SERIES IN COMPUTER SCIENCE

Soft Real-Time Systems

Predictability vs. Efficiency

Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo

Soft Real-Time Systems

Predictability vs. Efficiency

SERIES IN COMPUTER SCIENCE

Series Editor: Rami G. Melhem
University of Pittsburgh
Pittsburgh, Pennsylvania

DYNAMIC RECONFIGURATION
Architectures and Algorithms
Ramachandran Vaidyanathan and Jerry L. Trahan

ENGINEERING ELECTRONIC NEGOTIATIONS
A Guide to Electronic Negotiation Technologies for the Design and Implementation of Next-Generation Electronic Markets—Future Silkroads of eCommerce
Michael Ströbel

HIERARCHICAL SCHEDULING IN PARALLEL AND CLUSTER SYSTEMS
Sivarama Dandamudi

MOBILE IP
Present State and Future
Abdul Sakib Mondal

NEAREST NEIGHBOR SEARCH
A Database Perspective
Apostolos N. Papadopoulos and Yannis Manolopoulos

OBJECT-ORIENTED DISCRETE-EVENT SIMULATION WITH JAVA
A Practical Introduction
José M. Garrido

A PARALLEL ALGORITHM SYNTHESIS PROCEDURE FOR HIGH-PERFORMANCE COMPUTER ARCHITECTURES
Ian N. Dunn and Gerard G. L. Meyer

POWER AWARE COMPUTING
Edited by Robert Graybill and Rami Melhem

SOFT REAL-TIME SYSTEMS
Predictability vs. Efficiency
Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo

THE STRUCTURAL THEORY OF PROBABILITY
New Ideas from Computer Science on the Ancient Problem of Probability Interpretation
Paolo Rocchi

Soft Real-Time Systems

Predictability vs. Efficiency

Giorgio Buttazzo
University of Pavia
Pavia, Italy

Giuseppe Lipari
Scuola Superiore Sant'Anna
Pisa, Italy

Luca Abeni
MBI Group
Pisa, Italy

Marco Caccamo
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

Springer

ISBN 0-387-23701-1

©2005 Springer Science+Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America. (BS/DH)

9 8 7 6 5 4 3 2 1 SPIN 1136648

springeronline.com

CONTENTS

Preface vii

1 INTRODUCTION **1**
- 1.1 Basic terminology 1
- 1.2 From hard to soft real-time systems 7
- 1.3 Providing support for soft real-time systems 12

2 OVERLOAD MANAGEMENT **23**
- 2.1 Introduction 23
- 2.2 Load definitions 26
- 2.3 Admission control methods 27
- 2.4 Performance degradation methods 39
- 2.5 Service adaptation 39
- 2.6 Job skipping 42
- 2.7 Period adaptation 47

3 TEMPORAL PROTECTION **59**
- 3.1 Problems without temporal protection 59
- 3.2 Providing temporal protection 62
- 3.3 The GPS model 67
- 3.4 Proportional share scheduling 69
- 3.5 Resource reservation techniques 74
- 3.6 Resource reservations in dynamic priority systems 78
- 3.7 Temporal guarantees 88
- 3.8 Resource reservations in operating system kernels 89

4 MULTI-THREAD APPLICATIONS **95**
- 4.1 The thread model 95
- 4.2 Global approaches 103
- 4.3 Partition-based approaches 116
- 4.4 Concluding remarks and open problems 130

5 SYNCHRONIZATION PROTOCOLS **133**
5.1 Terminology and notation 134
5.2 Shared resource in real-time systems 134
5.3 Synchronization protocols for hard real-time systems 135
5.4 Shared resources in soft real-time systems 141
5.5 Extending resource reservation with the SRP 142
5.6 Resource constraints in dynamic systems 155
5.7 Concluding remarks 174

6 RESOURCE RECLAIMING **175**
6.1 Problems with reservations 175
6.2 The CASH algorithm 177
6.3 The GRUB algorithm 182
6.4 Other forms of reclaiming 190

7 QOS MANAGEMENT **195**
7.1 The QoS-based resource allocation model 195
7.2 Static vs. dynamic resource management 200
7.3 Integrating design & scheduling issues 202
7.4 Smooth rate adaptation 206

8 FEEDBACK SCHEDULING **219**
8.1 Controlling the number of missed deadlines 220
8.2 Adaptive reservations 222
8.3 Application level adaptation 227
8.4 Workload estimators 230

9 STOCHASTIC SCHEDULING **235**
9.1 Background and definitions 236
9.2 Statistical analysis of classical algorithms 238
9.3 Real-time queueing theory 243
9.4 Novel algorithms for stochastic scheduling 246
9.5 Reservations and stochastic guarantee 253

REFERENCES **259**

INDEX **271**

PREFACE

Real-time systems technology, traditionally developed for safety-critical systems, has recently been extended to support novel application domains, including multimedia systems, monitoring apparatuses, telecommunication networks, mobile robotics, virtual reality, and interactive computer games. Such systems are referred to as *soft* real-time systems, because they are often characterized by a highly dynamic behavior and flexible timing requirements. In such systems, missing a deadline does not cause catastrophic consequences on the environment, but only a performance degradation, often evaluated through some quality of service parameter.

Providing an appropriate support at the operating system level to such emerging applications is not trivial. In fact, whereas general purpose operating systems are not predictable enough for guaranteeing the required performance, the classical hard real-time design paradigm, based on worst-case assumptions and static resource allocation, would be too inefficient in this context, causing a waste of the available resources and increasing the overall system cost. For this reason, new methodologies have been investigated for achieving more flexibility in handling task sets with dynamic behavior, as well as higher efficiency in resource exploitation.

This book illustrates the typical characteristics of soft real-time applications and presents some recent methodologies proposed in the literature to support this kind of applications.

Chapter 1 introduces the basic terminology and concepts used in the book and clearly illustrates the main characteristics that distinguish soft real-time computing from other types of computation.

Chapter 2 is devoted to overload management techniques, which are essential in dynamic systems where the computational requirements are highly variable and cannot be predicted in advance.

Chapter 3 introduces the concept of temporal protection, a mechanism for isolating the temporal behavior of a task to prevent reciprocal interference with the other system activities.

Chapter 4 deals with the problem of executing several independent multi-thread applications in the same machine, presenting some methodologies to partition the processor into several virtual slower processors, in such a way that each application can be independently guaranteed from each other.

Chapter 5 presents a number of synchronization protocols for limiting blocking times when mutually exclusive resources are shared among hard and soft tasks.

Chapter 6 describes resource reclaiming techniques, which enhance resource exploitation when the actual resource usage of a task is different than the amount allocated off line. These techniques basically provide a method for reassigning the unused resources to the most demanding tasks.

Chapter 7 treats the issue of quality of service management. It is addressed through an adequate formulation that univocally maps subjective aspects (such as the perceived quality that may depend on the user) to objective values expressed by a real number.

Chapter 8 presents some feedback-based approach to real-time scheduling, useful to adapt the behavior of a real-time system to the actual workload conditions, in highly dynamic environments.

Chapter 9 addresses the problem of performing a probabilistic analysis of real-time task sets, with the aim of providing a relaxed form of guarantee for those real-time systems with highly variable execution behavior. The objective of the analysis is to derive for each task a probability to meet its deadline or, in general, to complete its execution within a given interval of time.

Acknowledgments

This work is the result of several years of research activity in the field of real-time systems. The majority of the material presented in this book is taken from research papers and has been elaborated to be presented in a simplified form and with a uniform structure. Though this book carries the names of four authors, it has been positively influenced by a number of people who gave a substantial contribution in this emerging field. The authors would like to acknowledge Enrico Bini, for his insightful discussions on schedulability analysis, Paolo Gai for his valuable work on kernel design and algorithms implementation, and Luigi Palopoli for his contribution on integrating real-time and control issues. Finally, we would like to thank the Kluwer editorial staff for the support we received during the preparation of the manuscript.

1 INTRODUCTION

In this chapter we explain the reasons why soft real-time computing is being deeply investigated during the last years for supporting a set of application domains for which the hard real-time approach is not suited. Examples of such application domains include multimedia systems, monitoring apparatuses, robotic systems, real-time graphics, interactive games, and virtual reality.

To better understand the difference between classical hard real-time applications and soft real-time applications, we first introduce some basic terminology that will be used throughout the book, then we present the classical design approach used for hard real-time systems, and then describe the characteristics of some soft real-time application. Hence, we identify the major problems that a hard real-time approach can cause in these systems and finally we derives a set of features that a soft real-time system should have in order to provide efficient support for these kind of applications.

1.1 BASIC TERMINOLOGY

In the common sense, a real-time system is a system that reacts to an event within a limited amount of time. So, for example, in a web page reporting the state of a Formula 1 race, we say that the race state is reported in real-time if the car positions are updated "as soon as" there is a change. In this particular case, the expression "as soon as" does not have a precise meaning and typically refers to intervals of a few seconds.

When a computer is used to control a physical device (e.g., a mobile robot), the time needed by the processor to react to events in the environment may significantly affect the overall system's performance. In the example of a mobile robot system, a correct maneuver performed too late could cause serious problems to the system and/or the

environment. For instance, if the robot is running at a certain speed and an obstacle is detected along the robot path, the action of pressing the brakes or changing the robot trajectory should be performed within a maximum delay (which depends on the obstacle distance and on the robot speed), otherwise the robot could not be able to avoid the obstacle, thus incurring in a crash.

Keeping the previous example in mind, a real-time system can be more precisely defined as a computing system in which computational activities must be performed within predefined timing constraints. Hence, the performance of a real-time system depends not only on the functional correctness of the results of computations, but also on the time at which such results are produced.

The word *real* indicates that the system time (that is, the time represented inside the computing system) should always be synchronized with the external time reference with which all time intervals in the environment are measured.

1.1.1 TIMING PARAMETERS

A real-time system is usually modeled as a set of concurrent *tasks*. Each task represents a computational activity that needs to be performed according to a set of constraints. The most significant timing parameters that are typically defined on a real-time computational activity are listed below.

- **Release time** r_i: is the time at which a task becomes ready for execution; it is also referred to as *arrival time* and denoted by a_i;
- **Start time** s_i: is the time at which a task starts its execution for the first time;
- **Computation time** C_i: is the time necessary to the processor for executing the task without interruption;
- **Finishing time** f_i: is the time at which a task finishes its execution;
- **Response time** R_i: is the time elapsed from the task release time and its finishing time ($R_i = f_i - r_i$);
- **Absolute deadline** d_i: is the time before which a task should be completed;
- **Relative deadline** D_i: is the time, relative to the release time, before which a task should be completed ($D_i = d_i - r_i$);

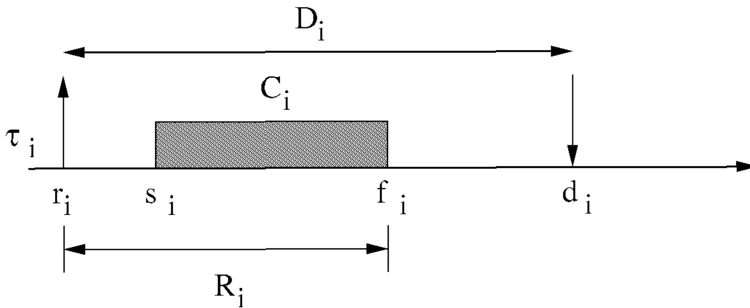


Figure 1.1 Typical timing parameters of a real-time task.

Such parameters are schematically illustrated in Figure 1.1, where the release time is represented by an up arrow and the absolute deadline is represented by a down arrow.

Other parameters that are usually defined on a task are:

- **Slack time** or **Laxity**: denotes the interval between the finishing time and the absolute deadline of a task ($slack_i = d_i - f_i$); it represents the maximum time a task can be delayed to still finish within its deadline;
- **Lateness** L_i: $L_i = f_i - d_i$ represents the completion delay of a task with respect to its deadline; note that if a task completes before its 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.

If the same computational activity needs to be executed several times on different data, then a task is characterized as a sequence of multiple instances, or *jobs*. In general, a task τ_i is modeled as a (finite or infinite) stream of jobs, $\tau_{i,j}$, $(j = 1, 2, \ldots)$, each characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$, a finishing time $f_{i,j}$, and an absolute deadline $d_{i,j}$.

1.1.2 TYPICAL TASK MODELS

Depending on the timing requirements defined on a computation, tasks are classified into four basic categories: hard, firm, soft, and non real time.

A task τ_i is said to be *hard* if all its jobs have to complete within their deadline ($\forall j \ \ f_{i,j} \leq d_{i,j}$), otherwise a critical failure may occur in the system.

A task is said to be *firm* if only a limited number of jobs are allowed to miss their deadline. In [KS95], Koren and Shasha defined a firm task model in which only one job every S is allowed to miss its deadline. When a job misses its deadline, it is aborted and the next $S-1$ jobs must be guaranteed to complete within their deadlines. A slightly different firm model, proposed by Hamdaoui and Ramanathan in [HR95], allows specifying tasks in which at least k jobs every m must meet their deadlines.

A task is said to be *soft* if the value of the produced result gracefully degrades with its response time. For some applications, there is no deadline associated with soft computations. In this case, the objective of the system is to reduce their response times as much as possible. In other cases, a *soft deadline* can be associated with each job, meaning that the job should complete before its deadline to achieve its best performance. However, if a soft deadline is missed, the system keeps working at a degraded level of performance. To precisely evaluate the performance degradation caused by a soft deadline miss, a performance value function can be associated with each soft task, as described in Chapter 2.

Finally, a task is said to be *non real time* if the value of the produced result does not depend on the completion time of its computation.

1.1.3 ACTIVATION MODES

In a computer controlled system, a computational activity can either be activated by a timer at predefined time instants (time-triggered activation) or by the occurrence of a specific event (event-triggered activation).

When jobs activations are triggered by time and are separated by a fixed interval of time, the task is said to be *periodic*. More precisely, a periodic task τ_i is a time-triggered task in which the first job $\tau_{i,1}$ is activated at time Φ_i, called the task phase, and each subsequent job $\tau_{i,j+1}$ is activated at time $r_{i,j+1} = r_{i,j} + T_i$, where T_i is the task period. If D_i is the relative deadline associated with each job, the absolute deadline of job $\tau_{i,j}$ can be computed as:

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

If job activation times are not regular, the task is said to be *aperiodic*. More precisely, an aperiodic task τ_i is a task in which the activation time of job $\tau_{i,k+1}$ is greater than or equal to that of its previous job $\tau_{i,k}$. That is, $r_{i,k+1} \geq r_{i,k}$.

If there is a minimum separation time between successive jobs of an aperiodic task, the task is said to be *sporadic*. More precisely, a sporadic task τ_i is a task in which the difference between the activation times of any two adjacent jobs $\tau_{i,k}$ and $\tau_{i,k+1}$ is greater than or equal to T_i. That is, $r_{i,k+1} \geq r_{i,k} + T_i$. The T_i parameter is called the *minimum interarrival time*.

1.1.4 PROCESSOR WORKLOAD AND BANDWIDTH

For a general purpose computing system, the processor workload depends on the amount of computation required in a unit of time. In a system characterized by aperiodic tasks, the average load $\overline{\rho}$ is computed as the product of the average computation time $\overline{C}$ requested by tasks and the average arrival rate λ:

$$\overline{\rho} = \overline{C}\lambda.$$

In a real-time system, however, the processor load also depends on tasks' timing constraints. The same set of tasks with given computation requirements and arrival patterns will cause a higher load if it has to be executed with more stringent timing constraints.

To measure the load of a real-time system in a given interval of time, Baruah, Howell and Rosier [BMR90] introduced the concept of *processor demand*, defined as follows:

Definition 1.1 *The processor demand $g(t_1, t_2)$ in an interval of time $[t_1, t_2]$ is the amount of computation that has been released at or after t_1 and must be completed within t_2.*

Hence, the processor demand $g_i(t_1, t_2)$ of task τ_i is equal to the computation time requested by those jobs whose arrival times and deadlines are within $[t_1, t_2]$. That is:

$$g_i(t_1, t_2) = \sum_{r_{i,j} \geq t_1, d_{i,j} \leq t_2} c_{i,j}$$

For example, given the set of jobs illustrated in Figure 1.2, the processor demand in the interval $[t_a, t_b]$ is given by the sum of computation times denoted with dark gray, that is, those jobs that arrived at or after t_a and have deadlines at or before t_b.

The total processor demand $g(t_1, t_2)$ of a task set in an interval of time $[t_1, t_2]$ is equal to the sum of the individual demands of each task. That is,

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

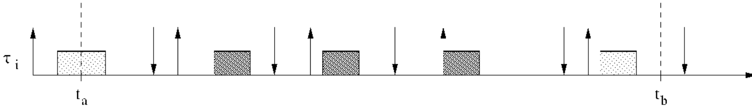


Figure 1.2 Processor demand for a set of jobs.

Then, the processor workload in an interval $[t_1, t_2]$ can be defined as the ratio of the processor demand in that interval and the length of the interval:

$$\rho(t_1, t_2) = \frac{g(t_1, t_2)}{t_2 - t_1}.$$

In the special case of a periodic hard task, the load produced by the task is also called the task *utilization* (U_i) and can be computed as the ratio between the task worst-case computation time C_i and its period T_i:

$$U_i = \frac{C_i}{T_i}.$$

Then, the total processor utilization is defined as the sum of the individual tasks' utilizations:

$$U_p = \sum_{i=1}^{n} U_i.$$

The utilization factor U_i of a periodic task τ_i basically represents the computational bandwidth requested by the task to the processor (assuming that each job will execute for its worst-case execution time). The concept of requested bandwidth can also be generalized to non periodic tasks as follows.

Definition 1.2 *A task τ_i is said to request a bandwidth U_i if, in any interval of time $[t_1, t_2]$, its computational demand $g_i(t_1, t_2)$ never exceeds $(t_2 - t_1)U_i$, and there exists an interval $[t_a, t_b]$ such that $g_i(t_a, t_b) = (t_b - t_a)U_i$.*

1.1.5 OVERLOAD AND OVERRUN

A system is said to be in overload condition when the computational demand of the task set exceeds the available processing time, that is, when there exists an interval of time $[t_a, t_b]$ such that $g(t_a, t_b) > (t_b - t_a)$. In such a situation, computational activities

start to accumulate in system's queues (which tend to become longer and longer, if the overload persists), and tasks response times tend to increase indefinitely. When tasks have timing constraints, an overload condition implies that one or more tasks will miss the deadline (assuming that all tasks execute for their expected computation time).

For a set of periodic tasks, the overload condition is reached when the processor utilization $U_p = \sum_{i=1}^{n} U_i$ exceeds one. Notice, however, that, depending on the adopted scheduling algorithm, tasks may also miss deadlines when the processor is not overloaded (as in the case of the Rate Monotonic algorithm, that has a schedulability bound less than one [LL73]).

While the overload is a condition related to the processor, the overrun is a condition related to a single task.

A task is said to overrun when there exists an interval of time in which its computational demand g_i exceeds its expected bandwidth U_i. This condition may occur either because jobs arrive more frequently than expected (*activation overrun*), or because computation times exceed their expected value (*execution overrun*). Notice that a task overrun does not necessarily cause an overload.

1.2 FROM HARD TO SOFT REAL-TIME SYSTEMS

Real-time systems technology, traditionally used for developing large systems with safety-critical requirements, has recently been extended to support novel application domains, often characterized by less stringent timing requirements, scarce resources, and more dynamic behavior. To provide appropriate support to such emerging applications, new methodologies have been investigated for achieving more flexibility in handling task sets with dynamic behavior, as well as higher efficiency in resource exploitation.

In this section we describe the typical characteristics of hard and soft real-time applications, and present some concrete example to illustrate their difference in terms of application requirements and execution behavior.

1.2.1 CLASSICAL HARD REAL-TIME APPLICATIONS

Real-time computing technology has been primarily developed to support safety-critical systems, such as military control systems, avionic devices, and nuclear power plants. However, it has been also applied to industrial systems that have to guarantee a

certain performance requirements with a limited degree of tolerance. In these systems, also called *hard* real-time systems, most computational activities are characterized by stringent timing requirements, that have to be met in all operating conditions in order to guarantee the correct system behavior. In such a context, missing a single deadline is not tolerated, either because it could have catastrophic effects on the controlled environment, or because it could jeopardize the guarantee of some stringent performance requirements.

For example, a defense missile could miss its target if launched a few milliseconds before or after the correct time. Similarly, a control system could become unstable if the control commands are not delivered at a given rate. For this reason, in such systems, computational activities are modeled as tasks with hard deadlines, that must be met in all predicted circumstances. A task finishing after its deadline is considered not only late, but also wrong, since it could jeopardize the whole system behavior.

In order to guarantee a given performance, hard real-time systems are designed under worst-case scenarios, derived by making pessimistic assumptions on system behavior and on the environment. Moreover, to avoid unpredictable delays due to resource contention, all resources are statically allocated to tasks based on their maximum requirements. Such a design approach allows system designers to perform an off-line analysis to guarantee that the system is able to achieve a minimum desired performance in all operating conditions that have been predicted in advance.

A crucial phase in performing the off-line guarantee is the evaluation of the worst-case computation times (WCETs) of all computational activities. This can be done either experimentally, by measuring the maximum execution time of each task over a large amount of input data, or analytically, by analyzing the source code, identifying the longest path, and computing the time needed to execute it on the specific processor platform. Both methods are not precise. In fact, the first experimental approach fails in that only a limited number of input data can be generated during testing, hence the worst-case execution may not be found. On the other hand, the analytical approach has to make so many assumptions on the low-level mechanisms present in the computer architecture, that the estimation becomes too pessimistic. In fact, in modern computer architectures, the execution time of an instruction depends on several factors, such as the prefetch queue, the DMA, the cache size, and so on. The effects of such mechanisms on task execution are difficult to predict, because they also depends on the previous computation and on the actual data. As a consequence, deriving a precise estimation of the WCET is very difficult (if not impossible). The WCET estimations used in practice are not precise and are affected by large errors (typically more than 20%). This means that to have an absolute off-line guarantee, all tasks execution times have to be overestimated.

Once all computation times are evaluated, the feasibility of the system can be analyzed using several guarantee algorithms proposed in the literature for different scheduling algorithms and task models (see [But97] for a survey of guarantee tests). To simplify the guarantee test and cope with peak load conditions, the schedulability analysis of a task set is also performed under pessimistic assumptions. For example, a set of periodic tasks is typically analyzed under the following assumptions:

- *All tasks start at the same time.* This assumption simplifies the analysis because it has been shown (both under fixed priorities [LL73] and dynamic priorities [BMR90]) that synchronous activations generate the highest workload. Hence, if the system is schedulable when tasks are synchronous, it is also schedulable when they have different activation phases.
- *Job interarrival time is constant for each task.* This is assumption can be enforced by a time-triggered activation mechanism, so making tasks purely periodic. However, there are cases in which the activation of a task depends on the occurrence of an external event (such as the arrival of a message from the network) whose periodicity cannot be precisely predicted.
- *All jobs of a task have the same computation time.* This assumption can be reasonable for tasks having a very simple structure (no branches or loops). In general, however, tasks have loops and branches inside their code, which depend on specific data that cannot be predicted in advance. Hence, the computation time of a job, is highly variable. As a consequence, modeling a task with a fixed computation time equal to the maximum execution time of all its jobs leads to a very pessimistic estimate, which causes a waste of the processing resources.

The consequence of such a worst-case design methodology is that high predictability is achieved at the price of a very low efficiency in resource utilization. Low efficiency also means high cost, since, in order to prevent deadline misses during sporadic peak load conditions, more resources (both in terms of memory and computational power) need to be statically allocated to tasks for coping with the maximum requirements, even though the average requirement of the system is much lower.

In conclusion, if the real-time application consists of tasks with a simple programming structure that can be modeled by a few fixed parameters, then classical schedulability analysis can be effectively used to provide an off-line guarantee under all anticipated scenarios. However, when tasks have a more complex and dynamic behavior, the classical hard real-time design paradigm becomes highly inefficient and less suited for developing embedded systems with scarce resources.

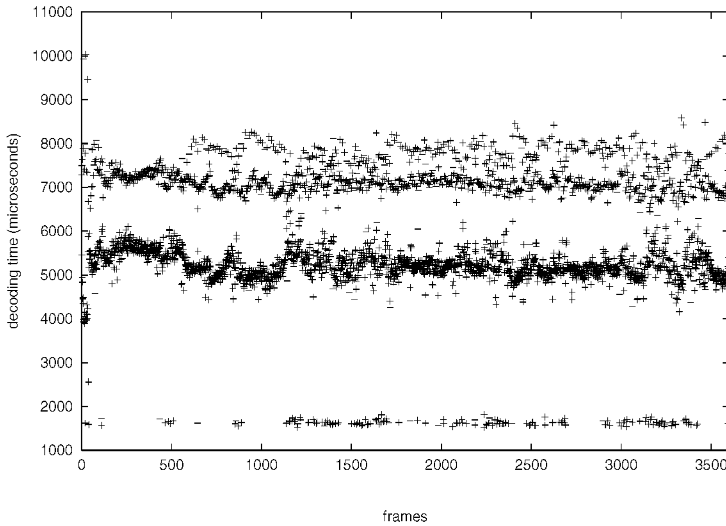


Figure 1.3 Decoding times for a sequence of frames taken from *Star Wars*.

1.2.2 NOVEL APPLICATION DOMAINS

In the last years, emerging time-sensitive applications brought real-time computing into several new different domains, including multimedia systems, monitoring apparatuses, telecommunication networks, mobile robotics, virtual reality, and interactive computer games. In such systems, also called *soft* real-time systems, application tasks are allowed to have less stringent timing constraints, because missing a deadline does not cause catastrophic consequences on the environment, but only a performance degradation, often evaluated through some quality of service (QoS) parameter.

In addition, often, such systems operate in more dynamic environments, where tasks can be created or killed at runtime, or task parameters can change from one job to the other.

There are many soft real-time applications in which the worst-case duration of some tasks is rare but much longer than the average case. In multimedia systems, for instance, the time for decoding a video frame in MPEG players can vary significantly as a function of the data contained in the frames. Figure 1.3 shows the decoding times of frames in a specific sequence of the *Star Wars* movie.

As another example of task with variable computation time, consider a visual tracking system where, in order to increase responsiveness, the moving target is searched in a

small window centered in a predicted position, rather than in the entire visual field. If the target is not found in the predicted area, the search has to be performed in a larger region until, eventually, the entire visual field is scanned in the worst-case. If the system is well designed, the target is found very quickly in the predicted area most of the times. Thus, the worst-case situation is very rare, but very expensive in terms of computational resources (computation time increases quadratically as a function of the number of trials). In this case, an off-line guarantee based on WCETs would drastically reduce the frequency of the tracking task, causing a severe performance degradation with respect to a soft guarantee based on the average execution time.

Just to give a concrete example, consider a videocamera producing images with 512x512 pixels, where the target is a round spot, with a 30 pixels diameter, moving inside the visual field. In this scenario, if U_i is the processor utilization required to track the target in a small window of 64x64 pixels at a rate T_i, a worst-case guarantee would require the tracking task to run 64 times slower in order to demand the same bandwidth in the entire visual field (which is 64 times bigger). Clearly, in this application, it is more convenient to perform a less pessimistic guarantee in order to increase the tracking rate and accept some sporadic overrun as a natural system behavior.

In other situations, periodic tasks could be executed at different rates in different operating conditions. For example, in a flight control system, the sampling rate of the altimeters is a function of the current altitude of the aircraft: the lower the altitude, the higher the sampling frequency. A similar need arises in robotic applications in which robots have to work in unknown environments, where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, in order to maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle. Another example of computation with variable activation rate is engine control, where computation is triggered by the shaft rotation angle, hence task activation is a function of the motor speed.

In all these examples, task parameters are not fixed, as typically considered in a hard task, but vary from a job to the other, depending on the data to be processed.

The problem becomes even more significant when the real-time software runs on top of modern hardware platforms, which include low-level mechanisms such as pipelining, prefetching, caching, or DMA. In fact, although these mechanisms improve the average behavior of tasks, they worsen the worst case, so making much more difficult to provide precise estimates the of worst-case computation times.

To provide a more precise information about the behavior of such dynamic computational activities, one could describe a parameter through a probability distribution derived by experimental data. Figure 1.4 illustrates the probability distribution function of job computation times for the process illustrated in Figure 1.3.

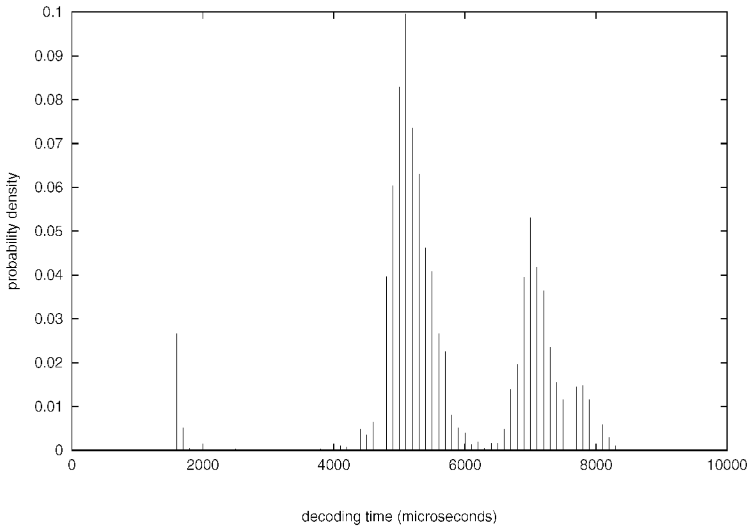


Figure 1.4 Distribution of job computation times for the frame sequence shown in Figure 1.3.

1.3 PROVIDING SUPPORT FOR SOFT REAL-TIME SYSTEMS

Considering the characteristics of the applications describes above, what is the most appropriate way to provide a predictable, as well as efficient, support for them? Can classical (non real-time) operating systems, such as Windows or Linux, be used to develop soft real-time applications? What are their limitations? Can hard real-time systems provide a sufficient support for soft real-time computing? If not, what are the desired features that a real-time kernel should have for this purpose?

In this section we explain why classical general purpose operating systems are not suited for supporting real-time applications. We also explain the limitations of the hard real-time systems and finally we conclude the section with a list of desired features that should be included in a kernel for providing efficient support for soft real-time applications.

1.3.1 PROBLEMS WITH NON REAL-TIME SYSTEMS

The fact that a soft real-time application may tolerate a certain degree of performance degradation does not mean that timing constraints can be completely ignored. For example, in a multimedia application, a quality of service level needs to be enforced on the computational tasks to satisfy a desired performance requirement. If too many deadlines are missed, there is no way to keep the system performance above a certain threshold.

Unfortunately, classical general purpose operating systems, such as Windows or Linux, are not suited for running real-time applications, because they include several internal mechanisms that introduce unbounded delays and cause a high level of unpredictability.

First of all, they do not provide support for controlling explicit timing constraints. System timers are available at a relatively low resolution, and the only kernel service for handling time is given by the *delay()* primitive, which suspends the calling tasks for a given interval of time. The problem with the delay primitive, however, is that, if a task requires to be suspended for an interval of time equal to Δ, the system only guarantees that the calling task will be delayed *at least* by Δ. When using shared resources, the delay primitive can be very unpredictable, as shown in the example illustrated in Figure 1.5. Here, although in normal conditions (a) task τ_1 has a slack equal to 4 units of time, a delay of 2 time units causes the task to miss its deadline (b).

Much longer and unpredictable delays can be introduced during task synchronization, if classical semaphores are used to enforce mutual exclusion in accessing shared resources. For example, consider two tasks, τ_a and τ_b, having priorities $P_a > P_b$, that share a common resource R protected by a semaphore. If τ_b is activated first and is preempted by τ_a inside its critical section, then τ_a is blocked on the semaphore to preserve data consistency in the shared resource. In the absence of other activities, we can clearly see that the maximum blocking time experienced by τ_a on the semaphore is equal to the worst-case duration of the critical section executed by τ_b. However, if other tasks are running in the system, a third task, τ_c, having intermediate priority ($P_b < P_c < P_a$), may preempt τ_b while τ_a is waiting for the resource, so prolonging the blocking time of τ_a for its entire execution. This phenomenon, known as a *priority inversion* [SRL90], is illustrated in Figure 1.6.

In general, if we cannot limit the number of intermediate priority tasks that can run while τ_a is waiting for the resource, the blocking time of τ_a cannot be bounded, preventing any performance guarantee on its execution. The priority inversion phenomenon illustrated above can be solved using specific concurrency control protocols when accessing shared resources, like the *Priority Inheritance Protocol*, the *Priority Ceiling Protocol* [SRL90], or the *Stack Resource Policy* [Bak91]. However, unfortunately, these protocols are not yet available in all general purpose operating systems.

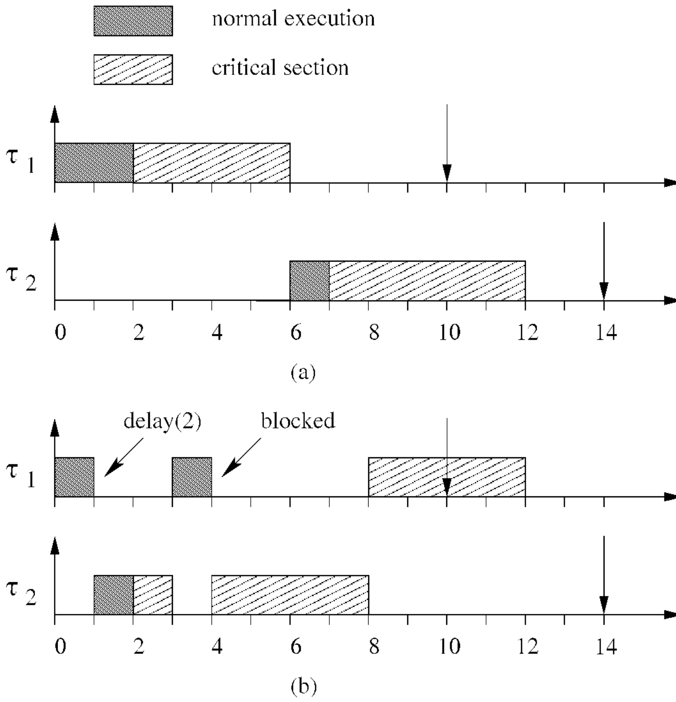


Figure 1.5 Scheduling anomaly caused by the *delay()* primitive: although τ_1 has a slack equal to 4 units of time (a), a *delay(2)* causes the task to miss its deadline (b).

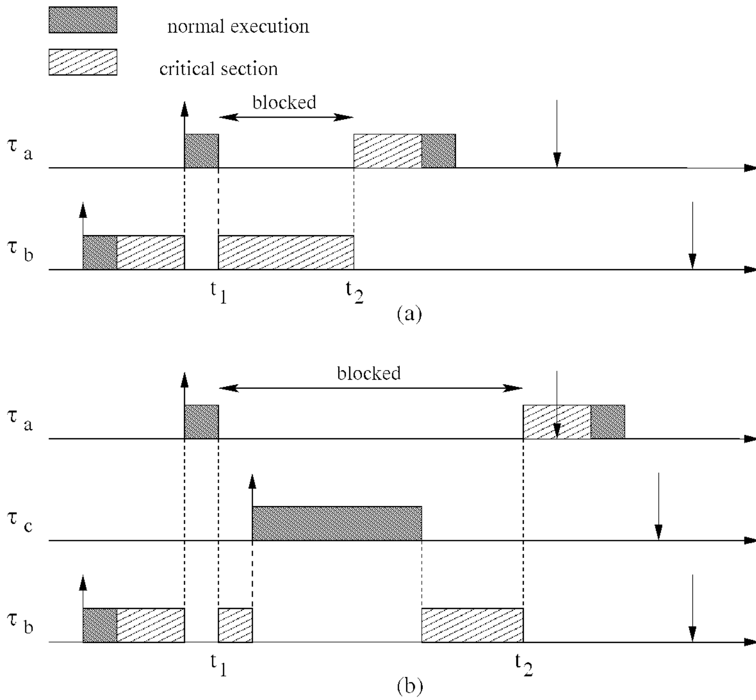


Figure 1.6 The priority inversion phenomenon. In case (a) τ_a is blocked for at most the duration of the critical section of τ_b. In case (b) τ_a is also delayed by the entire execution of τ_c, having intermediate priority.

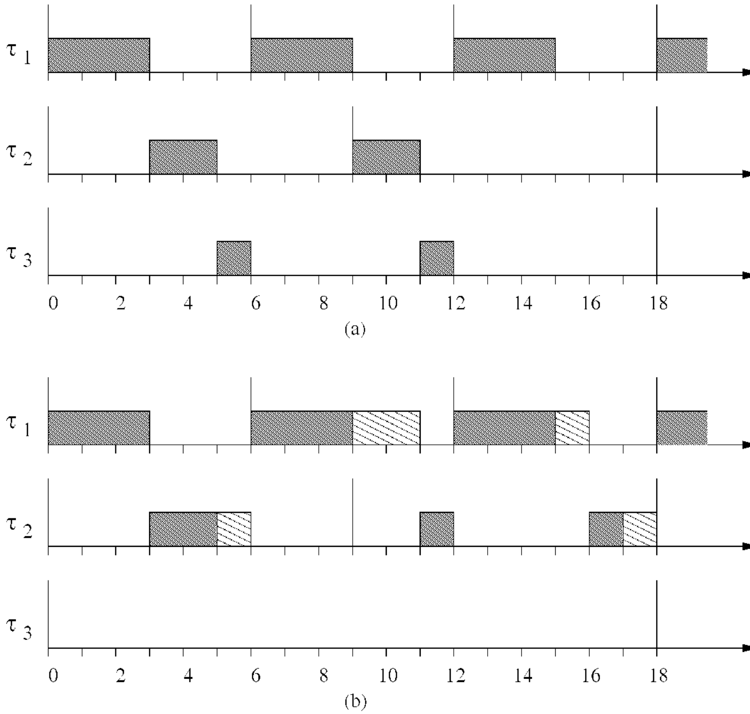


Figure 1.7 Effects of an execution overrun. In normal conditions (a) all tasks execute within their deadlines; but a sequence of overruns in τ_1 and τ_2 may prevent τ_3 to execute (b).

Another negative phenomenon that frequently occurs in general purpose operating systems is a high interference among task executions. Since there is not explicit control on the amount of time each task actually executes on the processor, a high priority task can basically steal as much time as it can to a lower priority computation. When tasks have a highly dynamic behavior, the consequences of the interference on task executions are very difficult to predict, causing a significant performance degradation in the system. In other words, in the absence of specific protection mechanisms in the kernel, an execution overrun occurring in a high priority task may cause very negative effects on several other tasks having lower priority. The problem is illustrated in Figure 1.7, where three periodic tasks, with periods $T_1 = 6$, $T_2 = 9$, $T_3 = 18$, and execution times $C_1 = 3$, $C_2 = 2$, $C_3 = 2$, are scheduled with the Rate Monotonic algorithm. As we can see, in normal conditions (Figure 1.7a) the task set is schedulable, however some execution overruns in τ_1 and τ_2 may prevent τ_3 to execute (Figure 1.7b).

The problem illustrated above becomes more serious in the presence of permanent overload conditions, occurring for example when new tasks are activated and the actual processor utilization is greater than one.

When using non real-time systems, several kernel mechanisms can cause negative effects on real-time computations. For example, typical message passing primitives (like *send* and *receive*) available in kernels for intertask communication adopt a blocking semantics when receiving a message from an empty queue or sending a message into a full queue. If the blocking time cannot be bounded, the delay introduced in task executions can be very high, preventing any performance guarantee. Moreover, a blocking semantics also prevents communication among periodic tasks having different frequencies.

Finally, also the interrupt mechanism, as implemented in general purpose operating systems, contributes to decrease the predictability of the system. In fact, in such systems, device drivers always execute with a priority higher than those assigned to application tasks, preempting the running task at any time. Hence, a bursty sequence of interrupts may introduce arbitrary long delays in the running tasks, causing a severe performance degradation.

1.3.2 PROBLEMS WITH THE HARD REAL-TIME APPROACH

In principle, if a set of tasks with hard timing constraints can be feasibly scheduled by a hard real-time kernel, it can also be feasibly scheduled if the same constraints are considered to be soft. However, there are a number of problems to be taken into account when using a hard real-time design approach for supporting a soft real-time application.

First of all, as we already mentioned above, the use of worst-case assumptions would cause a waste of resources, which would be underutilized for most of the time, just to cope with some sporadic peak load condition. For applications with heavy computational load (e.g., graphical activities), such a waste would imply a severe performance degradation or a significant increase of the system cost. Figure 1.8a shows that, whenever the load has large variations, keeping the load peaks always below one causes the average load to be very small (low efficiency). On the other hand, Figure 1.8b shows that efficiency can only be increased at the cost of accepting some transient overload, by allowing some peak load to exceed one, thus missing some deadlines.

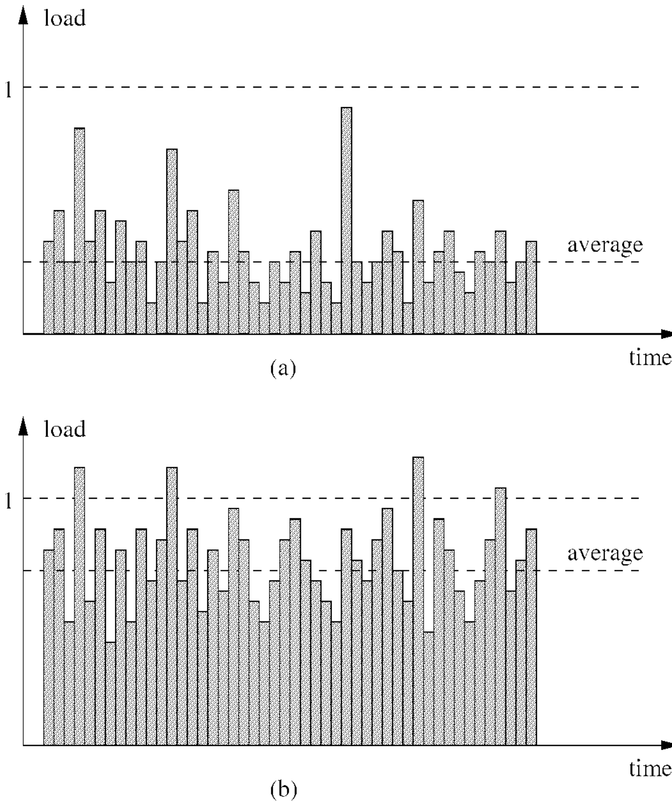


Figure 1.8 Two different load conditions: an underloaded system with low average resource usage (a), and a system with transient overloads but high average resource usage (b).

Another problem with the hard real-time approach is that, in many practical cases, a precise estimation of WCETs is very difficult to achieve. In fact, several low level mechanisms present in modern computer architectures (such as interrupts, DMA, pipelining, caching, and prefetching) introduce a non deterministic behavior in tasks' execution, whose duration cannot be predicted in advance.

Even though a precise WCET estimation could be derived for each task, a worst-case feasibility analysis would be very inefficient when task execution times have a high variance. In this case, a classical off-line hard guarantee would waste the system's computational resources for preserving the task set feasibility under sporadic peak load situations, even though the average workload is much lower. Such a waste of

Task	C_i^{avg}	C_i^{max}	T_i
τ_1	2	2	6
τ_2	3	3	10
τ_3	3	10	12

Table 1.1 Task set parameters.

resources (which increases the overall system's cost) can be justified for very critical applications (e.g., military defense systems or safety critical space missions), in which a single deadline miss may cause catastrophic consequences. However, it does not represent a good solution for those applications (the majority) in which several deadline misses can be tolerated by the system, as long as the average task rates are guaranteed off line.

On the other hand, uncontrolled overruns are very dangerous if not properly handled, since they may heavily interfere with the execution of other tasks, which could be more critical. Consider for example the task set given in Table 1.1, where two tasks, τ_1 and τ_2, have a constant execution time, whereas τ_3 has an average computation time ($C_3^{avg} = 3$) much lower than its worst-case value ($C_3^{max} = 10$). Here, if the schedulability analysis is performed using the average computation time C_3^{avg}, the total processor utilization becomes 0.92, meaning that the system is not overloaded; however, under the Earliest Deadline First (EDF) algorithm [LL73] the tasks can experience long delays during overruns, as illustrated in Figure 1.9. Similar examples can easily be found also under fixed priority assignments (e.g., under the Rate Monotonic algorithm [LL73]), when overruns occur in the high priority tasks (see for example the case illustrated in Figure 1.7).

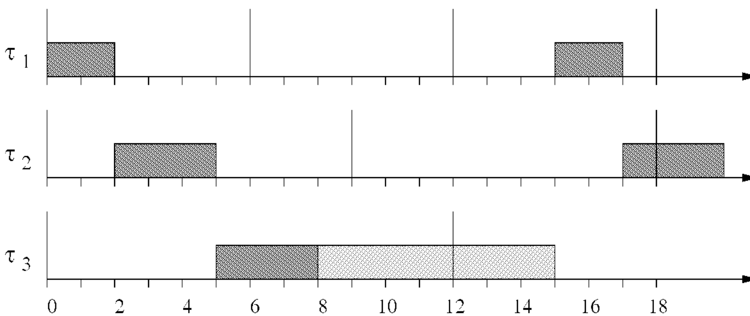


Figure 1.9 Negative effects of uncontrolled overruns under EDF.

To prevent an overrun to introduce unbounded delays on tasks' execution, the system could either decide to abort the current instance of the task experiencing the overrun or let it continue with a lower priority. The first solution is not safe, because the instance could be in a critical section when aborted, thus leaving a shared resource with inconsistent data (very dangerous). The second solution is much more flexible, since the degree of interference caused by the overrun on the other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation.

A general technique for limiting the effects of overruns is based on a resource reservation approach [MST94b, TDS+95, Abe98], according to which each task is assigned (off line) a fraction of the available resources and is handled by a dedicated server, which prevents the served task from demanding more than the reserved amount. Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent from a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

1.3.3 DESIRED FEATURES

From the problems illustrated in the previous sections, we can derive a set of ideal features that a real-time kernel should have to efficiently support soft real-time applications while guaranteeing a certain degree of performance. They are listed below.

- **Overload management**. Whenever the total computational load exceeds the processor capacity, the systems should properly decrease the demand of the task set to avoid uncontrolled performance degradation.
- **Temporal isolation**. The temporal behavior of a computational activity should not depend on the execution characteristics of other activities, but only on the fraction of processor (CPU bandwidth) that has been allocated to it. Whenever a job executes more than expected, or a sequence of jobs arrives more frequently than predicted, only the corresponding task should be delayed, avoiding reciprocal task interference.
- **Bounded priority inversion**. When tasks interact through shared resources, the maximum blocking time caused by mutual exclusion should be bounded by the duration of one or a few critical sections, preventing other tasks to increase blocking delays with their execution.

- **Aperiodic task handling**. Asynchronous arrival of aperiodic events should be handled in such a way that the performance of periodic tasks is guaranteed off-line, and aperiodic responsiveness is maximized.
- **Resource reclaiming**. Any spare time saved by early completions should be exploited for increasing aperiodic responsiveness or coping with transient overload conditions.
- **Adaptation**. For real-time systems working in very dynamic environments, any change in the application behavior should be detected and cause a system adaptation.

Most of the features outlined above are described in detail in the remaining chapters of this book. Aperiodic task scheduling is not treated in detail since it has been already discussed in [But97] in the context of hard real-time systems.

2

OVERLOAD MANAGEMENT

2.1 INTRODUCTION

A system is said to be in overload when the computational demand of the task set exceeds the available processing time. In a real-time system, an overload condition causes one or more tasks to miss their deadline and, if not properly handled, it may cause abrupt degradations of system performance.

Even when the system is properly designed, an overload can occur for different reasons, such as a new task activation, a system mode change, the simultaneous arrival of asynchronous events, a fault in a peripheral device, or the execution of system exceptions.

If the operating system is not conceived to handle overloads, the effect of a transient overload can be catastrophic. 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 2.1.

Figure 2.1a shows a feasible schedule of a task set executed under EDF. However, if at time t_0 task τ_0 is executed, all the previous tasks miss their deadlines (see Figure 2.1b). In general, under EDF, accepting a new task with deadline d^* causes all tasks with deadline longer than d^* to be delayed. Similarly, under fixed priority scheduling, the activation of a task τ_i with priority P_i delays all tasks with lower priority. 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.

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

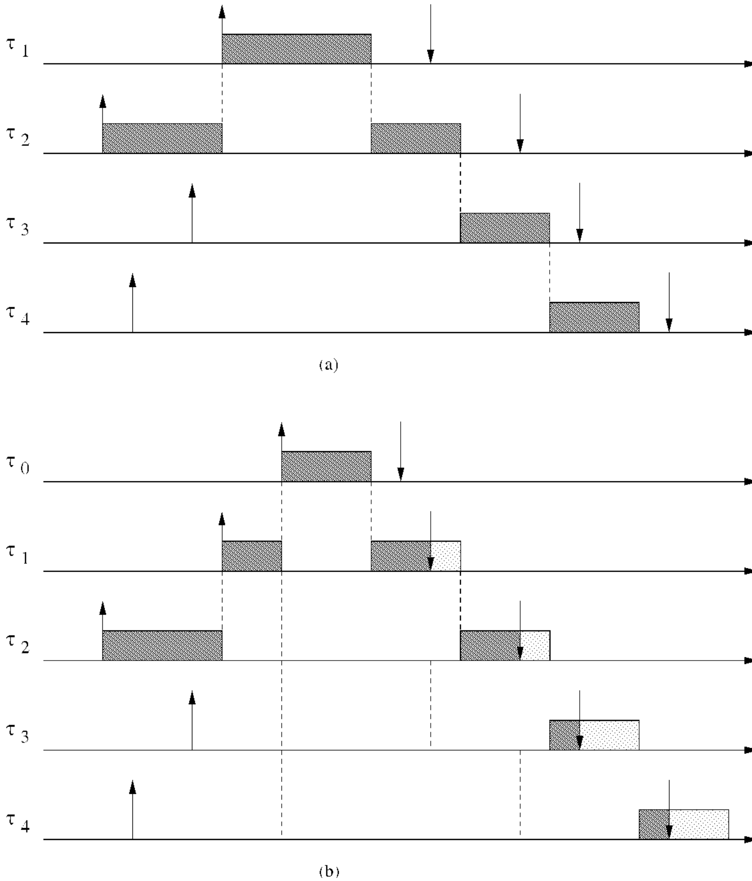


Figure 2.1 **a.** Feasible schedule with Earliest Deadline First, in normal load condition. **b.** Overload with domino effect due to the arrival of task τ_0.

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.

2.2 LOAD DEFINITIONS

In Chapter 1, the processor workload in an interval of time $[t_1, t_2]$ has been defined as the ratio of the processor demand in that interval and the length of the interval:

$$\rho(t_1, t_2) = \frac{g(t_1, t_2)}{t_2 - t_1}.$$

Hence, the system loadin a given schedule is given by

$$\rho = \max_{t_1, t_2} \frac{g(t_1, t_2)}{t_2 - t_1}.$$

Computing the load using the previous definition, however, may not be practical, because the number of intervals $[t_1, t_2]$ can be very large. When the task set consists only of aperiodic activities, then a more effective method is to compute the instantaneous load $\rho(t)$, originally introduced by Buttazzo and Stankovic in [BS95]. According to this method, the load is computed at time t, based on the current set of active aperiodic tasks, each characterized by a remaining computation time $c_i(t)$ and a deadline d_i. In particular, the load at time t is computed as

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

where

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

Figure 2.2a shows an example of load calculation for a set of three real-time aperiodic tasks. At time $t = 6$, when τ_1 arrives, the loading factor $\rho_i(t)$ of each task is shown on the right of the timeline, so the instantaneous load at time 6 is $\rho(6) = 0.833$. Figure 2.2b shows the load as a function of time.

For a set of synchronous periodic tasks with deadlines less than or equal to periods, the processor demand can be computed from time $t = 0$ in an interval of length L as

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

Hence, the total processor workload can be computed as

$$\rho = \max_{L>0} \frac{g(0, L)}{L}.$$

It is worth noticing that Baruah et al. [BMR90] showed that the maximum can be computed for L equal to task deadlines, up to a value $L_{max} = \min(H, L*)$, where H

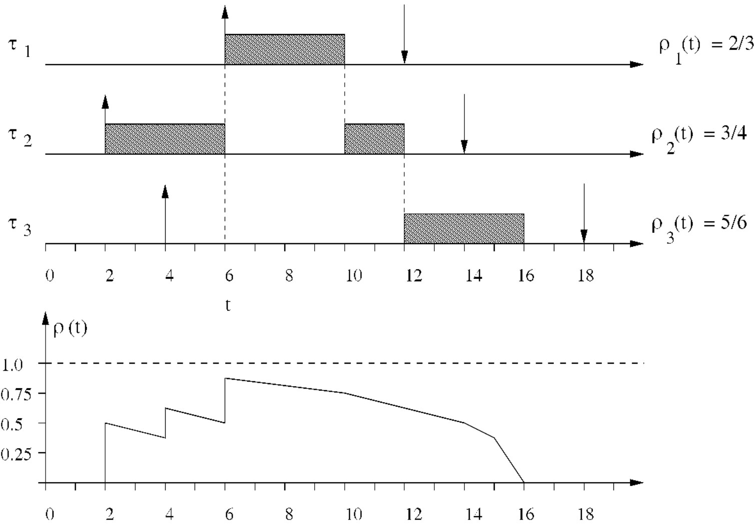


Figure 2.2 a. Load calculation for t = 6 in a set of three real-time tasks. b. Load as a function of time.

is the hyperperiod (i.e., the minimum common multiple of task periods) and

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

The methods proposed in the literature for dealing with permanent overload conditions can be grouped in two main categories:

1. **Admission control**. According to this method, each task is assigned an importance value. Then, in the presence of an overload, the least important tasks are rejected to keep the load under a desired threshold, whereas the other tasks receive full service.
2. **Performance degradation**. According to this method, no task is rejected, but the tasks are executed with reduced performance requirements.

2.3 ADMISSION CONTROL METHODS

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 missing 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.

2.3.1 DEFINING VALUES

The value associated with a task reflects its importance with respect to the other tasks in the set. The specific assignment depends on the particular application. For instance, there are situations in which the value is set equal to the task computation time; in other cases, it is an arbitrary integer number in a given range; in other applications, it is set equal to the ratio of an arbitrary number (which reflects the importance of the task) and the task computation time; this ratio is referred to as the *value density*.

In a real-time system, however, the actual value of a task also depends on the time at which the task is completed; hence, the task importance can be better described by a utility function. Figure 2.3 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 completes its execution. On the contrary, a hard task contributes to a value only if it completes within its deadline, and, since a deadline miss would jeopardize the behavior of the whole system, the value after its deadline can be considered minus infinity in many situations. A 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.

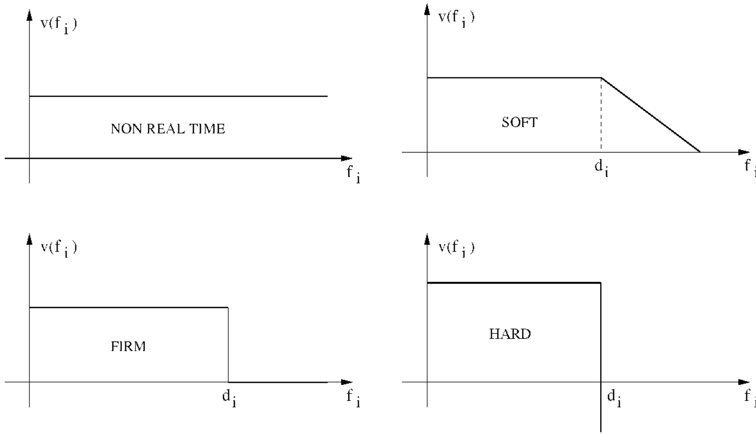


Figure 2.3 Utility functions that can be associated to a task to describe its importance.

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, the *cumulative value* achieved by a scheduling algorithm A is defined as follows:

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

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 possible cumulative value that can be achieved on the 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^*. In this context, a scheduling algorithm that is able to achieve a cumulative value equal to Γ^* is an optimal algorithm.

It is easy to show that no optimal on-line algorithms exist in overloads. 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.

A parameter that measures the worst-case performance of a scheduling algorithm in overload condition is the *competitive factor*, introduced by Baruah et al. in [BKM+92].

Definition 2.1 *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. 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 2.1 (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.

In general, the bound on the competitive factor as a function of the load has been computed in [BR91] and it is shown in Figure 2.4.

With respect to the strategy used to predict and handle overloads, most of the scheduling methods proposed in the literature can be divided into three main classes, illustrated in Figure 2.5:

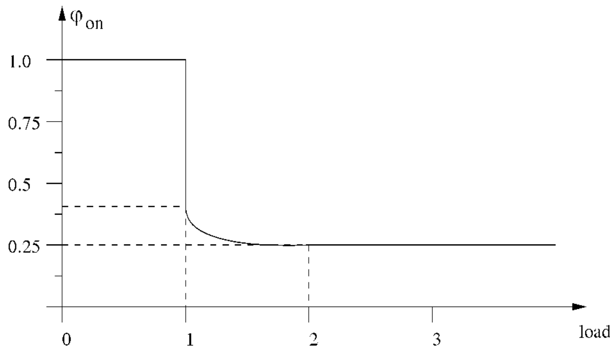


Figure 2.4 Bound of the competitive factor of an on-line scheduling algorithm as a function of the load.

- **Best Effort Scheduling**. 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.

- **Simple Admission Control**. 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 Scheduling**. 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; otherwise, one or more tasks are rejected based on a different policy.

Notice that the simple admission control 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). This scheme, however, does not take task importance into account and, during transient overloads, always rejects the newly arrived task, regardless of its value. In certain

conditions (such as when tasks have very different importance levels), this scheduling strategy may exhibit poor performance in terms of cumulative value, whereas a robust algorithm can be much more effective.

In the best effort scheme, the cumulative value can be increased using suitable heuristics for scheduling the tasks. For example, in the Spring kernel [SR87], Stankovic and Ramamritham proposed to schedule tasks by an appropriate heuristic function that can balance timing properties and importance values.

In 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.

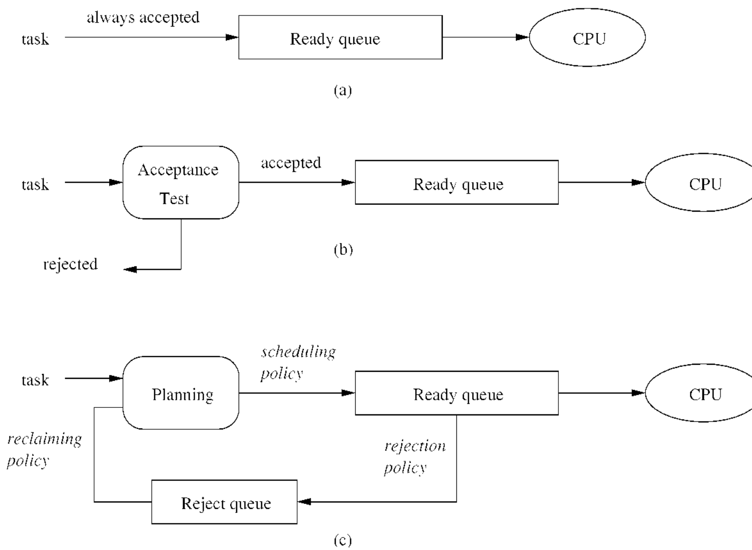


Figure 2.5 Scheduling schemes for handling overload situations. **a.** Best Effort scheduling. **b.** Admission control. **c.** Robust Scheduling.

2.3.2 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 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 L_i, defined as the interval between its estimated finishing time (f_i) and its primary (absolute) deadline (d_i). All residual laxities can be efficiently computed in $O(n)$, 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 executes after its secondary deadline:

$$E_i = \max(0, -(L_i + M_i)). \tag{2.1}$$

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 $E_{max} = 0$.

