

UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences

Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints

Thèse présentée en vue de l'obtention
du grade de Docteur en Sciences

Joel GOOSSENS
Année académique 1998-1999

Acknowledgements

Je tiens à adresser mes plus vifs remerciements à toutes les personnes qui m'ont aidé dans la réalisation de ce travail.

Je remercie Monsieur le Professeur Raymond DEVILLERS d'avoir accepté la direction de cette thèse. Je voudrais faire honneur ici à sa rigueur scientifique et à son esprit critique qui ont guidé mon travail de manière significative.

Je voudrais aussi remercier ceux qui ont contribué à mon encadrement de recherche, ceux qui m'ont donné l'opportunité de participer à des réunions scientifiques, à confronter mes idées avec d'autres chercheurs, je pense ici à Messieurs les Professeurs Thierry MASSART et Raymond DEVILLERS.

J'exprime aussi ma gratitude à ceux qui ont contribué à créer une atmosphère cordiale et plaisante dans mon environnement de travail ; je remercie ainsi Messieurs Laurent FRANCK, Christian HERNALSTEEN, Olivier MARKOWITCH, Grégory SERONT, Jean-Yves VINCENT et Frank WEIS.

Je tiens aussi à remercier Monsieur le Professeur Yves ROGGEMAN pour m'avoir permis d'intégrer le corps scientifique de notre Université.

Mes remerciements vont aussi aux membres de mon jury : Messieurs les Professeurs Raymond DEVILLERS, Guy LOUCHARD, Erik LUIT, Thierry MASSART et Yves ROGGEMAN.

Pour terminer, je voudrais également remercier tous ceux qui à des degrés divers m'ont apporté leur aide par leur intérêt et leurs encouragements.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Real-time systems | 2 |
| 1.2 | Tasks | 3 |
| 1.2.1 | Task constraints | 4 |
| 1.2.2 | Requests | 5 |
| 1.2.3 | Graphical conventions | 6 |
| 1.3 | Scheduling problem | 8 |
| 1.4 | Model of computation | 10 |
| 1.4.1 | Periodic task set | 10 |
| 1.4.2 | Schedule | 12 |
| 1.4.3 | Hypotheses | 13 |
| 1.5 | Overview of the thesis | 14 |
| | Bibliography | 15 |
| 2 | Static Schedulers | 19 |
| 2.1 | Introduction | 20 |
| 2.2 | The rate monotonic scheduler | 20 |
| 2.2.1 | Optimality | 21 |
| 2.2.2 | Feasibility tests | 27 |
| 2.3 | The deadline monotonic scheduler | 34 |
| 2.3.1 | Optimality | 34 |
| 2.3.2 | Feasibility tests | 36 |
| 2.4 | Synchronous arbitrary deadline systems | 36 |

| | | |
|----------|--|------------|
| 2.5 | Asynchronous general deadline systems | 43 |
| 2.6 | Asynchronous arbitrary deadline systems | 52 |
| 2.7 | Stability | 59 |
| 2.7.1 | Stability of the rate monotonic rule | 60 |
| 2.8 | Conclusion | 60 |
| | Bibliography | 60 |
| 3 | Response times for static schedulers | 63 |
| 3.1 | Introduction | 64 |
| 3.2 | 1 st request for synchronous general deadlines | 65 |
| 3.3 | k^{th} request for asynchronous general deadlines | 67 |
| 3.3.1 | Computation of ρ_i^k | 72 |
| 3.4 | The worst case response time | 83 |
| 3.5 | The best case response time | 88 |
| 3.6 | Feasibility tests for general deadline systems | 94 |
| 3.6.1 | Synchronous case | 94 |
| 3.6.2 | Asynchronous case | 95 |
| 3.7 | k^{th} request with arbitrary deadlines | 97 |
| 3.7.1 | Computation of ρ_i^k | 101 |
| 3.8 | Schedulability tests | 103 |
| 3.8.1 | The worst case response time for arbitrary systems . . . | 103 |
| 3.8.2 | Synchronous systems | 104 |
| 3.8.3 | Asynchronous systems | 104 |
| 3.9 | Comparison on the various feasibility tests | 105 |
| 3.10 | Conclusion | 106 |
| | Bibliography | 106 |
| 4 | Dynamic Schedulers | 109 |
| 4.1 | Introduction | 110 |
| 4.2 | Simplified model of computation | 111 |
| 4.3 | The deadline driven scheduler | 111 |

| | | |
|----------|---|------------|
| 4.4 | Optimality | 114 |
| 4.5 | Feasibility intervals | 124 |
| 4.5.1 | Synchronous systems | 138 |
| 4.6 | Response times | 139 |
| 4.6.1 | Introduction | 139 |
| 4.6.2 | 1 st request for synchronous systems | 140 |
| 4.6.3 | k^{th} request for asynchronous general deadlines | 146 |
| 4.6.4 | Computation of ρ_i^k | 150 |
| 4.6.5 | k^{th} request for asynchronous arbitrary deadlines | 153 |
| 4.6.6 | Computation of ρ_i^k | 156 |
| 4.7 | Feasibility tests for asynchronous systems | 161 |
| 4.8 | Feasibility tests for synchronous systems | 164 |
| 4.8.1 | Feasibility of bounded general deadline synchronous task sets | 164 |
| 4.8.2 | Worst case response time computation | 170 |
| 4.9 | The Least Laxity First scheduling algorithm | 184 |
| 4.10 | The (non-)stability of dynamic priority rules | 190 |
| 4.11 | Conclusion | 190 |
| | Bibliography | 192 |
| 5 | Offset free systems | 195 |
| 5.1 | Introduction | 196 |
| 5.2 | Offset granularity | 199 |
| 5.3 | Non-equivalent asynchronous systems | 204 |
| 5.4 | Non-optimality of monotonic schedulers | 210 |
| 5.4.1 | Definitions and properties | 212 |
| 5.4.2 | Optimality in special cases of offset free systems | 213 |
| 5.4.3 | Non-optimality of monotonic schedulers | 217 |
| 5.5 | Optimality of dynamic schedulers | 221 |
| 5.6 | Practical interest of offset free systems | 221 |
| 5.7 | Optimal offset assignment | 224 |

| | | |
|----------|--|------------|
| 5.7.1 | Two tasks | 226 |
| 5.7.2 | n tasks | 228 |
| 5.8 | Dissimilar offset assignment | 229 |
| 5.9 | Conclusion | 237 |
| | Bibliography | 237 |
| 6 | Conclusion | 241 |
| | List of Symbol | 245 |
| | Bibliography | 247 |
| | Index | 259 |

Chapter 1

Introduction

*Le temps est un grand maître, dit-on.
Le malheur est qu'il tue ses élèves.
— Hector Berlioz, Almanach des lettres françaises et étrangères.*

Contents

| | | |
|------------|---|-----------|
| 1.1 | Real-time systems | 2 |
| 1.2 | Tasks | 3 |
| 1.2.1 | Task constraints | 4 |
| 1.2.2 | Requests | 5 |
| 1.2.3 | Graphical conventions | 6 |
| 1.3 | Scheduling problem | 8 |
| 1.4 | Model of computation | 10 |
| 1.4.1 | Periodic task set | 10 |
| 1.4.2 | Schedule | 12 |
| 1.4.3 | Hypotheses | 13 |
| 1.5 | Overview of the thesis | 14 |
| | Bibliography | 15 |

1.1 Real-time systems

We shall consider in the framework of this thesis the scheduling problem of real-time systems. First, we must consider specific points that distinguish real-time systems from non real-time ones.

Real-time systems are computing systems which have timing constraints. Thus, the correctness of such a system depends not only on its logical results, i.e., it has to implement the intended algorithms, but also on the time at which the results are available. Real-time computing systems are widely used in many industrial applications. Examples of application domains that require real-time computing include

- Control of engines,
- Chemical and nuclear plant control,
- Traffic,
- Time-critical packet communications,
- Flight control systems,
- Railway switching systems,
- Robotics,
- Military systems,
- Space missions, and
- Virtual reality.

Figure 1.1 shows the architecture of a typical real-time system for controlling a *physical system*. We have the physical system, i.e., the system to be controlled. It can be a plant, a car, a robot, or any physical device that has to exhibit a desired behavior. The *control system* is the computing system which controls the system. The interactions between the physical system and the control system are in general bidirectional and occur by means of two peripheral subsystems: an *actuation system*, which modifies the physical system through a number of actuators (such as motors, pumps, etc.), and a *sensory system*, which acquires information from the physical system through a number of sensing devices (such as microphones, cameras, transducers, etc.).

It may be also noticed that a real-time system interacts with the physical system during the evolution of the latter. As a consequence, the system time

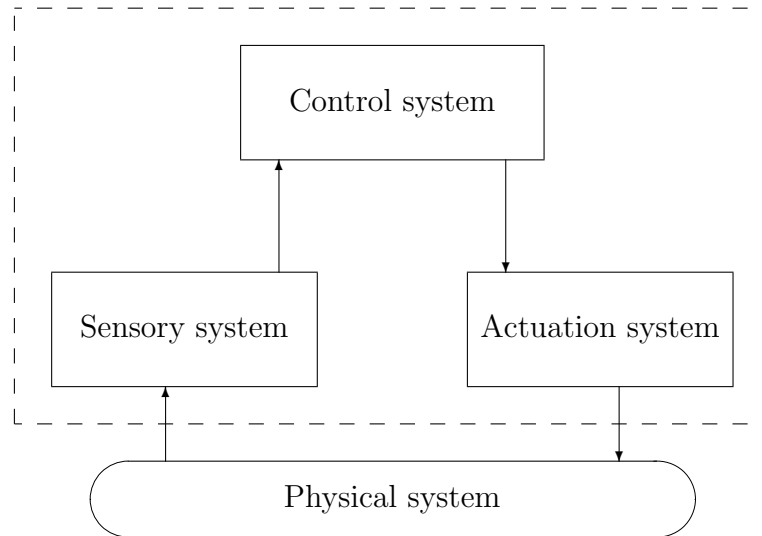


Figure 1.1: Real-time control system.

should be measured with the same time scale used for measuring the time in the controlled system. In several domains and especially in the operational research milieu, this notion is known as *on-line* systems. Remark that on-line systems are not necessarily real-time systems, since on-line systems do not necessarily have (hard) timing constraints.

In many real-time systems the consequences of a *failure*, e.g., the non respect of a timing constraint, can be catastrophic and lead to serious damage or even cost human lives. In this case we speak of real-time systems with *hard* (timing) constraints (*hard real-time systems*).

A major misconception about real-time systems is to think that they are equivalent to fast systems. Of course, minimizing the computation duration is helpful in satisfying the timing constraints, but it is not enough to meet all hard timing constraints. Instead of ensuring fast computation, in real-time systems we are concerned with a most important principle, called the *predictability* [SR90], i.e., the ability to predict, a priori, whether the system can meet all hard (also termed *critical*) timing requirements.

1.2 Tasks

The main part of a real-time system consists of *tasks*, i.e., of computer *processes*. A task is a computation that is executed by the CPU in a sequential

fashion: it is a sequential execution of code that does not suspend itself during execution. In the previous section we have introduced the constraint notion, characterizing real-time systems, these constraints concern more precisely the tasks.

1.2.1 Task constraints

Mainly, we can distinguish three kinds of constraints: timing constraints, precedence constraints and resource requirements.

Timing constraints

We have seen in the previous section that timing constraints are particular to real-time systems. This kind of constraint expresses a constraint on the time at which the computation of a task ends. These constraints are generally expressed by associating a *deadline* to the task. The deadline of a task represents the time before which the computation should complete; we can distinguish between two kinds of deadlines:

- If meeting a task deadline is critical for the system functionality, then the deadline is said to be *hard*; missing a hard deadline is considered a definite failure, and leads to catastrophic consequences.
- If it is desirable to meet a task deadline, but occasionally missing it can be tolerated (but with a certain cost, so that it is preferable to maintain these faults at the lowest possible level), then the deadline is said to be *soft*; a task with a soft deadline is expected to be completed either before the deadline or as early as possible after it. It may be noticed that for some task, called *soft tasks*, the deadlines are not specified but the system will simply respond as fast as possible while respecting the specified deadlines for the other tasks.

Precedence constraints

In some particular systems, the tasks have to respect some precedence relations. There is a precedence relation between two tasks (say τ_i and τ_j) if the computation of a task (say τ_j) must be started only (x time units) after the termination of the first one (say τ_i); in this case, task τ_i is said to be a *predecessor* of task τ_j .

Resource requirements

We have implicitly assumed in the previous section to have a single kind of resource: the CPU. It may be noticed that we have to distinguish between mono-processor and multi-processor systems, according to the number of processors in the system (respectively 1 processor and m processors, $m > 1$).

We may have to consider also other kinds of resources like: shared memory, message queue, exclusive access to a file, peripheral device, etc. If the various tasks share some resources (beside the CPU(S)), we speak of *dependent* tasks; in the other case we speak of *independent* tasks. In order to share the various resources and maintain data consistency, the dependent tasks are synchronized, for instance by ADA rendez-vous, semaphores or monitors.

1.2.2 Requests

Real-time systems are composed of a finite number of tasks, but each task may re-occur infinitely often, corresponding to activities in the real-time system. Examples of hard activities, i.e., activities with hard timing constraints, which may be present in a control application, include

- Sensory data acquisition,
- Detection of critical conditions,
- Actuator servoing, and
- Control of system components.

