $\text{wait}(S_i)$ primitive and end with a $\text{signal}(S_i)$ primitive.

All tasks blocked on the same 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 the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Figure 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 formulae found for periodic task sets.

7.2 THE PRIORITY INVERSION PHENOMENON

Consider two tasks J_1 and J_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 $\text{wait}(S_k)$ primitive and must end with a $\text{signal}(S_k)$ primitive (see Figure 7.2).

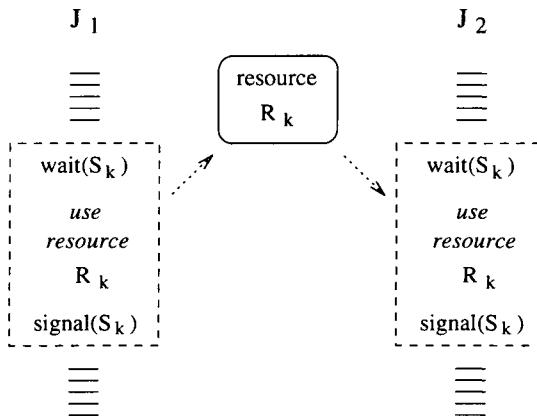


Figure 7.2 Structure of two tasks that share an exclusive resource.

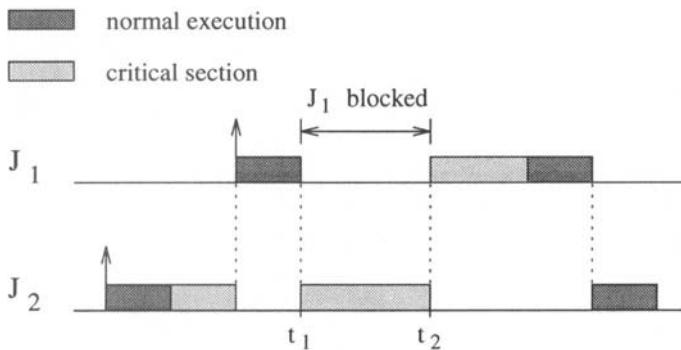


Figure 7.3 Example of blocking on an exclusive resource.

If preemption is allowed and J_1 has a higher priority than J_2 , then J_1 can be blocked in the situation depicted in Figure 7.3. Here, task J_2 is activated first, and, after a while, it enters the critical section and locks the semaphore. While J_2 is executing the critical section, task J_1 arrives and, since it has a higher priority, it preempts J_2 and starts executing. However, at time t_1 , when attempting to enter its critical section, J_1 is blocked on the semaphore, so J_2 resumes. J_1 has to wait until time t_2 , when J_2 releases the critical section by executing the $\text{signal}(S_k)$ primitive, which unlocks the semaphore.

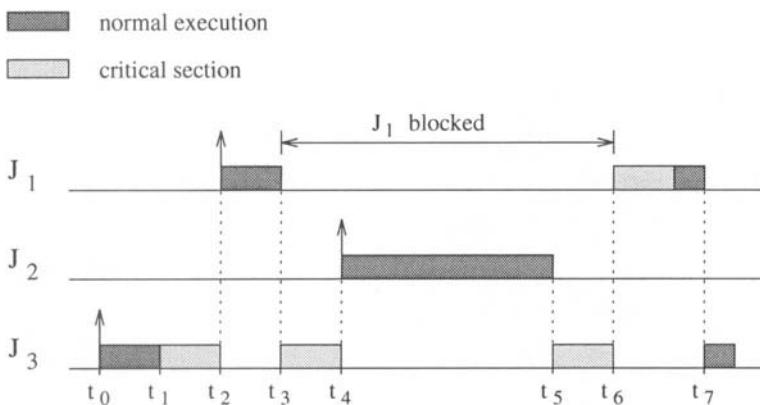


Figure 7.4 An example of priority inversion.

In this simple example, the maximum blocking time that J_1 may experience is equal to the time needed by J_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 Figure 7.4. Here, three tasks J_1 , J_2 , and J_3 have decreasing priorities, and J_1 and J_3 share an exclusive resource protected by a binary semaphore S .

If J_3 starts at time t_0 , it may happen that J_1 arrives at time t_2 and preempts J_3 inside its critical section. At time t_3 , J_1 attempts to use the resource, but it is blocked on the semaphore S ; thus, J_3 continues the execution inside its critical section. Now, if J_2 arrives at time t_4 , it preempts J_3 (because it has a higher priority) and increases the blocking time of J_1 by all its duration. As a consequence, the maximum blocking time that J_1 may experience does depend not only on the length of the critical section executed by J_3 but also on the worst-case execution time of J_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 J_1 waits for the execution of lower-priority tasks (J_2 and J_3). In general, the duration of priority inversion

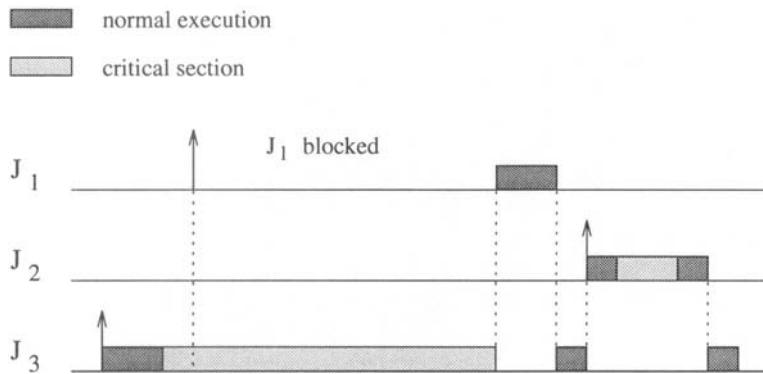


Figure 7.5 Scheduling with non-preemptive critical sections.

is unbounded, since any intermediate-priority task that can preempt J_3 will indirectly block J_1 .

Several approaches have been proposed to deal with the problem of scheduling tasks accessing shared resources. A simple solution that avoids the unbounded priority inversion problem is to disallow preemption during the execution of all critical sections. This method, however, is only appropriate for very short critical sections, because it creates unnecessary blocking. Consider, for example, the case depicted in Figure 7.5, where J_1 is the highest-priority task that does not use any resource, whereas J_2 and J_3 are low-priority tasks that share an exclusive resource. If the low-priority task J_3 enters a long critical section, J_1 may unnecessarily be blocked for a long period of time.

In other approaches, the priority inversion problem is solved through the use of appropriate protocols that control the accesses to any shared resource. The Priority Inheritance Protocol and the Priority Ceiling Protocol [SRL90] apply to fixed-priority systems,¹ whereas the Stack Resource Policy [Bak91] is suitable both for static and dynamic priority systems. These protocols are described in the following sections.

¹The 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].

7.3 PRIORITY INHERITANCE PROTOCOL

The *Priority Inheritance Protocol* (PIP), proposed by Sha, Rajkumar and Lehoczky [SRL90], offers a simple solution to the problem of unbounded priority inversion caused by resource constraints. The basic idea behind this protocol is to modify the priority of those tasks that cause blocking. In particular, when a task J_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 J_i and prolonging the blocking duration experienced by the higher-priority tasks. Before describing the protocol in detail, we first introduce the terminology and the basic assumptions made on the system.

7.3.1 Terminology and assumptions

Consider a set of n periodic tasks, $\tau_1, \tau_2, \dots, \tau_n$, which cooperate through m shared resources, R_1, R_2, \dots, R_m . Each task is characterized by a period T_i and a worst-case computation time C_i . The deadline of any periodic instance is assumed to be at the end of its period. Each resource R_k is guarded by a distinct semaphore S_k . Hence, all critical sections on resource R_k begin with a $\text{wait}(S_k)$ operation and end with a $\text{signal}(S_k)$ operation. The following notation is adopted throughout the discussion:

- J_i denotes a job; that is, a generic instance of task τ_i .
- Since the protocol can modify the priority of the tasks, for each task we distinguish a fixed *nominal* priority P_i (assigned, for example, 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 .
- $z_{i,j}$ denotes the j th critical section of job J_i .
- $d_{i,j}$ denotes the duration of $z_{i,j}$; that is, the time needed by J_i to execute $z_{i,j}$ without interruption.
- The semaphore guarding the critical section $z_{i,j}$ is denoted by $S_{i,j}$ and the resource associated with $z_{i,j}$ is denoted by $R_{i,j}$.
- We write $z_{i,j} \subset z_{i,k}$ to indicate that $z_{i,j}$ is entirely contained in $z_{i,k}$.

Moreover, the properties of the protocol are valid under the following assumptions:

- Jobs J_1, J_2, \dots, J_n are listed in descending order of nominal priority, with J_1 having the highest nominal priority.
- Jobs do not suspend themselves (for example, on I/O operations or on explicit synchronization primitives).
- The critical sections used by any task are *properly* nested; that is, given any pair $z_{i,j}$ and $z_{i,k}$, then either $z_{i,j} \subset z_{i,k}$, $z_{i,k} \subset z_{i,j}$, or $z_{i,j} \cap z_{i,k} = \emptyset$.
- Critical sections are guarded by binary semaphores. This means that only one job at a time can be within the critical section corresponding to a particular semaphore S_k .

7.3.2 Protocol definition

The Priority Inheritance Protocol can be defined as follows:

- Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a First Come First Served discipline.
- When job J_i tries to enter a critical section $z_{i,j}$ and resource $R_{i,j}$ is already held by a lower-priority job, J_i will be blocked. J_i is said to be blocked by the task that holds the resource. Otherwise, J_i enters the critical section $z_{i,j}$.
- When a job J_i is blocked on a semaphore, it transmits its active priority to the job, say J_k , that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$. J_k is said to *inherit* the priority of J_i . In general, a task inherits the highest priority of the jobs blocked by it.
- When J_k exits a critical section, it unlocks the semaphore, and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k , p_k is set to its nominal priority P_k , otherwise it is set to the highest priority of the jobs blocked by J_k .
- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2 , and J_2 blocks a job J_1 , then J_3 inherits the priority of J_1 via J_2 .

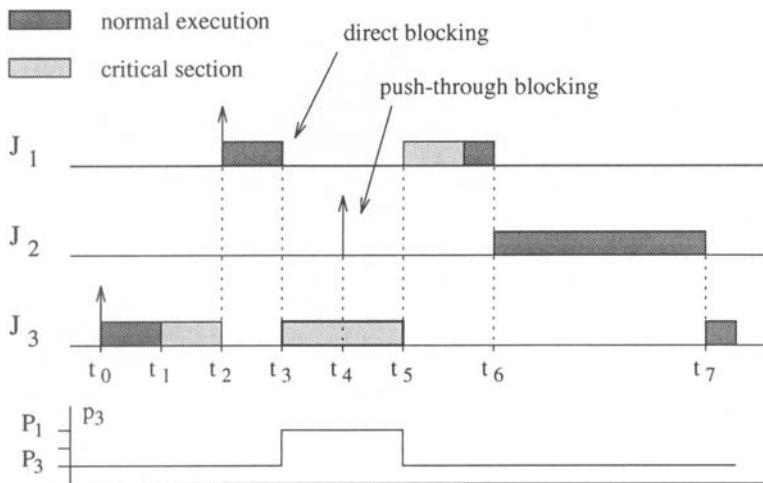


Figure 7.6 Example of Priority Inheritance Protocol.

Examples

We first consider the same situation presented in Figure 7.4 and show how the priority inversion phenomenon can be bounded by the Priority Inheritance Protocol. The modified schedule is illustrated in Figure 7.6. Until time t_3 there is no variation in the schedule, since no priority inheritance takes place. At time t_3 , J_1 is blocked by J_3 , thus J_3 inherits the priority of J_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 , J_2 cannot preempt J_3 and cannot create additional interference on J_1 . As J_3 exits its critical section, J_1 is awakened and J_3 resumes its original priority. At time t_5 , the processor is assigned to J_1 , which is the highest-priority task ready to execute, and task J_2 can only start at time t_6 , when J_1 has completed. The active priority of J_3 as a function of time is also shown in Figure 7.6 on the lowest timeline.

From this example, we can notice that a high-priority job can experience two kinds of blocking:

- **Direct blocking.** It occurs when a higher-priority job tries to acquire a resource already held by a lower-priority job. Direct blocking is necessary to ensure the consistency of the shared resources.

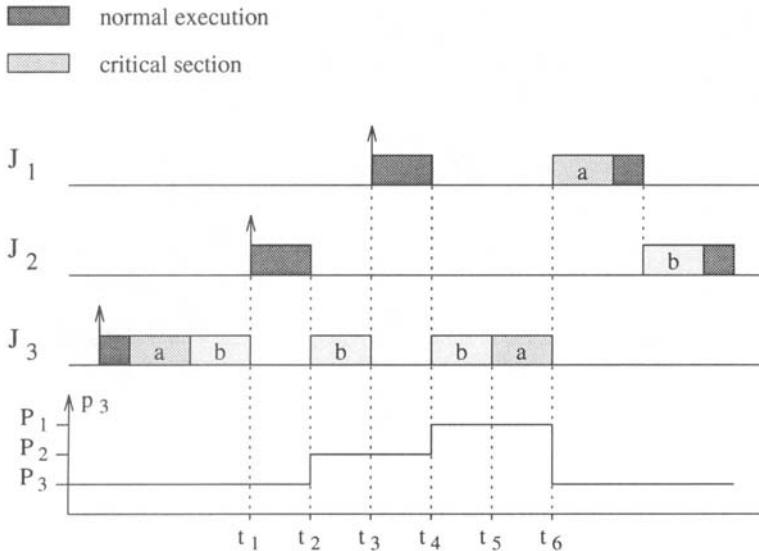


Figure 7.7 Priority inheritance with nested critical sections.

- **Push-through blocking.** It occurs when a medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks. Push-through blocking is necessary to avoid unbounded priority inversion.

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 Figure 7.7. Here, job J_1 uses a resource R_a guarded by a semaphore S_a , job J_2 uses a resource R_b guarded by a semaphore S_b , and job J_3 uses both resources in a nested fashion (S_a is locked first). At time t_1 , J_2 preempts J_3 within its nested critical section; hence, at time t_2 , when J_2 attempts to lock S_b , J_3 inherits its priority, P_2 . Similarly, at time t_3 , J_1 preempts J_3 within the same critical section and, at time t_4 , when J_1 attempts to lock S_a , J_3 inherits the priority P_1 . At time t_5 , when J_3 unlocks semaphore S_b , job J_2 is awakened but J_1 is still blocked; hence, J_3 continues its execution at the priority of J_1 . At time t_6 , J_3 unlocks S_a and, since no other jobs are blocked, J_3 resumes its original priority P_3 .

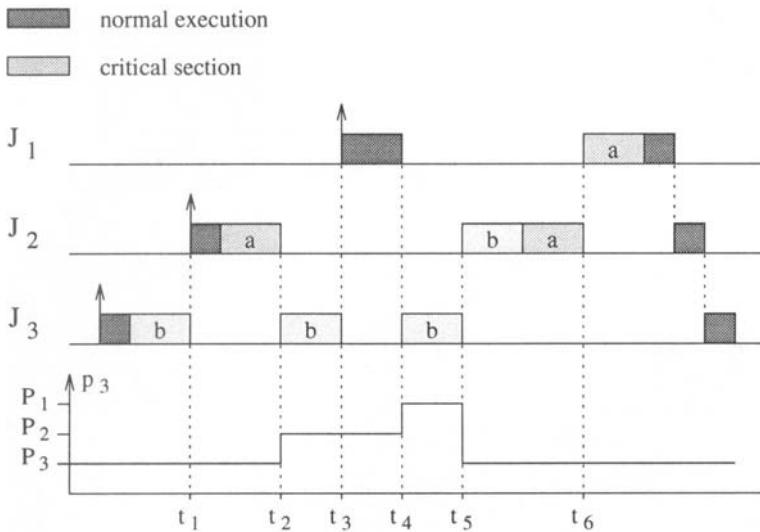


Figure 7.8 Example of transitive priority inheritance.

An example of transitive priority inheritance is shown in Figure 7.8. Here, job J_1 uses a resource R_a guarded by a semaphore S_a , job J_3 uses a resource R_b guarded by a semaphore S_b , and job J_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 , J_3 is preempted within its critical section by J_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, J_3 resumes and inherits the priority P_2 . At time t_3 , J_3 is preempted by J_1 , which at time t_4 tries to acquire R_a . Since S_a is locked by J_2 , J_2 inherits P_1 . However, J_2 is blocked by J_3 ; hence, for transitivity J_3 inherits the priority P_1 via J_2 . When J_3 exits its critical section, no other jobs are blocked by it, thus it resumes its nominal priority P_3 . Priority P_1 is now inherited by J_2 , which still blocks J_1 until time t_6 .

7.3.3 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 job J_i , only if S_k is accessed both by a job with priority lower than P_i and by a job that has or can inherit a priority equal to or higher than P_i .*

Proof. Suppose that semaphore S_k is accessed by a job J_l with priority lower than P_i . If S_k is not accessed by a job that has or can inherit a priority equal to or higher than P_i , then J_l cannot inherit a priority equal to or higher than P_i . Hence, J_l will be preempted by J_i and the lemma follows. \square

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 job J_H is blocked by a medium-priority job J_M , which in turn is blocked by a low-priority job J_L (see the example of Figure 7.8). Since J_H is blocked by J_M , J_M must hold a semaphore, say S_a . But J_M is also blocked by J_L on a different semaphore, say S_b . This means that J_M attempted to lock S_b inside the critical section guarded by S_a . The lemma follows. \square

Lemma 7.3 *If there are n lower-priority jobs that can block a job J_i , then J_i can be blocked for at most the duration of n critical sections (one for each of the n lower-priority jobs), regardless of the number of semaphores used by J_i .*

Proof. A job J_i can be blocked by a lower-priority job J_k only if J_k has been preempted within a critical section, say $z_{k,j}$, that can block J_i . Once J_k exits $z_{k,j}$, it can be preempted by J_i ; thus, J_i cannot be blocked by J_k again. The same situation may happen for each of the n lower-priority jobs; therefore, J_i can be blocked at most n times. \square

Lemma 7.4 *If there are m distinct semaphores that can block a job J_i , then J_i can be blocked for at most the duration of m critical sections, one for each of the m semaphores.*

Proof. Since semaphores are binary, only one of the lower-priority jobs, say J_k , can be within a blocking critical section corresponding to a particular semaphore S_j . Once S_j is unlocked, J_k can be preempted and can no longer block J_i . If all m semaphores that can block J_i are locked by m lower-priority jobs, then J_i can be blocked at most m times. \square

Theorem 7.1 (Sha-Rajkumar-Lehoczky) *Under the Priority Inheritance Protocol, a job J can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower-priority jobs that could block J and m is the number of distinct semaphores that can be used to block J .*

Proof. It immediately follows from Lemma 7.3 and Lemma 7.4. \square

7.3.4 Schedulability analysis

The most important property of the Priority Inheritance Protocol for real-time systems is that it bounds the maximum blocking time of each task. This allows to perform a feasibility analysis and extend the Rate-Monotonic schedulability test for sets of tasks with resource constraints. We recall that, in the absence of blocking, a set of independent periodic tasks is schedulable by the Rate-Monotonic algorithm if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \quad (7.1)$$

In order to perform a worst-case analysis, let B_i be the maximum blocking time, due to lower-priority jobs, that a job J_i may experience.

Theorem 7.2 *A set of n periodic tasks using the Priority Inheritance Protocol can be scheduled by the Rate-Monotonic algorithm if*

$$\forall i, \quad 1 \leq i \leq n, \quad \sum_{k=1}^i \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1). \quad (7.2)$$

Proof. Suppose that for each task τ_i equation (7.2) is satisfied. Then equation (7.1) is also satisfied with $n = i$ and C_i replaced by $C_i^* = (C_i + B_i)$. This means that, in the absence of blocking, any job of task τ_i will still meet its deadline even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i , if it executes for only C_i units of time, can be delayed by B_i and still meet its deadline. Hence, the theorem follows. \square

In other words, the schedulability test expressed in equation (7.2) can be interpreted as follows. In order to guarantee a task τ_i , we have to consider the effect of preemptions from all higher-priority tasks ($\sum_{k=1}^{i-1} C_k/T_k$), the execution of τ_i itself (C_i/T_i), and the effect of blocking due to all lower-priority tasks (B_i/T_i).

Suppose, for example, that we want to guarantee the following task set:

	C_i	T_i	B_i
J_1	1	2	1
J_2	1	4	1
J_3	2	8	0

Since the periods of these tasks are harmonic, the utilization bound for Rate Monotonic becomes 100%. Hence, we have to verify the following relations:

$$\begin{aligned}\frac{C_1}{T_1} + \frac{B_1}{T_1} &\leq 1 \\ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} &\leq 1 \\ \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} &\leq 1.\end{aligned}$$

Since all three equations hold, we can conclude that this task set is feasible and all tasks will meet their deadlines. Notice that, if the k th equation should not be satisfied, we would know that task τ_k would miss its deadline. In this case, we could correct the implementation of this task to achieve a feasible schedule.

A simpler but less tight schedulability test can be found by observing that

$$\frac{B_i}{T_i} \leq \max\left(\frac{B_1}{T_1}, \dots, \frac{B_n}{T_n}\right) \quad \text{and} \quad n(2^{1/n} - 1) \leq i(2^{1/i} - 1).$$

As a consequence, the feasibility of the schedule can be guaranteed if the following single equation holds:

$$\sum_{i=1}^n \frac{C_i}{T_i} + \max\left(\frac{B_1}{T_1}, \dots, \frac{B_n}{T_n}\right) \leq n(2^{1/n} - 1). \quad (7.3)$$

The schedulability test based on tasks' response times can also be extended to take resources into account. In this case, the blocking factor B_i must simply be added to the computation time of each task. Thus, the recurrent equation (4.12) for calculating the response time R_i becomes

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \quad (7.4)$$

Notice that, when introducing resource constraints, this test becomes only sufficient, since tasks characterized by a long maximum blocking time could actually never experience blocking.

7.3.5 Blocking time computation

The evaluation of the maximum blocking time for each task can be computed based on the result of Theorem 7.1. 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 J_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. The following notation is used to describe the algorithm:

- σ_i indicates the set of semaphores requested by J_i .
- $\beta_{i,j}$ indicates the set of all critical sections of the lower-priority job J_j that can block J_i .
- $\gamma_{i,k}$ indicates the set of all critical sections guarded by semaphore S_k that can block J_i .

- $Z_{i,k}$ denotes the longest critical section of task τ_i among those guarded by semaphore S_k .
- $D_{i,k}$ denotes the duration of $Z_{i,k}$.

Assuming that all durations $D_{i,k}$ are known (they can be estimated through code analysis), the algorithm for computing the blocking factor B_i of a job J_i can be logically divided into the following steps:

1. For each job J_j with priority lower than P_i , identify the set $\beta_{i,j}$ of all critical sections that can block J_i .
2. For each semaphore S_k , identify the set $\gamma_{i,k}$ of all critical sections guarded by S_k that can block J_i .
3. Sum the duration of the longest critical sections in each $\beta_{i,j}$, for any job J_j with priority lower than P_i ; let B_i^l be this sum.
4. Sum the duration of the longest critical sections in each $\gamma_{i,k}$, for any semaphore S_k ; let B_i^s be this sum.
5. Compute B_i as the minimum between B_i^l and B_i^s .