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 2.6.

```
RED_acceptance_test(J, J_new) {

    E = 0;                    /* Maximum Exceeding Time */
    L_0 = 0;
    d_0 = current_time();

    J' = J ∪ {J_new};
    k = <position of J_new in the task set J'>;

    for each task J'_i such that i ≥ 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 2.6 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.

2.3.3 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 2.3.4, the performance of D_{over} is tested for random task sets and compared with the one of other scheduling algorithms.

2.3.4 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 2.7, 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.

In Figure 2.7, the GED curve refers to the guaranteed EDF scheme implemented with a simple admission control, while the RED curve refers to the robust EDF algorithm. For small values of β, that is, when tasks execute for almost their maximum computation

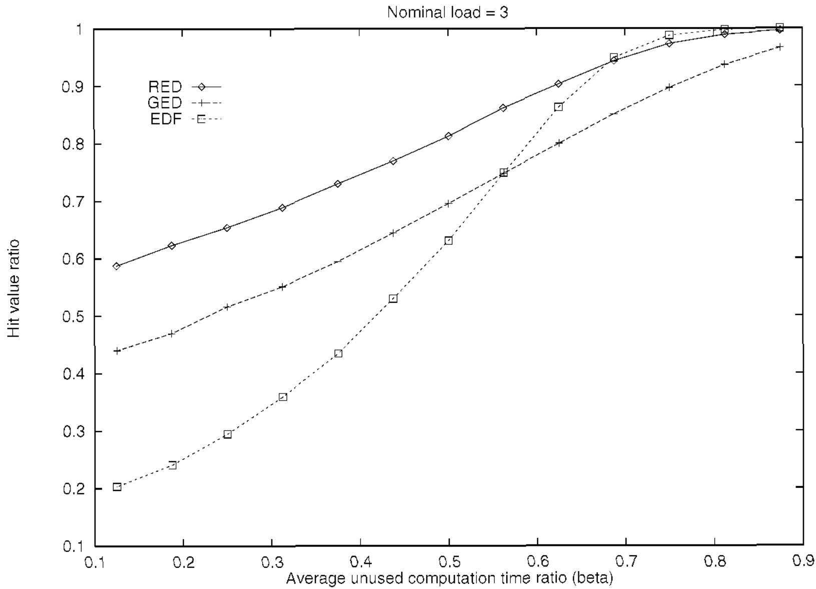


Figure 2.7 Performance of various EDF scheduling schemes: best-effort (EDF), guaranteed (GED) and robust (RED).

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 2.8.

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

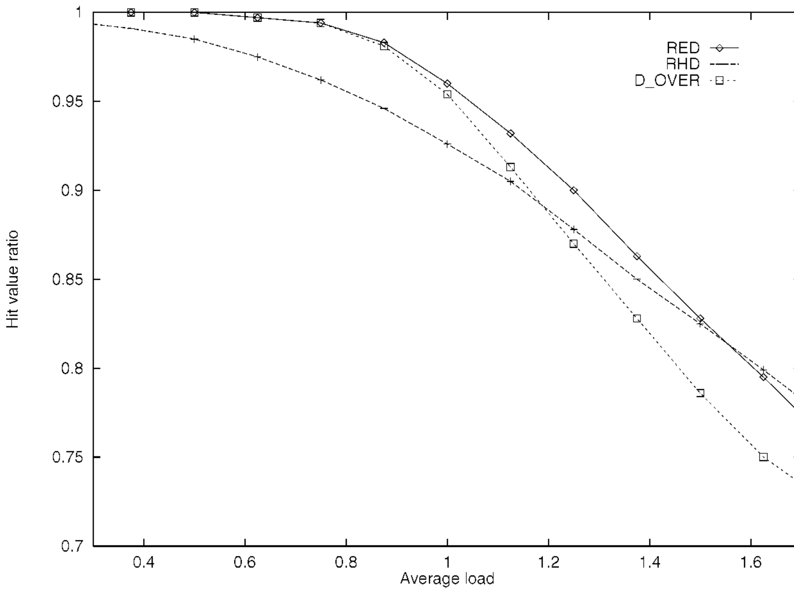


Figure 2.8 Performance of D_{over} against RED and RHD.

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.

2.4 PERFORMANCE DEGRADATION METHODS

According to this approach, the incoming activities that cause an overload are not rejected to preserve the active tasks; rather, the system load is reduced to accommodate them. Load reduction can be achieved at the price of a performance degradation through the following methods:

1. **Service adaptation**. The load is controlled by reducing the computation times of application tasks, thus affecting the quality of results.
2. **Job skipping**. The load is reduced by aborting entire task instances, but still guaranteeing that a minimum percentage of jobs is executed within the specified constraints.
3. **Period adaptation**. The load is reduced by relaxing timing constraints, thus allowing tasks to specify a range of values.

2.5 SERVICE ADAPTATION

A possible method for reducing the service time in overload conditions is to trade precision with computation time. 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$^+$91, LSL$^+$94]. In dynamic situations, where time and resources are not enough for computations to complete within the deadline, there may still be enough resources to produce approximate results. 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. Examples of applications that can exploit this technique include optimization methods (e.g., simulated annealing), cost-based heuristic searches, neural learning, and graphics activities.

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 noticing that the task model used in traditional real-time systems is a special case of the one adopted for imprecise computation. In fact, a hard task corresponds to a task with no optional part ($o_i = 0$), whereas a soft task is equivalent to a task with no mandatory part ($m_i = 0$).

In systems that support imprecise computation, the *error* ϵ_i in the result produced by 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 2.9a. 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 2.9b. In fact, all mandatory subtasks finish within their deadlines, whereas not all optional subtasks are able to complete. In particular, a time unit of execution is subtracted from O_1, two units from O_3, and one unit from O_5. Hence, assuming that all tasks have an importance value equal to one ($w_i = 1$), the average error on the task set is $\bar{\epsilon} = 4$.

When an application task cannot be split into a mandatory and an optional part that can be aborted at any time, service adaptation can still be performed, at a coarse granularity, if multiple versions are provided for the task, each having different length and quality (the longer the task, the higher the quality). In this case, to cope with an overload condition, a high-quality version of a task may be replaced with a shorter one with lower quality. If all the tasks follow such a model, the objective of the system would be to maximize the overall quality under feasibility constraints.

Task	r_i	d_i	C_i	m_i	o_i
τ_1	0	6	4	2	2
τ_2	2	7	4	1	3
τ_3	4	10	5	2	3
τ_4	12	15	3	1	2
τ_5	6	20	8	5	3

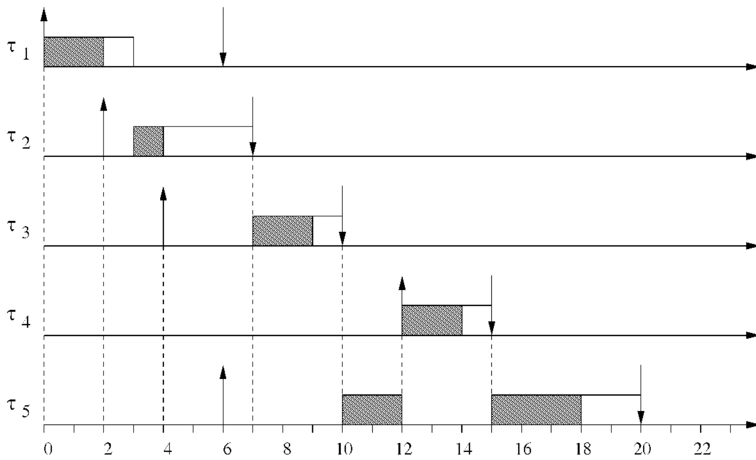


Figure 2.9 An example of an imprecise schedule.

2.6 JOB SKIPPING

Classical real-time scheduling theory ([LL73, SRL90, Bak91]) assumes that periodic tasks have hard timing constraints, meaning that all the instances of a periodic task must be guaranteed to complete within their deadlines. While such task model is suitable for critical control applications, it can be too restrictive for other applications: for example, in multimedia communication systems, skipping a video frame once in a while is acceptable to cope with transient overload situations. Even in more critical control applications, hard real-time tasks usually coexist with firm and soft real-time activities, which need to be treated in a different manner in order to optimize the available resources.

Permitting skips in periodic tasks increases system flexibility, since it allows to make a better use of resources and to schedule systems that would otherwise be overloaded. Consider for example two periodic tasks, τ_1 and τ_2, with periods $p_1 = 10, p_2 = 100$, and execution times $C_1 = 5$ and $C_2 = 55$, where τ_1 can skip an instance every 10 periods, whereas τ_2 is hard (i.e., no instances can be skipped). Clearly, the two tasks cannot be both scheduled as hard tasks, because the processor utilization factor is $U = 5/10 + 55/100 = 1.05 > 1$. However, if τ_1 is permitted to skip one instance every 10 periods, the spare time can be used to accommodate the execution of τ_2.

The *job skipping* model has been originally proposed by Koren and Shasha [KS95]. In their work, the authors showed that making optimal use of skips is NP-hard and presented two algorithms, called Skip-Over Algorithms (one a variant of rate monotonic scheduling and one of earliest deadline first) that exploit skips to increase the feasible periodic load and schedule slightly overloaded systems. According to the job skipping model, the maximum number of skips for each task is controlled by a specific parameter associated with the task. In particular, each periodic task τ_i is characterized by a worst-case computation time c_i, a period p_i, a relative deadline equal to its period, and a skip parameter $s_i, 2 \leq s_i \leq \infty$, which gives the minimum distance between two consecutive skips. For example, if $s_i = 5$ the task can skip one instance every five. When $s_i = \infty$ no skips are allowed and τ_i is equivalent to a hard periodic task. The skip parameter can be viewed as a *Quality of Service* (QoS) metric (the higher s, the better the quality of service).

Using the terminology introduced by Koren and Shasha in [KS95], every instance of a periodic task with skips can be *red* or *blue*. A red instance must complete before its deadline; a blue instance can be aborted at any time. When a blue instance is aborted, we say that it was *skipped*. The fact that $s \geq 2$ implies that, if a blue instance is skipped, then the next $s - 1$ instances must be red. On the other hand, if a blue instance completes successfully, the next task instance is also blue.

Two on-line scheduling algorithms were proposed in [KS95] to handle tasks with skips under EDF.

1. The first algorithm, called *Red Tasks Only* (RTO), always rejects the blue instances, whereas the red ones are scheduled according to EDF.
2. The second algorithm, called *Blue When Possible* (BWP), is more flexible than RTO and schedules blue instances whenever there are no ready red jobs to execute. Red instances are scheduled according to EDF.

It is easy to find examples that show that BWP improves RTO in the sense that it is able to schedule task sets that RTO cannot schedule. In the general case, the above algorithms are not optimal, but they become optimal under a particular task model, called the *deeply-red* model.

Definition 2.2 *A system is deeply-red if all tasks are synchronously activated and the first* $s_i - 1$ *instances of every task* τ_i *are red.*

In the same paper, Koren and Shasha showed that the worst case for a periodic skippable task set occurs when tasks are deeply-red, hence all the results are derived under this assumption. This means that, if a task set is schedulable under the deeply-red model, it is also schedulable without this assumption.

In the hard periodic model in which all task instances are red (i.e., no skips are permitted), the schedulability of a periodic task set can be tested using a simple necessary and sufficient condition based upon cumulative processor utilization. In particular, Liu and Layland [LL73] showed that a periodic task set is schedulable by EDF if and only if its cumulative processor utilization is no greater than 1. Analyzing the feasibility of firm periodic tasks is not equally easy. Koren and Shasha proved that determining whether a set of skippable periodic tasks is schedulable is NP-hard. They also found that, given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of firm periodic tasks that allow skips, then

$$\sum_{i=1}^{n} \frac{c_i(s_i - 1)}{p_i s_i} \leq 1 \qquad (2.2)$$

is a necessary condition for the feasibility of Γ, since it represents the utilization based on the computation that must take place.

To better clarify the concepts mentioned above consider the task set shown in Figure 2.10 and the corresponding feasible schedule, obtained by EDF. Notice that the

Task	Task1	Task2	Task3
Computation	1	2	5
Period	3	4	12
Skip Parameter	4	3	∞
U_p	1.25		

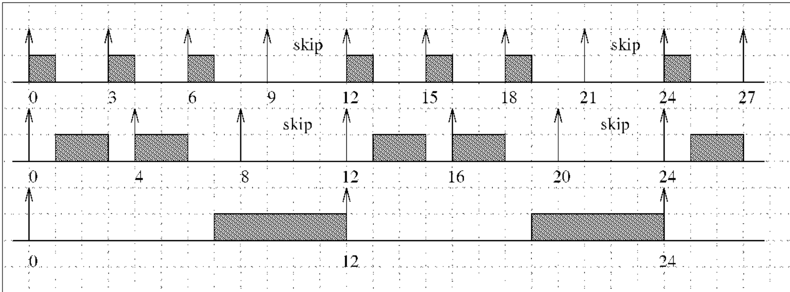


Figure 2.10 A schedulable set of firm periodic tasks.

processor utilization factor is greater than 1 ($U_p = 1.25$), but condition (2.2) is satisfied.

In the same work, Koren and Shasha proved the following theorem, which provides a sufficient condition for guaranteeing a set of skippable periodic tasks under EDF.

Theorem 2.2 *A set of firm (i.e., skippable) periodic tasks is schedulable if*

$$\forall L \geq 0 \qquad L \geq \sum_{i=1}^{n} D(i, [0, L]) \tag{2.3}$$

where

$$D(i, [0, L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i. \tag{2.4}$$

If skips are permitted in the periodic task set, the spare time saved by rejecting the blue instances can be reallocated for other purposes. For example, for scheduling slightly overloaded systems or for advancing the execution of soft aperiodic requests.

Unfortunately, the spare time has a "granular" distribution and cannot be reclaimed at any time. Nevertheless, it can be shown that skipping blue instances still produces a

bandwidth saving in the periodic schedule. In [CB97], Caccamo and Buttazzo generalized the results of Theorem 2.2 by identifying the amount of bandwidth saving achieved by skips. To express this fact with a simple parameter, they defined an *equivalent utilization factor* U_p^* for periodic tasks with skips.

Definition 2.3 *Given a set* $\Gamma = \{T_i(p_i, c_i, s_i)\}$ *of* n *periodic tasks that allow skips, an* equivalent processor utilization factor *can be defined as:*

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_i D(i, [0, L])}{L} \right\} \tag{2.5}$$

where

$$D(i, [0, L]) = \left(\left\lfloor \frac{L}{p_i} \right\rfloor - \left\lfloor \frac{L}{p_i s_i} \right\rfloor \right) c_i.$$

The following theorem ([CB97]) states the schedulability condition for a set of deeply-red skippable tasks.

Theorem 2.3 *A set* Γ *of skippable periodic tasks, which are deeply-red, is schedulable if and only if*

$$U_p^* \leq 1.$$

The bandwidth saved by skipping blue instances can easily be exploited by an aperiodic server (like CBS [AB98a] described in Chapter 3) to advance the execution of aperiodic tasks. The following theorem ([CB97]) provides a sufficient condition for guaranteeing a feasible schedule of a hybrid task set consisting of n firm periodic tasks and a number of soft aperiodic tasks handled by a server with bandwidth U_s.

Theorem 2.4 *Given a set of periodic tasks that allow skip with equivalent utilization* U_p^* *and a set of soft aperiodic tasks handled by a server with utilization factor* U_s, *the hybrid set is schedulable by RTO or BWP if:*

$$U_p^* + U_s \leq 1. \tag{2.6}$$

The sufficient condition of Theorem 2.4 is a consequence of the "granular" distribution of the spare time produced by skips. In fact, a fraction of this spare time is uniformly distributed along the schedule and can be used as an additional free bandwidth ($U_{skip} = U_p - U_p^*$) available for aperiodic service. The remaining portion of the spare time saved

by skips is discontinuous, and creates a kind of "holes" in the schedule, which cannot be used at any time, unfortunately. Whenever an aperiodic request falls into some hole, it can exploit a bandwidth greater than $1 - U_p^*$. Indeed, it is easy to find examples in which a periodic task set is schedulable by assigning the aperiodic server a bandwidth U_s greater than $1 - U_p^*$. The following theorem ([CB97]) gives a maximum bandwidth $U_{s_{max}}$ above which the schedule is certainly not feasible.

Theorem 2.5 *Given a set $\Gamma = \{T_i(p_i, c_i, s_i)\}$ of n periodic tasks that allow skips and an aperiodic server with bandwidth U_s, a necessary condition for the feasibility of Γ is:*

$$U_s \leq U_{s_{max}}$$

where

$$U_{s_{max}} = 1 - U_p + \sum_{i=1}^{n} \frac{c_i}{p_i s_i}. \tag{2.7}$$

AN EXAMPLE

As an illustrative example, let us consider the task set shown in Table2.1. The task set consists of two periodic tasks, τ_1 and τ_2, with periods 3 and 5, computation times 2 and 2, and skip parameters 2 and ∞ respectively. The equivalent utilization factor of the periodic task set is $U_p^* = 4/5$ while $U_{s_{max}} = 0.27$, leaving a bandwidth of $U_s = 1/5$ for the aperiodic tasks. Three aperiodic jobs J_1, J_2, and J_3 are released at times $t_1 = 0$, $t_2 = 6$, and $t_3 = 18$; moreover, they have computation times $c_1^{ape} = 1$, $c_2^{ape} = 2$, and $c_3^{ape} = 1$, respectively.

Task	**Task1**	**Task2**
Computation	2	2
Period	3	5
Skip Parameter	2	∞
U_p	1.07	
U_p^*	0.8	
$1 - U_p^*$	0.2	
$U_{s_{max}}$	0.27	

Table 2.1 A schedulable task set.

Supposing that the aperiodic activities are scheduled by a CBS server with maximum budget $Q^s = 1$ and server period $P^s = 5$, Figure 2.11 shows the resulting schedule

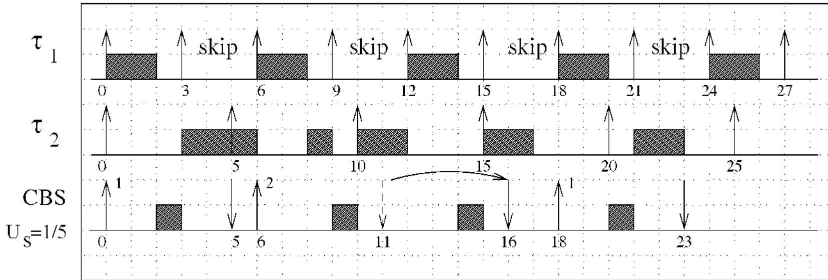


Figure 2.11 Schedule produced by RTO+CBS for the task set shown in Table 2.1.

by using RTO+CBS. Notice that J_2 has a deadline postponement (according to CBS rules) at time $t = 10$ with new server deadline $d^s_{new} = d^s_{old} + P^s = 11 + 5 = 16$. According to the sufficient schedulability test provided by Theorem 2.4, the task set is schedulable when the aperiodic server has assigned a bandwidth $U_s = 1 - U^*_p$.

2.7 PERIOD ADAPTATION

In a periodic task set, processor utilization can also be changed by varying task periods. Whenever the system load becomes greater than a maximum threshold, the periods can be enlarged in a proper way to bring the system load back to the desired value. Today, there are many real-time applications in which timing constraints are not rigid, but can be varied to better react to transient overload conditions.

For example, in multimedia systems, activities such as voice sampling, image acquisition, sound generation, data compression, and video playing, are performed periodically, but their execution rates are not as rigid as in control applications. Missing a deadline while displaying an MPEG video may decrease the quality of service (QoS), but does not cause critical system faults. Depending on the requested QoS, tasks may increase or decrease their execution rate to accommodate the requirements of other concurrent activities.

Even in some control application, there are situations in which periodic tasks could be executed at different rates in different operating conditions. For example, in a flight control system, the sampling rate of the altimeters is a function of the current altitude of the aircraft: the lower the altitude, the higher the sampling frequency. A similar need arises in robotic applications in which robots have to work in unknown environments

where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, in order to maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle.

In other situations, the possibility of varying tasks' rates increases the flexibility of the system in handling overload conditions, providing a more general admission control mechanism. For example, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

Unfortunately, there is no uniform approach for dealing with these situations. For example, Kuo and Mok [KM91] propose a load scaling technique to gracefully degrade the workload of a system by adjusting the periods of processes. In this work, tasks are assumed to be equally important and the objective is to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. In [NT94], Nakajima and Tezuka show how a real-time system can be used to support an adaptive application: whenever a deadline miss is detected, the period of the failed task is increased. In [SLSS97], Seto et al. change tasks' periods within a specified range to minimize a performance index defined over the task set. This approach is effective at a design stage to optimize the performance of a discrete control system, but cannot be used for on-line load adjustment. In [LRM96], Lee, Rajkumar and Mercer propose a number of policies to dynamically adjust the tasks' rates in overload conditions. In [AAS97], Abdelzaher, Atkins, and Shin present a model for QoS negotiation to meet both predictability and graceful degradation requirements during overloads. In this model, the QoS is specified as a set of negotiation options, in terms of rewards and rejection penalties. In [Nak98a, Nak98b], Nakajima shows how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational demand. However, it is not clear how the QoS can be increased when the system is underloaded. In [BCRZ99], Beccari et al. propose several policies for handling overload through period adjustment. The authors, however, do not address the problem of increasing the task rates when the processor is not fully utilized.

Although these approaches may lead to interesting results in specific applications, a more general framework can be used to avoid a proliferation of policies and treat different applications in a uniform fashion.

The elastic model presented in this section was originally introduced by Buttazzo et al. [BAL98] and then extended to a more general case [BLCA02]). It provides a novel theoretical framework for flexible workload management in real-time applications. In particular, the elastic approach provides the following advantages with respect to the classical "fixed-rate" approach.

- it allows tasks to intentionally change their execution rate to provide different quality of service;
- it can handle overload situations in a more flexible fashion;
- it provides a simple and efficient method for controlling the system's performance as a function of the current workload.

EXAMPLES

To better understand the idea behind the elastic model, consider a set of three periodic tasks, with computation times $C_1 = 10$, $C_2 = 10$, and $C_3 = 15$ and periods $T_1 = 20$, $T_2 = 40$, and $T_3 = 70$. Clearly, the task set is schedulable by EDF because

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.964 < 1.$$

Now, suppose that a new periodic task τ_4, with computation time $C_4 = 5$ and period $T_4 = 30$, enters the system at time t. The total processor utilization of the new task set is

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{50} + \frac{5}{30} = 1.131 > 1.$$

In a rigid scheduling framework, τ_4 should be rejected to preserve the timing behavior of the previously guaranteed tasks. However, τ_4 can be accepted if the periods of the other tasks can be increased in such a way that the total utilization is less than one. For example, if T_1 can be increased up to 23, the total utilization becomes $U_p = 0.989$, and hence τ_4 can be accepted.

As another example, if tasks are allowed to change their frequency and task τ_3 reduces its period to 50, no feasible schedule exists, since the utilization would be greater than 1:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{50} = 1.05 > 1.$$

However, notice that a feasible schedule exists ($U_p = 0.977$) for $T_1 = 22$, $T_2 = 45$, and $T_3 = 50$. Hence, the system can accept the higher request rate of τ_3 by slightly decreasing the rates of τ_1 and τ_2. Task τ_3 can even run with a period $T_3 = 40$, since a feasible schedule exists with periods T_1 and T_2 within their range. In fact, when $T_1 = 24$, $T_2 = 50$, and $T_3 = 40$, $U_p = 0.992$. Finally, notice that if τ_3 requires to run at its minimum period ($T_3 = 35$), there is no feasible schedule with periods T_1 and T_2 within their range, hence the request of τ_3 to execute with a period $T_3 = 35$ must be rejected.

Clearly, for a given value of T_3, there can be many different period configurations which lead to a feasible schedule; thus, one of the possible feasible configurations must be selected. The elastic approach provides an efficient way for quickly selecting a feasible period configuration among the all possible solutions.

2.7.1 THE ELASTIC MODEL

The basic idea behind the elastic model is to consider each task as flexible as a spring with a given rigidity coefficient and length constraints. In particular, the utilization of a task is treated as an elastic parameter, whose value can be modified by changing the period within a specified range.

Each task is characterized by four parameters: a computation time C_i, a nominal period T_{i_0} (considered as the minimum period), a maximum period $T_{i_{max}}$, and an elastic coefficient $E_i \geq 0$, which specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration. The greater E_i, the more elastic the task. Thus, an elastic task is denoted as:

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i).$$

In the following, T_i will denote the actual period of task τ_i, which is constrained to be in the range $[T_{i_0}, T_{i_{max}}]$. Any task can vary its period according to its needs within the specified range. Any variation, however, is subject to an *elastic* guarantee and is accepted only if there exists a feasible schedule in which all the other periods are within their range.

It is worth noting that the elastic model is more general than the classical Liu and Layland's task model, so it does not prevent a user from defining hard real-time tasks. In fact, a task having $T_{i_{max}} = T_{i_0}$ is equivalent to a hard real-time task with fixed period, independently of its elastic coefficient. A task with $E_i = 0$ can arbitrarily vary its period within its specified range, but it cannot be varied by the system during load reconfigurations.

To understand how an elastic guarantee is performed in this model, it is convenient to compare an elastic task τ_i with a linear spring S_i characterized by a rigidity coefficient k_i, a nominal length x_{i_0}, and a minimum length $x_{i_{min}}$. In the following, x_i will denote the actual length of spring S_i, which is constrained to be greater or equal to $x_{i_{min}}$.

In this comparison, the length x_i of the spring is equivalent to the task's utilization factor $U_i = C_i/T_i$, and the rigidity coefficient k_i is equivalent to the inverse of the task's elasticity ($k_i = 1/E_i$). Hence, a set of n tasks with total utilization factor $U_p = \sum_{i=1}^{n} U_i$ can be viewed as a sequence of n springs with total length $L = \sum_{i=1}^{n} x_i$.

Under the elastic model, given set of n periodic tasks with $U_p > U_{max}$, the objective of the guarantee is to compress tasks' utilization factors in order to achieve a new desired utilization $U_d \leq U_{max}$ such that all the periods are within their ranges. In the linear spring system, this is equivalent of compressing the springs so that the new total length L_d is less than or equal to a given maximum length L_{max}. More formally, in the spring system the problem can be stated as follows.

> Given a set of n springs with known rigidity and length constraints, if $L_0 = \sum_{i=1}^{n} x_{i_0} > L_{max}$, find a set of new lengths x_i such that $x_i \geq x_{i_{min}}$ and $L = L_d$, where L_d is any arbitrary desired length such that $L_d < L_{max}$.

For the sake of clarity, we first solve the problem for a spring system without length constraints, then we show how the solution can be modified by introducing length constraints, and finally we show how the solution can be adapted to the case of a task set.

SPRINGS WITH NO LENGTH CONSTRAINTS

Consider a set Γ of n springs with nominal length x_{i_0} and rigidity coefficient k_i positioned one after the other, as depicted in Figure 2.12. Let L_0 be the total length of the array, that is the sum of the nominal lengths: $L_0 = \sum_{i=1}^{n} x_{i_0}$. If the array is compressed so that its total length is equal to a desired length L_d ($0 < L_d < L_0$), the first problem we want to solve is to find the new length x_i of each spring, assuming that for all i, $0 < x_i < x_{i_0}$ (i.e., $x_{i_{min}} = 0$). Being L_d the total length of the compressed array of springs, we have:

$$L_d = \sum_{i=1}^{n} x_i. \tag{2.8}$$

If F is the force that keeps a spring in its compressed state, then, for the equilibrium of the system, it must be:

$$\forall i \quad F = k_i(x_{i_0} - x_i),$$

from which we derive

$$\forall i \quad x_i = x_{i_0} - \frac{F}{k_i}. \tag{2.9}$$

By summing equations (2.9) we have:

$$L_d = L_0 - F \sum_{i=1}^{n} \frac{1}{k_i}.$$

Thus, force F can be expressed as

$$F = K_p(L_0 - L_d), \tag{2.10}$$

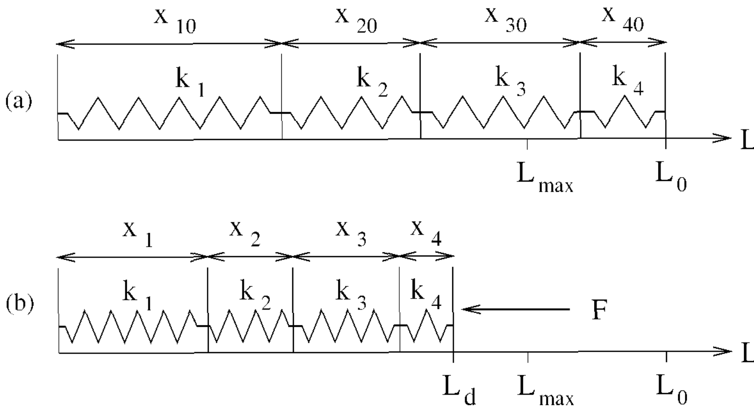


Figure 2.12 A linear spring system: the total length is L_0 when springs are uncompressed (a); and $L_d < L_0$ when springs are compressed by applying a force F (b).

where

$$K_p = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}}. \tag{2.11}$$

Substituting expression (2.10) into Equations (2.9) we finally achieve:

$$\forall i \quad x_i = x_{i_0} - (L_0 - L_d)\frac{K_p}{k_i}. \tag{2.12}$$

Equation (2.12) allows us to compute how each spring has to be compressed in order to have a desired total length L_d.

INTRODUCING LENGTH CONSTRAINTS

If each spring has a length constraint, in the sense that its length cannot be less than a minimum value $x_{i_{min}}$, then the problem of finding the values x_i requires an iterative solution. In fact, if during compression one or more springs reach their minimum length, the additional compression force will only deform the remaining springs. Thus, at each instant, the set Γ can be divided into two subsets: a set Γ_f of fixed springs having minimum length, and a set Γ_v of variable springs that can still be compressed. Applying equations (2.12) to the set Γ_v of variable springs, we have

$$\forall S_i \in \Gamma_v \quad x_i = x_{i_0} - (L_{v_0} - L_d + L_f)\frac{K_v}{k_i} \tag{2.13}$$

where

$$L_{v_0} = \sum_{S_i \in \Gamma_v} x_{i_0} \tag{2.14}$$

$$L_f = \sum_{S_i \in \Gamma_f} x_{i_{min}} \tag{2.15}$$

$$K_v = \frac{1}{\sum_{S_i \in \Gamma_v} \frac{1}{k_i}}. \tag{2.16}$$

Whenever there exists some spring for which equation (2.13) gives $x_i < x_{i_{min}}$, the length of that spring has to be fixed at its minimum value, sets Γ_f and Γ_v must be updated, and equations (2.13), (2.14), (2.15) and (2.16) recomputed for the new set Γ_v. If there exists a feasible solution, that is, if the desired final length L_d is greater than or equal to the minimum possible length of the array $L_{min} = \sum_{i=1}^{n} x_{i_{min}}$, the iterative process ends when each value computed by equations (2.13) is greater than or equal to its corresponding minimum $x_{i_{min}}$. The complete algorithm for compressing a set Γ of n springs with length constraints up to a desired length L_d is shown in Figure 2.13.

COMPRESSING TASKS' UTILIZATIONS

When dealing with a set of elastic tasks, equations (2.13), (2.14), (2.15) and (2.16) can be rewritten by substituting all length parameters with the corresponding utilization factors, and the rigidity coefficients k_i and K_v with the corresponding elastic coefficients E_i and E_v. Similarly, at each instant, the set Γ of periodic tasks can be divided into two subsets: a set Γ_f of fixed tasks having minimum utilization, and a set Γ_v of variable tasks that can still be compressed. Let $U_{i_0} = C_i/T_{i_0}$ be the nominal utilization of task τ_i, $U_0 = \sum_{i=1}^{n} U_{i_0}$ be the nominal utilization of the task set, U_{v_0} be the sum of the nominal utilizations of tasks in Γ_v, and U_f be the total utilization factor of tasks in Γ_f. Then, to achieve a desired utilization $U_d < U_0$ each task has to be compressed up to the following utilization:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - (U_{v_0} - U_d + U_f)\frac{E_i}{E_v} \tag{2.17}$$

where

$$U_{v_0} = \sum_{\tau_i \in \Gamma_v} U_{i_0} \tag{2.18}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{2.19}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{2.20}$$

If there exist tasks for which $U_i < U_{i_{min}}$, then the period of those tasks has to be fixed at its maximum value $T_{i_{max}}$ (so that $U_i = U_{i_{min}}$), sets Γ_f and Γ_v must be updated (hence, U_f and E_v recomputed), and equation (2.17) applied again to the tasks in

```
Algorithm Spring_compress(Γ, L_d) {

    L_0 = Σ_{i=1}^{n} x_{i_0};
    L_min = Σ_{i=1}^{n} x_{i_min};
    if (L_d < L_min) return INFEASIBLE;

    do {

        Γ_f = {S_i | x_i = x_{i_min}};
        Γ_v = Γ − Γ_f;

        L_{v0} = Σ_{S_i ∈ Γ_v} x_{i_0};
        L_f = Σ_{S_i ∈ Γ_f} x_{i_min};
        K_v = 1 / (Σ_{S_i ∈ Γ_v} 1/k_i);

        ok = 1;
        for (each S_i ∈ Γ_v) {
            x_i = x_{i_0} − (L_{v0} − L_d + L_f) K_v / k_i;
            if (x_i < x_{i_min}) {
                x_i = x_{i_min};
                ok = 0;
            }
        }

    } while (ok == 0);
    return FEASIBLE;
}
```

Figure 2.13 Algorithm for compressing a set of springs with length constraints.

Γ_v. If there exists a feasible solution, that is, if the desired utilization U_d is greater than or equal to the minimum possible utilization $U_{min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{max}}}$, the iterative process ends when each value computed by equation (2.17) is greater than or equal to its corresponding minimum $U_{i_{min}}$. The algorithm[1] for compressing a set Γ of n elastic tasks up to a desired utilization U_d is shown in Figure 2.14.

DECOMPRESSION

All tasks' utilizations that have been compressed to cope with an overload situation can return toward their nominal values when the overload is over. Let Γ_c be the subset of compressed tasks (that is, the set of tasks with $T_i > T_{i_0}$), let Γ_a be the set of remaining tasks in Γ (that is, the set of tasks with $T_i = T_{i_0}$), and let U_d be the current processor utilization of Γ. Whenever a task in Γ_a voluntarily increases its period, all tasks in Γ_c can expand their utilizations according to their elastic coefficients, so that the processor utilization is kept at the value of U_d.

Now, let U_c be the total utilization of Γ_c, let U_a be the total utilization of Γ_a after some task has increased its period, and let U_{c_0} be the total utilization of tasks in Γ_c at their nominal periods. It can easily be seen that if $U_{c_0} + U_a \leq U_{lub}$ all tasks in Γ_c can return to their nominal periods. On the other hand, if $U_{c_0} + U_a > U_{lub}$, then the release operation of the tasks in Γ_c can be viewed as a compression, where $\Gamma_f = \Gamma_a$ and $\Gamma_v = \Gamma_c$. Hence, it can still be performed by using equations (2.17), (2.19) and (2.20) and the algorithm presented in Figure 2.14.

PERIOD RESCALING

If the elastic coefficients are set equal to task nominal utilizations, elastic compression has the effect of a simple rescaling, where all the periods are increased by the same percentage. In order to work correctly, however, period rescaling must be uniformly applied to all the tasks, without restrictions on the maximum period. This means having $U_f = 0$ and $U_{v_0} = U_0$. Under this assumption, by setting $E_i = U_{i_0}$, equations (2.17) become:

$$\forall i \quad U_i = U_{i_0} - (U_0 - U_d)\frac{U_{i_0}}{U_0} = \frac{U_{i_0}}{U_0}[U_0 - (U_0 - U_d)] = \frac{U_{i_0}}{U_0}U_d$$

from which we have that

$$T_i = T_{i_0}\frac{U_0}{U_d}.$$

[1] The actual implementation of the algorithm contains more checks on tasks' variables, which are not shown here to simplify its description.

Algorithm Task_compress(Γ, U_d) {

$U_0 = \sum_{i=1}^{n} C_i/T_{i_0}$;
$U_{min} = \sum_{i=1}^{n} C_i/T_{i_{max}}$;
if ($U_d < U_{min}$) return INFEASIBLE;

do {

$U_f = U_{v_0} = E_v = 0$;
for ($each\ \tau_i$) {
if (($E_i == 0$) or ($T_i == T_{i_{max}}$))
$U_f = U_f + U_{i_{min}}$;
else {
$E_v = E_v + E_i$;
$U_{v_0} = U_{v_0} + U_{i_0}$
}
}

$ok\ =\ 1$;
for ($each\ \tau_i \in \Gamma_v$) {
if (($E_i > 0$) and ($T_i < T_{i_{max}}$)) {
$U_i = U_{i_0} - (U_{v_0} - U_d + U_f)E_i/E_v$;
$T_i = C_i/U_i$;
if ($T_i > T_{i_{max}}$) {
$T_i = T_{i_{max}}$;
$ok\ =\ 0$;
}
}
}

} **while** ($ok\ ==\ 0$);
return FEASIBLE;
}

Figure 2.14 Algorithm for compressing a set of elastic tasks.

This means that in overload situations ($U_0 > 1$) the compression algorithm causes all task periods to be increased by a common scale factor

$$\eta = \frac{U_0}{U_d}.$$

Notice that, after compression is performed, the total processor utilization becomes:

$$U = \sum_{i=1}^{n} \frac{C_i}{\eta T_{i_0}} = \frac{1}{\eta} U_0 = \frac{U_d}{U_0} U_0 = U_d$$

as desired.

If a maximum period needs to be defined for some task, an on-line guarantee test can easily be performed before compression to check whether all the new periods are less than or equal to the maximum value. This can be done in $O(n)$ by testing whether

$$\forall i = 1, \ldots, n \quad \eta T_{i_0} \leq T_i^{max}.$$

By deciding to apply period rescaling, we loose the freedom of choosing the elastic coefficients, since they must be set equal to task nominal utilizations. However, this technique has the advantage of leaving the task periods ordered as in the nominal configuration, which simplifies the compression algorithm in the presence of resource constraints.

CONCLUDING REMARKS

The elastic model offers a flexible way to handle overload conditions. In fact, whenever a new task cannot be guaranteed by the system, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request. As soon as a transient overload condition is over (because a task terminates or voluntarily increases its period) all the compressed tasks may expand up to their original utilization, eventually recovering their nominal periods.

The major advantage of the elastic method is that the policy for selecting a solution is implicitly encoded in the elastic coefficients provided by the user (for example, based on task importance). Each task is varied based on its current elastic status and a feasible configuration is found, if there exists one.

The elastic model is extremely useful for supporting both multimedia systems and control applications, in which the execution rates of some computational activities have to be dynamically tuned as a function of the current system state. Furthermore, the

elastic mechanism can easily be implemented on top of classical real-time kernels, and can be used under fixed or dynamic priority scheduling algorithms [But93a, LLB +97].

It is worth observing that the elastic approach is not limited to task scheduling. Rather, it represents a general resource allocation methodology which can be applied whenever a resource has to be allocated to objects whose constraints allow a certain degree of flexibility. For example, in a distributed system, dynamic changes in node transmission rates over the network could be efficiently handled by assigning each channel an elastic bandwidth, which could be tuned based on the actual network traffic. An application of the elastic model to the network has been proposed in [PGBA02].

Another interesting application of the elastic approach is to automatically adapt the task rates to the current load, without specifying the worst-case execution times of the tasks. If the system is able to monitor the actual execution time of each job, such data can be used to compute the actual processor utilization. If this is less than one, task rates can be increased according to elastic coefficients to fully utilize the processor. On the other hand, if the actual processor utilization is a little grater than one and some deadline misses are detected, task rates can be reduced to bring the processor utilization at a desired safe value.

The elastic model has also been extended in [BLCA02] to deal with resource constraints, thus allowing tasks to interact through shared memory buffers. In order to estimate maximum blocking times due to mutual exclusion and analyze task schedulability, critical sections are assumed to be accessed through the Stack Resource Policy [Bak91].

3

TEMPORAL PROTECTION

In critical real-time applications, where predictability is the main goal of the system, traditional real-time scheduling theory can be successfully used to verify the feasibility of the schedule under worst-case scenarios. However, when efficiency becomes relevant and when the worst-case parameters of the tasks are too pessimistic or unknown, the hard real-time approach presents some problems. In particular, if a task overruns the system can experience a temporary or permanent overload, and, as a consequence, some task can miss its deadline.

In the previous chapter, we presented techniques to deal with permanent overload, in which some high level decision must be taken to remove the overload. In this chapter, we will present techniques to deal with tasks with transient overruns. Such techniques provide the "temporal protection" property: if a task overruns, only the task itself will suffer the possible consequences. In this way, we "confine" the effect of a overrun so that each task can be analyzed in isolation.

After an introduction to the problem, we will present two different classes of algorithms for providing temporal protection: algorithms based on the fairness property, often referred to as proportional share algorithms, and algorithms based on resource reservation. Finally, we will describe some operating systems that provides resource reservation mechanisms.

3.1 PROBLEMS WITHOUT TEMPORAL PROTECTION

To introduce the problems that can occur when using traditional real-time algorithms for scheduling soft real-time tasks, consider a set of two periodic tasks τ_1 and τ_2, with

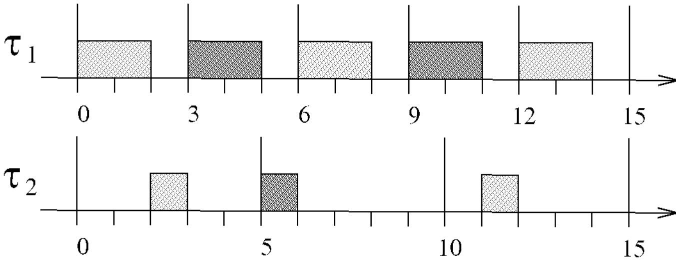


Figure 3.1 A task set schedulable under EDF.

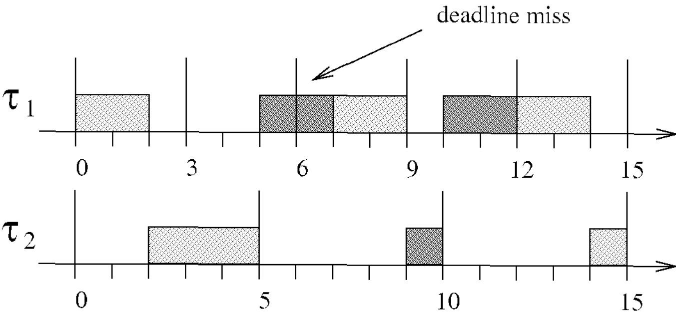


Figure 3.2 An instance of τ_2 executing for "too long" can cause a deadline miss in τ_1.

$T_1 = 3$ and $T_2 = 5$. If $C_1 = 2$ and $C_2 = 1$, the task set can be feasibly scheduled (in fact, $2/3 + 1/5 = 13/15 < 1$) and the schedule produced by EDF is illustrated in Figure 3.1. If, for some reason, the first instance of τ_2 increases its execution time to 3, then τ_1 misses its deadline, as shown in Figure 3.2. Notice that τ_1 is suffering for the misbehavior of another task (τ_2), although the two tasks are independent.

This problem does not specifically depend on EDF, but is inherent to all scheduling algorithms that rely on a guarantee based on worst-case execution times (WCETs). For instance, Figure 3.3 shows another example in which two tasks, $\tau_1 = (2, 3)$ and $\tau_2 = (1, 5)$, are feasibly scheduled by a fixed priority scheduler (where tasks have been assigned priorities based on the rate monotonic priority assignment). However, if the first instance of τ_1 increases its execution from 2 to 3 units of time, then the first instance of τ_2 will miss its deadline, as shown in Figure 3.4. Again, one task (τ_2) is suffering for the misbehavior of another task (τ_1).

Notice that, under fixed priority scheduling, a high priority task (τ_1 in the example) cannot be influenced by a lower priority task (τ_2). However, task priorities do not

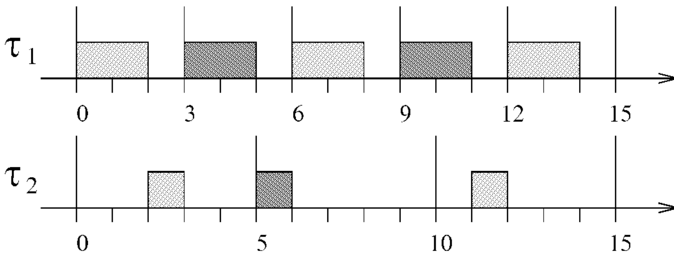


Figure 3.3 A task set schedulable under RM.

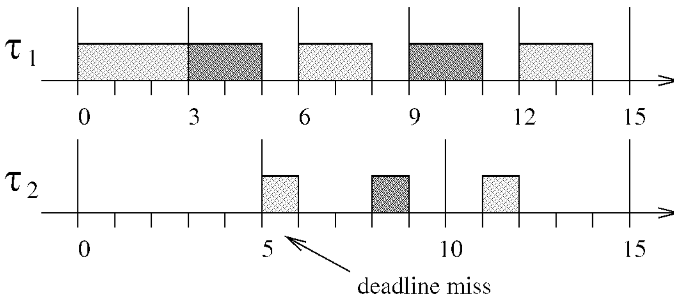


Figure 3.4 An instance of τ_1 executing for "too long" can cause a deadline miss in τ_2.

always reflect importance and are often assigned based on other considerations, like schedulability, as for the rate monotonic assignment. If importance values are not related with task rates, assigning priorities to tasks is not trivial, if a high schedulability bound has to be reached. For some specific task sets, schedulability can be increased by applying a period transformation technique [SG90], which basically splits a task with a long period into smaller subtasks with shorter periods. However, playing with priorities is not the best approach to follow, and the method becomes inefficient for large task sets with arbitrary periods.

The examples presented above show that when a real-time system includes tasks with variable (or unknown) parameters, some kind of *temporal protection* among tasks is desirable.

Definition 3.1 *The temporal protection property requires that the temporal behavior of a task is not affected by the temporal behavior of the other tasks running in the system.*

In a real-time system that provides temporal isolation, a task executing "too much" cannot cause the other tasks to miss their deadlines. For example, in the case illustrated in Figure 3.2, if temporal protection were enforced by the system, then the task missing the deadline would be τ_2.

Temporal protection (also referred to as *temporal isolation*, *temporal firewalling*, or *bandwidth isolation*) has the following advantages:

- it prevents an overrun occurring in a task to affect the temporal behavior of the other tasks;
- it allows partitioning the processor into tasks, so that each task can be guaranteed in "isolation" (that is, independently of the other tasks in the system) only based on the amount of processor utilization allocated to it;
- it provides different types of guarantee to different tasks, for example, a hard guarantee to a task and a probabilistic guarantee to another task;
- when applied to an aperiodic server, it protects hard tasks from the unpredictable behavior of aperiodic activities.

Another important thing to be noticed is that, if the system is in a permanent overload condition, some high level action must be taken to remove the overload. The techniques described in this chapter act at a lower level: they introduce temporal protection and allow the detection of the failing tasks. If the system detects that some task is in a permanent overrun condition, then some of the techniques presented in the previous chapter (for example the elastic task model, the RED algorithm, etc.) can be applied by either removing some task or by degrading the quality of the results of the application. Again, consider the example of Figure 3.4: if all instances of τ_1 execute for 3 instead of 2 units of time, the overload is permanent and would prevent the execution of τ_2. In this case, the overload could be removed for example by enlarging task periods according to the elastic task model, or rejecting some task based on some heuristics, or reducing the computation times by degrading the quality of results according to the imprecise computation model.

3.2 PROVIDING TEMPORAL PROTECTION

Several algorithms have been presented in the literature to provide some form of temporal protection, both in processor scheduling and in network scheduling [1]. To help

[1] In this chapter, for the sake of simplicity, we will use the terminology related to processor scheduling. Many of the properties and characteristics of the algorithms explained here are also applicable, with some difference, to the scheduling of other system resources. When necessary, we will specify the differences.

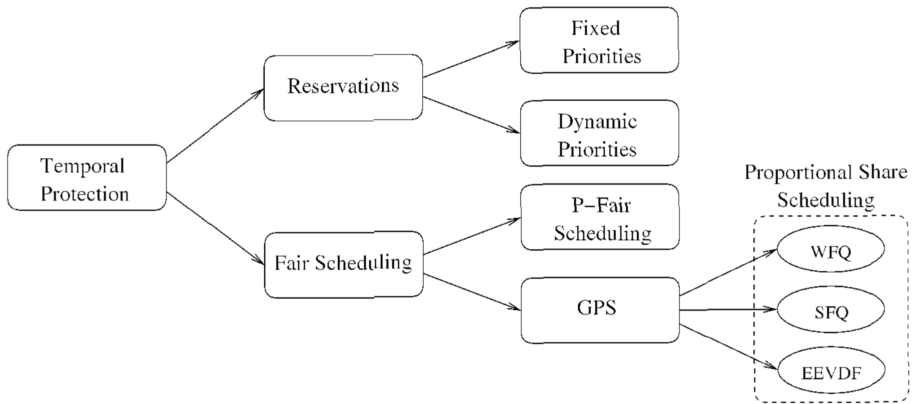


Figure 3.5 Schedulers providing temporal protection.

distinguishing the various algorithms and their characteristics, they have been categorized according to the taxonomy illustrated in Figure 3.5.

The temporal protection property is also referred to as "temporal isolation" property, since many authors stress the fact that each task is "isolated" from the others. In general, the idea is to assign each task a "share" of the processor, and the task is guaranteed to obtain at least its share.

The class of algorithms providing temporal protection can be divided in two main classes: the class of *fair scheduling algorithms* and the class of *resource reservation algorithms*.

3.2.1 FAIR SCHEDULING

The class of *fair share scheduling* algorithms is based on a theoretical model that assumes a fluid allocation of the resources. In some cases (like in P-Fair algorithms) each task is directly assigned a share of the processor, whereas in other cases (like in proportional share algorithms) each task is assigned a weight, from which a share is derived as *proportional* to the task weight.

In both cases, the objective is to allocate the processor so that in *every interval* of time each task precisely receives its share of the processor. Notice that such an objective cannot be realized in practice, since it would require a infinitely divisible resource: no matter how small the interval is, each task should receive its share of the processor. But the minimum time granularity of one processor is given by the clock! As

a consequence, any implementable algorithm can only approximate the ideal one. A theoretical algorithm based on the ideal fluid resource allocation model is the Generalized Processor Sharing (GPS), which will be presented in Section 3.3. The GPS is mainly used for evaluation purposes, to verify how closely an algorithm approximates the fluid model.

A parameter that can be used to measure how closely a realistic algorithm approximates the ideal one is the *lag*. For each task, the *lag* is defined as the difference between the execution time actually assigned to a task by the realistic algorithm and the amount of time assigned by the ideal fluid algorithm. Hence, the objective of a fair scheduler is to limit the lag to an interval as close as possible to 0.

Most of the algorithms belonging to this class divide the time line into intervals of fixed length, called "quantum", with the constraint that only one task per processor can be executed in one quantum. The idea is to approximate fluid allocation with small discrete intervals of time.

We can further divide the class of fair scheduling algorithms in p-fair scheduling and in proportional share algorithms. The main difference is on how the processor share is assigned to tasks.

In proportional share scheduling, each task is assigned a weight w_i, and it receives a share of the processor equal to:

$$F_i = \frac{w_i}{\sum_{i=1} N w_i}$$

where N is the number of tasks. If the number of tasks does not change during the system lifetime (i.e., not new tasks are allowed to dynamically join the system, nor tasks can leave the system), then the task share is a constant.

However, if tasks are allowed to dynamically join the system, task shares can change. If this change is not controlled, the temporal isolation property is broken: a new tasks joining the system can require a very high weight, reducing considerably the share of the existing tasks.

Therefore, if we want to provide temporal guarantees in proportional share schedulers, an admission control policy is needed to check whether after each insertion each task receives at least the desired share of the processor, and to "re-weight" the tasks in order to achieve the desired level of processor share. Proportional share algorithms can provide temporal protection only if complemented with proper admission control algorithms and re-weighting policies.

In p-fair scheduling, instead, each task is assigned a weight w_i, with $\sum_{i=0}^{N} w_i \leq M$, where N is the number of tasks and M is the number of processors. Since, the weights

are already normalized, each task receives a share of the system equal to its weight. Therefore, the admission control test is simply to check that the sum of all weights does not exceed the number of processors.

Proportional share algorithms were initially presented in the context of network scheduling, where the concept of task is substituted with the concept of packets "flow". A network link is shared among different flows, each flow is assigned a weight and the goal is to allocate the bandwidth of the link to the different flows in a fair manner, so that each flow receives a share proportional to its weight.

The same algorithms have also been applied to the context of processor scheduling. One difference between network scheduling and processor scheduling is that in network scheduling the basic scheduling unit is the packet. In fact, the packet must be transmitted entirely, and cannot be divided into smaller units. Hence, there is no need for specifying a "scheduling quantum": the length of the packet is itself the scheduling quantum. The problem becomes slightly more complex if packets have different lengths.

In Section 3.3, we present some of the most popular fair scheduling algorithms in the context of processor scheduling.

3.2.2 RESOURCE RESERVATION

The class of resource reservation algorithms consists of algorithms derived from classical real-time scheduling theory. The first algorithms, generically called "aperiodic servers", were proposed to schedule aperiodic soft real-time tasks together with periodic hard real-time tasks. The goal was to minimize the response time of aperiodic tasks without jeopardizing the hard real-time tasks.

Aperiodic server algorithms were proposed both for fixed priority scheduling and dynamic priority scheduling. In fixed priority scheduling, the main algorithms are the Polling Server, the Deferrable Server (DS) and the Sporadic Server (SS) [SSL89, LSS87, SLS95]. In dynamic priority scheduling, the most important algorithms are the Total Bandwidth Server (TBS) [SB94, SB96] and the Constant Bandwidth Server (CBS) [AB98a, AB04].

An approach similar to the server algorithms was applied for the first time to soft real-time multimedia applications by Mercer et al. [MST94a], with the explicit purpose of providing temporal protection. Later, Rajkumar et al. [RJMO98] introduced the term "resource reservation" to indicate this class of techniques.

In all the previous cited algorithms (with the exception of the TBS), a server is characterized by a budget Q and a period P. The processor share assigned to each server

is Q/P. In the original formulation of these algorithms, one server was defined for the entire system, with the purpose of serving all aperiodic tasks in First-Come-First-Served (FCFS) order. The behavior of the server is similar to that of a periodic hard real-time task with a worst-case execution time equal to the assigned budget Q and a period equal to P. Hence, it is possible to apply the existing real-time scheduling analysis techniques to check the schedulability of the system.

Resource reservation algorithms provide the temporal protection property. In one possible configuration, every task in the system (periodic or aperiodic, hard or soft real-time) is assigned a dedicated server with a share Q_i/P_i, under the constraint that:

$$\sum_{i=1}^{N} \frac{Q_i}{P_i} \leq U_{lub}$$

where N is the number of tasks in the system and U_{lub} is the schedulability utilization bound, which depends on the adopted scheduling algorithm. Then, each task is guaranteed to obtain a budget Q_i every server period P_i, regardless of the behavior of the other tasks in the system.

It is important to note that in the configuration "one server per task", the assumption of periodic or sporadic tasks can be removed. For example, consider a non-real-time non-interactive task (like for example a complex scientific computation or the compilation of a large program). By assigning a server with a certain budget and a period to this task, it will receive a steady and regular allocation of the processor, independently of the presence of other (real-time or non-real-time) tasks in the system.

Resource reservation techniques will be described in detail in Section 3.5, and the Constant Bandwidth Server (CBS) [AB98a, AB04] will be presented in Section 3.6.1. Before continuing the presentation of the different approaches to temporal protection, it is important to highlight the main differences between fair scheduling and resource reservation techniques.

The main objective of a fair scheduler is to keep the lag between the task execution and the ideal fluid allocation as close as possible to zero. For this reason, in processor scheduling, these algorithms need to introduce the concept of "scheduling quantum" that is the basic unit of allocation. The smaller the quantum, the smaller the lag bound. However, a small quantum implies a large number of context switches. Moreover, once the scheduling quantum has been fixed for the entire system, each task is assigned one single parameter, the weight (or the share in p-fair schedulers). The "granularity" of the allocation depends on the scheduling quantum while the share of the processor depends on the task weight. Therefore, if a task requires a very small granularity, we must reduce the scheduling quantum, causing a large number of context switches and more overhead.

Conversely, the goal of a resource reservation algorithm is to keep resource allocation under control so that a task can meet its timing constraints. To this end, each reservation is associated with two parameters, the budget Q and the period P. The period of the reservation represents the granularity of the allocation needed by the corresponding task, while the rate Q/P represents the share of the processor. Therefore, unlike fair schedulers, it is possible to select the most appropriate granularity for each task. If a task requires a very small granularity, its reservation period must be reduced accordingly, while the other reservations can keep a large period. In the general case, it is possible to show that, the number of context switches produced by a reservation scheduler is considerably less than the number of context switches produced by a proportional share scheduler.

3.3 THE GPS MODEL

As explained above, temporal protection can be provided by adding an admission control mechanism to a fair scheduler. In fact, if a real-time task is assigned a sufficient amount of resources, it can execute at a constant rate, while respecting its timing constraints.

Executing each task τ_i at a constant rate is the essence of the Generalized Processor Sharing (GPS) approach [PG93, PG94]. In this model, each shared resource needed by tasks (such as the CPU) is considered as a fluid that can be partitioned among the applications. Each task instantaneously receives a fraction $f_i(t)$ of the resource at time t, where $f_i(t)$ is defined as the task *share*. Note that the GPS model can be seen as an extreme form of a Weighted Round Robin policy.

To compute the share of a resource, each task τ_i is assigned a weight w_i, and its share is computed as

$$f_i(t) = \frac{w_i}{\sum_{\tau_j \in \Gamma(t)} w_j}$$

where $\Gamma(t)$ is the set of tasks that are active at time t.

Since each task consists of one or more requests for shared resources, tasks can block and unblock, and the $\Gamma(t)$ set can vary with time. Hence, the share $f_i(t)$ is a time varying quantity. The minimum guaranteed share is defined as the *rate*

$$F_i = \frac{w_i}{\sum_{\tau_j \in \Gamma} w_j},$$

where Γ is the set of all tasks in the system.

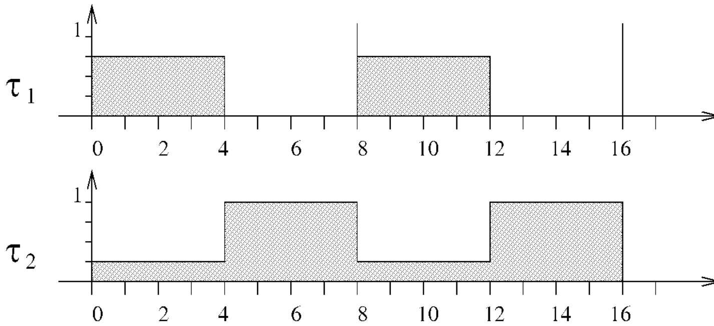


Figure 3.6 Ideal schedule of two GPS tasks. The height of task executions is proportional to CPU speed.

If an appropriate admission control is performed, it is possible to find an assignment of weights to tasks to guarantee real-time performance to all the time sensitive applications. In fact, based on the task rate, the maximum response time for each task can be computed as C_i/F_i. The problem with the GPS model is that the task response time C_i/F_i and the task throughput F_i are not independent (using real-time terminology, this means that the relative deadline of a task is implicitly equal to its period).

The GPS model describes a task system as a fluid flow system, in which each task τ_i is modeled as an infinitely divisible fluid, and executes at a minimum rate F_i that is proportional to a user specified weight w_i. For example, Figure 3.6 shows the ideal schedule of 2 GPS tasks, τ_1 and τ_2, with weights $w_1 = 3$ and $w_2 = 1$. Note that τ_2 is always active, whereas τ_1 is a periodic task with period $T_1 = 8$ and execution time $C_1 = 3$. At time $t = 0$, both tasks are active, hence they receives two shares $f_1(0) = 3/(1+3) = 3/4$ and $f_2(0) = 1/(1+3) = 1/4$. This means that the two tasks execute simultaneously, and τ_1 executes at $3/4$ of the CPU speed, whereas τ_2 executes at $1/4$ of the CPU speed. As a result, the first instance of τ_1 finishes at time $C_1/f_1(0) = 3/(3/4) = 4$, when τ_2 remains the only active task in the system and receives a share $f_2(4) = 1$. At time 8, τ_1 activates again and the schedule repeats as at time 0. Note that the schedule represented in Figure 3.6 cannot be realized in practice, because tasks execute simultaneously.

According to the ideal GPS model, task τ_i is guaranteed to execute for an amount of time $s_i(t_1, t_2) > (t_2 - t_1)F_i$ in each backlogged interval $[t_1, t_2]$. More precisely, the amount of time s_i executed by task τ_i in an ideal GPS system is:

$$s_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(t)dt.$$

As a result, in the ideal fluid flow model, tasks' execution can be described through the following **GPS guarantee**:

$$\forall \tau_i \text{ active in } [t_1, t_2], \frac{exec_i(t_1, t_2)}{exec_j(t_1, t_2)} \geq \frac{w_i}{w_j} \; j = 1, 2, ..., n \tag{3.1}$$

where $exec_i(t_1, t_2)$ is the amount of time actually executed by τ_i in the interval $[t_1, t_2]$. It can be easily seen that Equation 3.1 is equivalent to $exec_i(t_1, t_2) = s_i(t_1, t_2)$.

3.4 PROPORTIONAL SHARE SCHEDULING

Although the ideal GPS schedule cannot be realized on a real system, it can be used as a reference model to compare the performance of practical algorithms that attempt to approximate its behavior. In a real system, resources must be allocated in discrete time quanta of size Q, and such a quantum-based allocation causes an allocation error. Given two active tasks τ_1 and τ_2, the allocation error in the time interval $[t_1, t_2]$ can be expressed as

$$\frac{exec_i(t_1, t_2)}{w_i} - \frac{exec_j(t_1, t_2)}{w_j}.$$

An alternative way to express the allocation error is the *maximum lag*:

$$Lag_i = \max_{t_1, t_2} \{|exec_i(t_1, t_2) - s_i(t_1, t_2)|\}.$$

Hence, a more realistic version of the GPS guarantee is the following:

$$|exec_i(t_1, t_2) - \int_{t_1}^{t_2} f_i(t)dt| \leq Lag_i$$

Proportional Share (PS) scheduling was originally developed for handling network packets. It provides fairness among different streams by emulating the GPS allocation model in a real system, where multiple tasks do not run simultaneously on the same CPU, but are executed using a quantum-based allocation. In other words, in a Proportional Share scheduler, resources are allocated in discrete time quanta having maximum size Q: a process acquires a resource at the beginning of a time quantum and releases the resource at the end of the quantum. To do that, each task τ_i is divided in requests q_i^k of size Q.

Clearly, quantum-based allocation introduces an allocation error with respect to the fluid flow model. The minimum theoretical error bound is $H_{i,j} = \frac{1}{2}(\frac{Q_i}{w_i} + \frac{Q_j}{w_j})$, where

Q_i is the maximum dimension for τ_i requests and Q_j is the maximum dimension for τ_j requests.

An important properties of PS schedulers (that directly derives from the GPS definition) is that they are *work conserving algorithms.*

Definition 3.2 *An algorithm is said to be work conserving if it ensures that the CPU is not idle when there are jobs ready to execute.*

As we will see in the next sections, some algorithms providing temporal protection are not work conserving (for example, hard reservation algorithms).

In the rest of this section, some of the most important PS scheduling algorithms are analyzed, showing how they emulate the ideal GPS allocation, and evaluating their performance in terms of allocation error and lag.

3.4.1 WEIGHTED FAIR QUEUING

The first known Proportional Share scheduling algorithm is Weighted Fair Queuing (WFQ), which emulates the behavior of a GPS system by using the concept of *virtual time.* The virtual time $v(t)$ is defined by increments as follows:

$$\begin{cases} v(0) & = & 0 \\ dv(t) & = & \frac{1}{\sum_{\tau_i \in \Gamma(t)} w_i} dt \end{cases}.$$

Each quantum request q_i^k is assigned a virtual start time $S(q_i^k)$ and a virtual finish time $F(q_i^k)$ as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), F(q_i^{k-1})\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i} \end{aligned}$$

where $r_{i,k}$ is the time at which request q_i^k is generated and $Q_{i,k}$ is the request dimension (required execution time). Since $Q_{i,k}$ is not known a priori (a task may release the CPU before the end of the time quantum), it is assumed to be equal to the quantum size Q (note that the quantum size is the same for all the tasks, hence the i index can be removed). Tasks' requests are scheduled in order of increasing virtual finish time, and the definitions presented above guarantee that each request completes before its virtual finishing time.

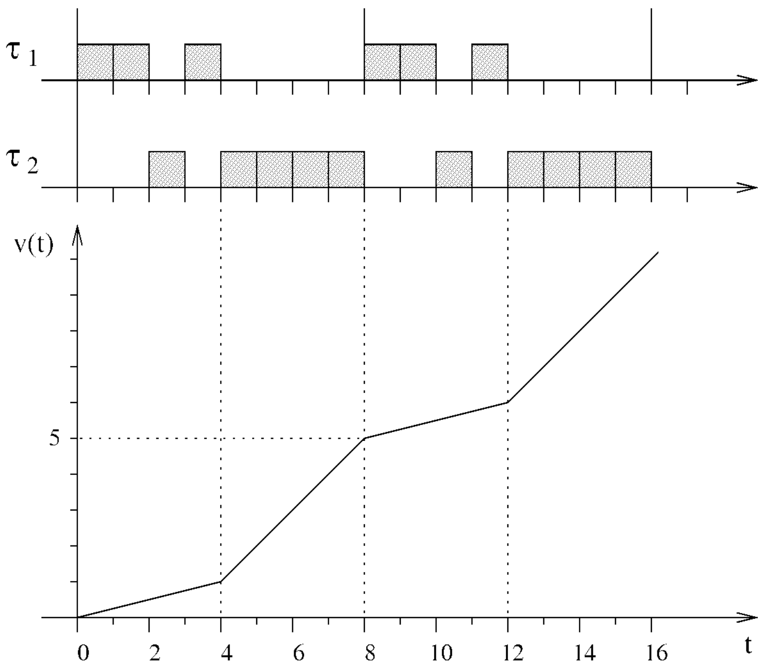


Figure 3.7 WFQ schedule generated by the task set of Figure 3.6.

Figure 3.7 shows an example of WFQ scheduling, with the same task set presented in Figure 3.6 and considering a quantum size $Q = 1$. The first quantum begins at time 0, hence its virtual start time is 0 for both tasks. Since the virtual finishing time of the first quantum is $0 + 1/3 = 1/3$ for task τ_1, and $0 + 1/1 = 1$ for task τ_2, such a quantum is assigned to τ_1. The virtual start time of the second quantum of task τ_1 is $\max\{1/4, 1/3\} = 1/3$, hence $F(q_1^2) = 1/3 + 1/3 = 2/3$ and τ_1 is scheduled again. In the same way, $S(q_1^3) = \max\{1/2, 2/3\} = 2/3$, and $F(q_1^3) = 1$. Since the virtual finishing time of the two tasks is the same, both τ_1 and τ_2 can be scheduled at time $t = 2$: let us assume that τ_2 is scheduled. As a result, $S(q_2^2) = \max\{1, 1/4\} = 1$ and $F(q_2^2) = 2$. Since $F(q_1^3) < F(q_2^2)$, τ_1 is scheduled at time $t = 3$ and finishes its first instance at time $t = 4$. At this point, the virtual time changes its increase rate to reflect the fact that τ_2 remains the only active task in the system ($w_2 = 1 \Rightarrow dv(t) = dt$). As a result, when τ_1 activates again at time $t = 8$, the virtual time $v(8) = 5$ is equal to the virtual finishing time $F(q_2^5) = 5$ of the latest quantum executed by τ_2. Hence, the virtual start time of the two competing quanta of τ_1 and τ_2 is the same (5), and the schedule repeats as at time 0.

The WFQ algorithm is one of the first known PS schedulers, and it is the basis for all the other PS algorithms. In fact, most of the PS schedulers are just modifications of WFQ that try to solve some of its problems. Some of the most notable problems presented by WFQ are:

- it needs a frequent recalculation of $v(t)$;
- it does not perform well in dynamic systems (when a task activates or blocks, the fairness of the schedule is compromised);
- it assumes each requests size equal to the maximum value (the scheduling quantum); in real situations this assumption is not correct.

In general, the main difference among the various PS schedulers consists in the way they define the virtual time, or in some additional rule that can be used to increase the fairness in some pathological situations.

3.4.2 START FAIR QUEUING

Start Fair Queuing (SFQ) [GGV96] is a proportional share scheduler that reduces the computational complexity of WFQ and increases the fairness by using a simpler definition of virtual time. The algorithm has been designed to hierarchically subdivide the CPU bandwidth among various application classes. Another difference with WFQ is that SFQ schedules the requests in order of increasing virtual start time.

The SFQ algorithm defines the virtual time $v(t)$ as follows:

$$v(t) = \begin{cases} 0 & \text{if } t = 0 \\ 0 \text{ or any value} & \text{if the CPU is idle} \\ S(q_i^k) & \text{if request } q_i^k \text{ is executing} \end{cases}$$

SFQ guarantees an allocation error bound of $2H_{i,j}$, so it is nearly-optimal. Moreover, SFQ calculates $v(t)$ in a way simpler than that used in WFQ (introducing less overhead) and does not need the virtual finish time of a request to schedule it, so it does not require any a priori knowledge of the request execution time ($F(q_i^k)$ can be computed at the end of q_i^k execution).

A Proportional Share algorithm schedules the tasks in order to reduce the allocation error experienced by each of them; to provide some form of real-time execution it is important to guarantee that $lag_i(t)$ is bounded.

SFQ and WFQ provide an optimal upper bound for the lag ($\max_t\{lag_i(t)\} = Q_i$), but do not provide an optimal bound for the absolute value of the lag. For example, for SFQ this bound is $\max_t\{|lag_i(t)|\} = Q_i + f_i \sum Q_j$, which depends on the number of active tasks.

3.4.3 EARLIEST ELIGIBLE VIRTUAL DEADLINE FIRST

In [SAWJ+96] the authors propose a scheduling algorithm, called Earliest Eligible Deadline First (EEVDF), that provides an optimal bound on the lag experienced by each task.

EEVDF defines the virtual time as WFQ and schedules the requests by virtual finish times (in this case called virtual deadlines), but uses the virtual start time (called virtual eligible time) to decide whether a task is eligible to be scheduled: if the virtual eligible time is greater than the actual virtual time, the request is not eligible. Virtual eligible and finish time are defined as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), S(q_i^{k-1}) + \frac{Q_{i,k-1}}{w_i}\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i}. \end{aligned}$$

When a task joins or leaves the competition (activates or blocks), $v(t)$ is adjusted in order to maintain the fairness in a dynamic system.

It can be proved that, although the EEVDF algorithm uses the concept of eligible time, it is still a work conserving algorithm (in other words, if there is at least a ready task in the system, then there is at least an eligible task).

The minimum theoretical bound guaranteed by EEVDF for the absolute value of the lag is Q; for this reason, EEVDF is said to be optimal. EEVDF can also schedule dynamic task sets and can use non uniform quantum sizes, so it can be used in a real operating system. To the best knowledge of the authors, EEVDF is the only algorithm that provides a fixed lag bound. If the lag is bounded, real-time execution can be guaranteed by maintaining the share of each real-time task constant:

$$f_i(t) = \frac{C_i + \max_t\{lag_i(t)\}}{D_i}.$$

3.5 RESOURCE RESERVATION TECHNIQUES

A simple and effective mechanism for implementing temporal protection in a real-time system is to reserve each task τ_i a specified amount of CPU time Q_i in every interval P_i. Such a general approach can also be applied to other resources different than the CPU, but in this context we will mainly focus on the CPU, because CPU scheduling is the topic of this book.

Some authors [RJMO98] tend to distinguish between *hard* and *soft* reservations.

Definition 3.3 *A hard reservation is an abstraction that guarantees the reserved amount of time to the served task, but allows such task to execute* at most *for Q_i units of time every P_i.*

Definition 3.4 *A soft reservation is a reservation guaranteeing that the task executes* at least *for Q_i time units every P_i, allowing it to execute more if there is some idle time available.*

A resource reservation technique for fixed priority scheduling was first presented in [MST94a]. According to this method, a task τ_i is first assigned a pair (Q_i, P_i) (denoted as a CPU *capacity reserve*) and then it is enabled to execute as a real-time task for Q_i units of time every P_i. When the task consumes its reserved quantum Q_i, it is blocked until the next period, if the reservation is hard, or it is scheduled in background as a non real-time task, if the reservation is soft. At the beginning of the next period, the

task is assigned another time quantum Q_i and it is scheduled as a real-time task until the budget expires.

In this way, a task is *reshaped* so that it behaves like a periodic real-time task with known parameters (Q_i, P_i) and can be properly scheduled by a classical real-time scheduler. A similar technique is used in computer networks by traffic shapers, such as the leaky bucket or the token bucket [Tan96].

More formally, a reservation technique can be defined as follows:

- a reservation RSV is characterized by two parameters (Q, P), referred to as the *maximum budget* and the *reservation period*;
- a *budget* q (also referred to as *capacity*), is associated with each reservation;
- at the beginning of each reservation period, the budget q is recharged to Q;
- when the reserved task executes, the budget is decreased accordingly;
- when the budget becomes zero, the reservation is said to be *depleted* and an appropriate action must be taken.

The action to be taken when a reservation is depleted depends on the reservation type (hard or soft). In a hard reservation, the task is suspended until the budget is recharged, and another task can be executed. If all tasks are suspended, the system remains idle until the first recharging event. Thus, in case of hard reservations, the scheduling algorithm is said to be "non work-conserving".

In a soft reservation, if the budget is depleted and the task has not yet completed, the task's priority is downgraded to background priority until the budget is recharged. In this way, the task can take advantage of unused bandwidth in the system. When all reservations are soft, the algorithm is work-conserving.

Figure 3.8 shows how the tasks of Figure 3.2 are scheduled using two hard CPU reservations RSV_1 and RSV_2 with $Q_1 = 2$, $P_1 = 3$, $Q_2 = 1$, and $P_2 = 5$, under RM. The same figure also shows the temporal evolution of the budgets q_1 and q_2. Since the reservations are based on RM, RSV_1 has priority over RSV_2, and task τ_1 starts to execute. After 2 time units, τ_1 completes and τ_2 starts executing. At time 3, τ_2 has not completed, but its current budget $q_2 = 0$ and the reservation RSV_2 is depleted. Hence, τ_2 is suspended waiting for its budget to be recharged. As we can see, τ_1 does not suffer from the overrun of τ_2.

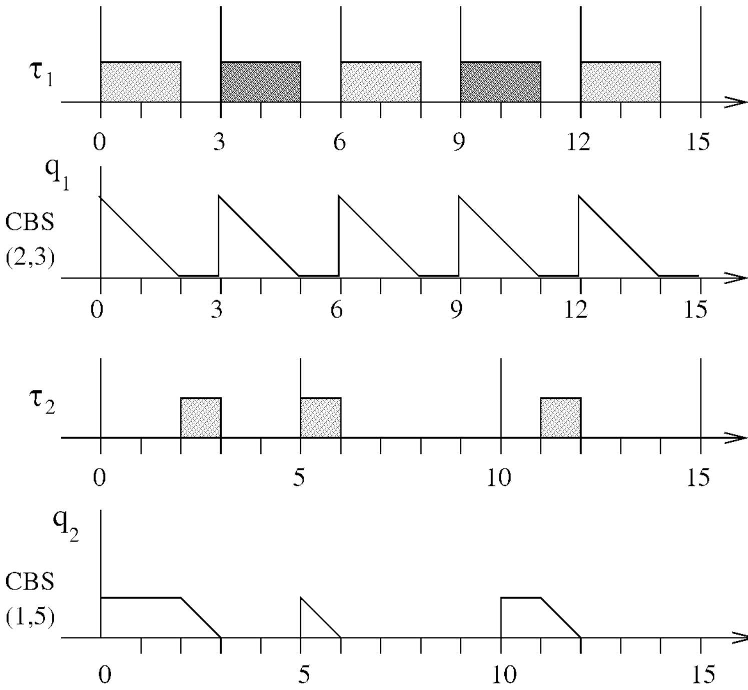


Figure 3.8 Example of CPU Reservations implemented over a fixed priority scheduler.

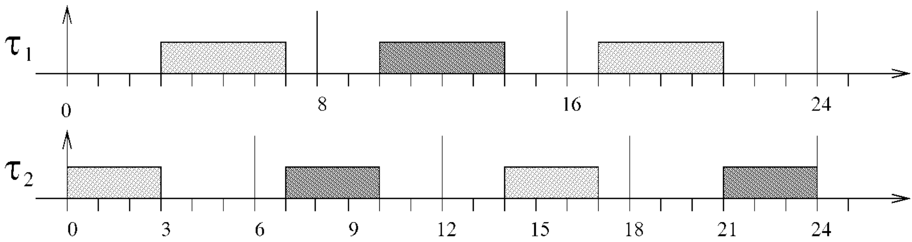


Figure 3.9 The task set is schedulable by CPU Reservations implemented over EDF.

At the same time, a new period for RSV_1 is activated, and budget q_1 is recharged to 2. Hence, τ_1 can execute again and complete its instance after one more unit of time. Notice that task τ_1 has missed its deadline at time 3. Moreover, since the task has a period of 3, at time 3 another instance should have been activated. Depending on the actual implementation of the scheduler and of the task, it may happen that the task activation at time 3 is skipped or buffered. In Figure 3.8 we assume that the task activation is buffered. Hence, at time 4 the task resumes executing the next buffered instance.

Note that, even if the first instance of τ_1 is "too long", the schedule is equivalent to the one generated by RM for two tasks $\tau_1 = (2, 3)$ and $\tau_2 = (1, 5)$. In other words, the CPU reservation mechanism provides temporal isolation between the two tasks: since τ_1 is the one executing "too much", it will miss some deadlines, but τ_2 is not affected.

3.5.1 PROBLEMS WITH TRADITIONAL RESERVATION SYSTEMS

The reservation mechanism presented in the previous section can be easily used also with dynamic priority schedulers, such as EDF, to obtain a better CPU utilization. However, when using CPU reservations in a dynamic priority system it can be useful to extend the scheduler to solve some problems that generally affect reservation based schedulers. Hence, before presenting some more advanced reservation algorithms, we show a typical problem encountered in traditional reservation systems.

In particular, a generic reservation based scheduling algorithm can have problems in handling aperiodic task's arrivals. Consider two tasks $\tau_1 = (4, 8)$ and $\tau_2 = (3, 6)$ served by two reservations $RSV_1 = (4, 8)$, and $RSV_2 = (3, 6)$. As shown in of Figure 3.9, if the EDF priority assignment is used to implement the reservation scheme, then the task set is schedulable (and each task will respect all its deadlines). However,

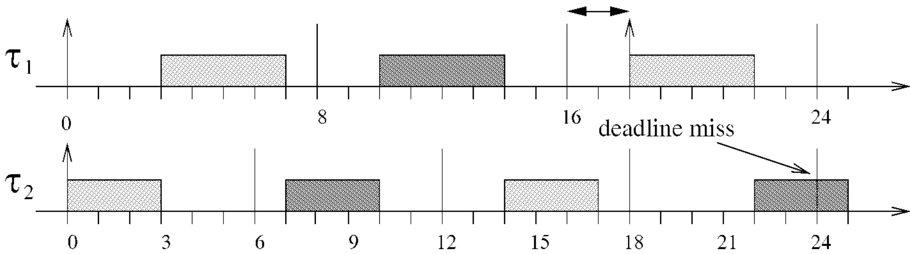


Figure 3.10 A late arrival in τ_2 can cause a deadline miss in τ_1.

if an instance of one of the two tasks is activated later, the temporal isolation provided by the reservation mechanism may be broken. For example, Figure 3.10 shows the schedule produced when the third instance of τ_1 arrives at time 18 instead of time 16: the system is idle between time 17 and 18, and task τ_2 (which is behaving correctly) misses a deadline.

If correctly used, dynamic priorities permit to fix this kind of problems and better exploit the CPU time, as shown in the next section.

3.6 RESOURCE RESERVATIONS IN DYNAMIC PRIORITY SYSTEMS

To better exploit the advantages of a dynamic priority system, resource reservations can be implemented by properly assigning a dynamic *scheduling deadline* to each task and by scheduling tasks by EDF based on their scheduling deadlines.

Definition 3.5 *A scheduling deadline $d^s_{i,j}$ is a dynamic deadline assigned to a job $\tau_{i,j}$ in order to schedule it by EDF.*

Note that a scheduling deadline is something different from the job deadline $d_{i,j}$, which in this case is only used for performance monitoring.

The abstract entity that is responsible for assigning a correct scheduling deadline to each job is called **aperiodic server**.

Definition 3.6 *A server is a mechanism used to assign scheduling deadlines to jobs in order to schedule them so that some properties (such as the reservation guarantee) are respected.*

Aperiodic servers are widely known in the real-time literature, [SSL89, LSS87, LRT92, TL92, SLS95, SB94, SB96, GB95], but, in general, they have been used to reduce the response time of aperiodic requests, and not to implement temporal protection.

The server assigns each job $\tau_{i,j}$ an absolute time-varying deadline $d^s_{i,j}$ which can be dynamically changed. This fact can be modeled by splitting each job $\tau_{i,j}$ into *chunks* $H_{i,j,k}$, each having a fixed scheduling deadline $d^s_{i,j,k}$.

Definition 3.7 *A* **chunk** $H_{i,j,k}$ *is a part of the job* $\tau_{i,j}$ *characterized by a fixed scheduling deadline* $d^s_{i,j,k}$. *Each chunk* $H_{i,j,k}$ *is characterized by an arrival time* $a_{i,j,k}$, *an execution time* $e_{i,j,k}$ *and a scheduling deadline. Note that the arrival time* $a_{i,j,0}$ *of the first chunk of a job* $\tau_{i,j}$ *is equal to the job release time:* $a_{i,j,0} = r_{i,j}$.

3.6.1 THE CONSTANT BANDWIDTH SERVER

The Constant Bandwidth Server (CBS) is a work conserving server (implementing soft reservations) that takes advantage of dynamic priorities to properly serve aperiodic requests and better exploit the CPU.

The CBS algorithm is formally defined as follows:

- A CBS S is characterized by a budget q^s and by a ordered pair (Q^s, P^s), where Q^s is the *server maximum budget* and P^s is the *server period*. The ratio $U^s = Q^s/P^s$ is denoted as the *server bandwidth*. At each instant, a fixed deadline d^s_k is associated with the server. At the beginning $d^s_0 = 0$.

- Each served job $\tau_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline d^s_k.

- Whenever a served job $\tau_{i,j}$ executes, the budget q^s of the server S serving τ_i is decreased by the same amount.

- When $q^s = 0$, the server budget is recharged at the maximum value Q^s and a new server deadline is generated as $d^s_{k+1} = d^s_k + P^s$. Notice that there are no finite intervals of time in which the budget is equal to zero.

- A CBS is said to be active at time t if there are pending jobs (remember the budget q^s is always greater than 0); that is, if there exists a served job $\tau_{i,j}$ such that $r_{i,j} \leq t < f_{i,j}$. A CBS is said to be idle at time t if it is not active.

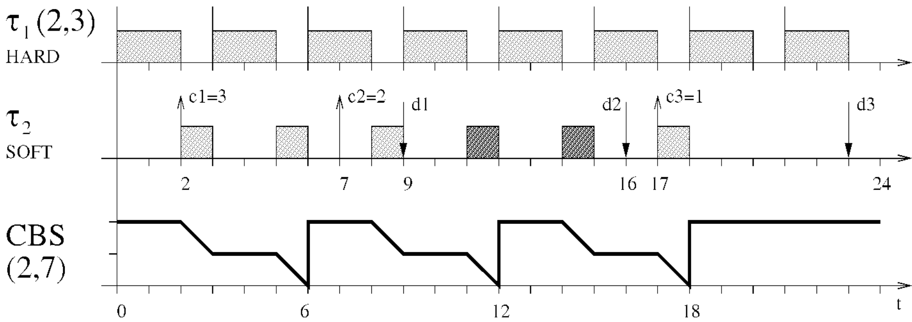


Figure 3.11 Simple example of CBS scheduling.

- When a job $\tau_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) non-preemptive discipline (e.g., FIFO, shortest execution time first, or earliest deadline first, if tasks have soft deadlines).
- When a job $\tau_{i,j}$ arrives and the server is idle, if $q^s \geq (d_k^s - r_{i,j})U^s$ the server generates a new deadline $d_{k+1}^s = r_{i,j} + P^s$ and q^s is recharged to the maximum value Q^s, otherwise the job is served with the last server deadline d_k^s using the current budget.
- When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.
- At any instant, a job is assigned the last deadline generated by the server.

Figure 3.11 illustrates an example in which a hard periodic task τ_1 is scheduled by EDF together with a soft task τ_2, served by a CBS having a budget $Q^s = 2$ and a period $P^s = 7$. The first job of τ_2 arrives at time $r_1 = 2$, when the server is idle. Being $q^s \geq (d_0^s - r_1)U^s$, the deadline assigned to the job is $d_1^s = r_1 + P^s = 9$ and q^s is recharged at $Q^s = 2$. At time $t_1 = 6$ the budget is exhausted, so a new deadline $d_2^s = d_1^s + P^s = 16$ is generated and q^s is replenished. At time $r_2 = 7$, the second job arrives when the server is active, so the request is enqueued. When the first job finishes, the second job is served with the actual server deadline ($d_2^s = 16$). At time $t_2 = 12$, the server budget is exhausted so a new server deadline $d_3^s = d_2^s + P^s = 23$ is generated and q^s is replenished to Q^s. The third job arrives at time 17, when the server is idle and $q^s = 1 < (d_3^s - r_3)U^s = (23 - 17)\frac{2}{7} = 1.71$, so it is scheduled with the actual server deadline d_3^s without changing the budget.

In Figure 3.12, a hard periodic task τ_1 is scheduled together with a soft task τ_2, having fixed inter-arrival time ($T_2 = 7$) and variable computation time, with a mean value equal

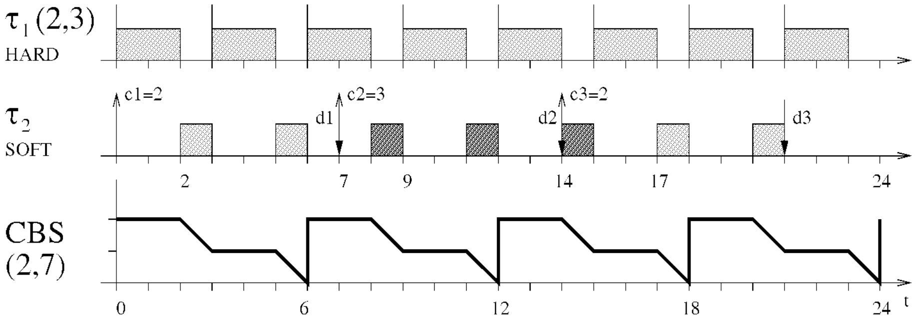


Figure 3.12 Example of CBS serving a task with variable execution time and constant inter-arrival time.

to $C_2 = 2$. This situation is typical in applications that manage continuous media: for example, a video stream requires to be played periodically, but the decoding/playing time of each frame is not constant. In order to optimize the processor utilization, τ_2 is served by a CBS with a maximum budget equal to the mean computation time of the task ($Q^s = 2$) and a period equal to the task period ($P^s = T_2 = 7$).

As we can see from Figure 3.12, the second job of task τ_2 is first assigned a deadline $d_2^s = r_2 + P^s = 14$. At time $t_2 = 12$, however, since q^s is exhausted and the job is not finished, the job is scheduled with a new deadline $d_3^s = d_2^s + P^s = 21$. As a result of a longer execution, only the soft task is delayed, while the hard task meets all its deadlines. Moreover, the exceeding portion of the late job is not executed in background, but is scheduled with a suitable dynamic priority.

In other situations, frequently encountered in continuous media (CM) applications, tasks have fixed computation times but variable inter-arrival times. For example, this is the case of a task activated by external events, such a driver process activated by interrupts coming from a communication network. In this case, the CBS behaves exactly like a Total Bandwidth Server (TBS) [SB96] with a bandwidth $U^s = Q^s/P^s$. In fact, if $C_i = Q^s$ each job finishes exactly when the budget arrives to 0, so the server deadline is increased of P^s. It is also interesting to observe that, in this situation, the CBS is also equivalent to a Rate-Based Execution (RBE) model [JB95] with parameters $x = 1, y = T_i, D = T_i$. An example of such a scenario is depicted in Figure 3.13.

Finally, Figure 3.14 shows how the tasks presented in Figure 3.10 are scheduled by a CBS. Since the CBS assigns a correct deadline to the instance arriving late (the third instance of τ_1), τ_2 does not miss any deadline, and temporal protection is preserved.

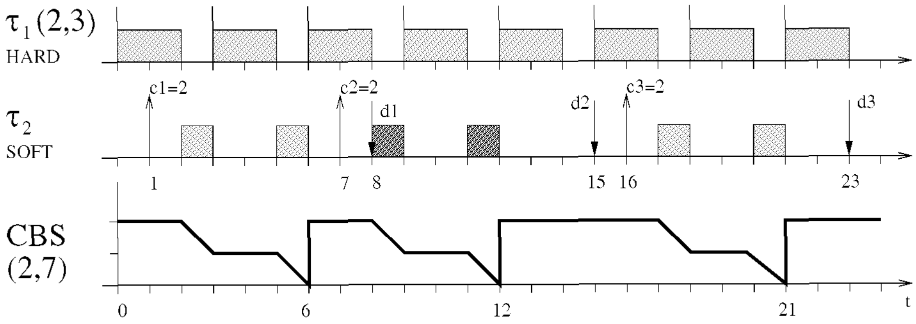


Figure 3.13 Example of CBS serving a task with constant execution time and variable inter-arrival time.

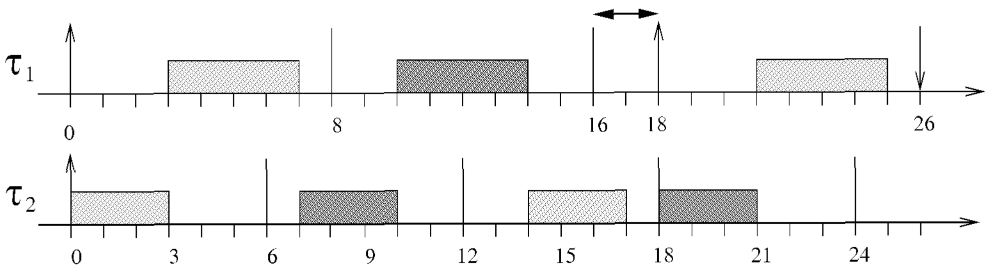


Figure 3.14 Example of a CBS coping with late arrivals.

3.6.2 CBS PROPERTIES

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting CM applications. The most important one, the *isolation property*, is formally expressed by the following theorem.

Theorem 3.1 *A CBS with parameters* (Q^s, P^s) *demands a bandwidth* $U^s = Q^s/P^s$.

To prove the theorem, we show that a CBS with parameters (Q^s, P^s) cannot occupy a bandwidth greater than $U^s = Q^s/P^s$. That is, the processor demand $g_s(t_1, t_2)$ (see Chapter 1) of the CBS in the interval $[t_1, t_2]$ is less than or equal to $(t_2 - t_1)Q^s/P^s$. That is, we show that

$$\forall t_1, t_2 \in \mathbb{N} : t_2 > t_1, \quad g_s(t_1, t_2) \leq \frac{Q^s}{P^s}(t_2 - t_1).$$

We recall that, under a CBS, a job $\tau_{i,j}$ is assigned an absolute time-varying deadline $d_{i,j}$ which can be postponed if the task requires more than the reserved bandwidth. Thus, each job $\tau_{i,j}$ is composed by a number of chunks $H_{i,j,k}$, each characterized by a release time $a_{i,j,k}$ and a fixed deadline $d_{i,j,k}$. To simplify the notation, we indicate all the chunks generated by a server with an increasing index k. The release time and the deadline of the kth chunk generated by the server will be denoted by a_k and d_k, respectively. Using this notation, the CBS algorithm can be formally described as illustrated in Figure 3.15.

If e_k denotes the server time demanded in the interval $[a_k, d_k]$ (that is, the execution time of chunk H_k), we can say that

$$\forall t_1, t_2, \quad \exists k_1, k_2 : \quad g_s(t_1, t_2) \;=\; \sum_{k: a_k \geq t_1 \wedge d_k \leq t_2} e_k \;=\; \sum_{k=k_1}^{k2} e_k.$$

If $q(t)$ is the server budget at time t and f_k is the time at which chunk H_k ends to execute, we can see that $q(f_k) = q(a_k) - e_k$, while $q(a_{k+1})$ is calculated from $q(f_k)$ in the following manner:

$$q(a_{k+1}) = \begin{cases} q(f_k) & \text{if } d_{k+1} \text{ was generated by Rule 2} \\ Q^s & \text{if } d_{k+1} \text{ was generated by Rule 1 or 3.} \end{cases}$$

Using these observations, the theorem can be proved by showing that:

$$g_s(a_{k_1}, d_{k_2}) + q(f_{k_2}) \leq (d_{k_2} - a_{k_1})U^s.$$