Examples of soft activities include

- The command interpreter of the user interface,
- Handling input data from the keyboard,
- Graphical activities, and
- Abnormal events like handling non fatal errors.

Hence a same task may arrive infinitely often, at different time instants. For this reason we have to distinguish between the task and its *requests*. The task is characterized by its resource needs (e.g., its computation time) and constraints

(e.g., its deadline); a *request of a task* (or a *task request*) corresponds to a specific occurrence of the task. It may happen that various requests of a same task coexist. The characteristics of a request vary during the evolution of the time; in other words, the characteristics of a request are dynamic (e.g., the remaining processing time, the time available before reaching the deadline, etc.).

1.2.3 Graphical conventions

In this section we introduce the graphical conventions used in this work to represent the execution of the various task requests. We suppose that the system execution starts at time $t = 0$ and we represent (with some graphical notations) what happens for each time unit (from time unit 0). The time units are numbered by the order given by the natural numbers, from the value 0.

Consider first, the simple case where the system is composed of a single task. We shall take the convention that, if the task is executing during the interval $[a, b]$ (and consequently during $c = b - a + 1$ consecutive time units), we represent this execution by the picture:



In the special case where $a = b$ (and consequently the task is executing during 1 time unit), we omit b in our representation and we represent this execution by the picture:



Suppose now to have several tasks (say τ_i and τ_j), in order to distinguish between the execution of τ_i and τ_j , we represent the execution of each task on a separate level, the time units of each level coincide as exhibited in Figure 1.2 where τ_i is executing during the interval $[1, 2]$ and τ_j during the interval $[3, 4]$.

In this graphical convention we assume that there is at most one request of each task at each time instant; later, we shall refine the conventions to represent more general behaviors.

Moreover, we represent a task request which occurs at time t by the picture \downarrow where the symbol \downarrow is placed at the beginning (i.e., on the left) of the time unit number t . We represent the task deadline which occurs at time t by the

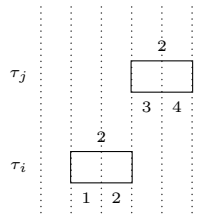


Figure 1.2: Graphical representation of the execution of two tasks.



Figure 1.3: Two requests of a single task, the first one occurs at time 0, the second one at time 10.

picture \circ where this symbol is placed at the beginning of the time unit number t . A deadline failure which occurs at time t is represented by the symbol \uparrow which is placed at the beginning of the time unit number t .

Now, we have the material to illustrate the notion of task request.

Example 1.1 Consider a real-time system composed of a single resource (the CPU) and a single task characterized by a computation time of 3 time units and a hard deadline of 5 time units. Figure 1.3 shows two requests of this task; the requests are numbered by increasing values of their arrival time, hence we distinguish the first request which occurs at time 0 and whose computation must be terminated before or at time 5 (the corresponding deadline); the second request occurs at time 10, the computation must be finished before or at time 15. The first request executes its computation during time $t \in [0, 2]$, i.e., during 3 consecutive time units; the second request executes its computation during time $t \in [10, 12]$. Hence, both requests meet their deadlines. ■

We shall distinguish between three kinds of real-time tasks, depending on the arrival pattern of their requests:

- *Periodic task.* The requests of a periodic task arrive regularly: two successive requests are separated by exactly the same time delay, called the *period*. The deadline of a periodic task is usually hard.

- *Sporadic task.* The requests of a sporadic task arrive irregularly with each arrival separated from the preceding one by at least a certain time delay called the *minimum separation time*. The deadline of a sporadic task is usually soft.
- *Aperiodic task.* The requests of an aperiodic task arrive irregularly with no minimum separation time. It may be noticed that this kind of task cannot have a hard deadline since no guarantee can be made for them. The deadline can be soft (if any) or not specified.


1.3 Scheduling problem

The general scheduling problem is to allocate available resources (e.g., the CPU) over a period of time to perform a set of tasks so as to achieve some performance measure (see [Bak74, Cof76] for an introduction to the theory of scheduling). The goal of the scheduling of hard real-time systems (also termed the *hard real-time scheduling*) is quite different, the main objective being to allocate resources in order to always meet hard (timing) constraints, to minimize the penalties for soft timing constraints and to ensure good performances for soft tasks. Although the theory of scheduling is born in the operational research domain, the scheduling of hard real-time systems seldom appears in the operational research literature. The reason is that the problem is of a different nature; classical approaches are well suited to maximize (or minimize) a performance measure but not to guarantee that the hard constraints are always met, i.e., the predictability of the system. Moreover, the classical approach concerns mainly the scheduling of tasks that have to be executed once instead of infinitively often.

In the framework of this thesis we are mainly concerned by the scheduling of mono-processor systems composed of independent tasks. Interestingly, as we shall see, even with such a simple setting the problems are far from obvious. In this situation, the scheduling algorithm can be understood as a rule which selects, at each instant, a single request among all *active* requests (if any), i.e., requests which are waiting for the CPU in order to start/continue their execution. During the life of the system, every time the system considers the CPU allocation, the processor is assigned to the request selected by the scheduling algorithm. This request executes its computation until the next time instant where the system considers the CPU allocation, or until the completion of its computation. If there is no active request, the processor is not assigned to any request and the CPU is said to be *idle*.

Among the various scheduling algorithms proposed for the scheduling of real-

time tasks, we can distinguish between the following approaches.

- *Static.* The scheduling algorithm computes the priorities of the tasks beforehand, based on the task characteristics (e.g., the worst case execution time, the deadline, the period, etc.). The priorities are then assigned to each task before the activation of all tasks. During the execution of the system, the system selects the highest priority request, i.e., the active request which corresponds to the highest priority task (if there are many active requests corresponding to this task, generally the oldest one is selected).
- *Dynamic.* The scheduling algorithm computes the priorities during the execution of the system. The priority of each active request is based on the system state, e.g., the current time, the request characteristics (e.g., the remaining execution time of each request, the time available before reaching the deadline), etc. It follows that the priority of a task or a request may change during system evolution.
- *Cyclic executive.* The scheduling algorithm prepares beforehand a table (this table is known as the *cyclic schedule* [BS88]) which determines when requests are scheduled. During the execution of the system this table is followed cyclicly. Hence, the cyclic executive approach needs to store a table during the execution of the system; the size of this table is generally proportional to the least common multiple (divided by the greatest common divisor) of all periods (of periodic tasks) and consequently increases considerably with the number of tasks in the system. Moreover, sporadic and aperiodic tasks prove to be a problem for the cyclic executive, since the time at which a sporadic/aperiodic request occurs is unknown during the construction of the table. Priority based schedulings (static and dynamic) are more flexible in comparison with the cyclic executive. For this reason, the priority approaches are more popular and more used in practice than the cyclic executive approach. Moreover the cyclic executive is based on the worst case execution time and cannot handle situations where the execution times vary.
- *Preemptive.* With preemptive algorithms, a request may be interrupted at any time to give the CPU to an active request with a higher priority. Note that in this case, the execution of a request may be suspended (by a higher priority one) and resumed later, consequently the execution of such a request is represented by several blocs with the shape: 

- *Non-preemptive.* With non-preemptive algorithms, a request, once assigned to the CPU, is executed until its completion. In this case, the scheduling algorithm is used when a task ends its execution, and also when one or more new requests occur while the CPU is idle.

1.4 Model of computation

*Il n'y a pas de grandeur pour qui veut grandir,
il n'y a pas de modèle pour qui cherche ce qu'il n'a jamais vu.
— Paul Eluard, L'Evidence poétique.*

We have introduced in the previous sections the scheduling problem of real-time systems in general. We shall here state precisely the scheduling problem considered in the framework of this thesis and the related model of computation.

In many real-time control systems, periodic activities represent the major computational demand in the system. Periodic tasks typically arise from sensor data acquisition, system monitoring, detection of critical condition, etc. Such activities need to be cyclically executed at a specific rate. We shall consider this kind of real-time system and focus our study on the scheduling of periodic task sets for mono-processor systems. In this case the scheduling algorithm has to guarantee that each periodic request completes its execution within its deadline.

In order to analyze this problem rigorously, we first need to construct a mathematical model describing the relevant aspects of the system.

1.4.1 Periodic task set

We consider real-time systems constituted of a set of n periodic tasks: τ_1, \dots, τ_n . Each periodic task (say τ_i , $1 \leq i \leq n$) is characterized by the quadruple (T_i, D_i, C_i, O_i) with $0 < C_i \leq D_i$, $C_i \leq T_i$ and $O_i \geq 0$, i.e., by a period T_i , a hard deadline D_i , an execution time C_i , and an offset O_i , giving the instant of the first request. The requests of τ_i are separated by T_i time units and occur at time $O_i + (k - 1)T_i$ ($k = 1, 2, \dots$). The execution time required for each request is C_i time units; C_i can be considered as the worst-case execution time for a request of τ_i . The execution of the k^{th} request of task τ_i , which occurs

at time $O_i + (k - 1)T_i$, must finish before or at time $O_i + (k - 1)T_i + D_i$; the deadline failure is fatal for the system: the deadlines are considered to be hard. Without loss of generality we can assume that $\min\{O_i | i = 1, \dots, n\} = 0$.

All timing characteristics of the tasks in our model of computation are assumed to be multiples of the “CPU tick”, the smallest indivisible CPU time unit. We assume that all task characteristics are natural integers.

Synchronous, asynchronous and offset free task sets

From a theoretical as well as from a practical point of view, it is interesting to distinguish between three classes of periodic task sets, regarding the offsets: *synchronous*, *asynchronous* and *offset free* task sets.

- **Synchronous systems.** The offsets are fixed by the constraints of the system and they are all the same, i.e., $O_1 = O_2 = \dots = O_n = 0$.
- **Asynchronous systems.** The offsets are fixed by the constraints of the system, but the tasks are not started at the same time: the offsets are different.
- **Offset free systems.** In such systems there is no definite requirement about the task start times. Hence, the offsets will be chosen beforehand by the scheduling algorithm itself. We shall consider specifically this kind of systems in Chapter 5.

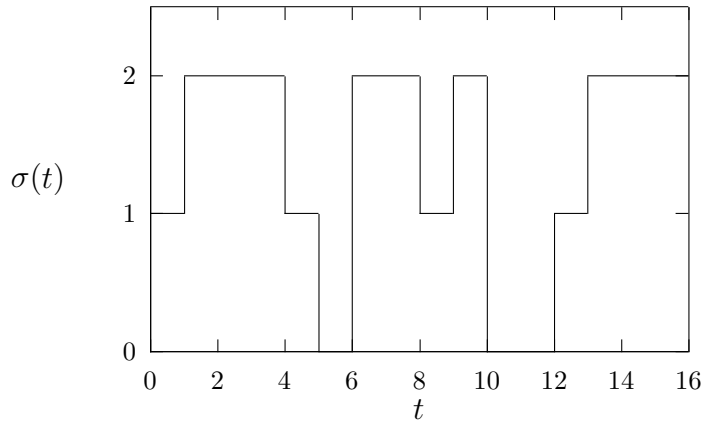
Relation between the period and the deadline

We shall also distinguish three classes of periodic task sets regarding the relation between the period and the deadline of each task: the *late deadline* case, the *general deadline* case and the *arbitrary deadline* case.

- **Late deadline** case corresponds to the case where the deadline of each task coincides with the period ($T_i = D_i$, $i = 1, \dots, n$). In this case, each request must simply be completed before the next request (of the same task) occurs. Since $T_i = D_i$, we shall omit the representation of the deadlines in the graphics for this kind of systems.
- **General deadline** case corresponds to the case where the deadlines are not greater than the periods: ($D_i \leq T_i$, $i = 1, \dots, n$).

| | T_i | D_i | C_i | O_i |
|----------|-------|-------|-------|-------|
| τ_1 | 4 | 4 | 1 | 0 |
| τ_2 | 6 | 5 | 3 | 0 |

Table 1.1: Characteristics of a periodic task set.

Figure 1.4: Representation of the function $\sigma(t)$; we assume that from time $t = 12$, the function repeats.

- **Arbitrary deadline** case corresponds to the case where no constraint exists between the deadline and the period: the deadline of a task τ_i may be less ($D_i \leq T_i$) or greater ($D_i > T_i$) than the period; in the latter situation, many requests of a same task may coexist at some instants.

1.4.2 Schedule

The scheduling problem is in our case to produce a schedule according to which all task requests can be executed while meeting their deadlines. Given a set of tasks, $\tau_1, \tau_2, \dots, \tau_n$, a *schedule* is an assignment of requests to the processor. More formally, a schedule can be defined as a function $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$. In other words, $\sigma(t)$ is an integer function and $\sigma(t) = k$, with $k > 0$, means that a request of task τ_k is executing at time t , while $\sigma(t) = 0$ means that the CPU is idle. In the late deadline and general deadline situations, since there is at most one active request for each task at each instant, this defines unambiguously the schedule. In the arbitrary deadline case it may be necessary to add a rule (like attributing the CPU to the oldest active request of task $\tau_{\sigma(t)}$), or an extra function, to lift ambiguities.

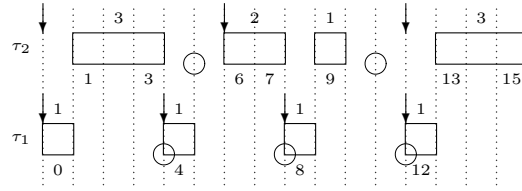


Figure 1.5: Representation of a schedule; we assume that from time $t = 12$ the schedule repeats.

