Scuola Superiore di Studi Universitari e Perfezionamento S. Anna

Classe di Scienze Sperimentali ed Applicate
Settore di Ingegneria

Tesi di Perfezionamento

Resource Reservation in Real-Time Systems

Candidato: Giuseppe Lipari

Collegio dei Docenti:

Prof. Paolo Ancilotti

Prof. Giorgio Buttazzo

Ing. Andrea Domenici
Acknowledgements

I would like to thank Giorgio and Sanjoy for their invaluable help and teaching and for the precious ideas and suggestions which produced so many research works; and, most important of all, for their friendship.

I would also like to thank Gerardo and Tonino for ten long years of sharing and friendship; my family, for their endless love and support; last (but absolutely first in my heart), Eleonora for her infinite love and patience.
## Contents

Abstract

1. Introduction to real-time systems
   1.1 Basic terminology ........................................ 1
   1.2 System model ............................................. 2
      1.2.1 Arrival patterns ..................................... 2
      1.2.2 Deadlines ........................................... 3
      1.2.3 Execution times ..................................... 3
      1.2.4 Other real-time constraints ......................... 4
      1.2.5 Applications ....................................... 5
   1.3 Scheduling algorithms .................................... 5
      1.3.1 Feasibility test ..................................... 6
   1.4 Hard Real-Time Systems ................................. 7
   1.5 Soft Real-Time Systems ................................ 8

2. The Resource Reservation Framework .......................... 10
   2.1 Open Systems ........................................... 10
   2.2 The Resource Reservation Framework .................... 10
      2.2.1 The QoS manager .................................... 11
      2.2.2 Run-Time support ................................... 11
   2.3 The server abstraction .................................. 12
   2.4 Proportional Share Resource Allocation .................. 13

3. The EDF scheduling algorithm ................................ 14
   3.1 Algorithm description ................................... 14
   3.2 Demand of a task set .................................... 15

4. Resource reservation for single-task applications ............ 17
   4.1 Resource Reservation Framework ......................... 17
   4.2 System model ........................................... 18
   4.3 The GRUB algorithm ..................................... 21
   4.4 Formal properties of Algorithm GRUB .................... 28
      4.4.1 Properties of Algorithm GRUB without reclamation ... 28
      4.4.2 Properties of GRUB with reclamation ................. 32
   4.4.3 Scheduling Hard Real-Time Tasks .................... 34
Contents

4.4.4 Scheduling Soft Real-Time Tasks ........................................... 35
4.5 Performance analysis ................................................................. 37
4.6 Experimental Evaluation of Algorithm GRUB .................................... 39
  4.6.1 Discussion ........................................................................... 40
  4.6.2 Simulation results ............................................................... 41
4.7 Conclusions ................................................................................. 45

5 Resource reservation for multi-task applications ................................ 46
  5.1 Introduction ............................................................................... 46
  5.2 Problem description .................................................................. 47
  5.3 Previous work ........................................................................... 48
  5.4 Definitions ............................................................................... 49
  5.5 The BSS Algorithm ................................................................. 50
    5.5.1 Budget calculation ............................................................. 52
    5.5.2 Adding a new element ....................................................... 53
    5.5.3 Updating the list ............................................................... 54
    5.5.4 Deleting elements ............................................................ 54
    5.5.5 An example ....................................................................... 55
  5.6 Support for different scheduling policies ...................................... 56
    5.6.1 Rules to postpone deadlines .............................................. 58
  5.7 Formal analysis ........................................................................ 58
    5.7.1 Properties of the BSS algorithm ........................................ 58
    5.7.2 Job Transformation ........................................................... 59
    5.7.3 Bandwidth Isolation Property .......................................... 60
    5.7.4 The Hard Schedulability Property .................................... 63
    5.7.5 Rate Monotonic Schedulability condition ......................... 66
  5.8 Improving complexity bounds .................................................... 67
    5.8.1 Inserting in the tree .......................................................... 69
    5.8.2 Updating the tree ............................................................. 71
    5.8.3 Deleting elements ............................................................ 73
    5.8.4 Complexity ...................................................................... 73
  5.9 Conclusions and future work ..................................................... 73
List of Figures

1.1 An example of infeasible task set ..................................................... 5
3.1 An example of scheduling of a periodic task set with the EDF algorithm. 14
4.1 State transition diagram ................................................................. 23
4.2 PseudoCode for Algorithm GRUB .................................................... 25
4.3 Scheduling without bandwidth reclamation ....................................... 26
4.4 Scheduling with bandwidth reclamation .......................................... 28
4.5 Example of job transformation. a) the jobs of task \( \tau_1 \) with their original deadlines  b) the sub-jobs generated by server \( S_1 \) ...................... 30
4.6 Job transformation in case of reclamation ........................................ 33
4.7 Preemption experiment – one. The total number of preemptions for a particular average load is seen to be higher in a CBS-generated schedule than in one generated by Algorithm GRUB ..................................................... 42
4.8 Preemption experiment – two. The number of preemptions per task for a particular average load is seen to be higher in a CBS-generated schedule than in one generated by Algorithm GRUB .......................... 43
4.9 Fairness experiments ......................................................................... 44
5.1 A real-time multi-task application scheduled by Algorithm GRUB .......... 47
5.2 System Architecture .......................................................................... 50
5.3 Computation of \( B_i \) ..................................................................... 53
5.4 Two applications scheduled by the BSS algorithm ............................... 55
5.5 An example of schedule produced by the BSS algorithm: the two tasks in application \( A \) are scheduled by Rate Monotonic .................................................. 56
5.6 Two task executed on a dedicated processor with speed 0.5 by the Rate Monotonic scheduling algorithm ......................................................... 66
5.7 An application scheduled by the BSS algorithm on a shared processor with speed 1, and the Rate Monotonic local scheduling strategy ............... 67
5.8 Different representations of the same data structure: a) Linear list; b) Incremental AVL tree ................................................................. 69
5.9 Balancing the left subtree .................................................................... 70
5.10 Left and right rotations on a sub-tree ............................................... 71
5.11 Right rotation .................................................................................... 72
5.12 Left rotation ....................................................................................... 72
List of Figures

5.13 The update() and adjust() functions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
Abstract

The argument of this thesis is the study and implementation of scheduling algorithms for providing real-time guarantees to hard and soft real-time systems in general purpose computer systems.

Many real-world applications have some sort of non-critical timing requirements: for example, video and audio streaming, multimedia playing, transmission protocols, and much more. In these systems, timing requirements are expressed in terms of Quality of Service (QoS). It is important to provide the final user the best possible Quality of Service and at the same time to optimize the resource usage.

This work is focused on Open Systems. An Open System is a general purpose computer system in which it is not possible to know a-priori how many applications need to be executed and the characteristics thereof. A general-purpose Operating System can be considered an Open System.

One of the problems that arises when using a general purpose operating systems for executing hard and soft real-time systems is that they are not able to provide any kind temporal guarantee. In this thesis, two algorithms are proposed to be used in Open Systems for providing guarantees to real-time applications. They have been developed in the context of a more general framework, the Resource Reservation Framework, which is based on the concept of reserving each application a fraction of the processor capacity. In this way, each application is isolated from misbehaviors of the others and it is possible to analyze the performance of each one of them independently, as if it were executed alone on a dedicated processor. The methodology proposed here is rather general and takes into account:

- single threaded and multi-threaded applications;
- soft real-time as well as hard real-time systems;
- time-driven and event-driven models of execution.

In chapter 1, a brief introduction to real-time system is given and the notation used throughout the thesis is introduced. In chapter 2, the basis of the Resource Reservation Framework is introduced. In chapter 3 we will recall the Earliest Deadline First scheduling algorithm, and introduce the Processor Demand Approach. In chapter 4 Algorithm GRUB is presented, to provide isolation and performances guarantees to hard
Abstract

and soft real-time applications consisting of one task. Finally, in chapter 5, the BSS a-algorithm is presented, that extends the kind of guarantees provided by Algorithm GRUB to multi-task applications.
1 Introduction to real-time systems

In this chapter, an overview of the research on Real-Time system is given. The basic terminology, the notation and some definitions are given in Section 1.2. In Sections 1.4 and 1.5, the main differences between hard and soft real-time application are discussed. This introduction is rather informal: for more complete and formal introductions on Real-Time Systems see [10, 50].

1.1 Basic terminology

A “Real-Time System” is a set of activities with timing requirements. The correctness of a Real-Time System depends not only on the correctness of the results but also on the time at which they are produced.

Real-Time systems are strongly characterized by the time: real-time does not mean fast but on-time. The “fastness” of the real-time system must be related to the “fastness” of the world to which the systems reacts.

An important concept in real-time systems is predictability: it is fundamental to be able to predict the temporal behavior of the system assuming the knowledge of the timing characteristics of all the components in the systems. For example, in mission-critical real-time systems it is of paramount importance to be able to bound the completion time of each activity.

A real-time system consists of a set of different activities (tasks) to be performed concurrently on a [set of] shared resource[s]. For example:

- a set of processes to be executed on a single processor computer system;
- a set of flows of data packets to be transmitted over a shared network link;
- a set of transactions to be performed on a database;
- etc.

In this work, we are only interested in computational resources (processors): we will not consider network links or other kinds of resources. The problem is in the limitation of the available resources: with an unlimited number of CPU with unbounded speed, there is no problem in building real-time systems. Additional difficulties are given by the fact that often those activities are not independent, but must synchronize and communicate each other.
1 Introduction to real-time systems

In the case of only one task, the solution is to use a CPU that is fast enough. With many concurrent tasks, it is also important to decide the order in which they are performed. This order is called “schedule”; formally, a “schedule” is a function that for every instant of time selects which task executes on each processor.

A real-time task is assigned a number of parameters that capture its characteristics and its timing requirements. One example of such requirements (maybe the most common) is the tasks’ deadline: it is the time by which the task must finish for the system to be correct. In Sections 1.4 and 1.5, the different meanings of this parameter in different contexts are discussed.

Given a set of tasks with certain parameters to be scheduled on one or more processors, a “scheduling algorithm” produces a schedule that satisfies certain properties. A schedule is “feasible” if every task respects its timing requirements. A real-time system is “schedulable” if there exists a feasible schedule for it.

The goal of any scheduling algorithm for real-time system is to schedule a set of tasks such that every task respects timing requirements (if possible).

1.2 System model

In this section we introduce the basic terminology, assumption and notation used throughout the thesis.

Definition 1.1 A task is a finite or infinite sequence of requests for execution on a shared resource (e.g. the CPU). The i-th task in the system will be denoted by \( \tau_i \).

Definition 1.2 A task request is also called job or instance: the j-th job of task \( \tau_i \) will be indicated with \( J_{i,j} \).

A job can be thought as a portion of program to be executed sequentially. When a task requests for execution, we say that a task job is released or that the job arrives in the system.

Definition 1.3 A task job is characterized by at least an arrival time and computation time. The arrival time of job \( J_{i,j} \) will be indicated by \( a_{i,j} \) and its computation time by \( c_{i,j} \). A real-time job has at least an additional parameter, the absolute deadline \( d_{i,j} \).

1.2.1 Arrival patterns

If a task has only one job, then it is a one-shot task. When the job finishes, the task is terminated. Each task \( \tau_i \) consisting of many jobs can be characterized by an arrival pattern, i.e. by the way the jobs are released. According to this classification, a task can be:

Periodic : if the arrivals are separated by a constant interval of time, called “period” and denoted by \( T_i \);

\[ a_{i,j+1} = a_{i,j} + T_i \]
1 Introduction to real-time systems

**Sporadic**: if the arrivals are separated by variable intervals of time with a lower bound, called *minimum inter-arrival time*; the minimum inter-arrival time is also denoted with $T_i$:

$$a_{i,j+1} \geq a_{i,j} + T_i$$

**Aperiodic**: if a lower bound is not known on the inter-arrival times.

Periodic tasks are also referred as *time-driven tasks* and sporadic and aperiodic tasks are called *event-driven tasks*. This is due to the way they are implemented in an actual real-time system: periodic tasks are activated by a hardware or software timer; sporadic and aperiodic tasks are activated by external or internal events, like interrupts or direct activations by other tasks. Parameter $T_i$ is very important because it gives a bound on the frequency of the task arrivals and hence on the load of the system.

The arrival time of the first job of a periodic task is also referred as the task’s *phase* and it is indicated with $\phi_i$. If all the tasks arrive at time 0, the task set is called “synchronous”. Phases play an important role in the analysis of the schedulability of periodic task sets; see [9] for more details.

1.2.2 Deadlines

By definition, a real-time task has timing requirements. The most common constraint is the *deadline*: the *absolute deadline* of a job is the instant of time by which the job must finish; the *relative deadline* of a job is the interval of time between the arrival time and the absolute deadline. Typically, a relative deadline is assigned to the task, and the absolute deadline for each job is computed as the arrival time plus the task’s relative deadline. A task without deadline is a non-real-time task.

In this thesis, we will use the following notation:

- $d_{i,j}$ will indicate the absolute deadline for job $J_{i,j}$.
- $D_i$ will indicate the relative deadline of task $\tau_i$ ($d_{i,j} = a_{i,j} + D_i$). Often, a periodic task is assigned a relative deadline equal to the period: $D_i = T_i$, $d_{i,j} = a_{i,j+1}$.
- Sometimes, for brevity, $d_i(t)$ will indicate the absolute deadline of the job of task $\tau_i$ that is active at time $t$; if at time $t$ the active job of $\tau_i$ is $J_{i,k}$, then $d_i(t) = d_{i,k}$.

When it is possible, we will simply write $d_i$ for $d_i(t)$.

1.2.3 Execution times

Jobs are also characterized by an *execution time* $c_{i,j}$, that is the net time the job will need to execute until completion on a dedicated processor, without considering synchronizations with other jobs and context switches. The execution time is needed for analyzing the schedulability of a task set: to see whether is possible for a job to finish within its deadline we need to know how long it will take to finish. This execution time is very difficult to measure, and often it is only possible to give a rough estimate. It depends on many things:
1 Introduction to real-time systems

- speed of the processor on which the job executes;
- the algorithm that the job implements;
- input data;
- duration of the kernel primitives;
- presence of cache, DMA, and other hardware specific features.

For this reason it is practically impossible to measure exactly the duration of each single job of a task. Therefore, a 
worst case execution time (WCET) is used for the whole task: it is a superior bound on the execution time of any task job that takes into account all the previous issues.

**Definition 1.4** The Worst Case Execution Time (WCET) of task $\tau_i$ is the maximum execution time of all the task jobs $J_{i,j}$ and it is denoted with $C_i$.

**Definition 1.5** The utilization of a periodic task $\tau_i$, denoted by $U_i$, is defined as the WCET divided by the period. The utilization of a sporadic task $\tau_i$ is the WCET divided by the minimum inter-arrival time:

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

The utilization gives a measure of the fraction of the processor execution time that the task needs in order to be executed without overrun.

**Definition 1.6** The processor is overloaded if the sum of the task utilization is greater than 1.

**Example 1.1** Consider the task set $T = \{\tau_1, \tau_2\}$, with $C_1 = 3, T_1 = 6, C_2 = 5, T_2 = 9$. Since $\sum_{i=1}^{2} U_i > 1$, the system is overloaded: it is easy to show that no scheduling algorithm can guarantee the execution of the task set without overrun. In Figure 1.1 is given an example of schedule produced by the Earliest Deadline First algorithm for task set $T$: at time $t = 18$ task $\tau_1$ misses its deadline. The Earliest Deadline First (EDF) algorithm will be presented in Chapter 3.

1.2.4 Other real-time constraints

If tasks do not interact in any way, they are independent tasks. Sometimes, tasks access mutually exclusive resources (other than processor): we will indicate these tasks as "resource constrained tasks"\(^1\). These constraints are not properly real-time constraints; however, in order to avoid deadlock, chained blocking and a particular phenomenon

---

\(^1\)This term is improperly used, because, if we consider the CPUs as a resource, then every task is resource constrained.
called priority inversion, several concurrency control algorithms have been devised for hard real-time systems [44, 13, 7, 23].

Another kind of timing requirement for a task is the jitter. It is defined as the difference between the time of two successive events in the system. For example, the output jitter is the difference between the finishing time of two successive jobs of a task; in some system consisting of periodic tasks, the requirement is to bound the output jitter to be equal to the period, or to be in the range \([T_i - \eta, T_i + \eta]\), for some \(\eta\). The input jitter is the difference between two successive start times of a task.

### 1.2.5 Applications

**Definition 1.7** The term application denotes a concurrent program, which consists of a set of dependent or independent cooperating tasks.

A synonym for application is process, used mainly in the UNIX terminology. For the purposes of this work, an application is simply a task or a set of tasks. A real-time application is a set of real-time tasks. We are interested in studying the schedulability of real-time applications when they are executed together with other application in a Open System.

In this thesis, applications are denoted by calligraphic letters, \(A, B, \ldots, Z\). The \(j\)-th task of application \(A\) is indicated with \(\tau_j^A\). The \(k\)-th job of task \(\tau_j^A\) is indicated with \(J_{jk}^A\).

Since every task is set of jobs, we can consider an application simply as a set of jobs. Hence, in order to simplify the notation, some time we will enumerate the jobs of an application in order of arrival time. Thus, we will refer to the \(k\)-th job of application \(A\) as \(J_{k}^A\), without specifying which task it belongs to.

### 1.3 Scheduling algorithms

Scheduling algorithms can be classified in:
off-line: the schedule is pre-computed off line. It is recorded in a table: an on-line
dispatcher choose which task to execute at the current time with a table look-up.
Of course, such method can be used if the main parameters of the tasks are known:
for example, the arrival time of every job and the worst case computation time. For
this reason, these algorithms are not well suited for sporadic or aperiodic task, and
in general for event-driven systems. These systems are also referred as “statically
scheduled” systems.

on-line: each task is assigned a priority, according to a certain algorithm. The on-line
dispatcher selects the task with the highest priority to execute.

On-line scheduling algorithms can be divided further in:

static: (or fixed priority) the priorities are statically assigned to tasks and do not change
during execution (not to be confused with off-line schedulers);

dynamic: the priorities are computed dynamically; for example, a different priority is
assigned to each job of a task.

Many scheduling algorithms have been presented in the real-time literature, with
different characteristics. We recall here maybe the two most important ones.

The Rate Monotonic (RM) scheduling algorithm is a static on-line scheduler which
assigns priorities which are inversely proportional to the periods of the tasks: the longer
the period, the smaller the priority. For tasks with relative deadline smaller than pe-
riod, the Deadline Monotonic (DM) scheduler assigns priorities which are inversely
proportional to the relative deadline. Liu and Layland [32] proved that Rate Monotonic
is optimal among all the static on-line scheduling algorithms, in the sense that if a set of
periodic or sporadic hard real-time tasks can be scheduled by a static on-line algorithm,
than it can be scheduled by Rate Monotonic.