```
When job τ_j arrives at time r_j
      enqueue the request in the server queue;
      n = n + 1;
      if (n == 1) /* (the server is idle) */
        if (r_j + (c / Q) * P >= d_k)
          /*---------------Rule 1---------------*/
          k = k + 1;
          a_k = r_j;
          d_k = a_k + P;
          c = Q;
        else
          /*---------------Rule 2---------------*/
          k = k + 1;
          a_k = r_j;
          d_k = d_{k-1};
          /* c remains unchanged */
When job τ_j terminates
      dequeue τ_j from the server queue;
      n = n - 1;
      if (n != 0) serve the next job in the queue with deadline d_k;
When job τ_j executes for a time unit
      c = c - 1;
When (c == 0)
      /*---------------Rule 3---------------*/
      k = k + 1;
      a_k = actual_time();
      d_k = d_{k-1} + P;
      c = Q;
```

Figure 3.15 The CBS algorithm.

We proceed by induction on $k_2 - k_1$, using the algorithmic definition of CBS shown in Figure 3.15.

Inductive base. If in $[t_1, t_2]$ there is only one active chunk ($k_1 = k_2 = k$), two cases have to be considered.

<u>*Case a*</u>: $d_k < a_k + P$.

If $d_k < a_k + P$, then d_k is generated by Rule 2, so $a_k + \frac{q(f_{k-1})}{Q^s} P^s < d_k$ and $a_k = f_{k-1}$, that is

$$a_k + \frac{q(a_k)}{Q^s} P^s < d_k.$$

Being $q(f_k) = q(a_k) - e_k = q(a_k) - g_s(a_k, d_k)$, we have

$$a_k + \frac{g_s(a_k, d_k) + q(f_k)}{Q^s} P^s < d_k$$

hence

$$g_s(a_k, d_k) + q(f_k) < (d_k - a_k) \frac{Q^s}{P^s}.$$

<u>*Case b*</u>: $d_k = a_k + P^s$.

If $d_k = a_k + P$, then $g_s(a_k, d_k) + q(f_k) = e_k + q(f_k) = Q^s$. Hence, in both cases, we have:

$$g_s(a_{k_1}, d_{k_2}) + q(f_{k_2}) = g_s(a_k, d_k) + q(f_k) \leq (d_k - a_k) \frac{Q^s}{P^s} = (d_{k_2} - a_{k_1}) \frac{Q^s}{P^s}.$$

Inductive step. The inductive hypothesis

$$g_s(a_{k_1}, d_{k_2-1}) + q(f_{k_2-1}) \leq (d_{k_2-1} - a_{k_1}) \frac{Q^s}{P^s}$$

is used to prove that

$$g_s(a_{k_1}, d_{k_2}) + q(f_{k_2}) \leq (d_{k_2} - a_{k_1}) \frac{Q^s}{P^s}.$$

Given the possible relations between d_k and d_{k-1}, three cases have to be considered:

- $d_k \geq d_{k-1} + P^s$. That is, d_k is generated by Rule 3 or Rule 1 when $r_j \geq d_{j-1}$.
- $d_k = d_{k-1}$. That is, d_k is generated by Rule 2.
- $d_{k-1} < d_k < d_{k-1} + P^s$. That is, d_k is generated by Rule 1 when $r_j < d_{j-1}$.

Case a: $d_{k_2} = d_{k_2-1} + P^s$.

In this case d_{k_2} can be generated only by Rule 1 or 3. Adding e_{k_2} to both sides of the inductive hypothesis, we obtain:

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + e_{k_2}$$

and, since $q(f_k) = q(a_k) - e_k$, we have

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + q(a_{k_2}) - q(f_{k_2}).$$

Since d_{k_2} is generated by Rule 1 or 3, it must be $q(a_{k_2}) = Q^s$, therefore:

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + Q^s - q(f_{k_2})$$

$$\sum_{k=k_1}^{k_2} e_k + q(f_{k_2}) \leq (d_{k2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + Q^s \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} + Q^s$$

and finally

$$g_s(a_{k_1}, d_{k_2}) + q(f_{k_2}) \leq (d_{k2-1} - a_{k_1})\frac{Q^s}{P^s} + Q^s = (d_{k2-1} + P^s - a_{k_1})\frac{Q^s}{P^s}$$

$$g_s(a_{k_1}, d_{k_2}) + q(f_{k_2}) \leq (d_{k_2} - a_{k_1})\frac{Q^s}{P^s}.$$

Case b: $d_{k_2} = d_{k_2-1}$.

If $d_{k_2} = d_{k_2-1}$, then d_{k_2} is generated by Rule 2. In this case,

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + e_{k_2}$$

but, being $d_{k_2} = d_{k_2-1}$, $q(f_{k_2}) + e_k = q(a_{k_2})$ and $q(a_{k_2}) = q(f_{k_2-1})$ (by Rule 2), we have:

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2} - a_{k_1})\frac{Q^s}{P^s} - q(a_{k_2}) + e_{k_2} = (d_{k_2} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2})$$

hence

$$g_s(k_1, k_2) + q(f_{k_2}) = \sum_{k=k_1}^{k_2} e_k \leq (d_{k_2} - a_{k_1})\frac{Q^s}{P^s}.$$

Case c: $d_{k_2-1} < d_{k_2} < d_{k_2-1} + P^s$.

If $d_{k_2} < d_{k_2-1} + P^s$, d_{k_2} is generated by Rule 1, so $a_{k_2} + \frac{q(f_{k_2-1})}{Q^s}P^s \geq d_{k_2-1}$, hence $c(f_{k_2-1}) \geq (d_{k_2-1} - a_{k_2})\frac{Q^s}{P^s}$. Applying the inductive hypothesis, we obtain

$$\sum_{k=k_1}^{k_2-1} e_k + e_{k_2} \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2-1}) + e_{k_2}$$

from which we have

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2-1} - a_{k_1})\frac{Q^s}{P^s} - (d_{k_2-1} - a_{k_2})\frac{Q^s}{P^s} + e_{k_2}$$

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2-1} - d_{k_2-1} - a_{k_1} + a_{k_2})\frac{Q^s}{P^s} + e_{k_2}.$$

Now, being $e_{k_2} = Q^s - q(f_{k_2})$, we have:

$$\sum_{k=k_1}^{k_2} e_k \leq (-a_{k_1} + a_{k_2})\frac{Q^s}{P^s} + Q^s - q(f_{k_2}) = (a_{k_2} + P^s - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2})$$

but, from Rule 1 and 3, we have $d_k = a_k + P^s$, so we can write

$$\sum_{k=k_1}^{k_2} e_k \leq (d_{k_2} - a_{k_1})\frac{Q^s}{P^s} - q(f_{k_2})$$

hence

$$g_s(k_1, k_2) + q(f_{k_2}) = \sum_{k=k_1}^{k_2} e_k \leq (d_{k_2} - a_{k_1})\frac{Q^s}{P^s}. \quad \square$$

The isolation property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to each task that cannot be guaranteed a priori. The most important consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee even in the case in which soft requests exceed the expected load.

In addition to the isolation property, the CBS has the following characteristics:

- No assumptions are required on the WCET and the minimum inter-arrival time of the served tasks: this allows the same program to be used on different systems without recalculating the computation times. This property allows decoupling the task model from the scheduling parameters.
- If the task's parameters are known in advance, a hard real-time guarantee can be performed (see Section 3.7).
- The CBS automatically reclaims any spare time caused by early completions or late arrivals. This is due to the fact that whenever the budget is exhausted, it is always immediately replenished at its full value and the server deadline is postponed. In this way, the server remains eligible and the budget can be exploited by the pending requests with the current deadline.
- Knowing the statistical distribution of the computation time of a task served by a CBS, it is possible to perform a statistical guarantee, expressed in terms of probability for each served job to meet its deadline (see Section 9.5).

3.7 TEMPORAL GUARANTEES

Resource reservations provide a basic *scheduling mechanism* that, thanks to the temporal isolation property, can be used in different ways to serve hard or soft tasks providing different kinds of guarantees.

In this section we briefly recall some possible *parameters assignment policies*; note that, although most of the presented results are applied to the CBS algorithm (because they were originally developed for the CBS), they can be extended to other reservation policies.

The first (and simplest) usage of a reservation algorithm is to use it for serving aperiodic tasks so that they do not interfere with the hard real-time activities. This is the approach followed in all the works on aperiodic servers [SSL89, LSS87, LRT92, TL92, SLS95, SB94, SB96, GB95].

Obviously, a single CBS can be used to serve all the soft real-time tasks, but in this case it might be very difficult to provide soft real-time guarantees. The best way to provide some kind of performance guarantee to soft real-time tasks is to serve each task with a dedicated CBS (or CPU reservation). In this way, it is possible to guarantee that each task is periodically assigned a given amount of time; if the task parameters are not know a priori this is the only performance guarantee that can be performed, but if some information is known about the task, more complex guarantee strategies can be used.

Finally, a dedicated server can also be used to schedule hard real-time tasks, which can be guaranteed thanks to the *hard schedulability* property, expressed by the following lemma:

Lemma 3.1 *A hard task τ_i with parameters (C_i, T_i) is schedulable by a CBS with parameters $Q^s \geq C_i$ and $P^s \leq T_i$ if and only if τ_i is schedulable without the CBS.*

Proof.
For any job of task τ_i, $r_{i,j+1} - r_{i,j} \geq T_i \geq P^s$ and $c_{i,j} \leq C_i \leq Q^s$. Hence, by definition of the CBS, each job $J_{i,j}$ is assigned a scheduling deadline $d^s_{i,j} = r_{i,j} + P^s$ (since $r_{i,j}$ is always greater than $d^s_{i,j-1}$) and it is scheduled with a budget $Q^s \geq C_i$. Moreover, since $c_{i,j} \leq Q^s$, each job finishes no later than the budget is exhausted, hence the deadline assigned to a job does not change and is exactly the same as the one used by EDF. □

All the policies described above can be used off-line for assigning reservations parameters during the system design phase, when tasks parameters are known a-priori. But, as explained in Chapter 1, such an a-priori information is often not available and static allocation techniques cannot be used. In this case, it is possible to dynamically change the reservation parameters as explained in Chapter 8.

3.8 RESOURCE RESERVATIONS IN OPERATING SYSTEM KERNELS

Resource Reservations have been implemented in various real-time kernels (mainly research kernels), starting from Real-Time Mach, and are available in a commercial real-time extension of Linux, (Linux/RK by TimeSys). While most of these systems

provide CPU reservations as an alternative to classical real-time scheduling algorithms, few of them base the whole kernel on the reservation concept and provide reservations for all the resources managed by the system.

3.8.1 REAL-TIME MACH

Real-Time Mach (RT-Mach) [TNR90] is a real-time extension of the Mach μkernel [RBF$^+$89], developed at the CMU. RT-Mach extends the standard Mach by increasing the predictability of the kernel, and providing a real-time threading library, a real-time scheduler, and a real-time communication mechanism.

The predictability of the kernel is increased by using eager evaluation policies (opposed to the lazy evaluation policies used by standard Mach) and by substituting the FIFO queues contained in the kernel with priority queues (where the priorities are derived by the tasks' temporal constraints). As an example of lazy evaluation policy used in standard Mach, when a task dynamically allocates some memory, the kernel really gives it to the task only when the task accesses the allocated memory. Such a "lazy allocation" allows enhancing the kernel efficiency, and enabling some optimizations such as copy-on-write, but increases the unpredictability of the system. Hence, RT-Mach modifies this behavior by immediately allocating the memory; other similar optimizations present in the Mach μkernel have been removed in RT-Mach for similar reasons. The real-time threading library coming with RT-Mach implements the periodic and sporadic thread models, enabling the user to express the WCET and the period (or the minimum interarrival time) for each thread. In this way, RT-Mach can perform the admission control and correctly schedule the treads using a Rate Monotonic (or Deadline Monotonic) scheduler. Finally, the real-time communication mechanism uses priority inheritance [SRL90] to bound the waiting times.

CPU reservations were added to RT-Mach by Mercer and others [MST94a] to support multimedia applications. In particular, the authors realized the lack of temporal protection presented by the priority-based RT-Mach scheduler (similar to the problem shown in Section 3.1), and implemented a CPU reservation mechanism based on the Rate Monotonic algorithm. This was done by enhancing the RT-Mach time accounting mechanism to exactly measure the execution time used by each thread (and keeping track of the reservation budget) and by implementing an enforcement mechanism. The enforcement mechanism downgrades a thread to non real-time when it consumes all its reserved time (the thread will be promoted again to real time priority at the beginning of the next reservation period). The authors argued that to compensate some approximations in accounting and enforcement, a fraction of the CPU time must be left unreserved, and they estimated this percentage in about $5 - 10\%$. Since in realistic situations the RM utilization bound is about 88% [LSD89], the authors claim that basing

the reservation mechanism on EDF would not give any sensible advantage with respect to RM, and thus they adopted the RM scheduler provided by RT-Mach as a basis for their CPU capacity reserves.

Nowadays, using modern hardware and OS kernels the overhead for accounting and enforcement is negligible, hence there are no more reasons for compensating it. As a consequence, basing the reservation mechanism on EDF can be a realistic choice.

3.8.2 OTHER RESEARCH SYSTEMS

CPU Reservations have also been implemented in other research kernels to support predictable CPU allocation in dynamic systems.

For example, Rialto is a research system developed by Microsoft [JIF+96] that permits to mix CPU reservations and other kinds of timing constraints. Rialto was designed to combine timesharing and soft real-time in a desktop operating system, and thus uses CPU reservations to isolate the different applications. The execution time is reserved to *activities* and monitored at runtime. Activities can be composed by more threads, and threads belonging to the same activity share its reserved time in a round-robin fashion.

Another difference between Rialto CPU reservations and traditional ones is that in Rialto reservations are continuously guaranteed. That is to say, if an activity has a (Q, T) reservation, then for every time t the activity will run for at least Q units of time in the interval $(t, t+T)$[2]. This result is impossible to obtain using a priority scheduler, and in fact Rialto uses a table driven schedule that is computed when a reservation is created and is repeated over time.

Moreover, Rialto provides *time constraints*: a time constraint is a tuple (s, c, b), indicating that a thread requires to execute for a time c, starting at time s, and terminating before b. Based on the thread's activity reserved time on the static schedule, and on the available spare time, Rialto can guarantee the time constraint or reject it. If the time constraint is accepted, the activity's threads are scheduled so that it is respected (the scheduling algorithm used inside the activity is based on EDF).

Another system supporting resource reservations is HARTIK [But93b], an experimental real-time kernel developed at the ReTiS Lab of the Scuola Superiore S. Anna of Pisa, to support real-time and control applications running on conventional PC hardware (based on Intel x86 processors). Like RT-Mach and Rialto, HARTIK permits to explicitly express the tasks' temporal constraints, implements an on-line admission

[2] In a traditional reservation, this is valid only for $t = kT + t_0$, where t_0 is a fixed offset.

test, and dynamically create and destroy processes. Moreover, HARTIK also provides some unique features that are rarely found all together in other kernels. They include a support for both periodic and aperiodic processes, the possibility to mix hard, soft, and non real-time tasks, the implementation of resource sharing protocols (based on SRP [Bak91]), and the presence of a non-blocking communication mechanism (the CAB [But93b]) for exchanging data among periodic tasks having different rates. The HARTIK scheduler is based on EDF.

The kernel was later extended to support multimedia applications through the CBS, which was explicitly designed to efficiently schedule periodic and aperiodic soft tasks with unknown execution times [AB00]. Nowadays, the CBS can be used in HARTIK to schedule both hard and soft real-time tasks, or to reserve a fixed fraction of the CPU bandwidth to non real-time tasks to prevent starvation. Moreover, the CBS is used to schedule all the drivers' tasks so that it is not necessary to adjust the drivers' WCET estimation on every new machine the first time a driver runs on it.

Another real-time kernel developed at the Retis Lab of Scuola Superiore S. Anna of Pisa is SHaRK [GAGB01]. ShaRK is an evolution of HARTIK and has been designed to easily implement new scheduling algorithms in the kernel as *scheduling modules*. The CBS is still provided as one of the standard scheduling modules, and other reservation mechanisms can be easily added, hence SHaRK provides full support for CPU reservations.

A similar concept (easy implementation of new scheduling algorithms) is proposed by RED Linux, that modifies the 2.2 Linux kernel to provide high-resolution timers, low kernel latency, and a modular scheduler. This latest feature permits to easily implement CPU reservations in RED Linux.

3.8.3 RESOURCE KERNELS

Extending the concepts presented by the kernels described above, it is possible to consider resource reservations as an abstraction for decoupling the applications from the scheduling algorithm. Hence, applications only need to express their resource requirements in terms of reservations (Q, T) (plus an optional parameter D indicating a relative deadline), so that the kernel can perform an admission test and schedule tasks in the proper way. This is the *resource centric* approach taken by Resource Kernels (RK) [RJMO98].

A resource kernel is based on the *Resource Set* abstraction, which describes all the resources that can be used by one or more tasks. A resource set may include multiple reservation types (for example, a CPU reservation, a network reservation, and a disk

reservation), and all the tasks attached to the resource set will be allowed to use those reservations. Hence, in order to be guaranteed to execute in a proper timely fashion, a task must create a resource set, create the proper resource reservations expressing its requirements, connect them to the resource set, and then attach itself to the resource set.

The RK concept was initially implemented in a modified version of RT-Mach, but it is fairly general [OR99], and has been, for example, ported to Linux [OR98]. Linux/RK provides high-resolution timers and an accurate accounting mechanism, and implements the resource set abstraction, CPU reservations (based on RM, DM, or EDF), network reservations, and disk reservations.

A commercial version of Linux/RK is distributed by TimeSys as TimeSys Linux [Tim03], which adds some additional feature (such as more predictable kernel services) to the original RK.

4

MULTI-THREAD APPLICATIONS

Computers are powerful enough to run several applications at the same time, each consisting of multiple concurrent activities. In this chapter we consider the problem of supporting multiple real-time applications in the same computing system, so that each application can be handled by its own scheduling policy and analyzed independently of the others.

4.1 THE THREAD MODEL

The thread model of concurrent programming is very popular and it is supported by most operating systems. In this model, concurrency is supported at two levels: processes and threads. Each process has its own address space and processes communicate mainly exchanging messages by means of operating system primitives. Creating a process is an expensive operation because it involves a lot of steps, like creating the address space, setting up the file descriptors, etc. Moreover, context switching among processes is an expensive operation.

A process can be multi-threaded, that is, it can consist of several concurrent threads. Different threads belonging to the same process share address space, file descriptors, and other resources. Since threads belonging to the same process share the address space, the communication is often realized by means of shared data structures protected by mutexes. Creating a new thread is far less expensive than creating a new process. Context switching among threads of the same process is faster.

The thread model is supported by all general purpose operating systems because it has a lot of advantages with respect to a pure process model. The designer of a concurrent application, in fact, can structure the application as a set of cooperating threads, simplifying the communication and reducing the overhead of the implementation.

When designing a concurrent application, in which tasks have to cooperate tightly and efficiently, the thread model is the most suited. As an example, consider a web server that can serve many clients at the same time. We can structure the program as one main thread that waits for new connections, and one active thread for each client. Another example is an MPEG player that plays streams coming from the network: a typical design structure for this application consists of a thread that waits for new data from the network and writes them into a buffer; a second thread that periodically reads the video frames and the audio data from the buffer, decodes and a displays them; and a third thread that waits for user commands. All the three threads interact tightly: therefore, communication and scheduling must be fast and efficient.

Classical hard real-time systems usually consist of periodic or sporadic tasks that tightly cooperate to fulfill the system goal. For efficiency reasons, they communicate mainly through shared memory, and appropriate synchronization mechanisms are used to regulate the access to shared data. Since all tasks in the system are designed to cooperate, a global schedulability analysis is done on the whole system to guarantee that the temporal constraints will be respected. There is no need to protect one subset of tasks from the others. Therefore, we can assimilate a hard real-time system to a single multi-threaded process where the real-time tasks are modelled by threads.

In general purpose operating systems, many processes are active at the same time, and they *compete* for the processor and other hardware resources. Therefore, two important goals of any general purpose operating system are to regulate the competition among the processes, allowing a *fair share* of the system resources to every process, and to *protect* each process from the interferences of the others.

Summarizing, in multi application systems:

- Processes are developed independently. They must be protected from each other to prevent reciprocal interference. If one process fails, the other process must not be affected. They compete for the system resources, so the global scheduler has to regulate such a competition to ensure fairness.

- Threads are designed and developed together. They cooperate for producing the application's results. If one thread fails, the whole application may fail.

4.1.1 MULTI-THREADED REAL-TIME APPLICATIONS

If some sort of real-time execution has to be supported in general-purpose operating systems, multi-threaded programming must be taken into consideration. According to

the multi-thread model, in this chapter we assume that real-time tasks are implemented as threads, and a classical real-time application as one single multi-threaded process. Therefore, a real-time application is a process that can be multi-threaded, that is, it can consists of many real-time tasks. In the remainder of this chapter, the terms *thread* and *task* will be used as synonyms, as the terms *application* and *process*.

A user that wants to execute (soft) real-time applications in a general-purpose operating system would like to have the following nice features:

1. It should be possible to assign each real-time application a *fraction* of the system resources, so that it executes as it were executing alone in a slower *virtual processor*;
2. Each application should receive execution in a timely manner, depending on its real-time characteristics (e.g., the tasks' deadlines);
3. A non real-time application should not be able to disrupt the allocation guaranteed to real-time applications.

Such properties can be very well supported through any resource reservation mechanisms, such the ones described in Chapter 3. However, in the case of multi-thread systems, the resource reservation mechanism must be applied not at the task level, but at the application level. This poses many problems, as we will see in the next sections.

4.1.2 CUSTOMIZED SCHEDULING

Figure 4.1 illustrates an example of a multi-thread real-time operating system. An interesting feature would be the possibility of specifying a *local scheduler* for the application's threads. For example, Application A could specify a non-preemptive FIFO scheduler, Application B could specify a Round Robin policy, whereas Application C a fixed priority scheduler.

Therefore, in this model, we distinguish two levels of scheduling. At the higher level, a *global scheduler* selects the application to be executed on the processor and, at a lower level, a *local scheduler* selects the task to be executed for each application. Such a two-level scheduling scheme has two main advantages:

- each application can use the scheduler that best fits its needs;
- legacy applications, designed for a particular scheduler, can be re-used by simply re-compiling, or at most, with some simple modification.

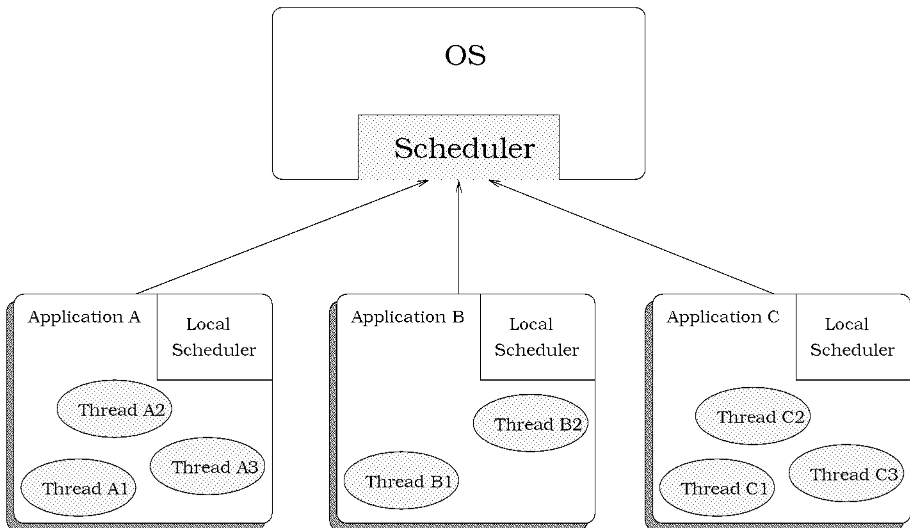


Figure 4.1 A multi-thread operating system: each application (or process) consists of one or more threads. In addition, each application can have its own scheduling algorithm.

The two-level approach can also be generalized to an arbitrary number of levels. In the literature, this class of systems is referred to as *hierarchical scheduling scheme*. In this chapter, however, we will consider only two levels, since this model is readily available in most operating systems, as discussed in the previous section.

As an example, consider an engine control system in the automotive domain. We can distinguish two sets of activities:

- Aperiodic activities, such as the one triggered by the *crank angle sensor*. In this task, the time between two consecutive activations is variable and depends on the rotations per minute of the engine. Therefore, the activity is triggered by an interrupt at a variable rate. Probably, the best way to schedule such an activity is by an on-line algorithm, under fixed or dynamic priorities.

- Periodic activities. Many activities in a control system are periodic, and must be activated by a timer. For example, in automotive applications, most systems are connected to other electronic boards in the car through a Time Triggered network, which requires precise timing. Therefore, this second set of activities are best handled with an off-line table-driven scheduler, executed according to a time-triggered paradigm [KDK+89].

If we are forced to use a single scheduling paradigm for the whole system, we must either reduce the two sets of activities to periodic tasks, scheduling them by a time-triggered paradigm, or re-program the second set of activities to be handled by an on-line priority scheduler. Both solutions require some extra programming effort and are not optimal in terms of resource usage. The best choice would be to program the two sets of activities as two distinct components, each one handled by its own scheduler, and then integrate the two components in the same system.

A similar problem arises when dealing with real-time systems with different criticality, because different scheduling paradigms are used for hard and soft real-time activities. Again, a way of composing such activities would be to implement them as different components, each one handled by its own scheduling algorithm.

Another motivation for composing schedulers is to have the possibility of re-using already existing components. Suppose we have two components consisting of many concurrent real-time tasks, one developed assuming a fixed priority scheduler, and one developed assuming a Round Robin scheduler. If we want to integrate the two components in a new system and we cannot go back to the design phase (for example for cost reasons), we need a method for combining and analyzing the two components together, *without changing the scheduling algorithms*.

4.1.3 SCHEDULING MULTI-THREADED APPLICATIONS

Traditional real-time schedulability analysis cannot be directly applied to independently developed multi-threaded applications. For example, consider an application that runs on an open system with a fixed priority local scheduler: we would like to know whether the application will meet its temporal requirements. If we use traditional hard real-time scheduling techniques, it becomes very difficult to analyze the system. First of all, in order to apply a global schedulability analysis, we should know how many applications are in the system and their temporal characteristics; even with this information, the schedulability analysis is not trivial, because the system includes several applications with different local schedulers.

If hard and soft real-time applications are mixed in the same system we have an additional problem: if the system does not provide temporal protection, a non-critical application could execute longer than expected and starve all other applications in the system.

In order to validate each application independently, we need to use a resource reservation mechanism and provide temporal protection. Moreover, the schedulability analysis

is greatly simplified, because we only need to take into account one application and one scheduling algorithm at a time. To use this approach, we have to assign each application a fraction of the processor utilization, as explained in Chapter 3.

Now the question is: what is the minimum fraction of processor that must be reserved to an application in order to guarantee its temporal requirements? An intuitive solution would be to assign each application an amount of resource equal to the utilization of the application tasks. For example, if the application tasks have a total maximum load of 0.3, we can assign 30% of the processor to the application. Indeed, if we use a scheduler that provides a perfect abstraction of a virtual dedicated processor whose speed is a fraction of the shared processor speed, then this approach is correct. One of such mechanisms is the GPS (General Processor Sharing) policy, described in Section 3.3.

Unfortunately, this solution is not feasible, as the GPS cannot be implemented in practice. All schedulers that can be implemented can only provide "imperfect" abstractions of the virtual processor. Any scheduler that supports temporal protection (see Chapter 3) provides at least one parameter that specifies the "granularity" of the allocation. For example, in Proportional Share algorithms (like EEVDF, described in Section 3.4.3) we must specify the system quantum; with the Constant Bandwidth Server (CBS) described in Section 3.6.1, we must specify the server period. The smaller the granularity, the closer the allocation of the resource to that of the ideal GPS algorithm. However, a small granularity implies a large system overhead. For example, in Proportional Share algorithms we have exactly one context switch every quantum boundary. With CBS, small periods imply a large number of deadline recalculations and queue re-orderings. Thus, to contain runtime overhead, the "granularity" should not be too small.

On the other hand, a coarse granularity of the allocation could lead to unfeasible schedules. An example of such a problem is illustrated in Figure 4.2. In this example, the system consists of two applications A_1 and A_2. Application A_1 comprises two sporadic tasks, τ_1 with computation time $C_1 = 1$ and minimum interarrival time $T_1 = 15$; τ_2 with computation time $C_2 = 1$ and minimum interarrival time $T_2 = 5$. The total utilization of application A_1 is $U_1 = 0.2$. Application A_2 is non real-time and consists of a single job, that arrives at time 0 and requires a very large amount of execution time (for example, it could be a scientific program performing a long computation).

In this example, each application is served by a dedicated CBS. Application A_1 is served by S_1, with a capacity $Q = 1$ and a period $P = 5$. Application A_2 is served by S_2, with $Q = 6.4$ and $P = 8$. The dashed arrows represent the deadlines of the servers, whereas the solid arrows are the deadlines of the jobs.

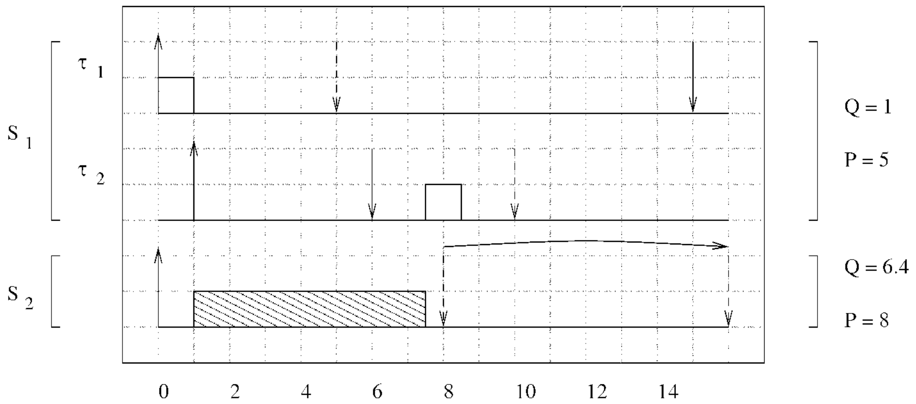


Figure 4.2 Problems with the time granularity of the resource allocation.

When task τ_1 arrives at time 0, the server S_1 is activated and, having the shortest deadline, it is selected to execute. When task τ_2 arrives at time 1, the server budget is already depleted. Therefore, the first instant at which τ_2 can be scheduled is $t = 7.4$, after the task's deadline.

Although the example is based on the CBS algorithm, the same thing happens with all resource reservation mechanisms presented so far.

There are two solutions to making application A_1 schedulable: assigning the server a larger share of the processor or assigning it a smaller period. For example, if all arrival times are integer numbers, by assigning S_1 a period of $P_1 = 1$, the above system becomes schedulable. The resulting schedule is shown in Figure 4.3. Note that there are a large number of context switches.

Another way to make the system schedulable is to increase the budget of application A_1. If we let $P_1 = 5$, we can make the system schedulable by raising the budget to $Q_1 = 2$. The resulting schedule is shown in Figure 4.4, and there are definitely less context switches. However, A_1 was assigned a bandwidth twice as much as before, "wasting" processor resources that could be assigned to other applications. Deciding which approach is better clearly depends on the overhead introduced by the algorithms and on the context switch time.

To implement a two-level scheduling scheme, the following problems need to be addressed:

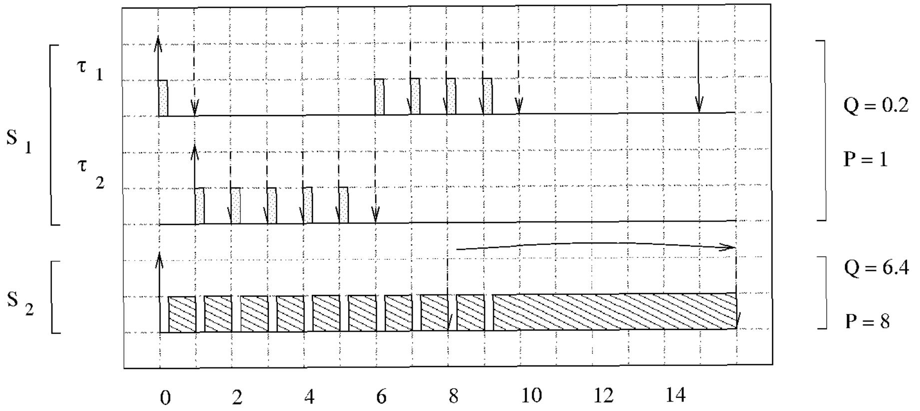


Figure 4.3 Solution a): reducing the granularity.

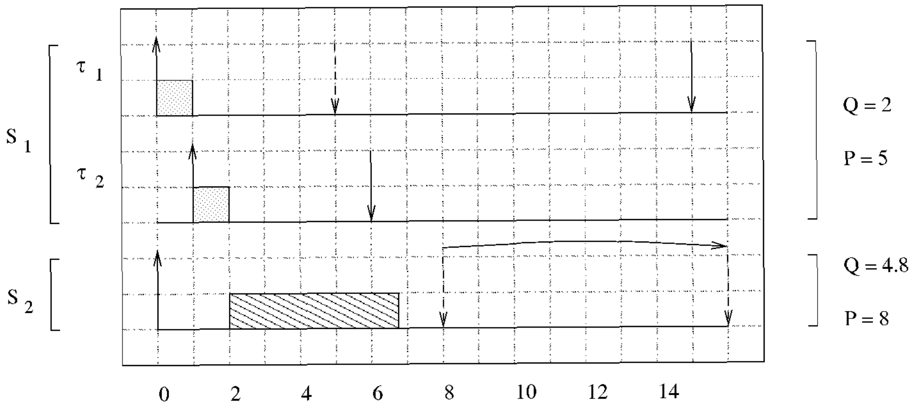


Figure 4.4 Solution b): increasing the budget.

- Which global scheduling algorithm can be used to allocate the processor to the different applications?
- Given the global scheduling algorithm and the application timing requirements, how to allocate the processor share to the applications so that their requirements are satisfied?
- Given the global scheduling algorithm and the processor share assigned to the applications, how can the schedulability be analyzed?

Some solutions to the problems stated above are presented in the next sections.

4.2 GLOBAL APPROACHES

In this section we will present two algorithms that take a global approach to the hierarchical scheduling problem. The Deng and Liu's [DLS97, DL97] was the first to tackle the problem of hierarchical scheduling in open systems, where there is little a-priori knowledge about the application requirements. The Bandwidth Sharing Server (BSS) [LB99, LCB00] was later proposed to improve the performance of the first algorithm.

The approach followed by these two algorithms is similar and only differs for the way the system scheduler is implemented. In both cases, the system scheduler takes advantage of the knowledge that tasks have about the budget assigned to each application. In Deng and Liu's algorithm, the global system scheduler requires the knowledge of the execution time of each task instance, whereas in the BSS the global scheduler requires the knowledge of the deadlines of the application tasks.

Other approaches will be discussed in Section 4.3, which do not require *any* information about the behavior of the applications.

4.2.1 THE DENG AND LIU'S ALGORITHM

The concept of *open system* was first introduced by Deng and Liu [DLS97, DL97]. Two main properties characterize an open system: each application can be validated independently and can be dynamically activated in the system. Unlike closed systems, where the temporal behavior of the system can be completely characterized a priori, in a open system it is impossible to perform an off-line schedulability analysis for the entire system, since we do not how many applications will be present at every instant of time.

According to the model previously described, an application consists of a set of tasks (periodic or sporadic) and a local scheduler. Many applications can coexist in the same system, and a global system scheduler allocates the processor to each application task.

The authors distinguish the following types of application:

- Non-preemptive applications. In this case, the application tasks are considered locally non preemptive. In other words, the task selected for execution by the local scheduler cannot be preempted by other tasks of the same application, but it can be preempted by tasks of other applications.
- Preemptive predictable applications. They are applications in which all the scheduling events are known in advance, as for the case of applications consisting of periodic real-time tasks.
- Preemptive non-predictable applications. They are those in which it is not possible to know all the scheduling events in advance. Applications containing one or more sporadic tasks belong to this category.

Each application category is handled by a different scheduling algorithm and requires a different type of guarantee.

Given this model, the authors proposed a scheme involving an on line acceptance test and a dynamic on line scheduler. When an application $\mathcal{A}_i$ wants to enter the system, it must declare its *quality of service* requirements in terms of desired utilization U_i. If there is enough free bandwidth to satisfy the new requirements, the application is accepted, otherwise it is rejected.

According to this algorithm, each application $\mathcal{A}_i$ is assigned a dedicated server S_i with a maximum utilization factor U_i. The sum of all server's bandwidths cannot exceed 1. Each server keeps track of a current budget q_i and a deadline d_i, which are both initialized to 0. The server can be eligible or non eligible, depending on the value of variable e_i, also called *eligibility time*. Initially, $e_i = 0$ and the server is eligible. Server S_i is eligible at time t if $t \geq e_i$. All eligible servers are scheduled by the global scheduler which is essentially EDF: the eligible server with the shortest deadline is executed by the global system.

The server dedicated to each application is the Constant Utilization Server (CUS) [DLS97], which is a variant of the Total Bandwidth Server (TBS) [SB96] proposed by Spuri and Buttazzo. Since the behavior of the CUS server differs for the three application categories, its details will be described in the corresponding sections.

NON-PREEMPTIVE APPLICATIONS

The CUS algorithm updates the server variables according to the following rules:

1. If a task of application $\mathcal{A}_i$ arrives at time t, requiring a computation time c_i:
   - If the local scheduler queue is empty (i.e., no other task in the application is active and the application is currently idle), and the server is eligible, then $q_i \leftarrow c_i$ and $d_i \leftarrow max(t, d_i) + \frac{c_i}{U_i}$. Moreover, the server is inserted in the global EDF queue. Notice that the computation time c_i of the task is required to compute the server deadline.
   - If the local scheduler queue is non-empty or the server is not eligible, the task is inserted in the ready queue of the local scheduler until it becomes the first in the queue and the server is eligible.
2. If a server is selected for execution by the global scheduler (because it is the server with the earliest deadline), it starts executing the corresponding task and decreases the budget q_i accordingly.
3. A server is allowed to execute until its budget is equal to 0 or until the task finishes. If the task finishes before the budget is depleted, the server eligibility time e_i is set to d_i and the server becomes non eligible. If the budget is depleted before the task finishes executing, an exception is raised to the application. What to do in this case is left unspecified and taking the most appropriate action is up to the application responsibility.

As an example of use of the CUS algorithm, consider a system consisting of two applications: $\mathcal{A}_1$, consisting of two periodic tasks, $\tau_1 = (1, 5)$ and $\tau_2 = (2, 12)$, and $\mathcal{A}_2$, consisting of a single periodic task $\tau_3 = (4, 8)$. Each application is handled by a CUS with bandwidth $U_1 = U_2 = 0.5$ and is scheduled by a non preemptive EDF local scheduler.

Figure 4.5a illustrates the schedule generated by executing application $\mathcal{A}_1$ on a dedicated processor with speed 0.5. The upward arrows denote the arrival times of the two tasks, whereas the deadlines are not shown as they coincide with the arrival times. Figure 4.5b illustrates the schedule generated for the two applications by the CUS algorithm. Downward arrows denote the deadline of server S_1. We now analyze the first scheduling events:

- At time $t = 0$, all tasks are ready to execute. The local scheduler of application $\mathcal{A}_1$ chooses τ_1 to be executed. Since $c_1 = 1$, the server deadline is set to $d_1 = 2$ and the budget $q_1 = 1$. Similarly, the deadline of server S_2 is set to $d_2 = 4/U_2 = 8$ and $q_2 = 4$. Hence, the global EDF scheduler selects server S_1 to be executed.

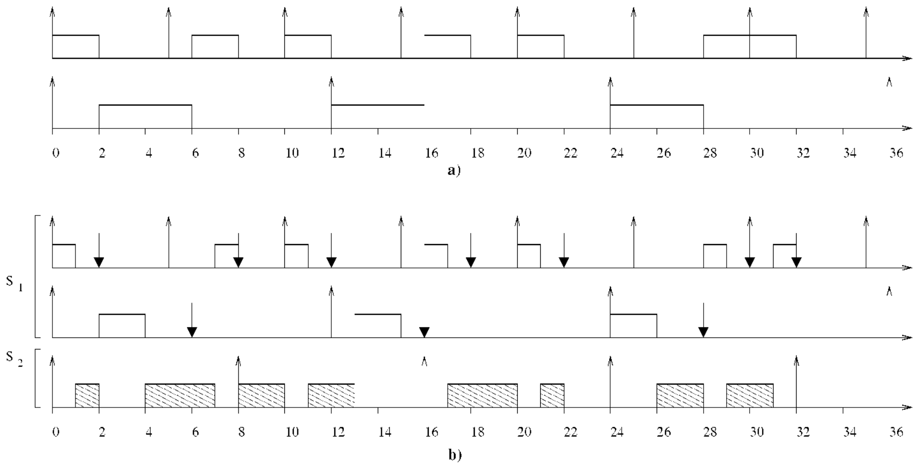


Figure 4.5 Example of schedule of a non preemptive application $\mathcal{A}_1$ by the CUS algorithm: a) schedule of $\mathcal{A}_1$ on a dedicated processor; b) schedule generated by CUS on the shared processor.

- At time $t = 1$, task τ_1 has completed. The eligibility time of S_1 is set to $e_1 = d_1 = 2$, and server S_1 is not eligible until time e_2.
- Server S_2 executes for 1 time unit. Then, at time $t = 2$, server S_1 becomes eligible and the next task in the queue is selected. As a consequence, the server variables are updated as follows: $d_1 \leftarrow d_1 + c_2/U_1 = 6$, $q_1 \leftarrow c_2 = 2$. S_1 is now the earliest deadline server, so it preempts server S_2. Notice that, while application $\mathcal{A}_1$ is non preemptive, the system as a whole is preemptive, as a server can preempt the others.
- At time $t = 4$ task τ_2 completes. Therefore, $e_1 = d_1 = 6$. Server S_1 is not eligible and server S_2 can execute. Notice that S_1 will be non eligible until time $t = 6$ even though task τ_1 becomes active at time $t = 5$.

Notice that the deadline of server S_1 is always less than or equal to the deadline of all the tasks that are currently being executed. Actually, by comparing Figures 4.5a and 4.5b, it is possible to see that the server deadline *is always equal to the finishing time of the task currently executed in the dedicated processor*. This is an interesting property of the CUS algorithm for non preemptive applications that is used to prove the following important theorem [DLS97].

Theorem 4.1 *If a non preemptive application $\mathcal{A}_i$ is schedulable on a dedicated processor with speed U_i, then it is schedulable with a CUS server with a bandwidth U_i.*

PREDICTABLE APPLICATIONS

The CUS algorithm presented above has a limited applicability, since it requires the local scheduling algorithm to be non-preemptive. In fact, Deng and Liu showed that, if a preemptive application is schedulable on a dedicated slower processor of speed U_i, it would require a server with bandwidth as large as 1 to be scheduled on the shared processor!

To overcome this limitation, Deng and Liu consider predictable and non predictable applications. In predictable applications, the operating system knows, at each time t, the next instant at which a scheduling decision must be taken for the application. For example, predictable applications are those consisting of periodic real-time tasks with known worst-case execution time. For these applications, Deng and Liu proposed to modify the CUS algorithm as follows:

- Let t be an instant at which a scheduling decision must be taken for application $\mathcal{A}_i$, and let $next_i(t)$ be the next of such instants;
- Let τ_{ij} be the task to be scheduled in application $\mathcal{A}_i$ and let U_i be the share of processor reserved for application $\mathcal{A}_i$. Let c_{ij} be the remaining computation time required by τ_{ij} at time t.
- The server deadline is set equal to

$$d_i = \min(next(t), \frac{c_{ij}}{U_i})$$

- The server budget is set equal to:

$$q_i = (d_i - t)U_i$$

- The server is inserted in the global EDF queue and the global scheduler is invoked;
- If the budget of the application goes to 0, or the task completes, the server replenishment time is set equal to d_i.

Then, Theorem 4.1 can be restated as follows.

Theorem 4.2 *If application $\mathcal{A}_i$ is non preemptive or predictable, and it is schedulable on a dedicated processor with speed U_i, then it is schedulable with a CUS server with a bandwidth U_i.*

NON PREDICTABLE APPLICATIONS

When dealing with non predictable applications (like those including one or more sporadic tasks) it is not possible to know in advance the next instant at which a scheduling decision must be taken by the local preemptive scheduler. To solve this problem, Deng and Liu proposed to use a quantum ϵ. The idea is to compute the server budget and deadline as follows:

$$q_i = \min(c_{ij}, \epsilon * U_i), \quad d_i = \frac{q_i}{U_i}$$

The smaller the ϵ, the higher the number of budget recalculations. However, the smaller the ϵ, the smaller the difference between the utilization of the server and the speed of the slower dedicated processor. The problem is very similar to the one described in Figures 4.2, 4.3 and 4.4. Actually, it is easy to see that in this case the CUS algorithm becomes very similar to the CBS algorithm. Computing the maximum error as a function of the quantum ϵ is not trivial and depends on the application characteristics. We remand to the original paper [DL97] for more details on the matter.

EXTENSIONS

Deng and Liu's approach has been extended by Kuo and Li [KL99], who presented a model in which the global scheduling strategy is based on fixed priorities together with a deferrable server or a sporadic server [SLS95, LSS87, SSL89]. Each application can be handled by a dedicated server with capacity Q_i and period P_i. To achieve maximum utilization, the following conditions must be satisfied:

- The period of each server must be a multiple or a divisor of the periods of all other servers in the system;
- The period of all the tasks must be multiple of the period of the server.

Kuo and Li also addressed the problem of sharing resources among tasks of different applications. Each task is allowed to share resources through mutually exclusive semaphores under the Priority Ceiling Protocol [SRL90]. This introduces potential blocking for a task accessing a resource locked by a task in another application. Therefore, it is necessary to use a global scheduling condition that takes into account such a blocking time. As a consequence, the isolation properties cannot be guaranteed as in the case of independent applications.

This algorithm has the advantage of not requiring the knowledge of the worst case execution time of all application tasks. However, the conditions on the periodicity of the server are quite strong. As a consequence the algorithm is not flexible enough to

be used in an open system. Nevertheless, this algorithm was the first one addressing the problem of scheduling an application through a dedicated periodic server, as the deferrable server algorithm. In Section 4.3 we will see other algorithms that extend and generalize this approach.

CONCLUDING REMARKS

Deng and Liu were the first considering the problem of hierarchical scheduling in open systems. They correctly identified the conditions under which an application that is schedulable on a dedicated slower processor, can be scheduled in the shared processor together with other applications. However, their approach has some limitations. First of all, it requires the knowledge of the worst-case execution time of each task. Although this assumption is reasonable for hard real-time applications, it is quite strong in open systems, for the reasons explained above. The second limitation is that the schedulability is possible only under certain restrictive conditions: non preemptive applications, predictable applications, or non predictable applications (within a given error). The algorithm presented in the next section overcomes such limitations and provides precise guarantees also to non predicable applications.

4.2.2 THE BANDWIDTH SHARING SERVER

The Bandwidth Sharing Server (BSS), proposed by Lipari et al. [LB99, Lip00, LCB00] [1], is a special kind of server that is able to handle multiple tasks. Its advantage with respect to Deng and Liu's algorithm is that the BSS does not require applications to obey any particular rule. However, it does require application tasks to be real-time, with a relative deadline D_i that can be hard or soft.

Each application $\mathcal{A}_i$ is handled by a dedicated *application server* S_i and is assigned a processor share U_i, with the assumption that the sum of the shares of all the applications in the system cannot exceed 1. The server maintains a queue of ready tasks: the ordering of the queue depends on the local scheduling policy.

Each time a task is ready to be executed in application $\mathcal{A}_i$, the server S_i calculates a budget B and a deadline d for the entire application. The active servers are then inserted in a global EDF queue, where the *global scheduler* selects the earliest deadline server to be executed. It will be allowed to execute for a maximum time equal to the server budget. In turn, the corresponding server selects the highest priority task in the ready queue to be executed according to the local scheduling policy.

[1] In [LCB00] the algorithm has been called PShED.

The server deadline is assigned by the server to be always equal to the deadline of the earliest-deadline task in the application. Notice that the task selected to be executed is chosen according to the local scheduler policy and might not be the earliest deadline task.

LIST OF RESIDUALS

To calculate the budget, every server uses a private data structure called *list of residuals*. For each task of an application $\mathcal{A}_i$, this list $\mathcal{L}_i$ contains one or more elements of the following type:

$$l = (B, d)$$

where d is the task's deadline and B is the budget available in interval $[a, d]$ (where a is the task's arrival time); that is, the maximum time that application $\mathcal{A}_i$ is allowed to demand in $[a, d]$.

Thus, an element l specifies for the interval $[a, d]$ the amount of execution time available in it. The goal of the server is to update the list such that in every interval of time the application cannot use more than its bandwidth. From now on, symbol $l_i(k)$ will denote the element in the k-th position of list $\mathcal{L}_i$.

List $\mathcal{L}_i$ is ordered by non-decreasing deadlines d. For the list to be consistent, the budgets must be assigned such that they are non-decreasing. Intuitively, this means that the total execution time allowed in an interval is never smaller than the execution time allowed in any contained interval.

The server assigns the application a pair (budget, deadline) corresponding to the element $l = (B, d)$ of the earliest deadline task in the application, regardless of the local scheduling policy. Only in the case the local scheduling policy is EDF, this element corresponds to the first task in the ready queue.

Two main operations are defined on this list: *adding* a new element and *updating* the list after some task has executed.

ADDING A NEW ELEMENT

A new element is created and inserted in the residual list when a newly activated task becomes the earliest deadline task among the ready tasks in the application. Let d_j be its deadline: first, the list is scanned in order to find the right position for the new element. Let k be such a position, that is:

$$\exists l_i(k-1), l_i(k) \quad d_{k-1} < d_j \leq d_k$$

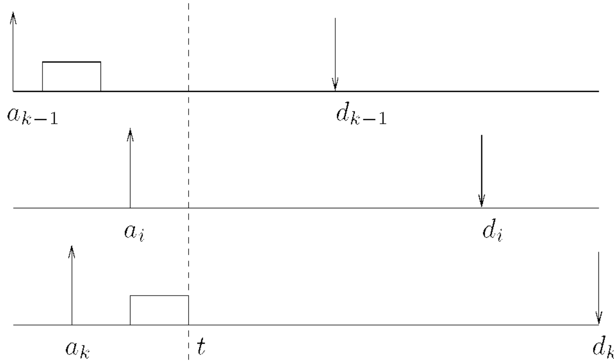


Figure 4.6 Computation of B_j

Now, the budget B_j is computed as:

$$B_j = min\{D_j U_i,\ (d_j - d_{k-1})U^{\mathcal{A}} + B_{k-1},\ B_k\} \tag{4.1}$$

where U_i is the bandwidth (share) assigned to application $\mathcal{A}_i$ and D_j is the task's relative deadline. At this point, the new element is completely specified as $l = (B_j, d_j)$ and can now be inserted at position k, so that the k-th element becomes now the $(k+1)$-th element, and so on.

The basic idea behind Equation (4.1) is that the budget for the newly arrived task must be constrained such that in any interval the application does not exceed its share. A typical situation is shown in Figure 4.6: when at time t task τ_j becomes the earliest deadline task, the algorithm must compute a new budget: it must not exceed the share in interval $[a_j, d_j]$, which is $D_j U_i$; it must not exceed the share in interval $[a_{k-1}, d_j]$ which is $B_{k-1} + (d_j - d_{k-1})U_i$, and must not exceed the share in interval $[a_k, d_k]$ which is B_k. It can be shown that, if B_j is the minimum among these values, then the application will not use more than its share in any other interval.

UPDATING THE LIST

Every time an application task is suspended or completes, the corresponding list must be updated. It could happen for any of the following reasons:

- the task has finished execution;

- the budget has been exhausted;
- the application has been preempted by another application with an earlier deadline.

Then, the algorithm picks the element in the list corresponding to the actual deadline of the server, say the k-th element, and updates the budgets in the following way:

$$\begin{array}{lllll} \forall l_j & j \geq k & B_j = B_j - e & & \\ \forall l_j & j < k \wedge B_j > B_k & \rightarrow & \textit{remove element } l_j \end{array}$$

DELETING ELEMENTS

We also need a policy to delete elements from the list whenever they are not necessary any longer. At time t, element $l_i(k)$ can be deleted if the corresponding task's instance has already finished and

- either $d_k \leq t$;
- or $B_k > (d_k - t)U_i$.

It can be seen from Equation 4.1 that in both cases element $l_i(k)$ is not taken into account in the calculation of the budget. In fact, suppose that element $l_i(j)$ is being inserted just after $l_i(k)$. Then

$$D_j U_i = (d_j - t)U_i < B_k + (d_j - d_k)U_i$$

and $B_k + (d_j - d_k)U_i$ cannot be chosen in the minimum. Suppose now that element $l_i(j)$ is being inserted just before $l_i(k)$. Then

$$D_j U_i = (d_j - t)U_i < (d_k - t)U_i < B_k$$

and B_k cannot be chosen in the minimum. Since $l_i(k)$ cannot contribute to the calculation of any new element, then it can be deleted safely.

EXAMPLE

To clarify the mechanism, consider the example in Figure 4.7, in which two applications are scheduled by the BSS algorithm: application $\mathcal{A}_1$ consists of two tasks, τ_1^1 and τ_2^1 and it is served by a server with a bandwidth of 0.5 and with a Deadline Monotonic scheduler. Application $\mathcal{A}_2$ consists of one task and it is served by a server with a

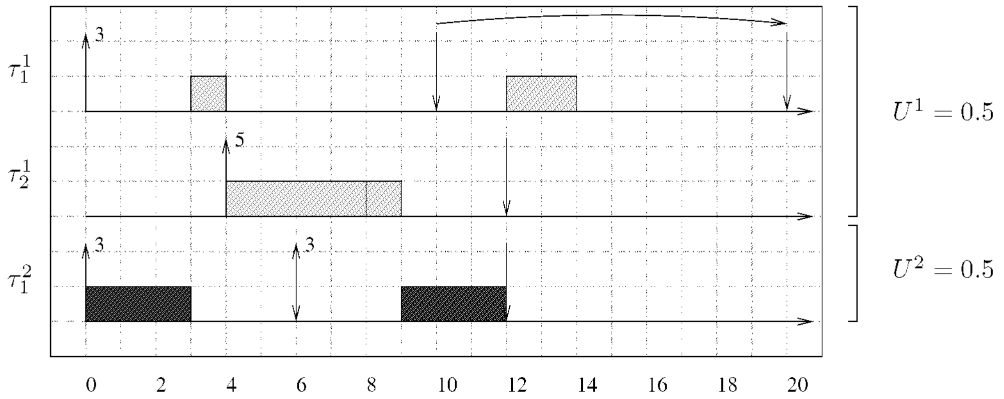


Figure 4.7 An example of schedule produced by the BSS algorithm: the two tasks τ_1^1 and τ_2^1 in application $\mathcal{A}_1$ are scheduled by Rate Monotonic.

bandwidth of 0.5 (since there is only one task, the local scheduling policy does not matter).

Let us concentrate our attention on application $\mathcal{A}_1$:

- an instance of task τ_1^1 arrives at time $t = 0$ with deadline $d_1^1 = 10$ and execution time (yet unknown) $c_1^1 = 3$. The server calculates a budget $B_1 = 5$ and inserts a new element in the residual list:

  $$\mathcal{L}_1 = \{(5, 10)\}.$$

  Then the server invokes the global scheduler. However, since the server of application $\mathcal{A}_2$ has an earlier deadline, the application is not executed until time $t = 3$;

- At time $t = 3$ the global scheduler signals the server of application $\mathcal{A}_1$ that it can execute;

- At time $t = 4$ an instance of task τ_2^1 arrives with deadline $d_2^1 = 12$ and an execution requirement of $c_2^1 = 5$. According to the DM scheduler, since task τ_2^1 has a smaller relative deadline than task τ_1^1, a *local preemption* is done. However, since the earliest deadline in application $\mathcal{A}_1$ is still $d_1^1 = 10$, the server budget and deadline are not changed.

- At time $t = 8$ the budget is exhausted: application $\mathcal{A}_1$ has executed for 5 units of time. The global scheduler suspends the server. The server first updates the list:

  $$\mathcal{L}_1 = \{(0, 10)\};$$