Example 1.2 Figure 1.4 shows an example of the function $\sigma(t)$ for scheduling the task set given by table 1.1; we suppose that the function $\sigma(t)$ repeats from time 12. Another representation of a schedule, including task requests and task deadlines can be found in Figure 1.5; here and later, the meaning of the various signs is the same as in Figure 1.3; it may be noticed that we shall preferably use this kind of representation for convenience in this work. ■

We have to distinguish between two kinds of schedules: *feasible* and *unfeasible* schedules.

- **Feasible schedule:** it is a schedule where all requests meet their deadline. A feasible schedule is defined for all non-negative time instants.
- **Unfeasible schedule:** in such a schedule a request misses its deadline (say at time f); we shall take the convention that in this case the schedule is defined for time instants in the interval $[0, f)$. In some particular circumstances we shall also define the schedule after (or at) a deadline failure.

Given the specification of a (periodic) task set and a (non ambiguous) scheduling rule, the schedule (i.e., the function $\sigma(t)$) is then univocally determined. For instance, the schedule given by Figure(s) 1.4 (and 1.5) can be produced by applying the static and preemptive scheduler on the task set given by table 1.1, where τ_1 has a higher priority than τ_2 .

1.4.3 Hypotheses

In our analysis the following hypotheses are assumed.

- We consider preemptive scheduling algorithms and we assume that switching times (including scheduling) may be neglected.

- Successive requests of a periodic task τ_i are exactly separated by the period T_i of the task.
- All requests of a periodic task τ_i have the same execution time C_i .
- All tasks are independent; there are no precedence relations nor resource constraints.

1.5 Overview of the thesis

In this section we give an overview of the contents of the rest of this thesis.

Chapter 2: We study static priority scheduling algorithms. We first present the rate/deadline monotonic priority assignments, we complete and correct the theory, in particular concerning the optimality of these priority rules. We then present our results concerning feasibility intervals for the various sub-classes of periodic task sets considered in this work, i.e., synchronous/asynchronous regarding the offsets and late/general/arbitrary deadline regarding the deadlines.

Chapter 3: We study the response time notion. We extend the theory to handle the response time computation of all sub-classes of periodic task sets considered in this work. We show the interest of this notion and we exploit these results for the feasibility and the schedulability of periodic task sets.

Chapter 4: We study dynamic priority scheduling algorithms. First, we present the deadline driven scheduler, and we complete and correct the theory. Then, we consider feasibility intervals for the dynamic deadline driven scheduler and we extend the theory to handle the various sub-classes of periodic task sets considered in this work. We extend the computation of the response time to the dynamic case, for the various kinds of periodic task sets considered in this work. We show the interest of our computation regarding previous results of the literature; in particular we show that our computation exhibits an exponential improvement in comparison with the works of Spuri and Baruah et al.

Chapter 5: We consider offset free systems. We first show the interest of this kind of systems, based on the study presented in previous chapters. We consider first the equivalence relationship between offset assignments. We study then the optimality of popular scheduling algorithms for offset free systems. In particular, we show the non-(weak) optimality of monotonic priority assignments for offset free systems and the (strong) optimality of dynamic scheduling algorithms. We then consider the offset assignment problem; first we propose a

method, based on the equivalence relationship, to reduce the number of offset assignments in the optimal assignment, then we present a heuristic rule which is a (pseudo-) polynomial time algorithm to compute the offsets.

Chapter 6: In this chapter we present concluding remarks; these ones concern mainly the contribution of our work, which includes a review (with some corrections and gap fillings) of the theory and its extension to handle more general classes of periodic task sets, including asynchronous and arbitrary deadline systems; the extension of the response time notion and its computation for more general classes of periodic task sets, its theoretical as well as its practical interest for static and dynamic schedulers, and in particular to obtain more efficient feasibility tests; the interest to consider more optimistic cases than the synchronous one and hence the interest of offset free systems. Suggestions for further researches are included in the conclusions of each chapter.

Bibliography

- [Bak74] K. R. Baker. *Introduction to sequencing and Scheduling*. John Wiley & Sons., 1974.
- [BS88] T. P. Baker and Alan Shaw. The cyclic executive and ada. *IEEE Computer Society Press*, pages 120–129, 1988.
- [But97] Giorgio C. Buttazzo. *Predictable Scheduling Algorithms and Applications*. Hard Real-Time Computing System. Kluwer Academic Publishers, 1997.
- [Cof76] E.G. Jr. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, 1976.
- [Kor92] Jan Korst. *Periodic Multiprocessor Scheduling*. PhD thesis, Technische Universiteit Eindhoven, 1992.
- [SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1990.
- [vTK91] André M. van Tilborg and Gary M. Koob. *Scheduling and Resource Management*. Foundations of Real-Time Computing. Kluwer Academic Publishers, 1991.

Chapter 2

Static Schedulers

*Le commencement de toutes les sciences,
c'est l'étonnement de ce que les choses sont ce qu'elles sont.
— Aristote, Métaphysique, I, 2.*

Contents

| | | |
|------------|--|-----------|
| 2.1 | Introduction | 20 |
| 2.2 | The rate monotonic scheduler | 20 |
| 2.2.1 | Optimality | 21 |
| 2.2.2 | Feasibility tests | 27 |
| 2.3 | The deadline monotonic scheduler | 34 |
| 2.3.1 | Optimality | 34 |
| 2.3.2 | Feasibility tests | 36 |
| 2.4 | Synchronous arbitrary deadline systems | 36 |
| 2.5 | Asynchronous general deadline systems | 43 |
| 2.6 | Asynchronous arbitrary deadline systems | 52 |
| 2.7 | Stability | 59 |
| 2.7.1 | Stability of the rate monotonic rule | 60 |
| 2.8 | Conclusion | 60 |
| | Bibliography | 60 |

2.1 Introduction

We shall consider in this chapter the first priority based scheduling family, i.e., the static priority schedulers. We consider the scheduling of a periodic task set: $\tau_1, \tau_2, \dots, \tau_n$; each task has a static and distinct priority; we take the convention that the notation $\tau_i > \tau_j$ means that task τ_i has a higher priority than task τ_j . We assume that each request of task τ_i “inherits” the priority of the corresponding task (τ_i) and that the priority of each request does not change with time. Without loss of generality we can assume that τ_1 is the highest priority task of the system, τ_n the lowest one and tasks with intermediate index have intermediate priorities; more formally we have: $\tau_1 > \tau_2 > \dots > \tau_n$. For this family of scheduling algorithms the definition of the schedule (i.e., the function $\sigma(t)$) can be refined. Since the priorities of the requests are static, a task request, once assigned to the CPU, is executed until a higher priority request occurs or the request ends its computation; in the last case the system selects the highest priority active request, if any. Since, the task characteristics are natural numbers, the scheduling rule is invoked at natural time instants corresponding to task requests and completion of requests. It follows that function $\sigma(t)$ changes its value at natural time instants; hence we can restrict the schedule as follows: $\sigma : \mathbb{N} \rightarrow \mathbb{N}$. Consequently, $\sigma(t)$ is an integer function and $\sigma(t) = k$, with $k > 0$, means that a¹ request of task τ_k is executing at time t during one time unit, while $\sigma(t) = 0$ means that the CPU is idle at time t (during one time unit).

The remainder of the chapter is organized as follows: in section 2.2 we present the rate monotonic scheduler, one of the most popular static scheduling algorithms; in section 2.3 we present the deadline monotonic scheduler, an extension of the rate monotonic scheduler; in sections 2.5, 2.4 and 2.6 we consider sub-classes of periodic task sets, more general than those for which the monotonic priority assignments were initially defined.

2.2 The rate monotonic scheduler

We study first the *rate monotonic scheduler*, one of the most popular static priority rules. The rate monotonic scheduler (RMS) was defined by Liu and Layland [LL73] and Serlin [Ser72] (termed the *intelligent fixed priority algorithm* -IFP- by Serlin); this scheduler was defined for a sub-class of periodic task sets:

¹We suppose here there are never two active requests for a same task at the same time instant (e.g., this is the case if $D_i \leq T_i, \forall i$). We shall see later how to relax this assumption.

- The rate monotonic scheduler is defined for synchronous systems, i.e., all tasks are started at the same time ($O_1 = O_2 = \dots = O_n = 0$), and
- for late deadline systems, i.e., the deadline of each task coincides with the period ($D_i = T_i, i = 1, \dots, n$).

We shall assume, in the first part of this chapter, that we schedule only synchronous and late deadline systems. More general classes of periodic task sets (e.g., asynchronous systems) will be handled later.

The rate monotonic scheduler is a static and preemptive scheduler, which assigns a priority to each task in inverse proportion of its arrival rate. More formally: given two tasks τ_i, τ_j and T_i, T_j their respective periods, if the task τ_i has a higher priority than τ_j ($\tau_i > \tau_j$) then $T_i \leq T_j$. It may be noticed that for task systems with some identical task periods (e.g., $T_i = T_j, i \neq j$) there are several rate monotonic priority assignments; the tie (if any) may be broken in an arbitrary way. The rate monotonic scheduler may therefore be ambiguous: it does not give a unique priority assignment if several periods are identical.

Computing the priorities of a set of n tasks for the rate monotonic priority rule amounts to ordering the task set according to their periods. Hence the time complexity of the rate monotonic priority assignment is the time complexity of a sorting algorithm, typically: $O(n \log n)$.

2.2.1 Optimality

A main property of the rate monotonic priority rule is its *optimality*. Let us first define what optimality means in this case.

Definition 2.1 A task set is said *schedulable* for a priority assignment if all deadlines of all task requests are met. ■

Definition 2.2 A priority assignment is said *feasible* if, with this priority assignment, the task set is schedulable. ■

Classically, optimality in the framework considered here is defined thus.

Definition 2.3 A static priority assignment rule R is *optimal* for synchronous and late deadline systems if, when a feasible priority static assignment exists for some synchronous and late deadline task set, the priority assignment given by the rule R is also feasible for that task set. ■

However, this only makes sense if R is unambiguous. Since many priority assignments, and in particular the rate monotonic priority assignment as defined by Liu and Layland, may be ambiguous, we shall refine the optimality notion and distinguish two kinds of optimality.

Definition 2.4 A static priority assignment rule R is *strongly optimal* for synchronous and late deadline systems if, when a feasible priority static assignment exists for some synchronous and late deadline task set, any priority assignment given by the rule R is also feasible for that task set, whatever the way in which the ambiguities are resolved. ■

Definition 2.5 A static priority assignment rule R is *weakly optimal* for synchronous and late deadline systems if, when a feasible priority static assignment exists for some synchronous and late deadline task set, some priority assignment given by the rule R is also feasible for that task set. ■

In other words, if a task set is unschedulable with a static and (weakly) optimal priority rule, it will also be unschedulable for any other static priority assignment.

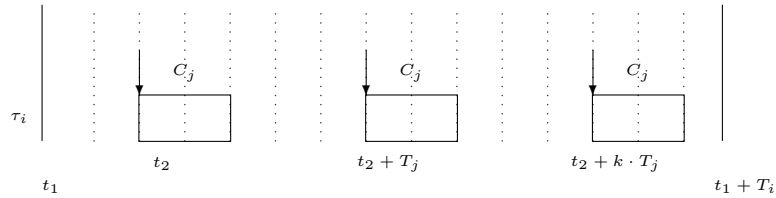
Liu and Layland [LL73] have “shown” the (strong) optimality of the rate monotonic scheduler for synchronous and late deadline systems. The optimality result given by Liu and Layland is based on the fact that the synchronous case is the worst case from a schedulability point of view. But this property was not really proved in their paper, so that their argument is not fully satisfactory. This property is based on the response time notion.

Definition 2.6 For a request of task τ_i , we define the *response time* as the time between the arrival of the request and the completion of its processing. ■

According to this definition, the property stated by Liu and Layland can be formulated as follows:

Conjecture 2.7 Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic synchronous task set with late deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The response time of the first request of any task τ_i is maximum among all requests of task τ_i . ■

Proof from Liu and Layland [LL73]. Let $\tau_1, \tau_2, \dots, \tau_n$ denote a set of periodic synchronous tasks with late deadlines. Consider a particular request

Figure 2.1: Requests of task τ_j in the interval $[t_1, t_1 + T_i]$.

of τ_i that occurs at time t_1 . Suppose that between t_1 and $t_1 + T_i$, the time at which the subsequent request of τ_i occurs, requests for task τ_j , $j < i$, occur at time $t_2, t_2 + T_j, \dots, t_2 + k \cdot T_j$ as illustrated in Figure 2.1. Clearly, the preemption of τ_i by τ_j will cause a certain amount of delay in the completion of the request for τ_i that occurred at time t_1 , unless the request for τ_i is completed before t_2 . Moreover, from Figure 2.1 we see immediately that advancing the request time t_2 will not speed up the completion of τ_i . The completion time of τ_i is either unchanged or delayed by such advancement. Consequently, the delay in the completion of τ_i is largest when t_2 coincides with t_1 . Repeating the argument for all τ_j , $j < i$, we prove the property. ■

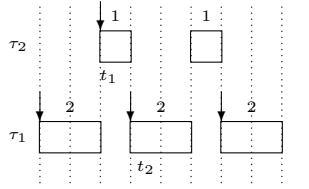
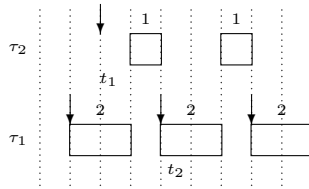
The reasoning of Liu and Layland is incorrect: in their proof, the authors do not consider the requests of task τ_j which occurs before time t_1 (only the nearest one matters, in fact). With the following example we show that the argument of the preceding proof is incorrect.