The Earliest Deadline First (EDF) scheduling algorithm assigns priorities inversely
proportional to the absolute deadlines of the jobs: the closer the deadline, the higher
the priority. Again Liu and Layland [32] and Dertouzos [18] have shown that for any
set of independent tasks (with deadlines equal to periods or not, periodic, sporadic or
aperiodic), EDF is an optimal uni-processor scheduling algorithm, in the sense that if the
task set is schedulable (it exists a feasible schedule for it) then it exists a feasible EDF
schedule. Since a static priority assignment is a sub-case of the more general dynamic
priority assignment, it follows that every task set that is schedulable by RM is also schedulable by EDF.

In Chapter 3 the EDF scheduling algorithm is presented in detail.

1.3.1 Feasibility test

It is important to analyze the correctness of the schedules produced by a given scheduling
algorithm. Hence, a scheduling algorithm is associated with a “feasibility test”, i.e. an
algorithm that, given a task set with certain parameters, tells if the schedule produced
by the scheduling algorithm is feasible. The feasibility test can be sufficient, or necessary
1 Introduction to real-time systems

and sufficient. In the first case, if the test gives a negative result, the task set may still be feasible.

The feasibility test depends also on the parameters of the task set and on the available informations. For example there are different feasibility tests for periodic task sets and for sporadic task sets. In this work, we will restrict our attention on the EDF scheduling algorithm and on some interesting variation: in particular, in Section 3.2 we will present a necessary and sufficient feasibility test for a set of independent real-time tasks.

1.4 Hard Real-Time Systems

In these systems, it is imperative that each task must produce a (correct) result before its deadline, otherwise something catastrophic might happen. Examples are especially found in the control system area:

- flight control program for aircraft, such as a fly-by-wire system;
- control for automobile engine, brakes, etc.;
- control for nuclear and chemical plants;
- robot control systems;
- etc.

Timing correctness requirements in a hard real-time system arise because of the physical impact of the controlling system activities upon its environment. Since hard real-time systems are critical, they must be designed and implemented accurately, and a precise timing analysis must be performed. In particular, after the system has been designed and partially implemented, an accurate timing analysis is performed on each task to compute the worst case execution time. Given these values and the scheduling algorithm, it is possible to check the feasibility of the system. If the system is feasible, then the process has finished. The programmer only needs to test the correctness of the implementation. If the system is not feasible, something need to be changed in a previous step. Several thing can be done: for example, it is possible to buy a faster hardware; or some task can be slowed down (relaxing constraints); or some task can be implemented in a different way, reducing its worst case computation time; or some other dedicated processor can be introduced; etc. What to do depends on which task has failed and on the system criticality.

The most difficult part is often the estimation of the worst case computation time for every task; it depends on the hardware architecture used to implement the system. However, in a critical system, often a dedicated hardware is used. The software is especially designed to work on that hardware, and it is very unlikely that it will run on a different platform.

The development process of a critical application is expensive. In fact, the criticality of the application impose a strict and rigorous process: the output of each step needs to
be checked with care to avoid potential misbehaviors. Predictability is necessary because an unexpected event could result in great loss of money and in some case in a danger for human lives. In particular, is of paramount importance to check feasibility of the system, since a late task could compromise the correctness of the entire system. There are several famous examples of critical systems that have failed due to timing issues, and in each case the loss could have been avoided with a more careful analysis.

Another great problem is testing. An error may be due to a particular task interleaving that is vary rare in the system life-cycle. And in such critical systems, one missed deadline could lead to some catastrophic fault. A similar problem happened on the Mars Lander: a particular task interleaving, never revealed at testing time, caused the system crash and the loss of the lander!

Hence, it is very important to use algorithms and methodologies that guarantee certain properties of the system so to reduce potential misbehavior and facilitate testing. For example, using the Stack Resource Policy [7] to handle task synchronization on shared mutually exclusive resources, avoids deadlock and reduces priority inversion.

### 1.5 Soft Real-Time Systems

Soft real-time systems have non-critical timing requirements. This means that nothing catastrophic happens if some of these requirements are not respected. There are a lot of examples of these system. Probably, the most popular in these days are multimedia applications: teleconference, video/audio players, Multimedia CDROMs, etc. .

All these systems have some sort of temporal constraints. For example, for a video player it is important to play at a minimum frame rate to ensure a certain level of quality. Another (non-temporal) constraint could be the image quality of the video, expressed for example as frame-size in pixel. Clearly, these constraints are not critical: if the video cannot be played within the specified constraints, nothing catastrophic happens. However, a low frame-rate, or a low image quality are undesired behaviors and should be avoided whenever possible, in order to maximize both resource utilization and user satisfaction.

The methodologies used to develop hard real-time application are not suitable for soft real-time application. There are many substantial differences between a hard real-time and a soft real-time system:

- a soft real-time application is expected to run on different hardware platforms. In some case, these platform are not even present at development time!
- the input to the program cannot be completely characterized; hence it is very difficult to give estimations on the duration of the tasks' jobs; for example, video streams have very irregular data requirements over time;
- a soft real-time application is expected to be run in a shared system together with other applications. Hence, we cannot assume exclusive access to the hardware resources.


1 Introduction to real-time systems

- nothing catastrophic happens if some temporal constraint is not respected; however, the constraints should be respected whenever possible, and if not possible, the user should be informed.

If we don’t know on which platform the application will run, and we don’t have precise information on the input data that the application will process, it is impossible to design the system for the worst case. Hence, a different methodology should be introduced.

The performance of a soft real-time application should be measured considering the quality of service it delivers to the end-users. Unfortunately, there is not an unique well-defined and universally accepted definition of quality of service, because it depends strongly on the types of application we are considering: video streaming has different characteristics and requirements from audio-streaming! Moreover, these application are expected to be run on widely used, general purpose operating systems (like Windows or Linux) and such systems offer no real-time support. Hence, there is not an assessed and well defined methodology for treating these applications, and often they are simply implemented with best-effort techniques.

The work presented in this thesis especially addresses the problem of scheduling soft and hard real-time applications in the same systems, providing isolation and real-time guarantees.
2 The Resource Reservation Framework

2.1 Open Systems

An Open System is a general purpose computer system in which it is not possible to know a-priori how many applications need to be executed and the characteristics thereof. Typically, in an Open System, applications with different levels of Quality of Service coexists, from multi-media applications to interactive non-real-time applications. A general-purpose Operating System can be considered an Open System, since the system itself has no information on the temporal characteristics of the application that is executing.

From the discussion made in Section 1.5, it follows that an open system is the target platform for soft real-time applications. In facts, we cannot think to build a dedicated platform and apply hard real-time techniques for executing multi-media applications, for they are too costly.

However, the problem that arise when using a general purpose operating systems for executing soft real-time applications is that it is not possible to provide any kind of temporal guarantee. In facts, in traditional operating systems scheduling decisions are not based on real-time parameters, like deadlines or periods. As a consequence, applications may have an unpredictable timing behavior and experience discontinuity under peak load situations.

2.2 The Resource Reservation Framework

The algorithms proposed in this work are based on a general framework, the Resource Reservation Framework. In this framework, an application that arrives in the system requires a certain level of QoS. An acceptance test is run: if, given the actual system load, the required level of QoS can be guaranteed, the application is accepted in the system and is assigned a certain fraction of the system resources.

Hence, the essence of Resource Reservation is a contract established between the system and each served application: if the application passes the acceptance test, a contract is signed between the application and the system and the application is executed guaranteeing the desired level of service. In the negative case, the application is rejected. The algorithm that negotiate this contract with the application is often referred as the QoS Manager.

Since no a-priori assumption is made on the characteristics of the applications, the Resource Reservation Framework is particularly suitable for Open Systems.
2.2.1 The QoS manager

The admission test is probably the most difficult part to implement. In fact, when an (unknown) application arrives in the system, we don’t know how much resource it is going to use. Recall the discussion on soft real-time systems made in section 1.5: the computation time of a soft real-time task is very difficult to estimate because it depends on many different factors as the hardware platform and the input data which cannot be easily characterized.

When a mapping between the desired QoS and the amount of resources needed by the application is known in advance, the resource allocation problem can be formulated using dynamic programming techniques, and the QoS manager can solve it in order to compute the optimal resource assignment [42, 28]. Other similar QoS models (although not based on strong mathematical bases) have been proposed in [5] and [11].

However, such scenario is not common. For this reason, often a feedback adaptive algorithm is needed: we can do a tentative admission of the application in order to estimate its resource requirements and see if the desired QoS can be achieved. If so, the application is definitively admitted. Also, we can “store” informations on an application in order to make a more precise estimation if the application is run again in the future.

Some researcher [15] claims that such an adaptation can be achieved without any specific support from the operating system, whereas others tend to provide an explicit support in the scheduler, through an entity referred as the QoS manager [38, 19, 37, 14].

In [38, 19, 37], a solution based on resource reservations and on a global QoS manager is presented. In particular, the QoS manager detects reservation overruns and adapts the tasks’ periods and computation times in order to reduce the overruns. However, it is not clear how the QoS manager can control the behavior of each single application.

The use of feedback control schemes in real-time scheduling is emerging in recent researches: for example, in [49, 33], the number of missed deadlines is used as a feedback variable to control the system workload with a proper admission control policy. The proposed method, however, does not provide isolation among tasks; thus, computational demands can hardly be controlled for each individual task.

A feedback mechanism for adapting the scheduler parameters is presented in [1]. In this paper, the feedback is used to adapt the fraction of CPU bandwidth reserved to each task. The bandwidth adaptation mechanism is global and weights are used to assign a different importance to each task, independently from its demanded resources.

2.2.2 Run-Time support

Once the resource requirements of the application are estimated (in one way or the other), they are converted into scheduling parameters that are then passed to the scheduler, in order to properly handle the application.

For what concerns the run-time support, the problem of providing a proper operating system structure has been addressed in [41, 22, 43, 30].

In [41], the concept of Resource Kernel is presented: a Resource Kernel (RK) introduce abstractions like Resource Set and Reservation that permit to reserve in advance
the proper amount of each resource to each application. Resources are processors, disks, network interfaces, and so on. A reservation is described by three parameters \((C_i, D_i, T_i)\), with the meaning that an amount \(C_i\) of the resource will be available at each period \(T_i\) within a deadline \(D_i\). While a RK is designed for \(\mu\)-kernel systems (such as RT-Mach [55]), it can be easily implemented in a monolithic kernel.

The Nemesis [43, 30] system, deriving from Nemo [22], propose a completely different OS concept, based on a vertical structure. In this model, resources are demultiplexed by the kernel at a very low level, and are then directly managed by the application. In this way performing resource accounting becomes very easy, and a very precise and fine-granularity resource allocation is possible.

The results presented in this thesis go in the direction of improving the run-time support, in particular the scheduling algorithm. The scheduler is an important part of the system, since it can enforce properties like isolation among applications and real-time guarantee to each individual application.

### 2.3 The server abstraction

The algorithms proposed in the next chapters are based on an abstraction called server. This abstraction was first introduced to handle aperiodic non-real-time tasks in hard real-time systems. In that context, the goal is to enhance the response time of aperiodic non-real-time tasks without jeopardizing the guarantee on hard real-time tasks. Lot of work has been done on servers for aperiodic tasks, both in fixed priority [29, 45, 53, 54] and dynamic priority environments [48, 46, 20, 2, 12].

The server abstraction is mainly based on reserving each application a fraction of the processor capacity: the server will ensure that the application does not use more than reserved. Each server is treated as a hard real-time task, so that it is possible to analyze the feasibility of the system using the classical real-time theory. In the following, an overview of the algorithms that most influenced this research are summarized.

In [48], Spuri and Buttazzo propose the Total Bandwidth Server: each time an aperiodic task arrives in the system, a deadline is computed for the server based on the WCET of the aperiodic task, and the server is inserted in a global EDF queue together with the hard real-time tasks. The algorithm is very simple and easy to implement and has a good performance compared to more complex algorithms; however it does not provide isolation, as a misbehaving aperiodic task that tries to execute more than declared could jeopardize the schedulability of the hard real-time tasks.

In [27], Kaneko et al. propose a scheme based on a periodic process (the multimedia server) dedicated to the service of all multimedia requests. This allows to nicely integrate multimedia tasks together with hard real-time tasks; however, being the server only one, it is not easy to control the QoS of each application.

In [2], Abeni and Buttazzo presented the Constant Bandwidth Server (CBS), which is able to schedule hard and soft real-time tasks together, guaranteeing the deadlines of the formers, and a minimum service to the latter. The algorithm is also able to isolate the behavior of each task from the others, even in the case in which the parameters
(WCET and minimum inter-arrival time) are not known a priori. However, in this work, only single-task applications are considered. The CBS algorithm has been improved by Algorithm GRUB which is presented in Chapter 4.

In [17], Liu and Deng describe a scheduling hierarchy which allows hard real-time, soft real-time, and non real-time applications to coexist in the same system, and to be created dynamically. According to this approach, which uses the EDF scheduling algorithm as a low-level scheduler, each application is handled by a dedicated Total Bandwidth Server. This solution can be used to isolate the effects of overloads at the application level, but requires the knowledge of the WCET even for soft and non real-time tasks. In Chapter 5 the BSS algorithm is presented which resolves some of the problems of the previous approach.

2.4 Proportional Share Resource Allocation

Recently, some scheduling algorithms have been proposed [52, 21, 24, 51, 56] to mix some form of real-time support with a notion of fairness. They are based on the Proportional Share Resource Allocation approach, first introduced in the field of network scheduling by Parekh and Gallager in [39] and by Demers [16].

In these works, the isolation is achieved by allocating the processor to applications in a weighted fair manner. The goal is to schedule each application approximately as it were executing alone on a slower dedicated processor. The time-line is divided in scheduling quanta which are allocated to applications in a fair manner. The smaller the quantum, the better is the approximation of the dedicated processor.

These methods are effective in scheduling soft real-time tasks, and nicely integrate with the existing software: for example, in [21] the authors realized a multilevel scheduling algorithm and it is possible to select the higher-level scheduling policy that best meets the application requirements.

However, these approaches may be unsuitable for guaranteeing applications consisting of hard real-time tasks, because these methods are quite sensitive to the length of quantum. In fact, if we want to be able to precisely guarantee hard real-time applications, we have to closely approximate a dedicated processor. Thus, the quantum must be as small as it is possible. However, a small quantum leads to a large amount of unnecessary context switches among applications that result in a large system overhead. Instead, it would be desirable to minimize the number of context switches in the system.

In [3], it has been shown that the server approach is substantially equivalent to the proportional share approach. However, the algorithms that are based on the server approach can better control the number of context switches in the systems as they use two parameters for each application, against one single parameter used by the Proportional Share based algorithms.

Also, in Chapter 5, we will show a problem that arises when scheduling a multi-task application with a Proportional Share based scheduler.
3 The EDF scheduling algorithm

The Earliest Deadline First (EDF) algorithm is an on-line dynamic scheduling algorithm that assigns each job a priority inversely proportional to its absolute deadline. In this Chapter, we present some results concerning the schedulability of an application with the EDF algorithm.

3.1 Algorithm description

The Earliest Deadline First (EDF) assigns each job a priority inversely proportional to its absolute deadline. The closer the deadline, the higher the priority. Ties are broken arbitrarily: if two or more jobs have the same absolute deadline, it does not matter which one is scheduled first.

Example 3.1

An example of EDF scheduling is shown in Figure 3.1. Comparing this schedule with the one produced by the Rate Monotonic Algorithm, we can see that at time 10 the Rate Monotonic algorithm would have chosen task $\tau_2$ instead of $\tau_3$ because $\tau_2$ has a smaller period. Same thing happens at time 18. Notice also that the task set is feasible with the EDF algorithm whereas is not feasible with the Rate Monotonic algorithm.

The EDF scheduling algorithm has been proven optimal for uni-processor scheduling in the sense that if a feasible schedule exists, then a feasible EDF schedule exists. This result was first shown by Liu and Layland [32] for periodic task sets. It was then proved

![Figure 3.1: An example of scheduling of a periodic task set with the EDF algorithm.](image-url)
by Dertouzos in [18] and Mok [35] for general task sets.

The proof presented here is based on a general methodology that has emerged in the last years: the Processor Demand Approach. This methodology was applied first by Baruah in [9, 8] and by Jeffay in [25], for providing schedulability conditions for periodic and sporadic task sets. Recently, it has been formalized by Buttazzo in [10]. Although we are not strictly interested in schedulability conditions for the EDF algorithms, we propose the same methodology here in the context of the Resource Reservation Approach.

### 3.2 Demand of a task set

**Definition 3.1** The demand of a task $\tau_i$ over an interval of time $[a, b]$ is defined as the sum of the computation times of all the jobs of $\tau_i$ with arrivals and deadlines within $[a, b]$. In formula:

$$D_i(a, b) = \sum_{a_{ij} \geq a} c_{ij}$$

**Definition 3.2** The demand of a task set $\mathcal{A}$ over an interval $[a, b]$ is defined as the sum of the demand of all the tasks in $\mathcal{A}$.

$$D^\mathcal{A}(a, b) = \sum_{\tau_i \in \mathcal{A}} D_i(a, b)$$

**Definition 3.3** The maximum demand of task set over an interval of length $L$ is the sum of the maximum demands of the tasks over all possible intervals of length $L$.

The concept of demand of a task set over an interval $[a, b]$ has a straightforward interpretation: it represents the amount of “work” that the processor must execute in interval $[a, b]$ so that every job can meet its deadline.

**Example 3.2** Consider again example 1.1: in interval $[6, 24]$ the demand of the task set is 13; however, the maximum demand of the task set over an interval of length 18 is 19 which is greater than the length of the interval.

Considering example 3.1, it is easy to see that there is not any interval in which the demand of the task set is greater of the length of the interval. In this following we will show that this is a necessary and sufficient condition for the schedulability of a set of independent jobs with the EDF algorithm.

**Theorem 3.1** A necessary condition for a task set $\mathcal{A}$ to be schedulable by a generic scheduling algorithm is that for every interval, the demand of the task set does not exceed the length of the interval. In formula:

$$\forall a, b \quad a \leq b \quad D^\mathcal{A}(a, b) \leq (b - a)$$

---

15
3 The EDF scheduling algorithm

Proof.
By contradiction. Suppose that there exists some interval \([\tau, \overline{b}]\) in which Equation (3.3) does not hold. It means that the amount of “work” that the processor must execute in \([\tau, \overline{b}]\) exceeds the length of the interval. Since no job can begin execution before its arrival time, at least one job whose arrival time and deadline are within interval \([\tau, \overline{b}]\), will finish after \(\overline{b}\), and hence after its deadline. □

Theorem 3.2 A task set \(A\) is schedulable by EDF if and only if:
\[
\forall a, b \quad a \leq b \quad D^A(a, b) \leq (b - a).
\] (3.4)

Proof.
The necessary part descends directly from the previous theorem. Now we will prove the sufficient part by contradiction.