  then it postpones by $T = 10$ units of time: $d_1' = d_1 + 10 = 20$. Now the earliest deadline in the application is $d_2 = 12$, and the server calculates a new budget equal to:

  $$B_2 = (d_2 - d_1)U_1 + B_1;$$

  and inserts it into the list:

  $$\mathcal{L}_1 = \{(0, 10); (1, 12)\};$$

  Finally, it invokes the global scheduler. Since it is again the earliest deadline server in the global ready queue, it is scheduled to execute.

- At time $t = 9$ task τ_2^1 finishes. The server updates the list:

  $$\mathcal{L}_1 = \{(0, 10); (0, 12)\};$$

  Now the earliest deadline in application $\mathcal{A}_1$ is $d_1' = 20$. Then the server calculates a new budget and inserts it into the list:

  $$\mathcal{L}_1 = \{(0, 10); (0, 12); (4, 20)\};$$

  finally, it invokes the global scheduler. Since it is not the earliest deadline server, another server is scheduled to execute.

It is important to notice that the earliest deadline in the application has been postponed, and this deadline can in general be different from the deadline of the executing task. Notice also that this framework is very general: basically it is possible to choose any kind of local scheduler. In particular, we can let tasks share local resources with any concurrency control mechanism, from simple semaphores to the more sophisticated Priority Ceiling Protocol or Stack Resource Policy.

FORMAL PROPERTIES

The BSS algorithm has two important properties. The *Bandwidth Isolation Property* says that, independently of the local scheduling algorithm, the execution times and the arrival rates of the tasks, no application misses its current server's deadline. In other words, the BSS provides the same kind of temporal protection property as the one provided by the CBS algorithm (see Chapter 3).

The *Hard Schedulability Property* permit us to guarantee a priori a hard real-time application scheduled by the BSS. The schedulability analysis for an application depends on the local scheduling algorithm. In [LB99, Lip00], schedulability conditions have been presented for the following local schedulers:

Earliest Deadline First: Application $\mathcal{A}_1$, which consists of periodic hard real-time periodic tasks, is schedulable if and only if:

$$\forall L > 0, \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \leq U^{\mathcal{A}} L$$

where C_i, D_i and T_i are the worst-case execution time, the relative deadline and the period for task i, respectively.

Rate Monotonic: Application $\mathcal{A}_1$, which consists of periodic or sporadic tasks with deadlines equal to periods, is schedulable if:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq T_i U^{\mathcal{A}}.$$

Stack Resource Policy with EDF: Application $\mathcal{A}_1$, which consists of periodic tasks with deadlines equal to periods, is schedulable if:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq U^{\mathcal{A}}$$

where B_i is the maximum blocking factor for task i.

Notice that the proposed schedulability tests are similar to the equivalent schedulability tests for a virtual dedicated processor of speed $U^{\mathcal{A}}$. Unfortunately, the BSS cannot provide perfect equivalence between the schedule in the dedicated processor and the schedule in the shared processor. In particular, it has been shown [Lip00] that an application schedulable by fixed priority on a dedicated processor of speed $U^{\mathcal{A}}$ may be unschedulable (i.e. some deadline could be missed) in the shared processor when served by a server with bandwidth $U^{\mathcal{A}}$.

COMPLEXITY

The BSS algorithm is quite complex to implement, because a linear list of budgets has to be kept for each single server. The complexity of the algorithm depends on the maximum number of elements that can be present in this list. It has been proved that if the application consists of hard real-time periodic task, the length of the list is at most equal to the number of tasks in the application. However, even in this case, the time spent by the algorithm can be quite high. Lipari and Baruah proposed a data structure, called *Incremental AVL tree* to reduce the time needed to update the list. With the new data structure, the complexity is now $O(logN)$, where N is the number of elements in the list.

CONCLUDING REMARKS

The BSS algorithm presented above, like the Deng and Liu's algorithm, uses information about the tasks to compute the server budget for the entire application. For this reason, we can classify these algorithm as *intrusive* algorithms. They have the following limitations:

- It may not be possible to have enough information on the application tasks. For example, an application may contain some non-real-time task for which it may be impossible to derive a deadline or the worst-case execution time.
- The strong interaction between the local scheduler and the global scheduler (i.e., the server mechanism) makes it difficult to implement such algorithms. It would be better to completely separate the local scheduler from the global scheduler.

For these reasons, researchers recently concentrated their efforts in a different direction, as explained in the following section.

4.3 PARTITION-BASED APPROACHES

In this section, we first introduce a general methodology to study the problem of partitioning a resource among different applications, and then we present methodologies to: a) analyze the schedulability of a real-time application in isolation from the others; and b) compute the "optimal" server parameters to make the application schedulable.

4.3.1 RESOURCE PARTITIONING

A general approach to the analysis of hierarchical scheduling systems based on the concept of *resource partitioning* was first proposed by Feng et al. [MFC01, FM02, MF01]. A resource partition is a periodic sequence of disjoint time intervals. An application that is assigned a resource partition can only execute in the partition time intervals.

Definition 4.1 *A periodic resource partition* Π *is a tuple* (Γ, P)*, where* Γ *is an array of* N *time pairs* $\{(S_1, E_1), (S_2, E_2), \ldots, (S_N, E_N)\}$ *that satisfies* $(0 \leq S_1 < E_1 < S_2 < E_2 < \ldots < S_N < E_N \leq P)$*, for some* $N \geq 1$*, and* P *is the partition period. The physical resource is available to a task group executing on this partition only during time intervals* $(S_i + jP, E_i + jP)$*,* $1 \leq i \leq N, j \geq 0$.

The following cases provide two examples of resource partition:

- If the global scheduler is an off-line algorithm, as in TDMA approaches, the time line is divided into slots, and each slot is assigned to a different application. In this context, the resource partition is the sequence of slots assigned to an application;
- If the global scheduler is an on-line algorithm (as for the Deng and Liu's algorithm, or the BSS algorithm), the partition for each application is computed dynamically depending on the sequence of arrival times and computation requirements of the application tasks.

Notice that the partition generated by an on-line algorithm may not be periodic, as it depends on the arrival times and execution times of the tasks. For the sake of clarity, we first introduce some definitions that apply to static periodic partitions and then generalize them to dynamic non-periodic partitions.

Definition 4.2 *The availability factor of a resource partition* Π *is a number* $\alpha(\Pi)$ *such that* $\alpha(\Pi) = (\sum_{i=1}^{N}(E_i - S_i))/P$.

Definition 4.3 *The supply function* $S(t)$ *of a partition* Π *is the total amount of time that is available in* Π *from time* 0 *to time* t.

Definition 4.4 *The Least Supply Function (LSF)* $S^*(t)$ *of a resource partition* Π *is the minimum of* $(S(t + d) - S(d))$ *where* $t, d \geq 0$

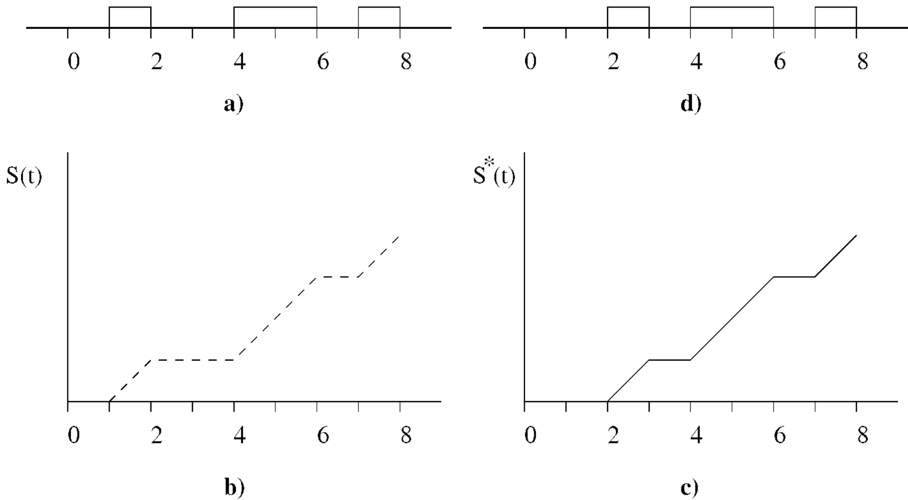


Figure 4.8 Example of a resource partition: a) the partition; b) the Supply Function; c) the Least Supply Function; c) the critical partition.

Definition 4.5 *A critical partition of a resource partition* $\Pi = (\Gamma, P)$ *is* $\Pi^* = (\Gamma^*, P)$ *where* Γ^* *has time pairs corresponding to the steps in* $S^*(t)$ *such that* Π^**'s supply function equals* $S^*(t)$ *in* $(0, P)$.

Example. Consider the resource partition $\Pi = (\{(1, 2), (4, 6), (7, 8)\}, 8)$. The partition is depicted in Figure 4.8a. Its supply function $S(t)$ is shown with a dashed line in Figure 4.8b. The availability factor is $\alpha(\Pi) = 0.5$. The LSF for the partition is depicted with a tick line in Figure 4.8c. Notice that $S^*(t)$ is always below $S(t)$, but $S(P) = S^*(P)$. Finally, Figure 4.8d shows the critical partition Π^*.

Given these definitions, Feng et al. proposed a feasibility analysis of an application with a local scheduler. Their basic idea is that an application is schedulable if all tasks complete within their deadlines, even when the critical instant is coincident with the beginning of a critical partition. This is formally stated in the following theorem.

Theorem 4.3 *Suppose a preemptive fixed priority scheduling policy is used to schedule an application on a partition by some priority assignment where all deadlines are no larger than the corresponding periods. If a task's first request is schedulable in the critical partition, then the task is schedulable in the partition.*

A similar theorem holds for earliest deadline first local scheduling.

4.3.2 BOUNDED-DELAY MODEL

The previous definitions and theorems are useful in the case the partitions are pre-computed by some off-line algorithm and are periodically repeated on line. Such a scheme is very predictable and allows "tailoring" the partition to the application needs.

When the global scheduler is an on-line algorithm (as the CBS server described in Chapter 3, or any other algorithm that provides temporal isolation), the resource partition model can be generalized to take the on-line partitioning of the resource into account. First of all, the partition needs not to be periodic. The following definition generalizes the concept of partition.

Definition 4.6 *A resource partition is an "allocation function" $\Pi(t)$ that has values in $\{0, 1\}$. If $\Pi(t) = 1$, then the resource is allocated to the corresponding application. If $\Pi(t) = 0$, the resource is not allocated to the application and cannot be used. A partition is periodic if there exists a $P > 0$ such that $\Pi(t) \equiv \Pi(t + P)$.*

The concepts of supply function and least supply function can easily be extended to the case of non-periodic partitions. Some additional care is needed for the availability factor.

Definition 4.7 *The availability factor α of a partition is defined as:*

$$\lim_{x \to \infty} \frac{\int_0^x \Pi(t)dt}{x}$$

Now, we want to characterize all possible partitions that are generated by an on-line algorithm. The idea is based on the observation that a resource partition is characterized by two important parameters: the availability factor α and the partition delay Δ. The latter is defined as follows:

Definition 4.8 *The partition delay Δ of partition Π is the smallest d such that*

$$\forall t, S^*(t) - (\alpha t - d)_0 \geq 0$$

where $(x)_0$ is a short form for $\max(0, x)$.

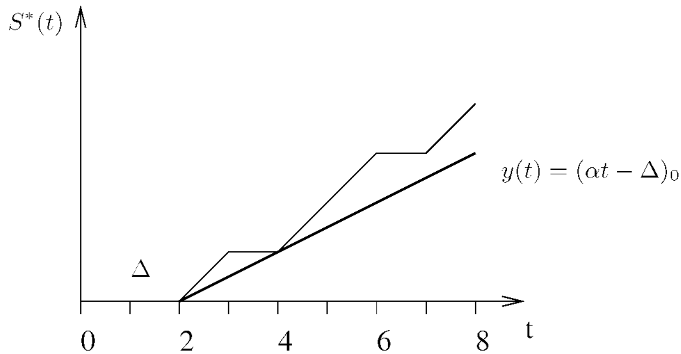


Figure 4.9 Relationship between α, Δ and the least supply function.

Notice that, in the definition of partition delay, we use the least supply function $S^*(t)$. This means that Δ is the maximum interval of time in which an application does not receive any service. Figure 4.9 shows the relationship between α, Δ and the least supply function for the partition of the previous example. In particular, function $y = (\alpha t - \Delta)_0$ is always below the least supply function, and there is at least one point in which they are coincident.

For any partition, it is possible to find its availability factor α and its delay Δ. Viceversa, for each pair (α, Δ), there is more than one partition with availability factor equal to α and delay equal to Δ. Therefore, the pair (α, Δ) defines a set of partitions. We now consider the class of all partitions with availability factor equal to α and delay *less than or equal to* Δ.

Definition 4.9 *The (α, Δ)-partition class is the class of all partitions with availability factor equal to α and delay not greater than Δ.*

The class of partitions generated by an on-line algorithm like the CBS (or any similar algorithm that provides resource reservation and temporal protection) is of particular interest. As we will see in the following, there is a direct relation between the parameters (α, Δ) and the parameters Q and P of the server.

Moreover, given (α, Δ) assigned to an application and its local scheduling algorithm, it is possible to check whether the application is schedulable (see next section). Also, given an application and its local scheduling algorithm, it is possible to compute all possible values of (α, Δ) that make the application schedulable.

4.3.3 PARTITION CLASS OF A SERVER

In this section we consider a particular version of the CBS algorithm, known as "hard reservation server". The main difference with the original algorithm is in the way the server behaves when the budget is exhausted. In the original algorithm (see Section 3.6.1), when the budget is depleted, it is immediately recharged to Q and the deadline is postponed to $d = d + P$. In this way, the algorithm tries to take advantage of the EDF rule: if the server deadline is still the earliest one, the server continues to execute and has a greater chance to meet its deadline.

In the "hard reservation" version, when the budget is exhausted, the server is suspended and the budget will be replenished at the server deadline d. At the replenishment time, the server budget is recharged to its maximum value Q and the server deadline is set to $d = d + P$. Notice that, by introducing this rule, the algorithm becomes non-work conserving. The hard reservation rule, however, is important to bound the maximum delay of a partition generated by a server.

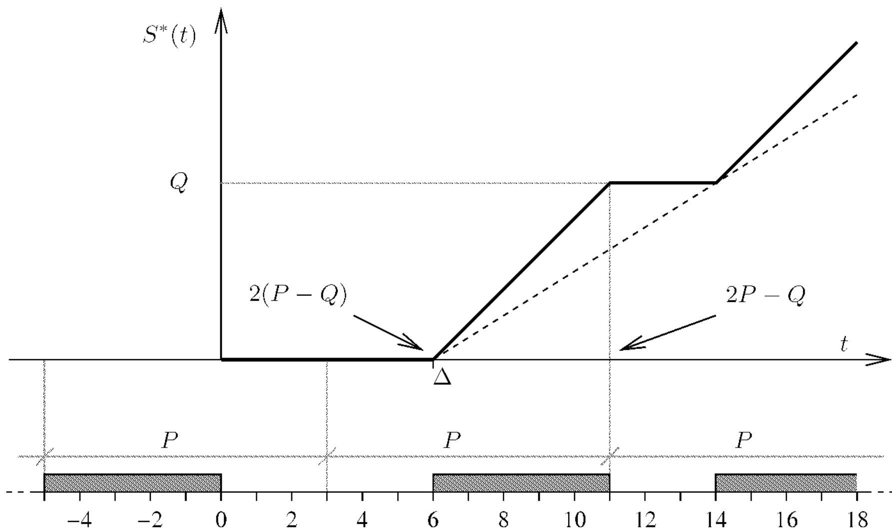


Figure 4.10 Worst-case partition generated by a server.

The following theorem identifies the partition with the maximum delay that can be generated by a server.

Theorem 4.4 *Given a CBS server with the hard reservation rule, and with parameters (Q, P), if $k = \left\lceil \frac{t-(P-Q)}{P} \right\rceil$, the partition with the maximum delay that it can generate has the following least supply function $S^*(t)$:*

$$S^*(t) = \begin{cases} 0 & \text{if } t \in [0, P-Q] \\ (k-1)Q & \text{if } t \in (kP-Q, (k+1)P-2Q] \\ t-(k+1)(P-Q) & \text{otherwise} \end{cases} . \qquad (4.2)$$

Proof.
We have to compute the worst-case allocation provided by the server for every interval of time. Consider an interval starting at time t, when a new request for execution arrives from the application. There are 2 possibilities:

case a. The server is inactive and a new request is activated at time t, with $q > (d-t)U$. In this case, a new budget $q = Q$ and a new deadline $d = t + P$ are computed. The worst-case allocation is depicted in Figure 4.11a.

case b. The server is *active* at time t, (or it is inactive and $q \leq (d-t)U$) and it has already consumed x units of budget. In this case, the worst possible situation is when the server is preempted by the global scheduler until time $t = d - (Q - x)$. The worst-case allocation is depicted in Figure 4.11b, and is minimum for $x = Q$.

By comparing the two cases, it is clear that case b, with $x = Q$, is the most pessimistic. The corresponding function is $S^*(t)$, as given by Equation (4.2).

□

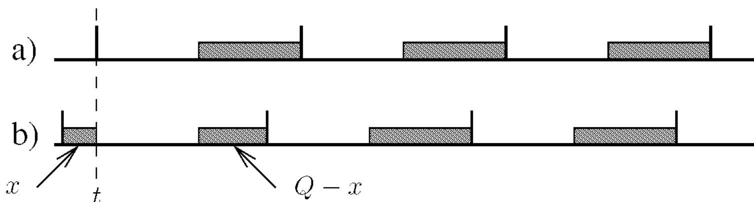


Figure 4.11 Worst-case allocation for the server.

Corollary 4.1 *All the partitions that a CBS server with parameters (Q, P) can possibly generate are in the class (α, Δ), where $\alpha = \frac{Q}{P}$ and $\Delta = 2(P - Q)$.*

4.3.4 SCHEDULABILITY CONDITIONS

To find a schedulability condition for an application scheduled on a partition with its own scheduling algorithm, we must compute the "demand" of the application in every interval of time. If the worst-case supply of the corresponding partition (given by the least supply function) is always greater than the demand of the application in every interval of time, then the application is schedulable.

Similar methodologies have been proposed by many authors. Saewong et al. [SRLK02] extended the response time analysis for fixed priority systems to the case of hierarchical schedulers. Their model assumes a Deferrable Server [SLS95] as a basic mechanism to partition the processor, and a fixed priority algorithm as a local scheduler.

Shin and Lee [SL03] proposed a more general framework for schedulability analysis of hierarchical scheduling, based on the Feng and Mok's model. They do not assume any particular global scheduling mechanism, as long as the global scheduler is able to provide a *periodic resource abstraction*. Such an abstraction can be provided by any hard reservation server mechanism. They also proposed a schedulability analysis for a local schedulers based on EDF and on fixed priorities.

Theorem 4.5 *Let $\mathcal{A}$ be a set of periodic or sporadic tasks $\{\tau_1, \ldots \tau_n\}$, with $\tau_i = (C_i, T_i, D_i)$, where C_i is the worst-case computation time, T_i is the task period and D_i is the task relative deadline. This task set is schedulable by the EDF scheduling algorithm on a resource partition with least supply function $S^*(t)$ if an only if:*

$$\forall\, 0 < t \leq 2H : dbf(t) \leq S^*(t)$$

where $H = lcm(T_1, \ldots, T_n)$ is the hyperperiod of $\mathcal{A}$ and $dbf(t)$ is the processor demand bound function, defined as:

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

Definition 4.10 *The maximum service time for a periodic resource abstraction is a function $tbf(t)$ that represents the maximum amount of time that it takes to receive a service equal to t. For a periodic resource abstraction (Q, P),*

$$tbf(t) = (P - Q) + P \left\lfloor \frac{t}{Q} \right\rfloor + \epsilon(t)$$

$$\epsilon(t) = \begin{cases} P - Q + t - Q \left\lfloor \frac{t}{Q} \right\rfloor & \textit{if} \left(t - Q \left\lfloor \frac{t}{Q} \right\rfloor > 0 \right) \\ 0 & \textit{otherwise} \end{cases}$$

Theorem 4.6 *Let $\mathcal{A}$ be a set of periodic or sporadic tasks $\{\tau_1, \ldots \tau_n\}$, with $\tau_i = (C_i, T_i, D_i)$, where C_i is the worst-case computation time, T_i is the task period and D_i is the task relative deadline. Assume that tasks are ordered by decreasing priorities. This task set is schedulable by fixed priority on a resource partition with maximum service time $tbf(t)$ if an only if:*

$$\forall i = 1, \ldots, n \quad R_i \leq D_i$$

where the response time R_i can be computed with the following iterative procedure:

$$\begin{cases} R_i^{(0)} = C_i \\ R_i^{(k)} = tbf(I_i^{(k)}) \end{cases}$$

where:

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

The iteration stops when $R_i^{(k+1)} = R_i^{(k)}$ or when $R_i^{(k)} > D_i$.

4.3.5 DERIVING THE SERVER PARAMETERS

In this section we present a methodology for deriving the "optimal" server parameters given the application and the local scheduling algorithm. This problem is the inverse of the schedulability problem: rather than testing the schedulability, we want to actively derive the "best" parameters that guarantee schedulability.

Bini and Lipari [LB03] proposed a methodology for the case of a fixed priority scheduler. It can be easily extended to other schedulers, like EDF, but we leave this extension to the reader.

Let us first tackle the problem of finding the minimum processor speed that maintains the task set schedulable. Slowing down the processor speed by a factor $\alpha \leq 1$ is equivalent to scale up the computation times by $1/\alpha$:

$$\forall i = 1, \ldots, n \quad \widetilde{C}_i = C_i/\alpha. \tag{4.3}$$

The problem is to find the minimum speed $\alpha_{\min}$, keeping the system schedulable. Bini and Buttazzo [BB02] found a new way to express the schedulability condition under a fixed priority scheduling algorithm as a set of linear inequalities in the computation times C_i.

Theorem 4.7 (Theorem 3 in [BB02]) *A set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of tasks with decreasing priorities is schedulable by a fixed priority algorithm if and only if:*

$$\bigwedge_{i=1\ldots n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t \tag{4.4}$$

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

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t). \end{cases} \tag{4.5}$$

By introducing the speed factor α, we can reformulate condition (4.4) taking into account the substitution given by Equation (4.3). The result is the following:

$$\bigwedge_{i=1\ldots n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} \leq t$$

$$\bigwedge_{i=1\ldots n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \frac{1}{\alpha} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t$$

$$\bigwedge_{i=1\ldots n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \alpha \geq \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t}$$

and finally:

$$\alpha \geq \alpha_{\min} = \max_{i=1\ldots n} \min_{t \in \mathcal{P}_{i-1}(T_i)} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \tag{4.6}$$

where $\alpha_{\min}$ is the minimum processor speed that guarantees the feasibility of the task set.

We now introduce the delay Δ in the analysis. In fact, when a task set is scheduled by a server, there can be a delay in the service because the server is not receiving any execution time from the global scheduler. To extend the previous result to the case when $\Delta > 0$ we need to look at Equation (4.6) from a different point of view. Figure 4.12 illustrates the worst-case workload for a task τ_i, called $W_i(t)$, and the line $\alpha_{\min} t$. The line represents the amount of time that a processor with speed $\alpha_{\min}$ provides to the task set. Task τ_i is schedulable if

$$\exists t^* \in (0, D_i] : \quad \alpha_{\min} t^* \geq W_i(t^*).$$

The presence of a delay Δ prevents us to allocate time slots for an interval of length Δ. This interval can start, in the worst case, at the critical instant for task τ_i, that is, when τ_i and all higher priority tasks are released. It follows that the time provided by the server is bounded from below by function $y(t) = (\alpha t - \Delta)_0$. Figure 4.12 also shows different functions $y(t)$ for different values (α, Δ). Therefore, when introducing Δ, task τ_i is schedulable on a server characterized by function $y(t)$, if:

$$\exists t^* \in (0, D_i] : \quad (\alpha t^* - \Delta)_0 \geq W_i(t^*). \tag{4.7}$$

Notice that, as Δ increases, the tangent point t^* may change. By using Equation (4.7), and increasing Δ we can find all possible α that make the task τ_i schedulable.

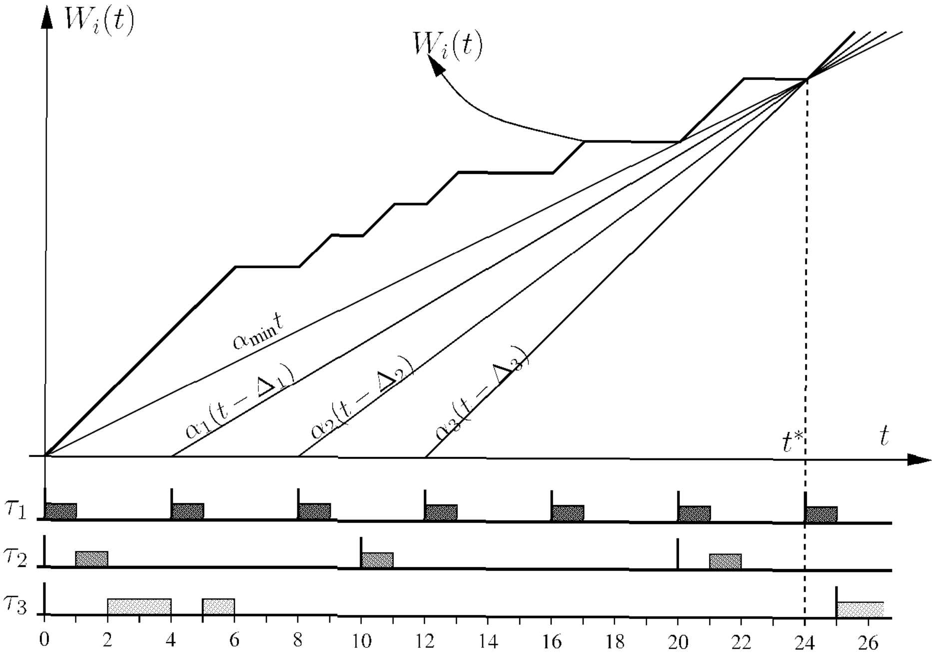


Figure 4.12 Workload and the α_{min} speed.

In order to find a closed formulation for the relation between α and Δ expressed by Equation (4.7), we need the following Lemma proved in [BB02].

Lemma 4.1 (Lemma 4 in [BB02]) *Given a task subset $\mathcal{T}_i = \{\tau_1, \ldots, \tau_i\}$ schedulable by fixed priorities and the set $\mathcal{P}_i(b)$ as defined in Equation (4.5), the workload $W_i(d)$ is*

$$W_i(d) = \min_{t \in \mathcal{P}_i(d)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j + (d - t).$$

By means of this lemma, the well known schedulability condition for the task set:

$$\forall i = 1 \ldots n \quad C_i + W_{i-1}(D_i) \leq D_i$$

can be rewritten as follows:

$$\forall i = 1 \ldots n \quad C_i + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j + (D_i - t) \leq D_i. \tag{4.8}$$

When the task set is served by a server with function $y(t) = (\alpha t - \Delta)_0$, the schedulability condition expressed by Equation (4.8) becomes the following:

$$\forall i = 1 \ldots n \quad \Delta + \frac{C_i}{\alpha} + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} + (D_i - t) \leq D_i \tag{4.9}$$

Since the link between (α, Δ) is now explicit, we can manipulate the previous expression to obtain a direct relationship between α and Δ. In fact, the schedulability condition of the single task τ_i can be written as:

$$\Delta \leq D_i - \left(\frac{C_i}{\alpha} + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} + (D_i - t) \right)$$

and, simplifying the expression, we finally obtain:

$$\Delta \leq \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha} \left(C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \right). \tag{4.10}$$

To take into account the schedulability of all the tasks in the set (and not only τ_i as done so far), this condition must be true for every task. Hence, we obtain the following theorem.

Theorem 4.8 *A task set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is schedulable by a server characterized by the lower bound function $(\alpha t - \Delta)_0$ if:*

$$\Delta \leq \min_{i=1\ldots n} \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha}\left(C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j\right) \tag{4.11}$$

where:

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

Proof.
If Δ satisfies Equation (4.11), then it satisfies all the equations (4.10) for every task in the set. Then every task is schedulable on such a local scheduler and so the whole set is, which proves the theorem. □

How to choose Q and P. In our process of designing a server for an application $\mathcal{A}$, the first step is to characterize the application by specifying all the individual task parameters. Once this step is carried out, by applying Theorem 4.8, a class of (α, Δ) pairs is obtained. On this class, which guarantees by definition the schedulability of application $\mathcal{A}$, we perform the server selection by optimizing a desired cost function. One possible cost function is the overhead of the scheduler. In fact, when choosing the server parameters, we must balance two opposite needs:

1. the required bandwidth should be small, not to waste the total processor capacity;
2. the server period should be large, otherwise the time wasted in context switches performed by the global scheduler would be too high.

Thus, a typical cost function to be minimized may be the following:

$$k_1 \frac{T_{\text{overhead}}}{P} + k_2 \alpha \tag{4.12}$$

where T_{overhead} is the global scheduler context switch time, P is the server period, α is the fraction of bandwidth, and k_1 and k_2 are two designer defined constants. Moreover, some additional constraints in the (α, Δ) domain, other than those specified by Equation (4.11), may be required. For example, if we use a fixed priority global

scheduler, to maximize the resource utilization we could impose the server periods to be harmonic.

To clarify the methodology, let us consider the following example. Suppose we have an application $\mathcal{A}$ consisting of three tasks with parameters reported in Table 4.1 (for simplicity, we choose $D_i = T_i$, but the approach is the same when $D_i < T_i$). The utilization is $U = 1/4 + 1/10 + 3/25 = 47/100$, hence α cannot be smaller than 0.47. The schedule corresponding to the worst-case scenario (i.e., the critical instant) when the application is scheduled alone on the processor is shown in Figure 4.13.

i	T_i	C_i	D_i
1	4	1	4
2	10	1	10
3	25	3	25

Table 4.1 Task set parameters for application $\mathcal{A}$.

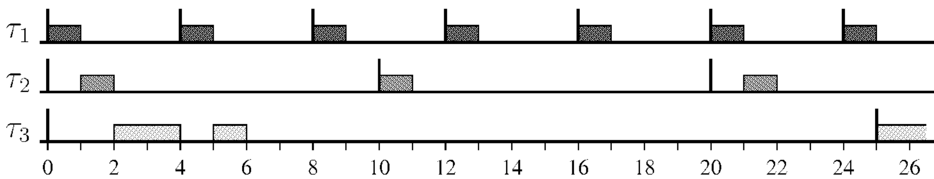


Figure 4.13 Worst-case schedule of application $\mathcal{A}$.

By expanding Equation (4.10) for τ_1 we obtain the following inequality:

$$\mathcal{P}_0(4) = \{4\}$$
$$\Delta \le 4 - 1/\alpha.$$

Doing the same for τ_2, we obtain:

$$\mathcal{P}_1(10) = \{8, 10\}$$
$$\Delta \le \max\{8 - 3/\alpha, 10 - 4/\alpha\}$$

and, finally, for the last task τ_3:

$$\mathcal{P}_2(25) = \{20, 24, 25\}$$
$$\Delta \le \max\{20 - 10/\alpha, 24 - 12/\alpha, 25 - 13/\alpha\}$$
$$\Delta \le 24 - 12/\alpha.$$

In order to make all the three tasks schedulable, all the inequalities must hold at the same time, as stated in Theorem 4.8. It follows that:

$$\Delta \leq \min\{4 - \frac{1}{\alpha}, \max\{8 - \frac{3}{\alpha},\ 10 - \frac{4}{\alpha}\},\ 24 - \frac{1}{2}\alpha\}. \tag{4.13}$$

Figure 4.14 illustrates the set of (α, Δ) pairs defined by Equation (4.13) as a gray area whose upper boundary is drawn by a thick line. This boundary is a piece-wise hyperbole, because it is the minimum between inequalities, each one of them is an hyperbole (see Equations (4.10) and (4.11)). Notice that, in this particular case, the schedulability condition for task τ_2 does not provide any additional constraint.

Figure 4.14 also shows a qualitative cost function that increases as α increases, and decreases as Δ increases (see Equation 4.12). If we minimize this qualitative function on the domain expressed by Equation (4.13), the solution is $\alpha = 11/20$ and $\Delta = 24/11$. We can now find the period P and the budget Q of the server corresponding to the selected solution:

$$\Delta = 2(P - Q) \qquad \alpha = \frac{Q}{P}$$

then:

$$P = \frac{\Delta}{2(1-\alpha)} \qquad Q = \alpha P$$

By substitution, we obtain the server parameters: $P = 80/33 \simeq 2.424$ and $Q = 4/3 \simeq 1.333$.

Finally, Figure (4.15) shows the schedule for the sample application, obtained by considering the worst-case scenario both for the time requested by tasks and for the time provided by the server. The shaded areas represent intervals where the server does not receive any allocation by the global scheduler. As expected, all tasks complete within their deadlines.

4.4 CONCLUDING REMARKS AND OPEN PROBLEMS

In this chapter, we discussed the problem of scheduling different applications in the same system, an application being a set of concurrent tasks handled by a dedicated customized scheduler. Then, we presented some algorithms to deal with this problem. These algorithms are based on the concept of "hierarchical scheduling": a global scheduler allocates the resource to the different applications, and a local scheduler for each application selects the task to be executed. We first presented the algorithm of Deng and Liu [DLS97, DL97] and the BSS algorithm: they both assume the knowledge

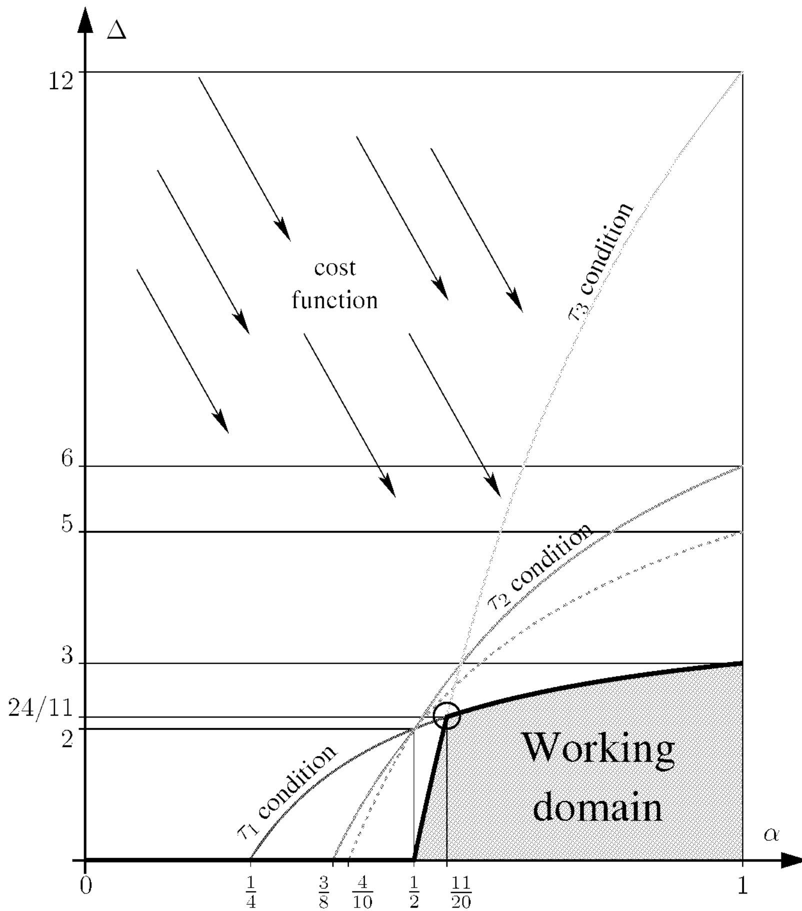


Figure 4.14 Server parameters in the (α, Δ) domain.

of the parameters of the applications' tasks to allocate the resource to the different applications. Then, we presented a more general approach that consists in using a resource reservation algorithm (like the CBS) at the global level. This approach seems the most promising, as the global scheduler does not need to know the internal details of the applications to be able to provide timing guarantees.

There are still some open issues that need to be addressed. One important question is whether it is possible to provide an "optimal" non fluid algorithm that can be im-

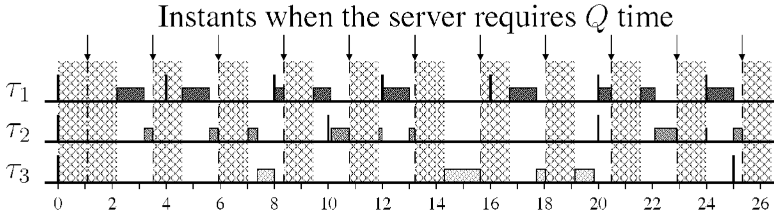


Figure 4.15 Worst-case schedule of application $\mathcal{A}$ on a server with the computed parameters.

plemented in a real system. More formally: given N applications $\mathcal{A}_1, \ldots, \mathcal{A}_N$, each schedulable on a dedicated processor of speed U_i with some scheduling algorithm S_i, and with $\sum_{i=1}^{N} U_i \leq 1$, is there a non-fluid algorithm able to schedule all applications in a shared processor of speed $U = 1$?

Of course, the GPS algorithm is ruled out, since it is a fluid algorithm that cannot be implemented in practice. We have seen that Deng and Liu's algorithm and the BSS algorithm are not able to provide such a property: both algorithm require that some extra capacity is assigned to each application in order to guarantee it.

Another problem that needs to be solved is how to synchronize two applications. For example, if two applications need to communicate through mutually exclusive resources, each application will experience some blocking time that must be taken into account in the schedulability test. Although a first proposal has been done by Kuo and Li [KL99] to extend Deng and Liu's algorithm, more research is needed to extend the other approaches.

5

SYNCHRONIZATION PROTOCOLS FOR HARD AND SOFT REAL-TIME SYSTEMS

Most of the real-time scheduling algorithms presented in the previous chapters assume that tasks cannot self-suspend and cannot use blocking primitives. These assumptions are quite restrictive because, in practice, tasks often use synchronous operating system calls that introduce blocking. For example, if tasks exchange data via shared memory buffers, access to shared memory has to be protected by mutually exclusive (mutex) semaphores to prevent possible inconsistencies in the data structures. However, if a task blocks on a mutex semaphore, the task model considered in the previous chapters is not valid anymore.

In hard real-time systems, the problem of blocking on a mutex semaphore can be solved by using appropriate synchronization protocols that bound the blocking time of the tasks. The extension of these protocols to soft real-time systems and, in particular, to reservation based real-time systems is not straightforward.

In this chapter, we first describe the problem caused by mutual exclusion and briefly describe some background work on real-time protocols for accessing mutually exclusive resources. Then, we present protocols and scheduling algorithms that extend the resource reservation framework to the case of mutex semaphores. We will discuss two different approaches. In the first approach, we extend the CBS algorithm to systems consisting of hard real-time periodic tasks and soft real-time aperiodic tasks that can share resources. The resulting algorithm is called CBS-R. Then, we consider a more general model of an open system, where tasks can be dynamically activated in the system, and we present the Bandwidth Inheritance (BWI) protocol.

5.1 TERMINOLOGY AND NOTATION

We consider a set $\mathcal{R}$ of r resources, a set $\mathcal{T}^{\mathcal{P}}$ of n hard periodic (or sporadic) tasks, and a set $\mathcal{T}^{\mathcal{A}}$ of m soft aperiodic tasks that have to be executed on a uniprocessor system. Tasks are preemptable and all the resources are accessed in exclusive mode using critical sections guarded by mutually exclusive semaphores. To simplify our presentation, only single-unit resources are considered, however the results can easily be extended to deal with multi-unit resources, as described in [Bak91]. Every resource is assigned a different semaphore. Therefore, we will often refer to a resource or to its corresponding semaphore with the same symbol $R_j \in \mathcal{R}$.

A critical section is a fragment of code that starts with a lock operation on a semaphore R_j and finishes with an unlock operation on the same semaphore. We denote the lock and unlock operation with $P(R_j)$ and $V(R_j)$ respectively. Critical sections can be *nested*, that is, it is possible to access resource R_j while holding the lock on resource R_k. We assume *properly nested* critical sections, so that a sequence of code like $P(R_1), \ldots, P(R_2), \ldots, V(R_2), \ldots, V(R_1)$ is permitted, whereas a sequence like $P(R_1), \ldots, P(R_2), \ldots, V(R_1), \ldots, V(R_2)$ is not permitted. *Internal* critical sections are nested inside other critical sections, whereas *external* critical sections are not. We denote the worst-case execution time of the longest critical section of task τ_i on resource R_k as $\xi_i(R_k)$. Note that $\xi_i(R_k)$ comprises the execution time of all the internal critical sections. We also assume that if a job performs a lock operation on semaphore R_k, it performs the corresponding unlock operation before its completion. Therefore, a critical section cannot span through two consecutive jobs.

5.2 SHARED RESOURCE IN REAL-TIME SYSTEMS

If a real-time task uses mutex semaphores, a particular problem can arise, referred in the literature as the *priority inversion problem*. This problem has become very famous in the real-time community because it was reported to happen in the NASA space mission called "Mars Pathfinder". A small robot, the Sojourner rover, was sent to Mars to explore the martian environment and collect useful information. The on-board control software consisted of many software threads, scheduled by a fixed priority scheduler. One high priority thread and one low priority thread were using the same software data structure through a shared semaphore[1]. At some instant, it happened that the low priority thread was interrupted by medium priority threads while blocking the high priority thread on the semaphore. The situation was very similar to the one depicted in Figure 5.1.

[1] The semaphore was actually used by a library that provided high level communication mechanisms among threads, namely the `pipe()` mechanism.

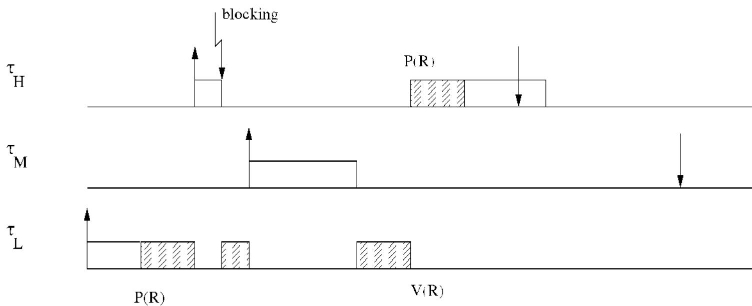


Figure 5.1 An example of priority inversion.

The low priority task τ_L acquires the lock on the resource R with the lock operation $P(R)$. Immediately after, it is preempted by the high priority task τ_H, which tries to lock the same resource and it is blocked. Therefore, τ_L resumes execution. Before being able to release the lock, τ_L is preempted by a medium priority task τ_M. As a consequence, the high priority task τ_H must now wait for τ_M to complete.

At the time of the Mars Pathfinder mission, the problem was already known. The first accounts of the problem and possible solutions date back to early '70s. However, the problem became widely known in the real-time community since the seminal paper of Sha, Rajkumar and Lehoczky [SRL90], who presented the Priority Inheritance Protocol and the Priority Ceiling Protocol to bound the time a real-time task can be blocked on a mutex semaphore.

Later, other similar protocols have been presented in the literature. Among the others, we would like to mention the Stack Resource Policy [Bak90, Bak91], which will be briefly recalled in 5.3.3, and the Dynamic Priority Ceiling [CL90].

5.3 SYNCHRONIZATION PROTOCOLS FOR HARD REAL-TIME SYSTEMS

5.3.1 PRIORITY INHERITANCE PROTOCOL

The Priority Inheritance Protocol (PIP) was first presented in [SRL90] to solve the priority inversion problem. According to this protocol, when a high-priority task τ_H is blocked on the entrance of a critical section, the low-priority task τ_L that holds the

lock on the resource *inherits* the priority of τ_H. When τ_L unlocks the resource and no other task is blocked by τ_L, it returns to its nominal priority. In general, a task τ_i always executes with a priority equal to the maximum between its nominal priority and the maximum priority of the tasks it is blocking.

Even though the PIP was developed in the context of fixed priority scheduling, it can also be applied to dynamic priority scheduling (its extension has been done by Spuri [SRBS98]). The following basic properties hold:

- *A task τ_i can be blocked for at most the duration of one critical section for each lower priority task, regardless of the number of semaphores that can potentially block τ_i.*
- *A task τ_i can be blocked for at most the duration of one critical section for each semaphore that can potentially block τ_i.*

Using these properties, it is possible to give a sufficient condition for the schedulability of a set of n hard real-time periodic tasks. If tasks are ordered by non decreasing periods ($T_i < T_j \Rightarrow i < j$), the schedulability condition can be expressed as follows [But97, SRBS98]:

$$\forall i \quad 1 \leq i \leq n \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq U_{lub}(A) \tag{5.1}$$

where B_i is the worst-case blocking time of task τ_i and $U_{lub}(A)$ is the least upper bound of the scheduling algorithm A used in the system.

If the PIP is applied on the example of Figure 5.1, the priority inversion phenomenon is reduced. The resulting schedule is illustrated in Figure 5.2. When task τ_H is blocked by τ_L, the latter *inherits* the priority of τ_H. Thus, when task τ_M is activated, it cannot preempt τ_L. When τ_L releases the lock, it returns to its original priority and it is preempted by τ_H.

When tasks are allowed to use nested critical sections, many complex blocking situations can occur. In particular, a task can inherit a new priority every time it blocks a task on a resource (*multiple inheritance*), and can be blocked by many tasks on different resources (*chained blocking*). Moreover, if critical sections can be nested arbitrarily, a *deadlock* may occur, as shown in Figure 5.3. In this example, both τ_1 and τ_2 access two resources R_1 and R_2 in a nested fashion. In particular, τ_1 accesses them in the order $P(R_2)$, ..., $P(R_1)$, ..., $V(R_1)$, ..., $V(R_2)$, whereas τ_2 accesses them in the order $P(R_1)$, ..., $P(R_2)$, ..., $V(R_2)$, ..., $V(R_1)$. If tasks arrive as shown in Figure 5.3, a deadlock occurs when τ_2 performs the $P(R_2)$ operation. Unfortunately, the PIP cannot prevent this to happen.

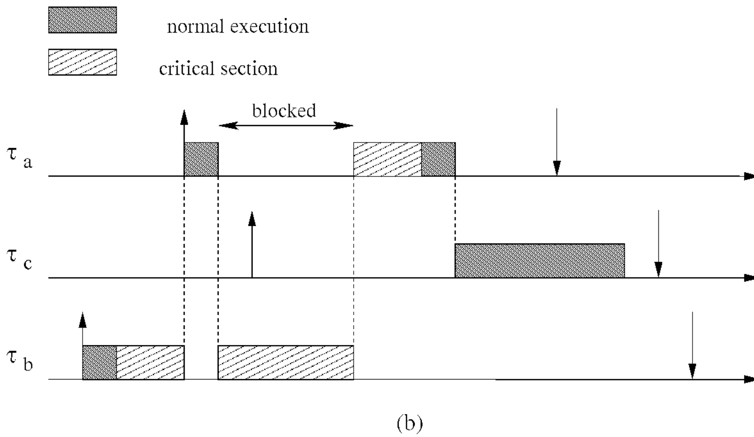


Figure 5.2 Schedule under the Priority Inheritance Protocol.

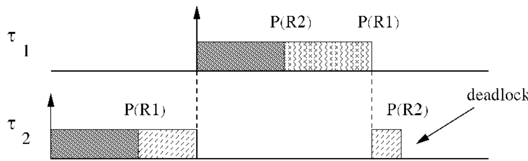


Figure 5.3 An example of deadlock situation.

5.3.2 PRIORITY CEILING PROTOCOL

The Priority Ceiling Protocol (PCP) [SRL90] improves the PIP by using additional a-priori information: each resource R is assigned a *ceiling* $C(R)$ equal to the maximum priority among all tasks that may lock R. The ceiling $C(R)$ is a static parameter that must be computed before runtime.

The PCP has the following rules:

- At time t, the executing task τ_i can lock resource R if its priority p_i is strictly higher than all the ceilings of the resources currently locked by other jobs. Let C^* be such a ceiling, and let R^* be the corresponding resource. If $p_i \leq C^*$, task τ_i is said to be blocked on R^* by the task is holding R^*.

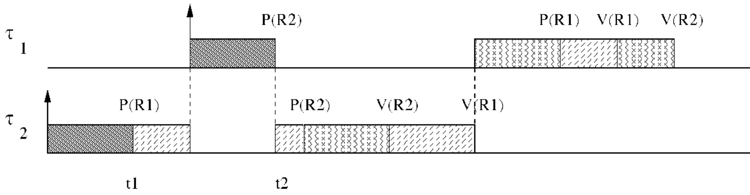


Figure 5.4 An example of schedule generated by the PCP.

- If τ_i is blocked by a lower priority task τ_L on resource R, τ_L inherits the priority of τ_i.
- When τ_L unlocks the resource and no other task is blocked by τ_L, it returns to its nominal priority. In general, a task τ_i always executes with a priority equal to the maximum between its nominal priority and the maximum priority of the tasks it is blocking.

We now revisit the example of Figure 5.3 by using the PCP. The resulting schedule is shown in Figure 5.4. Since both resources R_1 and R_2 are used by τ_1 and τ_2, their ceiling is equal to the priority of τ_1: $C(R_1) = C(R_2) = p_1$. At time t_1, τ_2 locks R_1, therefore $R^* = R_1$. At time t_2, τ_1 tries to lock R_2, but it is blocked because its priority is not strictly higher than $C(R_1)$. Hence τ_2 inherits the priority of τ_1 to avoid unbounded priority inversion by other medium priority tasks. When τ_2 releases the lock on R_1, τ_1 is awaken, accessing its resources without further blocking.

Sha, Rajkumar and Lehoczky [SRL90] formally proved that:

- the PCP prevents chained blocking;
- the PCP prevents deadlocks;
- under the PCP, a task can be blocked for at most the duration of one critical section.

5.3.3 STACK RESOURCE POLICY

The Stack Resource Policy (SRP) was proposed by Baker [Bak90, Bak91] to simplify the implementation of synchronization protocols in hard real-time embedded systems. As the PCP, the SRP prevents deadlocks, chained blocking, and limits the blocking time of each task to the length of a single critical section. In addition, the SRP allows the user

to define multi-unit resources, can work with fixed and dynamic priority scheduling, and allows all tasks to share a single stack, thus saving significant memory space. Finally, its implementation is straightforward, because SRP semaphores do not require waiting queues.

According to this protocol, each task τ_i is assigned a (static or dynamic) priority p_i and a static preemption level π_i, such that the following essential property holds:

Property 5.1 *Task τ_i is not allowed to preempt task τ_j, unless $\pi_i > \pi_j$.*

Under EDF and Deadline Monotonic (DM) scheduling, Property 5.1 is verified if a task τ_i is assigned a preemption level inversely proportional to its relative deadline:

$$\pi_i = \frac{1}{D_i}.$$

In addition, every resource R_k is assigned a static[2] *ceiling* defined as

$$C(R_k) = \max_i \{\pi_i \mid \tau_i \text{ needs } R_k\}.$$

A dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{C(R_k) \mid R_k \text{ is currently busy}\} \cup \{0\}].$$

Then, the SRP scheduling rule states that *"a job is not allowed to preempt until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling"*. It can be proved that the maximum time a task can be delayed by such a preemption test is at most equal to the length of a critical section. Once a task is started, the SRP ensures that it will never block until completion; it can only be preempted by higher priority tasks.

Figure 5.5 illustrates the same example of Figure 5.3 when resources are accessed by the SRP. By comparing Figures 5.4 and 5.5, we note that the schedule generated by the SRP has one less context switch. In fact, the high priority task is blocked upon arrival and not when it accesses the resource.

At a first sight, the reader might think that such an anticipated blocking is too pessimistic. However, it can be shown that the PCP and the SRP provide the same bound

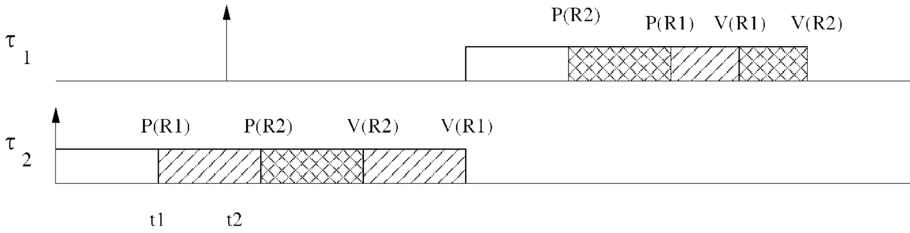


Figure 5.5 Schedule generated by the SRP.

on the blocking time of a task. In fact, the same task sets that are schedulable with the PCP, are also schedulable with the SRP.

Since a task never blocks once it starts executing, under the SRP there is no need to implement waiting queues. The blocking time B_i considered in the schedulability analysis refers to the time for which task τ_i is kept in the ready queue by the preemption test. Although a task never blocks on a locked resource, B_i is considered as a "blocking time" because it is caused by tasks having lower preemption levels.

Assuming relative deadlines equal to periods, the maximum blocking time for a task τ_i can be computed as the longest critical section among those belonging to tasks with preemption level less than π_i and with a ceiling greater than or equal to π_i:

$$B_i = \max j, h\{\gamma_{jh} \mid (\pi_j < \pi_i) \;\;\wedge\;\; \pi_i \leq C(R_h)\}. \tag{5.2}$$

where γ_{jh} is the length of the longest critical section of task τ_j on resource R_h.

Given these definitions, the feasibility of a set of periodic or sporadic tasks with resource constraints under EDF can be verified by the following sufficient condition:

$$\forall i, \;\; 1 \leq i \leq n \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \;\leq\; 1 \tag{5.3}$$

where we assumed that all the tasks are sorted by decreasing preemption levels, so that $\pi_i \geq \pi_j$ only if $i < j$.

The following theorem [LB00] extends the feasibility analysis by providing a tighter schedulability test.

[2] In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

Theorem 5.1 *Let $\mathcal{T}^{\mathcal{P}}$ be a set of n hard sporadic tasks ordered by decreasing preemption level (so that $\pi_i \geq \pi_j$ only if $i < j$), such that $U_p = \sum_{i=1}^{n} \frac{C_n}{T_n} \leq 1$. Then, $\mathcal{T}^{\mathcal{P}}$ is schedulable by EDF+SRP if and only if:*

$$\forall j,\ 1 \leq j \leq n \quad \forall L,\ 0 < L \leq T_j \qquad L \geq \sum_{i=1}^{j} \left\lfloor \frac{L}{T_i} \right\rfloor C_i + \max(0, B_j - 1). \quad (5.4)$$

It is worth noting that condition (5.4) is necessary and sufficient only for sporadic tasks, under the assumption that all tasks experience the maximum blocking time. This means that the SRP is optimal for sporadic tasks; that is, every feasible sporadic task set can be scheduled by EDF with the SRP. For periodic tasks the problem of deciding feasibility in the presence of resource constraints has been shown to be NP-hard [Jef92]. The complexity of the test is pseudo-polynomial; hence, it can be too costly for providing on-line guarantee in large task sets.

5.4 SHARED RESOURCES IN SOFT REAL-TIME SYSTEMS

The protocols described in the previous section were designed for hard real-time systems to limit priority inversion in the presence of resource constraints. When considering other task models (like aperiodic non-real-time tasks, soft real-time tasks and firm tasks) the problem of using shared memory protected by mutex semaphores becomes more complex.

A method for analyzing the schedulability of hybrid task sets where hard tasks may share resources with soft tasks handled by dynamic aperiodic servers was first presented by Ghazalie and Baker [GB95]. Their approach is based on reserving an extra budget to the aperiodic server for synchronization purpose and using the utilization-based test [LL73] for verifying the feasibility of the schedule. Lipari and Buttazzo [GB00] extended the analysis to a Total Bandwidth Server (TBS), using the Processor Demand Criterion [BRH90]; Buttazzo and Caccamo [BC99] used a similar approach to extend the analysis to the optimal Total Bandwidth (TB*) server.

In this chapter we describe possible solutions to the problem of bounding the blocking time of real-time tasks in hybrid systems consisting of hard and soft real-time tasks that can share resources. We first present an extension of the Constant Bandwidth Server (working with the SRP) that allows aperiodic tasks to share resources with hard

real-time periodic tasks. Then, we consider a more general model of open system, i.e., a system that has no a-priori knowledge of the task behavior, and we show how it is possible to provide real-time guarantees by using an algorithm that combines the Constant Bandwidth Server with the Priority Inheritance Protocol.

5.5 EXTENDING RESOURCE RESERVATION WITH THE SRP

The solution presented in this section, proposed by Caccamo and Sha [CS01], extends the Constant Bandwidth Server (CBS) algorithm [Abe98] to work under resource constraints and keep the key properties of the Stack Resource Policy (SRP) [Bak91] for resource sharing.

One of the problems in the integration of the CBS with the SRP is that the SRP protocol was developed under the assumption that preemption levels are fixed, and relative deadlines do not change. Unfortunately, under the CBS, server relative deadlines can be postponed, thus the resulting preemption level is dynamic.

Another problem is to avoid that an aperiodic task suspends its execution inside a critical section because the budget is exhausted. In fact, this would cause the blocked task to increase its blocking time, waiting until the budget is replenished. To avoid such an extra delay, we should allow the aperiodic task to continue until it leaves the critical section. Such an additional execution time is a kind of overrun, whose effects need to be taken into account in the feasibility test.

Two approaches can be pursued to deal with this problem. A first solution is to reserve extra budget for each CBS server for synchronization purposes and permit execution overruns. A second approach does not reserve extra synchronization budget, but prevents an aperiodic task from exhausting its budget inside a critical section. This section investigates the latter approach to improve the efficiency of the CBS budget management.

5.5.1 PREVENTING BUDGET EXHAUSTION INSIDE CRITICAL SECTIONS

To prevent long blocking delays due to the budget replenishment rule, a job exhausting its budget inside a critical section should be allowed to continue executing with the same deadline, using extra budget, until it leaves the critical section. At this time, the budget can be replenished at its full value and the deadline postponed.

In these situations, the maximum budget overrun occurs when the server exhausts its budget immediately after the job entered its longest critical section. Thus, if ξ is the duration of the longest critical section of task τ handled by server S, the bandwidth demanded by the server becomes $\frac{Q_s+\xi}{P_s}$. This approach clearly inflates the server utilization.

Alternatively, a job can perform a budget check before entering a critical section. If the current budget q_s is not sufficient to complete the job's critical section, the budget is replenished and the server deadline postponed. The remaining part of the job follows the same procedure until the job completes.

This approach dynamically partitions a job into *chunks*. Each chunk has execution time such that the consumed bandwidth is always less than or equal to the available server bandwidth U_s. By construction, a chunk has the property that it will never suspend inside a critical section. The following example illustrates two different solutions with the CBS+SRP, both using static preemption levels.

Example 5.1 The task set consists of an aperiodic job J_1, handled by a CBS with $Q_s = 4$ and $P_s = 10$, and two periodic tasks, τ_1 and τ_2, sharing two resources R_a and R_b. In particular, J_1 and τ_2 share resource R_b, whereas τ_1 and τ_2 share resource R_a. The task set parameters are reported in Table 5.1.

task	*type*	$Q_s or C$	$P_s or T$	R_a	R_b
J_1	soft aperiodic	4	10	-	3
τ_1	hard periodic	2	12	2	-
τ_2	hard periodic	6	24	4	2

Table 5.1 Parameters of the task set.

The first solution presented on this task set maintains a fixed relative deadline whenever the budget is replenished and the deadline postponed. The advantage of a fixed relative deadline is to keep the SRP policy unchanged for handling resource sharing between soft and hard tasks. According to this solution, when at time t the current budget is not sufficient for completing a critical section, a new deadline is computed as $d_s^{new} = t + P_s$ and the budget is recharged at the value $q_s = q_s + (d_s^{new} - d_s^{old})U_s$, where d_s^{old} is the previous server deadline.

A possible schedule of the task set produced by CBS+SRP is shown in Figure 5.6. Notice that the ceiling of resource R_a is $C(R_a) = 1/12$, and the ceiling of R_b is

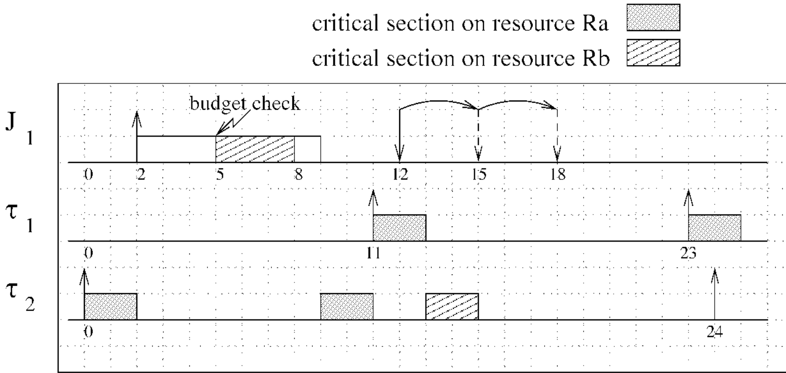


Figure 5.6 CBS+SRP with static preemption levels.

$C(R_b) = 1/10$. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + P_s = 12$, according to the CBS algorithm. At this time, τ_2 is already inside a critical section on resource R_a, however $H_{1,1}$ of job J_1 is able to preempt, since its preemption level is $\pi_1 = 1/10 > \Pi_s$. At time $t = 5$, J_1 tries to access resource R_b, but its residual budget is equal to 1 and is not sufficient to complete the whole critical section. As a consequence, a new chunk $H_{1,2}$ is generated with an arrival time $a_{1,2} = 5$ and a deadline $d_{1,2} = a_{1,2} + P_s = 15$ (the relative deadline is fixed). The budget is replenished according to the available server bandwidth; hence, $q_s = q_s + (d_s^{new} - d_s^{old})U_s = 1 + 1.2$. Unfortunately, the current budget is not sufficient to complete the critical section and an extra budget equal to 0.8 is needed. Hence, we have to inflate the budget, wasting bandwidth. The remaining part of the job follows the same procedure until the job completes. This approach has two main drawbacks: an extra budget still needs to be reserved, and jobs are cut in too many chunks, so increasing the algorithm overhead.

The second solution suspends a job whenever its budget is exhausted, until the current server deadline. Only at that time, the job will become eligible again, and a new chunk will be ready to execute with the budget recharged at its maximum value ($q_s = Q_s$) and the deadline postponed by a server period.

The schedule produced using this approach is shown in Figure 5.7. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + P_s = 12$, according to the CBS algorithm. At time $t = 5$, J_1 tries to access resource R_b, but its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, J_1 is temporarily suspended and a new chunk is released at time $t = 12$,

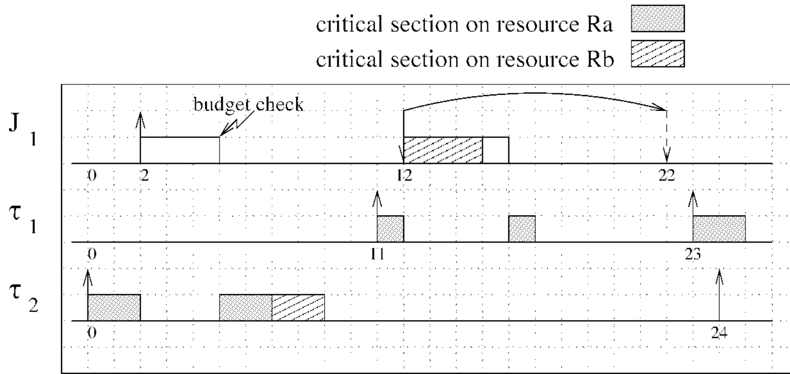


Figure 5.7 CBS+SRP with static preemption levels and job suspension.

with deadline $d_{1,2} = 22$, and the budget replenished ($q_s = Q_s = 4$). This approach has also two main drawbacks: it increases the response time of aperiodic tasks and, whenever the budget is recharged, the residual budget amount (if any) is wasted due to job suspension.

5.5.2 DYNAMIC PREEMPTION LEVELS

The two methods described above show that, although the budget check prevents budget exhaustion inside a critical section without inflating the server size, fixed relative deadlines and static preemption levels do not allow implementing an efficient solution to the addressed problem.

In this section we show that using dynamic preemption levels for aperiodic tasks allows achieving a simpler and elegant solution to the problem of sharing resources under CBS+SRP. According to the new method, whenever there is a replenishment, the server budget is always recharged by Q_s and the server deadline postponed by P_s. It follows that the server is always eligible, but each aperiodic task gets a dynamic relative deadline.