Example 2.8 Consider the task set given by table 2.1 and consider a request of τ_2 as illustrated in Figure 2.3 (for late deadline systems we do not display the deadlines since these ones coincide with the requests); the response time of this request is 5 time units, and advancing the request of task τ_1 by one time unit speeds up the response time of this request of task τ_2 , since in this case the response time is 4, as illustrated in Figure 2.2. ■

This first remark concerning a classical result of the literature, and in particular the wrong argument used in the proof of Liu and Layland, shows that we must be very careful; our intuition may lead to incorrect reasonings, for the kind of systems we consider in this work; even in very “simple” cases (e.g., synchronous

| | T_i | D_i | C_i |
|----------|----------|-------|-------|
| τ_1 | 3 | 3 | 2 |
| τ_2 | ≥ 5 | T_2 | 2 |

Table 2.1: Characteristics of a periodic task set.

Figure 2.2: Response time of τ_2 , $t_2 - t_1 = 1$.Figure 2.3: Response time of τ_2 , $t_2 - t_1 = 2$.

and late deadline systems with 2 periodic tasks) it is difficult to anticipate their behavior. We shall see that this is often the case, for the various kinds of problems considered during this work.

Nevertheless, while their argument was wrong, Liu and Layland had the correct intuition and Conjecture 2.7 is indeed valid. Remark that Example 2.8 does not contradict Conjecture 2.7 since the response time of the first request of τ_2 in the synchronous case is 6, which is greater than the response time in both asynchronous situations considered above.

We shall not give a proof of Conjecture 2.7 in this chapter, since our proof is based on the general response time computation, which is the purpose of Chapter 3. We suggest the reader to consider this property as a conjecture till the presentation of our proof in Chapter 3 (section 3.4). More precisely, we shall consider a more general Conjecture.

Conjecture 2.9 Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic synchronous task set with general deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The response time of the first request of task τ_i is maximum among all requests of task τ_i . ■

Notice that, as we shall later see, the Conjecture may not be extended to the arbitrary deadline case.

We come back to the optimality of the rate monotonic scheduler. Even if we accept the Conjecture 2.9, the optimality proof of Liu and Layland is still not

fully convincing. We shall here give a complete proof; in particular we first show two preliminary results, not considered in the work of Liu and Layland.

Definition 2.10 A scheduler is said to be *expedient* iff when active request(s) exist(s) at some time t , the CPU is given to some active request and consequently $\sigma(t) \neq 0$. ■

Expedience seems to be a natural property to be imposed on any preemptive scheduling (while for non-preemptive scheduling it may be wise to stay idle for a while in order to be able to start a critical request, which would be delayed if we started a long non-critical request before).

We supposed in the previous definition that the CPU allocation was considered at each natural time instant ($\sigma(t) : \mathbb{N} \rightarrow \{0, \dots, n\}$). We then have a very general property, stated as follows.

Lemma 2.11 Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic and asynchronous task set, with arbitrary deadlines. Let A_1 and A_2 be two feasible scheduling rules for this set. Consider the schedules $\sigma_1(t)$ and $\sigma_2(t)$ defined by the rule A_1 and A_2 , respectively. If A_1 and A_2 are expedient, it follows that $\forall t \in \mathbb{N} : \sigma_1(t) = 0$ iff $\sigma_2(t) = 0$.

Proof. We show the property by induction on time t . The property is obvious in the base case, $t = 0$, since $\min\{O_i | i = 1, \dots, n\} = 0$ and $\sigma_1(0) \neq 0 \neq \sigma_2(0)$, since at time 0 at least one task makes a request and needs the CPU for at least one time unit, and A_1 as well as A_2 are expedient. Suppose the property is true till time $t \geq 0$ and consider the instant $t + 1$. If the property is no longer true at time $t + 1$, we have without loss of generality: $\sigma_1(t + 1) = 0$ and $\sigma_2(t + 1) \neq 0$. We shall show that this situation is impossible and leads to a contradiction. Let t'_1 be the largest time instant strictly less than $t + 1$ such that $\sigma_1(t'_1) = 0$, with $t'_1 = -1$ if the CPU remains busy from time 0; t'_2 is defined in the same way for the schedule $\sigma_2()$. By induction hypothesis, we have that $t'_1 = t'_2$. Since the CPU is idle in both schedules at time t'_1 we can only consider the requests from time t'_1 . Since in $\sigma_2()$ the CPU remains busy in the interval $[t'_1 + 1, t + 1]$ the demand² in this interval is greater than or equal to $t - t'_1 + 1$, and since the demand does not rely on the scheduling rule, with rule A_1 the (same) demand may not be satisfied either at time $t + 1$. Since A_1 is expedient, we have that $\sigma_1(t + 1) \neq 0$, a contradiction. ■

²The demand in an interval $[a, b]$ is $\sum_{i=1}^n m_i \cdot C_i$, where m_i is the number of requests of τ_i which occur in $[a, b]$.

Notice that the property also holds for systems composed of aperiodic and periodic tasks (an aperiodic task may be considered as an asynchronous periodic task with infinite period).

We consider now a second preliminary result, not considered in the work of Liu and Layland.

Lemma 2.12 *Let S be an asynchronous system with arbitrary deadlines and $\tau_1 > \tau_2 > \dots > \tau_n$ be a feasible static priority assignment. For any priority assignment $\tau_{k_1} > \tau_{k_2} > \dots > \tau_{k_{n-1}} > \tau_n$, with $\{k_1, \dots, k_{n-1}\} = \{1, \dots, n-1\}$, feasible for the task sub-set $\{\tau_1, \dots, \tau_{n-1}\}$, the CPU allocation for any request of task τ_n remains unchanged with respect to the $\tau_1 > \dots > \tau_n$ priority assignment, and feasibility is preserved for the whole task set.*

Proof. With the priority assignment $\tau_1 > \dots > \tau_n$, the set is schedulable; for any $k > 0$, the k^{th} request of τ_n receives the first C_n idle time units left by the requests of $\tau_1, \dots, \tau_{n-1}$ from time $O_n + (k-1)T_n$. Consider now a priority assignment such that $\tau_{k_1} > \tau_{k_2} > \dots > \tau_{k_{n-1}} > \tau_n$ with $\{k_1, \dots, k_{n-1}\} = \{1, \dots, n-1\}$, feasible for the task sub-set $\{\tau_1, \dots, \tau_{n-1}\}$. Each request of τ_n receives the first C_n idle time units left by the requests of $\tau_1, \dots, \tau_{n-1}$ from its arrival time, which are the same than for the priority assignment $\tau_1 > \tau_2 > \dots > \tau_{n-1}$ by Lemma 2.11 (static priority assignments are expedient). The property follows. ■

Lemma 2.12 shows that the schedulability of task τ_i depends on the set of all higher priority tasks, and not on the priority assignment chosen for this set.

It may be noticed that both Lemma 2.11 and Lemma 2.12 hold for a more general class of periodic task sets than the one for which the rate monotonic scheduler was defined, since these properties concern asynchronous systems with arbitrary deadlines. Moreover, these properties are not dedicated to the rate monotonic priority rule and concern any static priority assignment for Lemma 2.12, and any expedient rule for Lemma 2.11.

Theorem 2.13 *The rate monotonic priority rule is strongly optimal for synchronous and late deadline task sets.*

Proof. We must prove that if a feasible static priority assignment exists for a task set with late deadlines, any rate monotonic priority assignment is also feasible for that task set. Let $\tau_1, \tau_2, \dots, \tau_n$ be a synchronous and late deadline task set. Suppose there exists a feasible priority assignment ($\tau_1 > \tau_2 > \dots > \tau_n$). Let τ_i and τ_j be two tasks of adjacent priorities ($\tau_i > \tau_j$, $j = i + 1$) with $T_i \geq T_j$. Let us exchange the priorities of τ_i and τ_j : if the task set is still

schedulable, since any rate monotonic priority assignment can be obtained from any priority ordering by a sequence of such priority exchanges, we may deduce that any rate monotonic priority assignment is also feasible. The priority exchange does not modify the schedulability of the tasks with a higher priority than τ_i (i.e., $\tau_1, \dots, \tau_{i-1}$). The task τ_j remains of course schedulable after the priority exchange, since it may use all the free slots left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}\}$ instead of only those left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}, \tau_i\}$. Assuming that the requests of τ_i remain schedulable, from Lemma 2.11 the scheduling of each task τ_k , for $k = i+2, i+3, \dots, n$ is not altered since the idle periods left by higher priority tasks are the same. Hence we must only verify that τ_i also remains schedulable. From Conjecture 2.7 we can restrict this question to the first request of task τ_i . Let r_j be the response time of the first request of τ_j before the priority exchange: the feasibility implies $r_j \leq D_j$; it is not difficult to see that during the interval $[0, r_j)$ the CPU (when left free by higher priority tasks) is assigned first to the (first) request of τ_i then to the (first) request of τ_j (the latter is not interrupted by subsequent request of τ_i since $T_i > T_j = D_j \geq r_j$). Hence, after the priority exchange, the CPU allocation is exchanged between τ_i and τ_j , and it follows that τ_i ends its computation at time r_j and meets its deadline since $r_j \leq D_j = T_j \leq T_i = D_i$. ■

2.2.2 Feasibility tests

We shall now consider the *feasibility problem*, i.e., deciding if a (synchronous and late deadline) system is feasible (or not), i.e., if there exists a priority rule which makes the system schedulable. Since the rate monotonic priority rule is optimal for such systems, we can restrict this question, by considering the schedulability of the system using the rate monotonic priority rule.

From Conjecture 2.7 it follows that we only have to check if the first request of each task meets its (first) deadline.

Corollary 2.14 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic synchronous and late deadline task set. The set is feasible iff the first request of each task meets its deadline with the rate monotonic rule.*

Proof. Immediately follows from Theorem 2.13 and Conjecture 2.7. ■

We shall see in Chapter 3, explicit formulas and algorithms for the computation of the response time of the first request of task τ_i in the synchronous case, hence the interest of Corollary 2.14.

Liu and Layland have also defined a rather efficient sufficient condition for the schedulability of a task set, based on the utilization factor.

We define the *utilization factor* as the fraction of processor time spent in the execution of the task set:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (U \in \mathbb{Q}).$$

Suppose that the system starts at time 0. It may be noticed that the utilization factor does not give the CPU utilization for all instants. For each task (τ_i) considered alone, $\frac{C_i}{T_i}$ represents for all instants multiple of T_i the CPU utilization by task τ_i , if the task set is schedulable. For the task set itself, U gives the total utilization for all instants multiple of $P = \text{lcm}\{T_i | i = 1, \dots, n\}$. P is called the *hyper-period*: we shall see that (when feasible) the schedule is periodic from time 0 with a period of P (see Corollary 2.49). But it is not difficult to see that the utilization factor approximates the used fraction of the processor time as close as we want for instants sufficiently large (in comparison with the T_i 's); this remains true for asynchronous systems with arbitrary (but finite) deadlines and dynamic priorities. Lastly, the function U has a true sense only if all tasks always meet their deadlines, i.e., if the schedule is feasible.

In all generality, i.e., whatever the scheduling algorithm (static or dynamic), whatever the offsets (synchronous, asynchronous or offset free) and whatever the deadline configuration (late deadline, general deadline or finite arbitrary deadline) we have that a task set is certainly not schedulable if $\sum_{i=1}^n \frac{C_i}{T_i} > 1$. For the case of synchronous and late deadline case, Liu and Layland have shown that if the utilization factor is less than 69 % the set is always schedulable with the rate monotonic rule. Again however while the end result is correct, the reasoning of Liu and Layland to justify this feasibility test is incomplete; we shall present the completed results here.

Definition 2.15 A task set τ_1, \dots, τ_n is said to *fully utilize* the processor if the task set is schedulable and if an increase of any C_i ($1 \leq i \leq n$) makes the task set unschedulable. ■

Lemma 2.16 *Let τ_1, \dots, τ_n be a synchronous and late deadline task set, schedulable with the rate monotonic priority assignment ($\tau_1 > \tau_2 > \dots > \tau_n$). If the increase of C_n makes the task set unschedulable, the task set fully utilizes the processor.*

Proof. If we cannot increase C_n , that means that the interval $[0, T_n)$ does not contain idle time units (see Figure 2.4). Hence, we cannot increase any C_i , since $T_n \geq T_{n-1} \geq \dots \geq T_1$. ■

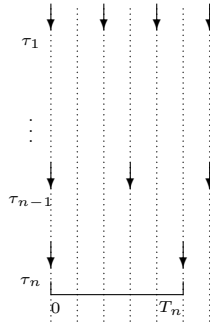


Figure 2.4: The interval $[0, T_n)$ with the rate monotonic priority assignment.

Definition 2.17 We define b_n as the least upper bound of the processor utilization factor over all the sets of n tasks which fully utilize the processor. ■

In the previous definition we have used the original definition of Liu and Layland, it may be noticed that b_n is in fact a lower bound regarding the utilization factor.

The sufficient feasibility test given by Liu and Layland is based on the computation of the bound b_n . Liu and Layland have stated that $b_n = n(\sqrt[n]{2} - 1)$ but this property was not really proved in their paper: the result is correct but their argument is not satisfactory.