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* $C(S_k)$ to be the priority of the highest-priority task that may use it:

$$C(S_k) = \max(P_j : S_k \in \sigma_j).$$

Then, the following lemma holds.

Lemma 7.5 *In the absence of nested critical sections, a critical section $Z_{j,k}$ of J_j guarded by S_k can block J_i only if $P_j < P_i \leq C(S_k)$.*

Proof. If $P_i \leq P_j$, then job J_i cannot preempt J_j ; hence, it cannot be blocked by J_j directly. On the other hand, if $C(S_k) < P_i$, by definition of $C(S_k)$, any job that uses S_k cannot have or inherit a priority equal to or higher than P_i . Hence, from Lemma 7.1, $Z_{j,k}$ cannot cause push-through blocking on J_i . Finally, since there are no nested critical sections, Lemma 7.2 guarantees that $Z_{j,k}$ cannot cause transitive blocking. The lemma follows. \square

Using the result of Lemma 7.5, the maximum blocking time B_i for each task τ_i can easily be determined as follows:

$$B_i = \min(B_i^l, B_i^s), \quad (7.5)$$

where

$$\begin{aligned} B_i^l &= \sum_{j=i+1}^n \max_k [D_{j,k} : C(S_k) \geq P_i] \\ B_i^s &= \sum_{k=1}^m \max_{j>i} [D_{j,k} : C(S_k) \geq P_i]. \end{aligned}$$

This computation is performed by the algorithm shown in Figure 7.9. This algorithm assumes that the task set consists of n periodic tasks that use m distinct binary semaphores. Tasks are ordered with decreasing priority, such that $P_i > P_j$ for all $i < j$. Critical sections are nonnested. 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 $O(mn^2)$.

This algorithm provides an upper bound for the blocking factors B_i ; however, such a bound is not tight, since B_i^l may be computed by considering two or more critical sections guarded by the same semaphore. Obviously, if two critical sections of different jobs are guarded by the same semaphore, they cannot be both blocking (see Lemma 7.4). Similarly, B_i^s may be computed by considering two or more critical sections belonging to the same job. But this cannot happen (see Lemma 7.3). In order to exclude these cases, however, the complexity grows exponentially because the maximum blocking time has to be computed among all possible combinations of blocking 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.

Example

To illustrate the algorithm presented above, consider the following example, in which four tasks share three semaphores. For each job J_i , the duration of the longest critical section among those that use the same semaphore S_k is denoted by $D_{i,k}$ and it is stored in a table. $D_{i,k} = 0$ means that job J_i does not use semaphore S_k . Suppose to have the following table (semaphore ceilings are indicated in parentheses):

```

Blocking_Time( $D_{i,k}$ ) {
    for  $i = 1$  to  $n - 1$  {                                /* for each task  $J_i$  */
         $B_i^l := 0;$ 
        for  $j = i + 1$  to  $n$  {                      /* for each  $J_j : P_j < P_i$  */
             $D\_max := 0;$ 
            for  $k = 1$  to  $m$  {                  /* for all semaphores */
                if ( $C(S_k) \geq P_i$ ) and ( $D_{j,k} > D\_max$ ) {
                     $D\_max = D_{j,k};$ 
                }
            }
             $B_i^l := B_i^l + D\_max;$ 
        }
         $B_i^s := 0;$ 
        for  $k = 1$  to  $m$  {                  /* for all semaphores */
             $D\_max := 0;$ 
            for  $j = i + 1$  to  $n$  {          /* for each  $J_j : P_j < P_i$  */
                if ( $C(S_k) \geq P_i$ ) and ( $D_{j,k} > D\_max$ ) {
                     $D\_max = D_{j,k};$ 
                }
            }
             $B_i^s := B_i^s + D\_max;$ 
        }
         $B_i := \min(B_i^l, B_i^s);$ 
    }
     $B_n := 0;$ 
}

```

Figure 7.9 Algorithm for computing the blocking factors.

	$S_1(P_1)$	$S_2(P_1)$	$S_3(P_2)$
J_1	1	2	0
J_2	0	9	3
J_3	8	7	0
J_4	6	5	4

According to the algorithm shown in Figure 7.9, the blocking factors of the tasks are computed as follows:

$$\begin{aligned} B_1^l &= 9 + 8 + 6 = 23 \\ B_1^s &= 8 + 9 = 17 \quad ==> \quad B_1 = 17 \end{aligned}$$

$$\begin{aligned} B_2^l &= 8 + 6 = 14 \\ B_2^s &= 8 + 7 + 4 = 19 \quad ==> \quad B_2 = 14 \end{aligned}$$

$$\begin{aligned} B_3^l &= 6 \\ B_3^s &= 6 + 5 + 4 = 15 \quad ==> \quad B_3 = 6 \end{aligned}$$

$$B_4^l = B_4^s = 0 \quad ==> \quad B_4 = 0$$

Note that B_2^l is computed by adding the duration of two critical sections both guarded by semaphore S_1 .

7.3.6 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 (that is, 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 $s.\text{holder}$, the active priority of the executing task is updated and the ready task with the highest priority is assigned to the processor.

7.3.7 Unsolved problems

Although the Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration for a job can still be substantial because a chain of blocking can be formed. Another problem is that the protocol does not prevent deadlocks.

Chained blocking

Consider three jobs J_1 , J_2 and J_3 with decreasing priorities that share two semaphores S_a and S_b . Suppose that J_1 needs to sequentially access S_a and S_b , J_2 accesses S_b , and J_3 S_a . Also suppose that J_3 locks S_a and it is preempted by J_2 within its critical section. Similarly, J_2 locks S_b and it is preempted by J_1 within its critical section. The example is shown in Figure 7.10. In this situation, when attempting to use its resources, J_1 is blocked for the duration of two critical sections, once to wait J_3 to release S_a and then to wait J_2 to release S_b . This is called a *chained blocking*. In the worst case, if J_1 accesses n distinct semaphores that have been locked by n lower-priority jobs, J_1 will be blocked for the duration of n critical sections.

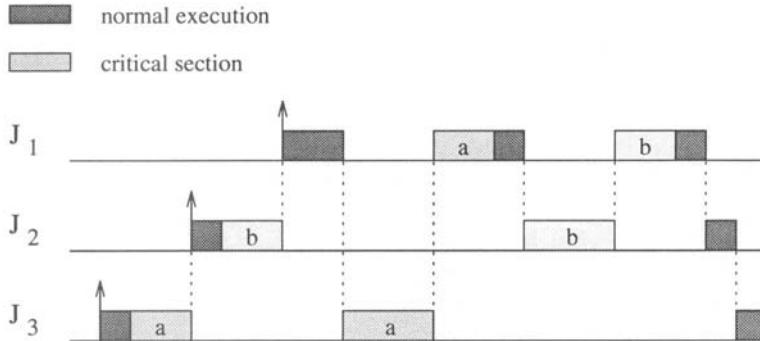


Figure 7.10 Example of chained blocking.

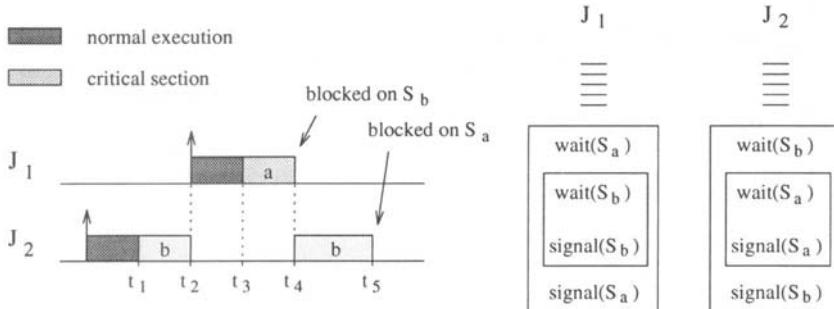


Figure 7.11 Example of deadlock.

Deadlock

Consider two jobs that use two semaphores in a nested fashion but in reverse order, as illustrated in Figure 7.11. Now suppose that, at time t_1 , J_2 locks semaphore S_b and enters its critical section. At time t_2 , J_1 preempts J_2 before it can lock S_a . At time t_3 , J_1 locks S_a , which is free, but then is blocked on S_b at time t_4 . At this time, J_2 resumes and continues the execution at the priority of J_1 . Priority inheritance does not prevent a deadlock, which occurs at time t_5 , when J_2 attempts to lock S_a . Notice, however, that the deadlock does not depend on the Priority Inheritance Protocol but 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.

7.4 PRIORITY CEILING PROTOCOL

The Priority Ceiling Protocol (PCP) has been 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 job to enter a critical section if there are locked semaphores that could block it. This means that, once a job enters its first critical section, it can never be blocked by lower-priority jobs until its completion.

In order to realize this idea, each semaphore is assigned a *priority ceiling* equal to the priority of the highest-priority job that can lock it. Then, a job J is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by jobs other than J .

7.4.1 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 priority of the highest-priority job that can lock it. Note that $C(S_k)$ is a static value that can be computed off-line.
- Let J_i be the job with the highest priority among all jobs ready to run; thus, J_i is assigned the processor.
- Let S^* be the semaphore with the highest-priority ceiling among all the semaphores currently locked by jobs other than J_i and let $C(S^*)$ be its ceiling.
- To enter a critical section guarded by a semaphore S_k , J_i must have a priority higher than $C(S^*)$. If $P_i \leq C(S^*)$, the lock on S_k is denied and J_i is said to be blocked on semaphore S^* by the job that holds the lock on S^* .
- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k , that holds that semaphore. Hence, J_k resumes and executes

the rest of its critical section with the priority of J_i . J_k is said to *inherit* the priority of J_i . In general, a task inherits the highest priority of the jobs blocked by it.

- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k , p_k is set to the nominal priority P_k ; otherwise, it is set to the highest priority of the jobs blocked by J_k .
- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2 , and J_2 blocks a job J_1 , then J_3 inherits the priority of J_1 via J_2 .

Example

In order to illustrate the Priority Ceiling Protocol, consider three jobs J_0 , J_1 , and J_2 having decreasing priorities. The highest-priority job J_0 sequentially accesses two critical sections guarded by semaphores S_0 and S_1 ; job J_1 accesses only a critical section guarded by semaphore S_2 ; whereas job J_2 uses semaphore S_2 and then makes a nested access to S_1 . From tasks' resource requirements, all semaphores are assigned the following priority ceilings:

$$\begin{cases} C(S_0) = P_0 \\ C(S_1) = P_0 \\ C(S_2) = P_1. \end{cases}$$

Now suppose that events evolve as illustrated in Figure 7.12.

- At time t_0 , J_2 is activated and, since it is the only job ready to run, it starts executing and later locks semaphore S_2 .
- At time t_1 , J_1 becomes ready and preempts J_2 .
- At time t_2 , J_1 attempts to lock S_2 , but it is blocked by the protocol because P_1 is not greater than $C(S_2)$. Then, J_2 inherits the priority of J_1 and resumes its execution.
- At time t_3 , J_2 successfully enters its nested critical section by locking S_1 . Note that J_2 is allowed to lock S_1 because no semaphores are locked by other jobs.

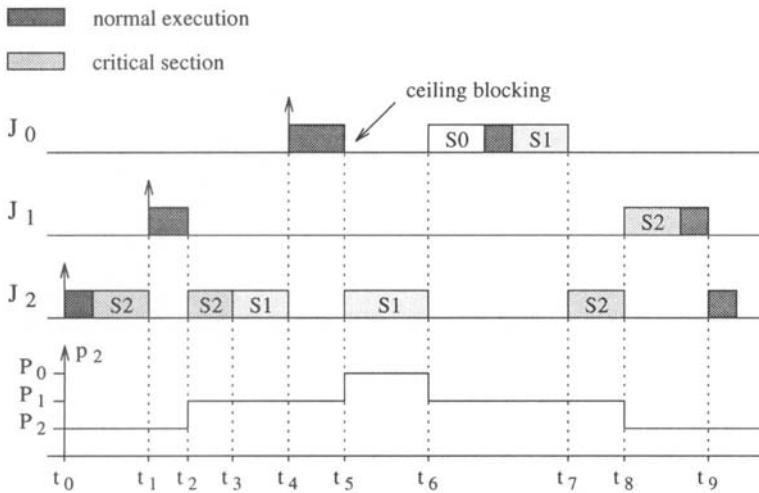


Figure 7.12 Example of Priority Ceiling Protocol.

- At time t_4 , while J_2 is executing at a priority $p_2 = P_1$, J_0 becomes ready and preempts J_2 because $P_0 > p_2$.
- At time t_5 , J_0 attempts to lock S_0 , which is not locked by any job. However, J_0 is blocked by the protocol because its priority is not higher than $C(S_1)$, which is the highest ceiling among all semaphores currently locked by the other jobs. Since S_1 is locked by J_2 , J_2 inherits the priority of J_0 and resumes its execution.
- At time t_6 , J_2 exits its nested critical section, unlocks S_1 , and, since J_0 is awakened, J_2 returns to priority $p_2 = P_1$. At this point, $P_0 > C(S_2)$; hence, J_0 preempts J_2 and executes until completion.
- At time t_7 , J_0 is completed, and J_2 resumes its execution at a priority $p_2 = P_1$.
- At time t_8 , J_2 exits its outer critical section, unlocks S_2 , and, since J_1 is awakened, J_2 returns to its nominal priority P_2 . At this point, J_1 preempts J_2 and executes until completion.
- At time t_9 , J_1 is completed; thus, J_2 resumes its execution.

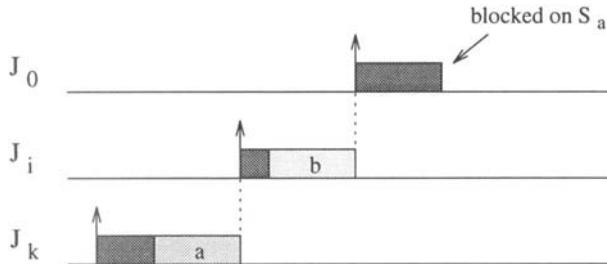


Figure 7.13 An absurd situation that cannot occur under the Priority Ceiling Protocol.

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 job J_0 at time t_5 .

7.4.2 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 job J_k is preempted within a critical section Z_a by a job J_i that enters a critical section Z_b , then, under the Priority Ceiling Protocol, J_k cannot inherit a priority higher than or equal to that of job J_i until J_i completes.*

Proof. If J_k inherits a priority higher than or equal to that of job J_i before J_i completes, there must exist a job J_0 blocked by J_k , such that $P_0 \geq P_i$. This situation is shown in Figure 7.13. However, this leads to the contradiction that J_0 cannot be blocked by J_k . In fact, since J_i enters its critical section, its priority must be higher than the maximum ceiling C^* of the semaphores currently locked by all lower-priority jobs. Hence, $P_0 \geq P_i > C^*$. But since $P_0 > C^*$, J_0 cannot be blocked by J_k , and the lemma follows. \square



Figure 7.14 Deadlock among n jobs.

Lemma 7.7 *The Priority Ceiling Protocol prevents transitive blocking.*

Proof. Suppose that a transitive block occurs; that is, there exist three jobs J_1 , J_2 , and J_3 , with decreasing priorities, such that J_3 blocks J_2 and J_2 blocks J_1 . By the transitivity of the protocol, J_3 will inherit the priority of J_1 . However, this contradicts Lemma 7.6, which shows that J_3 cannot inherit a priority higher than or equal to P_2 . Thus, the lemma follows. \square

Theorem 7.3 *The Priority Ceiling Protocol prevents deadlocks.*

Proof. Assuming that a job cannot deadlock by itself, a deadlock can only be formed by a cycle of jobs waiting for each other, as shown in Figure 7.14. In this situation, however, by the transitivity of the protocol, job J_n would inherit the priority of J_1 , which is assumed to be higher than P_n . This contradicts Lemma 7.6, and hence the theorem follows. \square

Theorem 7.4 (Sha-Rajkumar-Lehoczky) *Under the Priority Ceiling Protocol, a job J_i can be blocked for at most the duration of one critical section.*

Proof. Suppose that J_i is blocked by two lower-priority jobs J_1 and J_2 , where $P_2 < P_1 < P_i$. Let J_2 enter its blocking critical section first, and let C_2^* be the highest-priority ceiling among all the semaphores locked by J_2 . In this situation, if job J_1 enters its critical section we must have that $P_1 > C_2^*$. Moreover, since we assumed that J_i can be blocked by J_2 , we must have that $P_i \leq C_2^*$. This means that $P_1 > C_2^* \geq P_i$. This contradicts the assumption that $P_i > P_2$. Thus, the theorem follows. \square

7.4.3 Schedulability analysis

The feasibility test for a set of periodic tasks using the Priority Ceiling Protocol can be performed by the same formulae shown for the Priority Inheritance Protocol. The only difference is in the values of each blocking factor B_i , which, for the Priority Ceiling Protocol, corresponds to the duration of the longest critical section among those that can block τ_i .

7.4.4 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 job J_i can be blocked for at most the duration of the longest critical section among those that can block J_i . The set of critical sections that can block a job J_i is identified by the following lemma.

Lemma 7.8 *Under the Priority Ceiling Protocol, a critical section $Z_{j,k}$ (belonging to job J_j and guarded by semaphore S_k) can block a job J_i only if $P_j < P_i$ and $C(S_k) \geq P_i$.*

Proof. Clearly, if $P_j \geq P_i$, J_i cannot preempt J_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 J_i is blocked on $Z_{j,k}$. We show that this assumption leads to a contradiction. In fact, if J_i is blocked by J_j , its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than J_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 jobs other than J_i , we have that $C^* \geq C(S_k)$, which leads to a contradiction and proves the lemma. \square

Using the result of Lemma 7.8, the maximum blocking time B_i of job J_i can be computed as the duration of the longest critical section among those belonging to tasks with priority lower than P_i and guarded by a semaphore with ceiling higher than or equal to P_i . If $D_{j,k}$ denotes the duration of the longest critical section of task τ_j among those guarded by semaphore S_k , we can write

$$B_i = \max_{j,k} \{D_{j,k} \mid P_j < P_i, C(S_k) \geq P_i\}. \quad (7.6)$$

Consider the same example illustrated for the Priority Inheritance Protocol. For each job J_i , the duration of the longest critical section among those guarded by semaphore S_k is denoted by $D_{i,k}$ and it is stored in a table. $D_{i,k} = 0$ means that job J_i does not use semaphore S_k . Semaphore ceilings are indicated in parentheses:

	$S_1(P_1)$	$S_2(P_1)$	$S_3(P_2)$
J_1	1	2	0
J_2	0	9	3
J_3	8	7	0
J_4	6	5	4

According to equation (7.6), tasks' blocking factors are computed as follows:

$$\begin{cases} B_1 = \max(8, 6, 9, 7, 5) = 9 \\ B_2 = \max(8, 6, 7, 5, 4) = 8 \\ B_3 = \max(6, 5, 4) = 6 \\ B_4 = 0. \end{cases}$$

7.4.5 Implementation considerations

The major implication of the Priority Ceiling Protocol in the kernel data structures is that semaphores queues are no longer needed, since the tasks blocked by the protocol can be kept in the ready queue. In particular, whenever a job J_i is blocked by the protocol on a semaphore S_k , the job J_h that holds S_k inherits the priority of J_i and it is assigned to the processor, whereas J_i returns to the ready queue. As soon as J_h unlocks S_k , p_h is updated and, if p_h becomes less than the priority of the first ready job, 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, ordered by decreasing priority ceilings. This list is useful for computing the maximum priority ceiling that a job 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 jobs other than the one in execution (J_{exe}).
- If $p_{exe} \leq C^*$, transfer P_{exe} to the job that holds S^* , insert J_{exe} in the ready queue, and execute the ready job (other than J_{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 J_{exe} in $s.holder$.

pc_signal(s)

- Extract s from the list of currently locked semaphores.
- If no other jobs are blocked by J_{exe} , set $p_{exe} = P_{exe}$, else set p_{exe} to the highest priority of the jobs blocked by J_{exe} .
- Let p^* be the highest priority among the ready jobs. If $p_{exe} < p^*$, insert J_{exe} in the ready queue and execute the ready job (other than J_{exe}) with the highest priority.

7.5 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 multiunit 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.

7.5.1 Definitions

Before presenting the formal description of the SRP we introduce the following definitions.

Priority

Each task τ_i is assigned a priority p_i that indicates the importance (that is, 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).

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 job J_a can preempt another job J_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 J_a arrives after J_b and J_a has higher priority than J_b , then J_a must have a higher preemption level than J_b .

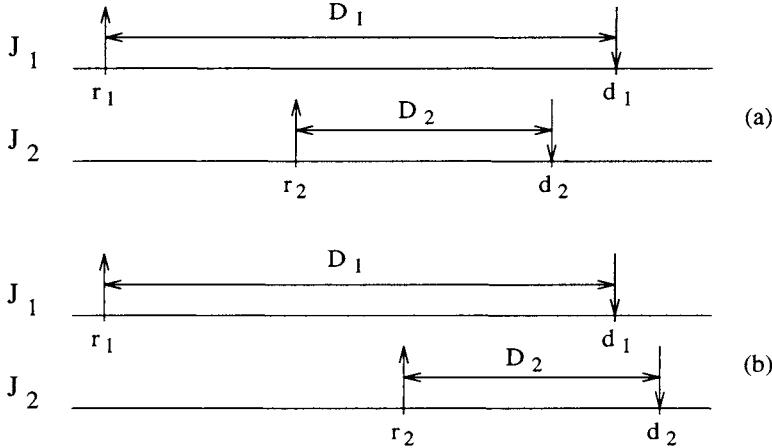


Figure 7.15 Although $\pi_2 > \pi_1$, under EDF p_2 can be higher than p_1 (a) or lower than p_1 (b).

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 Figure 7.15. Here we have two jobs J_1 and J_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 $\pi_1 < \pi_2$, J_1 can never preempt J_2 ; however, J_1 may have a priority higher than that of J_2 . In fact, under EDF, the priority of a job is dynamically assigned based on its absolute deadline. For example, in the case illustrated in Figure 7.15a, the absolute deadlines are such that $d_2 < d_1$; hence, J_2 will have higher priority than J_1 . On the other hand, as shown in Figure 7.15b, if J_2 arrives a time $r_1 + 6$, absolute deadlines are such that $d_2 > d_1$; hence, J_1 will have higher priority than J_2 .

Notice that, in the case of Figure 7.15b, although J_1 has priority higher than J_2 , J_2 cannot be preempted. This happens because, when $d_1 < d_2$ and $D_1 > D_2$, J_1 always starts before J_2 ; thus, it does not need to preempt J_2 .

	D_i	π_i	μ_{R1}	μ_{R2}	μ_{R3}
J_1	5	3	1	0	1
J_2	10	2	2	1	3
J_3	20	1	3	1	1

Figure 7.16 Task parameters and resource requirements.

Resource ceiling

Each resource R is required to have a *current ceiling* C_R , which is a dynamic value computed as a function of the units of R that are currently available. If n_R denotes the number of units of R that are currently available and $\mu_R(J)$ denotes the maximum requirement of job J for R , the current ceiling of R is defined to be

$$C_R(n_R) = \max[\{0\} \cup \{\pi(J) : n_R < \mu_R(J)\}].$$

In other words, if all units of R are available, then $C_R = 0$. However, if the units of R that are currently available cannot satisfy the requirement of one or more jobs, then C_R is equal to the highest preemption level of those jobs that could be blocked on R .