Suppose that a job \(J_m\) misses its deadline at time \(y\) and that condition (3.3) holds. It means that \(J_m\) has a deadline \(d_m = y\) and that has not yet completed by time \(y\). Let us define \(x\) as the last instant of time such that:

- \(x\) is the arrival time of a job with deadline less than or equal to \(y\);
- all the jobs that arrive before \(x\) with deadline less than or equal to \(y\) have completed by \(x\).

An instant \(x\) defined as above always exists: time 0 (or the first time at which a job is activated) is one such instant.

First, let us show that there cannot exist a job \(J_L\) with deadline greater than \(y\) that executes in \([x, y]\). In fact, if such job exists and executes at time \(w \in [x, y]\), it follows from the EDF rule that all the jobs arriving before \(w\) with deadline less than or equal to \(y\) have completed by \(w\). This is in contradiction with the definition of \(x\). The same reasoning can be applied to show that there cannot exists a time in interval \([x, y]\) in which the processor is idle.

Hence, in interval \([x, y]\) the processor is never idle and executes only jobs with arrival times and deadlines in \([x, y]\). Let us define as \(S\) the set of these jobs. Since a job \(J_m \in S\) has not yet completed by \(y\), it follows that the sum of the computation times of the jobs in \(S\) is greater than \((y - x)\):
\[
y - x < D^S(x, y) \leq D^A(x, y).
\]

Thus, we have found an interval \([x, y]\) for which condition 3.4 does not hold, against the hypothesis. □

Observe that the optimality of the EDF algorithm descends directly from theorem 3.2: if a task set can be scheduled on a single processor, then it is schedulable by the EDF algorithm.
4 Resource reservation for single-task applications

A framework\(^1\) for scheduling a number of different real-time applications on a single shared preemptable processor is proposed in this chapter, such that each application seems to be executing on a slower dedicated processor. A tradeoff is identified and evaluated between how precise a notion of real time (as measured by the granularity of its clock) an application needs supported on the one hand, and the added context-switch costs imposed by our scheduling framework on the other.

4.1 Resource Reservation Framework

As introduced in section 1.2.5, an application is a set of cooperating tasks. Multiprogrammed computer systems are expected to execute several applications concurrently. When some (or all) of these are real-time applications, it is important that the underlying scheduling policy possess the following features:

1. each individual application should be guaranteed a certain level of service, and

2. there should be effective isolation among applications – an errant application should not be able to cause an unacceptable degradation in performance in other – well-behaved – applications.

A popular conceptual framework for modeling the behaviour of such application systems is to associate a server with each application, with each server characterized by certain parameters which specify exactly its performance expectations. The goal of the system-wide scheduler is then to schedule run-time resources in such a manner that each server is guaranteed a certain (quantifiable) level of service, with the exact guarantee depending upon the server parameters. That is, a server’s parameters represent its contract with the system, and the system’s global scheduler is obliged to fulfill its part of the contract by providing the level of service contracted for. However, it is incumbent upon each server, and not the global scheduler, to ensure that the individual jobs that comprise the application being modeled by this server perform as expected.

Here, Algorithm GRUB (Greedy Reclamation of Unused Bandwidth) is presented, a global scheduling algorithm that achieves these goals in preemptive uniprocessor systems.

\(^1\)The work presented in this chapter has been carried out with the collaboration of S.K.Barnah, while the author was a visiting student at the University of North Carolina at Chapel Hill.
Our methodology, which is based upon the notion of reserving a fraction of the processor bandwidth for each server, builds upon the Constant Bandwidth Server (CBS) of Abeni and Buttazzo [4]; the CBS, in turn, derives inspiration from the service mechanisms proposed in the Dynamic Sporadic Server (DSS) [48] and the Total Bandwidth Server (TBS) [48, 47].

4.2 System model

Each application $A^i$ is assigned a server $S_i$, which is characterized by two parameters — a processor share $U_i$, and a period $P_i$. The processor share $U_i$ denotes the fraction of total processor capacity that is to be devoted to the application being modeled by $S_i$ (loosely speaking, it should seem to server $S_i$ as though its jobs are executing on a dedicated “virtual” processor, which is of speed $U_i$ times the speed of the actual processor). The period $P_i$ is an indication of the “granularity” of time from server $S_i$’s perspective — while this will be elaborated upon later, it suffices for the moment to assume that the smaller the value of $P_i$, the more fine-grained the notion of real time for $S_i$.

We make the following requirements of our scheduling discipline:

- The arrival times of the jobs (the $a_{i,j}$’s) are not a priori known, but are only revealed on line during system execution. Hence, our scheduling strategy cannot require knowledge of future arrival times.

- The exact execution requirements $c_{i,j}$ are also not known beforehand: they can only be determined by actually executing $J_{i,j}$ to completion. Nor do we require an a priori upper bound (a “worst-case execution time”) on the value of $c_{i,j}$.

Clearly, the kinds of performance guarantees that can be made by our scheduling algorithm are very much influenced by this requirement — in particular, it is not possible for the global scheduler to guarantee that individual jobs meet “hard” deadlines. However, we believe that making such guarantees is the responsibility of the individual server serving the hard-real-time job, and not of the global scheduler: it is incumbent upon the server, based upon the performance guarantees made to it by the global scheduler, to ensure that all hard-real-time guarantees will be met by it based upon the amount of service it is guaranteed to receive from the global scheduler. In other words, hard-real-time feasibility analysis in our model is the responsibility of the individual servers, and not of the global scheduler.

- We are interested in integrating our scheduling methodology with traditional real-time scheduling — in particular, we wish to design a scheduler that is a minor variant of the classical Earliest Deadline First (EDF) [18, 32]. We therefore require that our scheduling strategy be as similar to EDF as possible. In particular, this rules out the use of scheduling strategies based upon “fair” processor-sharing, such as GPS [39, 40] and its variants. There are several reasons for this, including the facts that very efficient implementations of EDF-based schedulers have been devised (see, e.g., [36]), and that it is quite likely that
the remainder of the real-time system will be scheduled using EDF. An additional
major reason for our choice is that we are concerned (see the discussion on choosing
job deadlines in Section 4.4) about the number of preemptions that may occur in
a schedule, and tight bounds are known [35] on the number of preemptions in
EDF-generated schedules.

In this paper, we will consider a system comprised of $n$ servers $S_1, S_2, \ldots, S_n$, with each
server $S_i$ characterized by the parameters $U_i$ and $P_i$ as described above. Furthermore,
we restrict our attention to systems where all of these servers execute on a single shared
processor (without loss of generality, this processor is assumed to have unit processing
capacity) — we therefore require that the sum of the processor shares of all the servers
sum to no more than one; i.e.,

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

The Monotonic Deadline Assumption. Algorithm GRUB provides isolation and per-
formance guarantees to applications consisting of only one task, or more generally con-
sisting of monotonically increasing deadline jobs.

A set of job $A^i = J_{i,0}, J_{i,1}, J_{i,2}, \ldots$ is said to have monotonically increasing deadlines
if:

$$\forall k, j \quad a_{i,k} < a_{i,j} \Rightarrow d_{i,k} \leq d_{i,j}$$

According to the definition given in 1.2, a task is a set of jobs having monotonically
increasing deadlines. For the remaining of the chapter, without loss of generality, we will
assume to deal with applications consisting of one single task. However the results stated
here are valid with few modification to applications consisting of jobs with monotonically
increasing deadlines.

We will also assume that jobs are served in a First Come First Served order (FCFS or
FIFO) — i.e, $J_{i,j}$ must complete before $J_{i,j+1}$ can begin execution. Hence, no preemption
is allowed between jobs of the same application. This is an important assumption for
the correctness of the results presented in this chapter.

In the next chapter, the BSS (Bandwidth Sharing Server) algorithm is presented,
which overcomes the limitation of the monotonic deadline assumption.

Performance Guarantee. Recall that our goal with respect to designing the global
scheduler is to be able to provide complete isolation among the servers, and to guarantee
a certain degree of service to each individual server. As stated above, the processor share
$U_i$ of server $S_i$ is a measure of the fraction of the total processor that should be devoted
to executing (jobs of) server $S_i$. The performance guarantee that is made by our global
scheduler is as follows (this will be formally proved in Section 4.4):

Suppose that job $J_{i,j}$ would begin execution at time-instant $A_{i,j}$, if all jobs of
server $S_i$ were executed on a dedicated processor of capacity $U_i$. In such a
dedicated processor, $J_{i,j}$ would complete at time-instant $F_{i,j} = A_{i,j} + (c_{i,j}/U_i)$,
where \( c_{i,j} \) denotes the execution requirement of \( J_{i,j} \). If \( J_{i,j} \) completes execution by time-instant \( f_{i,j} \) when our global scheduler is used, then it is guaranteed that

\[
  f_{i,j} \leq A_{i,j} + \left( \frac{c_{i,j} / U_i}{P_i} \right) 
  \cdot P_i
\]

From the above inequality, it directly follows that \( f_{i,j} < F_{i,j} + P_i \). This is what we mean when we refer to the period \( P_i \) of a server \( S_i \) as a measure of the “granularity” of time from the perspective of server \( S_i \) — jobs of \( S_i \) complete under Algorithm GRUB within a margin of \( P_i \) of the time they complete on a dedicated processor.

### Scheduling Hard Real-Time Tasks

Most bandwidth-server algorithms, including Algorithm GRUB, offer support to hard-real-time periodic tasks [32] in addition to soft real-time tasks of the kind we have discussed above. In general, this is achieved by a semantic reinterpretation of some of the servers — a server \( S_i \) with processor capacity \( U_i \) and period \( P_i \) can be considered to support a hard-real-time periodic task with execution requirement \( U_i \cdot P_i \) and period \( P_i \) (equivalently, a hard-real-time periodic task with execution requirement \( C \) and period \( T \) can be served by a server \( S_i \) with server capacity \( U_i \leftarrow C / T \) and server period \( P_i \leftarrow T \)), such that every hard deadline of the periodic task will be met. We will discuss this property of Algorithm GRUB in Section 4.4.3.

### Reclaiming unused capacity.

Several other server-based global schedulers (e.g., CBS [4]), can offer performance guarantees somewhat similar to the one made by Algorithm GRUB. However, Algorithm GRUB has an added feature that is not to be found in many of the other schedulers — an ability to reclaim unused processor capacity (“bandwidth”) that is not used because some of the servers may have no outstanding jobs awaiting execution. While such reclamation does not directly affect the performance guarantee that can be made by Algorithm GRUB (since in the worst case there may be no idle threads and hence no excess capacity to reclaim), we will show that reclamation tends to result in improved system performance, in particular, with regard to the total number of preemptions in the schedule. Furthermore, the unused capacity reclamation is achieved without any additional cost or complexity — the computational complexity of Algorithm GRUB is the same as that of previously-proposed schedulers, and reclaiming bandwidth at a particular instant in time does not compromise the ability of the system to live up to its performance guarantees in the future.

### Comparison to other work.

Much research has been performed on achieving guaranteed service and inter-thread isolation in uniprocessor multi-threaded environments (see, e.g., [47, 48, 4, 21, 52, 34, 27, 17]). Algorithm GRUB is most closely related to the CBS approach of Abeni and Buttazzo [4], hence, we will compare our approach to the CBS server. For an excellent comparison of the features of other servers, see [4].

Our algorithm differs from the CBS approach in two primary ways. First, we are able to more accurately characterize the behavior of our servers, by comparing the per-
formance of server $S_i$ under Algorithm GRUB to its behavior if executed on a dedicated server of capacity $U_i$. In contrast, many of the performance guarantees made by CBS are somewhat circular, in that assertions can be proved about meeting or missing deadlines which are themselves assigned by the CBS algorithms.

Second and more important, Algorithm GRUB is able to efficiently reclaim excess processor capacity. (CBS, too, reclaims excess processor capacity in the sense that the processor is not allowed to idle while there are jobs waiting execution; however, we will show that Algorithm GRUB is able to use reclaimed excess capacity in a more “intelligent” manner than other schedulers — i.e., to obtain a schedule with better characteristics (fewer preemptions, etc.) than that obtained by CBS and similar servers.)

### 4.3 The GRUB algorithm

In this section, we provide a detailed description of Algorithm GRUB, our global scheduler.

**Algorithm Variables.** For each server $S_i$ in the system, Algorithm GRUB maintains two variables: a deadline $d_i$ and a virtual time $V_i$.

- Intuitively, the value of $d_i$ at each instant is a measure of the priority that Algorithm GRUB accords server $S_i$ at that instant — Algorithm 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. Algorithm GRUB 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 $U_i$.

Algorithm GRUB is responsible for updating the values of these variables, and will make use of them in order to determine which job to execute at each instant in time.

**Server States.** At any instant in time during run-time, each server $S_i$ is in one of three states: inactive, activeContending, or activeNonContending. The initial state of each server is inactive. 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 $t_o$.

At each instant in time, Algorithm GRUB chooses for execution some server that is in its activeContending state (if there are no such servers, then the processor is idled). From among all the servers that are in their activeContending state, Algorithm GRUB chooses for execution (the next job needing execution of) the server $S_i$, whose deadline parameter $d_i$ is the smallest.
While (a job of) $S_i$ is executing, its virtual time $V_i$ increases (the exact rate of this increase will be specified later); while $S_i$ is not executing $V_i$ does not change. If at any time this virtual time becomes equal to the 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.

The following Lemma establishes a relationship that will be useful in the discussion that follows.

**Lemma 4.1** At all times and for all servers $S_i$ during run-time, the values of the variables $V_i$ and $d_i$ maintained by Algorithm GRUB satisfy the following inequalities

$$V_i \leq d_i \leq V_i + P_i.$$ (4.1)

**Proof.**
Follows immediately from the preceding discussion. $\Box$

**State Transitions.** Certain (external and internal) events cause a server to change its state (see Figure 4.1: the labels on the nodes and edge denote the name by which the respective states and transitions are referred to in this chapter):

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:

   $$d_i \leftarrow V_i + P_i$$

   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 Resource reservation for single-task applications

Figure 4.1: State transistion diagram.

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:

\[
d_i \leftarrow V_i + P_i ,
\]

and server \( S_i \) returns to the activeContending state.

5. There is one additional possible state change — if the processor is ever idle, then all servers in the system return to their inactive state.

**Incrementing virtual time.** It now remains to specify how the virtual time \( V_i \) of a server \( S_i \) changes when a job of \( S_i \) is executing. Let us first consider incrementing \( V_i \) at a rate \( 1/U_i \):

\[
\frac{d}{dt} V_i = \begin{cases} \frac{1}{U_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases}
\]  

(4.2)

—intuitively, executing \( S_i \) for one time unit is equivalent to executing it for \( 1/U_i \) time units on a dedicated processor of capacity \( U_i \), and we are updating \( V_i \) accordingly.

When \( V_i \) is incremented as above (i.e., at a rate \( 1/U_i \) while \( S_i \) is executing, and not at all the rest of the time), Algorithm GRUB is very similar to the CBS algorithm of Abeni and Buttazzo [4], and performance guarantees similar to the ones made by CBS can be proven for Algorithm GRUB as well. However, recall that one of the motivations driving our design of Algorithm GRUB is that we be able to reclaim processor capacity that may remain unused because some servers are in the inactive state, and that we make...
efficient use of this reclaimed bandwidth. 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 have no idea at any instant when the currently inactive servers will become active. That is, we should devise strategies for updating the virtual times — the $V_i$'s — in order to maximize reclamation of underutilized processor capacity, without compromising the future performance of individual servers.

**Definition 4.1** We define a server $S_i$ to be active at a particular instant in time if it is in either the activeContending or the activeNonContending state at that time, and inactive if it is in the inactive state.

Intuitively, a server is active at time $t$ if it is either waiting to execute jobs at instant $t$, or if it has already consumed its reserved processor capacity for time $t$.

Algorithm GRUB maintains an additional variable the system utilization $U$, 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. $U$ is initially set equal to zero; whenever a server $S_i$ undergoes the state-transition labelled “1” in Figure 4.1, $U$ is incremented by $U_i$; whenever $S_i$ undergoes the state-transition labelled “3”, $U$ is decremented by $U_i$.

Let $[t, t + \Delta t)$ denote a time interval during which $U$ does not change, and during which (a job of) $S_i$ is executing. We will assign all the excess processor capacity during this interval — a quantity of $\Delta t \cdot (1 - U)$ — to server $S_i$ for “free”. Consequently, $S_i$ has used an amount equal to

$$\Delta t \cdot (1 - U)$$

of its own processor capacity during this interval; equivalently, its virtual time $V_i$ should increase by an amount equal to $\frac{\Delta U_i}{U_i}$. Algorithm GRUB’s rule for updating virtual time is therefore as follows:

$$\frac{d}{dt} V_i = \begin{cases} \frac{U_i}{U}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases} \quad (4.3)$$

Next, we illustrate the operation of Algorithm GRUB by means of an example. The complete pseudo-code for Algorithm GRUB is given in Figure 4.2.

**Example 4.1** Suppose that there is a system with four servers, which have the following parameters:

<table>
<thead>
<tr>
<th>Server $S_i$</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity $U_i$</td>
<td>0.2</td>
<td>0.3</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Period $P_i$</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

We will evaluate the behavior of this system at run-time under two different algorithms: first, with no reclamation of excess capacity (i.e., when virtual-times are incremented as in Equation 4.2), and then with reclamation (i.e., when virtual-times are incremented as in Equation 4.3).
1. Algorithm GRUB always executes the job of the server $S_i$ which is in the activeContending state, and whose deadline parameter $d_i$ is the smallest. As in EDF, ties are broken arbitrarily.

2. While a job of $S_i$ is being executed, $V_i$ is incremented at a rate $\frac{P_i}{d_i}$.

3. If $V_i$ becomes equal to $d_i$ while a job of $S_i$ is executing, then $d_i$ is incremented by an amount $P_i$ ($d_i \leftarrow d_i + P_i$).

4. When job $J_{i,j}$ arrives
   a) If $S_i$ is in the inactive state:
      i. $V_i \leftarrow a_{i,j}$
      ii. $dT_i \leftarrow V_i + P_i$
      iii. $U \leftarrow U + U_i$
      iv. enter the activeContending state
   b) If $S_i$ is in the activeContending state:
      i. make a note of the time — $a_{i,j}$ — at which this arrival occurs
   c) If $S_i$ is in the activeNonContending state:
      i. $d_i \leftarrow V_i + P_i$
      ii. enter the activeContending state

5. When job $J_{i,j}$ completes execution ($S_i$ must be in the activeContending state at this instant) if $J_{i,j+1}$ has already arrived
   a) then $d_i \leftarrow V_i + P_i$
   b) else enter the activeNonContending state

6. If $S_i$ is in the activeNonContending state (in which case $V_i$ must be greater than the current time) and the current time becomes equal to $V_i$
   a) $U \leftarrow U - U_i$
   b) enter the inactive state.