The property stated by Liu and Layland can be formulated as follows:

Conjecture 2.18 For a set of n periodic synchronous tasks with late deadlines and the restriction that the ratio between any two task periods is less than 2, $b_n = n(\sqrt[n]{2} - 1)$, for the rate monotonic priority rule. ■

We give here the first part of the original proof of Liu and Layland, and we shall exhibit why their argument is incorrect.

Proof from Liu and Layland [LL73]. Let τ_1, \dots, τ_n denote the n tasks. Let C_1, \dots, C_n be the execution times of the tasks that fully utilize the processor and minimize the processor utilization factor (here we shall allow real-valued C_i 's, considering they may be arbitrarily approximated by rational values, hence by integer values up to a multiplication of all task characteristics by an adequate integer coefficient). We assume that $T_n > T_{n-1} > \dots > T_1$ (two tasks with the same period can be represented by a single task with the

same period and a computation time equals to the sum of the original computation times). In this situation we must have: $C_i = T_{i+1} - T_i$ ($i < n$) and $C_n = T_n - 2(C_1 + \dots + C_{n-1})$, consider first the case of C_1 :

1. If $C_1 = T_2 - T_1 + \Delta$ ($\Delta > 0$), the task set τ'_1, \dots, τ'_n with $D'_i = T'_i = T_i \forall i$ and $C'_1 = T_2 - T_1$, $C'_2 = C_2 + \Delta$, $C'_3 = C_3, \dots, C'_n = C_n$ also fully utilizes the processor (see Figure 2.5, left; let us recall that the feasibility of the system is determined by the feasibility of each first request), with $U - U' = \frac{\Delta}{T_1} - \frac{\Delta}{T_2} > 0$ (contradicting our hypothesis).
2. If $C_1 = T_2 - T_1 - \Delta$ ($\Delta > 0$), the task set $\tau''_1, \dots, \tau''_n$ with $D''_i = T''_i = T_i \forall i$ and $C''_1 = C_1 + \Delta$, $C''_2 = C_2 - 2\Delta$, $C''_i = C_i$ for $i = 3, 4, \dots, n$, also fully utilizes the processor see Figure 2.5, right), with $U - U'' = -\frac{\Delta}{T_1} + \frac{2\Delta}{T_2} > 0$ (contradicting our hypothesis).

■



Figure 2.5: Schedule of task set $\{\tau_1, \tau_2\}$, if $C_1 = T_2 - T_1 + \Delta$ (left) and $C_1 = T_2 - T_1 - \Delta$ (right).

The reasoning of Liu and Layland is incorrect: the second part of their proof (the case where $C_1 = T_2 - T_1 - \Delta$ ($\Delta > 0$)) is incorrect for at least two reasons:

- The proof assumes that $C_2 > 2\Delta$,
- The task set $\tau''_1, \dots, \tau''_n$ does not necessarily fully utilize the CPU, as exhibited by the following example.

Example 2.19 Consider the task set $\{\tau_1 = \{T_1 = D_1 = 8, C_1 = 2\}, \{T_2 = D_2 = 11, C_2 = 3\}, \{T_3 = D_3 = 15, C_3 = 5\}$ which fully utilizes the cpu as exhibited in Figure 2.6, with $U = \frac{1130}{1320}$. If we choose $C''_1 = T_2 - T_1 = C_1 + \Delta = 3$, $C''_2 = C_2 - 2\Delta = 1$ and $C''_3 = C_3$, the set does not fully utilize the CPU as exhibited in Figure 2.7. And if we try to correct the situation by increasing C_3 , i.e., if we choose $C'''_3 = C_3 + 2\Delta$, the set fully utilizes the CPU but in this

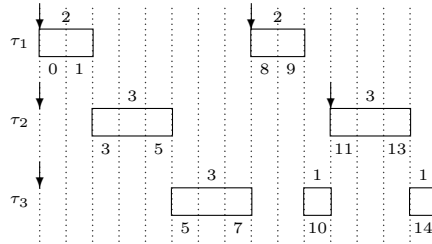


Figure 2.6: The task set fully utilizes the CPU.

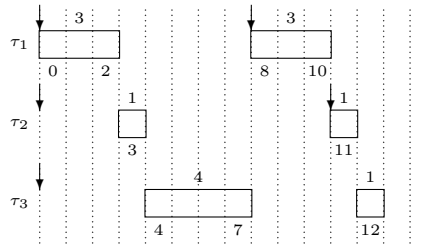


Figure 2.7: The task set does not fully utilize the CPU, the system is idle in the interval $[13, 15]$.

case $U'' = \frac{1231}{1320} > U$. This shows that the argument of Liu and Layland is wrong. ■

Remark however that Example 2.19 does not contradict the fact that we must have $C_i = T_{i+1} - T_i$ ($i < n$) and $C_n = T_n - 2(C_1 + \dots + C_{n-1})$, since in the previous example if we choose $C_1''' = 3, C_2''' = 4$ and $C_3''' = 1$, the set fully utilizes the CPU and $U''' = \frac{1063}{1320} < U$. Indeed, while their argument was wrong, Liu and Layland had again the correct intuition and Conjecture 2.18 is valid. We give here a complete proof of Conjecture 2.18:

Lemma 2.20 *For a set of n periodic synchronous tasks with late deadlines and the restriction that the ratio between any two task periods is less than 2, $b_n = n(\sqrt[n]{2} - 1)$, for the rate monotonic priority rule.*

Proof. Let τ_1, \dots, τ_n denote the n tasks. Let C_1, \dots, C_n be the execution times of the tasks that fully utilize the processor and minimize the processor utilization factor (here we shall allow real-valued C_i 's, considering they may be arbitrarily approximated by rational values, hence by integer values up to a multiplication of all task characteristics by an adequate integer coefficient). We assume that $T_n > T_{n-1} > \dots > T_1$ (two tasks with the same period can

be represented by a single task with the same period and a computation time equals to the sum of the original computation times). In this situation we must have: $C_i = T_{i+1} - T_i$ ($i < n$) and $C_n = T_n - 2(C_1 + \dots + C_{n-1})$. To show this, we shall first prove that $\forall i < n, C_i \leq T_{i+1} - T_i$. We shall proceed by contradiction: let us assume the property does not hold.

Consider first the case of C_1 and suppose that $C_1 = T_2 - T_1 + \Delta$ ($\Delta > 0$); notice that we must have that $T_2 < 2T_1$, otherwise $C_1 > T_1$ and the task set may not be schedulable; now, the task set τ'_1, \dots, τ'_n with $D'_i = T'_i = T_i \forall i$ and $C'_1 = T_2 - T_1, C'_2 = C_2 + \Delta, C'_3 = C_3, \dots, C'_n = C_n$ also fully utilizes the processor (see Figure 2.5, left), and $U - U' = \frac{\Delta}{T_1} - \frac{\Delta}{T_2} > 0$, since $T_2 > T_1$, contradicting our hypothesis; hence we must have $C_1 \leq T_2 - T_1$. We can apply the same argument for C_2, \dots, C_{n-1} .

Next, we may observe that if $\forall i < n : C_i \leq T_{i+1} - T_i$ and $T_n < 2T_1$, the task set fully utilizes the processor iff $C_n = T_n - 2\sum_{i=1}^{n-1} C_i$ (the first $n - 1$ tasks are schedulable and between 0 and T_1 , as well as between T_1 and T_n , they use $\sum_{i=1}^{n-1} C_i$ time units, with $\sum_{i=1}^{n-1} C_i \leq T_n - T_1 < T_1$).

Now, if $C_1 = T_2 - T_1 - \Delta$ ($\Delta > 0$), from the previous observation the task set $\tau''_1, \dots, \tau''_n$ with $D''_i = T''_i = T_i \forall i$ and $C''_1 = C'_1 + \Delta', C''_n = C'_n - 2\Delta', C''_i = C'_i$ for $i = 2, 4, \dots, n - 1$, also fully utilizes the processor, and $U' - U'' = -\frac{\Delta}{T_1} + \frac{2\Delta}{T_n} > 0$ since $2T_1 > T_n$, contradicting our hypothesis. We can apply the same argument for C_2, \dots, C_{n-1} and we get our property.

Let $g_i = \frac{T_n - T_i}{T_i}$ ($i = 1, \dots, n - 1$); we get

$$U = \sum_{i=1}^n \frac{C_i}{T_i} = 1 + g_1 \left(\frac{g_1 - 1}{g_1 + 1} \right) + \sum_{i=2}^{n-1} g_i \left(\frac{g_i - g_{i-1}}{g_i + 1} \right)$$

This expression must be minimal, hence $\frac{\partial U}{\partial g_j} = \frac{g_j^2 + 2g_j - g_{j-1}}{g_j + 1^2} - \frac{g_{j+1}}{g_{j+1} + 1} = 0$, for $j = 1, \dots, n - 1$.

The general solution can be shown to be $g_j = 2^{\frac{n-j}{n}} - 1$ ($j = 1, \dots, n - 1$).

It follows that $b_n = n(\sqrt[n]{2} - 1)$. ■

It may be noticed that $b_n < b_{n-1}$ and that $\lim_{n \rightarrow \infty} b_n = \ln 2$ (by the l'Hospital rule). Except for the trivial case $n = 1$, the bound b_n is never reached, since it is irrational, while from our assumption U is always rational.

The restriction that the ratio between task periods is less than 2 can now be relaxed.

Theorem 2.21 ([LL73]) *For a set of n periodic synchronous tasks with late deadlines, $b_n = n(\sqrt[n]{2} - 1)$, for the rate monotonic scheduler.*

Proof. Let $\tau_1, \tau_2, \dots, \tau_n$ be a set of n periodic tasks that fully utilize the processor. Let U denote the utilization factor of the task set. Suppose that for some i , $\left\lfloor \frac{T_n}{T_i} \right\rfloor > 1$. To be specific, let $T_n = q \cdot T_i + r$, $q > 1$ and $r \geq 0$. Let us replace the task τ_i by a task τ'_i such that $T'_i = q \cdot T_i$ and $C'_i = C_i$, and increase C_n by the amount needed to again fully utilize the processor. This increase is at most $C_i(q - 1)$, the time within the execution of τ_n occupied by τ_i but not by τ'_i (it may be less than C_i if some slots left by τ'_i are used by some τ_j with $i < j < n$). Let U' denote the utilization factor of such a set of tasks. We have

$$U' \leq U - \frac{C_i}{T_i} + \frac{C_i}{T'_i} + [(q - 1) \frac{C_i}{T_n}]$$

or

$$U' \leq U + C_i(q - 1) \left[\frac{1}{q \cdot T_i + r} - \frac{1}{q \cdot T_i} \right].$$

Since $q - 1 > 0$ and $\frac{1}{q \cdot T_i + r} - \frac{1}{q \cdot T_i} \leq 0$, $U' \leq U$. Therefore we conclude that in determining the least upper bound of the processor utilization factor, we need only consider task sets in which the ratio between any two task periods is less than 2. The Theorem follows from Lemma 2.20. ■

Liu and Layland [LL73] have formulated a sufficient condition for the schedulability of a task set based on the bound b_n , without a proof. The authors, from Theorem 2.21, stated that if a set of n tasks has an utilization factor less than the upper bound b_n it follows that the set is schedulable. This property does not immediately follow from Theorem 2.21, however, since the least upper bound b_n concerns the utilization factor of schedulable task sets only. There is no a priori reason to think that there are no unschedulable task set with utilization factor less than b_n . This problem is not obvious at all. We have investigated a lot of arguments before being able to show that, indeed, there are no unschedulable sets of n tasks with a utilization factor less than b_n . Our proof is based on the fact that b_n is strictly decreasing in n .

Theorem 2.22 *Let τ_1, \dots, τ_n be a task set. If $U = \sum_{i=1}^n \frac{C_i}{T_i} < b_n$, then the task set is schedulable.*

Proof. By induction on n . The property is trivially true for $n = 1$: τ_1 is schedulable iff $\frac{C_1}{T_1} \leq 1 = b_1$. Let us assume that the property is true up to $n - 1$, and consider a set of n tasks τ_1, \dots, τ_n with $U_n = \sum_{i=1}^n \frac{C_i}{T_i} < b_n$.

Since $b_n < b_{n-1}$, we have that $U_n < b_{n-1}$. Consider the $n - 1$ highest priority tasks $\tau_1, \dots, \tau_{n-1}$ with $U_{n-1} = \sum_{i=1}^{n-1} \frac{C_i}{T_i} = U_n - \frac{C_n}{T_n} < U_n$, hence $U_{n-1} < b_{n-1}$ and by induction hypothesis the task sub-set $\tau_1, \dots, \tau_{n-1}$ is schedulable. Suppose now that τ_1, \dots, τ_n is not schedulable, in this case we have, since the first $n - 1$ tasks are schedulable: $\exists x : 0 \leq x < C_n$: with $C'_1 = C_1, C'_2 = C_2, \dots, C'_{n-1} = C_{n-1}$, and $C'_n = x$ so that the task set is schedulable, while with $C'_1 = C_1, C'_2 = C_2, \dots, C'_{n-1} = C_{n-1}$, and $C'_n = x + 1$ the task set is not schedulable; by Lemma 2.16 the task set with $C'_n = x$ fully utilizes the processor and $U'_n = \sum_{i=1}^{n-1} \frac{C'_i}{T_i} + \frac{x}{T_n} < U_n < b_n$. By definition of U_n and U'_n this leads to a contradiction and proves the theorem. ■

This theorem gives us a sufficient condition. The series b_n converges to $\ln 2$. Hence, we are always sure that any task set (for any n) with a utilization factor less than 0.69 is schedulable, since $0.69 < \ln 2 < b_n < b_{n-1} < \dots < b_1$.