To better clarify this concept, consider the following example, where three tasks (J_1 , J_2 , J_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 Figure 7.16.

Based on these requirements, the current ceilings of the resources as a function of the number n_R of available units are reported in Figure 7.17 (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 Figure 7.16, we see that the only job that could be blocked in this condition is J_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 jobs that could be blocked are J_2 and J_3 ; hence, $C_{R1}(1) = \max(\pi_2, \pi_3) = 2$. Finally, if none of the units of R_1 is available, all three jobs could be blocked on R_1 ; hence, $C_{R1}(0) = \max(\pi_1, \pi_2, \pi_3) = 3$.

	$C_R(3)$	$C_R(2)$	$C_R(1)$	$C_R(0)$
R_1	0	1	2	3
R_2	-	-	0	2
R_3	0	2	2	3

Figure 7.17 Resource ceilings as a function of the number of available units. Dashes identify impossible cases.

Notice that, in the specific case of resources having a single unit (binary resources), the definition of current ceiling can be simplified as follows:

$$C_R = \max(\{0\} \cup \{\pi(J) : R \text{ could block } J\}).$$

This means that, if R is free, its ceiling is zero, whereas if R is busy, its ceiling is equal to the highest preemption level of the jobs that require R .

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,

$$\Pi_s = \max(C_{R_i} : i = 1, \dots, m).$$

Notice that Π_s is a dynamic parameter that can change every time a resource is accessed or released by a job.

7.5.2 Protocol definition

The key idea of the SRP is that, when a job 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 job is not allowed to start until the resources currently available are sufficient to meet the maximum requirement of every job that could preempt it. Using the definitions introduced in the previous paragraph, this is achieved by the following preemption test:

A job is not permitted to preempt until its priority is the highest among those of all the jobs 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 $\pi(J)$ of the ready job with the highest priority (the one at the head of the queue) with the system ceiling. If $\pi(J) > \Pi_s$, job J is executed, otherwise it is kept in the ready queue until Π_s becomes less than $\pi(J)$. The condition $\pi(J) > \Pi_s$ has to be tested every time Π_s may decrease; that is, every time a resource is released.

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 job J ensures that the resources that are currently available are sufficient to satisfy the maximum requirement of job J and the maximum requirement of every job that could preempt J . This means that, once J starts executing, it will never be blocked for resource contention.
- Although the preemption test for a job J is performed before J starts to execute, 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, decreases the number of context switches, reduces the run-time overhead, and simplifies the implementation of the protocol.
- The preemption test has the effect of imposing priority inheritance; that is, an executing job that holds a resource modifies the system ceiling and resists preemption as though it inherits the priority of any jobs that might need that resource. Note that this effect is accomplished without modifying the priority of the job.

Example

In order to illustrate how the SRP works, consider the task set already described in Figure 7.16. The structure of the tasks is shown in Figure 7.18, where $wait(R_i, n)$ denotes the request of n units of resource R_i , and $signal(R_i)$ denotes their release. The current ceilings of the resources have already been

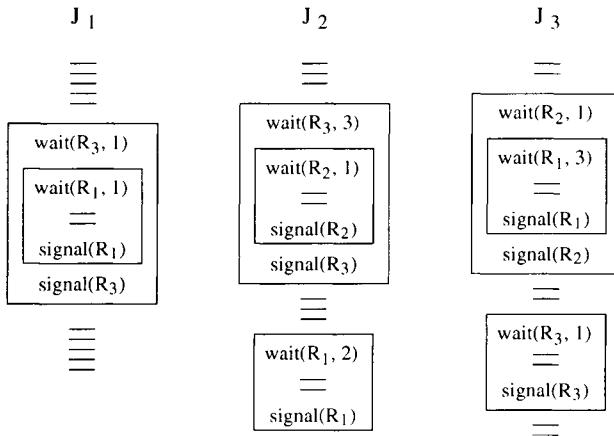


Figure 7.18 Structure of the tasks in the SRP example.

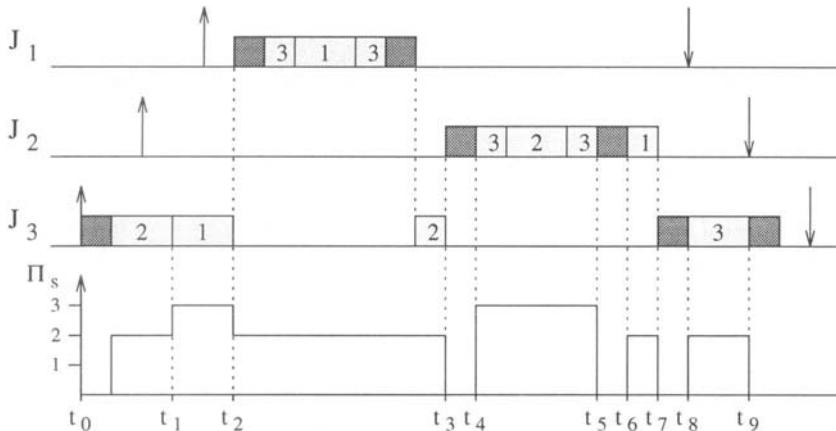


Figure 7.19 Example of a schedule under EDF and SRP. Numbers on tasks execution denote the resource indexes.

shown in Figure 7.17, and a possible EDF schedule for this task set is depicted in Figure 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 , J_3 starts executing and the system ceiling is zero because all resources are completely available. When J_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 Figure 7.17); that is, $\Pi_s = \pi_2 = 2$. As a consequence, J_2 is blocked by the preemption test and J_3 continues to execute. Note that when J_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 J_1 to be blocked by the preemption test.

As J_3 releases R_1 (at time t_2), the system ceiling becomes $\Pi_s = 2$; thus, J_1 preempts J_3 and starts executing. Note that, once J_1 is started, it is never blocked during its execution because the condition $\pi_1 > \Pi_s$ guarantees that all the resources needed by J_1 are available. As J_1 terminates, J_3 resumes the execution and releases resource R_2 . As R_2 is released, the system ceiling returns to zero and J_2 can preempt J_3 . Again, once J_2 is started, all the resources it needs are available; thus, J_2 is never blocked.

7.5.3 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 job J 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 J and*
2. *Meet the maximum requirement of every job that can preempt J .*

Proof. Assume $\pi(J) > C_R$, but suppose that the maximum request of J for R cannot be satisfied. Then, by definition of current ceiling of a resource, we have $C_R \geq \pi(J)$, which is a contradiction.

Assume $\pi(J) > C_R$, but suppose that there exists a job J_H that can preempt J such that the maximum request of J_H for R cannot be satisfied. Since J_H can preempt J , it must be $\pi(J_H) > \pi(J)$. Moreover, since the maximum request of J_H for R cannot be satisfied, by definition of current ceiling of a resource, we have $C_R \geq \pi(J_H)$. Hence, we derive that $\pi(J) < C_R$, which contradicts the assumption. \square

Theorem 7.5 (Baker) *If no job J is permitted to start until $\pi(J) > \Pi_s$, then no job can be blocked after it starts.*

Proof. Let N be the number of tasks that can preempt a job J and assume that no job is permitted to start until its preemption level is greater than Π_s . The thesis will be proved by induction on N .

If $N = 0$, there are no jobs that can preempt J . If J is started when $\pi(J) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by J are available when J preempts; hence, J will execute to completion without blocking.

If $N > 0$, suppose that J is preempted by J_H . If J_H is started when $\pi(J_H) > \Pi_s$, Lemma 7.9 guarantees that all the resources required by J_H are available when J_H preempts. Since any job that preempts J_H also preempts J , the induction hypothesis guarantees that J_H executes to completion without blocking, as will any job that preempts J_H , transitively. When all the jobs that preempted J complete, J can resume its execution without blocking, since the higher-priority jobs released all resources and when J started the resources available were sufficient to meet the maximum request of J . \square

Theorem 7.6 (Baker) *Under the Stack Resource Policy, a job J_i can be blocked for at most the duration of one critical section.*

Proof. Suppose that J_i is blocked for the duration of two critical sections shared with two lower-priority jobs, J_1 and J_2 . Without loss of generality, assume $\pi_2 < \pi_1 < \pi_i$. This can happen only if J_1 and J_2 hold two different resources (such as R_1 and R_2) and J_2 is preempted by J_1 inside its critical section. This situation is depicted in Figure 7.20. This immediately yields to a contradiction. In fact, since J_1 is not blocked by the preemption test, we have $\pi_1 > \Pi_s$. On the other hand, since J_i is blocked, we have $\pi_i \leq \Pi_s$. Hence, we obtain that $\pi_i < \pi_1$, which contradicts the assumption. \square

Theorem 7.7 (Baker) *The Stack Resource Policy prevents deadlocks.*

Proof. By Theorem 7.5, a job cannot be blocked after it starts. Since a job cannot be blocked while holding a resource, there can be no deadlock. \square

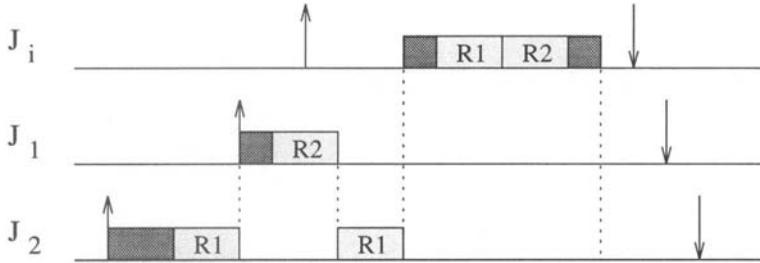


Figure 7.20 An absurd situation that cannot occur under SRP.

7.5.4 Schedulability analysis

As far as the schedulability analysis is concerned, the considerations done for the Priority Ceiling Protocol are also valid for the Stack Resource Policy, since the general result does not depend on the time on which a job is blocked. However, if the SRP is used along with the EDF scheduling algorithm, the guarantee test has to be modified by considering that under EDF the least upper bound of the processor utilization factor is 1.

As a result, a set of n periodic tasks using the Stack Resource Policy can be scheduled by the EDF algorithm if

$$\forall i, \quad 1 \leq i \leq n, \quad \left(\sum_{k=1}^i \frac{C_k}{T_k} \right) + \frac{B_i}{T_i} \leq 1. \quad (7.7)$$

As for the PCP, C_i denotes the worst-case execution time of task τ_i , T_i denotes its period, and B_i its maximum blocking time. For each task τ_i , the sum in parentheses represents the utilization factor due to τ_i itself and to all tasks having a preemption level higher than π_i , whereas the term B_i/T_i considers the blocking time caused by tasks having preemption level lower than π_i . Condition (7.7) can easily be extended to periodic tasks with deadlines less than periods. In this case, the schedulability test is modified as follows:

$$\forall i, \quad 1 \leq i \leq n, \quad \left(\sum_{k=1}^i \frac{C_k}{D_k} \right) + \frac{B_i}{D_i} \leq 1. \quad (7.8)$$

A more precise schedulability condition can be achieved through a processor demand approach [BRH90, JS93]. In particular, equation (4.18) has been extended in [BL97, Lip97], where it is proved that a set of periodic tasks that use

shared resources with SRP is schedulable by EDF if for all $L \geq 0$ and for all $1 \leq i \leq n$

$$\sum_{k=1}^i \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k + \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) B_i \leq L. \quad (7.9)$$

7.5.5 Blocking time computation

The maximum blocking time that a job 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 job J_i can be blocked for at most the duration of one critical section among those that can block J_i . Lemma 7.8, proved for the PCP, can be easily extended to the SRP, thus a critical section $Z_{j,k}$ belonging to job J_j and guarded by semaphore S_k can block a job J_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, the maximum blocking time B_i of job J_i can be computed as the duration of the longest critical section among those belonging to tasks with pre-emption level lower than π_i and guarded by a semaphore with maximum ceiling higher than or equal to π_i . If $D_{j,k}$ denotes the duration of the longest critical section of task τ_j among those guarded by semaphore S_k , we can write

$$B_i = \max_{j,k} \{D_{j,k} \mid \pi_j < \pi_i, C_{S_k}(0) \geq \pi_i\}. \quad (7.10)$$

7.5.6 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 (that is, 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.

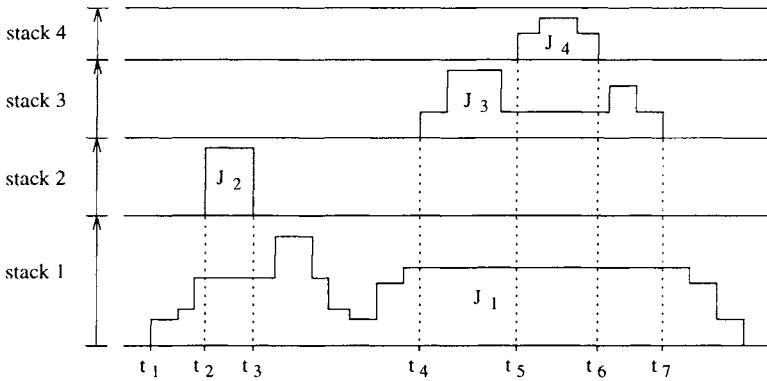


Figure 7.21 Possible evolution with one stack per task.

For example, consider four jobs J_1 , J_2 , J_3 , and J_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 job is allocated its own stack space, equal to its maximum requirement. At time t_1 , J_1 starts executing; J_2 preempts at time t_2 and completes at time t_3 , allowing J_1 to resume. At time t_4 , J_1 is preempted by J_3 , which in turn is preempted by J_4 at time t_5 . At time t_6 , J_4 completes and J_3 resumes. At time t_7 , J_3 completes and J_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 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 job J is preempted by a job J' , J maintains its stack and the stack of J' is allocated immediately above that of J . 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 job J can never be blocked once started, its stack can never be penetrated by the ones belonging to jobs with lower preemption levels, which can resume only after J is completed.

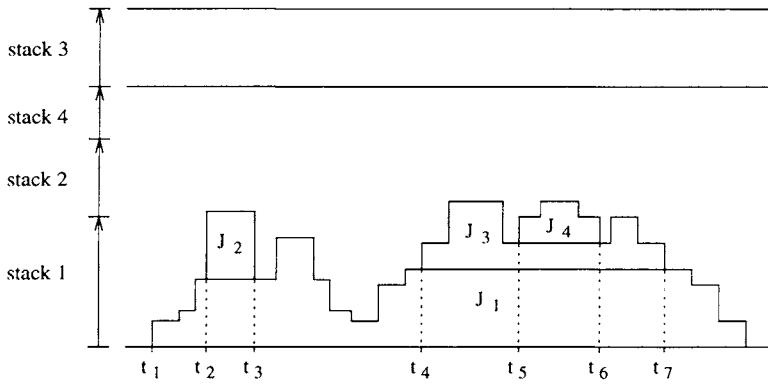


Figure 7.22 Possible evolution with a single stack for all tasks.

Note that the stack space between the two upper horizontal lines (which is equivalent to the minimum stack between J_2 and J_3) is no longer needed, since J_2 and J_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 jobs distributed on 10 preemption levels, with 10 jobs for each level, and suppose that each job needs up to 10 Kbytes of stack space. Using a stack per job, 1000 Kbytes would be required. On the contrary, using a single stack, only 100 Kbytes would be sufficient, since no more than one job 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.

7.5.7 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 job can never be blocked when attempting to lock a semaphore. When there are no sufficient resources available to satisfy the maximum requirement of a job, the job 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 job 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.

7.6 SUMMARY

The concurrency control protocols presented in this chapter can be compared with respect to several characteristics. Figure 7.23 provides a qualitative evaluation of the algorithms in terms of priority assignment, number of blockings, instant of blocking, programming transparency, deadlock prevention, implementation, and complexity for computing the blocking factors. Notice that the Priority Inheritance Protocol (PIP), although not so effective in terms of performance, is the only one that is transparent at the programming level. The other protocols, in fact, require the user to specify the list of resources used by each task, in order to compute the ceiling values. This feature of PIP makes it attractive for commercial operating systems (like VxWorks), where predictability can be improved without introducing new kernel primitives.

	priority assignment	number of blocking	blocking instant	transparency	deadlock prevention	implementation	B_i computation
PIP	fixed	$\min(n,m)$	on resource access	YES	NO	hard	hard
PCP	fixed	1	on resource access	NO	YES	medium	easy
SRP	fixed or dynamic	1	on preemption	NO	YES	easy	easy

Figure 7.23 Evaluation summary of resource access protocols.

Exercises

- 7.1 Verify whether the following task set is schedulable by the Rate-Monotonic algorithm (try both the processor utilization and the worst-case response approach):

	τ_1	τ_2	τ_3
C_i	4	3	2
B_i	5	3	0
T_i	10	15	20

- 7.2 Consider three periodic tasks τ_1 , τ_2 , and τ_3 (having decreasing priority), which share four resources, A , B , C , and D , accessed using the Priority Inheritance Protocol. Compute the maximum blocking time B_i for each task, knowing that the longest duration D_{iR} for a task τ_i on resource R is given in the following table (there are no nested critical sections):

	A	B	C	D
τ_1	3	2	4	6
τ_2	4	0	6	8
τ_3	2	1	0	5

- 7.3 Solve the same problem described in Exercise 7.2 when the resources are accessed by the Priority Ceiling Protocol.
- 7.4 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 D_{iR} 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	12	5	9	8	0
τ_2	10	0	7	0	6
τ_3	0	3	0	7	13
τ_4	10	0	8	0	5

- 7.5 Solve the same problem described in Exercise 7.4 when the resources are accessed by the Priority Ceiling Protocol.

- 7.6 Consider three tasks J_1 , J_2 , and J_3 , which share three multiunit 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_{R1}	μ_{R2}	μ_{R3}
J_1	5	1	0	1
J_2	10	2	1	3
J_3	20	3	1	1

8

HANDLING OVERLOAD CONDITIONS

8.1 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 time available on the processor, and hence not all tasks can complete within their deadlines.

In real-world applications, even when the system is properly designed and sized, a transient overload can occur for different reasons, such as changes in the environment, simultaneous arrivals of asynchronous events, faults of peripheral devices, or system exceptions. The major risk that could occur in these situations is that some critical task could miss its deadline, jeopardizing the correct behavior of the whole system.

If the operating system is not conceived to handle overloads, the effect of a transient overload can be catastrophic. Experiments carried out by Locke [Loc86] have shown that EDF can rapidly degrade its performance during overload intervals. This is due to the fact that EDF gives the highest priority to those processes that are close to missing their deadlines. 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 Figure 8.1.

Figure 8.1a shows a feasible 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 Figure 8.1b). In such a situation, EDF does not provide any type of guarantee on which tasks meet their timing constraints. This is a very undesirable

behavior in those control applications in which a critical subset of tasks has to be guaranteed in all anticipated load conditions. In order to avoid domino effects, the operating system and the scheduling algorithm must be explicitly designed to handle transient overloads in a controlled fashion, so that the damage due to a deadline miss can be minimized.

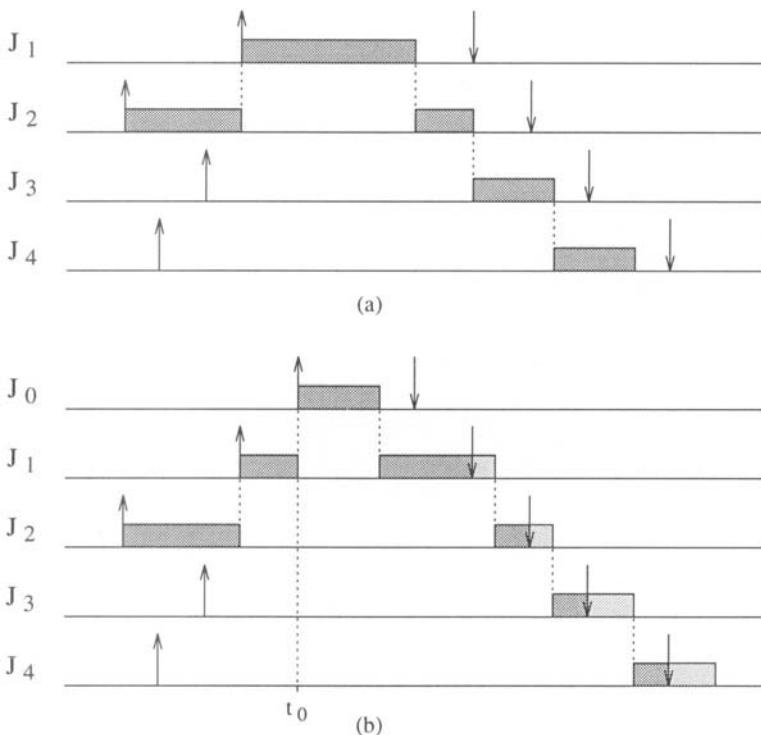


Figure 8.1 a. Feasible schedule with Earliest Deadline First, in normal load condition. b. Overload with domino effect due to the arrival of task J₀.

In the real-time literature, several scheduling algorithms have been proposed to deal with overloads. In 1984, Ramamritham and Stankovic [RS84] used EDF to dynamically guarantee incoming work via on-line planning, and, if a newly arriving task could not be guaranteed, the task was either dropped or distributed scheduling was attempted. The dynamic guarantee performed in this approach had the effect of avoiding the catastrophic effects of overload on EDF.

In 1986, Locke [Loc86] developed an algorithm that makes a best effort at scheduling tasks based on earliest deadline with a rejection policy based on removing tasks with the minimum value density. He also suggested that removed tasks remain in the system until their deadline has passed. The algorithm computes the variance of the total slack time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the tasks in increasing value density order.

In Biyabani et. al. [BSR88] the previous work of Ramamritham and Stankovic was extended 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. This work used values of tasks such as in Locke's work but used an exact characterization of the first overload point rather than a probabilistic estimate that overload might occur.

Haritsa, Livny, and Carey [HLC91] presented the use of a feedback controlled EDF algorithm for use in real-time database systems. The purpose of their work was to obtain good average performance for transactions even in overload. Since they were working in a database environment, they assumed no knowledge of transaction characteristics, and they considered tasks with soft deadlines that are not guaranteed.

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 work on overload in real-time systems has also been done. 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. However, they provided no details on the scheduling algorithm itself. Schwan and Zhou [SZ92] did on-line 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).

More recent approaches will be described in the following sections. Before presenting specific methods and theoretical results on overload, the concept of overload, and, in general, the meaning of computational load for real-time systems is defined in the next section.

8.2 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 C is the mean service time and λ is the average interarrival rate of the jobs, the load can be computed as

$$\rho = \lambda C.$$

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:

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

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. A general definition of load has been proposed by Baruah et al. [BKM⁺92], who

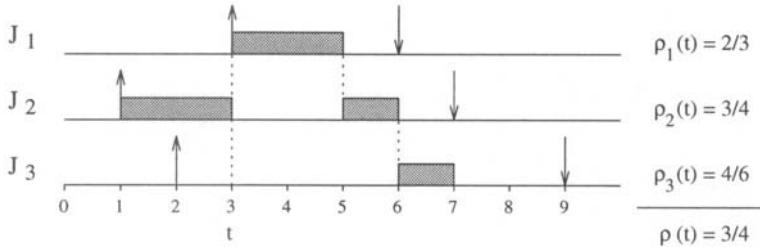


Figure 8.2 Load calculation for a set of three real-time tasks.

say that a hard real-time environment has a loading factor ρ if and only if it is guaranteed that there will be no interval of time $[t_x, t_y]$ such that the sum of the execution times of all jobs making requests and having deadlines within this interval is greater than $\rho(t_y - t_x)$. Although this definition is quite general and of theoretical value, it is of little practical use for load calculation, since the number of intervals $[t_x, t_y]$ can be very large.