Figure 4.2: PseudoCode for Algorithm GRUB.
No bandwidth reclamation. The schedule in this case is depicted in Figure 4.3:

- Initially, all four servers are in the inactive state.
- Suppose now that $J_{1,1}$ arrives at $a_{1,1} = 0$ with an (as yet unknown) execution requirement $e_{1,1} = 2$. Server $S_1$ changes state and enters the activeContending state, with $V_1$ set to 0 and $d_1$ to 5.
- Suppose that $J_{2,1}$ also arrives at $a_{2,1} = 0$ with (unknown) execution requirement $e_{2,1} = 5$. Server $S_2$, too, changes state and enters the activeContending state, with $V_2$ set to 0 and $d_2$ to 9.
- According to EDF, $S_1$’s server $J_{1,1}$ is selected for execution and $V_1$ is incremented at a rate $1/0.2 = 5$. At time 1, $V_1$ becomes equal to $d_1$ — this results in $d_1$ being incremented to $5 + P_1 = 10$.
- Server $S_2$ now becomes the earliest-deadline server, and consequently $J_{2,1}$ is executed and $V_2$ incremented at a rate $1/0.3$ until time 3.7, at which point in time $V_2$ becomes equal to $d_2$ ($J_{2,1}$ has executed for 2.7 time units by this time). This results in $d_2$ being incremented to $9 + P_2 = 18$.
- Consequently, server $S_1$ again becomes the earliest-deadline server, and $J_{1,1}$ executes, with $V_1$ increasing at a rate of $1/0.2$, to completion (this happens at time 4.7, when $V_1$ is equal to 10).
- Now, $S_2$ is the only server that has a job awaiting execution. $J_{2,1}$ therefore resumes execution with $V_2$ increasing at a rate of $1/0.3$, completing at time-instant 7 (when $V_2$ is equal to $16\frac{2}{3}$).
- Since there are no active servers in the system, both $S_1$ and $S_2$ return to the inactive state at instant 7.
4 Resource reservation for single-task applications

With bandwidth reclamation. In the scenario described above, observe that servers $S_2$ and $S_3$, which together have $(U_2 + U_3) = 0.5$ 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 $S_2$. Let us now consider once again the system above, and trace its behavior in the presence of capacity reclamation under the same sequence of job arrivals (see Figure 4.4).

- Initially, all four servers are in the inactive state.
- Job $J_{1,1}$ 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 $J_{2,1}$ also arrives at $a_{2,1} = 0$ and server $S_2$, too, changes state and enters the activeContending state, with $V_2$ set to 0 and $d_2$ to 9.
- $U$ — the total capacity of all currently active servers — is equal to $(U_1 + U_2) = (0.2 + 0.3) = 0.5$.
- According to EDF, $S_1$’s server $J_{1,1}$ 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 $c_{1,1}$ is equal to 2 (as was the case in Example 4.3), $J_{1,1}$ completes execution at this instant, and enters the activeNonContending state.
- Server $S_2$ now becomes the only active server in the system, and consequently $J_{2,1}$ 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 $V_2$ is equal to $(0.5/0.3) \times 0.3 = 5$. 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 as in Example 4.3, $J_{2,1}$ completes execution at instant 7 and enters the activeNonContending state — at this time, $V_2$ has increased to 7.
- Since there are no active servers in the system, both $S_1$ and $S_2$ return to the inactive state at instant 7.

□

Comparing the two schedules generated in the example above, we immediately see one of the advantages of Algorithm GRUB over non-reclaiming servers — since a reclaiming scheduler like Algorithm 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 Algorithm GRUB than in non-reclaiming servers, and be subject to fewer preemptions, on average. In Section 4.4, we will formally state (and somewhat quantify) these advantages.
4.4 Formal properties of Algorithm GRUB

In the following discussion we will consider real-time applications, whose jobs are characterized by an absolute deadline. As discussed in section 4.1, Algorithm GRUB can provide performance guarantee to both hard real-time and soft real-time tasks. Algorithm GRUB does not consider the absolute deadline of the task, since it assigns each server a server deadline depending on the parameters and on the current status of the server. This server deadline is in general different from the task deadline. We will show that, under certain assumptions, the task absolute deadline is strictly related to the server deadline. In particular, in section 4.4.3 we will show how it is possible to guarantee an hard real-time task with Algorithm GRUB.

4.4.1 Properties of Algorithm GRUB without reclamation

In this section we will analyze the properties of Algorithm GRUB without reclamation, i.e when the server’s virtual time is updated as in Equation (4.2). As said previously, Algorithm GRUB provides isolation and certain performances guarantees to real-time tasks. The isolation property can be expressed as follows:

**Property 4.1 (Bandwidth Isolation)** If the sum of the bandwidths assigned to the servers does not exceed one, then no server can miss its deadline.

In other words, if at time $t$ server $S_i$ is active with deadline $d_i$, then $t \leq d_i$. It doesn’t matter the behavior of the task served by $S_i$, nor the behavior of the other servers, $S_i$ will never miss its deadline. Before demonstrating this property, we’ll give a better look at the behavior of the algorithm.

**Job Transformation** We say that server $S_i$ operates a transformation on the application jobs. To explain this mechanism, consider a job $J_{i,1}$ arriving at $a_{i,1}$ and suppose that the server is inactive at time $t = a_{i,1}$. Then, the server is assigned a deadline $d_i = a_{i,1} + P_i$ and $V(a_{i,1}) = a_{i,1}$. It is allowed to execute with this deadline for at least $P_i \cdot U_i$ units of time. Suppose that the system utilization is $U = 1$.  

---

4 Resource reservation for single-task applications
4 Resource reservation for single-task applications

If \( c_{i,1} \leq P_i \cdot U_i \), when the job finishes the server becomes activeNonContending or inactive waiting for the next job. If \( c_{i,1} > P_i \cdot U_i \), at some time \( t \) the virtual time reaches the server deadline \( \left( V(t) = d_i \right) \); hence the server must postpone its deadline to \( d_i \leftarrow d_i + P_i \). In this case, from the perspective of the global scheduler, it is like having two different “sub-jobs”:

- the first sub-job is \( J'_{i,1}(1) \) with
  \[
  a'_{i,1}(1) = a_{i,1} \\
  c'_{i,1}(1) = (d_{i,1} - a_{i,1})U_i \\
  d'_{i,1}(1) = d_{i,1}
  \]

- the second sub-job is \( J'_{i,0}(2) \) with
  \[
  a'_{i,1}(2) = a_{i,1} \\
  c'_{i,1}(2) = c_{i,1} - c'_{i,1}(1) \\
  d'_{i,1}(2) = d_{i,1} + P_i
  \]

If \( c_{i,1}(2) > P_i \cdot U_i \), this second sub-job can be further divided into two sub-jobs, and so on.

Example 4.1 shows two job transformations (see Figure 4.3). Consider server \( S_i \): at time \( t = 1 \) the server virtual time \( V_i = d_i = 5 \), and the server deadline is postponed to \( d_i \leftarrow d_i + P_i = 10 \). From the global scheduler perspective, this is equivalent to have two sub-jobs, \( J'_{i,1}(1) = (0, 1, 5) \) and \( J'_{i,1}(2) = 0, 1, 10 \). The first sub-job finishes at time 1, while the second sub-job executes from \( t = 3.7 \) to \( t = 4.7 \). A similar transformation is done for server \( S_2 \).

In general, if \( k = \left\lceil \frac{c_{i,j}/U_i}{P_i} \right\rceil \), a job \( J_{i,j} \) will be transformed into \( k \) “sub-jobs” \( J'_{i,j}(1), J'_{i,j}(2), \ldots, J'_{i,j}(k) \), all of which become available for execution at time \( a_{i,j} \). These sub-jobs have execution requirements \( c'_{i,j}(1), c'_{i,j}(2), \ldots, c'_{i,j}(k) \), respectively, where

- \( c'_{i,j}(\ell) = U_i \cdot P_i \), for \( 1 \leq \ell < k \), and

- \( c'_{i,j}(k) = c'_{i,j} - \sum_{\ell=1}^{k-1} c'_{i,j}(\ell) \).

The deadlines of these sub-jobs are set equal to \( d'_{i,j}(1), d'_{i,j}(2), \ldots, d'_{i,j}(k) \), respectively, where

- \( d'_{i,j}(\ell) = a_{i,j} + \ell \cdot P_i \), for \( 1 \leq \ell \leq k \).

That is, we break \( J_{i,j} \) into \( k \) equal-sized sub-jobs, each of execution requirement \( U_i \cdot P_i \) and deadlines \( P_i \) units apart (except that the last transformed sub-job may have an execution requirement \( c'_{i,j}(k) < U_i \cdot P_i \)).

Observe that in the schedule obtained by executing server \( S_i \) on a dedicated processor of capacity \( U_i \), each sub-job will execute such that sub-jobs \( J'_{i,j}(1), J'_{i,j}(2), \ldots, J'_{i,j}(k-
4 Resource reservation for single-task applications

Figure 4.5: Example of job transformation. a) the jobs of task $\tau_1$ with their original deadlines b) the sub-jobs generated by server $S_1$.

1) all complete exactly at their deadlines $d'_{i,j}(1), d'_{i,j}(2), \ldots, d'_{i,j}(k-1)$, and that sub-job $J'_{i,j}(k)$ completes at or before its deadline $d'_{i,j}(k)$.

Example 4.2 In certain cases, the deadline of the original jobs are transformed not because they need to execute more than reserved, but because they arrive too early. Consider the job transformation that is shown in Figure 4.5. Task $\tau_1$ consists of two jobs, $J_{1,1} = (0,4,8)$ and $J_{1,2} = (6,4,14)$, and it is served by server $S_1$ with $U_1 = 0.5$ and $P_1 = 8$. The sub-jobs obtained by the transformation are $J'_{1,1}(1) = (0,4,8)$ and $J'_{1,2}(1) = (6,4,16)$. In fact, when the first job of task $\tau_1$ finishes, the server virtual time is $V_1 = 8$. Now, according to rule 4a (see Figure 4.2) when the next job arrives the server is assigned a deadline $d_1 \leftarrow V_1 + P_1 = 16$, instead of the original job deadline ($d_{1,1} = 14$). In this way, the execution of the server can be completed without deadline postponing. However, depending on the final schedule, the second job might finish after time $t = 14$, missing its original deadline. Algorithm GRUB cannot guarantee that task $\tau_1$ will meet all its deadlines, because $\tau_1$ has not fulfilled its contract (the second job arrived too early).

In the following, in order to simplify the notation, we will denote the $k$ jobs produced by the job transformation simply as $J'_{i,1}, \ldots, J'_{i,k}$.

Some consideration is worth to be done:

- The transformation done by server $S_i$ depends only on the characteristics of the jobs served by $S_i$; it is independent of the presence of other servers in the system. This property does not hold anymore when considering reclamation, because the rate at which the virtual time is updated depends upon the active servers in the
system. Also, this property does not hold when considering multi-task servers, as we will see in next chapter.

- Let’s define $D'_i(a, b)$ as the demand of the sub-jobs $J'_{i,1}, \ldots, J'_{i,k}$ with arrival times and deadlines in $[a, b]$. Observe that in the previous examples, $\forall a, b \ D'_i(a, b) \leq (b - a)U_i$. This is always true for every server, as we will prove with the following Lemma.

Now we will formally prove Property 4.1 in two steps: first we will prove that in any interval the demand of the sub-jobs produced by server $S_i$ never exceed the server share. Then we will use this result and the optimality of EDF to prove the main theorem.

**Lemma 4.2** The demand of the transformed sub-jobs produced by server $S_i$ in any interval $[a, b]$ never exceeds $(b - a)U_i$:

$$\forall a, b \quad D'_i(a, b) \leq (b - a)U_i$$

**Proof.**
Let $J'_{i,1}, \ldots, J'_{i,k}$ be the set of transformed sub-jobs produced by server $S_i$ such that $J'_{i,1}$ is the first sub-job with arrival $a'_{i,1} \geq a$ and $J'_{i,k}$ is the last sub-job with deadline $d'_{i,k} \leq b$. When $J'_{i,i}$ begins execution at time $s_{i,1}$, the server virtual time is $V_i(s_{i,1}) \geq a$. When $J'_{i,k}$ finishes at time $f_{i,k}$, the server virtual time is $V_i(f_{i,k}) \leq d'_{i,k} \leq b$. Since there is no reclamation (we are using Equation 4.2 for updating the virtual time), $V_i$ is incremented at a rate of $1/U_i$. Hence:

$$V_i(f_{i,k}) - V_i(s_{i,1}) = \sum_{\ell=1}^{k} \frac{c'_{i,\ell}}{U_i}$$

Recall that the sum of the computation times of the transformed sub-jobs $J'_{i,1}, \ldots, J'_{i,k}$ is the demand $D'_i(a, b)$:

$$\sum_{\ell=1}^{k} c'_{i,\ell} = [V_i(f_{i,k}) - V_i(s_{i,1})] U_i \leq (b - a)U_i$$

hence the lemma is proved. \(\square\)

**Theorem 4.1** Given a set of $n$ tasks, each one served by a dedicated server $S_i$ with bandwidth $U_i$, such that the sum of the bandwidths does not exceed one: then no server can miss its deadline.

**Proof.**
The proof descends directly from Lemma 4.2 and from Theorem 3.2: in fact, in every interval $[a, b]$ the sum of the demands of the servers is

$$\forall a, b \quad \sum_{i} D'_i(a, b) \leq \sum_{i} U_i(b - a) \leq (b - a).$$
Since Algorithm GRUB is performing EDF on the server sub-jobs, then from theorem 3.2 it descends that no server can miss its deadline. □

4.4.2 Properties of GRUB with reclamation

When considering Algorithm GRUB with reclamation, the proof of Lemma 4.2 is not correct. In fact, when doing reclamation, the virtual time of each server is updated according to the following Equation:

\[
\frac{d}{dt} V_i = \begin{cases} \frac{r}{b_i}, & \text{if } S_i \text{ is executing} \\ 0, & \text{otherwise} \end{cases}
\]

The rate at which the virtual time is incremented depends on the status of all the servers in the systems. It turns out that the expression \([V_i(f_{i,k}) - V_i(s_{i,1})] U_i\) used in Lemma 4.2 no longer represents the amount of time demanded by server \(S_i\) in \([a, b]\), but only a lower bound.

However, with a different job transformation, Lemma 4.2 and Theorem 4.1 remain valid. To explain this new transformation, let consider the following example.

Example 4.3 Consider again example 4.1 (Figure 4.4). At time 0 server \(S_1\) is selected to execute and its virtual time is incremented at a rate 0.5. After executing for 2 units of time, \(V_1 = 5\). The rate at which the virtual time \(V_1\) is incremented takes into account also the reclamation of the unused bandwidth of servers \(S_3\) and \(S_4\).

We can decompose the computation time executed by \(S_1\) into 3 terms, dividing the contribute of server \(S_1\) from the contribute due to reclamation of the unused bandwidth unused of servers \(S_3\) and \(S_4\).

Suppose that, while \(S_1\) executes, the virtual times of \(S_3\) and \(S_4\) are incremented at a rate of 1, such that at time \(t = 2\), when server \(S_1\) leaves the processor, \(V_3 = V_4 = t = 2\). Note that, updating the virtual time of all inactive servers in this way, we do not jeopardize the future behavior of these servers when a new job arrives — in fact, if a job of \(S_3\) arrives at any time \(\bar{t}\) between 0 and 2, the server virtual time is equal to the current time: \(V_3(\bar{t}) = \bar{t}\), and we can apply rule 4a just as before.

Hence, we can “charge” the following computation time to server \(S_3\):

\[
c'_{3,1} = [V_3(2) - V_3(0)] U_3 = 0.5.
\]

This is equivalent to having a sub-job \(J'_{3,1} = (0, 0.5, 5)\) for server \(S_3\) with the same deadline and arrival time of the currently executing job. In the same way, we can charge a computation time to server \(S_4\):

\[
c'_{4,1} = [V_4(2) - V_4(0)] U_4 = 0.5.
\]

This is equivalent to having a sub-job \(J'_{4,1} = (0, 0.5, 5)\) for server \(S_4\). Finally, the computation time that can be charged to server \(S_1\) is:

\[
c_{1,1} = [V_1(2) - V_1(0)] U_1 = 1.
\]

This is equivalent to having a sub-job \(J'_{1,1} = (0, 1, 5)\). Notice that
4 Resource reservation for single-task applications

![Diagram of job transformation](image)

Figure 4.6: Job transformation in case of reclaimation.

- the sum of the computation time of the 3 transformed sub-jobs \( c'_{1,1} \), \( c'_{3,1} \) and \( c'_{4,1} \) is exactly equal to the original computation time of job \( J_{1,1} \);
- all three sub-jobs have the same arrival times and the same deadlines.

A graphical description of this job transformation is shown in Figure 4.6: in this figure jobs \( J'_{1,1} \), \( J'_{3,1} \), \( J'_{4,1} \) are being scheduled by an ideal processor sharing scheduler, with the inactive servers \( S_3 \) and \( S_4 \) executing in proportion to their shares in every interval of time. Notice that every transformed sub-job finishes before its deadline.

From the previous example, we notice that if we update the virtual times of all the inactive servers at a rate of 1, Algorithm GRUB remains consistent. In this way, we can “charge” the reclaimed computation time to the inactive servers by considering additional transformed sub-jobs for each inactive server. These sub-jobs have:

- arrival time and deadline equal to the arrival time and deadline of the currently executing server, and
4 Resource reservation for single-task applications

- computation time equal to the computation time of the currently executing server multiplied by the bandwidth of the inactive server.

With this new job transformation, we can prove again Lemma 4.2.

Proof of Lemma 4.2 with bandwidth reclamation. Let $J'_{i,1}, \ldots, J'_{i,k}$ be the set of transformed sub-jobs for server $S_i$ (included the new transformed sub-jobs when server is inactive) such that $J'_{i,o}$ is the first sub-job with arrival $a'_{i,1} \geq a$ and $J'_{i,k}$ the last sub-job with deadline $d'_{i,k} \leq b$. The virtual time $V_i$ is such that

$$V_i(a'_{i,1}) \geq a'_{i,1}, \quad V_i(f'_{i,k}) \leq d'_{i,k}$$

Hence, the sum of the computation time of the transformed jobs is:

$$\sum_{k=1}^{k} \left[V_i(f'_{i,k}) - V_i(a'_{i,1})\right] U_i \leq (b - a)U_i$$

□

Hence, Theorem 4.1 remains valid for Algorithm GRUB with reclamation.

4.4.3 Scheduling Hard Real-Time Tasks

Using Algorithm GRUB, it is possible to provide hard real-time guarantees assigning the appropriate parameters to the servers. Consider a periodic (sporadic) hard real-time task $\tau_i$, characterized by a worst case computation time $C_i$ and a period (minimum inter-arrival time) $T_i$. If we want to guarantee that $\tau_i$ will meet all its deadlines, we must assign server $S_i$ a bandwidth $U_i \geq \frac{C_i}{T_i}$ and a period $P_i = T_i$.