2.3 The deadline monotonic scheduler

The deadline monotonic scheduler is an extension of the rate monotonic scheduler. It relaxes one of its precondition; namely it allows the deadline of a task to be less than the period ($D_i \leq T_i$). Hence, the deadline monotonic rule is defined for a larger sub-class of periodic task sets.

- The deadline monotonic scheduler is defined for synchronous systems, where all tasks are started at the same time ($O_1 = O_2 = \dots = O_n = 0$), and
- for general deadline systems, where the deadline of each task is less than or equal to the period ($D_i \leq T_i, i = 1, \dots, n$).

Leung and Whitehead [LW82] have defined the deadline monotonic priority assignment (also termed the *inverse-deadline* priority assignment): priorities assigned to tasks are inversely proportional to the deadline. It may be noticed that in the special case where $D_i = T_i$ ($1 \leq i \leq n$), the deadline monotonic assignment is equivalent to the rate monotonic priority assignment.

2.3.1 Optimality

Again we can distinguish between two kinds of optimality:

Definition 2.23 A static priority assignment rule is *strongly optimal* for synchronous and general deadline systems if, when a feasible static priority assignment exists for a synchronous and general deadline task set, any priority assignment given by the rule is also feasible for that task set, whatever the way in which the ambiguities are resolved. ■

Definition 2.24 A static priority assignment rule is *weakly optimal* for synchronous and general deadline systems if, when a feasible static priority assignment exists for a synchronous and general deadline task set, some priority assignment given by the rule is also feasible for that task set (for a particular way to resolve the ambiguities). ■

The deadline monotonic priority assignment is strongly optimal for synchronous systems with general deadlines.

The optimality proof of Leung and Whitehead [LW82] is not fully convincing however, for similar reasons than those invoked for the optimality of the rate monotonic scheduler; we complete here the proof, and we show the strong optimality.

Theorem 2.25 *The deadline monotonic priority assignment is strongly optimal for synchronous systems with general deadlines.*

Proof. We must prove that if a feasible static priority assignment exists for a task set with general deadlines, any deadline monotonic priority assignment is also feasible for that task set. Let $\tau_1, \tau_2, \dots, \tau_n$ be a synchronous and general deadline task set. Suppose there exists a feasible priority assignment ($\tau_1 > \tau_2 > \dots > \tau_n$). Let τ_i and τ_j be two tasks of adjacent priorities ($\tau_i > \tau_j$, $j = i + 1$) with $D_i \geq D_j$. Let us exchange the priorities of τ_i and τ_j : if the task set is still schedulable, since any deadline monotonic priority assignment can be obtained from any priority ordering by a sequence of such priority exchanges, we may deduce that any deadline monotonic priority assignment is also feasible. The priority exchange does not modify the schedulability of the tasks with a higher priority than τ_i (τ_k , $\forall k < i$). The task τ_j remains of course schedulable after the priority exchange, since it may use all the free slots left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}\}$ instead of only those left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}, \tau_i\}$. Assuming that the requests of τ_i remain schedulable, from Lemma 2.11 the scheduling of each task τ_k , for $k = i + 2, i + 3, \dots, n$ is not altered since the idle periods left by higher priority tasks are the same. Consequently we must only verify that τ_i also remains schedulable. From Conjecture 2.9 we can restrict this question to the first request of task τ_i . Let r_j be the response time of the first request of τ_j before the priority exchange: the feasibility implies $r_j \leq D_j$; it is not difficult to see

that during the interval $[0, r_j)$ the CPU (when left free by higher priority tasks) is assigned first to the (first) request of τ_i then to the (first) request of τ_j (the latter is not interrupted by subsequent requests of τ_i since $T_i \geq D_i \geq D_j \geq r_j$). Hence, after the priority exchange, the CPU allocation is exchanged between τ_i and τ_j , and it follows that τ_i ends its computation at time r_j and meets its deadline since $r_j \leq D_j \leq D_i$. ■

It may be noticed that the previous proof assumes the Conjecture 2.9, in the same way than the Conjecture 2.7 was used for the optimality of the rate monotonic scheduler. Again, we suggest to the reader to consider this property as a conjecture till the presentation of our proof (which will not rely on those optimality results).

2.3.2 Feasibility tests

From Conjecture 2.9 it follows again that we only have to check if all the tasks meet their first deadline.

Corollary 2.26 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic synchronous and general deadline task set. The set is feasible iff the first request of each task meets its deadline with the priority given by the deadline monotonic rule.*

Proof. Immediately follows from Theorem 2.25 and Conjecture 2.9. ■

We shall see in Chapter 3 formulas and algorithms for the computation of the response time of the first request of task τ_i in the synchronous case, hence the interest of Corollary 2.26.

It is not difficult to see that when deadlines are allowed to be less than the periods ($D_i \leq T_i$), a schedulability test cannot have the form $\sum_{j=1}^n \frac{C_j}{T_j} < k$, where k is a constant; the boundary k should at least be a function of the task deadlines. Such a sufficient condition based on the utilization factor for the deadline monotonic priority assignment would be relevant however, and remains open for future works.

2.4 Synchronous arbitrary deadline systems

We consider here a larger sub-class of periodic task sets:

- Synchronous systems, i.e., all tasks are started at the same time ($O_1 = O_2 = \dots = O_n = 0$), and

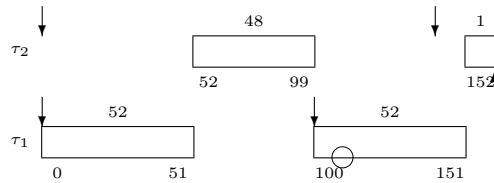


Figure 2.8: With the priority assignment $\tau_1 > \tau_2$, the system is unschedulable: the first request of τ_2 misses its deadline.

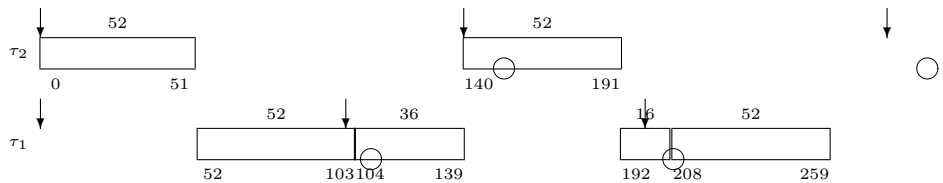


Figure 2.9: With the priority assignment $\tau_2 > \tau_1$, the system is schedulable (after the first idle time unit, at time 260, the situation is more favorable).

□ Arbitrary deadline: the deadline of each task τ_i is finite but may be less ($D_i \leq T_i$) or greater ($D_i > T_i$) than the period.

For this sub-class of periodic task sets the rate/deadline monotonic rule is no longer strongly optimal. This result follows from the following example given by Lehoczky [Leh90].

Example 2.27 Let $S = \{\tau_1 = \{C_1 = 52, T_1 = 100, D_1 = 110, O_1 = 0\}, \tau_2 = \{C_2 = 52, T_2 = 140, D_2 = 154, O_2 = 0\}\}$. Both rate/deadline priority assignments give the highest priority to task τ_1 . With this priority assignment, the set is unschedulable: the first request of τ_2 misses its deadline at time 154, as illustrated in Figure 2.8. With the priority assignment $\tau_2 > \tau_1$ the system is schedulable (see Figure 2.9), and fully utilizes the processor; we shall see that for this kind of system we may have to look further than the first request of each task to check feasibility, but we have only to consider the schedule until the first idle time unit. ■

Remark that this example shows more precisely that the deadline monotonic scheduler is not even weakly optimal for arbitrary deadline systems since all deadline monotonic priority assignments (here there is a single one) leave the system unschedulable.

Let us also notice that with this kind of systems, several requests of the same task can be active at the same time, while the system is schedulable. This is the case in our previous example: in the feasible schedule (i.e., the one given by the priority assignment $\tau_2 > \tau_1$, illustrated in Figure 2.9), at time $t = 100$ the second request of task τ_1 occurs while the first one is still active. Hence, in this situation several requests may have the same priority, and we did not define what is the scheduling rule nor what is our graphical conventions in this case. Hence, let us refine the scheduling rule: the system gives the CPU to the active request with the highest priority, the tie being broken by applying the first-in-first-out (FIFO) rule; in other words, the CPU is given in this case to the *oldest* active request with the highest priority, i.e., the request which arrived at the smallest time instant. In our example, the second request of τ_1 starts its execution after the end of the first request of τ_1 , at time $t = 104$. Our graphical conventions follow: all the requests of the same task are represented at the same level, and the completion of each request is represented with a vertical bar, as exhibited in Figure 2.9 at time $t = 104$, where the first request of τ_1 completes its execution and the second request of τ_1 starts its execution.

Although we consider here synchronous systems, the response time of the first request of τ_i is no longer always the largest one: in our feasible schedule (see Figure 2.9) the response times of the first requests of τ_1 are 104, 108 and 60, respectively. From a schedulability point of view, the second request of task τ_1 is in a worse situation since the first request of task τ_1 delays the execution of the second one. As a consequence, Conjecture 2.9 may not be extended to arbitrary deadline systems under our assumptions. Remark that Conjecture 2.9 could be extended by breaking the tie with the last-in-first-out (LIFO) rule, but this choice is not relevant regarding the feasibility of synchronous and arbitrary systems; for instance, the system introduced in the example 2.27 is certainly not schedulable with this new scheduling rule, even with the priority assignment: $\tau_2 > \tau_1$.

We have seen that, in some circumstances, it is possible to devise simple necessary and/or sufficient conditions for the schedulability of a given task set. For instance, for late deadline synchronous systems with the rate monotonic scheduling, Theorem 2.22 devises the rule that a task set is schedulable if (but not iff) $\sum_{i=1}^n \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$. In all generality, we also have that a task set is certainly not schedulable if $\sum_{i=1}^n \frac{C_i}{T_i} > 1$. If no such conditions are known, or if they do not give a definite answer for a given task set, of course it is not possible to simulate the evolution of the system till the end of ages in order to check if something goes wrong. However, it is generally possible to determine an interval such that if nothing goes wrong in it then nothing will ever go wrong.

Definition 2.28 For a given scheduling algorithm and a task set, a *feasibility interval* is a finite interval such that it is sure that no deadline will ever be missed iff, when we only keep the requests made in this interval, all deadlines for them in this interval are met. ■

For instance, we have:

Theorem 2.29 For a synchronous general deadline system with a static priority scheduler, $[0, D^{max}]$ is a feasibility interval, where $D^{max} = \max\{D_i | i = 1, \dots, n\}$.

Proof. Immediately follows from Conjecture 2.9. ■

For synchronous and arbitrary systems Lehoczky has exhibited such a feasibility interval, based on the fact that the largest response time for a request of τ_i occurs necessarily during the first *level- i busy period* (see Definition 2.34) of the synchronous system. His arguments were not fully developed however, and we shall here fill the gaps.

Up to now, we always assumed that either the considered system was feasible, or it stopped at its first deadline failure. However, once a static priority assignment has been chosen, together with a rule to break the ties if several requests are simultaneously active at the highest level, we may pursue the schedule after deadline failures (this amounts to considering them as soft), and get interesting properties and notions. It may be observed that even in the case of late or general deadline systems, after a deadline failure, we may have several active requests of a same task, hence we may need the FIFO rule to break the ties, like we did for arbitrary deadline systems. In the following, we shall consider such “extended” schedules.

Definition 2.30 $x \in \mathbb{N}$ is an idle processor point of the schedule of a system if all requests occurring strictly before x have completed their execution before or at time x . ■

This definition may be extended by considering that $t = \infty$ is also an idle point. Notice however that, if deadlines are missed, it may happen that some requests are not completed at $t = \infty$, i.e., their response time may be infinite and they are never completed, as in the example $\{\tau_1 = \{O_1 = 0, D_1 = C_1 = T_1 = 1\}, \tau_2 = \{O_2 = 0, D_2 = C_2 = T_2 = 2\}\}$ where no request of τ_2 will never receive the processor.

If the system is idle in an interval $[a, b)$, all (integer) instants between a and b (included) are idle points. More interesting, of course, are idle points where

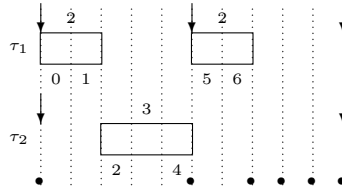


Figure 2.10: Various idle points.

either a request starts (0 is always an idle point of this kind), or gets completed, or both.