To maintain the main properties of the SRP, preemption levels are kept inversely proportional to relative deadlines, but are defined at a chunk level. The preemption level $\pi_{i,j}$ of a job chunk $H_{i,j}$ is defined as $\pi_{i,j} = 1/(d_{i,j} - a_{i,j})$. Notice that $\pi_{i,j}$ is assigned at run time and cannot be computed off line. As a consequence, a job J_i is characterized by a *dynamic preemption level* π_i^d equal to the preemption level of the current chunk.

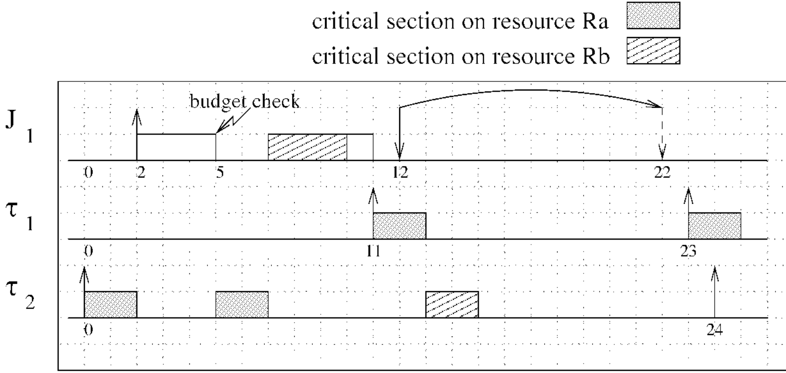


Figure 5.8 CBS+SRP with dynamic preemption levels.

To perform an off-line guarantee of the task set, it is necessary to know the *maximum preemption level* that can be assigned to each job J_i by the server. Therefore, the deadline assignment rule is modified to guarantee that each chunk has a minimum relative deadline D_i^{min} equal to its server period (the precise rules are reported in Section 5.5.4).

By setting $D_i^{min} = P_s$, each aperiodic task τ_i is characterized by a *maximum preemption level* π_i^{max} inversely proportional to the server period ($\pi_i^{max} = 1/P_s$). The maximum preemption levels are then used to compute the ceiling of every resource off line. Note that $\pi_i^d \leq \pi_i^{max}$, in fact, by definition,

$$\forall i,j \ \ \pi_{i,j} = \frac{1}{d_{i,j} - a_{i,j}} = \pi_i^d \leq \frac{1}{D_i^{min}} = \frac{1}{P_s} = \pi_i^{max}. \tag{5.5}$$

The schedule produced by CBS+SRP under dynamic preemption levels is shown in Figure 5.8. When job J_1 arrives at time $t = 2$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + P_s = 12$ according to the CBS algorithm. At this time, τ_2 is inside a critical section on resource R_a, but $H_{1,1}$ of job J_1 is able to preempt, since $\pi_{1,1} = 1/10 > \Pi_s$. At time $t = 5$, J_1 tries to access resource R_b, but its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, the deadline is postponed by P_s and the budget replenished by Q_s: $q_s = q_s + Q_s = 5$. Hence, the next chunk $H_{1,2}$ of J_1 starts at time $a_{1,2} = 5$ with deadline $d_{1,2} = d_{1,1} + P_s = 22$ and budget $q_s = 5$. However, chunk $H_{1,2}$ cannot start because its preemption level $\pi_{1,2} = 1/17 < \Pi_s$. It follows that τ_2 executes until

the end of its critical section. When the system ceiling becomes zero, J_1 is able to preempt τ_2. Note that the bandwidth consumed by any chunk is no greater than U_s, since whenever the budget is refilled by Q_s, the absolute deadline is postponed by P_s.

The main advantage of the proposed approach is that it does not require to reserve extra budget for synchronization purposes and does not waste the residual budget (if any) left by the previous chunk. However, we need to determine the effects that dynamic preemption levels have on the properties of the SRP protocol.

We first note that, since each chunk is scheduled by a fixed deadline assigned by the CBS, each chunk inherits the SRP properties. In particular, each chunk can be blocked for at most the duration of one critical section by the preemption test and, once started, it will never be blocked for resource contention. However, since a soft aperiodic job may consist of many chunks, it can be blocked more than once. The behavior of hard tasks remains unchanged, permitting resource sharing between hard and soft tasks without jeopardizing the hard tasks' guarantee. The details of the proposed technique are described in the next section.

5.5.3 CBS WITH RESOURCES CONSTRAINTS

In this section we first define the rules governing the CBS server with resource constraints, CBS-R, that have been informally introduced in the previous section. We then prove its properties. Under the CBS-R, each job J_i starts executing with the server current budget q_s and the server current deadline d_s. Whenever a chunk $H_{i,j}$ exhausts its budget at time $\bar{t}$, that chunk is terminated and a new chunk $H_{i,j+1}$ is released at time $a_{i,j+1} = \bar{t}$ with an absolute deadline $d_{i,j+1} = d_{i,j} + P_s$ (where P_s is the period of the server). When the job chunk $H_{i,j}$ attempts to lock a semaphore, the CBS-R server checks whether there is sufficient budget to complete the critical section. If not, a replenishment occurs and the execution performed by the job is labeled as chunk $H_{i,j+1}$, which is assigned a new deadline $d_{i,j+1} = d_{i,j} + P_s$. This procedure continues until the last chunk completes the job.

To comply with the SRP rules, a chunk $H_{i,j}$ starts its execution only if its priority is the highest among the active tasks and its *preemption level* $\pi_{i,j} = 1/(d_{i,j} - a_{i,j})$ is greater than the system ceiling. In order for the SRP protocol to be correct, every resource R_i is assigned a static[3] ceiling $C(R_i)$ (we assume binary semaphores) equal to the highest maximum preemption level of the tasks that could be blocked on R_i when the resource is busy. Hence, $C(R_i)$ can be computed as follows:

[3] In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the number of units actually free.

$$C(R_i) = \max_k \{\pi_k^{max} \mid \tau_k \;\; needs \;\; R_i\}. \tag{5.6}$$

It is easy to see that the ceiling of a resource computed by equation (5.6) is greater than or equal to the one computed using the dynamic preemption level of each task. In fact, as shown by equation (5.5), the maximum preemption level of each aperiodic task represents an upper bound of its dynamic value.

Finally, in computing the blocking time for a periodic/aperiodic task, we need to take into account the duration of the critical section of an aperiodic task without considering its relative deadline. In fact, the actual relative deadline of a chunk belonging to an aperiodic task is assigned on-line and it is not known in advance.

To simplify our formulation, we assume that each hard periodic task is handled by a dedicated CBS-R server with $Q_{s_i} \geq C_i$ and $T_{s_i} = T_i$. With such a parameters assignment, hard tasks do not really need a server in order to be scheduled; we prefer to use a server also for hard tasks, because this approach gives us the possibility to implement an efficient reclaiming mechanism on the top of the proposed approach. A reclaiming algorithm, like CASH [CBS00], is able to exploit spare capacities and can easily be integrated in this environment.

The blocking times can be computed as a function of the minimum relative deadline of each aperiodic task, as follows:

$$B_i = \max\{s_{j,h} \mid (T_{s_i} < T_{s_j}) \wedge \pi_i^{max} \leq C(\rho_{j,h})\}, \tag{5.7}$$

where $s_{j,h}$ is the worst-case execution time of the h-th critical section of task τ_j, $\rho_{j,h}$ is the resource accessed by the critical section $s_{j,h}$, and T_{s_i} is the period of the dedicated server. The B_i parameter computed by equation (5.7) is the blocking time experienced by a hard or soft task. In fact, $T_{s_i} = D_i^{min}$ for a soft aperiodic task and $T_{s_i} = D_i$ for a hard periodic task.

The correctness of our approach will be formally proved in Section 5.5.6. We will show that the modifications introduced in the CBS and SRP algorithms do not change any property of the SRP and permit to keep a static ceiling for the resources even though the relative deadline of each chunk is dynamically assigned at run time by the CBS-R server.

As shown in the examples illustrated above, some additional constraints have to be introduced to deal with shared resources. In particular, the correctness of the proposed technique relies on the following rules:

- Each job chunk must have a minimum relative deadline known a priori.
- A task must never exhaust its budget when it is inside a critical section.

In the following section we formally define the CBS-R algorithm which integrates the previous rules.

5.5.4 DEFINITION OF THE CBS-R

The CBS-R can be defined as follows:

1. A CBS-R is characterized by a budget q_s and by an ordered pair (Q_s, P_s), where Q_s is the maximum budget and P_s is the period of the server. The ratio $U_s = Q_s/P_s$ is denoted as the server bandwidth. At each instant, a fixed deadline d_s is associated with the server. At the beginning $d_s = 0$.
2. Each served job chunk $H_{i,j}$ is assigned a dynamic deadline $d_{i,j}$ equal to the current server deadline d_s.
3. Whenever a served job executes, the budget q_s is decreased by the same amount.
4. When $q_s = 0$, the server budget is recharged at the maximum value Q_s and a new server deadline is generated as $d_s = d_s + P_s$. Notice that there are no finite intervals of time in which the budget is equal to zero.
5. A CBS-R is said to be active at time t if there are pending jobs (remember the budget q_s is always greater than 0); that is, if there exists a served job J_i such that $r_i \leq t < f_i$. A CBS-R is said to be idle at time t if it is not active.
6. When a job J_i arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline (e.g., FIFO).
7. When a job J_i arrives and the server is idle, if $q_s \geq (d_s - r_i)U_s$ the server generates a new deadline $d_s = r_i + P_s$ and q_s is recharged at the maximum value Q_s, otherwise the server generates a new deadline $d_s = max(r_i + P_s, d_s)$ and the budget becomes $q_s = q_s + (d_s^{new} - d_s^{old})U_s$.
8. When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.
9. At any instant, a job is assigned the last deadline generated by the server.

10. Whenever a served job J_i tries to access a critical section, if $q_s < \xi_i$ (where ξ_i is the duration of the longest critical section of job J_i such that $\xi_i < Q_s$), a budget replenishment occurs, that is $q_s = q_s + Q_s$ and a new server deadline is generated as $d_s = d_s + P_s$.

It is worth noting that, with respect to the original definition given in [Abe98], we modified rule (7) and introduced rule (10). Rule (7) has been modified in order to guarantee that each job chunk has a minimum relative deadline equal to the server period. In fact, whenever a job J_i arrives and the server is idle, the job gets an absolute deadline grater than or equal to the arrival time plus the server period. The budget is recharged in such a way that the consumed bandwidth is always no greater than the reserved bandwidth $U_s = Q_s/P_s$.

Rule (10) has been added to prevent a task from exhausting its budget when it is using a shared resource. This is done by performing a budget check before entering a critical section. If the current budget is not sufficient to complete a critical section, the budget is replenished and the deadline postponed.

These two minor changes allow the CBS server to become compliant with the proposed approach without modifying its global behavior.

5.5.5 AN EXAMPLE

The following example illustrates the usage of the CBS-R server in the presence of resource constraints. The task set consists of an aperiodic job J_1, handled by a CBS-R with maximum budget $Q_s = 4$ and server period $P_s = 8$ and two periodic tasks τ_1, τ_2, which share two resources R_a and R_b; in particular, J_1 and τ_1 share resource R_b, while τ_1 and τ_2 share resource R_a. The task set parameters are shown in Table 5.2.

task	*type*	$Q_s or C$	$P_s or T$	R_a	R_b
J_1	soft aperiodic	4	8	-	3
τ_1	hard periodic	2	10	1	1
τ_2	hard periodic	6	24	3	-

Table 5.2 Parameters of the task set.

The schedule produced by CBS-R+SRP is shown in Figure 5.9. When job J_1 arrives at time $t = 3$, its first chunk $H_{1,1}$ receives a deadline $d_{1,1} = a_{1,1} + P_s = 11$ according to

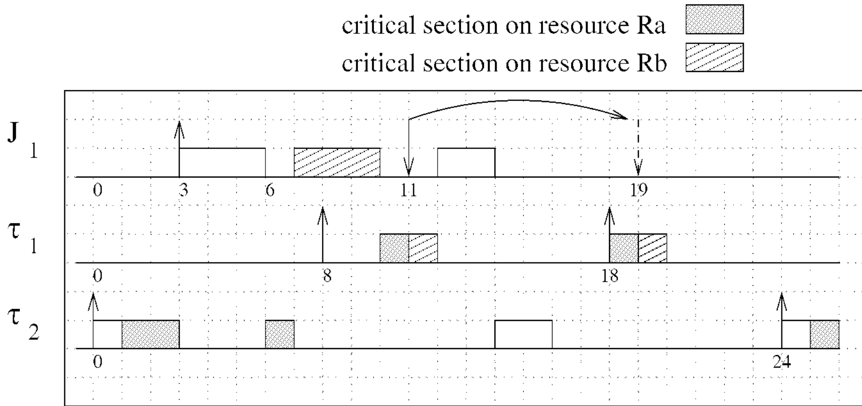


Figure 5.9 Schedule produced by CBS-R+SRP.

the CBS-R algorithm. At that time, τ_2 is already inside a critical section on resource R_a, however $H_{1,1}$ of job J_1 is able to preempt, having a preemption level $\pi_{1,1} = 1/8 > \Pi_s$. At time $t = 6$, J_1 tries to access resource R_b, but its residual budget is equal to one and is not sufficient to complete the whole critical section. As a consequence, the deadline is postponed and the budget replenished. Hence, the next chunk $H_{1,2}$ of J_1 starts at time $a_{1,2} = 6$ with deadline $d_{1,2} = 19$. The chunk $H_{1,2}$ of J_1 cannot start because its preemption level $\pi_{1,2} = 1/13 < \Pi_s$. It follows that τ_2 executes until the end of its critical section. When the system ceiling becomes zero, J_1 is able to preempt τ_2. When J_1 frees resource R_b, τ_1 starts executing. It is worth noting that each chunk can be blocked for at most the duration of one critical section by the preemption test and, once it is started, it will never be blocked for resource contention.

In the next section, the SRP properties are formally proved and the validity of the guarantee test is analyzed.

5.5.6 PROPERTIES OF THE CBS-R ALGORITHM

In this section we prove the properties of the CBS-R algorithm. In particular, we show that all the SRP properties are preserved for hard periodic tasks and for each chunk of the soft aperiodic tasks. Finally, we provide a sufficient guarantee test for verifying the schedulability of hybrid task sets consisting of hard and soft tasks.

Since a preemption level is always inversely proportional to the relative deadline of each chunk, the following properties can be derived in a straightforward fashion:

Property 5.2 A chunk $H_{i,h}$ is not allowed to preempt a chunk $H_{j,k}$, unless $\pi_{i,h} > \pi_{j,k}$.

Property 5.3 If the preemption level of a chunk $H_{i,j}$ is greater than the current system ceiling, then there are sufficient resources available to meet the requirement of $H_{i,j}$ and the requirement of every chunk that can preempt $H_{i,j}$.

Property 5.4 If no chunk $H_{i,j}$ is permitted to start until $\pi_{i,j} > \Pi_s$, then no chunk can be blocked after it starts.

Property 5.5 Under the CBS-R+SRP policy, a chunk $H_{i,j}$ can be blocked for at most the duration of one critical section.

Property 5.6 The CBS-R+SRP prevents deadlocks.

The proofs of properties listed above are similar to those in the original Baker's paper [Bak91]. The following lemma shows how hard periodic tasks maintain their behavior unchanged:

Lemma 5.1 *Under the CBS-R+SRP policy, each job of hard periodic task can be blocked at most once.*

Proof.
The schedule of hard periodic tasks produced by EDF is the same as the one produced by handling each hard periodic task by a dedicated CBS-R server with a maximum budget equal to the task WCET and server period equal to the task period; it follows that each hard task can never be cut into multiple chunks. Hence, using property (5.5), it follows that each instance of a hard periodic task can be blocked for at most the duration of one critical section. □

The following theorem provides a simple sufficient condition to guarantee the feasibility of hard tasks when they share resources with soft tasks under the CBS-R+SRP algorithm.

Theorem 5.2 *Let Γ be a task set composed by n hard periodic tasks and m soft aperiodic tasks, each one (soft and hard) scheduled by a dedicated CBS-R server. Supposing tasks are ordered by decreasing maximum preemption level (so that $\pi_i^{max} \geq \pi_j^{max}$ only if $i < j$), then the hard tasks are schedulable by CBS-R+SRP if*

$$\forall i, \;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i} \frac{Q_{s_j}}{T_{s_j}} + \frac{B_i}{T_{s_i}} \;\leq\; 1, \tag{5.8}$$

where Q_{s_j} is the maximum budget of the dedicated CBS-R server and T_{s_j} is the server period.

Proof.
Suppose equation (5.8) is satisfied for each τ_i. We have to analyze two cases:

Case A. Task τ_i has a relative deadline $D_i = T_{s_i}$. Using Baker's guarantee test (see Equation (5.3)), it follows that the task set Γ is schedulable if

$$\forall i, \;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1} \frac{Q_{s_j}}{D_j} + \frac{Q_{s_i}}{T_{s_i}} + \frac{B_i^{new}}{T_{s_i}} \;\leq\; 1,$$

where D_j is the relative deadline of task τ_j and B_i^{new} is the blocking time τ_i might experience when each τ_j has a relative deadline equal to D_j. Notice that a task τ_j can block as well as preempt τ_i varying its relative deadline D_j; however, τ_j cannot block and preempt τ_i simultaneously. In fact, if current instance of τ_j preempts τ_i, its absolute deadline must be before τ_i deadline; hence, the same instance of τ_j cannot also block τ_i, otherwise it should have its deadline after τ_i deadline. From considerations above, the worst-case scenario happens when τ_j makes preemption on τ_i, that is $D_j = T_{s_j}$. Hence, it follows that:

$$\forall i, \;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1} \frac{Q_{s_j}}{D_j} + \frac{Q_{s_i}}{T_{s_i}} + \frac{B_i^{new}}{T_{s_i}} \;\leq$$

$$\leq \sum_{j=1}^{i-1} \frac{Q_{s_j}}{T_{s_j}} + \frac{Q_{s_i}}{T_{s_i}} + \frac{B_i}{T_{s_i}} \;\leq\; 1.$$

Case B. Task τ_i has a relative deadline $D_i > T_{s_i}$. As in *Case A*, the task set Γ is schedulable if

$$\forall i, \;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1} \frac{Q_{s_j}}{D_j} + \frac{Q_{s_i}}{D_i} + \frac{B_i^{new}}{D_i} \;\leq\; 1.$$

From the considerations above, it follows that the worst-case scenario also occurs when $(\forall j, \;\; D_j = T_{s_j})$, hence

$$\forall i, \;\; 1 \leq i \leq n+m \quad \sum_{j=1}^{i-1} \frac{Q_{s_j}}{D_j} + \frac{Q_{s_i}}{D_i} + \frac{B_i^{new}}{D_i} \;\leq$$

$$\leq \sum_{j=1}^{i-1} \frac{Q_{s_j}}{T_{s_j}} + \frac{Q_{s_i}}{D_i} + \frac{B_i^{new}}{D_i}.$$

Notice that tasks are ordered by decreasing maximum preemption level and each task τ_j has the relative deadline set as $D_j = T_{s_j}$, except task τ_i whose relative deadline is $D_i > T_{s_i}$. Hence, from Equation (5.2) we derive that the new blocking time B_i^{new} of task τ_i is a function of the relative deadline D_i as follows:

$$B_i^{new} = \begin{cases} B_i & T_{s_i} \leq D_i < T_{s_{i+1}} \\ B_{i+1} & T_{s_{i+1}} \leq D_i < T_{s_{i-2}} \\ \vdots & \vdots \\ B_{n+m-1} & T_{s_{n-m-1}} \leq D_i < T_{s_{n+m}} \\ B_{n+m} = 0 & T_{s_{n-m}} \leq D_i \end{cases} \tag{5.9}$$

It is worth noting that the terms $B_i, B_{i+1}, ..., B_{n+m}$ are the blocking times computed by equation (5.7) and are experienced by hard or soft tasks if the relative deadline of each task is set equal to the period of its dedicated server. Finally, a $k \geq i$ will exist such that:

$$T_{s_k} \leq \;\; D_i \;\; < \;\; T_{s_{k+1}} \quad \Rightarrow \quad B_i^{new} = B_k,$$

so, it follows that:

$$\sum_{j=1}^{i-1} \frac{Q_{s_j}}{T_{s_j}} + \frac{Q_{s_i}}{D_i} + \frac{B_i^{new}}{D_i} = \sum_{j=1}^{i-1} \frac{Q_{s_j}}{T_{s_j}} + \frac{Q_{s_i}}{D_i} + \frac{B_k}{D_i} \leq$$

$$\leq \sum_{j=1}^{i} \frac{Q_{s_j}}{T_{s_j}} + \frac{B_k}{T_{s_k}} \leq \sum_{j=1}^{k} \frac{Q_{s_j}}{T_{s_j}} + \frac{B_k}{T_{s_k}} \leq 1.$$

The above inequality holds because k must be greater than or equal to i. Hence, it follows that the task set is schedulable. □

5.6 RESOURCE CONSTRAINTS IN DYNAMIC SYSTEMS

In *dynamic real-time systems* tasks can be activated dynamically and the system has no *a priori* knowledge about their run-time behavior. Resource reservation techniques described in Chapter 3 are appropriate for such dynamic systems, because they provide temporal isolation and guarantee to hard real-time tasks.

To provide temporal guarantees in dynamic real-time systems, the idea is to separate the "admission test" phase from the actual scheduling phase. When a task is first activated in the system and requires a guaranteed execution, it must go through an admission test. If the test is passed, the task is admitted into the system with a guaranteed execution profile. However, if the task tries to actually execute more than initially requested, the task is "slowed down" by the temporal isolation mechanism to prevent extra interference with the other tasks in the system. The main difference with a traditional real-time system is that a dynamic real-time system has no a-priori knowledge about the tasks that will be activated.

Such a lack of knowledge becomes very restrictive when tasks share resources with a synchronization protocol. Using the SRP, the admission test can be performed by Equations (5.3) or (5.4), which require the computation of task blocking times by Equation (5.2), which in turns requires the knowledge of the *resource ceilings*. To compute the resource ceilings the system must know in advance all the resources used by the tasks during execution. Such an information is not always readily available. Suppose for example that the code of a task is linked together with a shared library where shared data are protected by mutexes. The programmer of the task may not be aware of such a hidden implementation detail.

The problem becomes even more difficult if resource reservation techniques are introduced in general purpose operating systems. In fact, when dealing with hard real-time

systems, the effort of analyzing the task's code and the relations among tasks is necessary to guarantee the correctness of the entire application. In a general purpose operating system, however, it is not reasonable to ask the developer of a soft real-time application to specify the list of all the mutexes and the duration of each critical section during the application initialization, when the admission test is performed. As a matter of fact, the developer of a soft real-time task (like an MPEG player) might use a software component developed by another company, for which the source code is not available and that might use mutexes in its implementation.

Summarizing, the effectiveness of protocols like the SRP or the PCP is based on the a priori knowledge on tasks' behavior. However, especially when dealing with soft real-time systems, the behavior of a soft real-time task cannot always be completely characterized. As a consequence, global concepts like the *system ceiling* or the *resource ceilings* cannot be used.

In the remaining of this chapter, we will show how the problem of priority inversion in a dynamic real-time system can be solved without using global a priori knowledge about the tasks. The basic idea is to use the PIP, instead of PCP and SRP, as the PIP does not require a priori knowledge about the tasks.

Requirements. In the following, we present a novel protocol and its schedulability analysis for hard real-time tasks. The main objective is to simplify the admission test as much as possible, by only requiring information about the budget and the period of the server associated with each task. Our goal is also to find a scheduling algorithm that provides temporal isolation *without making any assumption on the temporal behavior of the tasks*. Then, if we are able to exactly characterize a priori the resource requirements of a task, this approach can be used to compute the server's budget and period that guarantee the task's deadlines. However, if our analysis is not correct, the temporal isolation property guarantees that the other tasks in the system will not be affected.

The scheduling algorithm we are looking for must fulfill the following requirements:

- Jobs arrival times (the $a_{i,j}$'s) are not known *a priori*, but are only revealed on line during system execution. Hence, the scheduling strategy cannot require any knowledge of future arrival times (e.g., cannot require tasks to be periodic).
- The exact execution requirements $c_{i,j}$ are also unknown, and can only be determined by actually executing $J_{i,j}$ to completion. Hence, the scheduling algorithm cannot require an *a priori* upper bound (a "worst-case execution time") on the value of $c_{i,j}$.
- The scheduling algorithm has no *a priori* knowledge of which resources a task will access; it can only be known on line when the task tries to lock a resource.

Hence, the scheduling algorithm cannot require any *a priori* upper bound on the worst-case execution time $\xi_{i,j}$ of a critical section.

The following information is needed only for performing a schedulability analysis on a hard task τ_i:

- the worst-case computation time C_i;
- the period T_i;
- the type (hard or soft) of every task that (directly or indirectly) interacts with τ_i (see Section 5.6.3 for a definition of interaction);
- for each interacting task τ_j, and for each shared resource R_k, the worst-case execution time $\xi_j(R_k)$ of the longest critical section of τ_j on R_k.

5.6.1 USING THE PRIORITY INHERITANCE PROTOCOL WITH THE CBS

When applying the PIP to the CBS, it is not clear how to account for blocking times. One possible way would be to consider the blocking time using the following admission test:

$$\forall i \quad 1 \leq i \leq n \quad \sum_{j=1}^{i} \frac{Q_j}{P_j} + \frac{B_i}{P_i} \leq 1 \tag{5.10}$$

where B_i represents the maximum blocking time experienced by each server.

However, this solution is not suitable for a dynamic system. In fact, in order to compute the maximum blocking time of each server, when a task is created we should "declare" the worst-case execution time of the critical sections on each accessed resource. This is in contrast with the goal of a scheduler that must be independent of the actual requirements of the tasks. In addition, if a soft task holds a critical section for longer than declared, *any server* could miss its deadline.

Example 5.2 To highlight this problem, consider the example shown in Figure 5.10. In this example, there are three servers $S_1 = (2, 6)$, $S_2 = (2, 6)$ and $S_3 = (6, 18)$. Server S_1 is assigned task τ_1, which accesses a resource R for the entire duration of its jobs (i.e., 2 units of time). Server S_2 is assigned task τ_2, which does not use any resource. Server S_3 is assigned task τ_3, which has an execution time of 6 units of time

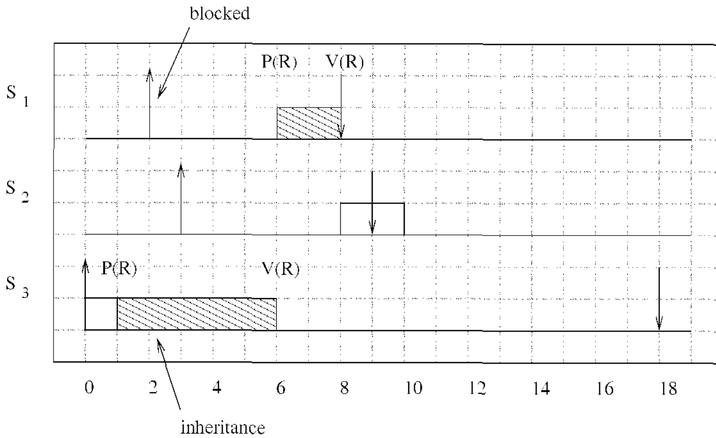


Figure 5.10 In the example, blocking times are not correctly accounted for.

and accesses resource R for 5 units of time. Now, suppose that τ_3 is a soft task that claims to use resource R for only 2 units of time. The system computes a maximum blocking time $B_1 = B_2 = 2$ for servers S_1 and S_2. According to Equation (5.10), the system is schedulable, and all servers can be admitted.

In the configuration of arrival times shown in Figure 5.10, server S_1 arrives at time t_2 and tries to access R. Since it is locked, server S_3 inherits a deadline $\delta'_3 = 8$ and continues executing. If no enforcement is put on the worst-case execution time of the critical section of task τ_3 on resource R, server S_2 misses its deadline. The simple fact that τ_3 executes more than expected inside the critical section invalidates the PIP and task τ_2, which does not use any resource, misses its deadline.

Another problem that must be considered is the depletion of the server budget while the task is in a critical section and has inherited the deadline of another server. In the original CBS formulation, the server deadline is postponed and the server budget is immediately recharged. When the PIP is applied it is not clear which deadline has to be postponed.

To solve the problems mentioned above, we combined the PIP and the CBS in a single algorithm called BandWidth Inheritance (BWI). The basic idea is that when a task executing inside a low-priority server blocks a high-priority server, it inherits the pair (q, δ) of the blocked server.

5.6.2 THE BANDWIDTH INHERITANCE PROTOCOL

Before starting with the description of the Bandwidth Inheritance protocol, we need to understand the meaning of *temporal isolation* when considering interacting tasks. In the original CBS formulation (see Section 3.6.1), tasks are assumed to be independent and hence do not interact in any way. When tasks access shared resources, they cannot be considered completely independent anymore. What does *isolation* mean in such a scenario?

Consider again the example shown in Figure 5.10. Server S_1 and server S_3 share a resource. It is easy to see that if S_3 holds the lock for longer than declared, some task will probably miss its deadline. Our goal is to prevent task τ_1 and τ_3 from interfering with τ_2. In fact, since τ_1 and τ_3 both access the same resource it is impossible to provide isolation among them.

5.6.3 BANDWIDTH ISOLATION IN THE PRESENCE OF SHARED RESOURCES

In this section, the concept of *interaction* among tasks is defined more precisely. Intuitively, a task τ_i can be affected by a task τ_j if it can be directly or indirectly blocked by τ_j. This relation is formalized by the following definition.

Definition 5.1 *A sequence* $BC_i = (\tau_1, R_1, \tau_2, R_2, \ldots, R_{z-1}, \tau_z)$, *with* $z \geq 2$, *is a* blocking chain on task τ_i *if:*

- $\tau_i = \tau_1$*;*
- $\forall k = 1, \ldots, z-1$, τ_k *and* τ_{k+1} *both use* R_k*; and*
- *if* $z > 2$ $\quad \forall k = 2, \ldots, z-1$, τ_k *accesses* R_k *with a critical section that is nested inside a critical section on* R_{k-1}.

If $z = 2$, then τ_i and τ_z directly share a resource. If $z > 2$, then τ_i and τ_z interact through nested critical sections.

As an example, consider the blocking chain $BC_1 = (\tau_1, R_1, \tau_2, R_2, \tau_3)$:

- task τ_3 uses resource R_2;

- task τ_2 uses R_2 with a critical section that is nested inside the critical section on R_1; and
- task τ_1 uses R_1.

Notice that, in the above example, τ_1 can be blocked by τ_2 and by τ_3, but τ_3 cannot be blocked by τ_1. Hence, a blocking chain defines an antisymmetric relation $\models$ between τ_i and τ_z: $\tau_i \models \tau_z$ but not viceversa.

In general, there can be more than one chain between two tasks τ_i and τ_j, because they can directly or indirectly share more than one resource. Let us enumerate the chains starting from task τ_i in any order. Let BC_i^h be the h-th blocking chain on τ_i. Without loss of generality, in the remainder of the paper we will sometimes drop the superscript on the chain. Moreover, let $\Gamma(BC_i)$ be the set of tasks $\tau_2, \ldots, \tau_z$ in the sequence BC_i (τ_i excluded), and let $R(BC_i)$ be the set of resources $R_1, \ldots, R_{z-1}$ in the sequence BC_i.

Definition 5.2 *The set Γ_i of tasks that may interact with τ_i is defined as follows:*

$$\Gamma_i = \bigcup_h \Gamma(BC_i^h)$$

Set Γ_i comprises all tasks that may directly or indirectly block τ_i.

Given these definitions, we can state the goals of our scheduling strategy more precisely. Whether task τ_i meets its deadlines should depend only on the timing requirements of τ_i and on the worst-case execution time of the critical sections of the tasks in Γ_i. Therefore, in order to guarantee a hard task τ_i, it is only necessary to know the behavior of the tasks in Γ_i.

5.6.4 THE PROBLEM OF DEADLOCKS

If we allow nested critical sections, the problem of deadlocks must be taken into account. A deadlock can be avoided by means of static or dynamic policies. One possibility is to use a protocol, like the PCP or the SRP, that prevents a deadlock from occurring. However, as we pointed out before, we cannot use the PCP or the SRP because they require a priori information on the behavior of the tasks.

To solve the deadlock problem, we consider another static policy. We assume that resources are totally ordered, and each task respects the ordering in accessing nested

critical sections. Thus, if $i < j$, then task τ can access a resource R_j with a critical section that is nested inside another critical section on resource R_i. When such order is defined, the sequence of resources in any blocking chain is naturally ordered. For a deadlock to be possible, a blocking chain must exist in which there is a circular relationship like $BC = (\ldots, R_i, \ldots, R_j, \ldots R_i, \ldots)$. Therefore, if the resources are ordered *a priori*, a deadlock cannot occur.

If the total order is not respected when accessing nested critical sections, a deadlock can still occur. As we will see in the next section, our protocol is able to detect it during runtime, but the action to be taken depends on the kind of resources. In the remainder of the paper, we shall assume that resources are ordered.

5.6.5 DESCRIPTION OF THE BANDWIDTH INHERITANCE PROTOCOL

The BWI protocol allows tasks to be executed on more than one server. Every server S_i maintains a list of served tasks. During runtime, it can happen that a task τ_i is in the list of more than one server. Let $e(i,t)$ be the index of the earliest deadline server among all the servers that have τ_i in their list at time t. Initially, each server S_i has only its own task τ_i in the list, hence $\forall i \; e(i,0) = i$[4]. We call server S_i the *default server* for task τ_i.

As long as no task is blocked, the BWI protocol follows the same rules of the CBS algorithm (see Section 3.6.1). In addition, the BWI protocol introduces the following rules:

Rule 10: if task τ_i is blocked when accessing a resource R that is locked by task τ_j, then τ_j is added to the list of server $S_{e(i,t)}$. If, in turn, τ_j is currently blocked on some other resource, then the chain of blocked tasks is followed, and server $S_{e(i,t)}$ adds all the tasks in the chain to its list, until it finds a ready task[5]. In this way, each server can have more than one task to serve, but only one of these tasks is not blocked.

Rule 11: when task τ_j releases resource R, if there is any task blocked on R, then τ_j was executing inside a server $S_{e(j,t)} \neq S_j$. Server $S_{e(i,t)}$ must now discard τ_j from its own list and the first blocked task in the list is now unblocked, let it be

[4]Note that index i denotes the task's index when it is the argument of function $e()$ and the server's index when it is the value of $e()$

[5]If, by following the chain, the algorithm finds a task that is already in the list, a deadlock is detected and an exception is raised.

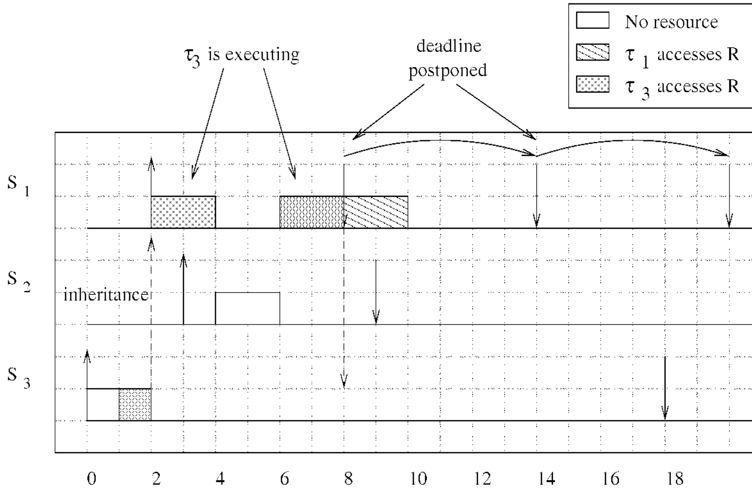


Figure 5.11 The BWI protocol is applied to the example of Figure 5.10.

τ_i. All the servers that added τ_j to their list while τ_j was holding R must discard τ_j and add τ_i.

The BWI is an inheritance protocol. When a high-priority task τ_i is blocked by a lower-priority task τ_j, τ_j *inherits* server $S_{e(i,t)}$ and the execution time of τ_j is then charged to $S_{e(i,t)}$. Therefore, $S_{e(j,t)} = S_{e(i,t)}$. When the budget of $S_{e(i,t)}$ is exhausted, $S_{e(i,t)}$'s deadline is postponed and τ_j can continue to execute on server $S_{e(j,t)}$ that may now be different from $S_{e(i,t)}$.

Example 5.3 The behavior of BWI is demonstrated by applying the algorithm to the example of Figure 5.10. The resulting schedule is depicted in Figure 5.11.

- At time $t = 1$, task τ_3, which is initially served by S_3, locks resource R.
- At time $t = 2$, server S_1 becomes the earliest deadline server and dispatches task τ_1, which immediately tries to lock resource R. However, the resource is already locked by τ_3. According to Rule 10, τ_3 is added to the list of S_1 and τ_1 is blocked. Now there are two tasks in S_1's list, but only τ_3 is ready. Hence, $S_{e(3,2)} = S_1$ and S_1 dispatches task τ_3.
- At time $t = 3$, server S_2 is activated but it is not the earliest deadline server. Thus τ_3 continues to execute.

- At time $t = 4$, the budget of server S_1 has been depleted. According to Rule 4 (see Section 3.6.1), the server deadline is postponed to $d_1^s \leftarrow d_1^s + P_1 = 14$ and the budget is recharged to $q_1 \leftarrow Q_1 = 2$. Since S_1 is no longer the earliest deadline server, S_2 is selected and task τ_2 is dispatched.
- At time $t = 6$, S_1 is again the earliest deadline server; hence task τ_3 is dispatched.
- At time $t = 8$, τ_3 releases the lock on R. According to Rule (11), τ_1 is unblocked, and τ_3 is discarded from the list of server S_1. Now S_1's list contains only task τ_1 and $S_{e(3,2)} = S_3$. Server S_1 is still the earliest deadline server but its budget has been depleted. According to Rule 4, $d_1^s \leftarrow d_1^s + P_1 = 20$ and $q_1 \leftarrow Q_1 = 2$.

Note that, task τ_2 is not influenced by the misbehavior of τ_3 and completes before its deadline.

5.6.6 PROPERTIES OF BWI

In this section, we extend the bandwidth isolation property and the hard schedulability property to the case of shared resources. Then we derive sufficient conditions for assigning server parameters that guarantee hard tasks. First, we present some preliminary results.

Lemma 5.2 *Each active server has always exactly one ready task in its list.*

Proof.
Initially, no task is blocked and the lemma is true. Suppose that the lemma holds just before time t_b, when task τ_i is blocked on resource R by task τ_j. After applying Rule 10, both servers S_j and S_i have task τ_j in their list, and task τ_i is blocked. By definition of $e(j, t_b)$, $S_{e(j,t_b)} = S_i$. Moreover, if τ_j is also blocked on another resource, the blocking chain is followed and all the blocked tasks are added to S_i until the first non-blocked task is reached. The lists of all the other servers remain unchanged, thus the lemma is true.

Now, suppose that the lemma is true just before time t_r. At this time, task τ_j releases the lock on resource R. If no other task was blocked on R, then the lists of all the servers remain unchanged. Otherwise, suppose that task τ_i was blocked on R and is now unblocked: server S_i has τ_j and τ_i in its list and, by applying Rule (11), discards τ_j. The lists of all the other servers remain unchanged, and the lemma holds. □

Theorem 5.3 *Consider a system consisting of n servers with*

$$\sum_{i=1}^{n} U_i \leq 1 \tag{5.11}$$

which uses the BWI protocol for accessing shared resources. Then, no server in the system misses its scheduling deadline[6].

Proof.
Lemma 5.2 implies that, at any time, the earliest deadline server has one and only one ready task in its list. As explained in [Lip00], from the viewpoint of the global scheduler, the resulting schedule can be regarded as a sequence of real-time jobs whose deadlines are equal to the deadlines of the servers (also referred as *chunks* in [Abe98] and [AB98a]). As the earliest deadline server never blocks, the computation times and the deadlines of the chunks generated by the server do not depend on the presence of shared resources. In [Abe98, Lip00], it was proved that in every interval of time the bandwidth *demanded* by the chunks produced by server S_i never exceeds $\frac{Q_i}{P_i}$, regardless of the behavior of the served tasks. Since Lemma 5.2 states that each active server always has one non blocked task in its list, the previous result is also valid for BWI. Hence, from the optimality of EDF and from $\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1$, it follows that none of these chunks misses its deadline. □

Note that the previous theorem states that no *scheduling deadline* will be missed, but it does not say anything about task's deadlines. To perform a guarantee on hard deadlines, we need a way to relate the tasks' deadlines to the server deadlines, and hence to the server parameters.

Definition 5.3 *Given a task τ_i, served by a server S_i with the BWI protocol, the* interference time *I_i is defined as the maximum time that all other tasks can execute inside server S_i for each job of τ_i.*

Theorem 5.4 *If hard task τ_i is served by a server S_i with the BWI protocol, with parameters $Q_i = C_i + I_i$ and $P_i = T_i$, where C_i is the WCET, T_i is the minimum interarrival time and I_i is the maximum interference time for S_i, then task τ_i will meet all its deadlines, regardless of the behavior of the other non-interacting tasks in the system.*

[6] We recall that the scheduling deadline is the deadline used by the server to schedule a served job with EDF.

Proof.
According to Theorem 3.1, the CBS algorithm guarantees that each server S_i receives up to Q_i units of execution every P_i units of time. Hence, if each instance of τ_i consumes less than Q_i and instances are separated by P_i or more, server S_i never postpones its scheduling deadline. From Theorem 3.1, $f_{i,j} \leq \delta_i$.

Theorem 5.3 extends the result of Theorem 3.1 to BWI. However, when considering the BWI protocol, other tasks can execute inside server S_i, consuming its budget (and hence postponing the deadline of server S_i even if $C_i \leq Q_i$). In order to ensure that server S_i will never postpone its scheduling deadline, we have to consider the interference time from those tasks. If I_i is the maximum time that other tasks can execute inside S_i, it follows that τ_i can execute for $Q_i - I_i$ units of time before exhausting the server budget. Hence, the theorem follows. □

Considerations. When our system consists only of hard tasks, BWI is not the best protocol to use. In fact, substituting Q_i and P_i into Equation (5.11), we obtain:

$$\sum_{i=1}^{n} \frac{C_i + I_i}{T_i} \leq 1,$$

which may result in a lower utilization than Equation (5.1) because all the interference times are summed together. Hence, if we are dealing with a hard real-time system, it is better to use other scheduling strategies like the PCP [SRL90] or the SRP [Bak90], for example by using a strategy like the one described in Section 5.5.

The BWI protocol is more suitable for dynamic real-time systems, where hard, soft and non real-time tasks can coexist and it is impossible to perform an off-line analysis for the entire system. Of course, this comes at the cost of a lower utilization for hard tasks.

5.6.7 COMPUTING THE INTERFERENCE TIME

In the general case, the exact computation of the interference I_i is a complex problem. In this section, we restrict our attention to its estimation for the hard tasks. At a first glance, the problem may seem similar to the problem of computing the blocking times B_i for the PIP. However, computing the interference is much more difficult, because it depends on the soft tasks: their unpredictable execution may cause the associated servers to exhaust their budgets and postpone their deadlines.

In many cases, it is desirable to guarantee a hard task τ_i even if it interacts with soft tasks. In fact, sometimes it is possible to know indirectly the worst-case execution

time of the critical sections of a soft task. For example, consider a hard task and a soft task that access the same resource by using common library functions. If the critical sections are implemented as library functions with bounded execution time, then we can still determine the amount of time a soft task can steal from the server's budget of a hard task. Indeed, this is a very common case in a real operating system.

Therefore, we will now consider the problem of computing I_i for a server S_i that is the default server of a hard task. We start by providing an important definition that simplifies the discussion.

Definition 5.4 *Let S_i be a server that never postpones its deadline (i.e., S_i's budget is never exhausted while there is a job that has not yet finished). We call S_i an* HRT server. *If the server deadline can be postponed (i.e., a time t exists in which $q_i = 0$ and the served job has not yet finished), we call S_i an* SRT server.

The distinction between HRT and SRT servers depends only on the kind of tasks they serve. Both HRT and SRT servers follow the same rules and have the same characteristics. However, it may be impossible to know the WCET of a soft task, so the corresponding default SRT server can *decrease its priority* while executing. The presence of SRT servers that interact with HRT servers complicates the computation of the interference.

The following examples show how one or more soft tasks can contribute (directly or indirectly) to the interference of a hard task.

Example 5.4 Consider a hard task τ_i, served by server S_i and a soft task τ_j, served by a server S_j with period $P_j < P_i$. We do not know the WCET of task τ_j. Therefore, we assign the budget of S_j according to some rule of thumb. Server S_j is an SRT server as it may postpone its deadline. If τ_j executes less than its server budget and the server deadline is not postponed, S_i cannot preempt S_j. If instead τ_j executes for more than its server budget, the server's deadline is postponed. The corresponding situation is shown in Figure 5.12a. S_j can be preempted by S_i while inside a critical section, and block τ_i, contributing to its interference I_i.

Example 5.5 Consider three tasks, τ_i, τ_j and τ_k, served by servers S_i, S_j and S_k, respectively, with $P_j < P_i < P_k$. Servers S_i and S_k are HRT servers, while S_j is an SRT server. All tasks access resource R. Task τ_i accesses resource R twice with two different critical sections. One possible blocking situation is shown in Figure 5.12b. The first time, τ_i can be blocked by task τ_k on the first critical section. Then, it can

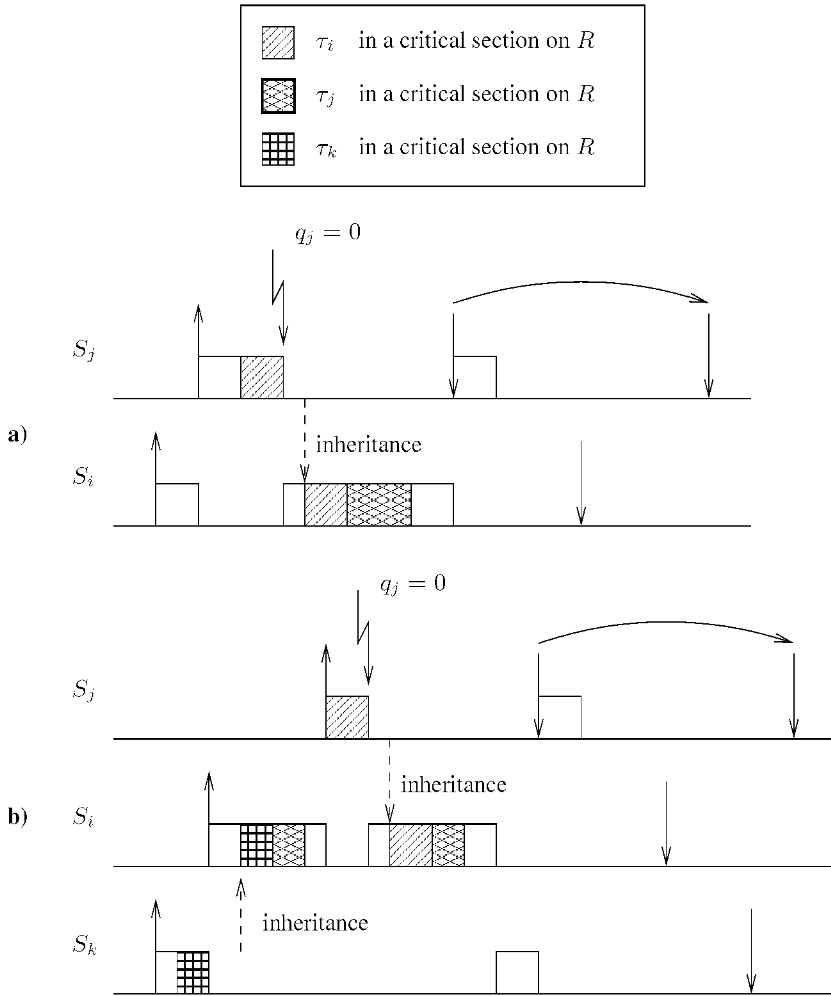


Figure 5.12 Example of blocking situations with soft tasks: a) A soft task with a short period blocks a hard task with a long period; b) a hard task is blocked twice on resource R.

be preempted by task τ_j which first locks R, and then, before releasing the resource, depletes the server budget and postpones its deadline. Thus, when τ_i executes, it can be blocked again on the second critical section on R. Note that both τ_j and τ_k belong to Γ_i.

Example 5.6 As a last example, we show one case in which, even if all tasks in Γ_i are hard tasks, it may happen that τ_j interferes with S_i with two different critical sections. Consider three tasks, τ_i, τ_j and τ_k. Task τ_i accesses only resource R_2 with two critical sections. Task τ_j accesses two resources R_1 and R_2 and R_2 is accessed twice with two critical sections both nested inside the critical section on R_1. Task τ_k accesses only R_1 with one critical section. The only blocking chain starting from task τ_i is $BC_i = (\tau_i, R_2, \tau_j)$. Hence $\Gamma_i = \{\tau_j\}$. Note that task τ_k cannot interfere with task τ_i.

Tasks τ_i, τ_j and τ_k are assigned servers S_i, S_j and S_k, respectively, with $P_k < P_i < P_j$. Tasks τ_i and τ_j are both hard tasks and we know their WCETs and periods. Task τ_k is a soft tasks and we do not know its WCET. Finally, we assume to know the duration of all critical sections (for example, because resources are accessed through shared libraries that we are able to analyze).

We assign budgets and periods of server S_i and S_j so that they are HRT servers (their interference is computed using the algorithm described in Figure 5.14, which will be presented later). The budget of server S_k is assigned according to some rule of thumb. Since we do not know whether S_k will exhaust its budget while executing, S_k is considered an SRT server.

One possible blocking situation is shown in Figure 5.13. Task τ_j locks resource R_1 and then resource R_2. At time t_1 it is preempted by task τ_i that tries to lock resource R_2 and it is blocked. As a consequence, task τ_j inherits server S_i and interferes with it for the duration of the first critical section on R_2. When τ_j releases R_2, it returns inside its server S_j and τ_i executes completing its critical section on R_2. Then, server S_k is activated and τ_k starts executing and tries to lock resource R_1. Since R_1 is still locked by τ_j, τ_k is blocked and τ_j inherits server S_k. While τ_j executes inside S_k, it locks resource R_2 again. Before releasing R_2, server S_k exhausts its budget and postpones its deadline. Now the earliest deadline server is S_i that continues to execute and tries to lock R_2 at time t_2. As a consequence, τ_j inherits S_i and interferes with it for the second time.

From the examples shown above, it is clear that there are many possible situations in which a task can interfere with a server. In the next section, we formally present a set of lemmas that identify the conditions under which a task can interfere with an HRT server.

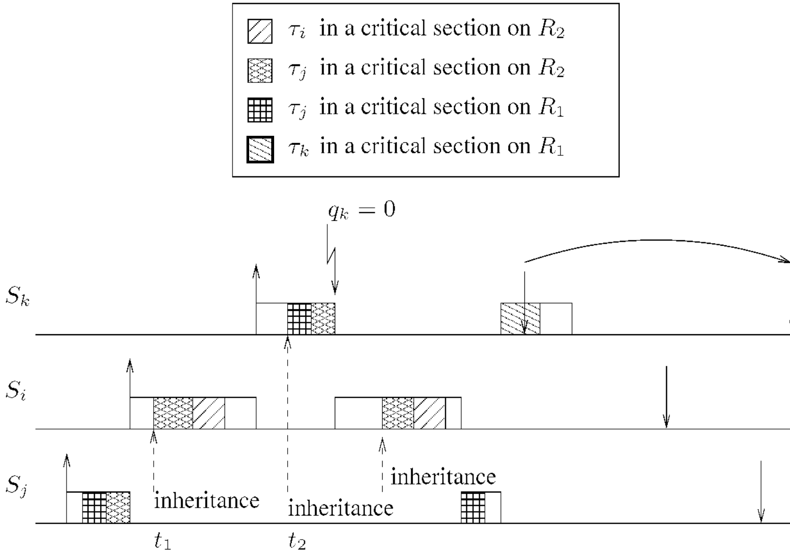


Figure 5.13 Example of blocking situation: task τ_j can interfere twice with S_i even if S_i and S_j are both HRT servers.

5.6.8 CONDITIONS FOR INTERFERENCE

We start by defining the set of servers that can be inherited by a task.

Definition 5.5 *Let Ψ_j be the set of all servers that can be "inherited" by task τ_j, S_j included:*

$$\Psi_j = \{S_i | \exists BC_i, \ \ \tau_j \in BC_i\} \cup \{S_j\}.$$

A task τ_j can only inherit tasks in Ψ_j, hence $\forall t \quad S_{e(j,t)} \in \Psi_j$.

Definition 5.6 *Let $\Psi_j^{SRT}(i)$ be the set of all SRT servers that can be "inherited" by task τ_j and interfere with server S_i:*

$$\Psi_j^{SRT}(i) = \{S_k | S_k \textit{is an SRT server} \wedge \exists BC_k = (\tau_k, \ldots, \tau_j, \ldots, \tau_i)\}.$$

If S_j is an SRT server, it is also included in $\Psi_j^{SRT}(i)$.

Consider Example 5.6. There is one chain from τ_k to τ_i: $BC_k = (\tau_k, R_1, \tau_j, R_2, \tau_i)$. Therefore, $S_k \in \Psi_j^{SRT}(i)$. Set $\Psi_j^{SRT}(i)$ is important in our analysis because it identifies the tasks that can inherit an SRT server before interfering with the server S_i

under analysis. In Example 5.6, task τ_j can inherit the SRT server S_k which may later postpone its deadline.

Now, we prove some important properties of the BWI protocol.

Lemma 5.3 *If server S_i is HRT, then $\forall t : d^s_{e(i,t)} \leq d^s_i$.*

Proof.
When τ_i inherits a server S_j, this server must have a scheduling deadline shorter than d^s_i. Recall that, by definition, $e(i, t)$ is the index of the server with the shortest scheduling deadline among all servers inherited by τ_i at time t. Hence $d^s_{e(i,t)} = d^s_j < d^s_i$. If S_j postpones its deadline before the time at which τ_i releases the resource, τ_i continues to execute inside the server with the shortest deadline among the inherited servers. Since S_i never postpones its deadline, the lemma follows. □

Lemma 5.4 *Given a task τ_i, only tasks in Γ_i can be added to server S_i and contribute to I_i.*

Proof.
It directly follows from Rule (10) and from the definition of Γ_i. □

Lemma 5.5 *Let S_i be an HRT server. Task τ_j with default server S_j cannot interfere with server S_i if:*

$$P_j \leq P_i \quad \wedge \quad \forall S_k \in \Psi_j^{SRT}(i) : P_k > P_i$$

Proof.
By contradiction. For τ_j to interfere with S_i it must happen that at a certain time t_1, τ_j locks a resource R; it is then preempted by server S_i at time t_2, which blocks on some resource; τ_j inherits S_i as a consequence of this blocking. Therefore, τ_j must start executing inside its default server before S_i arrives, and executes in a server $S_{e(j,t_2)}$ with deadline $d^s_{e(j,t_2)} > d^s_i$ when it is preempted. By hypothesis $P_j \leq P_i \Rightarrow d^s_j < d^s_i$.

Hence, τ_j inherits a server $S_{e(j,t_1)}$ with $d^s_{e(j,t_1)} > d^s_i$. However, from the hypothesis follows that server S_j never postpones its deadline ($S_j \notin \Psi^{SRT}_j(i)$), and from Lemma 5.3, $d^s_{e(j,t_1)} \leq d^s_j < d^s_i$. This is a contradiction, hence the lemma follows. □

The next definition precisely identifies the tasks that can interfere with server S_i.

Definition 5.7 *A* proper blocking chain BC_i *is a blocking chain that contains only tasks that can interfere with* S_i*:*

$$\forall \tau_j \in BC_i : P_j > P_i \quad \vee \quad \exists S_k \in \Psi^{SRT}_j(i) : P_k \leq P_i.$$

In some case, we have to consider multiple interferences from the same task and on the same resource. The following lemmas restrict the number of possible interference situations.

Lemma 5.6 *Let* S_i *be an HRT server and* τ_j *a task belonging to a proper blocking chain* BC_i*. If the following condition holds:*

$$P_j > P_i \quad \wedge \quad \forall S_k \in \Psi^{SRT}_j(i) : P_k \geq P_i$$

then τ_j *can interfere with server* S_i *for at most the worst-case execution of one critical section for each job.*

Proof.
Suppose τ_j interferes with S_i in two different intervals: the first time in interval $[t_1, t_2)$, the second time in interval $[t_3, t_4)$. Therefore, at time t_2, $d^s_{e(j,t_2)} > d^s_i$. If τ_j does not lock any resource in $[t_2, t_3)$, then at time t_3 server S_i blocks on some resource R that was locked by τ_j before t_1 and that it has not yet released. Therefore, τ_j interferes with S_i for the duration of the critical section on R, which includes the duration of the first critical section ($\xi(R) \geq (t_4 - t_3) + (t_2 - t_1)$) and the lemma follows.

Now suppose τ_j executes in interval $[t_2, t_3)$ and locks another resource R_1. It follows that it inherits a server S_k that preempts S_i with $d^s_k < d^s_i$. Hence $P_k < P_i$. From the hypothesis, S_k is an HRT server and d^s_k is not postponed before τ_j releases resource R_1. Hence, τ_j cannot inherit S_i while it is inside S_k, and we fall back in the previous case. □

Lemma 5.7 *Let S_i be an HRT server and R a resource. If the following condition holds:*

$$\forall BC_i^h, BC_i^h = (\ldots, R, \tau_j, \ldots), \quad \forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i$$

then at most one critical section on R can contribute to the interference I_i.

Proof.
The proof of this lemma is very similar to the proof of Lemma 5.6. By contradiction. Suppose two critical sections on the same resource R contribute to I_i. The first time, task τ_p inherits server S_i at time t_1 while it is holding the lock on R. The second time, task τ_j inherits server S_i at time $t_2 > t_1$ while it is holding the lock on R. It follows that:

- The lock on R was released between t_1 and t_2;
- τ_j arrives before t_2 and executes on a server $S_{e(j)}$ with $d^s_{e(j)} < d^s_i$;
- τ_j acquires the lock on R at $t_a < t_2$;
- just before t_2, $d^s_i < d^s_{e(j)}$.

Hence, at time t_a, τ_j is executing in an inherited server $S_k \in \Psi_j^{SRT}(i)$ that postpones its deadline before τ_j releases the lock on R. S_k must arrive after S_i with deadline $d^s_k < d^s_i$ and later postpone its deadline. This contradicts the hypothesis that $\forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i$. Hence, the lemma follows. □

The previous lemmas restrict the number of combinations that we must analyze when computing the interference time. In particular: Lemma 5.6 identifies the conditions under which a task can interfere with server S_i for at most one critical section; Lemma 5.7 identifies the conditions under which a certain resource can interfere with server S_i at most one time.

Now, the interference due to each blocking chain is quantified.

Lemma 5.8 *The worst-case interference for server S_i due to a proper blocking chain $BC_i = (\tau_1, R_1, \ldots, R_{z-1}, \tau_z)$ is*

$$\xi(BC_i) = \sum_{k=2}^{z} \xi_k(R_{k-1}). \tag{5.12}$$

Proof.
It simply follows from the definition of proper blocking chain. □

Given a proper blocking chain, we need to distinguish the tasks that can interfere with S_i for at most the duration of one critical section (i.e., that verify the hypothesis of Lemma 5.6), from the tasks that can interfere with S_i multiple times.

Definition 5.8 *Given a proper blocking chain BC_i^h, let $\overline{\Gamma}(BC_i^h)$ be the set of tasks in BC_i^h that verify the hypothesis of Lemma 5.6. Then,*

$$\overline{\Gamma}(BC_i^h) = \left\{\tau_j | \tau_j \in BC_i^h \wedge P_j > P_i \wedge (\forall S_k \in \Psi_j^{SRT}(i) : P_k \geq P_i)\right\}.$$

We do the same thing for the resources.

Definition 5.9 *Given a proper blocking chain BC_i^h, let $\overline{R}(BC_i^h)$ be the set of resources in BC_i^h that verify the hypothesis of Lemma 5.7. Then,*

$$\overline{R}(BC_i^h) = \left\{R_j | R_j \in BC_i^h \wedge P_{j+1} > P_i \wedge (\forall S_k \in \Psi_{j+1}^{SRT}(i) : P_k \geq P_i)\right\}.$$

5.6.9 ALGORITHM FOR COMPUTING THE INTERFERENCE

The pseudo-code of the algorithm for computing the interference is shown in Figure 5.14. The term cs_i denotes the number of critical sections for task τ_i, and $CS_i(k)$ denotes the set of proper blocking chains starting from the k-th critical. More formally, $CS_i(k)$ is the set of proper blocking chains of the form $BC_i^h = (\tau_i, R, \ldots)$ where R is the resource accessed in the k-th critical section of τ_i.

Function `interference(k, T, R)` is first called with $k = 1$, $\mathcal{T} = \Gamma_i$ and with $\mathcal{R}$ equal to the set of all possible resources. At line 5, we consider the case in which τ_i is not blocked on the k-th critical section. In this case, the function is recursively called for the (k+1)-th critical section.

Lines 6-12 consider the case in which τ_i is blocked on the k-th critical section. For each proper blocking chain BC_i in $CS_i(k)$, the algorithm checks if it is a legal blocking chain, that is, the resources in $\overline{R}(BC_i^k)$ and the tasks in $\overline{\Gamma}(BC_i^k)$ have not yet been considered in the interference time computation. If so, function `interference()` is