A simpler method for calculating the load in a dynamic real-time environment has been proposed by Buttazzo and Stankovic in [BS95], where the load is computed at each job activation (r_i), and the number of intervals in which the computation is done is limited by the number of deadlines (d_i). The method for computing the load is based on the consideration that, for a single job J_i , the load is given by the ratio of its computation time C_i and its relative deadline $D_i = d_i - r_i$. For example, if $C_i = D_i$ (that is, the job does not have slack time), the load in the interval $[r_i, d_i]$ is one. When a new job arrives, the load can be computed from the last request time, which is also the current time t , and the longest deadline, say d_n . In this case, the intervals that need to be considered for the computation are $[t, d_1], [t, d_2], \dots, [t, d_n]$. In general, the processor load in the interval $[t, d_i]$ is given by

$$\rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)},$$

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 in the interval $[t, d_n]$ can be computed as the maximum among all $\rho_i(t)$; that is,

$$\rho = \max_i \rho_i(t).$$

Figure 8.2 shows an example of load calculation for a set of three real-time tasks.

8.3 PERFORMANCE METRICS

When a real-time system is underloaded and dynamic activation of tasks is not allowed, there is no need to consider task importance in the scheduling policy, since there exist optimal scheduling algorithms that can guarantee a feasible schedule under a set of assumptions. For example, Dertouzos [Der74] proved that EDF is an optimal algorithm for preemptive, independent tasks when there is no overload.

On the contrary, when tasks can be 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 ten seconds is certainly more important than updating the clock picture on the user console every second. This means that, during a transient overload, 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 an utility function. Figure 8.3 illustrates some utility functions that can be associated with tasks in order to describe their importance. According to this view, 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 com-

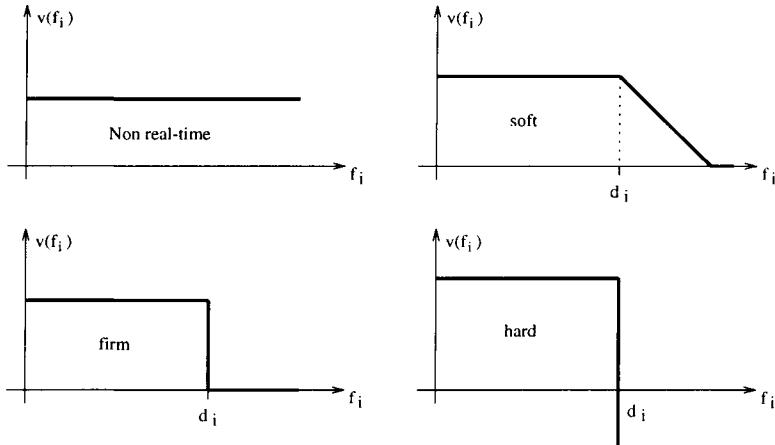


Figure 8.3 Utility functions that can be associated to a task to describe its importance.

pletes 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 task with a soft deadline, instead, can still give a value to the system if executed after its deadline, although this value may decrease with time. Then, there can be real-time activities, so-called *firm*, that do not jeopardize the system but give zero value if completed after their deadline.

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 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 (included

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 underload 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 is the relative deadline, and V_i is 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^* .

8.3.1 On-line versus clairvoyant scheduling

Since dynamic environments require on-line scheduling, it is important to analyze the properties and the performance of on-line scheduling algorithms in overload conditions.

Although there are optimal on-line algorithms in underload conditions, it is easy to show that no optimal on-line algorithms exist in overloads. Consider for example the task set shown in Figure 8.4, 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 Figure 8.4a). 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 Figure 8.4b). Notice that if J_3 arrives at time $t = 9$ or later (see Figure 8.4c), then the maximum cumulative value is $\Gamma^* = 16$ and can be accumulated by scheduling tasks J_1 and J_3 . Hence, at time $t = 0$, without knowing the arrival time of

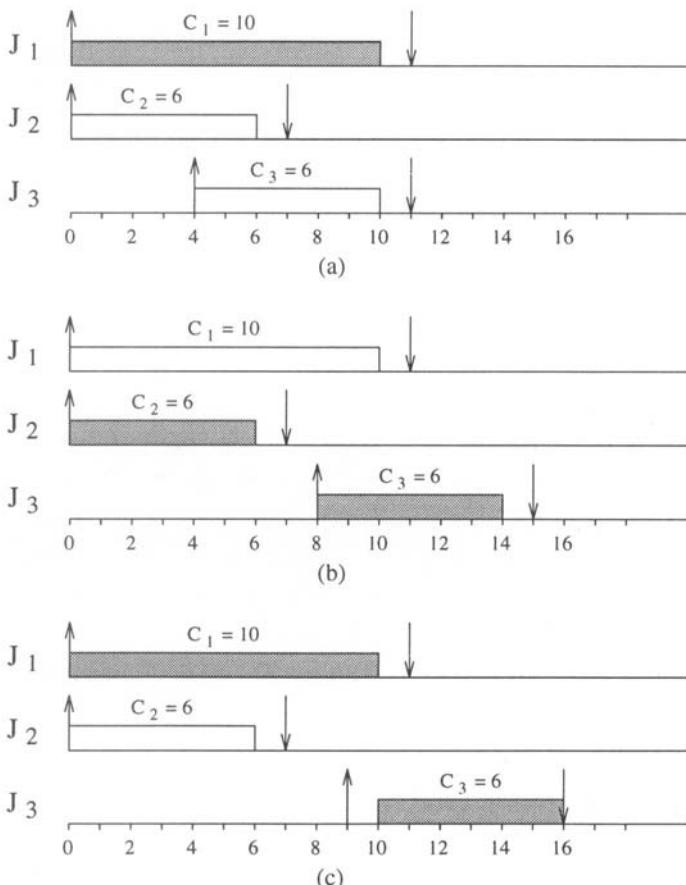


Figure 8.4 No optimal on-line optimal algorithms exist in overload conditions, since the schedule that maximizes Γ depends on the knowledge of future arrivals. **a.** $\Gamma_{max} = 10$. **b.** $\Gamma_{max} = 12$. **c.** $\Gamma_{max} = 16$.

J_3 , no on-line 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 on-line 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 on-line scheduling algorithms in overload conditions.

8.3.2 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 8.1 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 can notice 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 on-line 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 this fact it is sufficient to find an overload

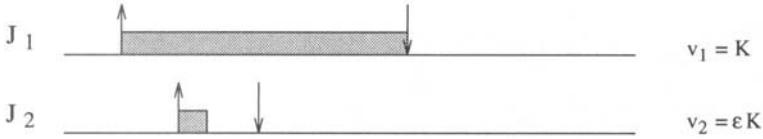


Figure 8.5 Situation in which EDF has an arbitrarily small competitive factor.

situation in which the cumulative value obtained by EDF can be arbitrarily reduced with respect to that one achieved by the clairvoyant scheduler. Consider the example shown in Figure 8.5, 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 on-line algorithm. This is stated by the following theorem.

Theorem 8.1 (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 on-line algorithm can guarantee a competitive factor greater than 0.25.*

The proof of this theorem is done by using an adversary argument, in which the on-line 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 on-line algorithm.

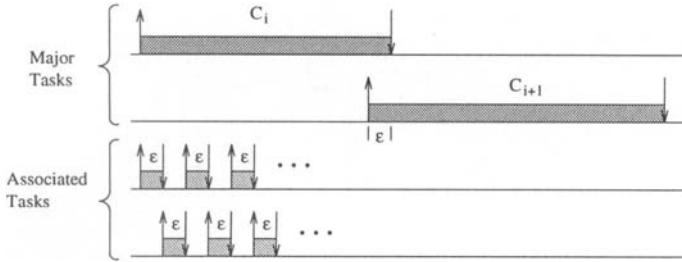


Figure 8.6 Task sequence generated by the adversary.

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 Figure 8.6):

- 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 Figure 8.6). 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.

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

$$J_0, J_1, J_2, \dots, J_i, \dots, J_m$$

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 $(C_0 + C_1 + \dots + C_{i+1})$. 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

$$\varphi_i = \varphi_{i-1} = \dots = \varphi_0 = \frac{1}{k}.$$

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:

$$\begin{cases} C_0 &= 1 \\ C_{i+1} &= kC_i - \sum_{j=0}^i C_j. \end{cases} \quad (8.1)$$

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,

$$\frac{C_m}{\sum_{j=0}^m C_j} \leq \frac{1}{k}, \quad (8.2)$$

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 equation (8.2), it also satisfies the following equation:

$$C_m \leq C_{m-1}. \quad (8.3)$$

Thus, (8.3) is satisfied if and only if (8.2) is satisfied.

From (8.1) we can also write

$$\begin{aligned} C_{i+2} &= kC_{i+1} - \sum_{j=0}^{i+1} C_j \\ C_{i+1} &= kC_i - \sum_{j=0}^i C_j, \end{aligned}$$

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, equation (8.1) is equivalent to

$$\begin{cases} C_0 &= 1 \\ C_1 &= k - 1 \\ C_{i+2} &= k(C_{i+1} - C_i). \end{cases} \quad (8.4)$$

From this result, we can say that the tightest bound on the competitive factor of an on-line algorithm is given by the smallest ratio $1/k$ (equivalently, the largest k) such that (8.4) satisfies (8.3). Equation (8.4) is a recurrence relation that can be solved by standard techniques [Sha85]. The characteristic equation of (8.4) is

$$x^2 - kx + k = 0,$$

which has roots

$$x_1 = \frac{k + \sqrt{k^2 - 4k}}{2} \quad \text{and} \quad x_2 = \frac{k - \sqrt{k^2 - 4k}}{2}.$$

When $k = 4$, we have

$$C_i = d_1 i 2^i + d_2 2^i, \quad (8.5)$$

and when $k \neq 4$ we have

$$C_i = d_1 (x_1)^i + d_2 (x_2)^i, \quad (8.6)$$

where values for d_1 and d_2 can be found from the boundary conditions expressed in (8.4). We now show that for ($k = 4$) and ($k > 4$) C_i will diverge, so equation (8.3) will not be satisfied, whereas for ($k < 4$) C_i will satisfy (8.3).

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,

$$C_i = \left(\frac{i}{2} + 1\right) 2^i,$$

which clearly diverges. Hence, for $k = 4$, equation (8.3) 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 $k > 4$, equation (8.3) 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:

$$\begin{cases} d_1 = se^{j\theta} \\ d_2 = se^{-j\theta} \end{cases} \quad \begin{cases} x_1 = re^{j\omega} \\ x_2 = re^{-j\omega}, \end{cases}$$

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 (8.6) 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 (8.3).

Since (8.3) is satisfied for $k < 4$, the largest k that determines the competitive factor of an on-line 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 on-line scheduling algorithm in an overloaded environment. Hence, Theorem 8.1 follows.

Extensions

Theorem 8.1 establishes an upper bound on the competitive factor of on-line 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 8.2 (Baruah et al.) *In real-time environments with a loading factor ρ , $1 < \rho \leq 2$, and task values equal to computation times, no on-line algorithm can guarantee a competitive factor greater than p , where p satisfies*

$$4[1 - (\rho - 1)p]^3 = 27p^2. \quad (8.7)$$

Notice that, for $\rho = 1 + \epsilon$, equation (8.7) 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 Figure 8.7.

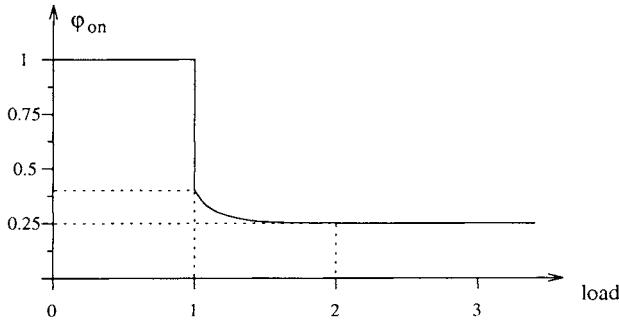


Figure 8.7 Bound of the competitive factor of an on-line scheduling algorithm as a function of the load.

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 on-line 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 on-line algorithm can achieve a competitive factor greater than

$$\frac{1}{(1 + \sqrt{q})^2},$$

whereas, if $q < 1$, no on-line 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, tasks characteristics are much less restrictive; therefore, the 1/4th bound has only a theoretical validity, and more work is needed to derive other bounds based on more knowledge of the actual environmental load conditions. An analysis of on-line scheduling algorithms under different types of adversaries has been presented by Karp in [Kar92].

8.4 SCHEDULING SCHEMES FOR OVERLOAD

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 Figure 8.8:

- **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.
- **Guarantee.** This class includes those algorithms in which the load on the processor is controlled by an acceptance test executed at each task arrival. Typically, 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 ready queue; otherwise, it is rejected.
- **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.

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). Guarantee algorithms, however, do not take task importance into account and, during transient overloads, always reject 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.

In guarantee and robust algorithms, 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 following sections we present a few examples of scheduling algorithms for handling overload situations and then compare their performance for different peak load conditions.

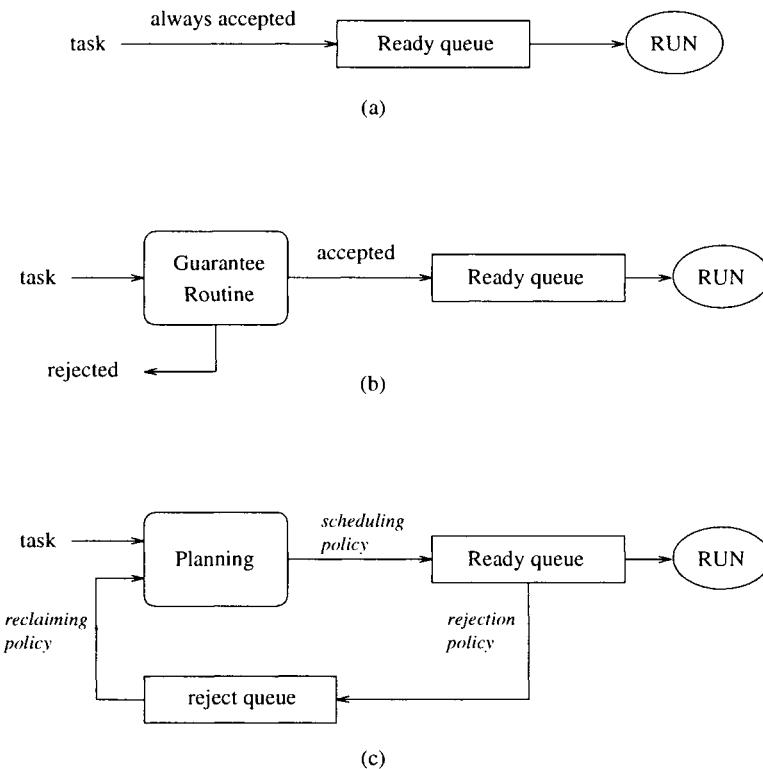


Figure 8.8 Scheduling schemes for handling overload situations. **a.** Best Effort. **b.** Guarantee. **c.** Robust.

8.4.1 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, we could find that a task set is not feasibly schedulable and hence decide to reject a task. But, in reality, the system could have been scheduled within the tolerance levels. Another positive effect of tolerance is that various tasks could actually finish before their worst-case times, so a resource reclaiming mechanism could then compensate, and the tasks with tolerance could actually finish on time.

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 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 8.1 *Given a set $J = \{J_1, J_2, \dots, J_n\}$ 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*

$$L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t), \quad (8.8)$$

where $L_0 = 0$, $d_0 = t$ (that is, 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 in the set J are ordered by increasing deadlines, task 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. \square

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

$$E_i = \max(0, -(L_i + M_i)). \quad (8.9)$$

We also define the *Maximum Exceeding Time* E_{max} as the maximum among all E_i 's in the tasks 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 $E_{max} = 0$.

¹If $M_i = 0$, the *Exceeding Time* is also called the *Tardiness*.

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 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 Figure 8.9.

```
RED_acceptance_test( $J, J_{new}$ ) {
     $E = 0;$  /* Maximum Exceeding Time */
     $L_0 = 0;$ 
     $d_0 = current\_time();$ 

     $J' = J \cup \{J_{new}\};$ 
     $k = <\text{position of } J_{new} \text{ in the task set } J'>;$ 

    for each task  $J'_i$  such that  $i \geq k$  do {
        /* compute the maximum exceeding time */
         $L_i = L_{i-1} + (d_i - d_{i-1}) - c_i;$ 
        if ( $L_i + M_i < -E$ ) then  $E = -(L_i + M_i);$ 
    }

    if ( $E > 0$ ) {
        <select a set  $J^*$  of least-value tasks to be rejected>;
        <reject all task in  $J^*$ >;
    }
}
```

Figure 8.9 The RED acceptance test.

In RED, a resource reclaiming mechanism is used to take advantage of those tasks that complete before their worst-case finishing time. To reclaim the spare time, rejected tasks are not removed forever but temporarily parked in a queue, called *Reject Queue*, ordered by decreasing values. Whenever a running task

completes its execution before its worst-case finishing time, the algorithm tries to reaccept the highest-value tasks in the Reject Queue having positive laxity. Tasks with negative laxity are removed from the system.

8.4.2 D_{over} : a competitive algorithm

Koren and Shasha [KS92] found an on-line 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 on-line 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 worth to observe that having the best competitive factor among all on-line 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 Section 8.5, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

8.5 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 guarantee and robust scheduling paradigm 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 Figure 8.10, 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.

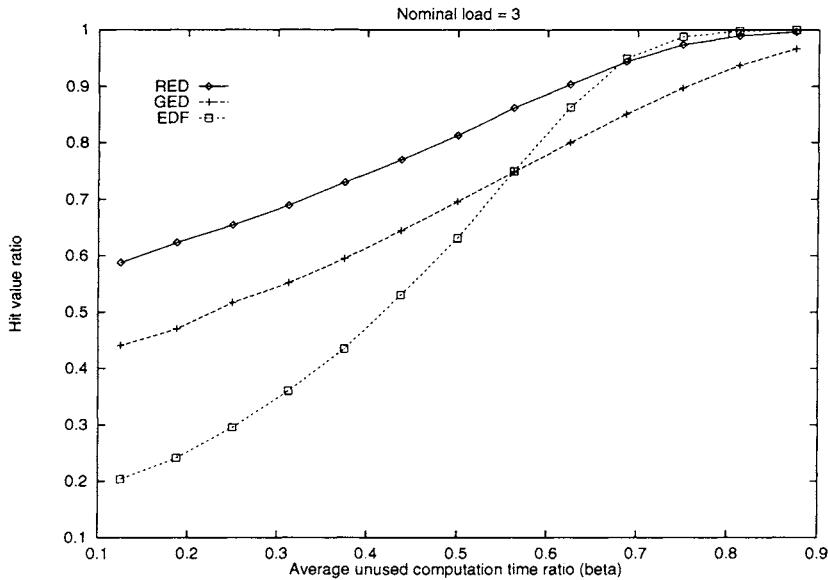


Figure 8.10 Performance of various EDF scheduling schemes: best-effort (EDF), guarantee (GED) and robust (RED).

For small values of β , that is, when tasks execute for almost their maximum computation time, the guarantee (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 with the highest value density (v_i/C_i) is scheduled first, regardless of its deadline. Performance results are shown in Figure 8.11.

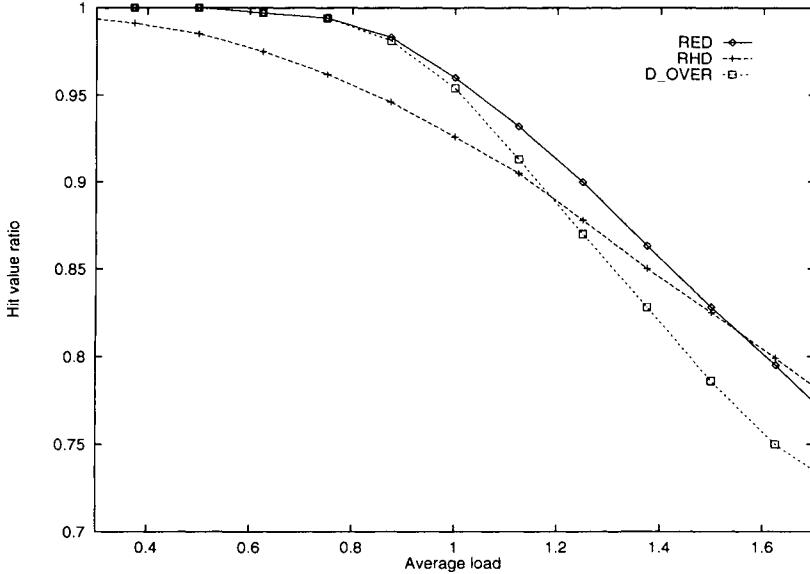


Figure 8.11 Performance of D_{over} against RED and RHD.

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 on-line 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.

9

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* Kernel), 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.

9.1 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, and
- 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 of a dedicated routine (driver) that will transfer data from the device to the main memory (or viceversa). 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 Chapter 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 (*DIdactic C Kernel*). 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.

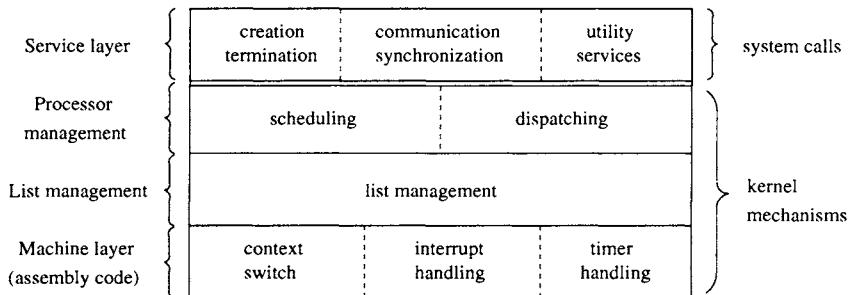


Figure 9.1 Hierarchical structure of DICK.

The various functions developed in DICK are organized according to the hierarchical structure illustrated in Figure 9.1. Those low-level activities that directly 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.

9.2 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 Figure 9.2.

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.

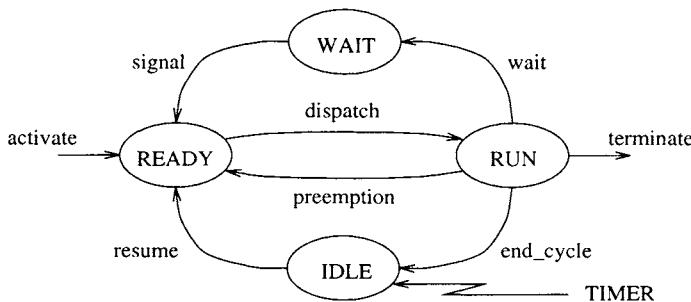


Figure 9.2 Minimum state transition diagram of a real-time kernel.