Example 2.31 Consider the following system $\{\tau_1 = \{O_1 = 0, C_1 = 2, T_1 = 5\}, \tau_2 = \{O_2 = 0, C_2 = 3, T_2 = 10\}\}$, as exhibited in Figure 2.10 (idle points are represented with the symbol \bullet): the system is idle in the interval $[7, 10)$ and 7, 8, 9, 10 are idle points; 10 is an idle point which coincides with two task requests; 7 is an idle point which coincides with the completion of the second request of τ_1 and 5 is an idle point which coincides with the completion of the first request of τ_2 and the second request of τ_1 . ■

Definition 2.32 An *elementary busy period* is a time interval $[a, b)$ such that a and b are idle points, the processor is busy in it and there is no intermediate idle point (i.e., there is no idle point at time c with $a < c < b$). ■

Lemma 2.33 *The set of time instants where there is an active request may be partitioned into elementary busy periods.*

Proof. Immediately follows from the Definition 2.32. ■

Definition 2.34 A *level- i busy period* is an elementary busy period in the schedule of the task sub-set $\{\tau_1, \dots, \tau_i\}$ where at least one request of τ_i occurs. ■

Lemma 2.35 *If $[a, b)$ is a level- i busy period and b is finite, b is the completion time of the last request of τ_i in this interval.*

Proof. From Definitions 2.34, 2.32 and 2.30, there is at least one request of τ_i in $[a, b)$, and since b is a finite idle point all these requests have been completed before or at time b ; if the last of these requests is completed at time $c < b$, c must be an idle point since all the other requests started before have a higher priority; but this contradicts the fact that $[a, b)$ is an elementary busy period. ■

Lemma 2.36 *In a busy level- i period, each request of τ_i but the first one starts strictly before the previous one is finished.*

Proof. If a request of τ_i in the interval finishes at time c and the next request starts at c or later in the interval, c is an idle point, which contradicts the fact that this interval is an elementary busy period. ■

Theorem 2.37 *The largest response time for a request of task τ_i in all asynchronous systems built from the same tasks occurs during the first level- i busy period $[0, \lambda_i)$ in the synchronous case, λ_i being the first idle point (after 0) in the synchronous schedule of the task subset $\{\tau_1, \dots, \tau_i\}$, and $[0, \lambda_i)$ is the largest level- i busy period.*

Proof inspired from [Leh90]. Let $[a, b)$ be a level- i busy period in some synchronous or asynchronous system built from $\{\tau_1, \dots, \tau_i\}$, and let us denote $a + \Delta_j$ the starting time of the first request of τ_j after a ($\Delta_j \geq 0$ and $1 \leq j \leq i$); from definitions above we have that $\Delta_i < b - a$ and $\Delta_k = 0$ for at least one k . Suppose first that $\Delta_i > 0$. Only tasks having a higher priority than τ_i are processed during $[a, a + \Delta_i)$; hence, if Δ_i were changed to any value in $[0, \Delta_i)$, each request of task τ_i in $[a, b)$ would finish at the same time as before (each request of τ_i occurs strictly before than in the original situation and uses the same free slots left by the task subset $\{\tau_1, \dots, \tau_{i-1}\}$ since there was no idle time unit left by τ_i), increasing each of their response time: the maximum response time occurs when $\Delta_i = 0$. Moreover, no idle point occurs before b since it is still true that each request of τ_i in the interval, but the first one, starts strictly before the previous one is completed (see Lemma 2.36); it could even happen that b is no longer an idle point, if the first request of τ_i which occurred after or at b now starts strictly before b : the level- i busy period is lengthened. If $\Delta_j > 0$ ($j < i$), then reducing Δ_j leads to increase (or leave unchanged) the processing requirement $r_j(t)$ of τ_j during $[0, t)$ for every $t \in (0, b]$, where $r_j(t) = k \cdot C_j$, k being the number of requests of τ_j occurring after or at a and strictly before t . Now it may be seen that the first request of τ_i after a finishes (if ever) at the first time instant t such that $(t - a) = C_i + \sum_{j=1}^{i-1} r_j(t)$, and more generally the k^{th} request of τ_i in the interval $[a, b)$ finishes at the first instant t such that $(t - a) = kC_i + \sum_{j=1}^{i-1} r_j(t)$: if $r_j(t)$ increases, this will delay accordingly each request of τ_i ; this may also enlarge the level- i busy period, and possibly blend it with the next one(s). Hence, the largest response times, and the largest level- i busy period, are achieved by setting each Δ_j to its smallest value: $\Delta_1 = \Delta_2 = \dots = \Delta_i = 0$. This configuration corresponds to the first level- i busy period in the synchronous case. ■

The proof above uses a rather qualitative argument on the response time of the requests of τ_i ; we shall make it fully quantitative in Chapter 3 (section 3.8.1).

Lemma 2.38 *If a level- i busy period $[a, b)$ is longer than $P_i = \text{lcm}\{T_1, \dots, T_i\}$, then $\sum_{j=1}^i \frac{C_j}{T_j} > 1$. Moreover, in the synchronous case, $a = 0$ and $b = \infty$.*

Proof. If a level- i busy period $[a, b)$ is such that $b - a > P_i$, since from Theorem 2.37 the first level- i busy period in the synchronous case is the longest, we have $\lambda_i > P_i$, i.e., there is no idle point till P_i ; $\sum_{j=1}^i \frac{P_i}{T_j} C_j > P_i$ since the demand occurring in $[0, P_i)$ cannot be satisfied, hence $\sum_{j=1}^i \frac{C_j}{T_j} > 1$, $\lambda_i = \infty$ since otherwise $\lambda_i = \sum_{j=1}^i \left\lceil \frac{\lambda_i}{T_j} \right\rceil C_j \geq \sum_{j=1}^i \frac{\lambda_i C_j}{T_j}$ so that $\sum_{j=1}^i \frac{C_j}{T_j} \leq 1$, a contradiction. As a consequence, $[0, \infty)$ is the only level- i busy period in the synchronous case. ■

Theorem 2.39 *For a synchronous arbitrary deadline system with a static priority scheduler, $[0, \lambda_n)$ is a feasibility interval.*

Proof. The property follows from the fact that $\lambda_1 < \lambda_2 < \dots < \lambda_n$ and the fact that if the largest response time of a request of τ_i is less than or equal to its deadline then all requests of task τ_i meet their deadline. ■

λ_n is the smallest positive solution ($\lambda_n > 0$) to the equation:

$$\lambda_n = \sum_{i=1}^n \left\lceil \frac{\lambda_n}{T_i} \right\rceil C_i, \quad (2.1)$$

$\left\lceil \frac{\lambda_n}{T_i} \right\rceil$ is the number of the requests of τ_i which occur strictly before time λ_n and can be computed by the iteration:

$$\begin{aligned} w_0 &= \sum_{i=1}^n C_i, \\ w_{k+1} &= \sum_{i=1}^n \left\lceil \frac{w_k}{T_i} \right\rceil C_i. \end{aligned}$$

The computation stops when two consecutive values are found equal or w_k exceeds P . We shall show that the iteration converges to the minimal solution (if any).

Theorem 2.40 *If $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, the iteration converges to the minimal solution.*

Proof. If $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, then from Lemma 2.38 $\lambda_n \leq P$.

The successive approximations w_k of λ_n are monotonically increasing. Indeed, by induction, we have that:

$$w_k \geq w_{k-1} \Rightarrow w_{k+1} \geq w_k;$$

since, if $w_k \geq w_{k-1}$ then $w_{k+1} = \sum_{i=1}^n \left\lceil \frac{w_k}{T_i} \right\rceil C_i \geq \sum_{i=1}^n \left\lceil \frac{w_{k-1}}{T_i} \right\rceil C_i = w_k$;

and $w_0 = \sum_{i=1}^n C_i \geq n$, so that $w_1 = \sum_{i=1}^n \left\lceil \frac{w_0}{T_i} \right\rceil C_i \geq w_0$.

Moreover, the iteration converges to the minimal solution. We shall show by induction that:

$$w_0 < w_1 < \dots \leq w_k \Rightarrow \lambda_n \geq w_k.$$

By induction on k . The property is true initially: $w_0 = \sum_{i=1}^n C_i$ while $\lambda_n \geq \sum_{i=1}^n C_i$. Suppose that the property is true up to k and we have $w_0 < w_1 < \dots < w_k \leq w_{k+1}$; by induction hypothesis $\lambda_n \geq w_k$. As a consequence, $[0, \lambda_n)$ contains at least the computation of all requests which occur in $[0, w_k)$: $\lambda_n \geq \sum_{i=1}^n \left\lceil \frac{w_k}{T_i} \right\rceil C_i = w_{k+1}$. ■

If $U \leq 1$ there is necessarily a solution less than or equal to P ; consequently if w_k exceeds P the iteration may stop since $U > 1$, the schedule is not feasible and $\lambda_n = \infty$ from Lemma 2.38.

The length of the feasibility interval (i.e., λ_n) depends on the utilization factor, and in the worst feasible case $\lambda_n = P$. Hence, the situation is less attractive than the one considered for the general deadline case, since P may grow exponentially with the number n of tasks.

Lehoczky has defined a feasibility criterion which consists in verifying the task requests in the interval $[0, \lambda_n)$; we shall not give details here. For convenience and uniformity of this work, we shall present another criterion based on our general response time computation in Chapter 3.

2.5 Asynchronous general deadline systems

Up to now, we have only considered synchronous systems. We shall now consider a more general class of periodic task sets:

- Asynchronous systems: the offsets are fixed by the constraints of the system and may be different (the tasks are not necessarily started at the same time).

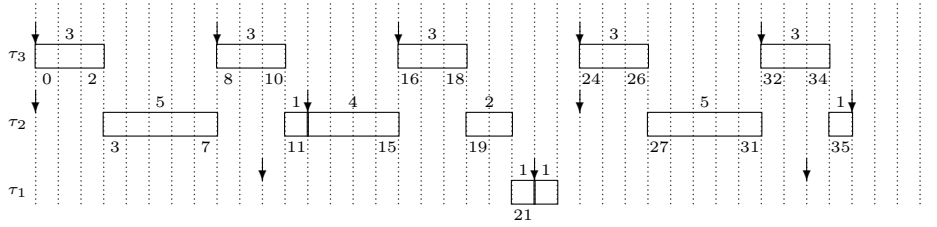


Figure 2.11: The task set is schedulable with $\tau_3 > \tau_2 > \tau_1$: at $t = 24$ the situation is the same as at $t = 0$ and the schedule repeats.

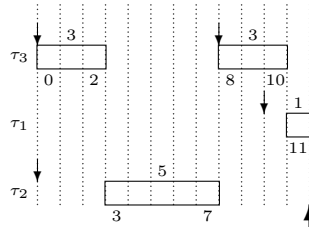


Figure 2.12: The task set is not schedulable with $\tau_3 > \tau_1 > \tau_2$: the first request of τ_2 fails.

- general deadline systems, where the deadline of each task is less than or equal to the period ($D_i \leq T_i$, $i = 1, \dots, n$).

For this more general class of periodic task sets, neither the rate monotonic nor the deadline monotonic schedulers are optimal priority assignments. Leung and Whitehead have shown that the rate monotonic scheduler is not (strongly) optimal for late deadline and asynchronous systems.

Lemma 2.41 ([LW82]) *The rate monotonic priority assignment is not strongly optimal for late asynchronous systems.*

Proof. This can be seen with the following system (introduced by Leung and Whitehead):

$\tau_1 = \{C_1 = 1, T_1 = D_1 = 12, O_1 = 10\}$, $\tau_2 = \{C_2 = 6, T_2 = D_2 = 12, O_2 = 0\}$, $\tau_3 = \{C_3 = 3, T_3 = D_3 = 8, O_3 = 0\}$. This system can be scheduled with the priority assignment ($\tau_3 > \tau_2 > \tau_1$) (see Figure 2.11), while the rate monotonic priority assignment ($\tau_3 > \tau_1 > \tau_2$) is not feasible (see Figure 2.12). ■

Corollary 2.42 *The deadline monotonic priority assignment is not strongly optimal for asynchronous systems.*

Proof. This results immediately from the previous example since, when $D_i = T_i \quad \forall i$, deadline monotonicity coincides with rate monotonicity. ■

This example, largely used in the literature [LW82, Aud91] to illustrate the non-optimality of the monotonic priority assignments for asynchronous systems raises one point. Both priority assignments $(\tau_3 > \tau_2 > \tau_1)$ and $(\tau_3 > \tau_1 > \tau_2)$ are in fact rate monotonic priority assignments. In this case, the non-optimality of the rate monotonic priority assignment is due to the choice made to resolve the tie between τ_1 and τ_2 . Hence, the example shows the non-strong optimality of monotonic priority assignments; but nothing can be inferred at that point for the weak optimality. We shall see in Chapter 5 (section 5.4.3) that the rate/deadline monotonic priority assignment is not even weakly optimal, by considering non-ambiguous situations where all periods are distinct. Hence the rate/deadline monotonic priority assignments are definitively non optimal for asynchronous systems.

Since the monotonic priority assignments are not optimal for asynchronous systems (late and general deadline situations), we shall assume here to have a static priority assignment $(\tau_1 > \tau_2 > \dots > \tau_n)$ but not necessarily the priority assignment given by the rate/deadline priority rule.

We come back to the feasibility problem of asynchronous systems. We shall present a feasibility interval for this kind of system. We may first observe that, in any case, the schedule produced by a static priority rule is periodic. Let us consider first some preliminary definitions.