```
1:  int interference(int k, set 𝒯, set ℛ)
2:  {
3:    int ret = 0;
4:    if (k > cs_i) return 0;
5:    ret = interference(k+1, 𝒯, ℛ);
6:    foreach (BC_i ∈ CS_i(k)) {
7:      if(Γ̄(BC_i) ⊆ 𝒯 and R̄(BC_i) ⊆ ℛ) {
8:        𝒯' = 𝒯 \ Γ̄(BC_i);
9:        ℛ' = ℛ \ R̄(BC_i);
10:       ret = max(ret, ξ(BC_i) + interference(k+1, 𝒯', ℛ'));
11:     }
12:   }
13:   return ret;
14: }
```

Figure 5.14 Algorithm for computing the interference time for server S_i.

recursively called with $k' = k + 1$, $\mathcal{T}' = \mathcal{T} \setminus \overline{\Gamma}(BC_i)$, and $\mathcal{R}' = \mathcal{R} \setminus \overline{R}(BC_i)$ (lines 8-10). Otherwise, it selects another chain from $CS_i(k)$. The recursion stops when $k > cs_i$ (line 4).

The algorithm has exponential complexity, since it explores all possible interference situations for server S_i. We conjecture that the problem of finding the interference time in the general case is NP-Hard. However, the proof of this claim is left as a future work.

5.7 CONCLUDING REMARKS

In this chapter, the resource reservation mechanism has been extended to work with a more general model where tasks can interact through shared resources protected by mutexes. This problem is of paramount importance for the implementation of resource reservation techniques in real operating systems.

Two different approaches have been analyzed. In the first approach, the CBS algorithm has been extended to work with the SRP. In the second approach, the CBS algorithm has been extended to work with the PIP. The first approach is best suited in hard real-time systems that also include soft real-time aperiodic tasks. The second approach is best suited in dynamic real-time systems where there is no a priori knowledge about the tasks requirements.

6

RESOURCE RECLAIMING

In most real-time systems, predictability is achieved by enforcing timing constraints on application tasks, whose feasibility is guaranteed off line by means of proper schedulability tests based on worst-case execution time (WCET) estimations. Theoretically, such an approach works fine if all the tasks have a regular behavior and all WCETs are precisely estimated. In practical cases, however, a precise estimation of WCETs is very difficult to achieve, because several low level mechanisms present in modern computer architectures (such as interrupts, DMA, pipelining, caching, and prefetching) introduce a form of non deterministic behavior in tasks' execution, whose duration cannot be predicted in advance.

A general technique for guaranteeing temporal constraints of hard activities in the presence tasks with unpredictable execution is based on the resource reservation approach [MST94b, TDS$^+$95, AB98a] (see Chapter 3). Using this methodology, however, the overall system's performance becomes quite dependent on a correct resource allocation. It follows that wrong resource assignments will result in either wasting the available resources or lowering tasks responsiveness. Such a problem can be overcome by introducing suitable resource reclaiming techniques which are able to exploit early completions to satisfy the extra execution requirements of other tasks.

This chapter introduces some resource reclaiming algorithms that are able to guarantee isolation among tasks while relaxing the bandwidth constraints enforced by resource reservations.

6.1 PROBLEMS WITH RESERVATIONS

According to the resource reservation approach, each task is assigned (off line) a fraction of the available resources and is handled by a dedicated server, which prevents the served task from demanding more than the reserved amount whenever the task experiences an overrun. Although such a method is essential for achieving predictability in

the presence of tasks with variable execution times, the overall system's performance becomes quite dependent on the actual resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources. This problem can be overcome by introducing reclaiming mechanisms able to share the spare budget left in the servers among all the active tasks, so that any overloaded server can benefit by additional spare capacities. The following example highlights the problem described above and gives a flavor about possible solutions.

Example. Consider the case shown in Figure 6.1, where three tasks are handled by three servers with budgets $Q_1 = 1, Q_2 = 5, Q_3 = 3$, and periods $T_1 = 4, T_2 = 10, T_3 = 12$, respectively. At time $t = 6$, job $\tau_{2,1}$ completes earlier with respect to the allocated budget, whereas job $\tau_{3,1}$ requires one extra unit of time. Figure 6.1a illustrates the classical case in which no reclaiming is used and tasks are served by the plain *Constant Bandwidth Server* (CBS) [AB98a] algorithm. Notice that, in spite of the budget saved by $\tau_{2,1}$, the third server is forced to postpone its current deadline when its budget is exhausted (it happens at time $t = 9$). As shown in Figure 6.1b, however, we observe that the spare capacity saved by $\tau_{2,1}$ could be used by $\tau_{3,1}$ to advance its execution and prevent the server from postponing its deadline. The intuition is that early completions of tasks generate spare capacities that are wasted by traditional resource reservation approaches, unless resource reclaiming is adopted to relax the bandwidth constraints, still providing isolation among tasks.

In the next sections, we present two scheduling techniques, the *CApacity SHaring* (CASH) algorithm [CBS00] and the *Greedy Reclamation of Unused Bandwidth* (GRUB) algorithm [GB00], which are able to reclaim unused resources (in terms of CPU capacities) while guaranteeing isolation among tasks. Both techniques handle hybrid task sets consisting of hard periodic tasks and soft aperiodic tasks. Moreover, both algorithms rely on the following assumptions:

1. tasks are scheduled by a dynamic priority assignment, namely, the Earliest Deadline First (EDF) algorithm;

2. tasks are assumed to be independent, that is, they do not compete for gaining access to shared and mutual exclusive resources;

3. each task τ_i is handled by a dedicated server S_i, which is assigned a fraction of the processor bandwidth;

4. both CASH and GRUB build upon the *Constant Bandwidth Server*.

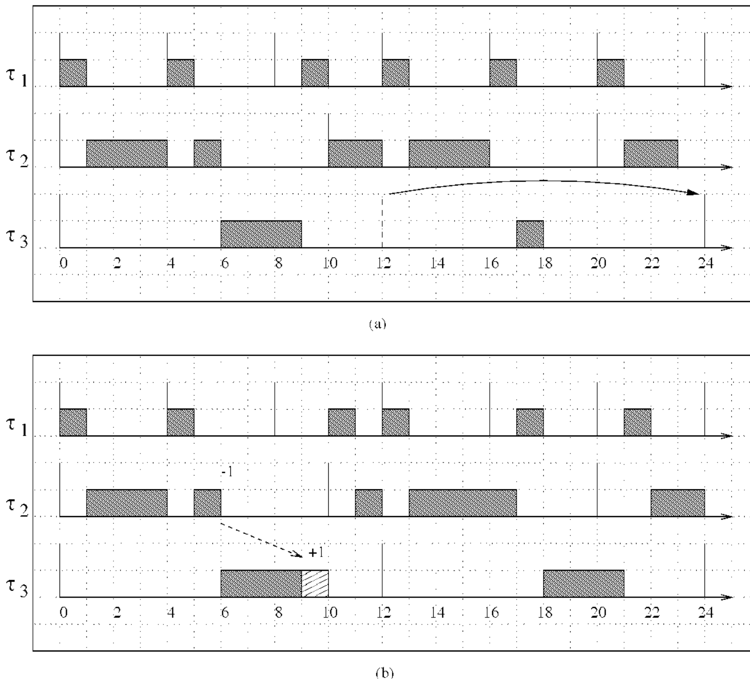


Figure 6.1 Overruns handled by a plain CBS (a) versus overruns handled by a CBS with a resource reclaiming mechanism (b).

6.2 THE CASH ALGORITHM

The *CApacity SHaring* (CASH) algorithm [CBS00] is a general scheduling methodology for managing overruns in a controlled fashion by reclaiming spare capacities on line. In particular, this technique allows to

- achieve isolation among tasks, through a resource reservation mechanism which bounds the effects of task overruns;
- perform efficient reclaiming of the unused computation times, through a global capacity sharing mechanism that allows exploiting early completions in order to relax the bandwidth constraints enforced by isolation;
- handle tasks with different criticality and flexible timing constraints, to enhance the performance of those real-time applications which allow a certain degree of flexibility.

The CASH mechanism works in conjunction with the *Constant Bandwidth Server* (CBS). Each task is handled by a dedicated CBS and the reclaiming mechanism uses a global queue, the CASH queue, of spare capacities ordered by deadline. Whenever a task completes its execution and its server budget is greater than zero, such a residual capacity is stored in the CASH queue along with its deadline and can be used by any active task to advance its execution. When using a spare capacity, the task can be scheduled using the corresponding server deadline associated with the spare capacity. In this way, each task can use its own capacity along with the residual capacities deriving by the other servers.

Whenever a new task instance is scheduled for execution, the server tries to use the residual capacities with deadlines less than or equal to the one assigned to the served instance; if these capacities are exhausted and the instance is not completed, the server starts using its own capacity. Every time a task ends its execution and the server becomes idle, the residual capacity (if any) is inserted with its deadline in the global queue of available capacities. Spare capacities are ordered by deadline and are consumed according to an EDF policy. The main benefit of the proposed reclaiming mechanism is to reduce the number of deadline shifts (typical of the CBS), so enhancing aperiodic tasks responsiveness. Notice that, due to the isolation mechanism introduced by the multiple server approach, there are no particular restrictions on the task model that can be handled by the CASH algorithm. Hence, tasks can be hard, soft, periodic, or aperiodic.

CASH RULES

The precise behavior of the CASH algorithm is defined by the following rules.

1. Each server S_i is characterized by a budget c_i and by an ordered pair (Q_i, T_i), where Q_i is the maximum budget and T_i is the period of the server. The ratio $U_i = Q_i/T_i$ is denoted as the server bandwidth. At each instant, a fixed deadline $d_{i,k}$ is associated with the server. At the beginning $\forall i,\ d_{i,0} = 0$.

2. Each task instance $\tau_{i,j}$ handled by server S_i is assigned a dynamic deadline equal to the current server deadline $d_{i,k}$.

3. A server S_i is said to be active at time t if there are pending instances. A server is said to be idle at time t if it is not active.

4. When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$ and c_i is recharged at the maximum value Q_i.

5. When a task instance $\tau_{i,j}$ arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline.

6. Whenever instance $\tau_{i,j}$ is scheduled for execution, the server S_i uses the capacity c_q in the CASH queue (if there is one) with the earliest deadline d_q, such that $d_q \leq d_{i,k}$, otherwise its own capacity c_i is used.

7. Whenever job $\tau_{i,j}$ executes, the used budget c_q or c_i is decreased by the same amount. When c_q becomes equal to zero, it is extracted from the CASH queue and the next capacity in the queue with deadline less than or equal to $d_{i,k}$ can be used.

8. When the server is active and c_i becomes equal to zero, the server budget is recharged at the maximum value Q_i and a new server deadline is generated as $d_{i,k} = d_{i,k-1} + T_i$.

9. When a task instance finishes, the next pending instance, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle, the residual capacity $c_i > 0$ (if any) is inserted in the CASH queue with deadline equal to the server deadline, and c_i is set equal to zero.

10. Whenever the processor becomes idle for an interval of time Δ, the capacity c_q (if exists) with the earliest deadline in the CASH queue is decreased by the same amount of time until the CASH queue becomes empty.

AN EXAMPLE

To better understand the proposed approach, we will describe a simple example which shows how the CASH reclaiming algorithm works. Consider a task set consisting of two periodic tasks, τ_1 and τ_2, with periods $P_1 = 4$ and $P_2 = 8$, maximum execution times $C_1^{max} = 4$ and $C_2^{max} = 3$, and average execution times $C_1^{avg} = 3$ and $C_2^{avg} = 2$. Each task is scheduled by a dedicated CBS having a period equal to the task period and a budget equal to the average execution time. Hence, a task completing before its average execution time saves some budget, whereas it experiences an overrun if it completes after. A possible execution of the task set is reported in Figure 6.2, which also shows the capacity of each server and the residual capacities generated by each task. At time $t = 2$, task τ_1 has an early completion and a residual capacity equal to one with deadline equal to 4 becomes available. After that, τ_2 consumes the above residual capacity before starting to use its own capacity; hence, at time $t = 4$, the overrun experienced by τ_2 is handled without postponing its deadline. Notice that each task tries to use the residual capacities before using its own capacity and that whenever an idle interval occurs (e.g., interval [19, 20]), the residual capacity with

the earliest deadline has to be discharged by the same amount in order to guarantee a correct behavior.

The example above shows that overruns can be handled efficiently without postponing any deadline. A classical CBS instead, would have postponed some deadlines in order to guarantee tasks isolation. Clearly, if all the tasks consume their allocated budget, no reclaiming can be done and this method performs as a plain CBS. However, this situation is very rare in practical situations, hence the CASH algorithm helps in improving the average system's performance.

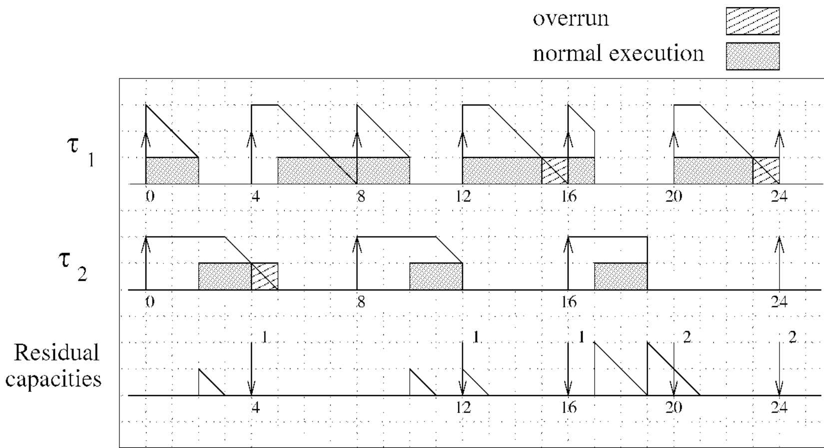


Figure 6.2 Example of schedule produced by CBS+CASH.

6.2.1 SCHEDULABILITY ANALYSIS

In this section we analyze the schedulability condition for a hybrid task set consisting of hard and soft periodic tasks. Each task is scheduled using a dedicated CBS. If each hard periodic task is scheduled by a server with a maximum budget equal to the task WCET and with a period equal to the task period, it behaves like a standard hard task scheduled by EDF. The difference is that each task can gain and use extra capacities and yields its residual capacity to other tasks. Such a run-time capacity exchange, however, does not affect schedulability; thus, the task set can be guaranteed using the classical Liu and Layland condition:

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1,$$

where Q_i is the maximum server budget and T_i is the server period. Before proving the schedulability condition, the following lemma will prove that all the generated capacities are exhausted before their respective deadlines.

Theorem 6.1 *Given a set Γ of capacity based servers along with the CASH algorithm, each capacity generated during the scheduling is exhausted before its deadline if and only if:*

$$\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1, \tag{6.1}$$

where Q_i is the maximum server budget and T_i is the server period.

Proof.
If. Assume equation (6.1) holds and suppose that a capacity c^* is not exhausted at time t^*, when the corresponding deadline is reached. Let $t_a \geq 0$ be the last time before t^* at which no capacity is discharging; that is, the last instant before t^* during which the CPU is idle and the CASH queue is empty (if there is no such time, set $t_a = 0$). Let $t_b \geq 0$ be the last time before t^* at which a capacity with deadline after t^* is discharging (if there is no such time, set $t_b = 0$). If we take $t = max(t_a, t_b)$, time t has the property that only capacities created after t and with deadline less than or equal to t^* are used during $[t, t^*]$. Let $Q_T(t_1, t_2)$ be the sum of capacities created after t_1 and with deadline less than or equal to t_2; since a capacity misses its deadline at time t^*, it must be that:

$$Q_T(t, t^*) > (t^* - t)$$

In the interval $[t, t^*]$, we can write that:

$$(t^* - t) < Q_T(t, t^*) \leq \sum_{i=1}^{n} \left\lfloor \frac{t^* - t}{T_i} \right\rfloor Q_i \leq (t^* - t) \sum_{i=1}^{n} \frac{Q_i}{T_i},$$

which is a contradiction.

Only if. Suppose that $\sum_i \frac{Q_i}{T_i} > 1$. Then, we show there exists an interval $[t_1, t_2]$ in which $Q_T(t_1, t_2) > (t_2 - t_1)$. Assume that all the servers are activated at time 0; then, for $L = lcm(T_1, ..., T_n)$ we can write that:

$$Q_T(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor Q_i = \sum_{i=1}^{n} \frac{L}{T_i} Q_i = L \sum_{i=1}^{n} \frac{Q_i}{T_i} > L,$$

hence, the "*only if* condition" follows. □

We now formally prove the schedulability condition with the following theorem:

Theorem 6.2 *Let $\mathcal{T}_h$ be a set of periodic hard tasks, where each task τ_i is scheduled by a dedicated server with $Q_i = C_i^{max}$ and $T_i = P_i$, and let $\mathcal{T}_s$ be a set of soft tasks scheduled by a group of servers with total utilization U^{soft}. Then, $\mathcal{T}_h$ is feasible if and only if*

$$\sum_{\tau_i \in \mathcal{T}_h} \frac{Q_i}{T_i} + U^{soft} \leq 1, \tag{6.2}$$

Proof.
The theorem directly follows from Lemma 6.1; in fact, we can notice that each hard task instance has available at least its own capacity equal to the task WCET. Lemma 6.1 states that each capacity is always discharged before its deadline, hence it follows that each hard task instance has to finish by its deadline. □

It is worth noting that Theorem 6.2 also holds under a generic capacity-based server having a periodic behavior and a bandwidth U_s.

6.3 THE GRUB ALGORITHM

The *Greedy Reclamation of Unused Bandwidth* (GRUB) algorithm [GB00] is a server-based global scheduling algorithm which is able to provide performance guarantee with the ability of *reclaiming* unused processor capacity ("bandwidth"). According to the GRUB algorithm, each task is executed under a dedicated server S_i, characterized by two parameters: a *processor share* U_i, and a *period* P_i.

When a set of tasks is scheduled by distinct servers to provide isolation in terms of consumed bandwidth, leftover CPU cycles can be originated by one or more servers which have no outstanding jobs waiting for execution. As a consequence, a resource reclaiming capability is desirable in order to achieve efficient exploitation of the available resources (i.e., the CPU). Notice that *the unused capacity reclamation is achieved by GRUB without any additional cost or complexity*. In fact, since the resource reclaiming feature is a direct consequence of the server scheduling rules (namely, the deadline assignment rule which affects the scheduling priorities under EDF), the computational complexity of GRUB algorithm is the same as that of previously-proposed schedulers (CBS). Moreover, while capacity reclamation does not directly affect the performance guarantee (since in the worst case there may be no idle servers and hence no excess capacity to reclaim), it tends to result in improved system performance (compared to capacity based servers like CBS) still enforcing isolation among tasks.

6.3.1 GRUB DYNAMIC VARIABLES

In the following, we provide a detailed description of the GRUB algorithm, which is the global scheduler. Assuming that each task τ_i is scheduled by a dedicated server S_i, GRUB maintains a global system variable (*system utilization* $U(t)$) in addition to three dynamic variables for each server S_i: a server *deadline* d_i, a server *virtual time* V_i, and a server *state*.

- Intuitively, the value of d_i at each instant is a measure of the *priority* that GRUB algorithm accords server S_i at that instant — GRUB will essentially be performing earliest deadline first (EDF) scheduling based upon these d_i values.
- The value of V_i at any time is a measure of how much of server S_i's "reserved" service has been consumed by that time. GRUB algorithm will attempt to update the value of V_i in such a manner that, *at each instant in time, server S_i has received the same amount of service that it would have received by time V_i if executing on a dedicated processor of capacity U_i*.
- At any instant in time during run-time, each server S_i is in one of three states: **inactive**, **activeContending**, or **activeNonContending**. Intuitively at time t_o a server is in the **activeContending** state if it has some jobs awaiting execution at that time; in the **activeNonContending** state if it has completed all jobs that arrived prior to t_o, but in doing so has "used up" its share of the processor until beyond t_o (i.e., its virtual time is greater than t_o); and in the **inactive** state if it has no jobs awaiting execution at time t_o, and it has *not* used up its processor share beyond t_o. Notice that a server is said to be *active* at time t if it is in either the **activeContending** or the **activeNonContending** state, and *inactive* otherwise.
- The GRUB algorithm maintains an additional variable, called the *system utilization $U(t)$*, which at each instant in time is equal to the sum of the capacities U_i of all servers S_i that are active at that instant in time. $U(t)$ is initially set equal to zero.

GRUB is responsible for updating the values of these variables, and will make use of these variables in order to determine which job to execute at each instant in time. At each time, GRUB chooses for execution some server that is in its **activeContending** state (if there are no such servers, then the processor is idled). From the servers that are in their **activeContending** state, GRUB algorithm chooses for execution the server with the earliest deadline.

While S_i is executing, its virtual time V_i increases; while S_i is not executing V_i does not change. If at any time this virtual time becomes equal to the server deadline

$(V_i == d_i)$, then the deadline parameter is incremented by P_i $(d_i \leftarrow d_i + P_i)$. Notice that this may cause S_i to no longer be the earliest-deadline active server, in which case it may surrender control of the processor to an earlier-deadline server.

6.3.2 GRUB RULES

After introducing the dynamic variables needed by GRUB, we now formally describe how the GRUB algorithm updates these variables, so that it can determine which job has to execute at each instant in time.

1. If server S_i is in the **inactive** state and a job J_i^j arrives (at time-instant a_i^j), then the following code is executed

$$V_i \leftarrow a_i^j$$

$$d_i \leftarrow V_i + P_i$$

   and server S_i enters the **activeContending** state.

2. When a job J_i^{j-1} of S_i completes (notice that S_i must then be in its **activeContending** state), the action taken depends upon whether the next job J_i^j of S_i has already arrived.

   (a) If so, then the deadline parameter d_i is updated as follows:

$$d_i \longleftarrow V_i + P_i \; ;$$

   and the server remains in the **activeContending** state.

   (b) If there is no job of S_i awaiting execution, then server S_i changes state and enters the **activeNonContending** state.

3. For server S_i to be in the **activeNonContending** state at any instant t, it is required that $V_i > t$. If this is not so, (either immediately upon transiting into this state, or because time has elapsed but V_i does not change for servers in the **activeNonContending** state), then the server enters the **inactive** state.

4. If a new job J_i^j arrives while server S_i is in the **activeNonContending** state, then the deadline parameter d_i is updated as follows:

$$d_i \longleftarrow V_i + P_i,$$

   and server S_i returns to the **activeContending** state.

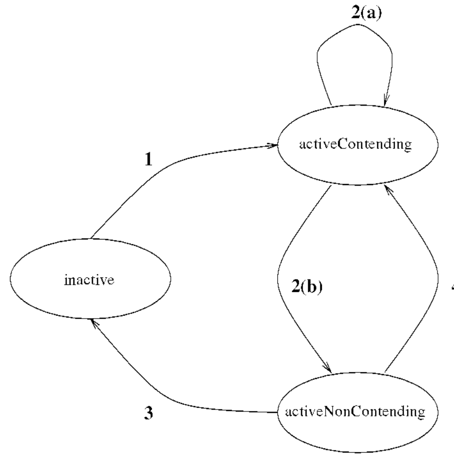


Figure 6.3 GRUB state transition diagram: node labels refer to server states and edges numbers to transition rules.

5. While a job of server S_i is executing, the server virtual time increases at a rate U/U_i:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases}$$

   If V_i becomes equal to d_i, then d_i is incremented by an amount P_i ($d_i \longleftarrow d_i + P_i$).

6. There is one additional possible state change: if the processor is ever idle, then *all* servers in the system return to their **inactive** state.

Figure 6.3 shows the state transition diagram according to the above rules.

Notice that the rate the virtual time is increasing at determines whether a server S_i reclaims unused capacity or not. In fact, suppose that U is equal to one (none of the servers is inactive and the system is fully utilized); intuitively, S_i will be allowed to execute for U_iP_i units within a server period P_i. Hence, since V_i is incremented at a rate $1/U_i$ while S_i is executing. In this case, GRUB is equivalent to the CBS algorithm and performance can be guaranteed as done for the CBS. However, if U becomes less than one, the resource reclaiming capability of GRUB is enabled and the current executing server S_i (V_i is incremented at a rate U/U_i) starts to reclaim unused bandwidth executing for $(U_iP_i)/U$ units within a server period P_i.

In using excess processor capacity, though, we must be very careful not to end up using any of the *future* capacity of currently inactive servers, since we do not know when the

currently inactive servers will become active. To this purpose, the slope of the virtual time $V(t)$ is dynamically updated as any server changes its current state.

As an example of the resource reclaiming capability of GRUB, just consider two servers S_1 and S_2, both having bandwidth utilization $U_1 = U_2 = 0.5$ and server period $P_1 = P_2 = 6$. Assume that S_1 has a pending request (job τ_1^1) at time $t = 0$, and consider two possible cases: 1) server S_2 is active; 2) server S_2 is inactive. In the first case (S_2 active), server S_1 will assign a deadline $d_1 = a_1^1 + P_1 = 0 + 6$ to job τ_1^1 and $\frac{d}{dt}V_1(t) = U/U_1 = 1/0.5 = 2$; it follows that τ_1^1 can execute for three units of time before postponing the server deadline by a server period (no reclaiming occurs and GRUB behaves like the CBS). On the other hand, if S_2 is inactive, server S_1 will assign the same deadline $d_1 = 6$ to job τ_1^1 as before, but the server virtual time will increase at a rate $\frac{d}{dt}V_1(t) = U/U_1 = 0.5/0.5 = 1$; it follows that τ_1^1 can execute for six units of time before postponing the server deadline by a server period. In the latter case, the reserved bandwidth of server S_2 is completely reclaimed by server S_1 fully utilizing the processor.

The behavior of the GRUB algorithm will be clarified by the following example.

EXAMPLE

Consider a system with four servers with the following parameters:

Server S_i	S_1	S_2	S_3	S_4
Bandwidth U_i	0.2	0.3	0.25	0.25
Period P_i	5	9	4	4

Let us assume that servers S_3 and S_4, which together have $(U_3 + U_4) = 0.5$ of the total processor capacity, are not active at all — this unused processor capacity could in fact have been allocated to servers S_1 and S_2. Figure 6.4 shows the server behavior when the following sequence of job arrivals occurs:

- Initially, all four servers are in the **inactive** state.
- Job τ_1^1 arrives at $a_1^1 = 0$; consequently, server S_1 changes state and enters the **activeContending** state, with V_1 set to 0 and d_1 to 5.
- Job τ_2^1 also arrives at $a_2^1 = 0$, thus server S_2 changes state and enters the **active-Contending** state, with V_2 set to 0 and d_2 to 9.

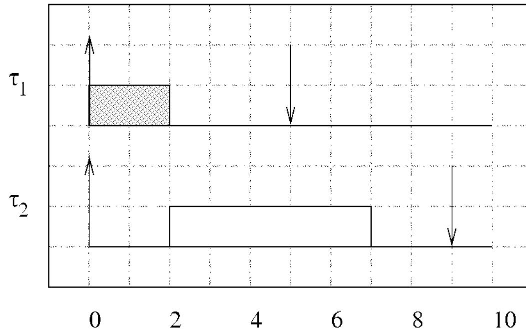


Figure 6.4 Schedule produced by the GRUB algorithm.

- U – the total capacity of all currently active servers – is equal to $(U_1 + U_2) = (0.2 + 0.3) = 0.5$.

- According to EDF, task τ_1^1 (served by S_1) is selected for execution, and V_1 is incremented at a rate $U/U_1 = 0.5/0.2 = 2.5$. At time 2, V_1 becomes equal to d_1; assuming that the computation time of τ_1^1 is $c_1^1 = 2$, τ_1^1 completes execution at this instant, and enters the **activeNonContending** state.

- Server S_2 now becomes the only **activeContending** server in the system, and consequently τ_2^1 is executed. V_2 is incremented at a rate $U/U_2 = 0.5/0.3$.

- At time 5, S_1 enters the inactive state. Now, U becomes equal to $U_2 = 0.3$, and $V_2(5)$ is equal to $(0.5/0.3) \times 3 = 5$. From now on, V_2 is incremented at a rate of $0.3/0.3 = 1$.

- Assuming that c_2^1 is equal to 5, τ_2^1 completes execution at instant 7 and enters the **activeNonContending** state – at this time, V_2 has increased to 7.

- Since $V_2(7) = 7$, S_2 returns to the **inactive** state at instant 7.

If the GRUB algorithm is substituted by four CBS servers with same bandwidth and server periods, job τ_1^1 would execute for one unit of time before exhausting its server budget. As a consequence, at time $t = 1$, server S_1 would postpone its deadline by a server period ($d_1 = d_1 + P_1 = 5 + 5 = 10$) releasing the CPU to τ_2^1. Similarly, job τ_2^1 would execute for 2.7 units of time before exhausting its server budget; hence, at time $t = 3.7$, server S_2 would postpone its deadline by a server period ($d_1 = d_1 + P_1 = 9 + 9 = 18$). Finally, both jobs would complete without postponing their server deadline again. The schedule in this case is depicted in Figure 6.5.

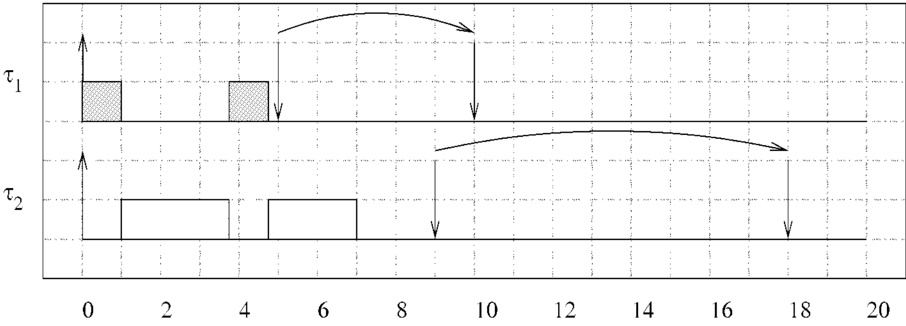


Figure 6.5 Schedule produced by the CBS server without bandwidth reclamation.

Comparing the two schedules generated in the example above, we immediately see one of the advantages of the GRUB algorithm over non-reclaiming servers (like CBS): since a reclaiming scheduler like GRUB is likely to execute a job for a longer interval than a non-reclaiming scheduler, we would in general expect to see individual jobs *complete earlier* in GRUB algorithm than in non-reclaiming servers.

6.3.3 FORMAL ANALYSIS OF GRUB

This section shows that the GRUB algorithm closely emulates the performance that servers would experience if they were executing on dedicated processors with lower capacity.

To precisely evaluate the GRUB ability of emulating a dedicated "virtual" processor, we first characterize the system behavior when each task τ_i executes on a dedicated processor of capacity U_i; then, we show how well the GRUB algorithm is able to emulate a virtual dedicated processor for each server.

Dedicated processor. Let A_i^j and F_i^j be the instants that job τ_i^j would begin and complete execution, respectively, if server S_i were executing on a dedicated processor of capacity U_i. The following expressions for A_i^j and F_i^j can be easily derived:

$$
\begin{aligned}
A_i^1 &= a_i^1 \\
F_i^1 &= A_i^1 + \frac{c_i^1}{U_i} \\
A_i^j &= \max\left(F_i^{j-1}, a_i^j\right), \text{ for } j > 1 \\
F_i^j &= A_i^j + \frac{c_i^j}{U_i}, \text{ for } j > 1
\end{aligned} \tag{6.3}
$$

GRUB virtual processor. In 2000, Lipari and Baruah [GB00] bounded the error introduced by the GRUB algorithm when emulating a virtual processor of capacity U_i. In fact, they proved that the following inequality holds:

$$f_i^j \leq A_i^j + \left\lceil \frac{(c_i^j / U_i)}{P_i} \right\rceil \cdot P_i, \tag{6.4}$$

where f_i^j denotes the time at which GRUB completes execution of job τ_i^j.

By using the results of Equation 6.3 and Equation 6.4, the following theorem can be easily proved:

Theorem 6.3 *The completion time of a job of server S_i when scheduled by the GRUB algorithm is less than P_i time units after the completion-time of the same job when S_i has its own dedicated processor.*

Proof.
Observe that

$$\begin{aligned}
f_i^j &\leq A_i^j + \left\lceil \frac{(c_i^j / U_i)}{P_i} \right\rceil \cdot P_i \\
&< A_i^j + \left(\frac{c_i^j}{U_i \cdot P_i} + 1 \right) \cdot P_i \\
&= A_i^j + \frac{c_i^j}{U_i} + P_i \\
&= \left(A_i^j + \frac{c_i^j}{U_i} \right) + P_i \\
&= F_i^j + P_i \quad \text{(By Equation 6.3)}
\end{aligned}$$

Thus, f_i^j (the completion time of the j-th job of server S_i when scheduled by the GRUB algorithm) is strictly less than P_i plus F_i^j (the completion-time of the same job when S_i has its own dedicated processor). □

It is worth noting that the above theorem helps to decide how to set the server period, which is a system parameter. In fact, the period P_i is an indication of the "granularity"

of time from server S_i's perspective; as a consequence, the smaller the value of P_i, the more fine-grained the notion of real time for S_i, even though an additional cost is introduced in terms of algorithm overhead (the deadline postponement of each server is a function of the server period).

6.4 OTHER FORMS OF RECLAIMING

In the previous sections, the CASH and GRUB algorithms have been described. While those algorithms represent powerful solutions when trying to reuse reserved but under-utilized CPU bandwidth, they cannot always be used due to their implementation costs. For instance, real-time embedded devices equipped with a microcontroller are very resource constrained in terms of available memory and CPU speed. In such a scenario, a light real-time operating system (in terms of memory footprint and context switch overhead) is a must due to the limited availability of resources. Moreover, these devices have also a limited temporal horizon when storing a variable with dense granularity. As a consequence, consecutive deadline postponements could overflow the server deadline variable introducing dangerous flaws in the kernel.

To limit these problems while still providing effective scheduling policies, we now introduce two simple reclaiming techniques characterized by a low computational cost, making them more suitable for small embedded devices.

6.4.1 ADVANCING SERVER DEADLINES

When using a CBS server, a task requiring additional CPU time can immediately recharge its budget and continue its execution with a postponed server deadline. Such a CBS property allows exploiting free CPU cycles whenever possible. However, too many consecutive deadline postponements could cause the server priority to become lower than that of the other servers. Such a problem does not occur when a server reclaims spare capacity from another under-loaded server, but it only appears when consuming its own bandwidth. This problem is illustrated in Figure 6.6, where there are two servers, S_1 and S_2 with maximum budget $Q_1 = 1, Q_2 = 3$, and server periods $T_1 = 4$ and $T_2 = 4$, respectively. In the example, server S_1 receives five requests of duration $C_{ape} = 1$ at times $0, 1, 2, 3, 4$ (they are indicated by dashed arrows). Notice that, while server S_1 is backlogged in interval $[0, 4]$ and four deadline postponements are required to serve the aperiodic jobs, server S_2 is idle up to time $t = 4$, when it receives its first request of duration $C_{ape} = 3$. Unfortunately, when server S_2 becomes active with deadline $d_2 = 8$, S_1 deadline is postponed until time $t = 20$; it follows that S_1 suffers starvation within interval $[4, 13]$ even though S_2 was idle during interval $[0, 4]$.

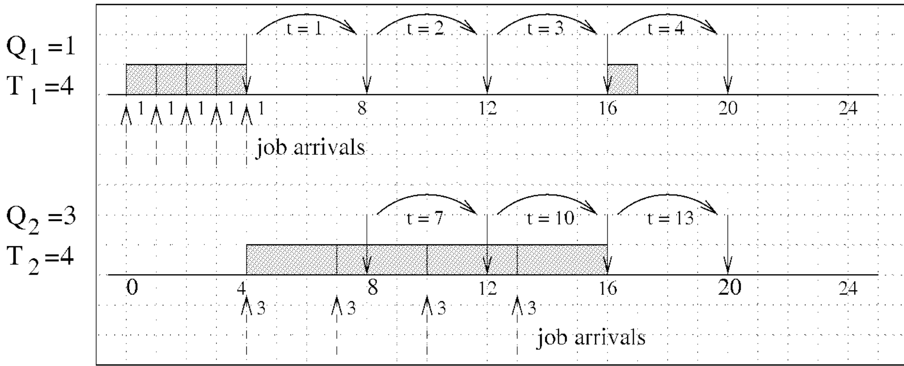


Figure 6.6 Example of CBS servers without resource reclaiming.

This problem, along with the limited temporal horizon problem of the server deadline, can be effectively contained by exploiting a well known property of the idle times. In fact, under static and dynamic priority scheduling, the following property holds:

Lemma 6.1 (Idle interval lemma) *Given any schedule σ, if an idle interval $[t_0, t]$ occurs, the schedulability of jobs released at or after t is not affected by jobs scheduled before t. Hence, as far as the task set schedulability is concerned, instant t can be chosen as the new system start time, ignoring all the scheduling events before t.*

A direct consequence of the *idle interval lemma* is to allow all the CBS servers to restart their budget and deadline every time an idle interval occurs. It is worth noting that the idle time interval lemma also applies to a "zero length" idle interval ($t_0 = t$). Such an anomalous idle interval occurs at time t whenever all the jobs released before t complete by instant t and at least one new job (J_{new}) is released exactly at t. The visible effect of this anomaly is that the processor is never idle, but from a scheduling point of view this event can be considered as a "zero length" idle interval [1] occurring at time t. This type of reclaiming, called **Deadline Advancement**, is illustrated in Figure 6.7, where the same task set of Figure 6.6 is analyzed.

According to the example above, it can be noticed that a zero length idle interval occurs at times $t = 1, 2, 3, 4$; as a consequence, server S_1 can be restarted four times before server S_2 starts to serve its first request. After that, both servers behave according to the classical CBS rules. It follows that the last S_1 job completes at time $t = 8$, instead of $t = 14$, by exploiting the deadline advancement technique.

[1] The reader can be easily convinced just imagining to delay J_{new} by an amount ϵ arbitrarily small: it immediately follows that an idle interval $[t, t + \epsilon]$ occurs, so that the validity of the idle interval lemma is finally claimed.

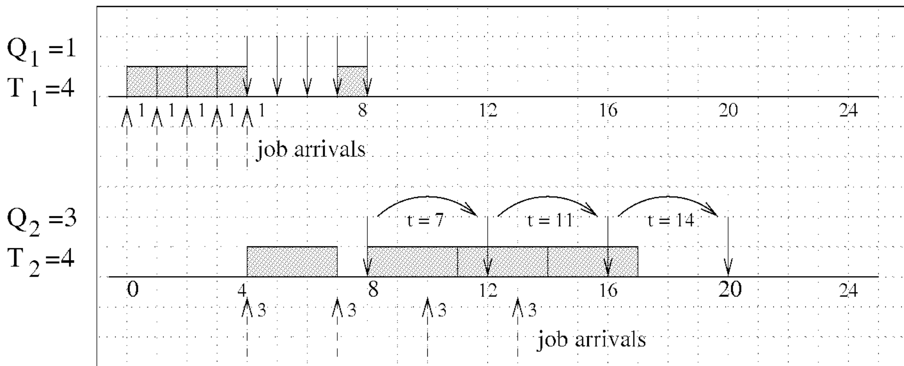


Figure 6.7 Example of CBS servers with deadline advancement.

6.4.2 CBS WITH BUDGET ADJUSTMENT

In this section we introduce another "light" (in terms of computational cost) technique that relies on the key idea of the CASH algorithm. To ensure the temporal correctness of the system, however, this method does not require maintaining a sorted queue of spare capacities, nor keeping track of the CPU idle intervals. Even though CASH orders all the current available capacities by absolute deadline to allow more than one server to have benefit from the reclaiming strategy, it is worth noting that, most of the time, only the subsequent active CBS with the highest priority will exploit the extra capacity, inheriting it from the previous running CBS.

According to the consideration above, the CASH algorithm can be significantly simplified, still achieving reasonable performance in terms of resource reclaiming. The resulting approach, named **Budget Adjustment**, is an extension of the CBS server by adding the following rule:

- **Rule:** Whenever the current executing CBS server S_a becomes idle and has some residual capacity ($\bar{Q}_r$) left, such an amount is transferred to the subsequent CBS server S_b (if any) present in the scheduler (EDF) ready queue. If there is no available server, the residual capacity is not transferred, but it is maintained by the idle CBS according to the classical CBS rules.

The validity of the above rule directly derives from the CASH properties. In fact, since the EDF ready queue is ordered by increasing deadlines, the residual capacity transfer can only occur from a server S_a to a server S_b with absolute deadline $d_b \geq d_a$. By

contradiction, if $d_b < d_a$, server S_b would be inserted at the head of the ready queue and a preemption would occur. Hence, according to the CASH rules, as the absolute deadline d_b is greater than or equal to the absolute deadline d_a, the capacity transfer from server S_a to server S_b can be safely performed.

To better understand the difference between *CASH* and *Budget Adjustment*, the task set of the example reported in Section 6.2 (used for the CASH algorithm) is illustrated here to show how this simple technique is able to reclaim spare resources.

Consider a task set consisting of two periodic tasks, τ_1 and τ_2, with periods $P_1 = 4$ and $P_2 = 8$, maximum execution times $C_1^{max} = 4$ and $C_2^{max} = 3$, and average execution times $C_1^{avg} = 3$ and $C_2^{avg} = 2$. Each task is scheduled by a dedicated CBS having a period equal to the task period and a budget equal to the average execution time. Hence, a task completing before its average execution time saves some budget, whereas it experiences an overrun if it completes after. A possible execution of the task set is reported in Figure 6.8, which also shows the capacity of each server and the capacity transfer among servers whenever the "Budget Adjustment" policy allows it.

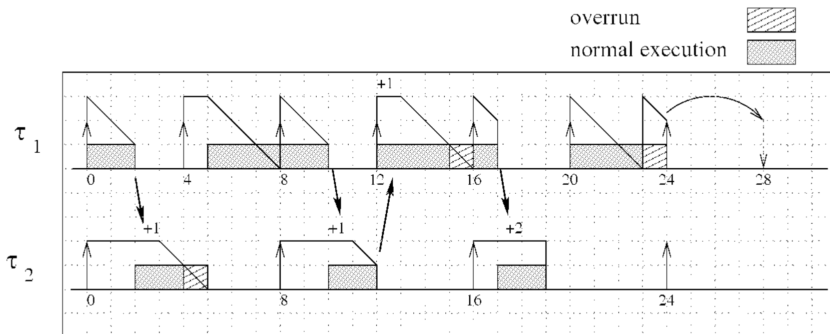


Figure 6.8 Example of resource reclaiming with the Budget Adjustment method.

At time $t = 2$, task τ_1 has an early completion, thus a residual capacity equal to one is transferred to task τ_2. After that, τ_2 consumes the above residual capacity before starting to use its own capacity; hence, at time $t = 4$, a τ_2 overrun is handled without postponing its deadline. Notice that each task tries to use residual capacities before using its own capacity and that, whenever an idle interval occurs (e.g., interval [19, 20]), the residual capacity (if any) cannot be transferred to any other server, but it is held by its owner to maintain the task set feasibility. Comparing this example with the one obtained by CASH (see Figure 6.2), it is worth noting that the resulting schedule is identical, except for the residual capacity of τ_2 at time $t = 19$, which

cannot be exploited by the next instance of τ_1. As a result, the overrun of τ_1 at time $t = 23$ causes a deadline postponement and a budget recharge to complete its execution. When CASH is used, such a deadline postponement is not needed, so increasing system performance. As a final remark, it is worth noting that, whenever a CBS server is used to handle aperiodic tasks, both the CASH and the Budget Adjustment technique can reduce aperiodic responsiveness due to the unpredictable arrivals of aperiodic requests. As a consequence, the user could decide to prevent an aperiodic server from freeing its spare capacity, since an aperiodic request might need it later. In such a case, spare capacities are generated only by periodic tasks and exploited either by periodic or aperiodic activities.

7

QUALITY OF SERVICE MANAGEMENT

In the previous chapters, the term "QoS" has been informally defined as something related to the quality perceived by the final user, making the implicit assumption that the QoS level is related to the number of respected deadlines.

To provide a more formal QoS definition, the quality achieved by each application must be quantified in some way, and this is usually done by associating a *utility* value to each task or application. Hence, QoS management can be defined as a resource allocation problem, and mathematical tools for solving it already exist.

The fundamental issue for formulating and solving such an optimization problem is to use an adequate *QoS model* that univocally maps subjective aspects (such as the perceived quality that may depend on the user) to objective values (such as the utility, that is a real number). In this chapter we will recall the most important QoS models and their applications.

7.1 THE QOS-BASED RESOURCE ALLOCATION MODEL

The QoS-based Resource Allocation Model (Q-RAM) [RJMO97] is a general framework that allows the user to describe multiple QoS dimensions, map them to a utility value, and solve the resource assignment problem with respect to multiple resources.

The first important aspect of this model is that each application is not characterized by a single constraint, but may need to satisfy multiple requirements. This is an important difference with respect to the traditional real-time model, in which each task is characterized by a single deadline or priority. An audio streaming program is

a typical example of application that can take advantage of the rich QoS description provided by Q-RAM: the audio data must be decoded and put in the audio card buffer before the buffer is empty (and this is a *timing* constraint), but can also be provided at different *sampling rates* and *encodings* (affecting the final quality of the reproduced audio). Moreover, different *compression* mechanisms can be used to reduce the needed network bandwidth, and they may introduce some loss in the quality of the decoded audio. Finally, there may be some additional *latency* constraints, or data may require to be *encrypted*.

In summary, some important QoS dimensions may be: timeliness, security, data quality, dependability, and so on. Each application may have a minimum QoS requirement along each dimension (for example, the audio must not be sampled at less than 8 bits per sample). In some cases, a maximum QoS requirement along some dimensions can make sense too (for example, it may be useless to sample audio data at more than $48000KHz$).

Another important characteristic of the Q-RAM model is that it recognizes that each application needs different resources for its execution (for example, each application will surely need some CPU time and some memory), and decreasing the need for one resource may increase the need for a different one. For example, let us consider the audio streaming application again: the amount of network bandwidth needed for receiving an audio stream can be decreased by compressing the audio data, but this will increase the CPU requirements. If a formalized model like Q-RAM is not used, finding the correct resource trade-off may be difficult, and empirical methodologies are often used to tune the application.

Formally speaking, Q-RAM models a system as a set of concurrent applications $\{\tau_1, \ldots, \tau_n\}$ (each application is considered as composed by a single task) served by using a set of resources $\{R_1, \ldots, R_m\}$. Since each resource R_j is shared (temporally or spatially) among different tasks and has a maximum capacity R_j^{max}, the goal of a resource allocation algorithm is to find an allocation R that is optimal based on some optimality criterion. An allocation R is a matrix composed by elements $R_{i,j}$ indicating the amount of resource R_j allocated to application τ_i. The resource allocation algorithm must ensure that $\forall j, \sum_{i=1}^{n} R_{i,j} \leq R_j^{max}$.

Note that R_j^{max} depends both on the resource and on the algorithm used to allocate the resource. For example, considering the CPU, R_{CPU}^{max} is 1 if EDF is used as a scheduling algorithm, and can be 0.69 if RM is used. To be generic enough, Q-RAM assumes that each resource is scheduled so that an amount $R_{i,j}$ of R_j is assigned to task τ_i, but does not make any assumption on the particular scheduling algorithm (the scheduling algorithm behavior is modeled through R_j^{max}). Returning to the CPU example, any algorithm providing temporal protection (such as a reservation algorithm or a proportional share algorithm) can be used.

In order to define an optimality criterion, each application is assigned a *utility* U_i, defined as the value achieved by assigning $(R_{i,1}, \ldots, R_{i,m})$ to τ_i. To be more precise, the utility is a function $U_i(R)$ of the resource assignment; for this reason, $U_i(R)$ is referred to as the *utility function* of τ_i. The total system utility is defined as a weighted sum of all the applications' utilities:

$$U(R) = \sum_{i=1}^{n} w_i U_i(R), \tag{7.1}$$

where w_i is the importance of application τ_i.

Note that throughout this chapter the $U_i()$ symbol is used to denote the utility function, and not the utilization as in the rest of the book. This notation has been adopted for consistency with the original work.

Since every application is characterized by multiple QoS dimensions Q_k, the application utility is decomposed along such different dimensions: in other words, $U_i(R)$ is a function of the *dimensional resource utility* $U_{i,k}(R)$, defined as the value achieved along the QoS dimension Q_k by application τ_i under the resource assignment R. The QoS requirements of each application combined with the dimensional resource utility functions result in *minimal resource requirements* $R_{i,j}^{min_k}$ on each QoS dimension Q_k. An application τ_i is feasible if it satisfies its minimal requirements on every QoS dimension. The minimum amount of resource R_j needed by application τ_i is given by $R_{i,j}^{min} = \sum_k R_{i,j}^{min_k}$.

The previous definitions provide the foundation for a QoS optimization problem that can be successfully solved under the following assumptions:

1. Applications are independent.
2. The system has enough resources to satisfy the minimal resource requirements of each application. In other words, $\forall j, \sum_{i=1}^{n} R_{i,j}^{min} \leq R_j^{max}$.
3. The utility functions $U_i()$ (and the dimensional utility functions $U_{i,j}()$) are non-decreasing in all the arguments $R_{i,j}$.

Note that the first assumption is only used to simplify the analysis, but it is not strictly needed. Also, note that the application importance w_i can be eliminated from the model by considering the weighted utility function $w_i U_i()$ instead of $U_i()$. At this point, it is clear that the goal of Q-RAM is to find a matrix R such that:

1. The schedulability constraint is satisfied: $\forall j, \sum_{i=1}^{n} R_{i,j} \leq R_j^{max}$;

2. The minimal resource requirements are satisfied: $\forall i, j, R_{i,j} \geq \sum_k R_{i,j}^{min_k}$;
3. The system utility $U(R)$ is maximized.

The generic case (multiple resources and multiple QoS dimensions with generic utility functions) is not easy to solve, hence the authors propose different solution algorithms that are valid under some simplifying assumptions. The single resource and single QoS dimension is analyzed first, under the additional hypothesis that the utility function $U_i(R)$ is twice continuously differentiable and concave. Let R_1 be the single resource considered in this case. Because of Assumption 2 (the minimum resource constraints can be satisfied), it is possible to focus only on the allocation of the "excess resource" $R'_i = R_{i,1} - R_{i,1}^{min_1}$ (the j and k indexes have been removed from R' because there is only one resource and one QoS dimension). By definition, $R'^{max} = R_1^{max} - \sum_{i=1}^{n} R_{i,1}^{min_1}$, and $R_i'^{min} = 0$. Standard results from operational research (the Kuhn Tucker theorem) ensure that a resource allocation R' is optimal only if $\forall i, R'_i = 0$, or for any $(i, h) : R'_i > 0$ and $R'_h > 0$, the first derivative of $U_i()$ computed in R'_i is equal to the first derivative of $U_j()$ computed in R'_j. Note that this condition is necessary, but not sufficient. Based on this result, it is possible to find an optimal allocation R', by using an iterative algorithm. At each step, the following quantities are computed:

- $R'_i = R_{i,1} - R_{i,1}^{min_1}$ is the excess resource currently allocated to τ_i;
- $U'_i()$ is the first derivative of the utility function;
- $\Gamma' = \{\tau_i : U'_i(R'_i) = \max_i\{U'_i(R'_i)\}\}$;
- $U'_m = \max_{i:\tau_i \notin \Gamma'}\{U'_i(R'_i)\}$.

Based on such definitions, the algorithm works as follows:

1. The algorithm starts by assigning the minimum resource requirement R_i^{min} to each task τ_i; since the excess resource R'_i is considered, the algorithm starts with $R'_i = 0$.
2. Each step begins by computing $U'_i(R'_i)$ and Γ'.
3. If $\max_i\{U'_i(R'_i)\} = 0$, then the algorithm stops because the current assignment is optimal.
4. If $U'_m = 0$, then the excess resource R'_i allocated to each task $\tau_i \in \Gamma'$ is increased so that the corresponding $U'_i(R'_i)$ are decreased (in a way to keep them equal to each other) until either they become equal to U'_m, or the entire resource R_1 is allocated (i.e., $\sum_i R'_i = R'^{max}$). In this second case, the algorithm stops.

5. If the algorithm does not stop at step 4, then Γ' is increased (because now $U'_m = \max_i\{U'_i(R'_i)\}$), and the algorithm returns to step 2.

If an application is characterized by multiple QoS dimensions, the problem is more complex because the optimal allocation depends on the relationship among such QoS dimensions. In general, QoS dimensions can be dependent or independent.

Two QoS dimensions are independent if a quality variation in one of them does not change the amount of resource needed to keep the quality level on the other dimension stable. In this case, dimension utilities are additive. Conversely, two QoS dimensions are dependent if a quality variation in one of them can cause a quality change on the other one, assuming the amount of resources is not increased.

If all the QoS dimensions are independent, then $U_i(R) = \sum_k U_{i,k}(R)$, and the optimization problem can be treated as a single-QoS-dimension problem, by introducing some fake applications τ' that describe the various QoS dimensions. Hence, the resource allocation problem is transformed into an equivalent problem, where the new task set is composed by $n * d$ tasks (remember that n is the number of applications and d is the number of QoS dimensions). Tasks from τ_1 to τ_d will describe the d QoS dimensions of the first application (and will be characterized by the utility functions $U_{1,1}(R), \ldots, U_{1,d}(R)$), tasks from τ_{d+1} to τ_{2d} will describe the QoS dimensions of the second application, and so on. This new problem is a single-QoS-dimension problem, and can be solved using the algorithm presented above.

If QoS dimensions are dependent, solving the problem is more complex. In this case, the total system utility is a multi-dimensional function of the dimensional utilities. If the dimension utility functions $U_{i,k}(R)$ are continuous, imposing $R_i = k$ (remember that there is only a single resource in the system) defines a surface in the QoS space that can be projected on the function that maps dimensional utilities to the system utility. By getting the maximum utility for each $R_i = k$ surface, the problem is again transformed into a single-QoS-dimension problem, and it is possible to apply the algorithm presented above.

The case in which the dimensional utility functions are not continuous (i.e., QoS dimensions are discrete) cannot be treated in this way, and is even more complex. In fact, the authors propose only a nearly-optimal algorithm, and further investigate the problem in a different paper [LRLS98]. In such a paper, the authors prove that solving the optimization problem in the case of dependent and discrete QoS dimensions is NP-hard, and propose an approximation based on a polynomial-time algorithm that provides a solution at a bounded distance from the optimal resource allocation.

If more than one resource is considered (multiple resource problem), the complexity of the optimization problem increases, because some new degrees of freedom are added.

To make resource allocation more tractable with conventional mathematical tools, the authors add some additional constraints: first of all, the system can work according to M different schemes, and for each scheme the utility function mapping the requirements of resource R_j to the utility does not depend on the other resources. Hence, once a scheme and a utility level U are chosen, the requirements for all the resources can be univocally determined. Moreover, all the utility functions are chosen to be linear (with a saturation). In this way, the resource allocation problem can be formulated as a *linearized mixer integer programming problem* that can be solved by using some numerical method.

A taxonomy of the different algorithms that can be used for allocating system resources when Q-RAM is adopted, together with a comparison of such algorithms based on accuracy and computational cost, can be found in [CLS99].

As a final remark, note that utility functions are generally the results of a subjective evaluation of the output quality, and may depend on the user (what a final user evaluates as a "good quality" can be unsatisfactory for a different user). Hence, assigning utility functions is not easy. In some cases, however, utility values can be deterministically associated to a resource configuration. For example, in control applications it is easy to define a control metric that describes the quality of the control action. It can be based on the period of the control tasks or on other quantities dependent on the amount of resources allocated to the control tasks. This will be better explained in Section 7.3.

7.2 STATIC VS. DYNAMIC RESOURCE MANAGEMENT

As explained in the previous section, the problem of designing a system fulfilling some specified QoS requirements can be often mapped to a resource management problem. Abstract QoS models, like Q-RAM, permit to map high-level specifications into low-level requirements, so that it is possible to assign the proper amount of resources to each task. Such a resource management can be performed either statically or dynamically.

Static resource management is performed at a system design phase, and can be formulated as an off-line optimization problem. During system design, if requirements and resource consumptions of each task are known in advance, then it is possible to formulate the optimization problem and solve it to find the optimal resource assignment and scheduling parameters. In this case, the complexity of the optimization algorithm and the time needed to find an optimal solution are not much critical, and the accuracy of the solution is the most important factor.

Static (a-priori) resource assignment has been traditionally used in designing critical real-time systems, and it is still a good choice for those systems in which an objective QoS metrics exists, and the relation between resource usage and QoS level is clear and known. Control systems are a good example: in a feedback controller, the quality of the control action can be clearly expressed by an objective metric, based on the difference between the response of the closed loop system and the desired response.

If a controller is designed to obtain a closed loop response $y(t)$ to an input $u(t)$, and the actual response of the closed loop system to $u(t)$ is $y'(t)$, then a control performance index can be defined as a quantity that is somehow proportional to the integral of $|y(t) - y'(t)|$, $(y(t) - y'(t))^2$, or something similar.

Moreover, a real-time controller is typically a static system, in which the resource requirements of control tasks can be known a priori. Hence, during the design phase, it is possible to exactly know the amount of CPU (or other system resources) needed to run the control tasks at a specified frequency. The designing techniques traditionally presented in the real-time literature tend to consider only schedulability issues, without considering the effect of the design choices on the control performance. Using a QoS model helps to complement such traditional techniques with a systematic evaluation of the impact of (for example) a frequency assignment on the quality of the final control action. As previously said, an example of usage of the QoS specifications in designing a control system will be presented in Section 7.3.

Dynamic resource management can be performed at run time to better cope with system unpredictability, or with the inherent dynamic nature of many real-time applications. In this case, the resource optimization problem is solved on line by an active entity, typically a *QoS Manager*. The QoS manager is a task responsible for dynamically assigning system resources, and tuning them so that the global utility is maximized.

The QoS manager partitions system resources by using a *mechanism* and a *policy*: the mechanism is used for assigning a specified amount of resources to each task, and can be based on modifying the scheduling parameters, or on changing the application behavior. The first approach does not require any modification in the applications, but implies a strict cooperation between the QoS manager and the scheduler. Hence, the QoS manager results to be tightly dependent on the kernel, and on the adopted scheduling algorithm. The second approach allows making the QoS manager and the applications independent of the kernel and of the scheduling algorithm, but requires to heavily modify the applications to support dynamic QoS adaptation. Every "QoS-Aware" application must support different service levels, and must be able to switch between them upon manager requests. Application-level QoS adaptation, as described for example is Section 8.3, is an example of this approach.

The *policy* is used by the QoS manager for deciding how to partition system resources among tasks. Such an assignment can be performed by the QoS manager by solving an on-line optimization problem (similar to the one proposed by Q-RAM) , or by using some kind of heuristics. In this case, the complexity of the optimization algorithm (and the amount of time needed to solve it) becomes relevant.

The QoS manager can perform its dynamic resource assignment decisions based on resource requirements explicitly declared by the applications, or it can use some form of feedback from the system. In the first case, applications have to explicitly declare their requirements and resource consumptions (e.g., by using something similar to the Q-RAM utility functions). Using this approach, the QoS manager decisions can be performed every time an application enters the system, leaves the system, explicitly requires to change its service level, or changes its requirements or declared resource consumptions. An example of this approach is given by the *Elastic Task Model*, presented in Section 2.7.1. When using some form of feedback, the QoS manager periodically monitors system performance and application resource usage to dynamically construct the utility functions. This approach results in a form of feedback scheduling, which is treated in Chapter 8.

Finally, it is worth observing that an abstract QoS model, such as Q-RAM, is fundamental for implementing any form of QoS management or any kind of QoS manager. In particular, the Q-RAM model is generic enough to be used in both dynamic and static resource allocation, and the authors put a lot of effort in developing optimization algorithms that are efficient enough to be used on line.

7.3 INTEGRATING DESIGN & SCHEDULING ISSUES

In digital control, the system performance is a function of the sampling rate: for a given controller design method, faster sampling permits, up to a limit, a better control performance. However, there is a lower bound on the frequency for each task (f_i^{min}, that is, the minimum frequency for each task τ_i), below which the performance is unacceptable because the system becomes unstable. In this formulation, $1/f_i^{min}$ represents the hard relative deadline that each instance of τ_i has to honor.

According to the considerations expressed above, the design of a digital controller should not be performed separately from the system schedulability analysis; in fact, optimal control performance and temporal predictability can only be achieved by addressing control design and scheduling issues together.

The problem of selecting a set of control task frequencies to optimize the system control performance subject to schedulability constraints was addressed by Seto, Lehoczky, Sha and Shin [SLSS97].

7.3.1 PERFORMANCE LOSS INDEX

According to the frequency optimization algorithm described in [SLSS97], each control task τ_i is characterized by a *Performance Loss Index* (PLI[1]), which measures the difference between a digital and a continuous control as a function of the sampling frequency. In particular, if J_c and $J_d(f)$ are the performance indices generated by a continuous-time control and its digital implementation at a sampling frequency f, the PLI is defined as $\Delta J(f) = |J_d(f) - J_c|$, which is convex and monotonically decreasing with the frequency. In [SLSS97], for each control task τ_i, $\Delta J_i(f_i)$ is approximated by the following exponential function:

$$\Delta J_i(f_i) = \alpha_i e^{-\beta_i f_i},$$

where f_i is the frequency of τ_i, α_i is a magnitude coefficient, and β_i is the decay rate. A typical PLI is illustrated in Figure 7.1, where f_m is the lower bound on the sampling frequency.

The performance loss index of the overall system $\Delta J(f_1, ..., f_n)$ is defined as follows:

$$\Delta J(f_1, ..., f_n) = \sum_i w_i \Delta J_i(f_i), \tag{7.2}$$

where w_i is a design parameter determined from the application. For instance, it can be the relative importance of the task in the control system with respect to the others.

Given the available bandwidth (A), the minimum permitted frequency (f_i^{min}), the worst-case execution time ($WCET_i$) and the weighed PLI ($w_i \Delta J_i(f_i)$) of each task τ_i as input parameters, Seto, Lehoczky, Sha and Shin provided an optimization algorithm to compute the frequencies f_i^{opt} which minimize the PLI of the system while guaranteeing the schedulability constraints (i.e., ensuring each task will meet its deadlines). Notice that each task frequency f_i^{opt} computed by this technique is always greater than or equal to the corresponding minimum frequency f_i^{min}. After defining the notion of PLI, the next section describes the control performance optimization algorithm used to assign the task frequencies when using digital control.

[1] In the original formulation, the performance loss index was simply called performance index or PI. In the following, it will be called PLI for more clarity.

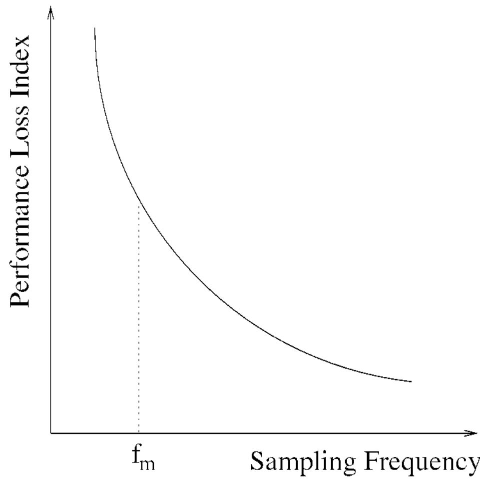


Figure 7.1 Control system Performance Loss Index as a function of the sampling frequency.

7.3.2 OPTIMIZING CONTROL PERFORMANCE

In the previous section, the Performance Loss Index (PLI) for a digital control system has been formally defined. Such a system PLI represents the objective function to be minimized by the control performance optimization algorithm when computing the optimal control frequencies. In fact, the *control optimization problem* can be formulated as:

$$\min_{(f_1,\ldots,f_n)} PLI = \sum_{i=1}^{n} w_i PLI_i(f_i) = \sum_{i=1}^{n} w_i \alpha_i e^{-\beta_i f_i} \tag{7.3}$$

subject to:

$$\sum_{i=1}^{n} c_i f_i \leq 1 \tag{7.4}$$

$$\forall i \in \{1, \ldots, n\}, \quad f_i \geq f_i^{min}, \tag{7.5}$$

where n is the total number of tasks in the system. Having defined the control optimization problem, the following theorem introduces the control performance optimization algorithm for computing its unique optimal solution.

Theorem 7.1 *Given the objective function and the constraints of the "control optimization problem", there exists a unique optimal solution given by:*

$$f_i = \begin{cases} f_i^{min} & when\ i \in \{1, \cdots, p\} \\ \frac{1}{\beta_i}\left(\ln \Gamma_i - Q\right) & when\ i \in \{p+1, \cdots, n\} \end{cases} \tag{7.6}$$

where f_i are ordered as f_i^{min}, and according to the following inequalities

$$\Gamma_1 e^{-\beta_1 f_1^{min}} \leq \Gamma_2 e^{-\beta_2 f_2^{min}} \leq ... \leq \Gamma_n e^{-\beta_n f_n^{min}}, \tag{7.7}$$

$p \in [0, ..., n]$ is the smallest integer such that

$$\sum_{l=1}^{p} c_l f_l^{min} + \sum_{l=p+1}^{n} \frac{c_l}{\beta_l}\left(\beta_p f_p^{min} + ln\frac{\Gamma_l}{\Gamma_p}\right) \geq 1, \tag{7.8}$$

and

$$\Gamma_i = \frac{w_i \alpha_i \beta_i}{c_i} \tag{7.9}$$