To prove this, we need to prove that:

- the server deadline $d_i$ is always coincident with the absolute deadline of the current task’s active job;
- the server never misses its deadline.

The second property has already been proved in the previous section. Now we need to prove the first property.

Lemma 4.3 Suppose that a periodic (sporadic) hard real-time task $\tau_i$ with worst case computation time $C_i$ and period (minimum inter-arrival time) $T_i$ is served by server $S_i$ with bandwidth $U_i \geq \frac{C_i}{T_i}$ and period $P_i = T_i$. Then, when in the active contending state, the server deadline is always coincident with the hard deadline of the currently served job.
Proof.
We will prove the Lemma by induction.

The Lemma is true for the first job served by $S_i$: according to rule 4a (see Figure 4.2), when $J_{i,0}$ arrives, $d_i = a_{i,0} + P_i = a_{i,0} + T_i = d_{i,0}$. The server virtual time is incremented at a rate\[ \frac{U_i}{U_i} \leq \frac{T_i}{C_i} \]

Hence, when job $J_{i,0}$ completes,
\[ V_i \leq a_{i,0} + C_i \frac{T_i}{C_i} = d_i \]

This means that rule 3 is not applied, and the server deadline is not postponed.

Now, suppose that the Lemma is true for job $J_{i,j-1}$; we will prove it correct for job $J_{i,j}$. When job $J_{i,j}$ arrives at time $a_{i,j} = a_{i,j-1} + T_i$, the server is in the inactive state — by Lemma 4.1, if the server is active (contending or not), the virtual time $V_i$ is always less than the current server deadline $d_i$.

Hence, by rule 4a,
\[ d_i = a_{i,j} + P_i = a_{i,j} + T_i = d_{i,j} \]

As in the previous case, rule 3 is never applied and the server deadline is not postponed.

\[ \Box \]

**Theorem 4.2** Suppose that a periodic (sporadic) hard real-time task $\tau_i$ with worst case computation time $C_i$ and period (minimum inter-arrival time) $T_i$ is served by server $S_i$ with bandwidth $U_i \geq \frac{C_i}{T_i}$ and period $P_i = T_i$. Then, task $\tau_i$ is guaranteed to meet all its deadlines.

Proof.
Descends directly from Theorem 4.1 and Lemma 4.3. \[ \Box \]

### 4.4.4 Scheduling Soft Real-Time Tasks

In this section, we will formally prove that Algorithm GRUB (i) closely emulates the performance that the servers would experience if they were each executing on dedicated processors of lower capacity (Theorem 4.3), and (ii) distributes excess bandwidth in a somewhat fair manner (Theorem 4.4). But first, some definitions.

Let $A_{i,j}$ and $F_{i,j}$ denote the instants that job $J_{i,j}$ 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 $F_{i,j}$ are easily seen to hold:
\[ A_{i,1} = a_{i,1} \]
\[ F_{i,1} = A_{i,1} + \frac{C_{i,1}}{U_i} \]
\[ A_{i,j} = \max \left( F_{i,j-1}, a_{i,j} \right) \text{, for } j > 1 \]
\[ F_{i,j} = A_{i,j} + \frac{c_{i,j}}{U_i} \text{, for } j > 1 \]  

(4.4)

The following theorem formally states the performance guarantee that can be made by Algorithm GRUB vis a vis the behavior of each server when executing on a dedicated processor:

**Theorem 4.3** Let \( f_{i,j} \) denote the time at which Algorithm GRUB completes execution of job \( J_{i,j} \). Then the following inequality holds:

\[ f_{i,j} \leq A_{i,j} + \left[ \frac{c_{i,j}}{U_i} \right] P_i. \]  

(4.5)

We will need the following lemma in our proof:

**Lemma 4.4** At the instant that job \( J_{i,j} \) is first considered for scheduling by Algorithm GRUB, \( V_i = A_{i,j} \).

**Proof.**

If \( S_i \) is in the inactive state when job \( J_{i,j} \) arrives and is considered, then Algorithm GRUB immediately sets \( V_i \) to \( a_{i,j} \) (which is in turn equal to \( A_{i,j} \)), and the result is seen to hold.

If \( S_i \) is in either the activeContending or activeNonContending state at the instant when \( J_{i,j} \) is first considered, then by definition of virtual time \( V_i \), server \( S_i \) has by that instant received the same amount of service that it would have received by time \( V_i \) if executing on a dedicated processor of capacity \( U_i \). But that is exactly the value of \( A_{i,j} \).

\( \Box \)

**Proof of Theorem 4.3.** Recall that \( J_{i,j} \) is the \( j \)th job generated by server \( S_i \), and that it has an execution requirement \( c_{i,j} \) (the value of \( \alpha_{i,j} \) is not known prior to completing the execution of \( J_{i,j} \)). Let \( k = \left[ \frac{c_{i,j}}{U_i/P_i} \right] \). As explained in Section 4.4.1, the server will “produce” \( k \) separate “sub-jobs” \( J_{i,j}(1), J_{i,j}(2), \ldots, J_{i,j}(k) \), all of which become available for execution at time \( A_{i,j} \).

Observe that in the schedule obtained by executing server \( S_i \) on a dedicated server of capacity \( U_i \), each such sub-job will execute such that sub-jobs \( J_{i,j}(1), J_{i,j}(2), \ldots, J_{i,j}(k-1) \) all complete exactly at their deadlines \( d_{i,j}(1), d_{i,j}(2), \ldots, d_{i,j}(k-1) \), and that sub-job \( J_{i,j}(k) \) completes at or before its deadline \( d_{i,j}(k) \).

Now when \( J_{i,j} \) is first considered for scheduling by Algorithm GRUB, it follows from Lemma 4.4 that \( V_i \) at this instant is equal to \( A_{i,j} \). Consequently, the value assigned to \( d_i \) (“\( d_i \leftarrow V_i + P_i \)” at this instant is exactly \( d_{i,j}(1) \), and that the subsequent values assigned \( d_i \) during the scheduling of \( J_{i,j} \) (“\( d_i \leftarrow d_i + P_i \)” are exactly the values \( d_{i,j}(2), d_{i,j}(3), \ldots \).

Finally, from Theorem 4.1, no server ever misses its deadline. Hence, sub-job \( J_{i,j}(k) \) completes before its deadline:

\[ f_{i,j} = f_{i,j}(k) \leq d_{i,j}(k) = A_{i,j} + \left[ \frac{c_{i,j}}{U_i} \right] P_i. \]
4 Resource reservation for single-task applications

\[ \square \]

**Corollary 4.1** The completion time of a job of server \( S_i \) when scheduled by Algorithm GRUB 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

\[
 f_{i,j} \leq A_{i,j} + \left\lfloor \frac{c_{i,j}}{U_i} \right\rfloor \cdot P_i
\]

\[
 < A_{i,j} + \left( \frac{c_{i,j}}{U_i} \cdot P_i + 1 \right) \cdot P_i
\]

\[
 = A_{i,j} + \frac{c_{i,j}}{U_i} + P_i
\]

\[
 = \left( A_{i,j} + \frac{c_{i,j}}{U_i} \right) + P_i
\]

\[
 = F_{i,j} + P_i \quad \text{(By Equation 4.4)}
\]

Thus, \( f_{i,j} \) — the completion time of the \( j \)'th job of server \( S_i \) when scheduled by Algorithm GRUB — 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. \( \square \)

4.5 Performance analysis

In section 4.4.3, we have proposed a rule for assigning the server parameters \( U_i \) and \( P_i \) in case we are dealing with an hard-real time periodic (sporadic) tasks. What about soft real-time tasks? Before answering to this question, we need to discuss the issues in scheduling soft real-time applications.

**Choosing server utilization.** As anticipated in section 1.5, it is difficult or even impossible to compute a worst case bound on the computation time of a typical soft real-time task. In addition, if some temporal constraint is not respected, nothing catastrophic happens, but the quality of service delivered to the end-user degrades. Hence, the goals in scheduling a soft real-time applications are:

- try to meet all the constraints whenever it is possible;
- otherwise, allow a temporary degradation in the performance.

In addition, we want to schedule a soft real-time application in a multi-programmed system, where it will share the processor with other applications. We assign each application a percentage of the processor utilization, and we expect that the application
will execute as it were scheduled alone on a dedicated processor. Of course, in the worst case, Algorithm GRUB cannot do better that the dedicated processor scenario: however, we expect that in the average case, it will reclaim the unused processor capacity from the inactive servers, improving the application performances.

We consider a soft real-time task as a special case of an hard real-time tasks, for which the job deadlines are not critical. For example, for a soft real-time periodic task, we could relax the constraint that each job must finish before the next job is released: if a job has not yet completed when the next job is released, then next job is buffered.

From the above discussion, it follows that a good strategy may be to assign the server a processor capacity proportional to the application average load, so to optimize the processor utilization. Even though some deadline will be missed in the worst case, for most of the time the application will behave correctly. Moreover, if some hard application is present in the system, the isolation property guarantees that it is not influenced by the misbehaviors of the soft real-time applications.

However, the issue of assigning the processor capacity is complex and it is beyond the scope of this work. In section 2.2.1 we discussed some of the issues involved in assigning a fraction of the CPU to each application. The interested reader can refer to one of the cited papers for more information on this topic.

Choosing job deadlines. Observe that in the statement of Theorem 4.3, as \( P_i \to 0 \), \( f_{i,j} \) approaches \( F_{i,j} \), the completion time of \( J_{i,j} \) on the dedicated processor. A natural question to ask at this point may be: why would each server \( S_i \) not choose \( P_i \) to be arbitrarily small, and hence obtain exact emulation of its behavior on a dedicated server? To answer this question, we need to look at the issue of job preemptions.

It has been shown [35] that if a set of jobs is scheduled using EDF, then the total number of context-switches due to preemptions is bounded from above at twice the number of jobs. The standard way in which these preemption costs are incorporated into the schedule is by increasing the execution requirement of each job by two context-switch times, and making each such job responsible for switching context twice: first, when it preempts another job to seize control of the processor for the first time; and next, when it completes execution and returns control of the processor to the job with the next highest deadline. (It is easily seen that all context switches in the system are accounted for in this manner.)

As we saw in Section 4.3, Algorithm GRUB schedules a job with a large execution requirement by successively postponing the deadline according to which it is scheduled. In particular, job \( J_{i,j} \)'s deadline may be changed as many as \( \frac{c_{i,j}/U_i}{P_i} \) times. For small \( P_i \), this becomes unacceptably large, and much of server \( S_i \)'s capacity could end up being spent thrashing in context switches. As a rule of thumb, it is probably best to choose a value for \( P_i \) such that \( c_{i,j}/U_i \leq P_i \) for most jobs \( J_{i,j} \) generated by \( S_i \) — i.e., to choose \( P_i \) to be such that the median (or an even higher percentile) of the execution requirements of jobs these are no larger than \( U_i \cdot P_i \).

It is noteworthy that if all the \( P_i \)'s are chosen arbitrarily close to zero, then Algorithm GRUB reduces to the Generalized Processor Sharing (GPS) algorithm of Parekh
4 Resource reservation for single-task applications

and Gallagher [39].

This next theorem concerns the manner in which Algorithm GRUB distributes excess bandwidth among needy servers.

**Theorem 4.4** Suppose that the system utilization $U$ is bounded from above by a constant $c < 1$ during run-time (i.e., the value of the variable $U$, as maintained by Algorithm GRUB, never exceeds $c$), and suppose that a server $S_i$ has jobs awaiting execution at all times. Then $S_i$ receives at least $U_i/c$ of the processor capacity — i.e., the total amount of execution allowed jobs of $S_i$ is at least $U_i/c$ times the total capacity of the processor.

**Proof.**

Since $S_i$ is active at all times, $D_i$ is initially set to $P_i$ and $V_i$ to 0. $V_i$ is now incremented at a rate $U_i/U$ while $S_i$ executes, until $V_i = D_i$ — i.e., $S_i$ executes for an interval of time equal to $(P_i - U_i)/U \geq (P_i - U_i)/c$. At that point, $D_i$ is incremented by $P_i$, and the process is repeated: i.e., for every increment of $D_i$ by $P_i$, server $S_i$ executes for $\geq (P_i - U_i)/c$ time units. □

We thus see that in systems in which the overall system utilization is always bounded from above by a constant, Algorithm GRUB allocates excess capacity to needy servers in direct proportion to their processor shares. In the next section, we will show via simulations that Algorithm GRUB tends to exhibit similar behavior — allocating excess capacity to needy servers in direct proportion to their processor shares — even when the excess bandwidth is not always present, but rather is randomly distributed across the time-line.

### 4.6 Experimental Evaluation of Algorithm GRUB

In Section 4.4, we formally proved that Algorithm GRUB (i) can guarantee that each server gets to execute as though on a dedicated server, to within an accuracy of its period (Theorem 4.3), and (ii) distributes excess capacity that is always available in a weighted max-min fair manner among servers that need this excess capacity (Theorem 4.4).

In designing Algorithm GRUB, we made several design decisions that may, at first glance, seem somewhat arbitrary. In this section, we will first (Section 4.6.1) explain our reasoning behind these decisions, and will argue why we believe that these are the right decisions to have made. In Section 4.6.2, we will provide experimental evidence in the form of simulation results to justify these decisions. We believe that our simulation results will demonstrate two important additional properties of Algorithm GRUB:

1. First, schedules generated by Algorithm GRUB tend to have fewer context-switches than schedules generated by the CBS bandwidth-allocation scheme.

2. Second, excess capacity always tends to be distributed in a weighted max-min fair manner among servers that need this excess capacity, even when the excess
bandwidth is not always present, but rather is randomly distributed across the time-line (i.e., even if the amount of this excess capacity varies over time depending upon the behavior of other servers).

4.6.1 Discussion

We now explain the rationale for some of the design decisions we have made with respect to Algorithm GRUB.

\[ \text{§1. Why does Algorithm GRUB always attempt to increment the deadline parameter when considering a new job?} \]

That is, why do we choose to execute the statement

\[ d_i \leftarrow V_i + P_i \]

when first considering each job \( J_{i,j} \), regardless of the current value of \( V_i \)? For instance, if \( V_i \) is smaller than \( d_i \) at the instant that \( J_{i,j} \) arrives (because \( J_{i,j-1} \) executed for less than expected), the CBS algorithm [2] would not increment \( d_i \) as we have done, but continue executing \( J_{i,j} \) with the old deadline (and consequently, with greater priority than in Algorithm GRUB).

In deciding whether to increment \( d_i \) as above or not, we had to consider the following tradeoff: If we choose to not increment the deadline and \( e_{i,j} \) (whose value is currently unknown as per our model) turns out to be quite small, then \( J_{i,j} \) is likely to complete within the current deadline, and hence to complete before it would in the presence of deadline-incrementing. On the other hand, not incrementing the deadline makes it more likely that \( J_{i,j} \) would not be able to complete within the current deadline, requiring deadline postponement and consequently, further preemptions.

What finally drove our design decision to always increment deadlines in Algorithm GRUB was this: by our assumption, the period parameter of a server is an indication of the granularity of the time from the perspective of the server. By not incrementing deadlines, we may obtain a response that is quicker than this granularity, but presumably this is not of much significance to the application (recall that our goal in designing Algorithm GRUB is not to obtain a fast-response system, but rather to have a system which behaves as predicted based upon the analysis of individual servers on dedicated processors — we are distinguishing between “fast” and “predictable” systems, and our primary goal is predictability and not speed). On the other hand, the potential drawback of additional preemptions is a very real concern — particularly in systems where most of the job execution requirements \( e_{i,j} \) are known to be no larger than server “quota” \( U_i \cdot P_i \), the likelihood of being able to complete without deadline postponement if we do implement deadline-incrementing is quite high, and seems worth the tradeoff.

\[ \text{§2. Why does Algorithm GRUB assign all its excess bandwidth to the currently-executing job, rather than attempting to distribute it evenly to all active servers?} \]

Once again, we carefully considered another alternative — assigning some of the unused bandwidth to each active server (perhaps in proportion to their processor share parameters). There
are some advantages to such a scheme: in particular, all excess capacity is very evenly distributed, and this scheme is fair in precisely the same sense that the GPS scheme [39] is fair. However, the factors that led us to instead assign all excess bandwidth to the currently executing server included:

**computational complexity:** If excess capacity is to be assigned to all active servers, then we would need to update the virtual times of all these servers — this could take time linear in the number of active servers. Our current scheme requires that only the virtual time of one server — the currently executing one — be updated and so has the same runtime complexity as EDF — logarithmic in the (total) number of servers.

**preemption count:** Once again, recall that our primary goal in distributing excess capacity is to reduce the number of deadline postponements — intuitively, it makes sense to devote as much excess capacity to one (or a few) server[s] in order that its job [their jobs] may complete without deadline postponement, rather than to distribute this capacity among a large number of servers, without providing enough excess capacity to any one to avoid deadline postponement. And if this be the case, then the excess capacity may as well be assigned “greedily” to the currently executing job, since its completion prior to its deadline — the earliest deadline in the system — would avoid a deadline postponement and consequent possible preemption.

In any case, while we can construct artificial workloads under which such greedy assignment of excess capacity is provably unfair to certain servers, we believe the results in Section 4.4 and Section 4.6.2 show that Algorithm GRUB does nevertheless tend to be “fair” under random workloads.

### 4.6.2 Simulation results

We have conducted extensive experiments to evaluate the performance of Algorithm GRUB, and to compare this performance with that of other bandwidth-sharing server algorithms. In this section, we will describe some of our findings. We believe that Algorithm GRUB is most similar to the CBS algorithm of Abeni and Buttazzo [2]; hence, we will for the most part focus on comparing Algorithm GRUB with CBS.

**Preemption Count.** We believe the following two experiments validate our contention that Algorithm GRUB tends to schedule a system with fewer preemptions than CBS.

1. In one experiment there is a set of 10 applications (“tasks”), each of which generates a sequence of jobs having a variable execution requirement and variable inter-arrival time. The total average load of the task set is varied from 0.7 to 0.98 times the processor capacity, while the maximum load varies from 2 to 3 times the processor capacity. The periods of the tasks are drawn from the uniform distribution, with a minimum of 100 and a maximum of 800.
Figure 4.7: Preemption experiment – one. The total number of preemptions for a particular average load is seen to be higher in a CBS-generated schedule than in one generated by Algorithm GRUB.

Each run of the simulation lasted for 1,000,000 units of time. The total number of preemptions occurring in the schedule when the system is scheduled using both CBS and Algorithm GRUB were counted, and are plotted in Figure 4.7. Observe that the total number of preemptions for a given average load is consistently greater for CBS than it is for Algorithm GRUB.

2. In this experiment, we measured the number of preemptions suffered by each task when scheduled by different algorithms. There is a total of eight tasks – of these, 5 tasks each generate jobs that have a constant execution requirements and variable inter-arrival times. The maximum load of these five tasks is constant and set equal to 0.79 (thus their average load is less than 0.79). The job inter-arrival time is uniformly distributed between 100 and 800.