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 Figure 9.3, 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).

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 Figure 9.4, 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.

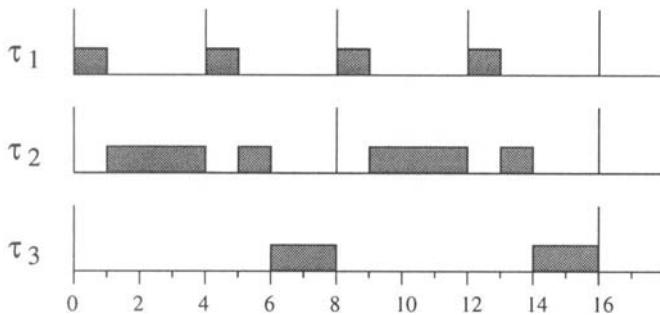


Figure 9.3 Feasible schedule of three periodic tasks under RM.

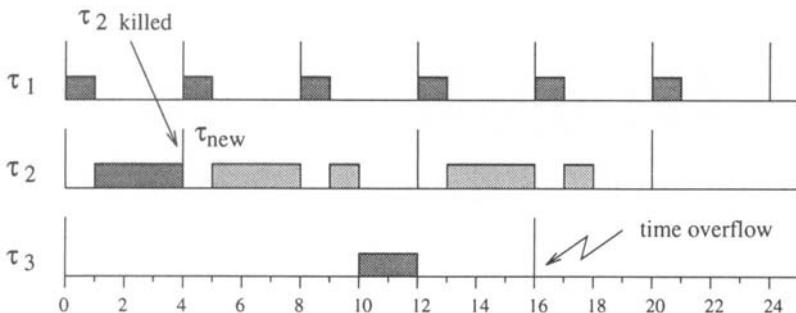


Figure 9.4 The effects of τ_2 do not cancel at the time it is aborted, but protract till the end of its period.

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 9.5 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 Figure 9.6. Notice that, at the end of its last period, a periodic task (aborted or terminated) leaves the system completely and all its data structures are deallocated.

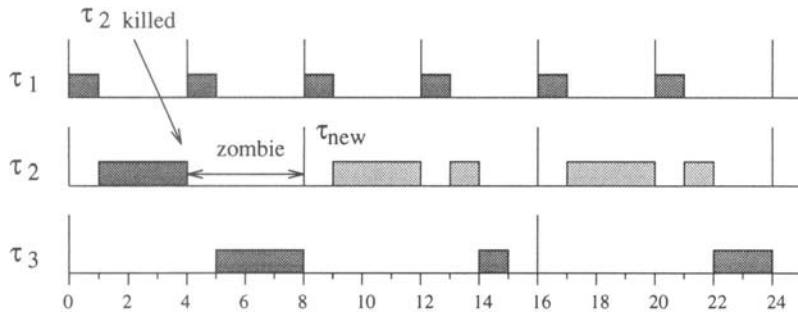


Figure 9.5 The new task set is schedulable when τ_{new} is activated at the end of the period of τ_2 .

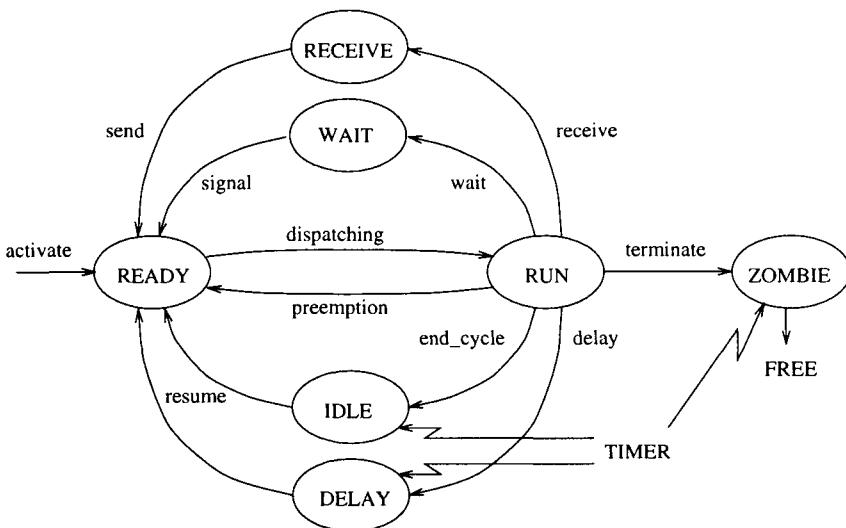


Figure 9.6 State transition diagram including RECEIVE, DELAY, and ZOMBIE states.

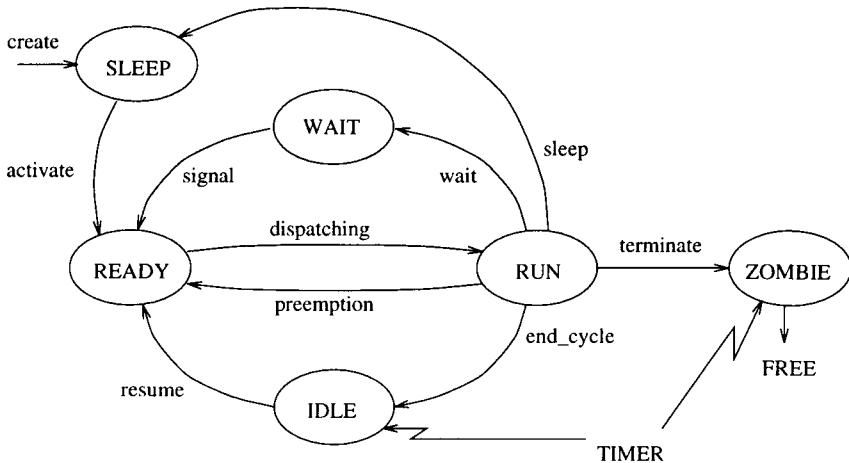


Figure 9.7 State transition diagram in DICK.

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 for reducing the runtime overhead of the activation primitive. The state transition diagram used in DICK is illustrated in Figure 9.7.

9.3 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 Figure 9.8 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 criticalness (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.

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.

Task Control Block
task identifier
task address
task type
criticalness
priority
state
computation time
period
relative deadline
absolute deadline
utilization factor
context pointer
precedence pointer
resource pointer
pointer to the next TCB

Figure 9.8 Structure of the Task Control Block.

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 9.9 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 Figure 9.10):

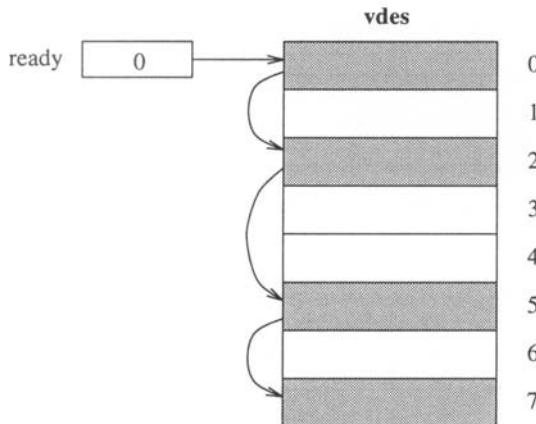


Figure 9.9 Implementation of the ready queue as a list of Task Control Blocks.

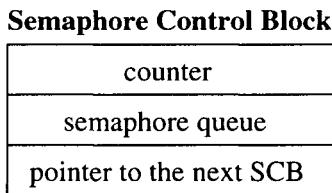


Figure 9.10 Semaphore Control Block.

- 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 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	*/

9.4 MISCELLANEOUS

9.4.1 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	<code>sys_clock;</code>	/* system time	*/
float	<code>time_unit;</code>	/* 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 9.1 shows the values of the system lifetime for some tick values.

<i>tick</i>	<i>lifetime</i>
1 ms	50 days
5 ms	8 months
10 ms	16 months
50 ms	7 years

Table 9.1 System lifetime for some typical 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 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 milliseconds. 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 the timer routine is proportional to its interrupt rate. In Section 9.7 we see how this overhead can be evaluated and taken into account in the schedulability analysis.

9.4.2 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 criticalness. To simplify the description of the kernel, only two classes of tasks are considered in DICK:

- HARD tasks, having a critical deadline, and
- 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 background based on their priority.

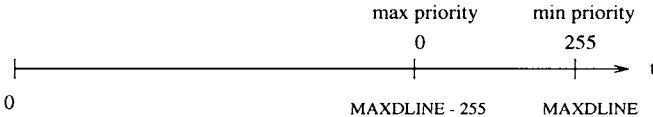


Figure 9.11 Mapping NRT priorities into deadlines.

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 Figure 9.11 and is such that

$$d_i^{NRT} = MAXLINE - PRT_LEV + P_i,$$

where MAXLINE 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.

9.4.3 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 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 MAXLINE	0x7FFFFFFF	/* max deadline */
#define PRT.LEV	255	/* priority levels */
#define NIL	-1	/* null pointer */
#define TRUE	1	
#define FALSE	0	
#define LIFETIME	MAXLINE - 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 */
/*-----*
```

9.4.4 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 a NRT task in which new concurrent tasks can be created.

The most important activities performed by *ini_system* concern

- 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>;
}
```

9.5 KERNEL PRIMITIVES

The structure of DICK is logically divided in a number of hierarchical layers, as illustrated in Figure 9.1. 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.

9.5.1 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 context of pexe */
    <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 context of pexe */
    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 */
    <return from interrupt>
}
```

9.5.2 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 (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 Figure 9.12.

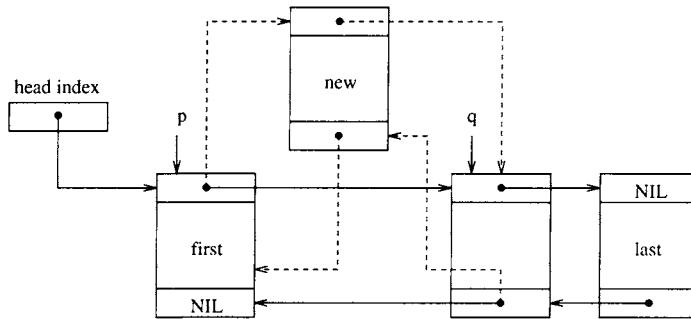


Figure 9.12 Inserting a TCB in a queue.

```
/*
 *-----*
 * 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;
}
```

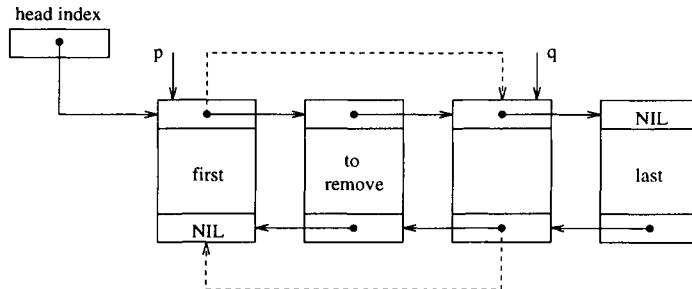


Figure 9.13 Extracting a TCB from a queue.

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 9.13 illustrates the extraction of a generic element, whereas Figure 9.14 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);
}
```

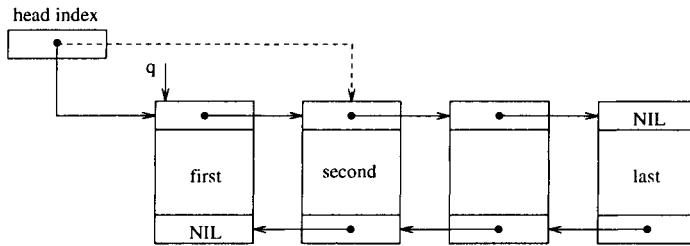


Figure 9.14 Extracting the TCB at the head of a queue.

```
/*
 *-----*
 * 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: *firstline* 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);
}
```

9.5.3 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 Section 9.4.1. 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 pre-emption 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();
}
```

9.5.4 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 - PRTLEV);
<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[p].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[p].dline = dl;
        vdes[p].state = READY;
        insert(pexe, &ready);
    }
    dispatch();
    load_context();
}
```

A typical example of periodic task is shown in the following code:

```
proc      cycle()
{
    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>
}
```

9.5.5 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 */
    /* of the freesem list */
    freesem = s;
    <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 Chapter 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 Section 9.6.

9.5.6 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);
}
```

9.6 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 the most 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 *rendez-vous*. 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 *rendez-vous*. In a static real-time environment, the problem can be solved off-line by transforming all synchronous interactions into precedence 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 Figure 9.15.

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

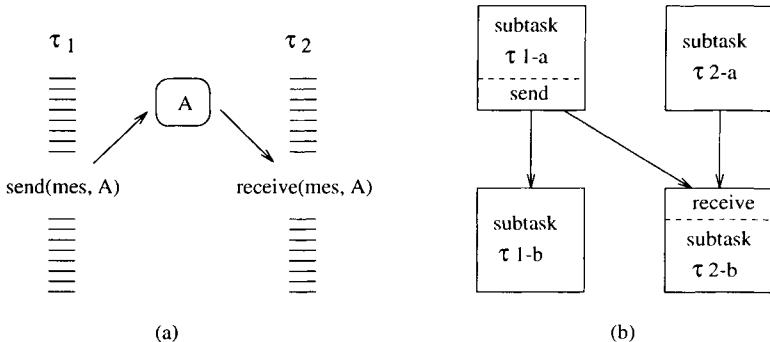


Figure 9.15 Decomposition of communicating tasks (a) into subtasks with precedence constraints (b).

complexity of the system (for example, overconstraining the task set with additional precedence relations). Remember that having simple on-line 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 Figure 9.16. 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 – namely, *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 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.

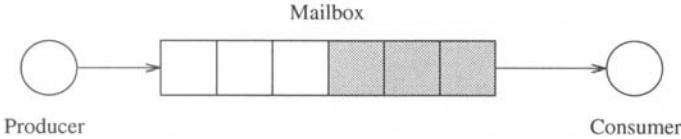


Figure 9.16 The mailbox scheme.

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). Viceversa, 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.

9.6.1 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, BD93] 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);
<copy message in *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);
<use message>
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.

9.6.2 CAB implementation

The data structure used to implement a CAB is shown in Figure 9.17. 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.

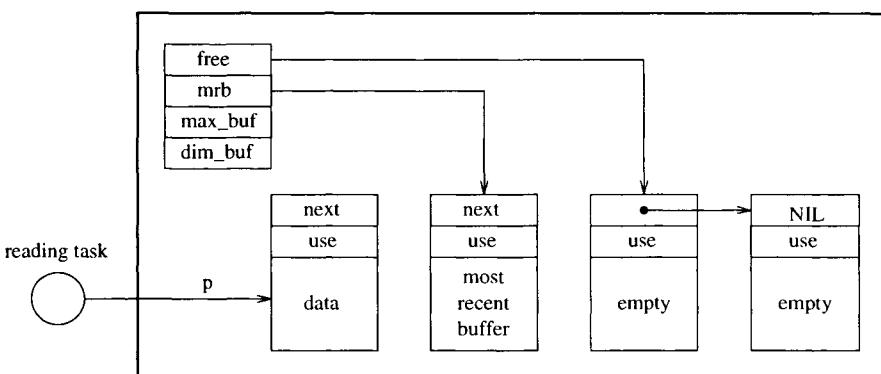


Figure 9.17 CAB data structure.

```
/*-----*/
/* 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 accessed */
/*           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>
}
```

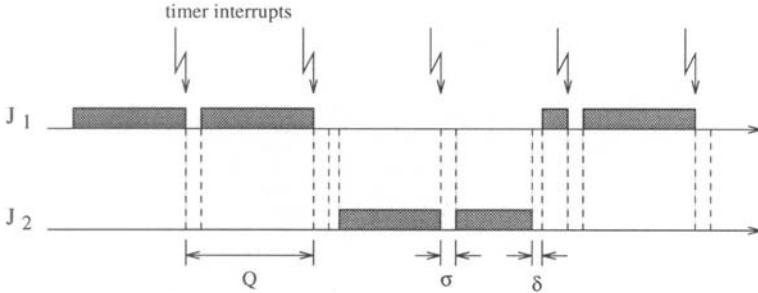


Figure 9.18 Effects of the overhead on tasks' execution.

9.7 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 9.18 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

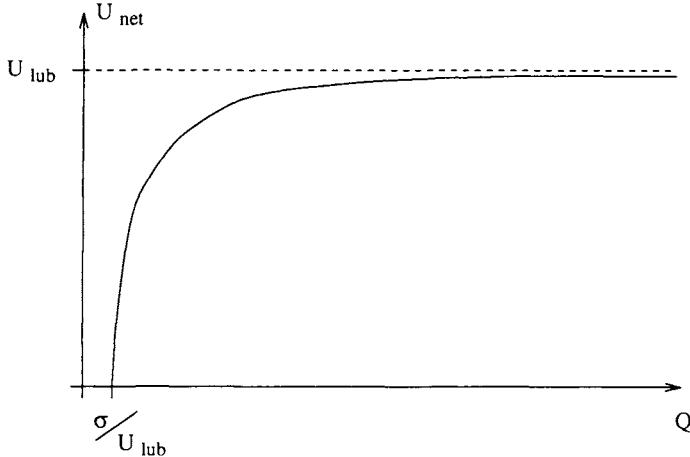


Figure 9.19 Net utilization bound as a function of the tick value.

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

$$U_{net} = U_{lub} - U_t = U_{lub} - \frac{\sigma}{Q} = U_{lub} \left(\frac{Q - \sigma/U_{lub}}{Q} \right).$$

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 Figure 9.19. 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 $\sigma = 100\mu s$. 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 off-line 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.

9.7.1 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 in to 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]:

$$\begin{aligned} f(0) &= 0 \\ f(l) &= \begin{cases} f(l-1) + 1 & \text{if } \sum_{i=1}^m \left\lceil \frac{l}{T_i^H} \right\rceil C_i^H > f(l-1) \\ f(l-1) & \text{otherwise.} \end{cases} \end{aligned} \quad (9.1)$$

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 processor in executing interrupt handlers in the interval $[0, l]$.

Theorem 9.1 (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 , $L \geq 0$,*

$$\sum_{i=1}^n \left\lceil \frac{L}{T_i} \right\rceil C_i \leq L - f(L). \quad (9.2)$$

The proof of Theorem 9.1 is very similar to the one presented for Theorem 4.2. 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 equation (9.2) 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.

10

APPLICATION DESIGN ISSUES

In this chapter we discuss some important 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 Chapter 9 is used to illustrate some concrete programming examples of real-time tasks for sensory processing and control activities.

10.1 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.
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 Figure 10.1.

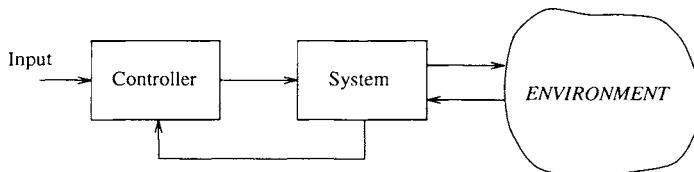


Figure 10.1 Block diagram of a generic control system.

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.

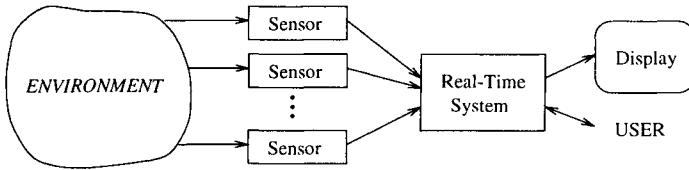


Figure 10.2 General structure of a monitoring system.

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 Figure 10.2. 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 feedback between sensors and actuators. This means that, once an action is planned, it can be executed independently of new sensory data (see Figure 10.3).

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 on-line; 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

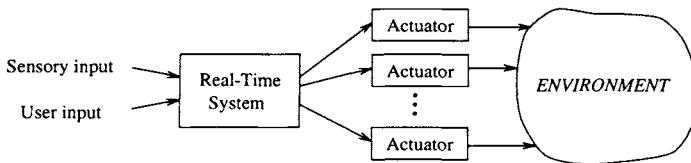


Figure 10.3 General structure of an open-loop control system.

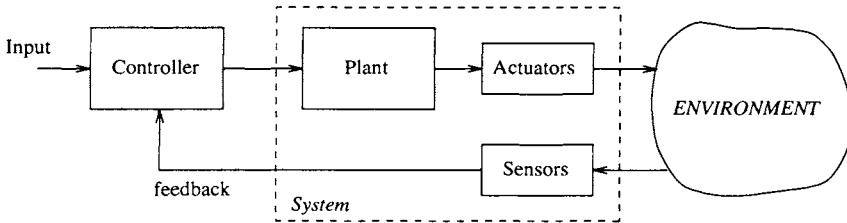


Figure 10.4 General structure of a feedback control system.

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 Figure 10.4).

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.

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 on-line. 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.

10.2 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. Viceversa, 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 criticalness), 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 rest of this section we illustrate a few examples of control systems to show how time constraints can be derived from the application requirements even in those cases in which they are not explicitly defined by the user.

10.2.1 Obstacle avoidance

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 autonomous system driven by a human, such as a car or a train having an automatic braking system for stopping motion in emergency situations.

In order to simplify our discussion and reduce the number of controlled variables, we will 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 Figure 10.5.

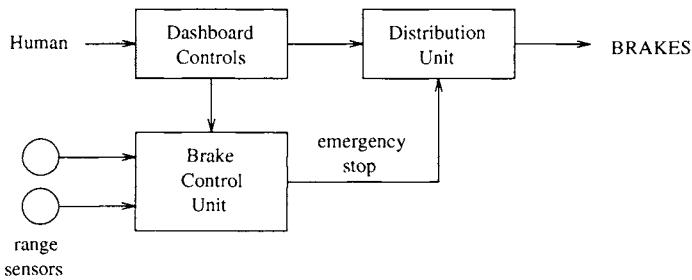


Figure 10.5 Scheme of the automatic braking system.

The Brake Control Unit (BCU) is responsible for acquiring a pair of range sensors, computing the distance of 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 criticalness 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

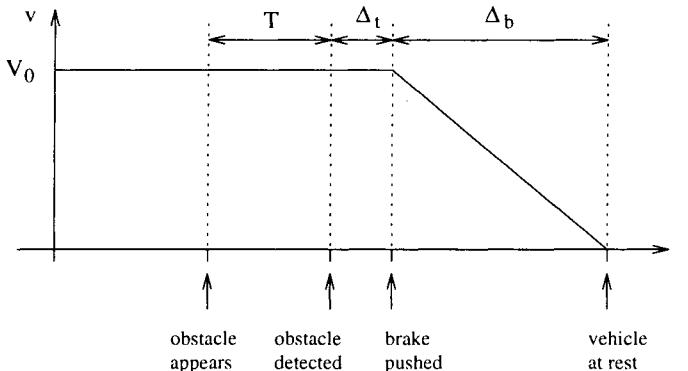


Figure 10.6 Velocity during brake.

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 Figure 10.6, which shows the velocity v of the vehicle as a function of time when an emergency stop is performed.

Notice that three time intervals have to be taken into 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

$$\Delta_b = \frac{v}{\mu_f g},$$

where g is the acceleration of gravity ($g = 9.8m/s^2$). 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:

$$D > \frac{v^2}{2\mu_f g} + v(T + \Delta_t). \quad (10.1)$$

If we assume that obstacles are fixed and are always detected at a distance D from the vehicle, equation (10.1) allows to determine the maximum value that can be assigned to period T :

$$T < \frac{D}{v} - \frac{v}{2\mu_f g} - \Delta_t. \quad (10.2)$$

For example, if $D = 100$ m, $\mu_f = 0.5$, $\Delta_t = 250$ ms, and $v_{max} = 30$ m/s (about 108 km/h), then the resulting sampling period T must be less than 22 ms.

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 $D = 50$ m (visibility under fog conditions), $\mu_f = 0.5$, $\Delta_t = 300$ ms (our typical reaction time), and $v = 60$ km/h (about 16.67 m/s or 37 mi/h), we can look away from the road for no more than one second!

10.2.2 Robot deburring

Consider a robot arm that has to polish an object surface with a grinding tool mounted on its wrist, as shown in Figure 10.7. This task can be specified as follows:

Slide the grinding tool on the object surface with a constant speed v , while exerting a constant normal force F that must not exceed a maximum value equal to F_{max} .

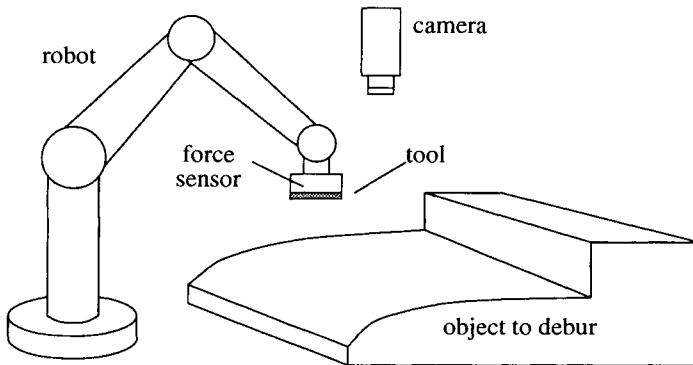


Figure 10.7 Example of a robot deburring workstation.

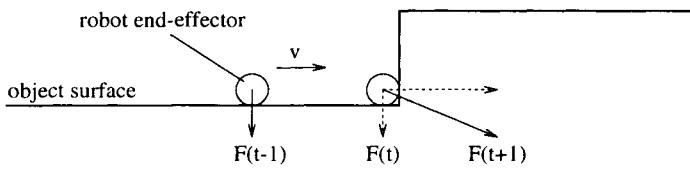


Figure 10.8 Force on the robot tool during deburring.

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 Figure 10.8, if T is the period of the control process and v is the robot horizontal 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.

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 horizontal 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

$$T < \left(\frac{F_{max}}{Kv} - \tau_d \right). \quad (10.3)$$

Notice that, in order to be feasible, the right side of condition (10.3) 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,

$$\frac{F_{max}}{Kv} - \tau_d > Q. \quad (10.4)$$

Equation (10.4) imposes an additional restriction on the application. For example, we may derive the maximum speed of the robot during the deburring operation as

$$v < \frac{F_{max}}{K(Q + \tau_d)}, \quad (10.5)$$

or, if v cannot be arbitrarily reduced, we may fix the tick resolution such that

$$Q \leq \left(\frac{F_{max}}{Kv} - \tau_d \right).$$

Once the feasibility is achieved – that is, condition (10.4) is satisfied – the result expressed in equation (10.3) 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} .

10.2.3 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 Figure 10.9. 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.

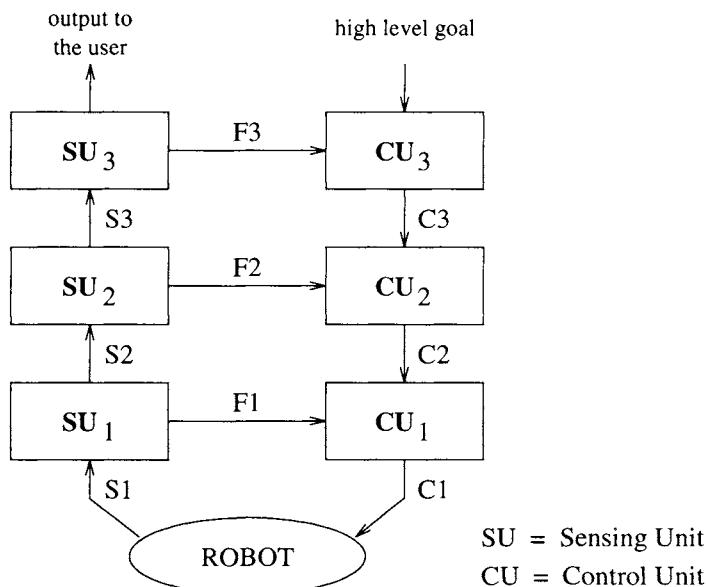


Figure 10.9 Example of a hierarchical control system.

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 nonconsumable 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*.

10.3 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.

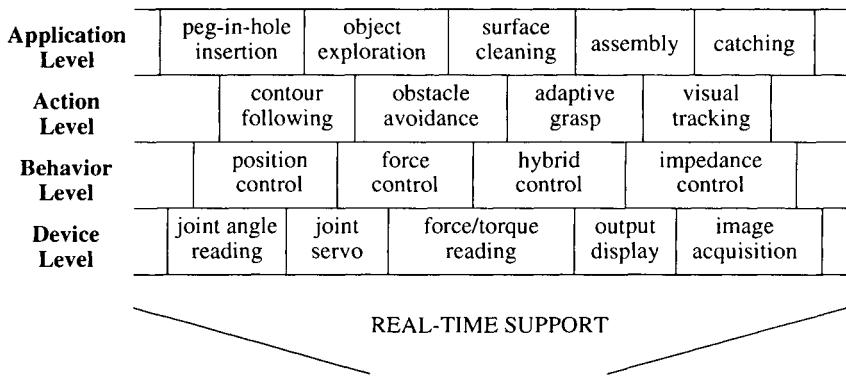


Figure 10.10 Hierarchical software environment for programming complex robotic applications.

Figure 10.10 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 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; (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.

10.3.1 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 precomputed 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.

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.

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.

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 given 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 Section 10.2.2. Other issues involved in robot tactile exploration are discussed in [DB87, Baj88].

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].

10.4 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 Figure 10.11, 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.

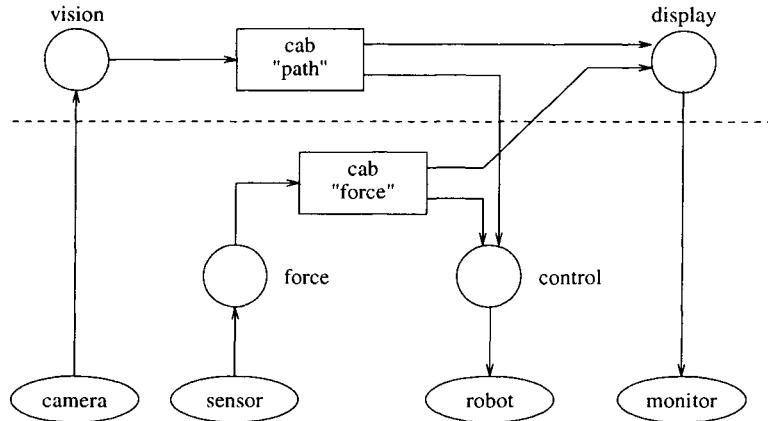


Figure 10.11 Process structure for the surface exploration example.

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 the previous chapter.