$$Q = \frac{1}{\sum_{l=p+1}^{n} \frac{c_l}{\beta_l}}\left(\sum_{l=1}^{p} c_l f_l^{min} + \sum_{l=p+1}^{n} \frac{c_l}{\beta_l}\, ln\Gamma_l - 1\right). \tag{7.10}$$

In practice, the first step for identifying the optimal value of each f_i is to evaluate the parameter p; that is, the smallest integer p such that equation (7.8) is verified. After that, the second step is straightforward and is just consists in evaluating equation (7.6) for each task.

AN EXAMPLE

The following example will clarify how the control performance optimization algorithm works. The technique is applied to a bubble control system, which is a simplified model designed to study diving control in submarines [SLSS97]. The bubble control system considered here consists of a tank filled with air and immersed in the water. Depth control of the diver is achieved by adjusting the piston connected to the air bubble. In this example, a camera monitors the diver as sensor for getting its position.

Suppose that two such systems with different physical dimensions are installed on an underwater vehicle to control the depth and orientation of the vehicle, and assume they are controlled by one on-board processor. The task set parameters are shown in Table 7.1, where, for each bubble control system i, $WCET_i$ (ms) is the control task worst-case execution time in each sampling period, f_i^{min} (Hz) is the lower bound on sampling frequency, and w_i is the weight assigned to system i.

The following data are given for the control design and scheduling problem: $\Delta J_i = \alpha_i e^{-\beta_i f_i}$, $i = 1, 2$, where the frequencies f_i must be determined.

Task	α_i	β_i	$WCET_i$ *(ms)*	f_i^{min} *(Hz)*	w_i
τ_1	1	0.4	25	10	2
τ_2	1	0.1	25	20	1

Table 7.1 Task parameters for the bubble control system.

A simple computation shows that the total CPU utilization of the overall bubble system is 75% when the minimum task frequencies are assigned. Supposing the total CPU utilization available for the bubble systems is 100%, the control performance optimization algorithm allows assigning the optimal task frequencies to fully utilize the CPU. In particular, to compute the frequencies f_i^{opt}, the correct value of the parameter p ($p = 0$) must determined first; then, the optimal frequencies are computed by means of equation (7.6). In conclusion, it follows that $f_1^{opt} = 12.16Hz$, and $f_2^{opt} = 27.84$ achieving a resulting Performance Loss Index $PLI = 0.0772$.

7.4 SMOOTH RATE ADAPTATION

In real-time applications that involve human-computer interactions, the quality of a delivered service depends not only on the absolute level of performance, but also on the way performance is changed during workload variations. For example, while

watching a movie, a continuous transition between color and black/white mode is considered much more annoying than watching the entire movie in black and white. In general, when human factors are involved in measuring the quality of a computing system, smooth QoS transitions are always preferred with respect to abrupt variations.

When considering periodic activities, the QoS can often be adapted by changing the activation rate of the application, and smooth QoS adaptation can be implemented by enforcing a gradual transition of the period. Typically, a rate change may be caused either by the task itself, as a response to a variation occurred in the environment, or by the system, as a way to cope with an overload condition. For example, whenever a new task cannot be guaranteed by the system to meet its timing constraints, instead of rejecting the task, the system can try to reduce the utilizations of the other tasks (by increasing their periods in a controlled fashion) to decrease the total load and accommodate the new request.

The problem of rate adaptation during overload conditions has been widely considered in the real-time literature and has been treated in Section 2.7. In this section, we describe a method for achieving smooth rate transitions in periodic tasks that are required to adapt to abrupt environmental or system changes. This method was originally proposed by Buttazzo and Abeni [BA02b] as an extension of the elastic task model (see Section 2.7.1). According to the elastic model, tasks utilizations are treated as springs with given elastic coefficients. To achieve smooth rate transitions, the model is extended by coupling each spring with a damping device which prevents abrupt period changes.

Hence, each task is characterized by five parameters: a worst-case computation time C_i, a minimum period T_{i_0} (considered as a nominal period), a maximum period $T_{i_{max}}$, an elastic coefficient E_i, and a damping coefficient B_i. The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater E_i, the more elastic the task. The coefficient B_i specifies the damping with which task τ_i performs a period transition: the greater B_i, the higher the damping effect, and hence the smoother the transition. Thus, an elastic task is denoted as:

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i, B_i).$$

From a design perspective, elastic coefficients can be set equal to values which are inversely proportional to tasks' importance, whereas damping coefficients can be set equal to values which are directly proportional to the level of quality specified by the user during transient phases.

In the following, T_i will denote the actual period of task τ_i, which is constrained to be in the range $[T_{i_0}, T_{i_{max}}]$. Any period variation is always subject to an *elastic* guarantee and is accepted only if there exists a feasible schedule in which all the other periods are within their range. In this framework, tasks are scheduled by EDF, hence, if $\sum \frac{C_i}{T_{i_0}} \leq 1$,

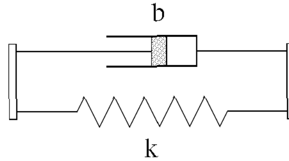


Figure 7.2 A damped elastic element.

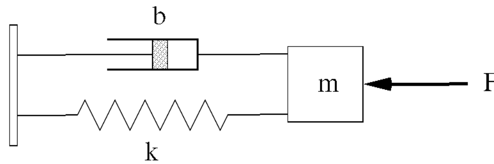


Figure 7.3 A generic mechanical impedance.

all tasks can be created at the minimum period T_{i_0}, otherwise the elastic algorithm is used to adapt the tasks' periods to T_i such that $\sum \frac{C_i}{T_i} = U_d \leq 1$, where U_d is some desired utilization factor.

7.4.1 IMPEDANCE CONTROL

When dealing with damped springs, each elastic element can be modeled as shown in Figure 7.2.

For the sake of completeness, a damped spring is a special case of a system which behaves as a mechanical impedance, with stiffness k, damping b, and mass m, as shown in Figure 7.3.

Such a system responds to an external force F as a second-order system according to the following equation:

$$F = m\ddot{x} + b\dot{x} + kx.$$

For linear, time-invariant continuous systems, the impedance Z is defined as the ratio of the Laplace transform of the effort (F) and the Laplace transform of the flow ($\phi = \dot{x}$), hence

$$Z(s) = \frac{F(s)}{\Phi(s)} = \frac{F(s)}{sX(s)} = ms + b + k/s.$$

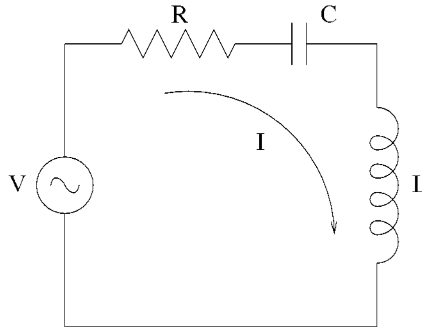


Figure 7.4 An electrical impedance.

From a system point of view, the input/output behavior of a linear system like this is described by the ratio of two variables: the effort and the flow. For a mechanical system, effort is represented by force and torque, and flow is represented by linear and angular velocity. Motors and batteries are equivalent from a system point of view, both being effort sources. Similarly, a current generator or a rotating shaft are both flow sources.

Using an electrical comparison, a mass (inertial element) is equivalent to an inductive element, a damper (dissipative element) is equivalent to a resistor, whereas a spring (conservative element) is equivalent to a capacitor. In this comparison, a force corresponds to a voltage generator, whereas the speed corresponds to a current. Hence, the electrical circuit illustrated in Figure 7.4 is equivalent to the mechanical impedance shown in Figure 7.3, and its electrical impedance $Z(s)$ is expressed as

$$Z(s) = Ls + R + \frac{1}{Cs}.$$

Thus, a damped spring is equivalent to an RC circuit, for which we can write:

$$I(s) = \frac{V(s)}{Z(s)} = \frac{V(s)}{R + 1/Cs} = \frac{Cs}{RCs + 1} V(s).$$

The position of the damping device can be computed as

$$X(s) = \frac{RI(s)}{s} = \frac{RC}{RCs + 1} V(s)$$

and the transfer function of the system can be rewritten as

$$G(s) = \frac{X(s)}{V(s)} = \frac{a}{s + a}$$

where $a = \frac{1}{RC}$ is the system's pole for the RC circuit, and $a = \frac{k}{b}$ in the case of a damped spring. Using the Z-transform, the transfer function can be expressed in a discrete time domain, as follows:

$$\begin{aligned}
G(z) &= \mathcal{Z}\left[\left(\frac{1-e^{-sT}}{s}\right)\frac{a}{s+a}\right] \\
&= (1-z^{-1})\mathcal{Z}\left[\frac{a}{s(s+a)}\right] \\
&= \frac{z-1}{z}\mathcal{Z}\left[\frac{1}{s}-\frac{1}{s+a}\right] \\
&= \frac{z-1}{z}\left(\frac{z}{z-1}-\frac{z}{z-p}\right) \\
&= \frac{1-p}{z-p} = \frac{(1-p)z^{-1}}{1-pz^{-1}}.
\end{aligned}$$

where T is the sampling period and $p = e^{-aT}$.

From $G(z)$, the discrete time equation expressing the position $x(t)$ of the damped spring as a function of the force $F(t)$ becomes:

$$x(t) = (1-p)F(t-1) + px(t-1). \tag{7.11}$$

It is worth observing that any transient law can be used to perform a transition from a period to another. The one expressed by equation (7.11) is just the one which describes the change occurring in a damped spring, which is exponential. In the experiments described below, a linear period transition will be also evaluated.

7.4.2 IMPLEMENTATION ISSUES

The elastic guarantee mechanism can be implemented as an aperiodic task, acting as an `Elastic Manager` (EM), activated by the other tasks when they are created or when they want to change their period. Whenever activated, the EM calculates the new periods and changes them atomically. The overhead caused by the elastic mechanism can easily be taken into account, since the EM is handled using a CBS server with a bounded utilization (e.g., $U_{EM} = 0.05$).

The graceful rate adaptation mechanism can been implemented on top of the EM, as a periodic task, the `Damping Manager` (DM). The purpose of the DM is to perform the

rate transition according to the transition law set by the user. To bound the transition time, the DM runs with a period T_{DM} and performs a full transition in N steps, requiring an interval of NT_{DM} time units.

The DM task can be in two states: active or idle; when the system is started, the DM is idle, and it becomes active when some other task wants to change its period. When the DM task becomes active at time t_0, instead of changing the periods immediately, it gracefully changes them during a transient of size $T = NT_{DM}$.

After N activations, the periods arrive to their final values and the period adapter returns to its idle state, waiting for the next request. More specifically, the graceful adaptation mechanism works as follows:

- When a task τ_i wants to change its period from T_i to T_i^{new}, it posts a request to the Damping Manager.
- When the DM is idle and receives a new request, it becomes active and computes the next period value $T_i(k)$ according to the transition law set by the user. We note that $T_i(0) = T_i$, and after N steps $T_i(N) = T_i^{new}$.
- At each period T_{DM}, the Damping Manager updates the period T_i to the next value $T_i(k)$ and invokes the Elastic Manager to achieve a feasible configuration.
- After N activations, the periods are adjusted to their final values, and the period adapter returns to its idle state.

There are some details to be considered in the implementation of the Damping Manager.

1. Transient period values can be generated by the DM using a generic transition law $T_i(k) = f(k, T_i(k-1), T_i^{new})$. In this chapter, we consider a linear function $T_i(k) = T_i(k-1) + \frac{T_i^{new} - T_i(0)}{N}$ and an exponential function derived from a typical RC circuit (see equation (7.11)): $T_i(k) = (1-p)T_i^{new} + pT_i(k-1)$, where $p = exp(-aT_{DM})$, and $a = \frac{1}{E_i B_i}$.
2. The periods converge to their final values in a transient time $T = NT_{DM}$ that can be specified by the user. If $T = 0$ ($N = 0$), the period change is immediate and the system is equivalent to the classical spring system. Hence, the original elastic model is a special case of the generalized model presented here.
3. If a task requests a period change while the adapter is still active, the request is enqueued and will be served only when all the periods will be stabilized. This is done to simplify the adapter, but other strategies can be adopted in this case.

Task	C_i	T_{i_0}	$T_{i_{max}}$	E_i
τ_1	30	100	500	1
τ_2	60	200	500	1
τ_3	90	300	500	1
τ_4	24	50	500	0

Table 7.2 Task set parameters used for the first experiment. Times are expressed in milliseconds.

7.4.3 EXPERIMENTAL RESULTS

Three sets of experiments are presented here to show the effectiveness of the approach. The first experiment shows the behavior of the elastic compression mechanism (without the damping devices) using the task set shown in Table 7.2. At time $t = 0$, the first three tasks start executing at their nominal period, whereas the fourth task starts at time $t_1 = 10sec$, so creating a dynamic workload variation.

In order to avoid deadline misses during transitions, periods are changed at the next release time of the task whose period is decreased. If more tasks ask to decrease their period, the EM will change them, if possible, at their next release time. See [BLCA02] for a theoretical validation of the compression algorithm.

When τ_4 is started, the task set is not schedulable with the current periods, thus the EM tries to accommodate the request of τ_4 by increasing the periods of the other tasks according to the elastic model. The actual execution rates of the tasks are shown in Figure 7.5. Notice that, although the first three tasks have the same elastic coefficients and the same initial utilization, their periods are changed by a different amount, because τ_3 reaches its maximum period.

To verify the behavior of the Damping Manager, another experiment has been performed using 4 tasks with the parameters shown in Table 7.3. Considering the utilization reserved for the EM, the DM and other device handlers in the system, the effective total utilization U_{max} available for the task set is 0.782. Since $23/100 + 23/100 + 23/100 + 23/100 > 0.782$, the periods are initially expanded by the elastic law, and the tasks start with current periods different from their nominal periods: $T_1 = 107$, $T_2 = 107$, $T_3 = 122$, and $T_4 = 143$. At time $t = 5$, τ_1 issues a request to change its period to $T_1^{new} = 50$, and the DM starts to gracefully adapts the periods. At time $t = 15$, τ_1 issues a request to change its period to 250, and all the other periods can gracefully go to their nominal values. In this experiment, the transient periods $T_i(k)$ for τ_1 were generated using a linear law. The result of this experiment is illustrated in Figure 7.6, which shows the number of executed jobs as a function of time. Figure

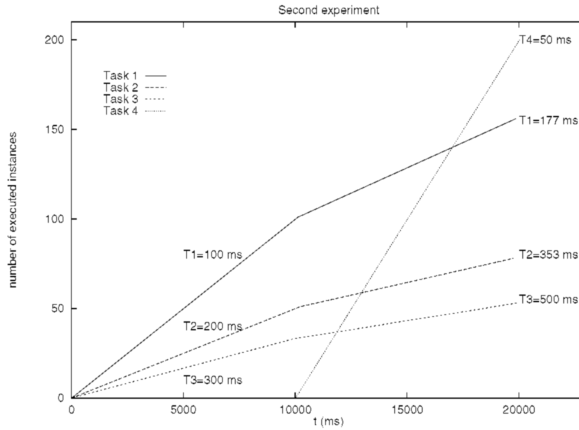


Figure 7.5 Dynamic task activation.

Task	C_i	T_{i_0}	$T_{i_{max}}$	E_i	B_i
τ_1	23	100	500	1	1
τ_2	23	100	500	1	1
τ_3	23	100	500	3	1
τ_4	23	100	500	5	1

Table 7.3 Task set parameters used for the second experiment. Times are expressed in milliseconds.

7.7 shows how task periods evolve during the transition. It is worth observing that, although T_1 is modified using a linear transition law, the other periods vary according to a non-linear function. This happens because, when T_1 is decreased, the total processor utilization increases, so the Elastic Manager performs a compression of the other tasks (enlarging their periods) to keep the total load constant. Given the non linear relation between total utilization and periods ($U = C_1/T_1 + \ldots + C_n/T_n$), the other periods change in a non-linear fashion. Finally, Figure 7.8 shows the period evolution when an exponential law is used for τ_1 to modify its period.

A different experiment has been performed using the task set shown in Table 7.4 to test the behavior of the DM in the presence of dynamic task activations. In this case, when a new task τ_h needs to enter the system with period T_h^* and there exists a feasible elastic schedule for it, it cannot be immediately activated with that period. In fact, the other tasks have to gradually reduce their utilizations (according to the damping law)

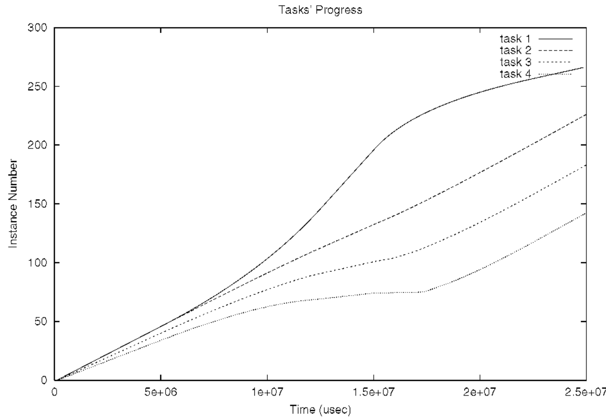


Figure 7.6 Number of processed jobs as a function of time when τ_1 changes its period with a linear law.

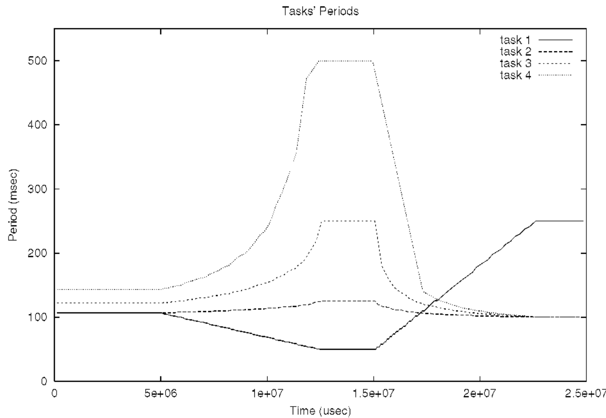


Figure 7.7 Periods evolution as a function of time when τ_1 changes its period with a linear law.

to decrease the load and create space for the new task. As a consequence, the new task is activated with a large period (theoretically infinite, practically equal to MAXINT), which is gradually reduced to the final T_h^* value by applying the damping law. We note that the time required to the transition is always bounded to NT_{DM}, where N is the number of steps fixed for the transition and T_{DM} is the period of the Damping Manager.

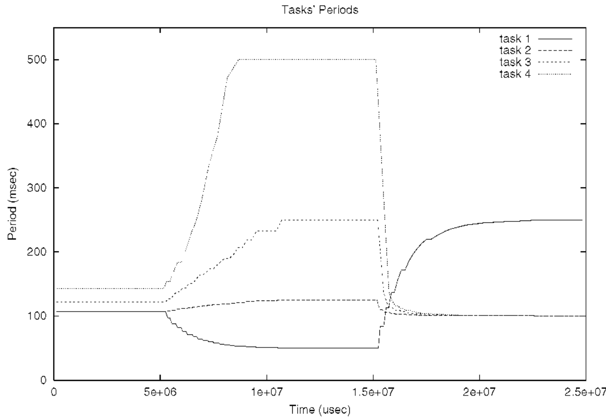


Figure 7.8 Periods evolution as a function of time when τ_1 changes its period with an exponential law.

Task	C_i	T_{i_0}	$T_{i_{max}}$	E_i	B_i
τ_1	40	100	500	1	1
τ_2	40	100	500	1	1
τ_3	40	100	500	1.5	1
τ_4	40	100	500	2	1

Table 7.4 Task set parameters used for the third experiment. Times are expressed in milliseconds.

In the experiment, task τ_2 was added at time $t = 5sec$, and the DM started to decrease its period towards the final value $T_{2_0} = 100ms$. Figures 7.9 and 7.10 show the results of this experiment when a linear transition law was used. It is worth noticing that, as a consequence of τ_2 arrival, all the task periods begin to gracefully expand to create space for τ_2, thus τ_2 actually begins to execute only when $T_2(k) < T_2^{max}$. From Figure 7.10 it is also interesting to observe that, as T_2 is decreased linearly, the other periods increase exponentially, based on their elastic coefficients, for the same reason explained in the previous experiment.

Figure 7.11 shows the result achieved on the same task set using an exponential transient law. In this case, the activation delay experienced by τ_2 is smaller with respect to the linear case. Moreover, the other periods reach their final values with a much smoother transition.

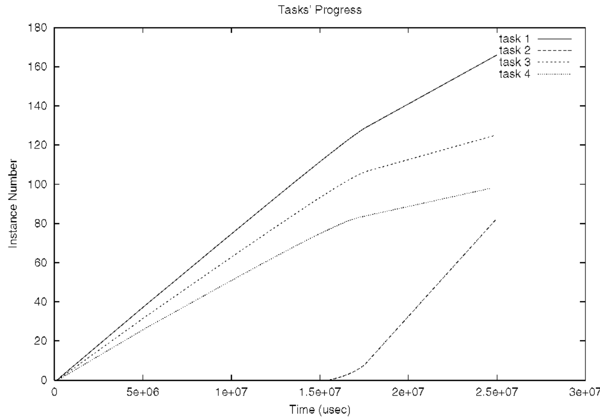


Figure 7.9 Number of processed jobs as a function of time when task τ_2 is dynamically activated and its period is changed using a linear transition law.

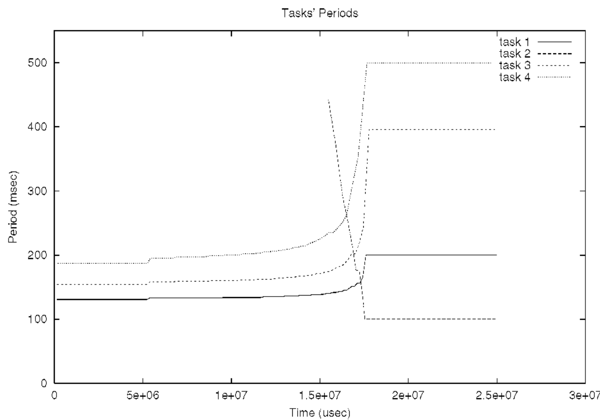


Figure 7.10 Periods evolution as a function of time when task τ_2 is dynamically activated and its period is changed using a linear transition law.

From the experiments presented above, it can be seen that, when an active task wants to change its period (either lower or higher than the current one), a linear transition law is able to achieve smoother period variations on the other tasks. On the other hand, when a new task needs to be activated in the system, an exponential law (for reducing its period to the required final value) is able to vary the other periods more gracefully and it also allows to reduce the activation delay of the newly arrived task.

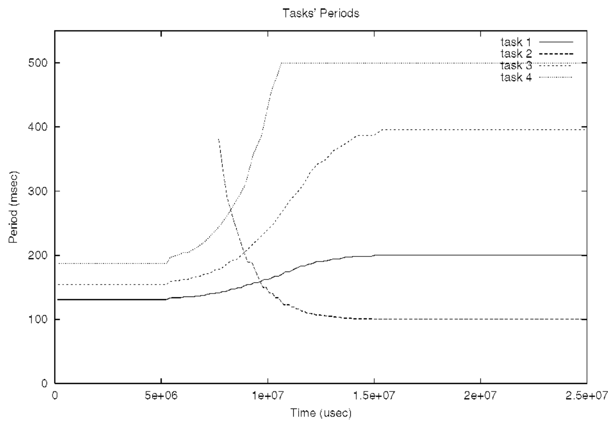


Figure 7.11 Periods as a function of time when T_2 is changed using an exponential transition law.

8

FEEDBACK SCHEDULING

Traditional hard real-time applications are designed to respect all deadlines of every task in worst-case scenarios. Although such an approach is very effective when the characteristics of the system are known in advance, it presents some problems for highly dynamic systems, where the characteristics of the environment can vary during system's lifetime, and the total utilization is subject to online (and often unpredictable) fluctuations. In these cases, the classical hard real-time approach suffers from the following problems:

- an exact a-priori knowledge of the tasks' parameters (minimum interarrival times and worst-case execution times) is very difficult to achieve in modern architectures;
- the resulting system is *not flexible*, because it cannot tolerate variations in tasks' parameters or errors in parameters' estimation;
- system's resources are underutilized most of the time, since hard guarantee is performed under pessimistic assumptions and worst-case scenarios (often very rare).

An interesting way to address system's unpredictability is to use some kind of feedback to dynamically adapt the scheduler behavior so that some selected QoS metric is kept under control even in the presence of overload situations. Since the resulting feedback loop creates a reactive system, it is not possible to prevent overloads, but it is possible to minimize their effects. To apply feedback techniques to real-time scheduling, it is necessary:

1. to select a **scheduling algorithm**;
2. to define a **QoS index** to control (the *feedback variable*);
3. to select a scheduling parameter **to be adapted** (the *actuator*).

In this chapter, the use of feedback techniques in real-time scheduling is illustrated in different situations. We first present an adaptive admission control method for controlling the number of missed deadlines of aperiodic jobs. Then, we show how adaptation techniques can be applied to resource reservations (introducing adaptive reservations), and we discuss how to adapt the QoS at the application level. We finally conclude the chapter by presenting some feedback mechanisms based on explicit workload estimation.

8.1 CONTROLLING THE NUMBER OF MISSED DEADLINES

Feedback techniques were originally proposed in time sharing systems [CMDD62], and have been also successively applied to real-time systems [Nak98c, RS01] and multimedia systems [SGG+99]. However, the development of a more theoretically founded basis for feedback scheduling (or closed-loop scheduling) has been advocated only recently.

One of the first proposed real-time closed-loop scheduler, Feedback Control EDF (FC-EDF) [SLS99], was originally developed for working with insufficient resources (i.e., in overload conditions), but it can also be used for handling dynamic systems characterized by unpredictable workloads. In particular, FC-EDF uses a feedback scheme on the EDF scheduling algorithm to tolerate uncertainty in tasks' execution times. The observed variable is the *deadline miss ratio* M, defined as the ratio of the number of missed deadlines and the total number of deadlines in an observation window. Admission control is used as an actuator to affect the system workload. In particular, the control action on the system utilization is performed by rejecting tasks or changing their service level.

When a new task τ_i arrives in the system, it has to pass and admission test that uses information from the feedback to decide whether τ_i can be accepted or not. Every accepted task is characterized by two or more service levels, having different execution times and different qualities of the output (see also the imprecise computation model [HLW91]). A service level controller uses information from the feedback to set the tasks' quality levels so that the system load is increased or decreased when needed. The structure of the resulting scheduler is shown in Figure 8.1.

FC-EDF uses a Proportional Integral Derivative (PID) controller to compute the variation ΔU to apply at the system load based on the observed deadline miss ratio M:

$$\Delta U(t) = -C_P M(t) - C_I \sum_{IW} M(j) - C_D \frac{M(t - DW) - M(t)}{DW},$$

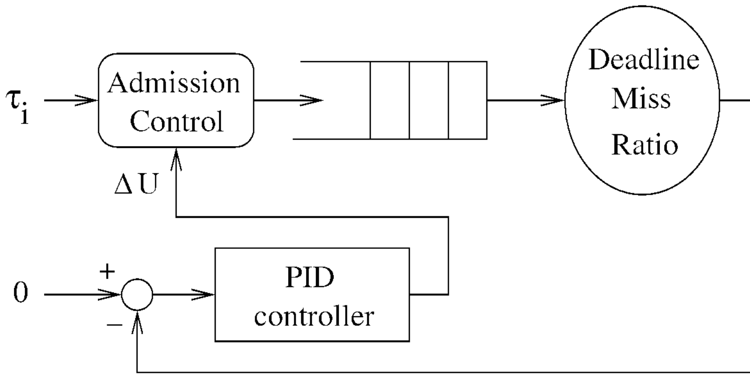


Figure 8.1 Structure of the FC-EDF closed-loop scheduler.

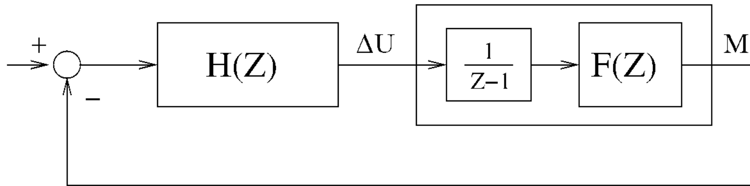


Figure 8.2 Model of the FC-EDF closed-loop scheduler.

where C_P, C_I, and C_D are the controller's parameters, IW is the integration window, and DW is the size of the differentiation interval.

Such a PID controller is used to increase the utilization when the system is not overloaded for a certain period of time. In this way, it is possible to increase system efficiency by accepting a higher number of tasks, or running them at the highest possible quality level. As an alternative [LSTS99], the reference value for the deadline miss ratio can be set to $M^0 > 0$, so that the system utilization is automatically increased when $M(t)$ arrives to 0. Using this setting, a simple PID (without any modification) can be successfully used without underutilizing the system.

The resulting FC-EDF scheduler can be modeled as shown in Figure 8.2; if the EDF scheduler is modeled as a tank (as proposed by Stankovic and others [SLS99, LSTS99]), it is possible to design the PID controller to properly stabilize the system. This can be done by using control theory, that already provides tools (such as the Z transform) to analyze the behavior of closed loop systems.

However, modeling the EDF scheduler as a simple tank system is an oversimplification that can lead to system instability [LSA+00]. Such an instability is visible when the input workload is constant: in this case, FC-EDF is able to control the deadline miss ratio to 0 in a short time, but after this transient the system continues to experience periodic deadline misses (using control terminology, this is a *limit cycle*). This happens because FC-EDF only monitors the overload (through the deadline miss ratio), and cannot monitor underload situations. The result is a control saturation (the deadline miss ratio cannot be less than 0): for example, the controller cannot make any distinction between a situation with $M = 0, U = 0.9$ and a situation with $M = 0, U = 0.1$. To avoid system underutilization, FC-EDF always tries to increase the system load when $M = 0$, but this causes the limit cycle.

The instability problem can be solved by monitoring both the deadline miss ratio and the system utilization [LSA+00]: the FC-EDF^2 scheduler uses two different PID controllers: the first one computes ΔU based on the deadline miss ratio, whereas the second one computes ΔU based on the utilization. The control signal is then selected by choosing the minimum between the outputs of the two controllers. In this way, FC-EDF^2 is able to achieve a stable deadline miss ratio in all workload conditions.

8.2 ADAPTIVE RESERVATIONS

One of the major problems presented by feedback schemes like FC-EDF is that they can only control a *global* QoS metric, such as the total number of missed deadlines, but they cannot control the performance of each single task. This issue can be addressed through a reservation-based approach (see Chapter 3), which allows using an independent controller for each reservation in the system (and hence for each task).

If a CPU reservation is used to schedule a task τ_i, then the amount of CPU time Q_i^s (or the amount of CPU bandwidth U_i^s) reserved to τ_i can be used as an actuator, and the number of reservation periods needed to serve each job can be used as an observed value. For example, the reservation and feedback approaches can be combined to adjust tasks' periods according to the actual CPU load [Nak98c], or the CPU proportion of each process can be adapted to control the task's performance. As an alternative, if the processes served by reservation-based scheduler are organized in a pipeline then the adaptation mechanism can control the length of the queues between pipeline's stages [SGG+99].

Adaptive Reservations are an interesting abstraction that allows separating task parameters from scheduling parameters [AB99a]. In fact, the traditional task models used in the real-time literature are useful to directly map each task to proper scheduling parameters, but have the disadvantage of exporting some low-level details of the scheduling

algorithm. Since users are not generally interested in such details and do not often know all tasks parameters, in many cases the (C, T) model is very different from the real needs, hence programmers are forced to assign low-level parameters according to complex mapping functions.

These problems can be addressed by introducing high-level task models which provide an interface closer to real user's requirements. In particular, such high-level task models eliminate the need for an a-priori knowledge of the worst-case execution time. Each task τ_i can be characterized by a weight w_i, representing its *importance* with respect to the others, and by some *temporal constraints*, such as a period or a desired service latency.

If the system is overloaded, and the CPU bandwidth is not sufficient to fulfill each task's requirement, a bandwidth compression algorithm has to be introduced to correct the fraction of CPU bandwidth assigned to each task using the task weights w_i (tasks with higher weights will receive a bandwidth nearest to the requested one).

The advantage of such a model is that it separates task temporal constraints (the period T_i, or the rate $R_i = 1/T_i$) from task importance, expressed by the weight w_i. In fact, one of the major problems of classical real-time scheduling algorithms (such as RM or EDF) is that the task importance implicitly results to be proportional to the inverse of the task period.

An adaptive reservation mechanism works as follows: a reservation $(U^s_{i,j}, P^s_i)$ is used to schedule τ_i; when a job $\tau_{i,j}$ finishes, an observed value $\epsilon_{i,j}$ is measured, and a new scheduling parameter $U^s_{i,j+1} = g(U^s_{i,j}, \epsilon_{i,j} \ldots)$ is computed.

In Adaptive Bandwidth reservations [AB99a], the basic scheduling algorithm is the CBS (see Section 3.6.1), and the observed value (the *scheduling error*) is defined as the difference between the latest scheduling deadline assigned by the CBS to a job and the job soft deadline:

$$\epsilon_{i,j} = d^s_{i,j} - (r_{i,j} + T_i).$$

Since the underlying priority assignment is based on EDF, if the server is schedulable, each instance $\tau_{i,j}$ is guaranteed to finish within the last assigned server deadline $d^s_{i,j}$. Hence, the CBS scheduling error $\epsilon_{i,j}$ represents the difference between the deadline $d^s_{i,j}$ that $\tau_{i,j}$ *is guaranteed* to respect and the deadline $d_{i,j} = r_{i,j} + T_i$ that it *should* respect. A value $\epsilon_{i,j} = 0$ means that job $\tau_{i,j}$ met its soft deadline, whereas a value $\epsilon_{i,j} > 0$ means that job $\tau_{i,j}$ completed after its (soft) deadline, because the reserved bandwidth $U^s_i = Q^s_i/P^s_i$ was not enough to properly serve it. Hence, the objective of the system is to control the scheduling error to 0: if this value increases, Q^s_i has to be increased accordingly, otherwise it can be left unchanged.

If $\sum_i U^s_{i,j} > U_{lub}$ (where U_{lub} is the utilization least upper bound of the scheduling algorithm), then the reserved bandwidths must be rescaled using a compression mechanism, to maintain the system schedulable. To better understand the compression mechanism, some additional definitions are needed:

Definition 8.1 *Given a task set* $\Gamma = \{\tau_1, \dots \tau_n\}$ *composed of* n *tasks, a* bandwidth assignment $\hat{U}$ *is a vector* $\hat{U} = (U^s_1, \dots U^s_n) \in R^n$ *such that* $\forall i \leq n, 0 \leq U^s_i \leq U_{lub}$*, and at every time* $U^s_i = U^s_{i,j}$*.*

Definition 8.2 *A bandwidth assignment* $\hat{U}$ *is said to be* feasible *if* $\sum_i U^s_i \leq U_{lub}$*.*

Note that the feasibility of a bandwidth assignment is a *global* condition, because it depends on all the servers in the system, whereas the feedback controller only performs a *local* adaptation, since it operates on individual tasks and does not consider any schedulability or feasibility issue.[1] Hence, the bandwidth compression is a global mechanism necessary to preserve the feasibility of a bandwidth assignment, and is performed by the *compression function* $\hat{U}' = h(\hat{U})$. The compression function is a function $h : R^n \rightarrow R^n$ that transforms an infeasible bandwidth assignment into a feasible one; in practice, if $\hat{U}' = h(\hat{U})$, then $\sum_i U^{s\prime}_i \leq U_{lub}$. In particular, the ith component of vector $U^{s\prime}_i$ is computed as

$$U^{s\prime}_i = \begin{cases} U^s_i & \text{if } \sum_i U^s_i \leq U_{lub} \\ h_i(\hat{U}) & \text{otherwise} \end{cases}$$

where $h_i()$ is the ith component of vector h. A simple solution to perform such a bandwidth compression is to scale tasks' utilizations in a proportional way:

$$U^{s\prime}_i = U^s_i s_i$$

being s_i the scaling factor. Since the compression must be done according to the tasks' weights, s_i must be proportional to w_i: $s_i = w_i M$. For the sake of generality, the sum of the reserved bandwidths is set to $U^{max} \leq U_{lub}$, hence imposing $\sum_j U^{s\prime}_j = U^{max}$ we have:

$$\sum_j U^{s\prime}_j = U^{max} \Rightarrow \sum_j U^s_j s_j = U^{max} \Rightarrow \sum_j U^s_j w_j M = U^{max} \Rightarrow$$

$$M \sum_j U^s_j w_j = U^{max} \Rightarrow M = \frac{U^{max}}{\sum_j U^s_j w_j}$$

[1] Each feedback controller is not aware of all the other reserved tasks in the system.

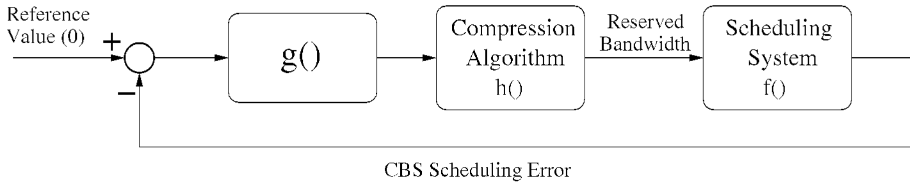


Figure 8.3 Description of an Adaptive Reservation.

Hence,

$$U_i^{s\prime} = U_i^s w_i \frac{U^{max}}{\sum_j U_j^s w_j} \tag{8.1}$$

This simple method can be slightly modified to guarantee a minimum bandwidth U^{min} to each task.

The closed loop control used to adjust the reserved bandwidth is shown in Figure 8.3. When implementing an Adaptive Reservation abstraction, it is important to design the feedback function so that the resulting adaptive scheduler is able to assign the correct amount of resources to each task (when possible) in a short time and with an acceptable accuracy. Since control theory has already been proven to be a valid tool for designing feedback schedulers (see FC-EDF [SLS99, LSTS99, LSA+00]), it is interesting to use it for evaluating the performance of an adaptive reservation. Using control theory terminology, the closed loop system must be stable, and the response time, overshoot, and steady-state error must be compliant with some specifications.

Since a proper feedback scheme providing the required characteristics can be designed only based on an accurate model of the system, a precise model of a reservation scheduler has been developed [APLW02]. Such a model is highly non-linear, and contains some quantization effects (given by the presence of a ceiling operator in the model), hence it is very difficult to control. However, by applying some approximations it is possible to linearize the model, and to design a feedback controller that is able to stabilize the closed-loop system. As for FC-EDF^2, the resulting controller is based on a *switching dynamic* (two different controllers are designed, and the one to be used is dynamically chosen based on the value of the controlled variable ϵ). The classical "pole-placement" technique can be used to synthesize the two controllers; in this way it is possible to comply with requirements on the closed loop dynamics (i.e., the evolution of the scheduler under the action of a feedback controller).

Figure 8.4 Dynamic system representing a linearized reservation with a feedback mechanism.

To achieve the control goals, a feedback controller $G(Z)$ is used as in Figure 8.4: $U(Z) = -G(Z)\epsilon(Z)$. Since the open loop dynamics is $\epsilon(Z) = F_u(Z)U(Z) + F_c(Z)C(Z)$, the closed loop dynamics is described by the transfer function between $C(Z)$ and $\epsilon(Z)$:

$$\epsilon(Z) = F(Z)C(Z) = \frac{F_c(Z)}{1 + G(Z)F_u(Z)}C(Z) \tag{8.2}$$

The simplicity of the system (whose dynamic equations are similar to those of a tank) suggested the use of a Proportional Integral (PI) controller (this is a difference with respect to FC-EDF^2, which needs a PID). A PI controller is described by:

$$U_k = C_P(-\epsilon_k) + C_I \sum_{j=0}^{k-1}(-\epsilon_{k-1})$$

where C_P and C_I are the coefficients of the proportional and integral actions, respectively. By manipulating the previous equation, a PI can also be described as:

$$U_k = U_{k-1} + \alpha(-\epsilon_k) + \beta(-\epsilon_{k-1}).$$

where $\alpha = C_P$ and $\beta = C_I - C_P$.

According to control theory, the closed-loop system is stable if the poles Z_i of the closed-loop system (i.e. the zeros of the denominator of the closed-loop transfer function) have norm strictly lower than 1: $||Z_i|| < 1$. Moreover, the decay rate ρ (i.e., the "speed" with which the closed-loop system returns to a stable state) is given by the maximum norm of the poles. Observe that the use of the PI controller enables the choice of the two closed-loop poles.

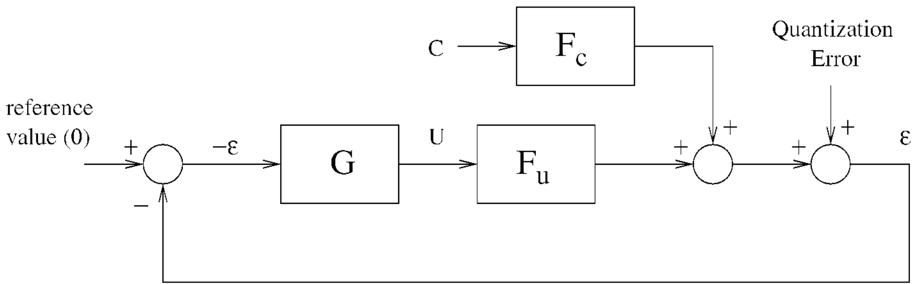


Figure 8.5 Dynamic system representing a linearized CBS with a feedback mechanism.

Finally, note that the linearization performed to use the Z transform introduced a *quantization error* due to the approximation of a ceiling. This quantization error can be taken into account (as shown in [APLW02]), and its effects can be bounded. A model of the system accounting for the quantization error is shown in Figure 8.5. It has been proven that the quantization error has no effects on ϵ, but only causes an overestimation in the reserved bandwidth. If $P^s < T$, the maximum overestimation is such that

$$\frac{\bar{c}}{T} \leq \tilde{U} \leq \frac{\bar{c}}{T - P^s},$$

where $\bar{c}$ is the actual execution time, T is the task period, and $\tilde{U}$ is the estimated bandwidth. The previous equation shows that by reducing the server period P^s it is possible to increase the accuracy of the feedback (i.e., the accuracy of the bandwidth estimation).

8.3 APPLICATION LEVEL ADAPTATION

The feedback scheduling techniques shown in the previous sections help to cope with unpredictability in the system workload by adapting the behavior of the scheduler. When the system is overloaded, the scheduler can either decrease the load by rejecting some tasks (as done by FC-EDF), or can decrease the QoS perceived by some tasks (as done by the compression mechanism in Adaptive Reservations). In any case, all the decisions about adaptation are taken by the scheduler. However, the scheduler is not aware of applications' semantics and cannot select the best strategy for decreasing the workload. For example, consider a task whose reserved bandwidth is not sufficient for executing it with the required QoS level. Such a situation can be handled in two ways, which may also be combined together:

1. The reserved bandwidth U_i^s can be increased to satisfy the task QoS requirements;
2. The amount of CPU time demanded by the task can be decreased to fit with the reserved bandwidth.

The first strategy is the one used by Adaptive Reservations, where the scheduler, or a QoS manager, adapts the reserved bandwidths based on tasks' specifications. The second strategy seems to be similar to the one used by FC-EDF, in which a service level controller can switch the service level of each task. However, it presents a fundamental difference, because each application *explicitly* scales down its own QoS (and consequently its resource requests) to remove the overload condition and make the system schedulable. Since the centralized scheduler is unaware of such a QoS adaptation, this second mechanism is referred to as *application-level adaptation*. Each application has the responsibility to cope with its own overloads and can scale down its QoS in different ways, because it is the only entity that knows how to perform such a QoS adaptation, without any help from the scheduler.

Several approaches for performing such an application-level adaptation have been proposed in the literature and are well known in the multimedia community, ranging from enlarging task periods to skipping some task instances. For example, DQM [BNBM98] is a feedback-based QoS manager which does not require any support from the operating system. DQM is a middleware solution aimed at supporting soft real-time applications in a conventional OS (Linux). Application execution levels are changed based on resource usage, by monitoring the benefit directly experienced by the application, and based on the system load estimated by the middleware itself. A similar approach is adopted by FARA [RSY98], where a resource allocator monitors the resource usage and coordinates the adaptation. This solution addresses the problem of integrating QoS adaptation with real-time techniques, however it relies on the a-priori knowledge about the resources required by each application in each operating mode. The QRAM model, presented in Section 7.1, can also be used for performing QoS adaptation at the application level.

In other solutions [Apa98], each application QoS can be scaled by a global QoS manager in order to better respond to the user needs. The adaptation is based on specific *modes of operation* provided by each application, but it is still performed on a global basis.

Note that application-level adaptation is mainly useful when the system is permanently overloaded. For example, if the sum of all CPU utilizations requested by the applications is less than the maximum available bandwidth (1 for CBS/EDF), then the adaptive reservation mechanism is able to find a feasible bandwidth assignment

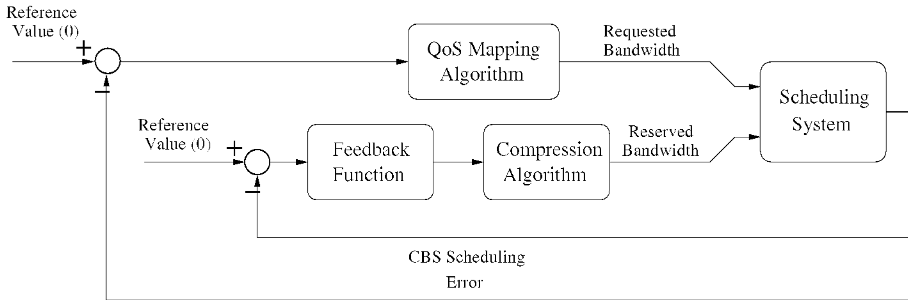


Figure 8.6 Hierarchical adaptation achieved by two feedback loops.

$\hat{U} = (U_1^s, \dots U_n^s)$ such that each task will receive enough CPU time. In this case, application-level adaptation is not needed. On the other hand, if the sum of CPU utilizations requested by the applications is continuously greater than the available CPU bandwidth, then the least important tasks (i.e., the tasks having smaller weights w_i) can suffer from local overloads. Indeed, the goal of the global adaptive reservation mechanism is to isolate task overloads in the least important tasks, independently of their requirements and periods[2]. In this case, an overloaded task can use application-level adaptation to scale down its requirements and resolve the overload condition *reaching a lower QoS level in a controlled fashion*, otherwise the QoS degradation would be unpredictable.

When application-level adaptation is used together with a global feedback scheduler, such as the one implemented in the adaptive reservation approach, there are two *orthogonal* forms of adaptation:

- the scheduler adaptation, realized by an active entity having a global system visibility, such as a QoS manager or the scheduler itself;
- the application-level QoS adaptation, performed by each single application.

Such an integrated approach, referred to as *hierarchical adaptation* [AB01a], presents the advantages of both methods, allowing applications to scale their QoS level when the closed-loop scheduling mechanism cannot serve them properly. Hierarchical adaptation introduces a new level of feedback, as shown in Figure 8.6. The inner loop implements a global feedback scheduling mechanism (in this case, Adaptive Reservations), while the outer loop controls the amount of CPU time requested by the application, using a local adaptation method.

[2] Remember that adaptive reservations separate task importance from timing requirements.

One of the major problems with this kind of hierarchy is that it can easily reach unstable conditions. For example, consider two tasks τ_1 and τ_2: by reacting to a transient overload, the global adaptive reservation mechanism could decrease U_1^s; if τ_1 reacts immediately by decreasing its QoS, when the transient overload is over the bandwidth adaptation mechanism could increase U_2^s. In this way, τ_2 would increase its QoS level, stealing bandwidth from τ_1, thus preventing it to recover its initial QoS level.

To prevent such a behavior, the application-level adaptation has to act slower than the scheduler adaptation, so that QoS is changed only when the overload condition is long (in most cases, the QoS is not scaled in response to transient overloads).

8.4 WORKLOAD ESTIMATORS

Some of the feedback techniques presented in the previous sections assume the knowledge of tasks' execution times to derive the computational demand and assign the reserved bandwidth to each task. When such values are not known in advance, or are highly variable with time, they can be estimated on line by execution time estimators explicitly monitoring tasks' execution. To do that, the real-time kernel must provide a specific support for monitoring the execution time actually consumed by each job. Traditional Unix systems provide the `getrusage()` system call for reading the amount of time consumed by a process, but it is not precise enough, since it does resource accounting at a tick granularity.

Once an on-line estimation of task execution times is available, the estimated value c_i can be used as an observed value, and some kind of workload adaptation mechanism can be used as an actuator. For example, the elastic model presented in Section 2.7.1 can be combined with the on-line execution time estimation to dynamically adjust the tasks' periods [BA02a].

When the system workload estimated through such an explicit monitoring is found to be greater than a predefined threshold, the adaptation mechanism can be used to find a feasible tasks' configuration. This approach can be combined with CPU reservations [FNT95, Nak98c] to enforce the maximum execution time to each task. With this technique, the amount of time reserved to each task in each period must still be defined based on some off-line estimation, whereas the period is dynamically adapted based on the actual execution time. If the reserved budget is too small, the task will experience large overruns that will cause the algorithm to increase its period too much. On the other hand, if the estimation is too big, the periods are not optimized, the reserved budget is never used completely, and the system is underutilized.

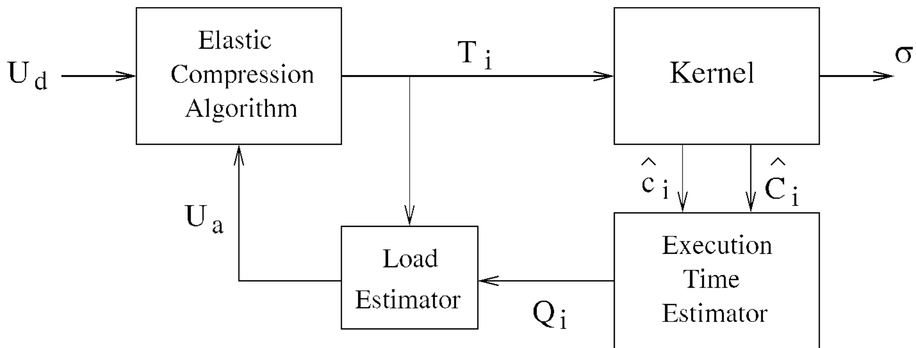


Figure 8.7 Feedback-based architecture for elastic rate adaptation.

If resource reservation is not used, the elastic approach (see Section 2.7.1) provides a powerful and flexible methodology for adapting tasks' rates to different working conditions. However, it strongly relies on the knowledge of the worst-case execution times (WCETs). When WCETs are not precisely estimated, the elastic compression algorithm will lead to wrong period assignment. In particular, if WCETs are underestimated the compressed tasks may start missing deadlines, whereas, if WCETs are overestimated, the algorithm will cause a waste of resources, as well as a performance degradation.

To overcome this problem, on-line estimates of tasks' execution times can be used as feedback for achieving workload adaptation. Such estimates are derived by a runtime monitoring mechanism embedded in the kernel. When a task starts its execution, it is created at its minimum rate, and, at the end of each period, a runtime monitoring mechanism updates the mean execution time $\hat{c}_i$ and the maximum execution time $\hat{C}_i$. Figure 8.7 shows the architecture used to perform rate adaptation. The two values $\hat{C}_i$ and $\hat{c}_i$ derived by the monitoring mechanism are used to compute an execution time estimate Q_i, used by the load estimator to compute the actual workload $U_a = \sum \frac{Q_i}{T_i}$. Such a value is then used by the elastic algorithm (periodically invoked with a period P) to adapt tasks' rates. Thus, the objective of the global control loop is to maintain the estimated actual load U_a as close as possible to a desired value U_d.

The advantage of using the elastic compression algorithm is that rate variations can be controlled individually for each task by means of elastic coefficients, whose values can be set to be inversely proportional to tasks' importance. Using this approach, the application is automatically adapted to the actual computational power of the hardware platform. The effectiveness of the adaptation depends on whether tasks' utilizations are computed based on worst-case ($\hat{C}_i$) or average-case ($\hat{c}_i$) estimates. If the $\hat{C}_i$ estimate

is used to compute tasks' utilizations for the elastic algorithm, tasks are assigned larger periods and the number of deadline misses quickly reduces to zero. However, this solution can cause a waste of resources, since tasks seldom experience their worst case simultaneously.

To increase efficiency, a more optimistic estimation can be used to exploit system resources. The resulting approach is a trade off between "rigid" reservation systems, in which each task is assigned a fixed amount of resources and cannot demand more (even though other tasks require less than the reserved amount), and completely unprotected systems, such as those based on bare EDF or RM. In this sense, this approach is similar to the "Bandwidth Sharing Server" (BSS) presented in Section 4.2.2, where tasks belonging to the same application can share the same resources. In the BSS case, however, the fraction of CPU bandwidth that can be exchanged among tasks belonging to a particular application cannot be controlled, whereas in the elastic model it depends on the elastic coefficients.

To prevent the number of deadline misses per time unit to increase indefinitely, the execution time estimate used to perform the elastic compression must be greater than the mean execution time, so a value between $\hat{c}_i$ and $\hat{C}_i$ is typically acceptable. Hence, the elastic compression algorithm is invoked using a value

$$Q_i = \hat{c}_i + k(\hat{C}_i - \hat{c}_i)$$

where $k \in [0, 1]$ is referred to as the *guarantee factor*. Then, the utilization factor $\hat{U}_i$ is computed as

$$\hat{U}_i = \frac{Q_i}{T_i}$$

and the actual load U_a is estimated as

$$U_a = \sum \hat{U}_i.$$

It is worth noting that, if $k = 1$, the elastic algorithm results to be based on WCET estimations, so only few deadlines can be missed when the estimated WCET $\hat{C}_i$ is smaller than the real one. In general, if no information about execution times is provided, the first $\hat{C}_i$ values will be underestimated and it will cause some missed deadline in the task startup time.

A smaller value of k allows increasing the actual system utilization at the cost of an increased number of possible deadline misses (remember that a deadline is missed when many tasks require a long execution at the same time). A value of $k = 0$ allows maximum efficiency, but is the limit under which the system overload becomes permanent.

The estimation method described above allows using the feedback mechanism either when no a-priori information about execution times is provided, or when an approximated estimation of the mean or maximum execution time is known. In practice, the mean execution time estimation is computed iteratively (that is, $\hat{c}_i$ is periodically updated based on the last execution times experienced by the task) starting from an initial value c^0. If nothing is known about the task parameters, an arbitrary value can be assumed for c^0; if, on the other hand, an approximate estimation of the execution time is known in advance, it can be used as c^0, reducing the initial transient during which $\hat{c}_i$ converges to a reasonable estimation and increasing the speed at which the periods converge to a stable value.

As a final observation, it is worth noting that this approach can be successfully applied to task sets characterized by variable execution times, allowing periods to vary according to execution times variations. If, instead, the proposed mechanism is applied to tasks characterized by fixed execution times, it allows adapting task periods to the unknown execution times without any deadline miss. In this case, the mean execution time is equal to the WCET, but if the starting estimation c^0 is different from the actual value, the mean execution time estimation needs some time to converge to the correct value. In this transient, if the guarantee factor is less than 1, there could be missed deadlines. Hence, in the case of fixed execution times, a guarantee factor $k = 1$ is more appropriate.

If some additional information about the application is provided, the explicit workload estimation can be combined with an "ad hoc" adaptation mechanism, instead of using a generic mechanism such as the elastic model. For example, if the real-time tasks implement a control algorithm, a feedback scheduling architecture for control tasks can be designed to optimize some control performance metric [Cer03, ACr02, CE00].

Such a feedback scheduler attempts to keep the CPU at a high utilization level while avoiding overload and distributing the available computing resources among the control tasks. It is composed by 3 modules: a *workload estimator*, a *resource allocator*, and a *proactive action*. The resource allocator assigns periods to tasks so that the total estimated utilization is controlled to a *set point* U_d (typically less than U_{lub}), and a cost function (see Section 7.3) is minimized. The proactive action is based on the fact that in the proposed model (see [ACr02]) each controller can work in different modes, and each mode is characterized by a different profile of execution times. The controller switches between different modes depending on the controlled system state, and can signal the feedback scheduler in advance when a mode switch is going to happen. Such a feedforward information can be used by the feedback scheduler to implement the proactive action, reacting in advance with respect to changes in the execution times.

As an example, the feedback scheduler can be implemented as a periodic task (similarly to [BA02a]) that periodically reads the monitored mean execution times $\hat{c}_i$ and computes the estimated utilization $U = \sum \hat{c}_i/T_i$, where T_i is the period assigned to task τ_i (i.e., the sampling period of the ith controller). Each mode of each control task τ_i is characterized by a *nominal period* T_i^{nom}, and the feedback scheduler tries to assign periods to the control tasks starting from the nominal periods: in other words, if $\sum \hat{c}_i/T_i^{nom} < U_d$, then all tasks are allowed to execute at their nominal periods. If, on the other hand, $\sum \hat{c}_i/T_i^{nom} > U_d$, then the feedback scheduler uses its resource allocator to assign new periods T_i to tasks so that $U = U_d$. As said, the optimal periods T_i can be derived by using control theoretical arguments to minimize a cost function $J(T_i)$ associated with the ith controller. The PLI presented in Section 7.3 can be used as a cost function, so that standard techniques can be adopted to optimize the tasks' periods [SLSS97]. However, some simulations show that a simple linear rescaling of the tasks' periods is able to achieve good results [Cer03]; this is due to the fact that under certain assumptions the linear rescaling can be proven to be optimal with respect to the overall control performance.

Finally, the proactive action is used by the feedback scheduler in 3 different ways:

- to execute more frequently, if needed. Usually, the feedback scheduler is activated periodically, but if a controller is going to switch its operation mode, then the feedback scheduler can be activated immediately to react more rapidly to the workload variation caused by the mode switch;
- to keep track of the operating mode of each controller, in order to assign more suitable sampling periods;
- to run separate workload estimators for each mode of each task: since each task τ_i is characterized by a different execution time profile for each operation mode m, maintaining different execution time estimations $\hat{c}_i^m$ can actually help improving the accuracy of the estimation.

9

STOCHASTIC SCHEDULING

In this chapter we address the problem of performing a probabilistic schedulability analysis of real-time task sets, with the aim of providing a relaxed form of guarantee for systems with highly variable execution behavior. The objective of the analysis is to derive a probability for each task to meet its deadline or, in general, to complete its execution within a given interval of time.

As already explained in the previous chapters, traditional hard real-time guarantee (based on worst-case scenarios) is very pessimistic, because tasks' execution times and interarrival times typically present a high variability, where the worst-case situation is very rare. The consequence of such a pessimistic analysis is that system resources are underutilized most of the time, with a significant impact on the overall cost required to develop the system.

Consider, for example, the traditional utilization-based admission test $\sum_i U_i \leq U^{lub}$. If the mean execution times $\overline{c_i}$ are much smaller than the worst-case values $C_i = \max_j\{c_{i,j}\}$ (or the mean interarrival times are much larger than the worst-case ones), a guarantee test based on worst-case utilizations $U_i = C_i/T_i$ would lead to a significant resource waste. On the other hand, performing the admission test using the mean utilizations would increase efficiency, but would not provide any guarantee about the respected deadlines. This problem can be addressed by adopting a probabilistic framework for characterizing a soft real-time system in a more rigorous way. Such a probabilistic analysis can be performed using different approaches:

1. Classical real-time analysis of fixed or dynamic priority systems can be extended to cope with statistically distributed execution times (and/or interarrival times).
2. Traditional queueing theory (typically used for computing the average response times of tasks characterized by random arrival and service times) can be extended to cope with priority schedulers (such as RM and EDF).

3. Novel scheduling algorithms can be developed for simplifying the use of stochastic analysis.
4. The temporal protection property provided by some scheduling algorithms (see Chapter 3) can be used to perform a probabilistic guarantee of each task, individually and independently of the others.

Each of these approaches is discussed in detail in the rest of this chapter.

9.1 BACKGROUND AND DEFINITIONS

Stochastic analysis and, in particular, probabilistic analysis of real-time scheduling algorithms require advanced mathematical tools that sometime are not so easy to understand. To make this chapter more readable and understandable, we decided to present a simplified description of the various stochastic analysis techniques, at the cost of making the description less rigorous. Interested readers are invited to consult the original papers (cited in the chapter) to understand the whole mathematics used there.

This section briefly recalls the most important definitions and concepts needed to understand the chapter, and introduces some of the most important mathematical tools used to deal with probabilities.

Informally speaking, the final goal of a stochastic guarantee is to compute the probability to miss a deadline. To perform a formal analysis of such a probability, we need to define the concept of *random variable* and specify the probability of some events (such as "job $\tau_{i,j}$ has execution time $c_{i,j}$") through the *probability distribution function* and the *cumulative distribution function*.

Definition 9.1 *A* Random Variable *is a variable X defined in a set $\mathcal{D}$, which can randomly assume values $x \in \mathcal{D}$ with probabilities $P\{X = x\}$.*

Definition 9.2 *If X is a random variable, the* Cumulative Distribution Function (CDF) *of X is defined as $C_X(x) = P\{X \leq x\}$.*

Definition 9.3 *If X is a random variable defined in $\mathcal{D}$, and $C_X(x)$ is its CDF, the* Probability Distribution Function (PDF) *$F_X(x)$ of X is a function $\mathcal{D} \rightarrow \mathcal{R}$ defined as $F_X(x) = C_X(x+1) - C_X(x)$, if X is a discrete random variable, or $F_X(x) = dC_X(x)/dx$, if X is a continuous random variable.*

For discrete random variables (i.e., if $\mathcal{D} = \mathcal{N}$), the PDF $F_X(x)$ gives the probability that the variable assumes a random value x: $F_X(x) = P\{X = x\}$. By definition, if C is a discrete random variable and $U(c)$ is its PDF, the corresponding CDF can be computed as $\sum_{j=0}^{c} U(c)$. If the random variable is continuous, the sum must be replaced with an integral.

Note that the execution time $c_{i,j}$ and the interarrival time $r_{i,j+1} - r_{i,j}$ of a job $\tau_{i,j}$ can be considered as two *discrete* random variables, defined in $\mathcal{N}$, and their PDFs and CDFs are discrete functions $\mathcal{N} \to \mathcal{R}$.

To simplify the notation, the PDF and CDF of the random variable X, will not be denoted with $F_X()$ and $C_X()$, but simply with $X(x)$, specifying the meaning in the text.

If X and Y are random variables, then $Z = X + Y$ is also a random variable, and its PDF $Z(z)$ can be computed based on the PDFs $X(x)$ and $Y(y)$ of the two original variables. In particular, $Z(z)$ is given by the *convolution* of $X(x)$ and $Y(y)$. The convolution of two discrete functions is defined as

$$Z(z) = X(x) \otimes Y(y) = \sum_{k=0}^{\infty} X(k)Y(z-k).$$

The definition for continuous functions is similar, using an integral instead of a sum. The convolution is frequently used in the probabilistic analysis of scheduling algorithms to compute the distribution of the sum of tasks' execution times, or similar quantities.

If the observed quantities evolve with time, the concept of random variable is not sufficient to describe them, and we must introduce the concept of a *stochastic process*.

Definition 9.4 *A* stochastic process *is a time-dependent random variable. Depending on the time dominion, it can be a* discrete time process X_i, *or a* continuous time process $X(t)$. *In particular, a discrete time process can be seen as a sequence of random variables.*

According to the previous definition, the sequence $c_{i,j}$ of job execution times of task τ_i is a stochastic process. However, note that the probability $P\{c_{i,j} = c\}$ of having a job execution time equal to c does not depend on the job index j, hence the execution times of a task can be described by a simple PDF $U(c)$. A stochastic process having such a property is said to be a *time invariant* process. Another important class of processes is represented by *Markov processes*:

Definition 9.5 *A stochastic process X_i is a Markov process if the value X_i only depends on X_{i-1}; that is, it does not depend on $X_{i-2}, X_{i-3}, \ldots, X_1$ or on time i.*

Since discrete functions are simpler to work with, time instants are often considered as integers, so that a task τ_i can be described by a pair of time invariant stochastic processes $(U(c), V(t))$, with $U(c) = P\{c_{i,j} = c\}$ and $V(t) = P\{r_{i,j+1} - r_{i,j} = t\}$. The only case in which execution times and arrival times are considered to be real numbers is in the real-time queueing theory (RTQT)(see Section 9.3), because traditional queueing theory is based on continuous time, and its real-time extension did not change this assumption.

Most of the algorithms related to probabilistic analysis are based on a simplified task model, in which the interarrival times are constant (i.e., $V(t) = 1$ if $t = T_i$, and $V(t) = 0$ otherwise). This is the so called *semiperiodic task model* [TDS+95], in which a task τ_i is described by the tuple $(U(c), T_i)$. In other words, the semiperiodic task model simply extends the traditional Liu & Layland periodic task model by replacing the worst-case execution times with stochastically distributed execution times.

Finally, we need some way for indicating the "average value" of a random variable: using a more rigorous formalism, this is called the *expectation*, or the *expected value*.

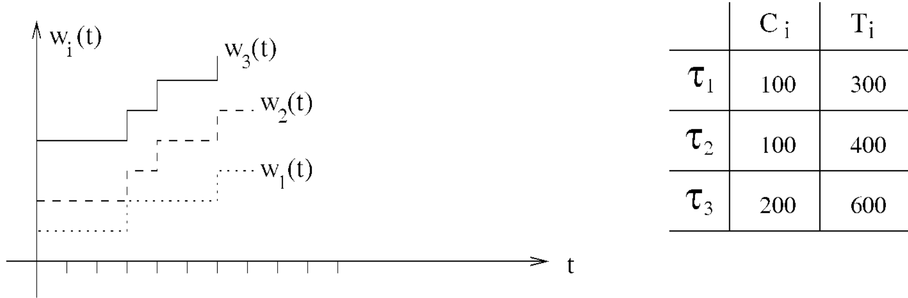
Definition 9.6 *The* expectation $E[X]$ *of a discrete random variable X described by a PDF $X(x)$ is defined as*

$$E[X] = \sum_{x=0}^{\infty} X(x)x$$

The definition for continuous random variables is similar, using an integral instead of the sum.

9.2 STATISTICAL ANALYSIS OF CLASSICAL ALGORITHMS

The hard schedulability analysis of classical real-time scheduling algorithms (based on fixed or dynamic priorities) can be performed using the Time Demand Analysis (TDA) approach. TDA consists in computing the amount of time demanded by each real-time task, expressed by the *time demand function* $w_j(t)$, and checking whether such a time is available. Since computing $w_j(t)$ for all time intervals is too onerous, the worst-case situation is considered. As it is well known from the real-time theory,



	C_i	T_i
τ_1	100	300
τ_2	100	400
τ_3	200	600

Figure 9.1 An example of time demand function for three tasks $\{\tau_1 = (100, 300), \tau_2 = (100, 400), \tau_3 = (200, 600)$ scheduled by RM.

in fixed priority systems, a job $\tau_{j,k}$ experiences its worst-case response time when it is released simultaneously with all the jobs with higher priority. Such a release time is called the *critical instant*. If job $\tau_{j,k}$ is released at the critical instant, $w_j(t)$ is defined as the maximum amount of time demanded between $r_{j,k}$ and t by $\tau_{j,k}$ and by all higher priority jobs finishing before $f_{j,k}$.

As shown in Figure 9.1, the time demand function $w_j(t)$ is a step function, increasing by C_i every time a higher priority job $\tau_{i,h}$ is released. According to the TDA, if $\exists t : r_{j,k} \le t \le d_{j,k}$ and $w_{j,k} \le t - r_{j,k}$, then τ_j and all higher priority tasks will not miss their deadlines.