The other two tasks generate jobs with variable execution requirements and variable inter-arrival times. Their parameters are as follows:
Figure 4.8: Preemption experiment – two. The number of preemptions per task for a particular average load is seen to be higher in a CBS-generated schedule than in one generated by Algorithm GRUB.

<table>
<thead>
<tr>
<th>Task</th>
<th>parameter</th>
<th>minimum</th>
<th>average</th>
<th>maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_o$</td>
<td>task load</td>
<td>0.0231839</td>
<td>0.0697485</td>
<td>0.209302</td>
</tr>
<tr>
<td></td>
<td>execution requirement</td>
<td>15</td>
<td>30</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>inter-arrival time</td>
<td>215</td>
<td>431</td>
<td>647</td>
</tr>
<tr>
<td>$T_1$</td>
<td>task load</td>
<td>0.00998336</td>
<td>0.0302515</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>execution requirement</td>
<td>6</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>inter-arrival time</td>
<td>200</td>
<td>400</td>
<td>601</td>
</tr>
</tbody>
</table>

While task $T_o$ has more than twice the average execution requirement of task $T_1$, the task average inter-arrival times are close to each other (431 and 400).

Each run of the simulation lasts for 1,000,000 time units, and the execution requirements of $T_o$ and $T_1$ are slightly incremented in each succeeding run. In this manner, the average load for tasks $T_o$ and $T_1$ is increased from 0.1 to 0.2. The number of preemptions suffered by jobs of $T_o$ and $T_1$ under both CBS and Algorithm GRUB are depicted in Figure 4.8. Once again, observe that jobs of both tasks consistently suffer fewer preemptions in the GRUB-generated schedule than they do in the CBS-generated one. While the previous experiment showed that the total number of preemptions tends to be minimized in GRUB-generated schedules, this experiment thus shows that this fall in number of preemptions is distributed among all the tasks.
4 Resource reservation for single-task applications

Fairness properties. We conducted several experiments to evaluate and better characterize the behavior of Algorithm GRUB from the perspective of fair distribution of excess bandwidth. We will now describe one of these experiments. In this experiment we have

- Three applications (tasks) — $T_0, T_1,$ and $T_2$ — with capacities equal to 0.1, 0.2 and 0.3, and periods equal to 200, 400, and 600 respectively. These three tasks continually backlogged — i.e., they always have jobs awaiting execution.

- Some other “periodic” tasks with constant average load, which each have constant periods, and execution-requirements that are drawn from a uniform distribution. In successive runs, we increased the total average execution requirements of these tasks, thus obtaining results as the average load of these periodic tasks increased from 0.1 to 0.4.

In Figure 4.9, we have plotted the average fraction of the total processor capacity that Algorithm GRUB assigned to the continually backlogged tasks $T_0, T_1,$ and $T_2,$ respectively, as a function of the average load of the other tasks (since $U_1 + U_2 + U_3 = 0.6,$ a “periodic task load” of $x$ on the x-axis Figure 4.9 implies an average excess processor capacity of $0.4 - x$).

Observe that the excess processor capacity tends to be distributed among $T_0, T_1,$ and $T_2$ in the ratio 0.1 : 0.2 : 0.3 — equivalently, for a given load (a vertical line through the graph) the average utilizations of $T_0, T_1,$ and $T_2$ (the intercepts of the vertical line with the three curves) are in the ratio 0.1 : 0.2 : 0.3. This validates our contention that Algorithm GRUB exhibits fair behavior in the sense that it tends, on average, to
distribute excess capacity among needy servers in proportion to their processor-share parameters.

4.7 Conclusions

We have proposed a global scheduling algorithm for use in preemptive uniprocessor systems in which several different real-time applications are to execute simultaneously, such that each application is assured certain performance guarantees — the illusion of executing on a dedicated processor — and isolation from any ill-effects of other misbehaving applications. In addition, our proposed algorithm has a well-defined strategy for exploiting excess processor capacity in a manner that

* uses this excess capacity to reduce the number of preemptions and deadline-postponements in the resulting schedule, and

* is, generally speaking, fair (although pathological scenarios can be constructed in which some applications get most of the excess capacity while others get nothing beyond their guaranteed capacities).

We have simulated and implemented our algorithm, and have performed extensive experiments comparing its run-time behavior with that of other bandwidth-sharing algorithms — some of these results are reported in Section 4.6.
5 Resource reservation for multi-task applications

In this chapter\(^1\), the Bandwidth Sharing Server (BSS) algorithm is presented. It is a scheduling strategy for multi-task real-time applications that provides the dual properties of performance guarantees and inter-application isolation. It extends Algorithm GRUB in the sense that it can provide the same kind of guarantees to multi-task application.

5.1 Introduction

The thread model of concurrent programming is very popular and it is supported by most of the operating systems. In this model, each application (or process) has its own address space and applications communicate by means of operating system primitives. Context switch among applications is often an expensive operation. An application can be multi-threaded, i.e. it can consist of several concurrent tasks (also called threads): different tasks belonging to the same application share address space and other resources. Context switch among tasks of the same application is faster and the communication easier.

In many cases, applications may require timing constraints to exhibit the desired performance. Such constraints can be hard (as in control systems) or soft (as in multimedia systems). In order to provide a predefined Quality of Service (QoS) to soft real-time applications and guarantee hard timing constraints, operating systems should support suitable resource reservation mechanisms and scheduling algorithms. Unfortunately, traditional operating systems do not meet this goal because scheduling decisions are not based on real-time parameters, like deadlines or periods. As a consequence, applications may have an unpredictable timing behavior and experience discontinuity under peak load situations.

Our interest is in being able to provide scheduling support in highly-dynamic systems where new applications may join the system at any instant, and currently active applications may leave the system. Furthermore, the behavior of each application cannot be completely a-priori characterized. In such systems, our scheduling goals are to provide:

- some kind of concrete performance guarantee to each admitted application, and

\(^1\)The work presented in this chapter has been done with the collaboration of G. Buttazzo and S.K. Baruah.
Figure 5.1: A real-time multi-task application scheduled by Algorithm GRUB.

- inter-application *isolation* — each application should be protected from the potential misbehavior of other applications.

In this chapter, we will describe the Bandwidth Sharing Server (BSS) algorithm that fulfills the above requirements in an efficient way.

### 5.2 Problem description

As we said in the previous Chapter, Algorithm GRUB cannot provide real-time guarantees to multi-task real-time applications. In the following example we give some hint at the reasons.

**Example 5.1** Consider Application A consisting of two real-time jobs, \( J_1 = (0, 1, 15) \) and \( J_2 = (1, 1, 5) \). Observe that in interval \([0, 15]\) the demand of the application is \( \frac{2}{15} \leq \frac{1}{4} \), and in interval \([1, 6]\) its demand is \( \frac{1}{4} \). So, we decide to assign application \( \mathcal{A} \) to server \( S_1 \) with period \( P_1 = 5 \) and bandwidth \( U_1 = 0.2 \), and let it be scheduled by Algorithm GRUB.

Suppose that another server \( S_2 \) is present in the system with \( P_2 = 8, U_2 = 0.8 \), which is continually backlogged. The resulting schedule is shown in Figure 5.1. Let us analyze the behavior of the first server:

- at time \( t = 0 \), server \( S_1 \) is assigned a deadline \( d_1 = 5 \);

- at time \( t = 1 \), job \( J_1 \) has completed, and the server virtual time is \( V_1 = 5 \). At this time the second job arrives, and \( S_1 \) is assigned a deadline \( d_1 = V_1 + P_1 = 10 \). However, it is not the earliest deadline server, so \( S_2 \) is selected for execution;

- since \( S_2 \) is continually backlogged, it executes until time \( t = 7.4 \), when its virtual time \( V_2 = d_2 = 8 \). At this point, job \( J_2 \) has already missed its deadline.
5.3 Previous work

The problem of scheduling hard and soft multi-task applications in dynamic systems has already been addressed by [7]. In their approach, a dedicated scheduler is used to handle the scheduling of hard and soft tasks. However, this approach requires a high overhead and is not scalable to large systems. In our opinion, this is a serious limitation of the algorithm in [7], because most of the previous approaches are limited to fixed bandwidth allocations. In this work, we present a new algorithm that provides isolation and precise real-time execution to hard and soft applications. Every non-predictable application can be guaranteed only paying some penalty in processor utilization.

In our algorithm, the application level scheduler is handled by a custom scheduler, based on EDF, selects the application with a fixed bandwidth, and a system-level algorithm solves the scheduling problem. In this chapter, we present the Bandwidth Sharing Server (BSS) algorithm, which provides isolation and precise real-time execution to hard and soft applications. Every non-predictable application is to be executed in the exact time window, whereas real-time execution to hard and soft applications is guaranteed only paying some penalty in processor utilization.

In conclusion, this is a serious limitation of the algorithm in [7], because most of the previous approaches are limited to fixed bandwidth allocations. In this work, we present a new algorithm that provides isolation and precise real-time execution to hard and soft applications. Every non-predictable application can be guaranteed only paying some penalty in processor utilization. In our opinion, this is a serious limitation of the algorithm in [7], because most of the previous approaches are limited to fixed bandwidth allocations. In this work, we present a new algorithm that provides isolation and precise real-time execution to hard and soft applications. Every non-predictable application can be guaranteed only paying some penalty in processor utilization. In conclusion, this is a serious limitation of the algorithm in [7], because most of the previous approaches are limited to fixed bandwidth allocations. In this work, we present a new algorithm that provides isolation and precise real-time execution to hard and soft applications. Every non-predictable application can be guaranteed only paying some penalty in processor utilization.
some of the problems of the previous approaches:

- no information on the application is needed other than the desired bandwidth;
- the scheduling strategy is *de-coupled* from the guarantee algorithm: no information is needed on the execution time or arrival rate in order to schedule the tasks. These information are only needed before run-time, in order to calculate the bandwidth to assign to the application server.
- there is no need to distinguish between predictable and non-predictable application; in particular, no processor bandwidth must be wasted when scheduling hard-real-time tasks with a preemptive algorithm;
- the BSS algorithm *isolates* applications: a misbehaving task cannot affect the guaranteed performance of another application; hence, the BSS is particularly suitable to schedule soft-real-time applications, i.e. applications for which we have no exact information on the worst case execution time or arrival rate of the tasks.

In the following sections we will describe the BSS algorithm in great details. Section 5.4 introduce the system model; in section 5.5, a description of the BSS algorithm is given; in section 5.6, the BSS is extended to handle different local scheduling policies; in section 5.7 we will formally analyze the properties of the BSS algorithm; finally, in section 5.8 we describe a way to improve its runtime overhead.

### 5.4 Definitions

In our model, a task $\tau_i$ is a finite or infinite sequence of requests for a shared resource (e.g. the CPU). Each job $J_{ij} = (a_{ij}, c_{ij}, d_{ij})$, is characterized by a request time (or arrival time) $a_{ij}$, a computation time $c_{ij}$ and a deadline $d_{ij}$. There is no constraint on the arrival times: therefore, a task can be time-driven (periodic) or event-driven (sporadic or even aperiodic).

The meaning of $d_{ij}$ depends on the task type. If the task is hard-real-time, $d_{ij}$ represents the *absolute deadline*, that is time by which a job must complete in order for the application to be correct. If the task is soft-real-time, $d_{ij}$ represents the time by which a job should complete, if there is enough resource available. Informally, we can define the percentage of jobs that complete before the deadline as a measure of the Quality of Service (QoS) of a soft task: the more the jobs that complete before $d_{ij}$, the higher is the QoS experienced by the soft task.

Thus, it is the responsibility of the system designer to assign critical tasks a sufficient fraction of the processor bandwidth such that no deadline is missed. On the other end, if the application tasks are not critical, the system designer can also decide to assign resource shares basing on average values, so that deadlines can be met with a certain probability (QoS guarantee).

An application is a set of tasks:

$$\mathcal{A} = \{\tau_1, \tau_2, \ldots, \tau_n\}.$$
In the following, symbol $\tau_i^A$ will indicate the $i$-th task of application $A$. Each application $A$ is assigned a bandwidth $U^A$ which is the fraction of the processor time that the application is allowed to use. Our system consists of a set of applications $S$ which share the same resource. We assume that
\[
\sum_{A \in S} U^A \leq 1.
\]

5.5 The BSS Algorithm

In this section we give an overview of the BSS algorithm. In Figure 5.2, the general system architecture is outlined. Each application $A$ is handled by a dedicated application server $S^A$ and it is assigned a share (or fraction or processor utilization, or bandwidth) $U^A$, 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 $A$, server $S^A$ 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.

If the budget is exhausted, the global scheduler suspends the application: at this point, if the executing task is a hard-real-time task, an exception is raised; if the task is a soft one, the server must postpone the application deadline (i.e. lower its priority) and activate it again with the new deadline and a new budget.

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, as per the description above,
the task selected to be executed is chosen according to the local scheduler policy and might not be the earliest deadline task). The budget calculation will be described in the next section.

To better understand the dynamic of the system, in the following we describe the system events, the interface that the global scheduler module exports towards the application server and the interface that each server exports toward the global scheduler.

The interface that the kernel exports toward the server consists of only two functions:

- **activate (Server S, Budget B, Deadline d):** inserts server S in the global EDF queue with deadline d and budget B. If S becomes the earliest deadline server, then it is selected to execute (its schedule() function is called).

- **suspend (Server S):** extracts server S from the global EDF queue. If S was executing, then its deschedule() function is called; a new server is selected to be executed (its schedule() function is called).

Each server exports the following interface to the global scheduler:

- **schedule ():** the server is selected to execute. In turn the highest priority task is selected to be executed according to the local scheduler policy.

- **deschedule (time e):** the application is no longer executing, and the processor was held for e consecutive units of time.

- **budgetExhausted (time e):** the budget is over and the application has been suspended. If the application consists of hard tasks, then the server raises an exception. If the application consists of soft tasks, the server calculates a new tuple (budget, deadline) and activates the application again.

The dynamic of the system is described by the following set of events:

**Task arrival:** a new task instance is released in an application. If the task is already active, the arrival is buffered. If instead the task is not active, then it is inserted in the local ready queue: this could cause a local preemption. Also, if the newly activated task is also the earliest deadline task in the application, then the server

1. suspend itself calling the suspend(S) function;
2. calculates a new tuple (B, d), where d is the earliest deadline among those of the active tasks in the application and B is calculated according to the algorithm described in section 5.5.1;
3. activates itself calling the activate (S, B, d).

Since the server deadline has changed, a global preemption could occur. In this case, the global scheduler calls the deschedule(E) function of the preempted server, where E is the amount of time that the server held the processor; then it calls the schedule() function of the preemption server to signal that it has gained the processor.
Task end: a task instance finishes execution. If there is some buffered instance for this task, it is activated and inserted in the ready queue. If the ready queue becomes empty, then the server suspend itself, waiting for the next task arrival. Otherwise, the next task in the ready queue is selected to execute. If the finished task was the earliest deadline task in the application, then the server deadline changes: again the server suspends itself, calculates a new tuple \((B,d)\), and activates itself. Just as before, if the server deadline changes, a global preemption can occur.

Budget Exhausted: the server budget is exhausted. The global scheduler suspends the executing server and calls its `budgetExhausted()` function. In this case the server cannot execute any longer with the current deadline, otherwise some other application could be affected. We have two cases:

1. If the application consists of hard tasks, then something catastrophic could happen if the task misses its deadline. This can be considered a fault: therefore, an exception must be raised.

2. If the application is soft, we can simply degrade the level of service of the application lowering its priority. Hence, the server postpones its deadline, a new budget is calculated and the server activate itself again. In section 5.6 we discuss different methods of postponing the deadline.

5.5.1 Budget calculation

To calculate the budget, every server uses a private data structure called list of residuals. For each task of an application \(A\), this list \(L^A\) contains one or more of the following type of elements:

\[ l = (B,d) \]

where

- \(d\) is the task's deadline;
- \(B\) is the budget available in interval \([a,d]\) (where \(a\) is the task's arrival time); that is, the maximum time that application \(A\) 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 \(l_k\) will denote the element in the \(k\)-th position of the list.

List \(L^A\) 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
5 Resource reservation for multi-task applications

![Resource Reservation Diagram](image)

Figure 5.3: Computation of $B_i$

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.

### 5.5.2 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 $d_i$ 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:

$$\exists_{k-1,i} \; d_{k-1} < d_i \leq d_k$$

Now, the budget $B_i$ is computed as:

$$B_i = \min \{ D_i U^A, (d_i - d_{k-1}) U^A + B_{k-1}, B_k \}$$  \hspace{1cm} (5.1)

where $U^A$ is the bandwidth (share) assigned to application $A$ and $D_i$ is the task’s relative deadline. At this point, the new element is completely specified as $l = (B_i, d_i)$ 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 (5.1) 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 5.3: when at time $t$ task $\tau_i$ becomes the earliest deadline task, the algorithm must compute a new budget: it must not exceed the share in interval $[a_i, d_i]$, which is $D_i U^A$; it must not exceed the share in interval $[a_{k-1}, d_i]$ which is $B_{k-1} + (d_i - d_{k-1}) U^A$, and must not exceed the share in interval $[a_k, d_k]$ which is $B_k$. It can be shown that, if $B_i$ is the minimum among these values, then the application will not use more than its share in any other interval.
5.5.3 Updating the list

Every time the application leaves the processor, (i.e. the deschedule(time e) is called), the 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;
- suspend() has been called while the application was executing.

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:

\[ \forall l_j \quad j \geq k \quad B_j = B_j - e \]
\[ \forall l_j \quad j < k \quad \land B_j > B_k \rightarrow \text{remove element } l_j \]

5.5.4 Deleting elements

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

- either \( d_k \leq t \);
- or \( B_k > (d_k - t)U^A \).

It can be seen from Equation 5.1 that in both cases element \( l_k \) is not taken into account in the calculation of the budget. In fact, suppose that element \( l_i \) is being inserted just after \( l_k \). Then

\[ D_lU^A = (d_l - t)U^A < B_k + (d_l - d_k)U^A \]

and \( B_k + (d_l - d_k)U^A \) cannot be chosen in the minimum. Suppose now that element \( l_i \) is being inserted just before \( l_k \). Then

\[ D_lU^A = (d_l - t)U^A < (d_k - t)U^A < B_k \]

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

54
5.5.5 An example

An example of schedule produced by the BSS algorithm is illustrated in Figure 5.4. In this example there are two applications, \( \mathcal{A} = \{\tau_1^A\} \) and \( \mathcal{B} = \{\tau_1^B, \tau_2^B\} \), each of them is served by a BSS algorithm with a bandwidth of 0.5. Let us focus our attention on application \( \mathcal{B} \). The local scheduling policy adopted by server \( S^B \) is EDF.