```

/*
 *-----*
 * 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 from cabs 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 the 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

EXAMPLES OF REAL-TIME SYSTEMS

11.1 INTRODUCTION

Current operating systems having real-time characteristics can be divided into three main categories:

1. Priority-based kernel for embedded applications,
2. Real-time extensions of timesharing operating systems, and
3. Research operating systems.

The first category includes many commercial kernels (such as VRTX32, pSOS, OS9, VxWorks, Chorus, and so on) that, for many aspects, are optimized versions of timesharing operating systems. In general, the objective of such kernels is to achieve high performance in terms of average response time to external events. As a consequence, the main features that distinguish these kernels are a fast context switch, a small size, efficient interrupt handling, the ability to keep process code and data in main memory, the use of preemptable primitives, and the presence of fast communication mechanisms to send signals and events.

In these systems, time management is realized through a real-time clock, which is used to start computations, generate alarm signals, and check timeouts on system services. Task scheduling is typically based on fixed priorities and does not consider explicit time constraints into account, such periods or deadlines. As a result, in order to handle real-time activities, the programmer has to map a set of timing constraints into a set of fixed priorities.

Interprocess communication and synchronization usually occur by means of binary semaphores, mailboxes, events, and signals. However, mutually exclusive resources are seldom controlled by access protocols that prevent priority inversion; hence, blocking times on critical sections are practically unbounded. Only a few kernels (such as VxWorks) support a priority inheritance protocol and provide a special type of semaphores for this purpose.

The second category of operating systems includes the real-time extensions of commercial timesharing systems. For instance, RT-UNIX and RT-MACH represent the real-time extensions of UNIX and MACH, respectively.

The advantage of this approach mainly consists in the use of standard peripheral devices and interfaces that allow to speed up the development of real-time applications and simplify portability on different hardware platforms. On the other hand, the main disadvantage of such extensions is that their basic kernel mechanisms are not appropriate for handling computations with real-time constraints. For example, the use of fixed priorities can be a serious limitation in applications that require a dynamic creation of tasks; moreover, a single priority can be reductive to represent a task with different attributes, such as importance, deadline, period, periodicity, and so on.

There are other internal characteristics of timesharing operating systems that are inappropriate for supporting the real-time extensions. For example, most internal queues are handled with a FIFO policy, which is often preserved even in the real-time version of the system. In some system, the virtual memory management mechanism does not allow to lock pages in main memory; hence, page-fault handling may introduce large and unbounded delays on process execution. Other delays are introduced by non-preemptable system calls, by synchronous communication channels, and by the interrupt handling mechanism. These features degrade the predictability of the system and prevent any form of guarantee on the application tasks.

The observations above are sufficient to conclude that the real-time extensions of timesharing operating systems can only be used in noncritical real-time applications, where missing timing constraints does not cause serious consequences on the controlled environment.

The lack of commercial operating systems capable of efficiently handling task sets with hard timing constraints, induced researchers to investigate new computational paradigms and new scheduling strategies aimed at guaranteeing a highly predictable timing behavior. The operating systems conceived with such

a novel software technology are called *hard real-time operating systems* and form the third category of systems outlined above.

The main characteristics that distinguish this new generation 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.

Expressive examples of 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 [LTC89], HARTOS [KKS89], YARTOS [JSP92], and HARTIK [But93]. Most of these kernels do not represent a commercial product but are the result of considerable efforts carried out in universities and research centers.

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.

In the rest of this chapter, some of these systems are illustrated to provide a more complete view of the techniques and methodologies that can be adopted to develop a new generation of real-time operating systems with highly predictable behavior.

11.2 MARS

MARS (MAintainable Real-time System) is a fault-tolerant distributed real-time system developed at the University of Vienna [DRSK89, KDK⁺89] to

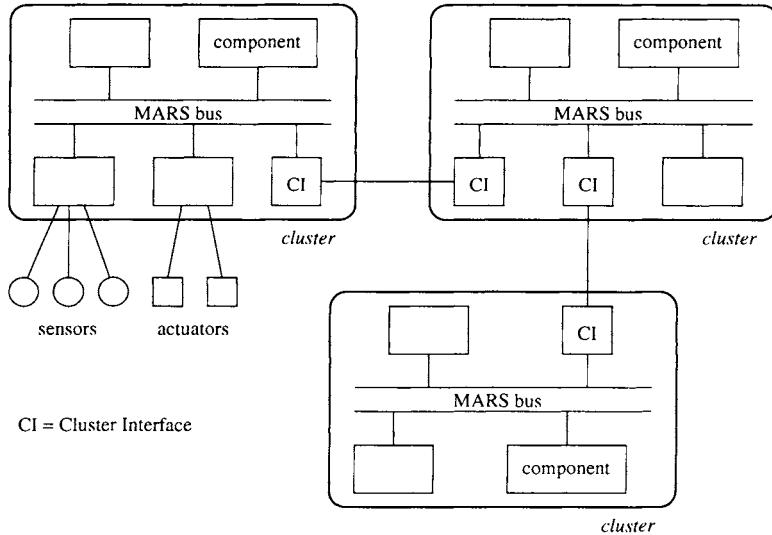


Figure 11.1 The MARS target architecture.

support complex control applications (such as air traffic control systems, railway switching systems, and so on) where hard deadlines are imposed by the controlled environment.

The MARS target architecture consists of a set of computing nodes (*clusters*) connected through high speed communication channels. Each cluster is composed of a number of acquisition and processing units (*components*) interconnected by a synchronous real-time bus, the MARS-bus. Each component is a self-contained computer on which a set of real-time application tasks and an identical copy of the MARS operating system is executed. A typical configuration of the MARS target architecture is outlined in Figure 11.1.

The main feature that distinguishes MARS from other distributed real-time systems is its deterministic behavior even in peak-load conditions; that is, when all possible events occur at their maximum specified frequency. Fault-tolerance is realized at the cluster level through active redundant components, which are grouped in a set of *Fault-Tolerant Units* (FTUs). A high error-detection coverage is achieved by the use of software mechanisms at the kernel level and hardware mechanisms at the processor level.

Within an FTU, a single redundant component fails silently; that is, it either operates correctly or does not produce any results. This feature facilitates system maintainability and extensibility, since redundant components may be removed from a running cluster, repaired, and reintegrated later, without affecting the operation of the cluster. Moreover, a component can be expanded into a new cluster that shows the same I/O behavior. In this way, a new cluster can be designed independently from the rest of the system, as long as the I/O characteristics of the interface component remain unchanged.

Predictability under peak-load situations is achieved by using a static scheduling approach combined with a time-driven dispatching policy. In MARS, the entire schedule is precomputed off-line considering the timing characteristics of the tasks, their cooperation by message exchange, as well as the protocol used to access the bus. The resulting tables produced by the off-line scheduler are then linked to the core image of each component and executed in a time-driven fashion. Dynamic scheduling is avoided by treating all critical activities as periodic tasks.

Although the static approach limits the flexibility of the system in dynamic environments, it is highly predictable and minimizes the runtime overhead for task selection. Moreover, since scheduling decisions are taken off-line, a static approach allows the use of sophisticated algorithms to solve problems (such as jitter control and fault-tolerance requirements) that are more complex than those typically handled in dynamic systems.

All MARS components have access to a common global time base, the system time, with known synchronization accuracy. It is used to test the validity of real-time information, detect timing errors, control the access to the real-time bus, and discard the redundant information.

From the hardware point of view, each MARS component is a slightly modified standard single-board computer, consisting of a Motorola 680x0 CPU, a Local Area Network Controller for Ethernet (LANC), a Clock Synchronization Unit (CSU), two RS-232 serial interfaces, and one Small Computer System Interface (SCSI).

The software residing in a MARS component can be split into the following three classes:

1. **Operating System Kernel.** Its primary goals are resource management (CPU, memory, bus, and so on) and hardware transparency.

2. **Hard Real-Time Tasks** (HRT-tasks). HRT-tasks are periodic activities that receive, process, and send messages. Each instance of a task is characterized by a hard deadline, within which it has to be completed. The set of HRT-tasks consists of application tasks and system tasks, which perform specific functions of the kernel, such as time synchronization and protocol conversions.
3. **Soft Real-Time Tasks** (SRT-tasks). SRT-tasks are activities that are not subject to strict deadlines. Usually, they are aperiodic tasks scheduled in background, during the idle time of the processor.

All hardware details are hidden within the kernel, and all kernel data structures cannot be accessed directly. Both application tasks and system tasks access the kernel only by means of defined system calls. To facilitate porting of MARS to other hardware platforms, most of the operating system code is written in standard C language.

11.2.1 Communication

In MARS, communication among tasks, components, clusters, and peripherals occurs through a uniform message passing mechanism. All messages are sent periodically to exchange information about the state of the environment or about an internal state. State-messages are not consumed when read, so they can be read more than once by an arbitrary number of tasks. Each time a new version of a message is received, the previous version is overwritten, and the state described in the message is updated.

All MARS messages have an identical structure, consisting of a standard header, a constant length, and a standard trailer. Besides the LAN dependent standard fields, the header contains several other fields that include the observation time of the information contained in the message, the validity interval, as well as the send and receive time stamped on the message by the SCU. The trailer basically contains a checksum. The structure of the message body is defined by the application programmer, whereas its size is fixed and predefined in the system.

Since messages describe real-time entities that cannot be altered by tasks, messages are kept in read-only buffers of the operating system. Message exchange between the kernel and the application tasks does not require an explicit copy of the message, but it is performed by passing a pointer. In MARS, process

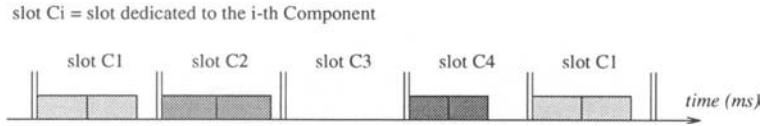


Figure 11.2 Timing of the MARS-bus using the TDMA-protocol with redundant message transmission.

communication is completely asynchronous; hence, there is no need for explicit flow control. If the sender has a frequency higher than that of the receiver, the state is updated faster than read, but no buffer overflow will occur because the latest message replaces the previous one.

Messages among components travel on the MARS-bus, which is an Ethernet link controlled by a TDMA-protocol (Time Division Multiple Access). This protocol provides a collision-free access to the Ethernet even under peak-load conditions. A disadvantage of the TDMA-protocol is a low efficiency under low-load conditions because the sending capacity of a component cannot exceed a fixed limit (approximately equal to the network capacity divided by the number of components in the cluster) even if no other component in the cluster has to send messages. Nevertheless, since MARS has mainly been designed to be predictable even under peak-load conditions, TDMA is the protocol that best satisfies this requirement. As shown in Figure 11.2, each message is sent twice on the MARS-bus.

In order to detect timing errors during communication, each message receives two time stamps from the CSU (when sent and when received), with an accuracy of about three microseconds.

11.2.2 Scheduling

In MARS, the scheduling of hard real-time activities is performed off-line considering the worst-case execution times of tasks, their interaction by message exchange, and the assignment of messages to TDMA slots. The static schedule produced by the off-line scheduler is stored in a table and loaded into each individual component. At runtime, the scheduling table is executed by a dispatcher, which performs task activation and context switches at predefined time instants. The disadvantage of this scheduling approach is that no tasks can be created dynamically, so the system is inflexible and cannot adapt to changes in the environment. On the other hand, if the assumptions on the controlled

environment are valid, the static approach is quite predictable and minimizes the runtime overhead for making scheduling decisions.

Scheduling techniques that increase the flexibility of MARS in dynamic environments have been proposed by Fohler for realizing changes of operational modes [Foh93] and allowing on-line service of aperiodic tasks [Foh95].

The MARS system also allows different scheduling strategies to be adopted in different operating phases. That is, during the design phase, the programmer of the application can define several operational phases of the system characterized by different task sets, each handled by an appropriate scheduling algorithm. For example, for an aircraft control application, five phases can be distinguished: loading, taking off, flying, landing, and unloading. And each phase may require different tasks or a different scheduling policy. The change between two schedules (*mode change*) may be caused either by an explicit system call in an application task or by the reception of a message associated with a scheduling switch.

Two types of scheduling switches are supported by the kernel: a *consistent* switch and an *immediate* switch. When performing a consistent scheduling switch, tasks can only be suspended at opportune instants (determined during the design stage) so that they are guaranteed to remain in a consistent state with the environment. The immediate switch, instead, does not preserve consistency, but it guarantees that switching will be performed as soon as possible; that is, at the next invocation of the major interrupt handler, which has a period of eight milliseconds.

11.2.3 Interrupt handling

In MARS, all interrupts to the CPU are disabled, except for the clock interrupt from the CSU. Allowing each device to interrupt the CPU, in fact, would cause an unpredictable load on the system that could jeopardize the guarantee performed on the hard tasks. A priority scheme for interrupts has also been discarded because it would give advantage to high-priority devices, while low-priority devices might starve for the CPU, causing missed deadlines in consequence. Since interrupts are disabled, peripheral devices are polled periodically within the clock interrupt handler.

The clock interrupt handler is split into two sections activated with different frequencies. The first section (*minor handler*), written in assembler for efficiency

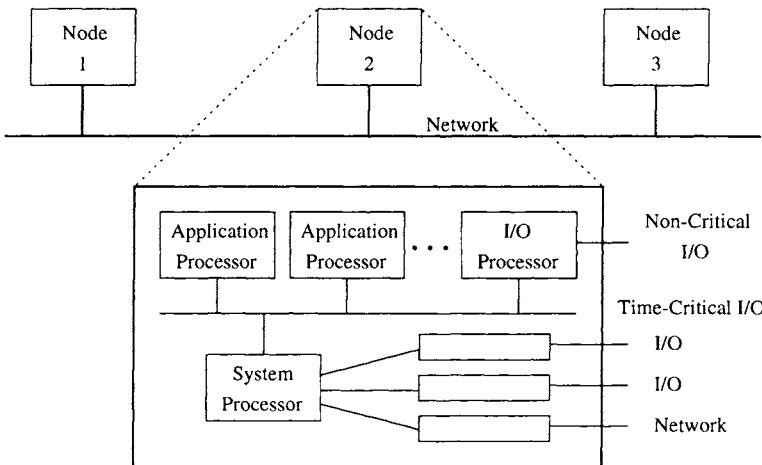


Figure 11.3 The Spring distributed architecture.

reasons, is carried out every millisecond. The second section (*major handler*), written in C, is activated every 8 milliseconds, immediately after the execution of the first part. The minor interrupt handler may suspend any system call, whereas the major handler is delayed until the end of the system call.

11.3 SPRING

Spring is a real-time distributed operating system developed at the University of Massachusetts at Amherst [SR89, SR91] for supporting large complex control applications characterized by hard timing constraints. The Spring target architecture is illustrated in Figure 11.3 and consists of a set of multiprocessor nodes connected through a high speed communication network. Each node contains three types of processors: one or more application processors, a system processor, and an I/O subsystem (front end).

- Application processors are dedicated to the execution of critical application tasks that have been guaranteed by the system.
- The system processor is responsible for executing the scheduling algorithm (a crucial part of the system) and supporting all kernel activities. Such a physical separation between system activities and application activities

allows to reduce the system overhead on the application processors and remove unpredictable delays on tasks' execution.

- The I/O subsystem is responsible for handling non-critical interrupts, coming from slow peripheral devices or from sensors that do not have a predictable response time. Time critical I/O is directly performed on the system processor.

An identical copy of the Spring kernel is executed on each application processor and on the system processor, whereas the I/O processor can be controlled by any commercial priority-based operating system. Within a node, each processing unit consists of a commercial Motorola MVME136A board, plugged in a VME bus. On this board, part of the main memory is local to the processor and is used for storing programs and private data, while another part is shared among the other processors through the VME bus.