This approach can be extended by considering probabilistic distributions for the execution times instead of worst-case values. This is done, for example, in the Probabilistic Time Demand Analysis (PTDA) [TDS+95] or in the Stochastic Time Demand Analysis (STDA) [GL99]. Both PTDA and STDA are based on the previously introduced *semiperiodic* task model. The analysis is performed by considering a demanded time distribution $W_j(t) = P\{w_j(t) \le t\}$ instead of the time demand function $w_j(t)$.

PTDA performs a simple analysis of semiperiodic tasks, assuming $D_i \le T_i$ (hence, $d_{i,j} \le r_{i,j+1}$). The computation of $w_j(t)$ (and hence of $W_j(t)$) is in general not easy, but since the analysis is restricted to (semi)periodic tasks, the following bound is valid:

$$w_j(t) \le c_{j,k} + \sum_{i=1}^{j-1} \sum_{l=1}^{t/T_i} c_{i,l}. \qquad (9.1)$$

According to the previous definitions, a lower bound of the probability to respect a deadline $d_{j,k}$ can be found:

$$P\{f_{j,k} \le d_{j,k}\} \ge \max_{t \in \mathcal{E}} W_j(t),$$

where $\mathcal{E} = \{d_{j,k}, r_{j,k} + T_i, r_{j,k} + 2T_i, \ldots, r_{j,k} + \lfloor D_j/T_i \rfloor \ : \ p_i \geq p_j\}$, where p_i is the priority of task τ_i. The PDF $W_j(t)$ is computed by using a straightforward extension of Equation 9.1: the PDFs $U_i(c)$ are used instead of the deterministic execution times $c_{i,j}$, and the *convolution* operation is used instead of the sum.

The STDA analysis [GL99] extends PTDA to the case of $D_i > T_i$, and fixes some inaccuracies in the analysis (in fact, PTDA is not very accurate when the average utilization approaches 1).

The key point in STDA is the concept of *level-j busy interval*: if tasks can miss their deadlines (as it happens when a stochastic guarantee is used), analyzing the first job after the critical instant is not enough, because the successive jobs can be affected by its behavior. A level-j busy interval is an interval of time beginning when a job $\tau_{j,k}$ or a higher priority job is released and immediately prior to the instant no job in those tasks is ready for execution. The interval ends at the first time instant t at which all jobs of τ_j and higher priority tasks released before t have completed. If a level-j busy interval begins at a critical instant (i.e., it begins with the release of a job τ_j and all tasks $\tau_i : p_i \geq p_j$), it is called *in-phase level-*Φ_j *busy interval.*

According to the previous observation, the analysis cannot be stopped at the first job $\tau_{j,k}$ after a critical instant, but must cover the whole in-phase level-Φ_j busy interval that follows the critical instant. No job after the end of the interval will be affected by the previous history of the system, as already known in real-time theory [Leh90]. Since the time demand function and the demanded time distribution are computed on more than one job, an additional index must be added, so they will be denoted as $w_{j,k}(t)$ and $W_{j,k}(t)$, respectively.

As the demanded time distribution $W_{j,i}(t)$ must be computed on the whole in-phase level-Φ_j busy interval, computing the length of such an interval is crucial for STDA. Such a computation can be easily performed by considering that a job $\tau_{j,k}$ terminates when $w_{j,k}(t) = t$, hence $W_{j,j}(t)$ is also the probability that job $\tau_{j,k}$ finishes within a time t. In other words, the demanded time distribution coincides with the response time distribution.

For the first job of a level-Φ_j busy interval, $W_{j,k}(t)$ can be easily computed as in PTDA, since

$$W_{j,k}(t) = P\{w_{j,k}(t) \leq t\}.$$

The response time (demanded time) distribution for the successive jobs in a level-Φ_j busy interval is computed by conditioning the probability to the previous workload:

$$W_{j,k}(t) = P\{w_{j,k}(t) \leq t | w_{j,k-1}(r_{j,k}) > r_{i,j}\}.$$

Hence, $W_{j,k}$ is computed by convolving the execution time distribution of the task with the distribution of the backlog obtained by conditioning. This iterative computation must be repeated until the end of the busy interval. To take into account the effects of higher priority tasks $\tau_i : p_i \geq p_j$, the time interval $(r_{j,k}, f_{j,k})$ is divided into sub-intervals delimited by the releases of higher priority jobs $\tau_{i,l} : r_{i,l} \in (r_{j,k}, f_{j,k})$, and the response time probability in each interval is conditioned to the workload in the previous one. Finally, the probability of a $\tau_{j,k}$ to complete within its deadline is given by $W_{j,k}(D_j)$.

If there is a single task per priority level, the length of a level-Φ_j busy interval can be computed by checking whether $\tau_{j,k}$ finishes before the release of $\tau_{j,k+1}$, i.e., whether $f_{j,k} < r_{j,k+1}$. Therefore, the computation of the response time distribution $W_{j,k}(t)$ can be stopped when $P\{w_{j,k}(r_{j,k+1}) \leq r_{j,k+1}\} = 1.0$.

In a probabilistic framework, it is important to observe that the simultaneous release of all the tasks does not represent the worst-case situation (as for the deterministic analysis). Indeed, this turns out to be true only if the maximum system utilization is less than 1. Since the worst-case release configuration is still not clear, various task sets with different phases have been simulated [GL99]. Experimental results show that, in general, the in-phase case seems to give the worst-case response time CDF (although the precise relation among the PDFs it is still not clear).

The case in which the maximum system workload is greater than one can be analyzed in a rigorous way only by applying a different approach that does not use the time demand analysis [DGK+02]. Such an alternative approach is based on computing the finishing time distribution based on the concept of *P-level backlog*. The P-level backlog observed at time t is defined as the sum of the remaining execution times of all the jobs having priorities higher than P that are not completed at time t.

To find a mathematical formulation of the problem that can be easily solved, the analysis must be extended to an hyperperiod: if B_k is the P-level backlog for the lowest priority at the beginning of the kth hyperperiod, then

$$P\{B_k = y\} = \sum_{x=0}^{\infty} P\{B_{k-1} = x\} P\{B_k = y | B_{k-1} = x\}. \tag{9.2}$$

Since task arrivals are periodic, the same arrival pattern is repeated in each hyperperiod, hence $P\{B_k = y | B_{k-1} = x\}$ does not depend on k. In other words, the backlog process is a Markov chain, and Equation 9.2 can be expressed as

$$b_k = M b_{k-1}$$

where $b_k = (P\{B_k = 0\}, P\{B_k = 1\}, \ldots)^T$ and M is a matrix composed by elements $m_{x,y} = P\{B_k = y | B_{k-1} = x\}$.

To derive the elements $m_{x,y}$ of the matrix M, we need to compute the P-level backlog at time t' as a function of the P-level backlog at time t. The PDF of the P-level backlog immediately after the release of a higher priority job $\tau_{i,j}$ (having priority $p_i > P$) can be computed by convolving the PDF of the P-level backlog with the PDF $U_i(c)$ of τ_i execution times. If, on the other hand, no job arrives in the interval (t, t'), then the PDF of the P-level backlog at time t' can be computed by shifting the PDF of the P-level backlog at time t to the left $(t' - t)$ units of time and by accumulating in the origin all the probability values of the PDF that would be shifted to negative time values. Therefore, since the pattern of job arrivals in the hyperperiod is known, the procedure shown above can be used to compute the P-level backlog during all the hyperperiod, and at the end of the hyperperiod, by computing $m_{x,y}$. Obviously, the result depends on the scheduling algorithm (since jobs priorities depend on the scheduling algorithm). Solutions have been proposed for computing M, both in the fixed priority and dynamic priority (EDF) case [DGK$^+$02].

If the maximum system load $\sum_i C_i/T_i$ is less than or equal to 1 and a RM scheduler is used, the computation shown above is equivalent to STDA. If the average system load $\sum_i E[U_i(c)]/T_i$ is less than or equal to 1, then the process $b_k = Mb_{k-1}$ has a stationary solution; that is, after a transient b_k, it converges to a probability distribution $b : b = Mb$. This solution can be found either by using an exact computation based on some regularity in the matrix M, or by truncating M and b to a reasonable dimension and by using some numerical technique to solve the resulting eigenvector problem. See Section 9.5 for more details about the approximation process.

Starting from the P-level backlog, it is possible to compute the job-level backlog, defined as the backlog due to jobs with priorities higher than or equal to the priority of a specified job. Under a fixed priority scheme, the job-level backlog is equal to the P-level backlog, where P is the priority of the specified job. Under EDF, the computation is a little bit different, but it can still be performed. Since the job-level backlog can be used to compute the job response time as $f_{i,j} = x_{i,j} + c_{i,j} + \sum_{\tau_{k,h} \in H} c_{k,h}$ (where $x_{i,j}$ is the job-level backlog of job $\tau_{i,j}$, and H is the set of jobs that may preempt $\tau_{i,j}$), the PDF of the job response time can be obtained by convolving the job-level backlog with the PDFs of the execution times of τ_i and of the jobs in H.

The computation of the response time PDF is performed in two stages: in the first stage, the convolution $U_i'(c)$ between $U_i(c)$ and the PDF of the job-level backlog is performed, and, in the second stage, the effects of preemptions from jobs $\tau_{k,h} \in H$ are computed. Since a job $\tau_{k,h}$ cannot preempt $\tau_{i,j}$ before $r_{k,h}$, its effects are computed

by splitting $U_i'(c)$ in two PDFs, $U_i''(c)$ and $U_i'''(c)$, with

$$\begin{array}{lcll} U_i''(c) & = & U_i'(c) & \text{if } c < r_{k,h} \\ U_i''(c) & = & 0 & \text{if } c \geq r_{k,h} \\ U_i'''(c) & = & 0 & \text{if } c < r_{k,h} \\ U_i'''(c) & = & U_i'(c) & \text{if } c \geq r_{k,h}. \end{array}$$

Then $U_k(c)$ is convolved with $U_i'''(c)$, and $U_i''(c)$ and $U_i'''(c)$ are recomposed in a single PDF.

Note that the algorithm described above may seem to be complex and inefficient; however, to obtain the deadline miss probability, it is not necessary to compute the whole response time PDF, but only the values for $c \leq D_i$ are sufficient.

Finally, it is worth noting that the worst-case assumptions used by STDA (for example, the simultaneous arrival of all the tasks) are not used in the backlog-based analysis. As a consequence, the results of this kind of analysis are less pessimistic, and better approximate the deadline miss probability, as shown by the authors [DGK$^+$02]. However, the cost for such an increased accuracy is to perform a complete analysis in the whole hyperperiod.

9.3 REAL-TIME QUEUEING THEORY

The approach described in the previous section was developed by extending the deterministic real-time analysis. A completely different approach for analyzing the performance of a probabilistic real-time system is to extend the traditional queueing theory. Queueing theory models a system as a queue characterized by an *arrival process* and a server characterized by a *service process*:

- clients arrive in the queue with an interarrival distribution $V(t)$;
- each client is served in a random time distributed as $U(c)$.

If the server is idle when a new client arrives, then the server immediately starts to serve the client and will finish in a time distributed according to $U(c)$, otherwise the client is inserted in a queue. When the server finishes to serve a client, the next client is extracted from the queue; if the queue is empty, the server becomes idle. The mean number of clients in the queue is indicated by w, whereas T_w indicates the mean time a client spends in the queue. Similarly, T_s indicates the mean time needed by the server

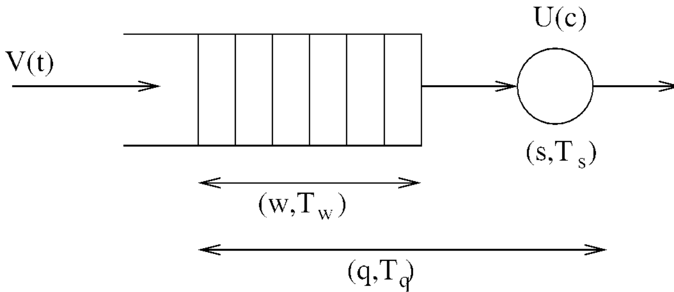


Figure 9.2 Model of a queue.

to serve a client, $q = w + 1$ indicates the mean number of clients in the system (server + queue), and T_q indicates the mean time spent by a client in the system. The model described above is illustrated in Figure 9.2.

The standard queueing theory provides tools for analyzing the statistical properties of a system modeled as a queue, under the simplifying assumption that the queueing discipline is FIFO [Kle75]. For example, Little's formula ensures that

$$q = \frac{T_q}{E[V(t)]}, \quad w = \frac{T_w}{E[V(t)]}, \quad \text{and} \quad T_q = T_w + T_s.$$

Moreover, in the case of Poisson arrivals and exponential service times (the M/M/1 case),

$$q = \frac{\rho}{1-\rho} \quad \text{and} \quad w = \frac{\rho^2}{1-\rho}, \quad \text{where} \quad \rho = \frac{T_s}{E[V(t)]}.$$

Other interesting results can be found on standard queueing theory books [Kle75]. Note that the previous formulas are only valid if $\rho < 1$ (i.e., if the mean interarrival time is greater than the mean service time). If $\rho > 1$, the queue will not reach a steady state, and its size will increase towards $+\infty$, whereas $\rho = 1$ is the *metastable* situation, in which nothing can be told about the queue state.

Classical queueing theory is very useful for analyzing network systems or computer systems where clients (packets or tasks) are served in a FIFO (or Round Robin) order, but it is not suitable for analyzing real-time schedulers that use different (and more complex) algorithms. For this reason, a real-time queueing theory was developed by Lehoczky [Leh96] to cope with complex scheduling algorithms and task timing constraints. Traditional queueing analysis is fairly simple, because a system (queue + server) can be described by a single state variable, as the number of clients in the queue. On the contrary, real-time queueing theory must distinguish the various clients

(tasks) to schedule them, and must characterize each task with its deadline. Hence, the system state becomes a vector $(m, l_1, \ldots, l_m)$, where m is the number of tasks in the queue, and l_i is the *lead time* of task τ_i, defined as $d_i - t$, where d_i is the deadline of the current job of task τ_i, and t is the current time.

For an M/M/1 queue (with $\lambda = 1/E[V(t)]$, and $\mu = 1/E[U(c)]$) scheduled using EDF, if $(m, l_1, \ldots, l_m)$ is the state at time t, the state at time $t + h$ (for a small h) can be computed as follows. If no jobs arrive and no jobs depart in the time interval $(t, t+h)$, then the state evolves in $(m, l_1 - h, \ldots, l_m - h)$; this happens with probability $1 - (\lambda + \mu)h + o(h)$. If a job ends during the interval $(t, t + h)$, then the new state is $(m - 1, l_2 - h, \ldots, l_m - h)$ (remember that jobs are assumed to be ordered by deadline); this happens with probability $\mu h + o(h)$. If, instead, a new job arrives with lead time a in $(t, t + h)$, then the new state is $(m + 1, a, l_1 - h, \ldots, l_m - h)$. This event has probability $dG(a)\lambda h + o(h)$, where $G(a)$ is the CDF of the relative deadlines. In a similar way, the state (0) evolves in (0) with probability $1 - (\lambda + \mu)h + o(h)$, and evolves in $(1, a)$ with probability $dG(a)\lambda h + o(h)$.

Similar computations can be repeated for other scheduling algorithms (such as a proportional share algorithm, or a fixed priority algorithm), permitting to reconstruct the evolution of the system state. Although the previous equations can be used to compute a probability distribution of the system state (e.g., the queue length distribution, the deadline miss probability, and so on), such a computation is very complex and cannot be easily extended to other (non M/M/1) queue models. To simplify the analysis, Lehoczky proposed to consider the case of a scheduler under *heavy traffic* conditions.

The heavy traffic analysis of a queue permits to compute a simple and insightful approximation of the complex exact solution, under some simplifying assumptions. In particular, when the traffic on a queueing system is high enough (i.e., ρ is near enough to 1), the queue can be described using a simpler model that can be easily analyzed [Dai95]. This is similar to the approach taken by the central limit theorem, which approximates the sum of a large number of independent random variables with a normal distribution.

The heavy traffic approximation is based on rescaling the time and the queue length in the model, and applying the heavy traffic condition $\lambda_n = \lambda(1 - \gamma/\sqrt{n}), \mu_n = \lambda$ (thus, the load is $\rho = 1 - \gamma/\sqrt{n}$). This approach can be taken to analyze the Markov process describing the real-time queue presented above. Although the formal analysis is fairly complex, the final results are very simple: for example, under EDF, the mean queue length turns out to be $q = \rho/(1 - \rho)$, and the PDF of the lead time results to be $f(x) = \lambda(1 - G(x))/q$.

Note that the heavy traffic assumption may seem to be too restrictive, but it is generally reasonable, because it covers the case that is interesting for stochastic real-time systems.

In fact, the deadline miss probability becomes significant when the system is near to the full utilization, and hence when it is under heavy traffic.

Finally, the heavy traffic analysis of a real-time queueing system can also be applied to fixed priority schedulers (obtaining an analysis of generalized RM or DM systems), or to hard real-time schedulers, in which clients with a negative lead time are automatically removed from the queue. This method has also been extended to networks of real-time queues [Leh97].

9.4 NOVEL ALGORITHMS FOR STOCHASTIC SCHEDULING

Another possible approach for performing a probabilistic guarantee of a real-time system is to modify the scheduling algorithm, instead of modifying only the admission test. For example, the transform-task method [TDS+95] allows splitting a semiperiodic task τ_i in two subtasks: a periodic subtask τ_i^P, which can be subject to a hard guarantee, and a sporadic subtask τ_i^S.

The periodic subtask τ_i^P has period T_i, a well known WCET C_i^P, and a relative deadline $D_i^P = \alpha D_i$, with $\alpha_i \in (0, 1)$. Note that C_i^P and α_i can be chosen during the design phase so that the set $\Gamma^P = \{\tau_i^P\}$ of the periodic subtasks is schedulable.

The sporadic subtask τ_i^S is used to serve the jobs $\tau_{i,j}$ having $c_{i,j} > C_i^P$, which cannot be completely served by the periodic subtask. Hence, a job $\tau_{i,j}^S$ of the sporadic task is released only if $c_{i,j} > C_i^P$, has execution time $c_{i,j}^S = c_{i,j} - C_i^P$, and arrives at time $r_{i,j}^S = f_{i,j}^P$ when the corresponding periodic job finishes. The sporadic subtask execution times are distributed according to the following PDF:

$$U_i^S(c) = \begin{cases} 0 & \text{if } c < 0 \\ \frac{U_i(c+C_i^S)}{A_i} & \text{if } x \geq 0 \end{cases}$$

where A_i is the probability $P\{c_{i,j}^S > 0\}$ of arrival of a sporadic job. Note that $U_i(c + C_i^S)$ must be divided by A_i to ensure that $\sum_{c=0}^{\infty} U_i^S(c) = 1$.

The sporadic subtasks can be served by using an aperiodic server, such as the Sporadic Server [SSL89] (but other service mechanisms can be used as well), or by using a Slack Stealer [LRT92, TL92]. In the original paper, the authors presented a probabilistic guarantee based on RM + Sporadic Server. Using the Sporadic Server, tasks with similar periods are "clustered" together; that is, Γ is partitioned into *clusters* Γ_k such

that all the sporadic subtasks belonging to a cluster are served by a server with a period $P_k^s : \forall \tau_i \in \Gamma_k, P_k^s \leq D_i$; that is, P_k^s is less than or equal to the minimum relative deadlines in the cluster. Moreover, the server budget is set to $C_k^s = C^k/K$, where $C^k = \max_{i:\tau_i \in \Gamma_k}\{C_i^S\}$ and $K = \min_{i:\tau_i \in \Gamma_k}\{\lfloor D_i/P_k^s \rfloor\}$. In this way, if a sporadic subtask arrives when the server is idle, then it is served in less than $KP_k^s \leq \min_{i:\tau_i \in \Gamma_k}\{D_i\}$, and it will respect its deadline.

The actual deadline miss probability can be computed by combining (through a convolution) the probability that a sporadic job $\tau_{i,j}^S$ requires h server periods to complete with the probability that the backlog in the server queue when $\tau_{i,j}^S$ arrives requires x server periods to be consumed. If $H_k(h)$ is the PDF of the number of server periods required to serve a sporadic job, and $X_k(x)$ is the PDF of the server backlog found by a sporadic job when it arrives, then

$$P\{\tau_i \in \Gamma_k \text{ misses a deadline}\} = \sum_{h=1}^{K} H(h) \sum_{x=K-h+1}^{\infty} X(x). \qquad (9.3)$$

The distribution $H(h)$ of the number of server periods needed to serve a job can be computed based on the PDFs of the execution times of the tasks belonging to the cluster Γ_k , whereas the Z transform of $X(x)$ is given by

$$\mathcal{X}(Z) = \frac{(1 - E[\delta])(1 - Z)}{Q(Z) - Z}, \qquad (9.4)$$

where $E[\delta]$ is the expected value of the number of server periods needed to serve all the requests arriving in a server period. Clearly, if $E[\delta] \geq 1$, the server queue will explode, because the amount of time to be served in a server period is bigger than the amount of time the server can serve in period. Hence, if $E[\delta] \geq 1$, a stationary distribution $X(x)$ cannot be found; note that $E[\delta]$ is similar to the load ρ of a queue. More details can be found on the original paper [TDS+95].

An alternative approach is to serve the periodic subtasks with an EDF scheduler, and a to use a Slack Stealer for serving the sporadic subtasks. The advantage of this second methodology is that tasks response times are expected to improve. However, no stochastic analysis of the EDF + Slack Stealer case has been performed, because of the difficulty of modeling the Slack Stealer behavior.

The two task transformation approaches presented above have been compared through a set of simulations [TDS+95], showing that EDF + Slack Stealer provides better response times than RM + Sporadic Server. However, RM + Sporadic Server gives more control on the deadline miss probability: there are tasks that when scheduled by RM + Sporadic Server have a larger average response time, but a lower deadline

miss probability (that is to say, the response time probability is concentrated on values smaller than the relative deadline).

A different approach has been proposed by the Statistical Rate Monotonic (SRMS) algorithm [AB98c], which provides a firm guarantee by accepting or rejecting each job on its arrival: if a job is accepted, it is guaranteed to respect its deadline, otherwise it does not even start executing. Moreover, a *per task* guarantee is performed on task creation, to guarantee a deadline miss probability for the task.

SRMS is based on a variation of the semiperiodic task model, in which each task τ_i is described by three parameters $(U_i(c), T_i, \delta_i)$, where $\delta_i = P\{f_{i,j} > d_{i,j}\}$ is the deadline miss probability requested by the task (in the original paper, it is called *task's requested QoS*). On task arrival, the system runs an admission test, checking whether a deadline miss probability equal to δ_i can be guaranteed to τ_i. If the test does not fail, the task is accepted. Once τ_i is accepted in the system, each arriving job $\tau_{i,j}$ is guaranteed to be accepted with a probability δ_i. Such a stochastic guarantee is achieved by SRMS through two different mechanisms: *accounting* of the time consumed by τ_i, and *aggregation* of consecutive jobs $\tau_{i,j}, \tau_{i,j+1}, \ldots$. This method can be used only if the execution time $c_{i,j}$ of job $\tau_{i,j}$ is known at the job arrival time.

To perform execution time accounting, SRMS associates a budget (the maximum budget is called *allowance* in the original paper) to each task: as in a reservation based algorithm, the budget is periodically replenished every P_i^s units of time (P_i^s is referred as *superperiod*) and is decreased when a job $\tau_{i,j}$ executes. However, since SRMS is a firm algorithm, the budget can be immediately decreased when a job arrives, and can be used to accept or reject a job, as will be shown later. In the original paper, the superperiod of task τ_i is defined to be equal to the period T_{i+1} of the next lower priority task τ_{i+1}.

More formally, SRMS works as follows:

- at the beginning of each superperiod, the budget q_i of task τ_i is replenished to its allowance;
- when a job $\tau_{i,j}$ requiring an execution time $c_{i,j}$ arrives at time $r_{i,j}$, it is accepted if $c_{i,j} \le q_i$ and $c_{i,j} \le T_i - \sum_{k=1}^{i-1} Q_k^s T_i / P_k^s$. If $\tau_{i,j}$ is accepted, q_i is decreased by $c_{i,j}$.
- accepted jobs are scheduled according to RM.

Note that the first condition for admitting a job ($c_{i,j} \le q_i$) ensures that each task τ_i will never consume more than its allowance in a superperiod, whereas the second condition

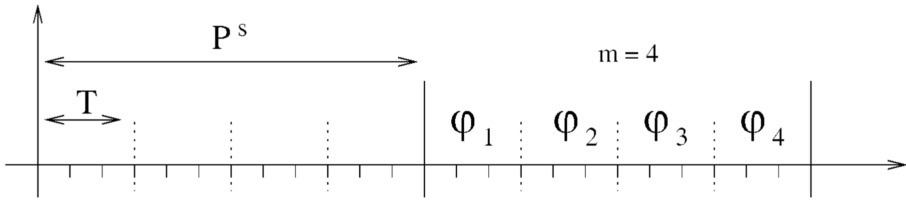


Figure 9.3 Since the task superperiod P_i^s is an integer multiple of the task period T_i, it is divided into $m = P_i^s/T_i$ phases.

($c_{i,j} \leq T_i - \sum_{k=1}^{i-1} Q_k^s T_i/P_k^s$) guarantees that an admitted job will finish within its deadline, because the difference between the budget and the maximum amount of time required by high priority tasks is greater than the job execution time.

The task's admission control used to guarantee that each task will receive the desired QoS (that is, that a job $\tau_{i,j}$ will be rejected with a probability not greater than $1 - \delta_i$) is performed in two steps: a first test guarantees that each task τ_i will receive its allowance Q_i^s in its superperiod; then, a second test verifies whether the pair (Q_i^s, P_i^s) is enough for guaranteeing δ_i. As for a traditional reservation algorithm, each task is guaranteed to receive Q_i^s units of time every P_i^s if $\sum_{i=1}^{n} Q_i^s/P_i^s \leq U^{lub}$. Note that $U^{lub} = 1$, because the analysis is performed under the assumption that periods are harmonic. Also, P_i^s is set to be equal to T_{i+1}. As a result, the first part of the admission control is

$$\sum_{i=1}^{n} \frac{Q_i^s}{T_{i+1}} \leq 1.$$

If this condition is verified, each task τ_i is automatically guaranteed to receive its allowance, hence for harmonic task sets the probability of rejecting a job can be computed by considering only the first of the two job admission conditions.

Since the task set is harmonic, the superperiod T_{i+1}, will be an integer multiple of T_i, and $m = T_{i+1}/T_i$ jobs $\tau_{i,j}, \ldots, \tau_{i,j+m-1}$ will be released in a superperiod, dividing it into m phases, as illustrated in Figure 9.3. A job arriving in the first phase of a superperiod always finds a budget $q_i = Q_i^s$, and will be accepted if $c_{i,j} \leq Q_i^s$. Hence, if $\pi_{i,k}$ is the probability to accept a job $\tau_{i,j}$ arriving in the kth phase,

$$\pi_{i,1} = P\{c_{i,j} \leq Q_i^s\} = \sum_{c=0}^{Q_i^s} U(c).$$

The probability $\pi_{i,2}$ to accept a job $\tau_{i,j}$ arriving in the second phase is given by the sum of two terms, considering the two cases in which the job arriving in the first phase

was accepted or rejected:

$$\pi_{i,2} = P\{c_{i,j} \leq Q_i^s\}(1 - \pi_{i,1}) + P\{c_{i,j} + c_{i,j-1} \leq Q_i^s\}\pi_{i,1};$$

since $c_{i,j}$ and $c_{i,j-1}$ have the same PDF (because jobs are supposed to be independent), $P\{c_{i,j} + c_{i,j-1} \leq Q_i^s\} = P\{2c_{i,j} \leq Q_i^s\}$, and the probability can be easily computed. The other probabilities $\pi_{i,k} : 0 < k \leq T_{i+1}/T_i$ can be computed in a similar way, by considering all the possible histories of the task inside the first k phases, and by expressing $\pi_{i,k}$ as a sum of 2^{k-1} terms.

If the jobs are uniformly distributed in the various phases, the probability δ_i to accept a job $\tau_{i,j}$ is

$$\delta_i = \frac{T_i}{T_{i+1}} \sum_{k=1}^{T_{i+1}/T_i} \pi_{i,k}.$$

If the task periods are not harmonic, the complexity of the analysis increases, because there are situations in which a job released in a superperiod can have its deadline in the next superperiod. This case can be addressed in three different ways:

1. admitting the job based on the current budget (the budget of the superperiod in which the job arrives);
2. admitting the job based on the budget in the next superperiod (the superperiod in which the job deadline is). In this case, the job execution must be delayed until the next superperiod, or until all the lower priority tasks are inactive;
3. splitting the job in two sub-jobs, guaranteeing the first one in the current superperiod, and the second one in the next superperiod.

These possible solutions have been considered in a separate paper [AB98b], and the analysis is omitted here for the sake of simplicity. Note that in this case the second job admission rule has to be considered too.

Quality-Assuring Scheduling [HLR$^+$01] uses a technique similar to Task Transformation to perform a stochastic guarantee of tasks composed by mandatory parts and optional parts (this task model is similar to the imprecise computation model [LLN87]). More specifically, every job $\tau_{i,j}$ consists of a *mandatory part* $\tau_{i,j}^M$ and o_i optional parts $\tau_{i,j,1}^O, \ldots, \tau_{i,j,o_i}^O$. The mandatory part is released periodically with a period T_i, is executed before all the optional parts, and must be completed within the job deadline $d_{i,j} = r_{i,j} + T_i$. Optional parts are sequentially released (that is, $\tau_{i,j,k+1}^O$ starts when $\tau_{i,j,k}^O$ finishes) after the end of $\tau_{i,j}^M$ and can miss the job deadline; if the optional part

$\tau^O_{i,j,k}$ misses the deadline, then all the successive optional parts $\tau_{i,j,k+1}, \ldots$ of the current job are skipped.

In Quality-Assuring Scheduling, each task $\tau_i = (U_i^M(c), U_i^O(c), o_i, q_i, T_i)$ is described by the PDF $U_i^M(c)$ of the mandatory part execution time, the PDF $U_i^O(c)$ of the optional parts execution times, the number of optional parts o_i, the fraction q_i of optional parts that must finish within their deadline to respect the task's QoS requirements, and the task period T_i. The worst-case execution time C_i of the mandatory parts is implicitly described by $U_i^M(c)$, since $C_i = max\{c : U_i^M(c) > 0\}$, by definition.

Tasks are scheduled according to a reservation approach, by reserving a time C_i to the mandatory part of task τ_i, and a time Q_i^O to the optional parts of τ_i. All the optional parts are scheduled in background respect to the mandatory parts of each task, and a reservation (Q_i^O, T_i) is used for serving them. Clearly, all the mandatory parts will respect their deadlines if $\sum_i C_i/T_i \leq U^{lub}$.

One of the most interesting characteristics of Quality-Assuring Scheduling is that it is one of the few algorithms that have been also applied to resources other than the CPU, such as a SCSI disk. When applying the algorithm to disk scheduling, all the tasks happen to have the same period, and the scheduled resource is non-preemptable. If all tasks have the same period T, all the mandatory parts can be scheduled at the same priority (higher than all the priorities of the optional parts), and the admission test for the mandatory parts is $\sum_i C_i \leq T$. Without any loss of generality, tasks can be ordered according to the optional parts' priorities, hence τ_1 is the task with the highest priority optional parts, and the first optional part $\tau^O_{1,j,1}$ of τ_1 starts executing immediately after the mandatory parts $\tau^M_{1,j}, \ldots, \tau^M_{n,j}$.

To guarantee that a task τ_i will respect its QoS parameter q_i (i.e., that it will complete at least a fraction q_i of its optional parts), the fraction q_i of completed optional parts can be computed as $q_i = E[A_i]/o_i$, where A_i is a random variable indicating the number of optional parts of task τ_i completed in a period (note that since the scheduler is non-preemptable, this is equal to the number of optional parts that can be started in a period). By definition,

$$\begin{aligned} E[A_i] &= \textstyle\sum_{k=1}^{o_i} P\{A_i = k\}k \\ &= \textstyle\sum_{k=1}^{o_i} (P\{A_i \geq k\} - P\{A_i \geq k+1\})k \\ &= \textstyle\sum_{k=1}^{o_i} P\{A_i \geq k\} - P\{A_i \geq o_i + 1\}o_i, \end{aligned}$$

and, since $P\{A_i \geq o_i + 1\} = 0$, we have:

$$E[A_i] = \sum_{k=1}^{o_i} P\{A_i \geq k\}.$$

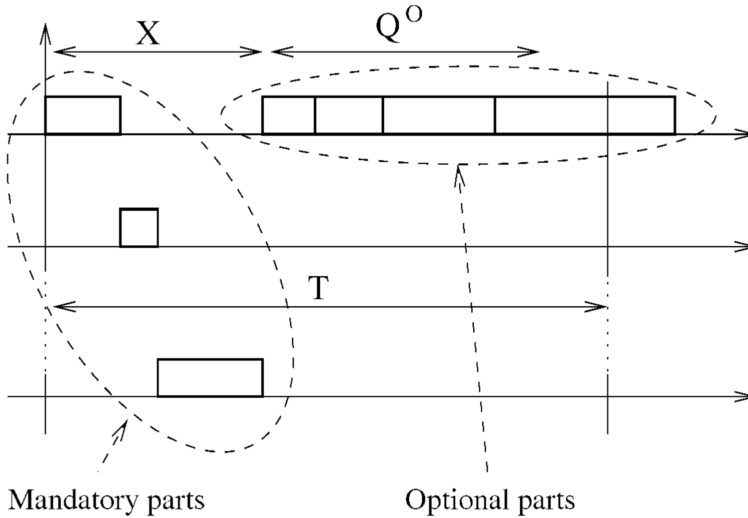


Figure 9.4 Number of optional parts scheduled for the first task.

The PDFs $P\{A_i = k\}$ of the number of completed optional parts are computed starting from task τ_1: if X is a random variable given by the sum of the execution times $c_1, \ldots, c_n$ of the mandatory parts, then $P\{A_1 = k\}$ can be computed by finding the probability that X leaves enough free time for k optional parts in a period, and that the time Q_1^O reserved for optional parts is enough (see Figure 9.4). This computation can be easily performed because $P\{X = x\}$ can be obtained by convolving the $U_i^M(c)$ PDFs.

Once $P\{A_1 = k\}$ is known, it is possible to obtain the PDF of the time consumed by the optional parts of τ_1. Note that since the scheduled resource is non preemptable the optional parts can consume more than Q_i^O, as shown in Figure 9.4. Such a distribution $P\{kc_1^O = y\}$ is obtained by convolving $U_1^O(c)$ with itself $k - 1$ times, and then convolved it with the PDF of X, obtaining the PDF of a new variable X', that can be used to compute $P\{A_2 = k\}$ (that is, the distribution of the number of optional parts started for task τ_2). In fact, the probability $P\{A_2 = k\}$ is computed by repeating the process used for computing $P\{A_1 = k\}$, but using X' instead of X. This process can be iterated to obtain the probability for all the other tasks.

Since all the tasks have the same period T, the RM assignment does not give any useful hints in deriving tasks' priorities. The authors propose a priority assignment called Quality Monotonic Scheduling (QMS), that assigns higher priorities to tasks

with a higher quality parameter q_i (remember that mandatory parts always have higher priorities than optional parts).

The proposed analysis can be extended to preemptable resources by changing the way in which X' is computed (see the original paper [HLR+01] for all the details).

When considering arbitrary periods, the admission test for mandatory parts ($\sum_i C_i \leq T$) must be changed: if periods are harmonic, U^{lub} is still 1, and the test is $\sum_i C_i/T_i \leq 1$. After passing this admission test, the optional parts can be guaranteed as shown above. If periods are not harmonic, an exact schedulability test (based on response time analysis or on time demand analysis) should be used, although the authors analyzed the behavior of the algorithm by simulation.

As a final consideration, it is worth noting that all the modifications proposed to the traditional scheduling algorithms for controlling the deadline miss probability (i.e., the Task Transformation method, SRMS, and Quality-Assuring Scheduling) tend to implement some kind of temporal protection among tasks (see also Chapter 3). This fact seems to suggest that temporal protection is helpful to simplify the probabilistic analysis of a real-time system: for this reason, a stochastic guarantee of reservation based schedulers is presented in the next section.

9.5 RESERVATIONS AND STOCHASTIC GUARANTEE

One of the advantages of using a reservation-based scheduling approach such as the CBS, is that the scheduling parameters (Q_i^s, P_i^s) can be separated from the task characteristics (such as execution and interarrival times). In this way, if task τ_i is described by a pair of PDFs of the execution and interarrival times, then it is possible to perform a probabilistic guarantee, as defined in the previous sections. For example, a reservation system can be analyzed by modeling each task as a queue to perform a probabilistic guarantee [AB99b, AB01b, AB04]. This model works as follows:

1. each P_i^s units of time, Q_i^s units of time can be served;
2. the arrival of job $\tau_{i,j}$ corresponds to a request of $c_{i,j}$ units of time entering the queue.

This model is similar to the one used by the Task Transformation method to analyze the sporadic subtasks, with the difference that here each task has its own queue. Having per-task queues also simplifies the analysis with respect to the real-time queueing

theory: in fact, since all the jobs of a single task are served in a FIFO order, there is no need to model the scheduler behavior in the queue.

The evolution of a reservation is described by a random process x_j (indicating the amount of execution time that "has accumulated" in the queue immediately after the beginning of a reservation period, or immediately after a job arrival). Hence, it is possible to compute the probability distribution of x_j, and to derive the deadline miss probability from it.

To perform a stochastic analysis of a generic reservation-based scheduling algorithm, the simplified case in which $r_{i,j+1} - r_{i,j}$ is a multiple of P_i^s is considered first, and the analysis is then extended to cope with generic interarrival times. Hence, a new property is added to the previously described model:

3. when a job arrives, the next request of c_{j+1} units will arrive after $r_{i,j+1} - r_{i,j} = z_{i,j} P_i^s$ units of time

where $z_{i,j} = (r_{i,j+1} - r_{i,j})/P_i^s$.

Since execution and interarrival times are random variables described by the PDFs $U_i(c)$ and $V_i(t)$, the amount of execution time x that still has to be served immediately after a job arrival is a random variable too, described by a PDF $\pi_k^{(i,j)} = P\{x_{i,j} = k\}$.

Being Q_i^s time units served every period P_i^s, job $\tau_{i,j}$ will finish before time

$$r_{i,j} + \left\lceil \frac{x_{i,j}}{Q_i^s} \right\rceil P_i^s,$$

hence the probability $\pi_k^{(i,j)}$ that the queue length $x_{i,j}$ is k, immediately after a job arrival, is a lower bound of the probability $P\{f_{i,j} - r_{i,j} \leq \delta_i\}$ that the job finishes before the probabilistic deadline

$$\delta_i = \left\lceil \frac{k}{Q_i^s} \right\rceil P_i^s.$$

Being the interarrival times multiple of the server period P_i^s, it is possible to define $V_i'(z) = P\{r_{i,j} - r_{i,j-1} = zP_i^s\}$ as the probability that the interarrival time between two consecutive jobs is zP_i^s. Hence,

$$V(t) = \begin{cases} 0 & if \quad t \bmod P^s \neq 0 \\ V'(\frac{t}{P^s}) & \text{otherwise.} \end{cases} \tag{9.5}$$

Note that, since $c_{i,j}$ and $r_{i,j+1} - r_{i,j}$ are time invariant, $U_i(c)$ and $V_i'(z)$ do not depend on j. Under these assumptions, it is possible to compute $\pi_k^{(i,j)}$ as follows:

$$\pi_k^{(i,j)} = P\{x_{i,j} = k\}P\{\max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k\} =$$

$$= \sum_{h=-\infty}^{\infty} P\{\max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k \wedge x_{i,j-1} = h\} =$$

$$= \sum_{z=-\infty}^{\infty} \sum_{h=-\infty}^{\infty} P\{max\{0, x_{i,j-1} - z_{i,j}Q^s\} + c_{i,j} = k \wedge x_{i,j-1} = h \wedge z_{i,j} = z\}.$$

Being $x_{i,j}$ and $z_{i,j}$ greater than 0, by definition, the sums can be computed for h and z going from 0 to infinity:

$$\pi_k^{(i,j)} = \sum_{z=0}^{\infty} \sum_{h=0}^{\infty} P\{max\{0, h - zQ^s\} + c_{i,j} = k\}P\{x_{i,j-1} = h\}P\{z_{i,j} = z\}$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} P\{max\{0, h - zQ^s\} + c_{i,j} = k\}V_i'(z)\pi_h^{(i,j-1)} =$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} P\{c_{i,j} = k - max\{0, h - zQ^s\}\}V_i'(z)\pi_h^{(i,j-1)} =$$

$$= \sum_{h=0}^{\infty} \sum_{z=0}^{\infty} U(k - max\{0, h - zQ^s\})V_i'(z)\pi_h^{(i,j-1)}$$

Hence,

$$\pi_h^{(i,j)} = \sum_{h=0}^{\infty} m_{h,k}\pi_h^{(i,j-1)}$$

with

$$m_{h,k}^i = \sum_{z=0}^{\infty} U_i(k - max\{0, h - zQ_i^s\})V_i'(z). \tag{9.6}$$

Considering $m_{h,k}^i$ as an element of a matrix M^i, $\pi_k^{(i,i)}$ can be computed by solving the equation

$$\Pi^{(i,j)} = M^i\Pi^{(i,j-1)} \tag{9.7}$$

where

$$\Pi^{(i,j)} = \begin{pmatrix} \pi_0^{(i,j)} \\ \pi_1^{(i,j)} \\ \pi_2^{(i,j)} \\ \pi_3^{(i,j)} \\ \cdot \\ \cdot \\ \cdot \end{pmatrix}.$$

9.5.1 STABILITY CONSIDERATIONS

As already stated in Section 9.3, it is known that a generic queue is stable (i.e., the number of elements in the queue do not diverge to infinity) if

$$\rho = \frac{mean\ interarrival\ time}{mean\ service\ time} < 1.$$

Hence, the stability can be achieved under the condition

$$\frac{E[C_i]}{E[T_i]} < \frac{Q_i^s}{P_i^s}$$

where $E[C_i]$ is the execution time expectation and $E[T_i]$ is the interarrival time expectation.

If this condition is not satisfied, then the difference $f_{i,j} - r_{i,j}$ between the finishing time $f_{i,j}$ and the arrival time $r_{i,j}$ of each job $\tau_{i,j}$ of task τ_i will increase indefinitely, diverging to infinity as j increases:

$$\lim_{j \to \infty} (f_{i,j} - r_{i,j}) = \infty.$$

This means that, for preserving the schedulability of the other tasks, τ_i will slow down in an unpredictable manner.

If a queue is stable, a stationary solution of the Markov chain describing the queue can be found; that is, there exists a finite solution Π^i such that $\Pi^i = \lim_{j \to \infty} \Pi^{(i,j)}$. Since $\Pi^{(i,j)} = M^i \Pi^{(i,j-1)}$, we can compute Π as follows:

$$\Pi^i = \lim_{j \to \infty} \Pi^{(i,j)} = \lim_{j \to \infty} M^i \Pi^{(i,j-1)} =$$

$$= M^i \lim_{j \to \infty} \Pi^{(i,j-1)} = M^i \Pi^i.$$

Hence, Π^i can be computed by solving the eigenvector problem

$$\Pi^i = M^i \Pi^i.$$

This solution can be approximated by truncating the infinite dimension matrix M^i to an nxn matrix $\tilde{M}^i$ and solving the eigenvector problem $\tilde{\Pi}^i = \tilde{M}^i \tilde{\Pi}^i$ with some numerical calculus technique.

9.5.2 RELAXING THE HYPOTHESIS ON INTERARRIVAL TIMES

In the previous analysis, task interarrival times are assumed to be multiple of an integer value P_i^s, so that Equation (9.5) is verified. This assumption is very useful to simplify the queue analysis, but can be unrealistic in practical situations.

Using some appropriate approximations, it is possible to relax the assumption on the interarrival times without compromising the analysis based on it. When Equation (9.5) is not respected, it is convenient to introduce a new distribution $\tilde{V}_i(t)$ that approximates $V_i(t)$ for enabling the previously developed analysis. In this way, it is possible to analyze the task behavior based on the approximate PDF $\tilde{V}_i(t)$ instead of the actual PDF $V_i(t)$. In order for this approximation to be correct, $\tilde{V}_i(t)$ must

- be conservative (pessimistic);
- verify Equation (9.5).

Being "conservative" means that *if a probabilistic deadline can be guaranteed using $\tilde{V}_i(t)$, it is guaranteed also according to the real distribution $V_i(t)$*. Since the opposite is not true, this approach is pessimistic.

The new PDF $\tilde{V}_i(t)$ is conservative if

$$\forall k, \sum_{n=0}^{k} \tilde{V}_i(n) \geq \sum_{n=0}^{k} V_i(n), \tag{9.8}$$

while the second requirement states that

$$\tilde{V}_i(t) = \begin{cases} 0 & if \quad t \bmod P_i^s \neq 0 \\ V_i'(\frac{t}{P_i^s}) & \text{otherwise.} \end{cases}$$

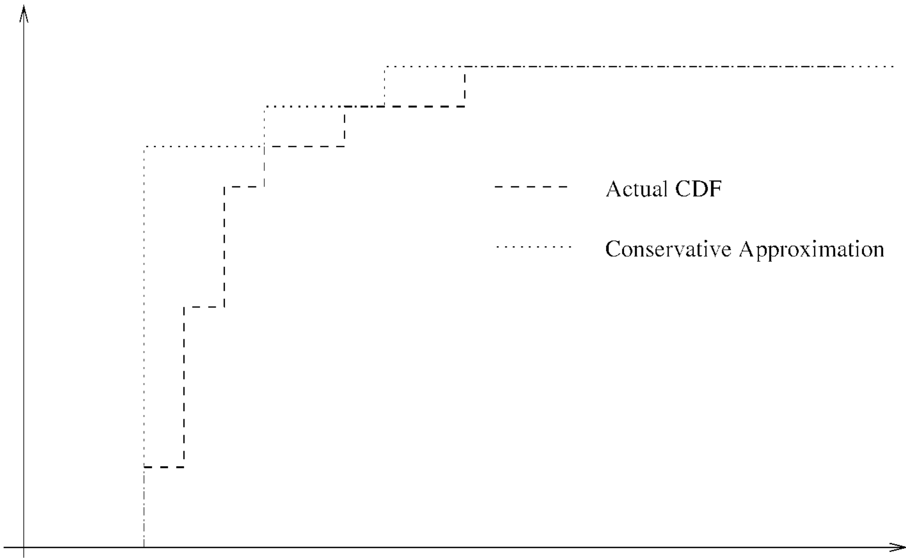


Figure 9.5 Conservative approximation of a CDF.

Equation (9.8) states that the approximated interarrival times CDF $\tilde{W}_i(t)$ computed from $\tilde{V}_i(t)$ must be greater than or equal to the interarrival times CDF $W_i(t)$ computed from $V_i(t)$ (recall that the CDF of a stochastic variable expresses the probability that the variable is less than or equal to a given value).

In practice, the intuitive interpretation of Equation (9.8) is that $\tilde{V}_i(t)$ is conservative if the probability that the interarrival time is smaller than t according to $\tilde{V}_i(t)$ is bigger than according to $V_i(t)$. This concept is explained in Figure 9.5.

Given a generic PDF $V_i(t)$, it is possible to generate a conservative approximation $\tilde{V}_i(t)$ if $\exists k : t < k \Rightarrow V_i(t) = 0$. In this case, it is possible to set $P_i^s < k$ and to compute

$$\tilde{V}_i(t) = \begin{cases} 0 & if \quad t \bmod P_i^s \neq 0 \\ \sum_{i=t-P_i^s+1}^{t} V_i(t) & \text{otherwise.} \end{cases} \tag{9.9}$$

It can easily be verified that, if $\tilde{V}_i(t)$ is computed according to Equation (9.9), then it will have both the required properties.

REFERENCES

[AAS97] T.F. Abdelzaher, E.M. Atkins, and K.G. Shin. Qos negotiation in real-time systems and its applications to automated flight control. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[AB98a] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, december 1998. IEEE.

[AB98b] A. K. Atlas and A. Bestavros. Multiplexing vbr traffic flows with guaranteed application-level qos using statistical rate monotonic scheduling. Technical Report BUCS-TR-98-011, Boston University, 1998.

[AB98c] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[AB99a] Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[AB99b] Luca Abeni and Giorgio Buttazzo. Qos guarantee using probabilistic deadlines. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[AB00] Luca Abeni and Giorgio Buttazzo. Support for dynamic QoS in the HARTIK kernel. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[AB01a] Luca Abeni and Giorgio Buttazzo. Hierarchical qos management for time sensitive applications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[AB01b] Luca Abeni and Giorgio Buttazzo. Stochastic analysis of a reservation based system. In *Proc. of the 9th International Workshop on Parallel and Distributed Real-Time Systems*, San Francisco, CA, April 2001.

[AB04] Luca Abeni and Giorgio Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[Abe98] Luca Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[ACr02] B. Bernhardsson A. Cervin, J. Eker and K.-E. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, July 2002.

[Apa98] D. Aparah. Adaptive resource management in a multimedia operating system. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, Cambridge, UK, July 1998.

[APLW02] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, December 2002.

[BA02a] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1-2):7–24, July-September 2002.

[BA02b] Giorgio Buttazzo and Luca Abeni. Smooth rate adaptation through impedance control. In *IEEE Proceedings of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[Bak90] T. P. Baker. A stack-based allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, december 1990.

[Bak91] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[BAL98] Giorgio Buttazzo, Luca Abeni, and Giuseppe Lipari. Elastic task model for adaptive rate control. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[BB02] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic algorithm. In *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, December 2002.

[BC99] Giorgio C. Buttazzo and Marco Caccamo. Minimizing aperiodic response times in a firm real-time environment. *IEEE Transactions on Software Engineering*, 25(1), January/February 1999.

[BCRZ99] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli. Rate modulation of soft real-time tasks in autonomous robot control systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, York, June 1999.

[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.

[BLCA02] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[BNBM98] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1998.

[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. *The 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.

[BSR88] S. Biyabani, J. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proc. of the IEEE Real-Time Systems Symposium*, 1988.

[But93a] G. Buttazzo. Hartik: A real-time kernel for robotics applications. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 201–205, Raleigh-Durham, December 1993.

[But93b] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[But97] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[CB97] M. Caccamo and G.C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *IEEE Real-Time Systems Symposium*, pages 330–339, San Francisco, 1997.

[CBS00] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium*, Orlando, Florida, December 2000.

[CE00] A. Cervin and J. Eker. Feedback scheduling of control tasks. In *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, December 2000.

[Cer03] A. Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, April 2003.

[CL90] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2, 1990.

[CLS99] Raj Rajkumar Chen Lee, John Lehoczky and Dan Siewiorek. On quality of service optimization with discrete qos options. In *Proceedings of the IEEE Real-time Technology and Applications Symposium*, june 1999.

[CMDD62] F. J. Corbato, M. Merwin-Dagget, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Joint Computer Conference*, May 1962.

[CS01] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[Dai95] J. G. Dai. On positive harris recurrence of multiclass queuing networks: a unified approach via fluid limit model. *Ann. of App. Prob.*, 1995.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 1974.

[DGK$^+$02] José Luis Diaz, Daniel F. Garcia, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José Maria López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis op periodic real-time systems. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, December 2002.

[DL97] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[DLS97] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Ninth Euromicro Workshop on Real-Time Systems*, 1997.

[FM02] Xiang Feng and Aloysius K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the* 23$^{\text{rd}}$ *IEEE Real-Time Systems Symposium*, pages 26–35, Austin, TX, USA, December 2002.

[FNT95] Hiroshi Fujita, Tatsuo Nakajima, and Hiroshi Tezuka. A processor reservation system supporting dynamic qos control. In *2nd International Workshop on Real-Time Computing Systems and Applications*, October 1995.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems developmet. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[GB95] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System*, 9, 1995.

[GB00] G.Lipari and S.K. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[GGV96] Pawan Goyal, Xingang Guo, and Harrik M. Vin. A hierical cpu scheduler for multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

[GL99] Mark K. Gardner and Jane W. S. Liu. Analyzing stochastic fixed/priority real-time systems. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Amsterdam, March 1999.

[HLC91] J.R. Haritsa, M. Livny, and M.J. Carey. Earliest deadline scheduling for real-time database systems. In *Proc. of the IEEE Real-Time Systems Symposium*, December 1991.

[HLR+01] Claude-Joachin Hamann, Jork Loser, Lars Reuther, Sebastian Schonberg, Jean Wolter, and Hermann Haertig. Quality-assuring scheduling — using stochastic behavior to improve resource utilization. In *Proceedings of the IEEE Real-Time Systems Symposium*, London, UK, December 2001.

[HLW91] K. Ho, J. Leung, and W.-D. Wei. Scheduling imprecise computation tasks with 0/1-constraints. Technical report, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 1991.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Treansactions on Computers*, 44(12):1443–1451, december 1995.

[JB95] K. Jeffay and D. Bennet. A rate-based execution abstraction for multimedia computing. In *Network and Operating System Support for Digital Audio and Video*, 1995.

[Jef92] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, December 1992.

[JIF+96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An overview of the rialto real-time architecture. In *In Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[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.

[KL99] Tei-Wei Kuo and Ching-Hui Li. Fixed-priority-driven open environment for real-time applications. In *Proceedings of the IEEE Real Systems Symposium*, December 1999.

[Kle75] L. Kleinrock. *Queuing Systems*. Wiley-Interscience, 1975.

[KM91] T.-W. Kuo and A. K. Mok. Load adjustment in adaptive real-time systems. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[KS92] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[KS95] G. Koren and D. Shasha. Skip-over:algorithms and complexity for overloaded systems that allow skips. In *Proceedings of IEEE Real-Time System Symposium*, pages 110–117, December 1995.

[LB99] G. Lipari and G.C. Buttazzo. Scheduling real-time multi-task applications in an open system. In *Proceeding of the 11th Euromicro Workshop on Real-Time Systems*, York, UK, June 1999.

[LB00] G. Lipari and G. C. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of System Architecture*, 46:327–338, 2000.

[LB03] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Euromicro Conference on Real-Time Systems (ECRTS 03)*, Porto, June 2003.

[LCB00] G. Lipari, J. Carpenter, and S.K. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *Proceedings of the 21st Real-Time System Symposium*, Dec. 2000.

[Leh90] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1990.

[Leh96] John P. Lehoczky. Real-time queueing theory. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[Leh97] J. P. Lehoczky. Real-time queuing network theory. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Francisco, 1997.

[Lip00] Giuseppe Lipari. *Resource Reservation in Real-Time Systems*. PhD thesis, Scuola Superiore S.Anna, Pisa, Italy, 2000.

[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.

[LLB+97] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Real-Time Computing Systems and Applications*, October 1997.

[LLN87] J. W. S. Liu, K. J. Lin, and S. Natarjan. Scheduling real-time, periodic jobs using imprecise results. In *IEEE Real Time System Symposium*, San Jose, California, December 1987.

[LLS+91] J.W.S. Liu, K. Lin, W. Shih, A. Yu, C. Chung, J. Yao, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[LNL87] K.J. Lin, S. Natarajan, and J.W.S. Liu. Concord: a system of imprecise computation. In *Proceedings of the 1987 IEEE Compsac*, October 1987.

[Loc86] C.D. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, 1986.

[LRLS98] Chen Lee, Raj Rajkumar, John Lehoczky, and Dan Siewiorek. Practical solutions for qos-based resource allocation. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[LRM96] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *Proceedings of Multimedia*, Japan, April 1996.

[LRT92] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992.

[LSA+00] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Orlando, FL, December 2000.

[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.

[LSL+94] J.W.S. Liu, W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, January 1994.

[LSS87] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[LSTS99] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the IEEE Real Time Systems Symposium*, Phoenix, Arizona, December 1999.

[MF01] Aloysius K. Mok and Xiang Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.

[MFC01] Aloysius K. Mok, Xiang Feng, and Deji Chen. Resource partition for real-time systems. In *Proceedings of the* 7th *IEEE Real-Time Technology and Applications Symposium*, pages 75–84, 2001.

[MST94a] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of IEEE international conference on Multimedia Computing and System*, may 1994.

[MST94b] C. W. Mercer, S. Savage, and H. Tokuda. Temporal protection in real-time operating systems. In *Proceedings of th 11th IEEE workshop on Real-Time Operating System and Software*, pages 79–83. IEEE, may 1994.

[Nak98a] T. Nakajima. Dynamic qos control and resource reservation. In *IEICE (RTP 98)*, 1998.

[Nak98b] T. Nakajima. Resource reservation for adaptive qos mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems*, April 1998.

[Nak98c] T. Nakajima. Resource reservation for adaptive qos mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 1998.

[Nat95] Swaminathan Natarajan, editor. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[NT94] Tatsuo Nakajima and Hiroshi Tezuka. A continuous media application supporting dynamic qos control on real-time mach. In *ACM Multimedia*, 1994.

[OR98] Shui Oikawa and Raj Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

[OR99] Shui Oikawa and Raj Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, June 1999.

[PG93] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[PG94] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in intergrated services networks: the multiple node case. *IEEE/ACM Transanctions on Networking*, 2:137–150, April 1994.

[PGBA02] P. Pedreira, P. Gai, G. Buttazzo, and L. Almeida. Ftt-ethernet: A platform to implement the elastic task model over message streams. In *Proceedings of the 4th IEEE Workshop on Factory Communication Systems (WFCS 2002)*, pages 225–232, Västeras, Sweden, August 2002.

[RBF+89] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*, September 1989.

[RJMO97] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the 4th Real-Time Computing Systems and Application Workshop*. IEEE, November 1997.

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[RS84] K. Ramamritham and J.A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), July 1984.

[RS01] John Regehr and John A. Stankovic. Augmented CPU Reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[RSY98] D. Rosu, K. Schwan, and S. Yalamanchili. Fara - a framework for adaptive resource allocation in complex real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[SAWJ+96] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real Time System Symposium*, 1996.

[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.

[SG90] L. Sha and J. Goodenough. Real-time scheduling theory and ada. *IEEE Computer*, 23(4), April 1990.

[SGG+99] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.

[SL03] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2003.

[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 enhanced aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 4(1), January 1995.

[SLS99] J. A. Stankovic, C. Lu, and S. H. Son. The case for feedback control in real-time scheduling. In *Proceedings of the IEEE Euromicro Conference on Real-Time*, York, England, June 1999.

[SLSS97] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[SR87] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.

[SRBS98] J.A. Stankovic, K. Ramamritham, G.C. Buttazzo, and M. Spuri. *Deadline Scheduling for Ral-Time Systems – EDF and Related Algorithms*. Kluwer Academic Publisher, 1998.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[SRLK02] Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the* 14th *IEEE Euromicro Conference on Real-Time Systems*, June 2002.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1, July 1989.

[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.

[Tan96] Andrew S. Tanembaum. *Computer Networks*. Prentice Hall, 1996.

[TDS+95] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Real-Time Technology and Applications Symposium*, pages 164–173, Chicago,Illinois, January 1995.

[Tim03] Timesys linux. www.timesys.com, 2003.

[TL92] S. R. Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed priority systems using slack stealing. Technical report, AT&T Bell Laboratories, Holmdel, NJ, 1992.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, pages 73–82, October 1990.

[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.

[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

Abeni, 207
Absolute deadline, 2
Activation overrun, 7
Adaptation, 21
Adaptive reservations, 222
Admission control, 27, 31, 64, 220, 249
Admission test, 235
Allocation error, 69
Aperiodic servers, 65
Aperiodic task, 4
Application, 97
Application-level adaptation, 228
Arrival time, 2
Asynchronous events, 23
Availability factor, 117, 119

B

Baker, 138
Bandwidth, 6
Bandwidth compression, 223
Bandwidth Inheritance, 158
Bandwidth isolation, 62
Baruah, 30, 116, 189
Best-Effort Scheduling, 31
Bini, 124
Biyabani, 25
Blocking time, 136, 140, 148, 157
Bounded-delay model, 119
BSS, 109
Budget, 75, 248
Buttazzo, 26, 33, 45, 48, 104, 124, 141, 207
BWI, 158

C

Caccamo, 45, 141–142
Capacity Sharing algorithm, 177
Carey, 25
CASH, 177
CBS, 65, 79
CBS-R, 147
CDF, 236
Chunk, 79
Clairvoyant scheduling, 30
Closed-loop scheduling, 220
Competitive factor, 30
Completion time, 2
Composition of schedulers, 99
Compression function, 224
Computation time, 2
Concurrent activities, 95
Concurrency control, 13, 114
Constant Bandwidth Server, 65, 79
Constant Utilization Server, 104
Control theory, 221
Convolution, 237, 240–241, 252
Cost function, 128, 130, 233–234
CPU quantum, 69
Critical instant, 239
Critical partition, 118
Cumulative distribution function, 236
Cumulative value, 29
CUS, 104

D

Deadline miss probability, 236, 243, 247, 253–254
Deadline miss ratio, 220
Deadline tolerance, 33
Deadlock, 136, 160
Deferrable Server, 65
Demand, 5, 26
Deng, 103, 107–108, 130
Domino effect, 23
D-over algorithm, 35
Dynamic priority, 238
Dynamic systems, 219

E

Earliest Eligible Deadline First, 73
EDF, 19, 23, 28
EEVDF, 73
Elastic model, 50, 230
Event-triggered activation, 4
Exceeding time, 3, 33
Execution overrun, 7
Execution time, 2
Execution time estimation, 230
Expectation, 238

F

Fair Share, 63
FC-EDF, 220
Feedback, 219
Feedback function, 225
Finishing time, 2
Firm deadline, 28
Firm task, 28
Fixed execution times, 233
Fixed priority, 238
Fixed priority algorithm, 245
Fluid flow system, 68

G

Generalized Processor Sharing, 67
Ghazalie, 141
Global scheduler, 97
GPS, 67
Graceful degradation, 28, 33
GRUB, 182
Guarantee, 31, 235

H

Hard guarantee, 62
Hard real-time system, 8
Hard reservations, 74
Haritsa, 25
HARTIK, 91
Heavy traffic, 245
Hierarchical scheduling, 98, 103
Hit Value Ratio, 36
Hyperperiod, 27

I

Imprecise computation, 39
Instantaneous load, 26

J

Job, 3
Job skipping, 39, 42

K

Koren, 35, 43
Kuo, 108, 132

L

Lag, 69
Lateness, 3

Laxity, 3
Layland, 43, 50
Least Supply Function, 117
Least supply function, 119
Legacy application, 97
Lehoczky, 135, 138, 203, 244–245
Linux/RK, 89
Lipari, 109, 116, 124, 141, 189
List of residuals, 110
Livny, 25
Liu, 43, 50, 103, 107–108, 130
Load, 5, 26
Local scheduler, 97
Locke, 24

M

Mach, 25
Mandatory parts, 250
Markov process, 237, 241, 245, 256
Maximum budget, 75
Maximum service time, 123
Metrics, 27
Minimum interarrival time, 5
MPEG, 10, 47
Multimedia, 10, 13, 47
Multi-threading, 95
Mutex semaphores, 133

N

Non real-time system, 13
Non-preemptive application, 104

O

Open system, 103
Optional parts, 250
Overload, 6, 23
Overload management, 20
Overrun, 7

P

Parameters assignment, 89
Partition class, 121
Partition delay, 119
PCP, 137
PDF, 236
Performance, 27
Performance degradation, 27
Period, 4
Period adaptation, 39, 47
Periodic resource abstraction, 123
PI, 226
PID, 220
PIP, 135
Polling Server, 65
Predictable application, 104
Priority Ceiling Protocol, 137
Priority Inheritance Protocol, 135
Priority inversion, 13, 134
Probabilistic deadline, 254
Probabilistic guarantee, 62, 253
Probabilistic schedulability analysis, 235
Probabilistic Time Demand Analysis, 239
Probability, 235
Probability distribution function, 236
Process, 95
Processor demand, 5, 26
Processor utilization, 6
Proportional Share, 69
PShED, 109

Q

QoS, 47, 195
QoS adaptation, 228
QoS dimensions, 195
QoS manager, 201, 228
QoS optimization, 197

Q-RAM, 195
Quality of Service, 47, 195
Queueing theory, 235, 243, 253, 256

R

Rajkumar, 135, 138
Ramamritham, 23
Random process, 254
Random variable, 236
Rate Monotonic, 16
Real-time guarantee, 235
Real-Time Mach, 89
Reclaiming mechanism, 32
Recovery strategy, 34
RED algorithm, 33
RED Linux, 92
Relative deadline, 2
Release time, 2
Reservation, 251, 254
Reservation period, 75
Residual laxity, 33
Resource constraints, 57–58, 140–142, 147, 150
Resource Kernels, 92
Resource management, 200
Resource partitioning, 117
Resource reclaiming, 21, 33, 35
Resource reservation, 63, 93
Resource set, 92
Response time, 2
Rialto, 91
RM, 7
Robust Scheduling, 31

S

Scheduling algorithm
  D-over, 35
  Robust Earliest Deadline, 33
Scheduling deadline, 78
Scheduling error, 223
Scheduling
  elastic, 50
  best effort, 31
  robust, 31
Schwan, 25
Semiperiodic task model, 238, 246
Server, 65, 78, 88
Server budget, 79–80, 83, 101, 108, 121, 145, 149, 158, 179, 181
Service adaptation, 39
Service levels, 220
Seto, 203
SFQ, 72
Sha, 135, 138, 142, 203
Shared memory, 133
Shared resources, 134, 141, 148, 159
SHaRK, 92
Shasha, 35, 43
Shin, 203
Slack time, 3
Soft deadline, 4
Soft real-time system, 10
Soft reservations, 74
Sporadic Server, 65
Spuri, 136
SRP, 138
Stack Resource Policy, 138
Stankovic, 23, 26, 33
Start Fair Queuing, 72
Start time, 2
Statistical Rate Monotonic, 248
Statistically distributed execution times, 235, 238, 241, 247
Stochastic analysis, 236
Stochastic guarantee, 236, 250
Stochastic process, 237
Stochastic Time Demand Analysis, 239
Supply function, 117, 119
System ceiling, 139, 147, 151, 156

T

Table-driven scheduler, 98
Tardiness, 3
Task, 2
Task models, 223
Task overloads, 229
Task
  aperiodic, 4
  firm, 4
  hard, 4
  non real time, 4
  periodic, 4
  soft, 4
  sporadic, 5
TBS, 65, 104
Temporal firewalling, 62
Temporal granularity, 100
Temporal isolation, 20, 62, 83
Temporal protection, 61, 236, 253
Thambidurai, 25
Thread, 95
Time Demand Analysis, 238
Time-triggered, 98
Time-triggered activation, 4
Total Bandwidth Server, 104, 65
Trivedi, 25

U

Underutilization, 235
Utility, 195
Utility function, 28, 197
Utilization, 6

V

Value, 28
Value density, 28
Variable computation time, 10
Variable execution time, 233
Virtual deadlines, 73
Virtual eligible time, 73
Virtual finish time, 70
Virtual processor, 97, 100
Virtual start time, 70
Virtual time, 70

W

WCET, 8, 231
Weighted Fair Queuing, 70
WFQ, 70
Work conserving, 70
Workload, 5–6, 26
Worst-case computation time, 8

Z

Zhou, 25
Zlokapa, 25