- At time \( t = 0 \), \( L^B \) is empty and task \( \tau_2^B \) arrives with a request of 4 units of computation time and a deadline of 20. Then, a new element \( l = (10, 20) \) is added to the list:

\[
L^B = \{(10, 20)\}.
\]

it means that all the tasks of \( \mathcal{B} \) with deadline less than or equal to 20 are allowed to execute at most for 10 units of time. Hence, server \( S^B \) activates itself with budget \( B = 10 \) and deadline \( d = 20 \). Since it is the earliest deadline server in the system, it is scheduled to execute.

- At time \( t = 3 \), task \( \tau_1^B \) issues a request of 6 units of computation time with a deadline of 11. Since it is now the earliest deadline task, a local preemption occurs. Thus, task \( \tau_2^B \) leaves processor and the list is updated by decreasing the residual of 3 units of time:

\[
L^B = \{(7, 20)\}.
\]

Then, a new element for task \( \tau_2^B \) is created and inserted in the list:

\[
L^B = \{(4, 11), (7, 20)\}.
\]

Finally, the server suspends itself and activates again with budget \( B = 4 \) and deadline \( d = 11 \).
Figure 5.5: An example of schedule produced by the BSS algorithm: the two tasks in application $A$ are scheduled by Rate Monotonic.

- At time $t = 7$ task $\tau_1^B$ leaves processor because its budget is exhausted. Its deadline is postponed at $d_1 = 11 + 8 = 19$, and the list is updated as follows:

$$L_R^B = \{ (0, 11), (3, 19), (3, 20) \}.$$  

Notice that $\tau_1^B$ has consumed part of the budget of task $\tau_2^B$. The new budget for $\tau_1^B$ is not 4, which is the maximum possible, but it is 3, because it is bounded by the residual in the interval $[0, 20]$. Now, server $S^B$ is assigned a budget $B = 3$ and a deadline $d = 19$. However, it is no longer the earliest deadline server: server $S^A$ has a deadline of 15, so it is scheduled to execute.

- At time $t = 12$, server $S^B$ is again the earliest deadline task, so it can execute. In particular, task $\tau_1^B$ will now execute.

5.6 Support for different scheduling policies

If the local scheduling policy is EDF, then for each application, the server selects its currently active task with the earliest deadline; hence the server deadline is always coincident with the deadline of the first task in the ready queue.

With a different local scheduling policy, the algorithm is the same except that the task selected to be executed might not be the earliest deadline task. However, the server is always assigned a pair (budget, deadline) equal to element $l = (B, d)$ in the residual list corresponding to the earliest deadline task.

To clarify the mechanism, consider the example in Figure 5.5 in which two application are scheduled by the BSS algorithm: application $A$ consists of two tasks, $\tau_1^A$ and $\tau_2^A$ and it is served by a server with a bandwidth of 0.5 and with a Deadline Monotonic scheduler. Application $B$ consists only of one task and it is served by a server with a bandwidth of 0.5 (since there is only one task, the local scheduling policy doesn’t matter).
5 Resource reservation for multi-task applications

Let’s concentrate our attention on application $\mathcal{A}$:

- an instance of task $\tau^A_1$ arrives at time $t = 0$ with deadline $d_1 = 10$ and an (as yet unknown) execution requirement $c_1 = 3$ units. Then the server calculates a budget $B_1 = 5$ and inserts a new element in the residual list:

$$L^A = \{(5, 10)\}$$

Then the server invokes $\text{activate}(A, 5, 10)$, and the global scheduler put it in the global ready queue. However, since the server of application $B$ has an earlier deadline, the $\text{schedule}()$ function is not called until time $t = 3$;

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

- At time $t = 4$ an instance of task $\tau^A_2$ arrives with deadline $d_2 = 12$ and an execution requirement of $c_2 = 5$. According to the DM scheduler, since task $\tau^A_2$ has a smaller relative deadline than task $\tau^A_1$, then a local preemption is done. However, since the earliest deadline in application $\mathcal{A}$ is still $d_1 = 10$, the server budget and deadline are not changed.

- At time 8 the budget is exhausted: application $\mathcal{A}$ has executed for 5 units of time. The global scheduler suspend the server and invokes its $\text{budgetExhausted}()$ function. The server first updates the list:

$$L^A = \{(0, 10)\};$$

then it postpones its deadline: suppose the deadline is postponed by $T = 10$ units of time: $d_1 \leftarrow d_1 + 10 = 20$ (see next section for alternative rules for postponing the deadline). Now the earliest deadline in the application is $d_2 = 12$, and the server calculates a new budget equal to:

$$B_2 = (d_2 - d_1)U^A + B_1;$$

and inserts it into the list:

$$L^A = \{(0, 10); (1, 12)\};$$

Finally, it invokes $\text{activate}(A, 1, 12)$. Since it is again the earliest deadline server in the global ready queue, it is scheduled to execute.

- At time $t = 9$ task $\tau^A_2$ finishes. The server updates the list:

$$L^A = \{(0, 10); (0, 12)\};$$

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

$$L^A = \{(0, 10); (0, 12); (4, 20)\};$$

finally, it invokes $\text{activate}(A, 4, 20)$. Since it is not the earliest deadline server, another server is scheduled to execute.
It is noteworthy 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.

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 or Stack Resource Policy.

5.6.1 Rules to postpone deadlines

When the application budget is exhausted, the server cannot execute any longer with the current deadline otherwise some other application could be affected: thus, its deadline is postponed by a certain amount. Even though postponing the server deadline is necessary, it could cause a global preemption, as we are lowering the priority of the executing server. There are several ways to postpone the deadline depending on the desired level of service and from the local scheduling algorithm.

The simplest rule is to postpone the deadline of the executing task of an amount equal to the task’s relative deadline. Then the server selects the next task in the ready queue and calculates a new budget for it. This rule is intuitive and flexible, however it needs some adjustment for the case of local schedulers different from EDF. In fact, the earliest deadline in the application must be postponed in order to lower the server priority, and in general this deadline can be different from the deadline of the executing task. For example, in a static priority scheduler, after an event of budget exhaustion, the priority of the executing task remains the same but the server executes with a longer deadline.

Another possibility is to postpone the server deadline of a small fixed amount $T$. Suppose for example that, even if the budget is exhausted, the executing task needs to execute only a little more to complete. Thus, postponing the deadline by $T$ could give the task the small amount of budget that is needed to complete without causing a preemption. However, if the task has an highly variable execution requirement, this rule could results in a large amount of unnecessary preemptions.

A good compromise between the two previous rules is to postpone the task deadline by an amount that increases exponentially: the first time a task instance exhausts its budget, its deadline is postponed by $T$; if the same instance exhausts its budget a second time, its deadline is postponed by $2T$; and so on. In this way, the system automatically adapts to the application needs. It is an engineering issue to select more appropriate $T$ in order to minimize the number of preemptions and the completion time of the soft tasks.

5.7 Formal analysis

5.7.1 Properties of the BSS algorithm

The BSS algorithm has two important properties:
5 Resource reservation for multi-task applications

- The **Bandwidth Isolation Property** says that, no matter the local scheduling algorithm, the execution times and the arrival rates of the tasks, no server misses its current deadline.

- The **Hard Schedulability Property** says that if an application $\mathcal{A}$ is scheduleable when executed alone on a dedicated processor with speed $U^A$, then it is scheduleable when handled by a server with bandwidth $U^A$ and local scheduler EDF on a processor with speed 1.

The Bandwidth Isolation Property ensures that if an application demands more service time than expected, the schedulability of the others is not affected, but only the "wrong" application slows down.

On the other end, the Hard Schedulability property ensures that, if the task parameters are correctly estimated, an a-priori guarantee can be performed and the application executes as it was running alone on a slower dedicated processor. Of course, the schedulability condition for an application depends on the local scheduling algorithm and on the application bandwidth. We will demonstrate in the next sections that if we choose EDF as local scheduling strategy, then an application schedulable on a slower dedicated processor is also schedulable with the BSSalgorithm. This property does not hold for local scheduler other than EDF: therefore, if we want to use another local scheduler, we have to provide a schedulability condition.

Here is some example of schedulability conditions for different local schedulers:

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

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

where $C_i$, $D_i$ and $T_i$ are respectively the worst-case execution time in the processor with speed 1, the relative deadline and the period for task $\tau_i$.

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

$$\forall i = 1, \ldots, n \sum_{j=1}^{i} \left[ \frac{T_i}{T_{ij}} \right] \frac{C_j}{U^A} \leq T_i.$$

We will show that the schedulability condition for the EDF algorithm is correct in section 5.7.4. For the Rate Monotonic algorithm, the above formula has not yet been proven correct: a discussion on the problems that arise when using a local strategy different from the EDF algorithm is done in section 5.7.5.

5.7.2 Job Transformation.

The formal properties of the server are analyzed using the processor demand approach that has been introduced in chapter 3.
5 Resource reservation for multi-task applications

Job Transformation. During its execution, server $S$ can assume different deadlines. Just as Algorithm GRUB, we can say that the BSS algorithm operates a job transformation: from the perspective of the global scheduler, server $S$ produces a sequence of sub-jobs that is a transformation of the original application job sequence.

Consider again example 5.5: server $S^A$ is assigned in turn $d = 10$, $d^A = 12$ and $d = 20$. From the perspective of the global scheduler, it produced the following sequence of sub-jobs:

$$J'_1 = \{0, 5, 10\}, \quad J'_2 = \{4, 1, 12\}, \quad J'_3 = \{0, 2, 20\}$$

As in the case of Algorithm GRUB (see section 4.4.1), observe that in every interval $[a, b]$, the demand of the sub-jobs produced by the server is less than the length of the interval multiplied by the server bandwidth:

$$D^A(a, b) \leq (b - a) U^A$$

Now some formal definition.

Application demand and server demand. The demand of a real-time application $A_i$ is defined as the sum of the demands of all the tasks in $A_i$:

$$D_i(a, b) = \sum_{a_{ij} \geq a, d_{ij} \leq b} c_{ij}$$

where $J_{i,j} = \{a_{i,j}, c_{i,j}, d_{i,j}\}$ is a job of application $A_i$. Note that the demand of an application does not depend upon the adopted scheduling algorithm, but only on its temporal requirements.

Similarly, the server demand is defined as:

$$D'_i(a, b) = \sum_{a'_{ij} \geq a, d'_{ij} \leq b} c'_{ij}$$

where jobs $J'_{i,j} = \{a'_{i,j}, c'_{i,j}, d'_{i,j}\}$ are those sub-jobs that have been produced by server $S_i$ having arrival time and deadline in $[a, b]$.

Given these definitions, in the following sections we will prove some fundamental results for our server.

5.7.3 Bandwidth Isolation Property

Lemma 5.1 Let $l = (B, d)$ be an entry that is inserted at time $s$ into the residual list of server $S$. Then, the sum of the execution times of the sub-jobs produced by $S$ that execute after $s$ with deadline less than or equal to $d$ cannot exceed $B$:

$$\sum_i c'_i \leq B.$$
Proof.
It descends directly from the rules of the BSS algorithm. In fact,

- the elements of the residual list have monotonically non-decreasing budgets;
- whenever the server $S$ executes with a deadline less than or equal to $d$, the execution time is subtracted from $B$ and from the budgets of all the elements following $l$;
- if the server executes a job with deadline $d$, and after $B$ units of execution time the job has not yet finished, the server deadline is postponed.

Hence, it is not possible that the budget of some element becomes negative.

\[\square\]

Theorem 5.1 Given a server $\mathcal{S}^A$ which is assigned a bandwidth $U^A$, then in every interval $[a, b]$ the server demand (i.e. the demand of the transformed sub-jobs produced by $\mathcal{S}^A$) never exceeds the length of the interval multiplied by the server bandwidth:

$$\forall a, b \quad D^A(a, b) \leq (b - a)U^A$$

Proof.

Let $J'_1, J'_2, \ldots, J'_n$ be the set of sub-jobs generated by server $\mathcal{S}^A$ with arrival times and deadlines in $[a, b]$. The server demand is defined as:

$$D^A(a, b) = \sum_{i=1}^{n} d_i$$

Let $l_1, l_2, \ldots, l_M$ a sequence of elements that are inserted in the residual list of server $\mathcal{S}^A$ during the evolution of the server status in interval $[a, b]$, such that:

- $l_1 = (B_1, d_1)$ is the first element to be inserted in the residual list of server $\mathcal{S}^A$ at time $s_1 \geq a$, with $d_1 \leq b$ and corresponding to a job with arrival time $a_1 \geq a$.
- for any $k$, $2 \leq k \leq M$, $l_k = (B_k, d_k)$ is inserted at time $s_k \geq s_{k-1}$ with $d_{k-1} < d_k \leq b$ and corresponding to a job with arrival time $a_k \geq a$.

Remember that at time $s_k$ element $l_k$ corresponds to the earliest deadline job in the server queue\(^2\). We can partition the set of produced sub-jobs $J'_1, J'_2, \ldots, J'_n$ into $M$ subsets in the following way:

\(^2\)Of course, $l_1, l_2, \ldots, l_M$ is not the complete sequence of elements that are inserted in the residual list of server $\mathcal{S}^A$, but it is possible that some element $l'$ is inserted in the list between time $s_i$ and $s_{i+1}$ with deadline $d' < d_i$, that is not comprised in the above list. For example, if element $l_1 = (10, 35)$ is inserted in the list at time $s_1 = 15$, and a job with deadline $d' = 30$ arrives at time $s' = 16$, then element $l' = (7, 30)$ is inserted in the list at time $s'$ just before element $s_1$. However, using Lemma 5.1, we can bound the sum of the execution times of all the sub-jobs that execute with deadline less than $d_1$ with $B_1$; so we are not interested in $l'$. 

61
Resource reservation for multi-task applications

- let $J_1', \ldots, J_{m_1}'$ be the set of sub-jobs that execute between $s_1$ and $s_2$; these sub-jobs have deadlines less than or equal to $d_1$;
- for any $k$, $1 < k < M$, let $J_{m_{k-1}+1}', \ldots, J_{m_k}'$ be the set of sub-jobs that execute between $s_k$ and $s_{k+1}$; these sub-jobs have deadlines less than or equal to $d_k$;
- let $J_{m_M+1}', \ldots, J_n'$ the set of sub-jobs that execute after $s_M$.

From Lemma 5.1, it follows that:

$$\sum_{i=1}^{m_1} c_i' \leq B_1.$$ 

From the rule for calculating the budgets (see Equation (5.1)), we can bound $B_1$ with:

$$B_1 \leq (d_1 - a_1)U^A \leq (d_1 - a)U^A.$$ 

When element $t_2$ is inserted at time $s_2$, the time that the server executed since $s_1$ is subtracted from $B_1$:

$$B_1 \leftarrow B_1 - \sum_{i=1}^{m_1} c_i'.$$

From Equation (5.1), we can bound the budget $B_2$ at time $s_2$ with:

$$B_2 \leq B_1 - \sum_{i=1}^{m_1} c_i' + (d_2 - d_1)U^A \leq (d_1 - a)U^A + (d_2 - d_1)U^A - \sum_{i=1}^{m_1} c_i' \leq (d_2 - a)U^A - \sum_{i=1}^{m_1} c_i'$$

From Lemma 5.1, it follows that:

$$\sum_{i=m_1+1}^{m_2} c_i' \leq B_2 \leq (d_2 - a)U^A - \sum_{i=1}^{m_1} c_i'$$

$$\sum_{i=1}^{m_2} c_i' \leq (d_2 - a)U^A$$

Now the line of demonstration should be clear. We can iterate the above reasoning for all $k$, $1 \leq k \leq M$. In particular, the last step is:

$$\sum_{i=m_M+1}^{n} c_i' \leq B_M \leq (d_M - a)U^A - \sum_{i=m_{M-1}+1}^{m_M} c_i'$$

$$\sum_{i=1}^{n} c_i' \leq (d_M - a)U^A \leq (b - a)U^A$$

And this proves the Theorem. \(\Box\)
Theorem 5.2 Given a set of applications \( T = \{ A_1, A_2, \ldots \} \), each of them served by a BSS server with bandwidth \( U^{A_i} \), with \( \sum_i U^{A_i} \leq 1 \), then no server misses its deadline.

Proof.
Descends directly from Theorem 5.1 and Theorem 3.2. \( \square \)

5.7.4 The Hard Schedulability Property

In this section we will show some schedulability conditions that can be used to check the schedulability of an application when handled by the BSS algorithm. The schedulability condition depends on the bandwidth available to the server and on the local scheduling strategy. First we will show the schedulability condition when the local scheduling strategy is EDF. In the next subsection, we will discuss some issues related to using different local strategies like Rate Monotonic.

Theorem 5.3 An application \( \mathcal{A} \), scheduled by the BSS algorithm, with a server \( S^A \) which is assigned bandwidth \( U^A \) and local scheduler is EDF, is schedulable (i.e. each job completes before its deadline) if:

\[
\forall a, b \quad a < b \quad D(a, b) \leq (b - a)U^A.
\] (5.2)

As in the case of Algorithm GRUB the proof will be performed in two different steps. First we will prove that the server deadline is always coincident with the deadline of earliest deadline job in the server; then, using Theorem 5.1, we will prove that no job can miss its deadline.

According to the BSS algorithm, when a job becomes the earliest deadline job in the server queue, then the server is assigned a deadline equal to the job deadline. However, while the server is executing, if a job is requesting more than the server budget, the server deadline is postponed. In this case, it is possible that the job will miss its original deadline. We are going to prove that, under the hypotheses of Theorem 5.3, the server never postpones its deadlines.

Proof.
By contradiction. Suppose that \( t_{ov} \) is the first instant of time at which the server deadline is postponed from \( d = y \) to \( d = y + T \). This means that at time \( t_{ov} \) there is an element in the residual list \( l_j = (0, y) \), whose budget has been exhausted. Let \( l_k = (B_k, d_k) \) be the element such that:

- \( k \geq j \) (\( l_k \) is after \( l_j \) in the list, or it coincides with \( l_j \));
- \( B_k = 0 \).

In other words, \( l_k \) is the element with the greatest deadline which has budget equal to 0. Now, we go back at the time \( s_k \) when \( l_k \) was inserted in the list. At \( s_k \) the earliest
deadline job in the server queue has deadline equal to $d_k$, and the server is assigned budget $B_k$. From Lemma 5.1, the sum of the execution times of the sub-jobs produced by the server in interval $[s_k, t_{on}]$ is exactly $B_k$. Notice that the local scheduling policy is EDF: thus, until time $t_{on}$ the server deadline always coincides with the deadline of the earliest deadline job in the server queue, and the sub-jobs produced by the server coincide with the job of the application. Hence, if we indicate with $a_k$ the minimum among the arrival times of the application jobs that execute in $[s_k, t_{on}]$ with deadline less than or equal to $y$,

$$\sum_{a_i \geq a_k} c_i > B_k.$$ \(d_i \leq d_k\)

In fact, since the last executing job has not yet finished at $t_{on}$, the budget has been exhausted without completing the last job.