Spring allows dynamic task activation, however the assignment of tasks to processors is done statically to improve speed and eliminate unpredictable delays. To increase efficiency at runtime, some tasks can be loaded on more processors, so that, if an overload occurs when a task is activated, the task can be executed on another processor without large overhead.

The scheduling mechanism is divided in four modules:

- At the lowest level, there is a dispatcher running on each application processor. It simply removes the next ready task from a system task table that contains all guaranteed tasks arranged in the proper order. The rest of the scheduling modules are executed on the system processor.
- The second module consists of a local scheduler (resident on the system processor), which is responsible for dynamically guaranteeing the schedulability of a task set on a particular application processor. Such a scheduler produces a system task table that is then passed to the application processor.
- The third scheduling level is a distributed scheduler that tries to find a node available in the case in which a task cannot be locally guaranteed.
- The fourth scheduling module is a metalevel controller that adapts the various parameters of the scheduling algorithm to the different load conditions.

11.3.1 Task management

In Spring, tasks are classified based on two main criteria: importance and timing requirements. The importance of a task is the value gained by the system when the task completes before its deadline. Timing requirements represent the real-time specification of a task and may range over a wide spectrum, including hard or soft deadlines, periodic or aperiodic execution, or no explicit timing constraints.

Based on importance and timing requirements, three types of tasks are defined in Spring: critical tasks, essential tasks, and unessential tasks.

- Critical tasks are those tasks that must absolutely meet their deadlines; otherwise, a catastrophic result might occur on the controlled system. Due to their criticalness, these tasks must have all resources reserved in advance and must be guaranteed off-line. Usually, in real-world applications, the number of critical tasks is relatively small compared to the total number of tasks in the system.
- Essential tasks are those tasks that are necessary to the operation of the system; however, a missed deadline does not cause catastrophic consequences, but only degrades system's performance. The number of essential tasks is typically large in complex control applications; hence, they must be handled dynamically or it would be impossible (or highly expensive) to reserve enough resources for all of them.
- Unessential tasks are processes with or without deadlines that are executed in background; that is, during the idle times of the processor. For this reason, unessential tasks do not affect the execution of critical and essential tasks. Long-range planning tasks and maintenance activities usually belong to this class.

Spring tasks are characterized by a large number of parameters. In particular, for each task, the user has to specify a worst-case execution time, a deadline, an interarrival time, a type (critical, essential, or unessential), a preemptive or non-preemptive property, an importance level, a list of resources needed, a precedence graph, a list of tasks with which the task communicates, and a list of nodes on which the task code has to be loaded. This information is used by the scheduling algorithm to find a feasible schedule.

11.3.2 Scheduling

The objective of the Spring scheduling algorithm is to dynamically guarantee the execution of newly arrived tasks in the context of the current load. The feasibility of the schedule is determined considering many issues, such as timing constraints, precedence relations, mutual exclusion on shared resources, non-preemption properties, and fault-tolerant requirements. Since this problem is NP-hard, the guarantee algorithm uses a heuristic approach to reduce the search space and find a solution in polynomial time. It starts at the root of the search tree (an empty schedule) and tries to find a leaf (a complete schedule) corresponding to a feasible schedule.

On each level of the search, a *heuristic 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 $H = d_i$ (absolute deadline), the algorithm is equivalent to Earliest Deadline First.

To consider resource constraints in the scheduling algorithm, each task τ_i has to declare a binary array of resources $R_i = [R_1(i), \dots, R_r(i)]$, where $R_k(i) = 0$ if τ_i does not use resource R_k , and $R_k(i) = 1$ if τ_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 $T_{est}(i)$ that task τ_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 τ_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

$$\begin{aligned} H &= d + W \cdot C \\ H &= d + W \cdot T_{est}, \end{aligned}$$

where W is a weight that can be adjusted for different application environments.

Precedence constraints can be handled by introducing a new factor E , called *eligibility*. A task becomes eligible to execute only when all its ancestors in the precedence graph are completed. If a task is not eligible, 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, it 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 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. Do to 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.

11.3.3 I/O and interrupt handling

In Spring, peripheral I/O devices are divided in two classes: slow and fast I/O devices. Slow I/O devices are multiplexed through a front-end dedicated processor (I/O processor), controlled by a commercial operating system. Device drivers running on this processor are not subject to the dynamic guarantee algorithm, although they can activate critical or essential tasks. Fast I/O devices are handled by the system processor, so they do not affect the execution of application tasks. Interrupts from fast I/O devices are treated as instantiating a new task that is subject to the guarantee routine just like any other task in the system.

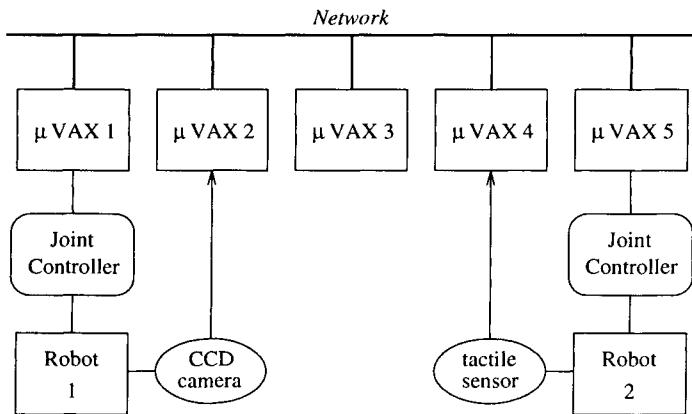


Figure 11.4 Target architecture for RK.

11.4 RK

RK (Real-time Kernel) is a distributed real-time system developed at the University of Pennsylvania [LKP88, LK88] to support multisensor robotic applications. The presence of hard timing constraints in robotic control activities is necessary for two important reasons. First, sensors and actuators require regular acquisition and feedback control in order to achieve continuous and smooth operations. Second, some high-level tasks (such as trajectory planning, obstacle avoidance, and so on) may require timely execution to avoid possible catastrophic results.

The target architecture for which RK has been designed is illustrated in Figure 11.4. It consists of five processors (MicroVAX) connected through a 10 Mb Ethernet, two robot manipulators (PUMA 560) with a joint controller each, a tactile sensor, and a camera. One of the processors (P3) works as a supervisor, two (P1 and P5) are connected to the joint controllers via a parallel interface, one (P2) is responsible for image acquisition and processing, and one (P4) is dedicated to the tactile sensor. In order to support all sensory and control activities needed for this robot system, an identical copy of the kernel is executed on each of the five processors.

To achieve predictable behavior, RK provides a set of services whose worst-case execution time is bounded. In addition, the kernel allows the programmer to specify timing constraints for process execution and interprocess communication.

11.4.1 Scheduling

RK supports both real-time and non-real-time tasks. Real-time tasks are divided in three classes with different level of criticalness: imperative, hard, and soft. The assignment of the CPU to tasks is done according to a priority order. Within the same class, imperative processes are executed on a First-Come-First-Served (FCFS) basis, whereas hard and soft processes are executed based on their timing constraints by the EDF algorithm. The difference between hard and soft tasks is that hard tasks are subject to a guarantee algorithm that verifies their schedulability at creation time, whereas soft tasks are not guaranteed. Finally, non-real-time tasks are scheduled in background using FCFS. Timing constraints on real-time tasks can also be specified as periodic or sporadic and can be defined on the whole process, on a part of a process, and on messages.

To facilitate the programming of timing constraints, RK supports a notion of *temporal scope*, which identifies explicit timing constraints with a sequence of statements. Each temporal scope consists of five attributes: a hard/soft flag, a start time, a maximum execution time, a deadline, and a unique identifier. Whenever a temporal scope with a hard flag is entered, the scheduler checks whether the corresponding timing constraints can be guaranteed in the context of the current load. If the request cannot be guaranteed, an error message is generated by the kernel.

A timing constraint is violated if either a process executes longer than the maximum declared execution time or its deadline is exceeded. When this happens, the kernel sends a signal to the process. If the process is hard, a critical system error has occurred (since the timing constraint was guaranteed by the scheduler); thus, the task that missed the deadline becomes an *imperative* process, and a controlled shutdown of the system is performed as soon as possible.

11.4.2 Communication

RK provides three basic communication methods among real-time tasks:

- Signals, for notification of critical system errors;
- Timed events, for notification of events with timing constraints;
- Ports, for asynchronous message passing with timing constraints.

Signals

Signals are used by the kernel to notify that an error has occurred. The purpose of sending such a signal is to give the process a chance to clean up its state or to perform a controlled shutdown of the system. There are three types of errors: timing errors, process errors, and system errors. Timing errors occur when either a process executes longer than its maximum execution time or its deadline is exceeded. Process errors occur when a task executes an illegal operation – for example, an access to an invalid memory address. System errors are due to the kernel; for example, running out of buffers that have been guaranteed to a task. When the kernel sends a signal to a process, the process executes an appropriate signal handler and then resumes the previous execution flow when the handler is finished.

Timed events

Events are the most basic mechanism for interprocess communication. Unlike a signal, an event can be sent, waited on, delayed, and preempted. In addition, each event can have timing constraints and an integer value, which can be used to pass a small amount of data. For each event, the kernel remembers only the last occurrence of the event. Thus, if an event arrives while another one of the same type is pending, only the value of the last one is remembered.

Like signals, whenever a process receives an event, it executes an associated event handler; the previous execution flow resumes once the handler is finished. There are two ways to associate timing constraints with events. According to the first way, the receiver of an event may specify a timeout for executing the event handler. Alternatively, the sender may include a deadline when the event is sent. If both the sender and the receiver specify timing constraints for the same event, then the earliest deadline is used for the execution of the handler.

If a non-real-time process receives a timed event, the corresponding event handler is executed immediately, and, during the handling of the event, the process is treated as real-time. This feature allows non-real-time server processes to handle requests from real-time processes.

Ports

The port construct is widely used in operating systems for interprocess communication. In RK, it is extended for real-time communication by allowing

the sender to specify timing constraints in messages and the receiver to control message queueing and reception strategies. Sending a message to a port is always nonblocking, and the execution time for a transmission is bounded to ensure a predictable delay. For critical message communication, the sender can include a set of timing attributes within each message, such as the start time, the maximum duration and the deadline. Receiving a message can be either explicit or asynchronous. When using an explicit receive primitive, the process can specify a timeout to limit the delay in waiting for a message. For asynchronous receive, the receiver associates a timed event with a port and each message arrival is notified through the timed event.

Every RK process is created with a default reception port, used during initialization and to request services from system server processes. Additional ports can be created using the following system call:

```
port_id = port_create(type).
```

The argument *type* specifies whether the port is for receiving messages or for multicasting messages. For a reception port, any process can send a message to it, but only the creator can receive from it. A multicast port realizes a *one-to-many* communication channel. Each multicast port has a list of destination ports to which messages are to be forwarded. When a message is sent to a multicast port, it is forwarded to all ports connected to it, and this forwarding is repeated until the message reaches a reception port.

When creating a reception port, various attributes can be specified by the creator. They allow the following characteristics to be defined:

- **The ordering of messages within the port queue.** It can be done either by transmission time, arrival time, or deadline.
- **The size of the queue** – that is, the maximum number of messages that can be stored in the queue. In case of overflow, it is possible to specify whether messages are thrown away at the head or at the tail of the queue.
- **Communication semantics.** Normally, messages are removed from the queue when they are received. However, when the *stick* attribute is set, a message remains in the queue even after it is received, and it is replaced only when a new message arrives.

In RK, the *send* and *receive* system calls have the following syntax:

```
send(portid, reply_portid, t_record, msg, size);  
receive(portid, reply_portid, timeout, t_record, msg, size);
```

where *portid* is the identifier of the port; *reply_portid* specifies where to send a reply; *timeout* (only in reception) specifies the maximum amount of time that the primitive should block waiting for a message; *t_record* is a pointer to a record containing the three timing attributes (start time, maximum duration, and deadline) specified by the sender; *msg* is a pointer to the message; and *size* is the size of the message.

11.4.3 I/O and interrupt handling

Traditional operating systems provide device drivers that simplify the interactions between application processes and peripheral devices. This approach allows the same device to be used by many processes; however, it introduces additional delays during processes's execution that may jeopardize the guarantee of hard real-time activities.

In robotics applications, this problem is not so relevant, since sensory devices are not shared among processes but are controlled by dedicated tasks that collect data and preprocess them. For this reason, RK allows processes to directly control devices by sharing memory and accessing device registers. In addition, a process may request the kernel to convert device interrupts into timed events.

Although this approach requires the programmer to know low-level details about devices, it is faster than the traditional method, since no context switching is needed to apply feedback to a device. Furthermore, the kernel needs not to be changed when removing or adding new devices.

11.5 ARTS

ARTS (Advanced Real-time Technology System) is a distributed real-time operating system developed at the Carnegie Mellon University [TK88, TM89] for verifying advanced computing technologies for a distributed environment. The target architecture for which ARTS has been developed consists of a set of SUN workstations connected by a real-time network based on IEEE 802.5 To-

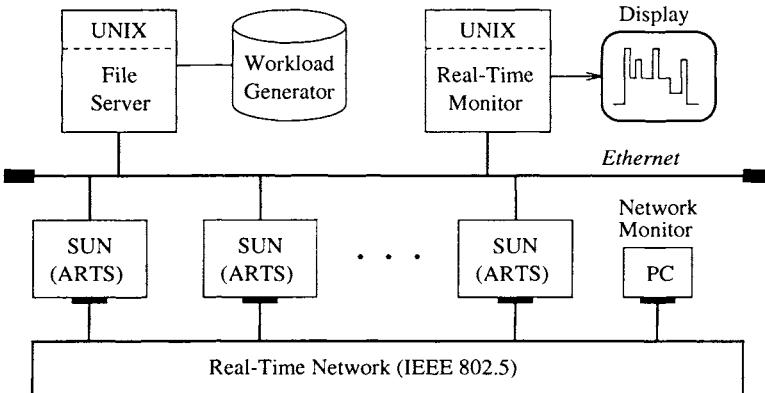


Figure 11.5 The ARTS target architecture.

ken Ring. Figure 11.5 shows the typical configuration of the system and the relation between the kernel and its real-time tools.

The programming environment provided by the ARTS system is based on an object-oriented paradigm, in which every computational entity is represented by an object. Objects can be defined as real-time or non-real-time objects. Each operation associated with a real-time object has a worst-case execution time, called a *time fence*, and a time exception handling routine. In addition, an ARTS object can be *passive* or *active*. Active objects are characterized by the presence of one or more internal threads (defined by the user) that accept incoming invocation requests.

All threads are implemented as lightweight processes that share the same address space. A thread can be defined as a periodic or aperiodic task depending on its timing attributes. The timing attributes of a thread consist of a value function, a worst-case execution time, a period, a phase, and a delay value.

ARTS supports the creation and destruction of objects at a local node, as well as at a remote node. Although process migration is a very important mechanism in non-real-time distributed operating systems, the ARTS kernel does not support object migration during runtime. Instead, it can move an object by shutting down the activities and reinitiating the object at the target host with appropriate parameters.

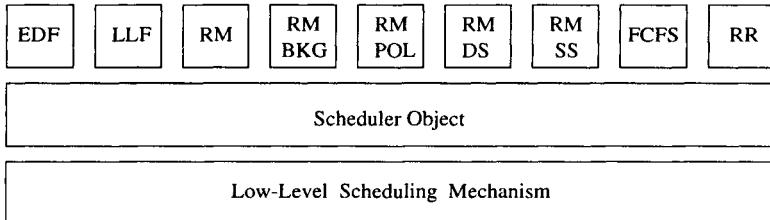


Figure 11.6 Structure of the ARTS scheduler.

11.5.1 Scheduling

In ARTS, the scheduling policy is implemented as a self-contained kernel object and is separated from the thread handling mechanism, which performs only dispatching and blocking.

For experimental purposes, several scheduling policies have been implemented in the ARTS kernel, including static algorithms such as Rate Monotonic (RM) and dynamic algorithms such as Earliest Deadline First (EDF) and Least Laxity First (LLF). In conjunction with Rate Monotonic, a number of strategies for handling aperiodic threads have been realized, such as Background servicing (BKG), Polling (POL), Deferrable Server (DS), and Sporadic Server (SS). More common scheduling algorithms such as First Come First Served (FCFS) and Round Robin (RR) have also been realized for comparison with real-time scheduling policies. A scheduling policy object can be selected either during system initialization or during runtime. Figure 11.6 shows the general structure of the ARTS scheduler.

A schedulability analyzer associated with each scheduling algorithm allows the following to be guaranteed:

- The feasibility of hard tasks within their deadlines,
- A high cumulative value for soft tasks, and
- Overload control based on the value functions of aperiodic tasks.

When selecting a server mechanism for handling aperiodic tasks, the server parameters (period and capacity) are set to fully utilize the processor. This allows to reserve the maximum CPU time for aperiodic service while guaranteeing the schedulability of periodic hard tasks.

11.5.2 Communication

In traditional real-time operating systems, interprocess communication mechanisms are realized to be fast and efficient (that is, characterized by a low overhead). In ARTS, however, the main goal has been to realize a communication mechanism characterized by a predictable and analyzable behavior. To achieve this goal, ARTS system calls require detailed information about communication patterns among objects, including the specification of periodic message traffic and rates for aperiodic traffic.

In ARTS, every message communication is caused by an invocation of a target object's operation, and the actual message communication is performed in a Request-Accept-Reply fashion. Unlike traditional message passing paradigms, the caller must specify the destination object, the identifier of the requested operation, the pointer to the message, and the pointer to a buffer area for the reply message.

To avoid priority inversion among objects inside each node, message transmission is integrated with a Priority Inheritance mechanism, which allows to propagate priority information across object invocations. All network messages are handled by a Communication Manager (CM), where different protocols are implemented using a state table specification. The CM prevents priority inversion over the network by using priority queues with priority inheritance. Thus, if a low-priority message is processed when a higher-priority message arrives, the low-priority message will execute at the highest priority. In this way, the highest-priority message remains in the queue for at most the time it takes to process one message.

11.5.3 Supporting tools

ARTS provides a set of supporting tools, the ARTS Tool-Set [TK88], aimed at reducing the complexity of application development in a distributed real-time environment. This tool-set includes a schedulability analyzer, a support for debugging, and a system monitoring tool.

Schedulability analyzer

The main objective of this tool is to verify the schedulability of a given set of hard real-time tasks under a particular scheduling algorithm. The performance of soft aperiodic tasks are computed under specific service mechanisms, such as

Background, Polling, Deferrable Server, and Sporadic Server. An interactive graphical user interface is provided on a window system to quickly select the scheduling algorithm and the task set to be analyzed. To confirm the schedulability of the given task set in a practical environment, this tool also includes a synthetic workload generator, which creates a particular sequence of requests based on a workload table specified by the user. The synthetic task set can then be executed by a scheduling simulator to test the observance of the hard timing constraints.

Debugging

The ARTS system provides the programmer with a set of basic primitives that can be used for building a debugger and for monitoring process variables. For example, the *Thread_Freeze* primitive halts a specific thread for inspection, while the *Object_Freeze* primitive stops the execution of an ARTS object (that is, all its associated threads). *Thread_Unfreeze* and *Object_Unfreeze* primitives resume a suspended thread and object, respectively. While a thread is in a *frozen* state, the *Fetch* primitive allows to inspect its status in terms of a set of values of data objects. The value of any data object can be replaced using the *Store* primitive. Finally, the *Thread_Capture* and *Object_Capture* primitives allow to capture on-going communication messages from a specified thread and object, respectively.

System monitoring

ARTS includes a monitoring tool, called *Advance Real-time Monitor* (ARM), whose objective is to observe and visualize the system's runtime behavior. Typical events that can be visualized by this tool are context switches among tasks caused by scheduling decisions. ARM is divided into three functional units: the *Event Tap*, the *Reporter*, and the *Visualizer*. The Event Tap is a probe embedded inside the kernel to pick up the raw data on interesting events. The Reporter is in charge of sending the raw data to the Visualizer on a remote host, which analyzes the events and visualizes them in an interactive graphical environment. The Visualizer is designed to be easily ported to different graphical interfaces.

11.6 HARTIK

HARTIK (HARD Real-TIme Kernel) is a hard real-time operating environment developed at the Scuola Superiore S. Anna of Pisa [BDN93, But93] to support advanced robot control applications characterized by stringent timing constraints.

Complex robot systems are usually equipped with different types of sensors and actuators and hence require the concurrent execution of computational activities characterized by different types of timing constraints. For example, processing activities related to sensory acquisition and low-level servoing must be periodically executed with regular activation rates to ensure a correct reconstruction of external signals and guarantee a smooth and stable behavior of the robot system. Other activities (such as planning special actions, modifying the control parameters, or handling exceptional situations) are intrinsically aperiodic and are triggered when some particular condition occurs. To achieve a predictable timing behavior and to satisfy system stability requirements, most acquisition and control tasks require stringent timing constraints, that have to be met in all anticipated workload conditions. In addition, complex robot systems are typically built using disparate peripheral devices that may be distributed on heterogeneous computers.

For the reasons mentioned above, HARTIK has been designed to support the following major characteristics:

- **Flexibility.** It is possible to schedule hybrid task sets consisting of periodic and aperiodic tasks with different level of criticalness.
- **Portability.** The kernel has been designed in a modular fashion, and all hardware-dependent code is encapsulated in a small layer that provides a virtual machine environment.
- **Dynamic preemptive scheduling and on-line guarantee.** Any hard task is subject to a feasibility test. If a task cannot be guaranteed, the system raises an exception that allows to take an alternative action.
- **Efficient aperiodic service.** An integrated scheduling algorithm enhances responsiveness of soft aperiodic requests without jeopardizing the guarantee of the hard tasks.
- **Predictable resource sharing.** Special semaphores allow to bound the maximum blocking time on critical sections, preventing deadlock and chained blocking.

- **Fully asynchronous communication.** A particular nonblocking mechanism, called CAB, is provided for exchanging messages among periodic tasks with different periods, thus allowing the implementation of multilevel feedback control loops.
- **Efficient and predictable interrupt handling mechanism.** Any interrupt request can either be served immediately, or cause the activation of an handler task, which is guaranteed and scheduled as any other hard task in the system.

To facilitate the development of real-time control applications on heterogeneous architectures, HARTIK has been designed to be easily ported on different hardware platforms. At present, the kernel is available for Motorola MC 680x0 boards with VME bus, Intel 80x86 and Pentium with ISA/PCI bus, and DEC AXP-Alpha stations with PCI bus.

Figure 11.7 illustrates a possible architecture that can be used to build a control application. In this solution, control algorithms, trajectory planning, and feedback loops are executed on a Pentium-based computer; sensory acquisition and data preprocessing are executed on a Motorola 68030 processor; whereas the application development is carried out on a DEC Alpha workstation. In this node, a set of tools is available for designing the application structure, estimating the maximum execution time of the tasks, analyzing the schedulability of the task set, and monitoring the system activity.

11.6.1 Task management and scheduling

HARTIK distinguishes three classes of tasks with different criticalness:

- **HARD tasks.** They are periodic or aperiodic processes with critical deadline that are guaranteed by the kernel at creation time. Moreover, the system performs a runtime check on hard deadlines, notifying a time overflow when a hard deadline is missed.
- **SOFT tasks.** They are periodic or aperiodic processes with non-critical deadline that are not guaranteed by the system. Soft tasks are handled by the Total Bandwidth Server[SB94, SB96], which enhances their response time without jeopardizing the guarantee of hard tasks.