Now, let us see how much is $B_k$. At time $s_k$, the budget $B_k$ is calculated according to Equation (5.1). There are three possible cases:

**Case A**: $B_k = B_{k+1}$, i.e., the budget for element $l_k$ is bounded by the budget of the next element in the list. This cannot happen, otherwise at time $t_{on}$ element $l_{k+1}$ would have a budget equal to zero, whereas we choose $l_k$ as the last element in the list with budget equal to 0.

**Case B**: $B_k = (d_k - a_k)U^A$, i.e., the budget for element $l_k$ is bounded by the current earliest deadline job. In this case,

$$\sum_{a_i \geq a_k} c_i > (d_k - a_k)U^A$$ \(d_i \leq d_k\)

and this contradict the hypothesis, because we have found an interval $[a_k, d_k]$ in which Equation (5.2) does not hold. In this case the Theorem is proved.

**Case C**: $B_k = B_{k-1} + (d_k - d_{k-1})U^A$, i.e., the budget for element $l_k$ is bounded by the budget of the previous element. This case is more difficult because we need now to see how $B_{k-1}$ was calculated.

Let $s_{k-1}$ be the time at which element $l_{k-1}$ was inserted in the list. In interval $[s_{k-1}, s_k]$, the sub-jobs produced by the server coincide with the application jobs. Let $a_{k-1}$ be the minimum arrival time among the jobs that execute in $[s_{k-1}, s_k]$ with deadline less than or equal to $d_{k-1}$. Hence, at time $s_k$:

$$B_{k-1} = B_{k-1} - \sum_{a_i \geq a_{k-1}} c_i$$ \(d_i \leq d_{k-1}\)

and

$$B_k = B_{k-1} - \sum_{a_i \geq a_{k-1}} c_i + (d_k - d_{k-1})U^A.$$

64
Again, there are three cases to consider:

**Case C.A**: $B_{k-1}$ is bounded by the next element in the list. With a reasoning similar to that used in case A, it is easily seen that this case is not possible.

**Case C.B**: $B_{k-1} = (d_{k-1} - a_{k-1})U^A$, i.e. the budget for element $l_{k-1}$ is bounded by the current earliest deadline job. In this case,

$$\sum_{a_i \geq a_k \atop d_i \leq d_k} c_i > B_k = B_{k-1} - \sum_{a_i \geq a_{k-1} \atop d_i \leq d_{k-1}} c_i + (d_k - d_{k-1})U^A = (d_{k-1} - a_{k-1})U^A - \sum_{a_i \geq a_{k-1} \atop d_i \leq d_{k-1}} c_i + (d_k - d_{k-1})U^A$$

$$\sum_{a_i \geq a_{k-1} \atop d_i \leq d_k} c_i > (d_k - a_{k-1})U^A$$

and this contradicts the hypothesis, because we have found an interval $[a_{k-1}, d_k]$ in which Equation (5.2) does not hold. In this case the Theorem is proved.

**Case C.C**: $B_{k-1} = B_{k-2} + (d_{k-1} - d_{k-2})U^A$, i.e. the budget for element $l_{k-1}$ is bounded by the previous element. In this case we have to go back some more to see how $B_{k-2}$ was calculated.

It is easy to see that case C.C can be further divided into three cases, C.C.A, C.C.B and C.C.C. However, this recursive reasoning has always a stop: in the worst case, we must go back to the first job of the application, and for that job the server budget is surely computed as in case B.

Hence, it is always possible to find an interval $[a_{k-m}, d_k]$ such that Equation (5.2) does not hold. This contradicts the hypothesis and demonstrates the Theorem. □

**Corollary 5.1** If application $A$ is schedulable with the EDF algorithm when executed on a dedicated processor with speed $U^A$, then it is schedulable by the BSS algorithm on a processor with speed $1$ when served by a server with bandwidth $U^A$ and an EDF local scheduler.

**Proof.**

The schedulability condition for application $A$ when scheduled on a dedicated processor with speed $U^A$ is the following:

$$\forall a, b \quad \sum_{a_i \geq a \atop d_i \leq b} \frac{c_i}{U^A} \leq (b - a)$$

65
5 Resource reservation for multi-task applications

Figure 5.6: Two task executed on a dedicated processor with speed 0.5 by the Rate Monotonic scheduling algorithm.

and substituting, Equation (5.2) is obtained. □

From the previous Corollary, it follows that if a feasible schedule exists for application \(A\) when executed on a slower dedicated processor, than a feasible schedule exists when the application is scheduled by the BSS algorithm using a server whose local scheduler is the EDF algorithm. The same is not true if we consider a different local scheduling algorithm, as we will see in the next section.

5.7.5 Rate Monotonic Schedulability condition

When the application is served by a server that has a local scheduling strategy different from the EDF algorithm, the Hard Schedulability Property does not hold. In other words, the fact that an application is schedulable on a dedicated processor does not imply that the application is schedulable by the BSS algorithm when using a local scheduling algorithm different from EDF. We present here an example with the Rate Monotonic algorithm.

Example 5.2 Consider an application consisting of two periodic hard real-time tasks: \(\tau_1\) with worst case execution time \(C_1 = 1.5\) and period \(T_1 = 5\); and \(\tau_2\) with worst case execution time \(C_2 = 2\) and period \(T_2 = 12\). First we schedule this application on a dedicated processor with speed \(U = 0.5\): the result is shown in Figure 5.6. Applying the Response Time Analysis [6], it is easy to see that the produced schedule is feasible.

However, when this application is scheduled by the BSS algorithm on a processor with speed 1, and served by a server with bandwidth \(U = 0.5\) and RM local scheduler, the application may not be schedulable. Consider the case in which there is another server \(S_2\) in the system with bandwidth \(U_2 = 0.5\) and deadline at 12. The resulting schedule is shown in Figure 5.7: at time \(t = 12\) the server budget is exhausted, the server postpones its deadline and task \(\tau_2\) misses its deadline.

If we analyze the behavior of the server, we notice that at time \(t = 10\) task \(\tau_1\) arrives with deadline at 15. However, it is not the earliest deadline task in the serve queue: then, the server keeps executing with deadline \(d = 12\) and budget \(B = 2\), and this budget is
Figure 5.7: An application scheduled by the BSS algorithm on a shared processor with speed 1, and the Rate Monotonic local scheduling strategy.

not sufficient to complete both $\tau_1$ and $\tau_2$.

Note that, if at time $t = 1.5$ server $S_1$ executes instead of server $S_2$, task $\tau_2$ would have completed by time $t = 3.5$ and no deadline would have been missed. In fact, if task $\tau_2$ completes before time $t = 10$, at that time the server is assigned a pair budget/deadline equal to $(2.5, 15)$, that is enough for executing task $\tau_1$. Hence, the behavior of the first application depends on the presence of other applications in the system.

Is there any way to guarantee an hard real-time application with the BSS scheduling algorithm, using a local strategy different from EDF? We believe that the answer is positive. The following conjecture gives a possible schedulability condition for the case of the Rate Monotonic scheduling algorithm.

**Conjecture** Application $\mathcal{A}$, which consists of periodic or sporadic tasks with deadlines equal to periods, is schedulable by the BSS algorithm using a server with bandwidth $U^\mathcal{A}$ and the Rate Monotonic algorithm as local scheduler if:

$$\forall i = 1, \ldots, n \quad \frac{1}{\sum_{j=1}^{i} \frac{T_i}{T_j}} \frac{C_i}{U^\mathcal{A}} \leq T_i. \quad (5.3)$$

Unfortunately, the above conjecture has not yet been proven.

### 5.8 Improving complexity bounds

It is easy to see that if the list of residuals $\mathcal{L}^\mathcal{A}$ is implemented as a linear list, both the *insert* and the *update* operations take time $O(N)$, where $N$ is the maximum number of elements in the list. In [31], we showed that $N$ is bounded by $O(D_{\text{max}})$, where $D_{\text{max}}$ if the maximum relative deadline among the tasks of the application; hence the complexity
of an operation on the list of residuals is $O(D_{\text{max}})$ which is pseudo-polynomial in the input.

To obtain a more efficient algorithm, we decided to implement the residual list as a novel data structure that we call Incremental AVL tree. An AVL tree is an binary tree with the following properties:

- It is ordered: for each node $n$, every element in its left subtree is "less" than the element in node $n$, and each element in its right subtree is not "less" than the element in node $n$, where "less" is a relation of ordering defined on the elements.
- It is balanced: for each node $n$ the height of the left subtree differs form the height of the right subtree of 1, 0 or -1.
- The usual operations of insertion, deletion and search of an element take time $O(\log(N))$ where $N$ is the number of elements in the tree.

An Incremental AVL Tree is introduced here as a particular kind of AVL tree in which elements are tuples $(b,d)$ that correspond to elements in the residual list. The tree is ordered by increasing values of $d$. While in a residual list we store the budgets $B$, in an incremental AVL tree we store the budgets in a relative way.

For each node $n$, we denote with $n.parent$, $n.left$ and $n.right$ respectively its parent node, its left child and its right child, and with $n.b$ and $n.d$ the relative budget and the deadline. The budget $B$ in each node can be calculated as:

$$B = n.b + \text{reference}(n)$$

where $\text{reference}()$ is a function defined recursively on $n$:

- if $n$ is the root node, $\text{reference}(n)$ returns 0;
- if $n$ is the left child of its parent node, $\text{reference}(n)$ returns $\text{reference}(n.parent)$;
- if $n$ is the right child of its parent node, $\text{reference}(n)$ returns $n.parent.b + \text{reference}(n.parent)$;

In Figure 5.8, a list of residuals is shown, represented as a linear linked list and as an incremental AVL tree. For example, in case b), the budget of node C can be calculated as:

$$C.B = C.b + \text{reference}(B) = \ldots = C.b + B.b = 5$$

and the budget of node F as:

$$F.B = F.b + \text{reference}(F) = \ldots = F.b + E.b + D.b = 9$$

In the following section, we will briefly explain the operation of insertion, update and deletion in an Incremental AVL Tree.
5 Resource reservation for multi-task applications

![Diagram of a resource reservation system](image)

Figure 5.8: Different representations of the same data structure: a) Linear list; b) Incremental AVL tree.

### 5.8.1 Inserting in the tree

When inserting a new element in the tree with deadline \(n.d\), we have to calculate the absolute budget \(B\) from Equation 5.1 and hence the relative budget \(n.b\). This can be done as we descend the tree to find the right position for the element. We keep track of the following variables:

- **\(n\)** is the node we are currently exploring. It is initialized to the root of the tree;
- **before** is the node immediately before \(n.d\), or \(null\) if \(n\) is the element with the earliest deadline. It is initialized to \(null\);
- **after** is the node immediately after \(d\), of \(null\) if \(n\) is the element with the latest deadline. It is initialized to \(null\);
- **isLeft** is a boolean equal true if we are inserting in the left subtree, otherwise it is false. It is initialized to false;
- **ref** is the reference of the current node. It is initialized to 0.

When inserting a new element, the algorithm explores the tree in order to find the correct position for the element. It starts descending from the root of the tree: suppose that currently we are examining node \(curr\) in the tree. If the deadline of the element we are trying to insert is greater than or equal to \(curr.d\), we must insert in the right subtree:

```plaintext
ref = ref + curr.b;
isLeft = false;
before = curr;
curr = curr.right;
```
node leftBalance(n) {
    if (n.left.balance() < 0) return rotateRight(n);
    else if (n.left.balance() > 0) {
        n.left = rotateLeft(n.left);
        return rotateRight(n);
    }
    return n;
}

Figure 5.9: Balancing the left subtree

If instead the deadline of the element we are trying to insert is less than curr.d, we must insert in the right subtree:

isLeft = true;
        after = curr;
        curr = curr.left.

Finally, if curr = null, then the algorithm has found the correct position in the tree and the new relative budget is calculated:

    if (before == null) b1 = \infty
    else b1 = before.b + (d_k - before.d)U^A
    if (after == null) b2 = \infty
    else {
        if (isLeft || before == null) b2 = after.b + ref;
        else b2 = ref - before.b + after.b;
    }
    n.b = \min \left( b_1, b_2, D_k U^A \right);

After an insertion, the tree could result unbalanced. Suppose for example that the right subtree of node n has a height equal to h and the left subtree has a height equal to \( h + 2 \). In an AVL tree, to re-balance the tree, we need to perform one or two rotations, depending on the structure of the left subtree. In Figure 5.9 a pseudo-code for the balancing operation is shown. Of course, there is also a symmetric operation to perform when the right subtree is higher than the left one, not reported here for space limitations.

In Figure 5.10 the operations of left rotation and the right rotation are graphically depicted. These are similar to the corresponding operation in an AVL Tree: in addition we need the adjust the values of the relative budgets to ensure that the tree remains
consistent. The pseudo-code for the left and right rotation is shown in Figures 5.12 and 5.11 respectively.

5.8.2 Updating the tree

After the server has consumed $E$ units of execution time with deadline $d$, the data structure must be updated. As explained in section 5.5.3, we need to find the element with deadline equal to $d$, and subtract $E$ from the budget of this element and from all the following ones. Moreover, we have to check that the list remains consistent deleting all the elements that have a deadline smaller than $d$ and a budget greater than $B$. This can result in a great number of elements to delete.

To minimize the number of elements to delete we use the technique of “lazy” deletion. In other words, we avoid to delete an element during an update operation: for consistency reasons, we only update the elements in the path from the root to the node with deadline $d$.

Suppose that we have to subtract $E$ units of budget from node $n$ in the tree. If $E \leq n.b$ then the update is quite an easy operation: we only need to subtract $E$ from $n.b$. Notice that:

- The nodes in the right subtree of $n$ don’t need to be updated. In fact, their budget is relative to $n.b + \text{reference}(n)$. 


node rotateRight(n) {
    tmp = n.left;
    node.left = tmp.right;
    tmp.right = n;
    n.b = n.b - tmp.b;
    return tmp;
}

Figure 5.11: Right rotation

node rotateLeft(n) {
    tmp = n.right;
    node.right = tmp.left;
    tmp.left = n;
    tmp.b = tmp.b + n.b;
    return tmp;
}

Figure 5.12: Left rotation
5 Resource reservation for multi-task applications

- We should check the elements in the left subtree of $n$ to see if some of them must be deleted. However we can avoid to do this operation just now: next time we go down that path (for an insertion or for another update) we can update those elements (lazy update):

- All the other nodes in the tree don’t need to be updated.

The case in which $E > n.b$ is more complex, as we have to adjust the reference of $n$, that is, we have to adjust the relative budgets of the parent node of $n$. The pseudo-code for the final update operation is shown in Figure 5.13. Initially, the update is invoked on the root node: then, it recursively invokes itself on the right or left subtree. As it goes down to the left (line 8-12), it checks if the left node is inconsistent due to a previous update operation: in this case, before performing any update, simply adjust the relative budget of the left child. The variable `diff` takes into account the case in which $E > n.b$: if `diff < 0` it means that the relative budget of the current node must be adjusted properly (function `adjust()`).

5.8.3 Deleting elements

To minimize the number of operations to perform, we delete only when the current time is greater than the deadline of the root element. At that point, the algorithm deletes all at once the root element and all its left subtree. As a consequence, the right child becomes the new root.

Of course, doing lazy deletion, we keep in the tree more elements than necessary. However, this does not increase dramatically the overall complexity of the algorithm. It can be shown that the maximum number of elements in the tree is $O(2D_{\text{max}})$, where $D_{\text{max}}$ is the largest relative deadline in the application.

5.8.4 Complexity

It has been shown that the complexity of the insert and update operations in a balanced binary tree $O(\log(N))$, with $N$ number of elements in the tree. According to the discussion in the previous section, $N = O(2D_{\text{max}})$. Since the complexity of implementing the ready queue in each server can be as low as $O(\log(N_A))$ (where $N_A$ is the number of tasks in application $\mathcal{A}$), the overall complexity of the BSS algorithm is $O\{\log(N_{\text{MAX}}) + \log(N_{\text{App}}) + \log(D_{\text{MAX}})\}$ where $N_{\text{App}}$ is the number of applications and $D_{\text{MAX}}$ and $N_{\text{MAX}}$ are respectively the maximum relative deadline and the maximum number of tasks among all the applications.

5.9 Conclusions and future work

If general-purpose computers are to support both real-time and non-real-time applications, it is important that (i) performance guarantees be provided to individual real-time applications, and (ii) each application be isolated from the potential misbehavior of other active applications.
double update(node, dline, amount) {
    double diff = 0;
    if (node.d == dline) diff = adjust(node, -amount);
    else if (node.d < dline) {
        diff = update(node.right, dline, amount);
        diff = adjust(node, diff);
    }
    else {
        if (node.left.b > node.b) node.left.b = node.b;
        diff = update(node.left, dline, amount);
        node.b = node.b - (amount + diff);
    }
    return diff;
}

double adjust(node, delta) {
    double ret = 0;
    if (delta < 0) {
        node.b = node.b + delta;
        if (node.b < 0) {
            ret = node.b;
            node.b = 0;
        }
    }
    return ret;
}

Figure 5.13: The update() and adjust() functions.
5 Resource reservation for multi-task applications

In this chapter, we have described the BSS algorithm, which provides inter-application isolation and performance guarantee to hard and soft real-time multi-task application. It improves Algorithm GRUB, generalizing the guarantee to multi-task applications. Moreover,

- we managed to delink each application’s internal scheduling strategy from the BSS specification, so that it is now possible to choose an arbitrary scheduling discipline for the application;
- we proposed a new data structure to implement the lists of residuals that are maintained by BSS. By storing these lists of residuals as balanced binary trees rather than as linked lists/arrays, we have obtained a significant reduction of the complexity, and therefore reduced the run-time overhead.

As future work, we are considering several issues. First we are trying to understand what are the schedulability conditions when using a scheduler different from EDF is chosen as local scheduling strategy. This is important if we want to use mechanisms like PI and PC [44] or SRP [7], to control access to critical sections among task of the same application.

Then, the issue of global synchronization is being considered. Until now, applications have been considered to be independent. In the future we plan to further extend the BSS and GRUB algorithms to permit different application to access global-shared resources. We believe that this problem is crucial for an implementation of techniques based on bandwidth allocation in real operating systems.

Another interesting issue is the reclaiming of unused bandwidth. In fact, during the system life, some application can be temporarily idle; also, there can be time intervals in which the system load is strictly less than 1. Similarly to what happens with Algorithm GRUB, the active applications may take advantage of this unused bandwidth improving their quality of service and performance expectation.
Bibliography


Bibliography


system supporting dynamic qos control. In *2nd International Workshop on Real-


[21] Pawan Goyal, Xingang Guo, and Harrik M. Vin. A hierical cpu scheduler for
multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

puter Laboratory, University of Cambridge, February 1994.

[23] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time sys-

of operating systems services for real-time applications. In *Proceedings of the 19th
IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, december
1998. IEEE.

priority task systems. In *Proceedings of the IEEE Real-Time Systems Symposium*,
pages 212–221, December 1993.
Bibliography


Bibliography