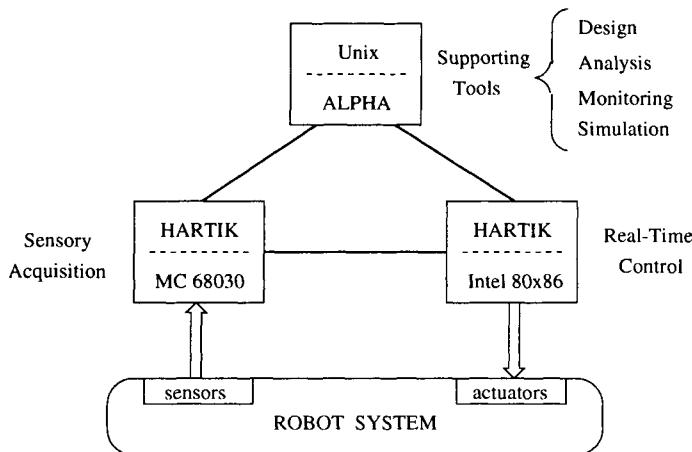


Figure 11.7 Example of heterogeneous architecture that can be used with HARTIK.

- **NRT tasks.** They are Non-Real-Time aperiodic processes with no timing constraints. NRT tasks are scheduled in background and are characterized by a static priority level assigned by the user.

When a task is created, several parameters have to be specified, such as its name, its class (HARD, SOFT, or NRT), its type (periodic or aperiodic), a relative deadline, a period (or a minimum interarrival time for sporadic tasks), a worst-case execution time, a pointer to a list of resources handled by the Stack Resource Policy, and a maximum blocking time. Hard and soft tasks are scheduled according to the Earliest-Deadline-First scheduling policy, which is optimal and achieves full processor utilization.

Real-time tasks can share resources in a predictable fashion through the *Stack Resource Policy (SRP)*. The SRP ensures that, once started, a task will never block until completion but can be preempted only by higher-priority tasks. Furthermore, the SRP avoids priority inversion, chained blocking, deadlock, and reduces the number of context switches due to resource acquisition. Using SRP, the maximum blocking time that any task can experience is equal to the duration of the longest critical section, among those that can block it.

11.6.2 Process communication

HARTIK provides both synchronous and asynchronous communication primitives to adapt to different task requirements. For synchronous communication, tasks can use two types of ports: RECEIVE and BROADCAST.

A RECEIVE port is a channel where many tasks can send messages to, but only one, the owner, is allowed to receive them. Sending messages to and receiving messages from a receive port is always synchronous with timeout. Hence, these ports can be used by soft and NRT tasks and by those hard tasks that must absolutely perform synchronous communication.

BROADCAST ports provide a one-to-many communication channel. They have not only some buffering capability for incoming messages but also a list of destination ports to which messages are to be forwarded. When a message is sent to a broadcast port, it is redirected to all ports specified in the list. BROADCAST ports allow asynchronous send, but they are not directly addressable by a receive. These ports are suited for soft and non-real-time tasks.

A third type of port available in the kernel is the STICK port, which is a *one-to-many* communication channel with asynchronous semantics. When a process receives a message from a STICK port, the port does not consume the message but leaves it stuck until it is overwritten by another incoming message. As a consequence, a process is never blocked for an empty or full buffer. For this property, the use of STICK ports is strongly recommended for exchanging state information among HARD tasks.

Asynchronous communication is supported by the *Cyclic Asynchronous Buffer* (CAB) mechanism, purposely designed for the cooperation among periodic activities with different activation rate, such as sensory acquisition and control loops. A CAB provides a one-to-many communication channel which contains, at any instant, the latest message inserted in its structure.

A message is not consumed by a receiving task, but it is maintained into the CAB until a new message is overwritten. In this way, a receiving task will always find data in a CAB, so that unpredictable delays due to synchronization can be eliminated. It is important to point out that CABs do not use semaphores to protect their internal data structures, so they are not subject to priority inversion.

CAB messages are always accessed through a pointer, so that the overhead of CAB primitives is small and independent of the message size. The kernel also allows tasks to perform simultaneous read and write operations to a CAB. This is achieved through the use of multiple memory buffers. For example, if a task wants to write a new message in a CAB that is being used by another task (which is reading the current message), a new buffer is assigned to the writer, so that no memory conflict occurs. As the writing operation is completed, the written message becomes the most recent information in that CAB, and it will be available to any other task. The maximum number of buffers needed for a CAB to avoid blocking must be equal to the number of tasks that share the CAB plus one.

11.6.3 Interrupt handling

In HARTIK, a device driver is split into two parts: a *fast handler* and a *safe handler*. When an interrupt is triggered by an I/O device, the fast handler is executed in the context of the currently running task to avoid the overhead due to a context switch. It typically performs some basic input/output operations and acknowledges the peripheral. Then, the kernel automatically activates the safe handler, which is subject to the scheduling algorithm as any other aperiodic task in the system. The safe handler can be declared as a soft or sporadic task depending on the characteristics of the device. It is in charge of doing any remaining computation on the device – for example, data multiplexing among user tasks. This approach is quite flexible, since it allows to nicely combine two different service techniques: the event-driven approach (obtained by the fast handler) and the time-driven approach (obtained by the safe handler).

11.6.4 Programming tools

The HARTIK system includes a set of tools [ABDNB96, ABDNS96] to assist the development of time-critical applications from the design stage to the monitoring phase. In particular, the tool set includes a design tool to describe the structure of the application, a schedulability analyzer to verify the feasibility of critical tasks, a scheduling simulator to test the performance of the system under a synthetic workload, a worst-case execution time estimator, and a tracer to monitor and visualize the actual evolution of the application.

Design tool

The design tool includes an interactive graphics environment that allows the user to describe the application requirements according to three hierarchical levels. At the highest level, the application is described as a number of virtual nodes that communicate through channels. Virtual nodes and channels are graphically represented by icons linked with arrows. Opening the icon of a virtual node we reach the second hierarchical level. At this stage, the developer specifies the set of concurrent tasks running in the virtual node and communicating through shared critical sections or through channels. Tasks, shared resources, and channels are graphically represented by icons that the developer can move and link with arrows. Any possible object (a task, a resource, a channel, or a message) is an instance of a class for that type of object.

Scheduling analyzer

The Schedulability Analyzer Tool (SAT) is very useful for designing predictable real-time applications because it enables the developer to analyze a set of critical tasks and statically verify their schedulability. If the schedulability analysis gives a negative result, the user can change the task parameters and rerun the guarantee test. For instance, some adjustments are possible by rearranging the task deadlines or by producing a more compact and efficient code for some critical tasks or even changing the target machine.

Scheduling simulator

Many practical real-time applications do not contain critical activities but only tasks with soft time constraints, where a deadline miss does not cause any serious damage. In these applications, the user may be interested in evaluating the performance of the system in the average-case behavior rather than in the worst-case behavior. In order to do that, a statistical analysis through a graphic simulation is required. For this purpose, the tool kit includes a scheduling simulator and a load generator for creating random aperiodic activities. Actual computation times, arrival times, durations, and positions of critical sections in the tasks are computed by the load generator as random variables, whose distribution is provided by the user.

Maximum execution time evaluator

The execution time of tasks it is estimated by a proper tool, which performs a static analysis of the application code, supported by a programming style and specific language constructs to get analyzable programs. The language used to develop time-bounded code is an extension of the C language, where monitors are added to isolate and evaluate the duration of critical sections. Optional bounds are programmable to limit the number of iterations in loop statements or to limit the maximum number of processing conditional branches inside loops. The present implementation has models of Intel i386 and i486 CPUs, but the tool can be easily adapted to different kind of processors.

The model includes the simulation of the processor in a table-driven fashion, where assembly instructions are translated into execution times depending on their operating code, operands, and addressing mode. The tool works in conjunction with the C compiler and produces a graph representation of the program's control structure in terms of temporal behavior, where a weight is assigned to every branch of the graph, corresponding to the number of CPU cycles needed for the execution of a segment of sequential code. With this representation, calculating the worst-case behavior of an algorithm means evaluating the maximum cost path in the graph.

Real-time tracer

This tool allows the monitoring of the system evolution while an application is running. It consists of four main parts: a probe, a data structure in the kernel, an event recorder, and a visualizer. The probe is a kernel routine inserted in the system calls, capable of keeping track of all events occurring in the system. At each context switch, the probe saves in main memory the system time (with a microsecond resolution) at which the event takes place, the name of the recorded primitive, the process identifier, its current deadline, and its state before the primitive execution. At system termination, the recorder saves the application trace in a file, which can be later interpreted and displayed by the visualizer. This tool produces a graphics representation of the system evolution in a desired time scale, under Windows NT/95.

The user has the possibility of moving along the trace, changing the scale factor (zoom), and displaying information about task properties, such as type, periodicity class, deadline, and period. Statistical information on waiting times into the various queues are also calculated and displayed both in graphical and textual fashion. On-line help is also provided.

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.

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 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.

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 off-line, 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 set of periodic tasks.

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

- [ABDNB96] P. Ancilotti, G. C. Buttazzo, M. Di Natale, and M. Bizzarri. A flexible tool kit for the development of real-time applications. In *Proceedings of the IEEE Real-time Technology and Application Symposium*, pages 260–262, June 1996.
- [ABDNS96] P. Ancilotti, G. C. Buttazzo, M. Di Natale, and M. Spuri. A development environment for real-time applications. *Journal of Software Engineering and Knowledge Engineering*, 6(3):91–99, September 1996.
- [ABR⁺93] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
- [ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: the deadline-monotonic approach. In *Proceedings of Eighth IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.
- [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.
- [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.
- [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.

- [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.
- [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.
- [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.
- [BL97] G. Buttazzo and G. Lipari. Scheduling analysis of hybrid real-time task sets. In *Proceedings of the IEEE Euromicro Workshop on Real-Time Systems*, 1997.
- [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.
- [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.

- [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.
- [BS97] G. Buttazzo and F. Sensini. Deadline assignment methods for soft aperiodic scheduling in hard real-time systems. In *Submitted to IEEE Euromicro Workshop on Real-Time Systems*, 1997.
- [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.
- [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.
- [CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10), 1989.
- [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.

- [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.
- [Der74] M.L. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 74, 1974.
- [Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genys, 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.
- [Foh93] G. Fohler. Realizing changes of operational modes with pre run-time scheduled hard real-time systems. In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems*, pages 287–300. Springer-Verlag, 1993.
- [Foh95] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 152–161, December 1995.
- [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.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: a survey. *Annals of Discrete Mathematics*, 5, 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.
- [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.
- [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.
- [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.
- [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.
- [L⁺94] J.W.S. Liu et al. Imprecise computations. In *Proceedings of the IEEE*, January 1994.
- [Law73] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19, 1973.
- [Lip97] G. Lipari. Resource constraints among periodic and aperiodic tasks. RETIS LAB, TR-97 01, Scuola Superiore S. Anna, Pisa, Italy, February 1997.
- [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.
- [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, J. Chung, and W. Zhao. Algorithms for scheduling imprecise calculations. *IEEE Computer*, 24(5), 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.
- [LRK77] J.K. Lenstra and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, (5):287–326, 1977.
- [LRKB77] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, (1):343–362, 1977.
- [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.
- [LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.
- [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.W. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.
- [Nat95] Swaminathan Natarajan, editor. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

- [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.
- [Rea86] J. Ready. Vrtx: A real-time operating system for embedded microprocessor applications. *IEEE Micro*, August 1986.
- [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.
- [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.
- [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.
- [Sha85] S. Shani. *Concepts in Discrete Mathematics*. Camelot Publishing Company, 1985.
- [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.
- [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.
- [SR89] J. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time operating systems. *Operating System Review*, 23(3), July 1989.
- [SR90] J.A. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Journal of Real-Time Systems*, 2, 1990.
- [SR91] J.A. Stankovic and K. Ramamritham. The spring kernel: a new paradigm for real-time systems. *IEEE Software*, 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.
- [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.
- [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.
- [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.
- [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

A

Absolute Finishing Jitter, 80
Absolute Release Jitter, 79
Accidents, 2
Actuators, 304
Ada language, 19
Adversary argument, 235
Aperiodic service
 Background scheduling, 110
 Deferrable Server, 116
 Dynamic Priority Exchange, 150
 Dynamic Sporadic Server, 155
 EDL server, 163
 IPE server, 168
 Polling Server, 112
 Priority Exchange, 125
 Slack Stealer, 138
 Sporadic Server, 132
 TB* server, 171
 Total Bandwidth Server, 160
Aperiodic task, 27, 51
Applications, 1, 301
Arrival time, 26
ARTS, 325, 340
Assembly language, 2
Asynchronous communication, 290
Audsley, 92, 98
Autonomous system, 307
Average response time, 7–8, 11

B

Background scheduling, 110

Baker, 208
Baruah, 102, 228, 234, 241
Best-effort, 38
Biyabani, 227
Blocking factor, 194
Blocking, 181
Bouchentouf, 71
Braking control system, 307
Bratley, 64
Burns, 140
Busy period, 103
Busy wait, 16–18
Buttazzo, 150, 160, 229, 245

C

CAB, 291
Cache, 13
Carey, 227
Ceiling blocking, 204
Ceiling, 201
Chained blocking, 199
CHAOS, 325
Chen, 185
Chetto, 71, 163
Chorus, 323
Clairvoyant scheduler, 35
Clairvoyant scheduling, 234
Clark, 291
Communication channel, 291
Competitive factor, 234
Complete schedule, 63
Completion time, 27
Computation time, 27

Concurrency control protocols, 221
 Context switch, 24, 255
 Control applications, 301
 Control loops, 301
 Cost function, 40
 Critical instant, 79
 Critical section, 31, 181
 Critical time zone, 79
 Criticalness, 27
 Cumulative value, 43, 231
 Cyclical Asynchronous Buffers, 291

D

D-over algorithm, 248
 D-over, 248
 Dashboard, 307
 Davis, 140
 Deadline Monotonic, 96
 Deadline tolerance, 245
 Deadline, 8, 27
 firm, 231
 hard, 8
 soft, 8
 Deadlock prevention, 201, 205, 216
 Deadlock, 200
 Deferrable Server, 116
 Dertouzos, 57
 DICK, 253, 260
 Ding, 92
 Direct blocking, 188
 Directed acyclic graph, 28
 Dispatching, 23, 271
 DMA, 13
 cycle stealing, 13
 timeslice, 13
 Domino effect, 37, 225
 Driver, 15
 Dynamic Priority Exchange, 150
 Dynamic priority servers, 149
 Dynamic Priority Exchange, 150

Dynamic Sporadic Server, 155
 EDL server, 163
 IPE server, 168
 TB* server, 171
 Total Bandwidth Server, 160
 Dynamic scheduling, 35
 Dynamic Sporadic Server, 155

E

Earliest Deadline First, 56, 93
 Earliest Due Date, 53
 EDL server, 163
 Eligibility, 335
 Empty schedule, 63
 Environment, 302
 Ethernet, 329, 336
 Event, 6, 17
 Event-driven scheduling, 109
 Exceeding time, 27, 246
 Exclusive resource, 31, 181
 Execution time, 27
 Exhaustive search, 63
 Exponential time algorithm, 34

F

Fault tolerance, 12, 326–327
 Feasible schedule, 25
 Feedback, 304
 Finishing time, 27
 Firm task, 109, 231
 First Come First Served, 111
 Fixed-priority servers, 110
 Deferrable Server, 116
 Polling Server, 112
 Priority Exchange, 125
 Slack Stealer, 138
 Fohler, 330
 Friction, 308

G

Graceful degradation, 230, 245
Graham, 44
Graham's notation, 51
Guarantee mechanism, 36
Guarantee, 243
Gulf War, 3

H

Hard real-time system, 8
Hard task, 8, 26
Haritsa, 227
HARTIK, 291, 325, 345
HARTOS, 325
Heuristic function, 66, 334
Heuristic scheduling, 35
Hierarchical design, 313
Hit Value Ratio, 249
Horn's algorithm, 56
Howell, 102
Hybrid task sets, 109
Hyperperiod, 103

I

Idle state, 256
Idle time, 24
Imprecise computation, 38
Instance, 27
Interarrival time, 109
Interference, 98–99, 172
Interrupt handling, 15
Intertask communication, 289
IPE server, 168

J

Jackson's rule, 53
Jeffay, 62, 102, 299

Jitter, 79
Job, 27

K

Karp, 242
Kernel primitive
 activate, 280
 create, 260
 end_cycle, 281
 end_process, 283
 kill, 283
 sleep, 260
Kernel, 253
Koren, 248

L

Language, 12, 19
Lateness, 27
Latest Deadline First, 68
Lawler, 68
Laxity, 27
Layland, 82
Lehoczky, 92, 116, 125, 138, 186,
 201
Leung, 96
Lifetime, 265
Lin, 185
List management, 272
Liu, 82, 142
Livny, 227
Load, 228
Locke, 227

M

Mach, 227
MACH, 324
Mailbox, 290
Maintainability, 12

MARS, 325
 Martel, 62
 MARUTI, 325
 Maximum lateness, 40
 Memory management, 19
 Message passing, 289
 Message, 290
 Metrics, 41, 230
 Mode change, 330
 Motorola, 327
 Multimedia, 38
 Murphy's Laws, 4
 Mutual exclusion, 18, 31, 181, 284

N

Nested critical section, 187
 Non-idle scheduling, 62
 Non-preemptive scheduling, 62
 Non-real-time task, 109
 NP-complete, 34
 NP-hard, 34

O

Off-line scheduling, 35
 On-line guarantee, 36
 On-line scheduling, 35
 Optimal scheduling, 35
 OS9, 323
 Overhead, 296
 Overload, 225

P

Partial schedule, 63
 Patriot missiles, 3
 Peak load, 12
 Performance, 40, 43, 230
 Period, 28, 78
 Periodic task, 27, 77

Phase, 28, 78
 Polling Server, 112
 Polynomial algorithm, 34
 Precedence constraints, 28, 68
 Precedence graph, 28
 Predecessor, 28
 Predictability, 12
 Preemption level, 209
 Preemption, 24
 Preemptive scheduling, 35
 Priority Ceiling Protocol, 201
 Priority Exchange Server, 125
 Priority Inheritance Protocol, 186
 Priority inversion, 184
 Process, 23
 Processor demand, 102
 Processor utilization factor, 80
 Programming language, 12, 19
 Pruning, 64
 PSOS, 323
 PUMA, 336
 Push-through blocking, 189

Q

Quality of service, 38
 Queue operations
 extract, 274
 insert, 272
 Queue, 24
 idle, 256
 ready, 24, 256
 wait, 32, 182, 256

R

Rajkumar, 186, 201
 Ramamritham, 65, 226
 Ramos-Thuel, 138
 Rate Monotonic, 82
 Ready queue, 24, 256

- Real Time, 4
Receive operation, 290
Reclaiming mechanism, 154, 244
Recovery strategy, 247
Recursion, 20
RED algorithm, 245
Relative Finishing Jitter, 80
Relative Release Jitter, 79
Release time, 77
Residual laxity, 245
Resource access protocol, 181
Resource constraints, 31, 181, 186
Resource reclaiming, 154, 245, 247
Resource, 31, 181
Response time, 79
Richard's anomalies, 44
RK, 325, 336
Robot assembly, 315
Robotic applications, 301
Robust scheduling, 243
Rosier, 102
RT-MACH, 324
RT-UNIX, 324
Running state, 23
-
- S**
- Schedulable task set, 25
Schedule, 24
 feasible, 25
 preemptive, 25
Scheduling algorithm, 23
 D-over, 248
 Deadline Monotonic, 96
 Earliest Deadline First, 56, 93
 Earliest Due Date, 53
 Horn's algorithm, 56
 Jackson's rule, 53
 Latest Deadline First, 68
 Rate Monotonic, 82
 Robust Earliest Deadline, 245
- Scheduling anomalies, 44
Scheduling policy, 23
Scheduling problem, 34
Scheduling, 271
 best effort, 243
 dynamic, 35
 guarantee, 243
 heuristic, 35
 non-preemptive, 35
 off-line, 35
 on-line, 35
 optimal, 35
 preemptive, 35
 robust, 243
 static, 35
Schwan, 227
Search tree, 63
Semaphore Control Block, 262
Semaphore queue, 256
Semaphore, 18, 32, 181, 284
Send operation, 290
Sensory acquisition, 301
Server capacity, 112
Sha, 92, 116, 125, 186, 201
Shankar, 142
Shared resource, 31, 181
Shasha, 248
Signal, 32, 182, 286
Silly, 71
Slack Stealer, 138
Slack time, 27
Sleep state, 260
Soft task, 8, 26
Sporadic Server, 132
Sporadic tasks, 109
Spring algorithm, 66
Spring, 325, 331
Sprunt, 132
Spuri, 150, 160, 185
Stack Resource Policy, 208
Stack sharing, 218
Stanat, 62

Stankovic, 65, 226, 229, 245
 Start time, 27
 Static scheduling, 35
 Stone, 102, 299
 Strosnider, 116, 125
 Synchronization, 284
 Synchronous communication, 289
 System call
 activate, 280
 create, 260
 end_cycle, 281
 end_process, 283
 kill, 283
 sleep, 260
 System ceiling, 212
 System tick, 265

T

Tactile exploration, 317
 Tardiness, 27, 246
 Task Control Block, 261
 Task instance, 27
 Task states, 256
 delay, 256
 idle, 256
 ready, 256
 receive, 257
 running, 256
 sleep, 260
 waiting, 256
 zombie, 258
 Task, 23
 active, 23
 firm, 231
 ready, 23
 running, 23
 TDMA, 329
 Thambidurai, 227
 Tia, 142
 Tick, 265

Time resolution, 265
 Time slice, 25
 Time, 4
 Time-driven scheduling, 109
 Time-overflow, 94–95, 267
 Timeliness, 12
 Timer interrupt, 266
 Timing constraints, 26
 TIMIX, 325
 Tindell, 140
 Total Bandwidth Server, 160
 Transitive inheritance, 190
 Trivedi, 227
 Turing machine, 34

U

UNIX, 324
 Utility function, 43, 230
 Utilization factor, 80

V

Value density, 230, 242
 Value, 27, 230
 Vehicle, 307
 VRTX32, 323
 VxWorks, 221, 323–324

W

Wait, 32, 182, 285
 Waiting state, 32, 182
 Whitehead, 96
 Workload, 228
 Worst-case scenario, 36

Y

YARTOS, 325

Z

- Zhou, 227
Zlokapa, 227
Zombie state, 258



Giorgio C. Buttazzo graduated in 1985 in Electronic Engineering at the University of Pisa (Italy) and in 1987 received a M.S. degree where he also worked on active perception and real-time control at the G.R.A.S.P. (General Robotics and Active Sensory Processing) Laboratory of the University of Pennsylvania. In 1991, he received a Ph.D. degree in robotics at the Scuola Superiore S. Anna of Pisa. He is currently Assistant Professor of Computer Engineering at the Scuola Superiore S. Anna of Pisa. His main research areas include real-time computing, dynamic scheduling algorithms, sensor-based control, advanced robotics, and neural networks.