Definition 2.43 We define $\epsilon_i(t)$ as the amount of processor time used by the last request of τ_i in the interval $[0, t)$, if $t \geq O_i$ and no deadline was missed before or at t . $\epsilon_i(t) = 0$ if $t < O_i$ and no deadline was missed before or at t . $\epsilon_i(t)$ is undefined otherwise, i.e., if a deadline was missed before. ■

Definition 2.44 We define $\gamma_i(t)$ as the time elapsed since the last request of τ_i , if $t \geq O_i$ and no deadline was missed before. $\gamma_i(t) = t - O_i$ if $t < O_i$ and no deadline was missed before or at t . $\gamma_i(t)$ is undefined otherwise, i.e., if a deadline was missed before or at t . ■

Definition 2.45 We define the configuration of the schedule S at time t for the system R as $C_S(R, t) = ((\gamma_1(t), \epsilon_1(t)), (\gamma_2(t), \epsilon_2(t)), \dots, (\gamma_n(t), \epsilon_n(t)))$. ■

Lemma 2.46 *The configuration of the schedule S at time $t + 1$ is univocally determined by the configuration at time $t \geq 0$.*

Proof. Consider first the case of the quantities $\gamma_i(t+1)$:
if $\gamma_i(t)$ is undefined, then $\gamma_i(t+1)$ is undefined since a deadline was missed before or at t (notice that at time $t=0$, all $\gamma_i(t)$'s are defined).

Else-if $\exists j : \gamma_j(t)+1 = D_j$ (i.e., $t+1$ coincides with the next deadline of τ_j) and $(\epsilon_j(t) < C_j - 1$ or $(\epsilon_j(t) = C_j - 1$ and $(\exists k < j : \epsilon_k(t) < C_k$ with $\gamma_k(t) \geq 0))$), i.e., the deadline of τ_j cannot be met, either because the last time unit is not enough to satisfy τ_j 's request or because the last time unit is used by someone else, with a higher priority, then $\gamma_i(t+1)$ is undefined. It may be noticed that this condition does not depend on i ; consequently if the condition is satisfied all $\gamma_i(t+1)$'s are undefined.

Else-if $\gamma_i(t) < 0$, then $\gamma_i(t+1) = \gamma_i(t) + 1$
otherwise $\gamma_i(t+1) = (\gamma_i(t) + 1) \bmod T_i$.

Consider now the quantities $\epsilon_i(t+1)$:

If $\gamma_i(t+1)$ is undefined then $\epsilon_i(t+1)$ is also undefined since a deadline was missed before (notice that at time $t=0$, all $\epsilon_i(t)$'s are defined).

Else-if $\gamma_i(t+1) = 0$, then $\epsilon_i(t+1) = 0$ (i.e., a new request arrives).

Else-if $\epsilon_i(t) < C_i$ and $\gamma_i(t) \geq 0$ (i.e., the last request of τ_i is active) and $\nexists j < i$ such that $\epsilon_j(t) < C_j$ with $\gamma_j(t) \geq 0$ (i.e., there is no higher priority active task than τ_i), $\epsilon_i(t+1) = \epsilon_i(t) + 1$ (i.e., the request of τ_i runs at time t).

Otherwise $\epsilon_i(t+1) = \epsilon_i(t)$ (if τ_i is not executing at time t , $\epsilon_i(t+1)$ is unchanged). ■

Theorem 2.47 *Any feasible schedule of an asynchronous general deadline system is finally periodic, i.e., periodic from some point.*

Proof. For any natural instant time t , we consider the configuration of a feasible schedule $C_S(R, t)$. Since the configuration of the schedule S at time $t+1$ is univocally determined by the configuration at time $t \geq 0$ (from Lemma 2.46) and $\forall i \in [1, n] : 0 \leq \epsilon_i(t) \leq C_i$, $-O_i \leq \gamma_i(t) < T_i$ and $\epsilon_i(t), \gamma_i(t)$ are integer numbers (unless $\epsilon_i(t)$ or $\gamma_i(t)$ is undefined, but then they are all undefined and the schedule is unfeasible), there are finitely many possible configurations and we may find two instants t_1, t_2 ($t_1 < t_2$) with the same configuration. Hence, from t_1 , the schedule will repeat periodically (with a period dividing $t_2 - t_1$). ■

Notice that if the schedule is unfeasible, it is also periodic in some sense, since from some point the configuration is constant (all components are undefined).

It may be noticed that this proof is valid not only for any static priority schedule, but more generally for any schedule where the priorities only depend

on the configuration $C_S(R, t)$. For instance, this is also true for the (dynamic) deadline driven scheduler (dynamic schedulers are studied in Chapter 4).

For static priority schedules, the Theorem 2.47 may even be refined:

Theorem 2.48 *For any asynchronous general deadline system ordered by decreasing priorities, let S_i be inductively defined by $S_1 = O_1$, $S_i = O_i + \lceil \frac{(S_{i-1} - O_i)^+}{T_i} \rceil T_i$ ($i = 2, 3, \dots, n$); then, if the schedule is feasible up to $S_n + P$, with $P = \text{lcm}\{T_i | i = 1, \dots, n\}$ and $x^+ = \max\{x, 0\}$, it is feasible and periodic from S_n with the period P .*

Proof. This immediately results by induction on n . The property is true in the trivial case where $n = 1$: the schedule for τ_1 is periodic of period T_1 from the first release of τ_1 ($S_1 = O_1$), if $C_1 \leq D_1$; otherwise the first deadline in $O_1 + D_1 \leq S_1 + T_1$ is missed. Let us now assume that the property is true up to $i - 1$ and the schedule of the first i tasks is feasible up to $S_i + P_i$, with $P_i = \text{lcm}\{T_j | j = 1, \dots, i\}$. S_i is the first release of task τ_i after (or at) S_{i-1} ; hence $S_i \geq S_{i-1}$, $P_i \geq P_{i-1}$, $S_i + P_i \geq S_{i-1} + P_{i-1}$ and by induction hypothesis, the schedule for the task subset $\{\tau_1, \dots, \tau_{i-1}\}$ is feasible and periodic from S_{i-1} of period P_{i-1} . Since the tasks are ordered by priority, the periodicity of the first ones is unchanged by the requests of task τ_i and the schedule repeats at time $S_i + \text{lcm}\{P_{i-1}, T_i\}$. Hence, for the task set $\{\tau_1, \dots, \tau_i\}$ the schedule is feasible and repeats from S_i with period P_i . ■

It may be noticed that, while P is indeed the true period of the periodic part of the schedule (it is also the period for the relative phasing of successive requests of the various tasks), it is not said that S_n is the earliest point from which the periodicity occurs: even for the first level, i.e., task τ_1 alone, we could have started from $S'_1 = O_1 - (T_1 - C_1)$, since the idle phase from S'_1 to S_1 corresponds to the one from $S_1 + C_1$ to $S_1 + T_1$; and similarly, if the schedule for τ_2 leaves an idle period of length δ'_2 before the instant $S_2 + \text{lcm}\{T_1, T_2\}$, we could replace S_2 by $S'_2 = \max\{S_1, S_2 - \delta'_2\}$, etc.

For synchronous systems Theorem 2.48 can be simplified as follows.

Corollary 2.49 *For any feasible schedule of a synchronous general deadline system, the schedule is periodic from time $t = 0$ with period $P = \text{lcm}\{T_i | i = 1, \dots, n\}$.*

Proof. Immediately follows from Theorem 2.48 and the fact that we consider synchronous systems, where $O_1 = O_2 = \dots = O_n = 0$ so that $S_1 = S_2 = \dots = S_n = 0$. ■

From a schedulability point of view, the first part of a schedule constructed by a given priority assignment may be neglected: we may only consider the schedule from its periodic part (in other words, in “steady state” situation). Indeed, from a schedulability point of view, the steady state situation is worse than the initial situation.

Lemma 2.50 ([LW82]) *Consider an asynchronous system with general deadlines. If the schedule is feasible, we have that $\forall i \in \{1, \dots, n\}$, at any instant $t \geq O_i$, $\forall k \in \mathbb{N} : t - k \cdot P \geq O_i \implies \epsilon_i(t - k \cdot P) \geq \epsilon_i(t)$.*

Proof. We prove the lemma by contradiction: suppose there is some task τ_j and instant $t_1 \geq O_j$ such that $\epsilon_j(t_1) < \epsilon_j(t_2)$ where $t_2 = t_1 + k \cdot P$ (notice that $t_1 \not\equiv O_j \pmod{T_j}$, since otherwise $\epsilon_j(t_1) = 0 = \epsilon_j(t_2)$). In this case, there must exist some time $t'_1 < t_1$ such that τ_j is active at both t'_1 and $t'_2 = t'_1 + k \cdot P$ (i.e., $\epsilon_j(t'_1) < C_j$ and $\epsilon_j(t'_2) < C_j$), and τ_j is executing at t'_2 but not at t'_1 . This can only occur if there is another task τ_i ($\tau_i > \tau_j$) which is active at t'_1 but not at t'_2 (i.e. $\epsilon_i(t'_1) < C_i$ and $\epsilon_i(t'_2) = C_i$). But this means that $\epsilon_i(t'_1) < \epsilon_i(t'_2)$ and repeating the above argument, we have: $\epsilon_{i_1}(t'_1) < \epsilon_{i_1}(t'_2), \epsilon_{i_2}(t'_1) < \epsilon_{i_2}(t'_2), \dots$ ($\tau_{i_1} < \tau_{i_2} < \dots$), contradicting the fact that we have a finite number of tasks in the system. ■

As a consequence, the response time of requests during the periodic part of the schedule is worse than the corresponding ones in the initial phase, and if no deadline is missed during a period, the same is true in general. As a result of Theorem 2.48, we may limit a simulation to the interval $[0, S_n + P]$. In other words, $[0, S_n + P]$ is a feasibility interval. But the lower bound of this interval can be improved. First, we may show that the periodic part of the schedule does not depend on the totality of the first part, which leads the system to its periodic behavior. In other words, we have to show that the periodic part remains unchanged if we add or drop several requests of τ_i .

Definition 2.51 We define the *partial schedule* σ_t of a system as the schedule obtained by only considering, in the schedule of the system, instants greater than t . ■

Lemma 2.52 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be a feasible asynchronous general deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$, and t be a time instant. Let X_j inductively defined by $X_{n+1} = t$, $X_i = O_i + \left\lfloor \frac{X_{i+1} - O_i}{T_i} \right\rfloor T_i$, for $i = n, n-1, \dots, 1$. If $X_{i+1} \geq O_i$ ($1 \leq i \leq n$), then the partial schedule σ_t only depends on the requests of τ_j ($1 \leq j \leq n$) occurring after (or at) the time X_j .*

Proof. We shall prove the property by (descending) induction on j . The property is true in the initial case, where $j = n$: in the schedule σ_t we have only to consider the requests of τ_n from time $X_n = O_n + \left\lfloor \frac{t - O_n}{T_n} \right\rfloor T_n$ (the time of the last request of τ_n which occurs before or at t ; this request exists since $t = X_{n+1} \geq O_n$). Indeed, the requests of τ_n which occur strictly before time X_n are terminated at time X_n (the schedule is feasible) and do not have any impact on σ_t (in particular, they have no impact on higher priority tasks). Let us assume now that the property is true for the tasks $\tau_n, \tau_{n-1}, \dots, \tau_{j+1}$ and let us consider the requests of task τ_j . The requests which occur before time $X_j = O_j + \left\lfloor \frac{X_{j+1} - O_j}{T_j} \right\rfloor T_j$ (the time of the last request of τ_j which occurs before or at X_{j+1} ; this request exists since $X_{j+1} \geq O_j$) are terminated before X_j (the schedule is feasible) and do not impact on the requests of τ_k after time X_k ($k = j + 1, \dots, n$), since $X_j \leq X_{j+1} \leq \dots \leq X_{n+1}$, nor of course on the schedule of higher priority tasks. The property that $X_j \leq X_{j+1}$ follows from the definition of X_j , the properties of the function $\lfloor \cdot \rfloor$ and from $X_{j+1} \geq O_j$. Hence, dropping some or all of those useless requests (from σ_t 's point of view) may not render the schedule unfeasible, and the schedule σ_t only depends on the requests of τ_j which occur after (or at) time X_j . ■

Corollary 2.53 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be a feasible asynchronous general deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$, and t be a time instant. Let X_j be inductively defined by $X_{n+1} = t$, $X_i = O_i + \left\lfloor \frac{X_{i+1} - O_i}{T_i} \right\rfloor T_i$, for $i = n, n - 1, \dots, 1$. If $t \geq O_{max} + \sum_{j=2}^n (T_j - 1)$, then the partial schedule σ_t , only depends on the requests of τ_j ($1 \leq j \leq n$) from time X_j .*

Proof. The following property holds:
 $t \geq O_{max} + \sum_{k=2}^n (T_k - 1) \Rightarrow X_{j+1} \geq O_{max} + \sum_{k=2}^j (T_k - 1) \geq O_j$. Let us proceed by descending induction on j . The property is true if $j = n$: $X_{n+1} = t \geq O_{max} + \sum_{k=2}^n (T_k - 1) \geq O_n$. Assume that the property is true down to $j+1$: $X_{j+1} \geq O_{max} + \sum_{k=2}^j (T_k - 1) \geq O_j$. The property that $0 \leq (X_{j+1} - X_j) < T_j$ follows from the definition of X_j , the properties of the function $\lfloor \cdot \rfloor$ and from $X_j \geq O_j$. Hence $X_j \geq O_{max} + \sum_{k=2}^{j-1} (T_k - 1) \geq O_{max} \geq O_{j-1}$ and the corollary follows from Lemma 2.52. ■

Corollary 2.54 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be a feasible asynchronous general deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$, and t be a time instant. Let X_j be inductively defined by $X_{n+1} = t$, $X_i = O_i + \left\lfloor \frac{X_{i+1} - O_i}{T_i} \right\rfloor T_i$, for $i = n, n - 1, \dots, 1$. If $X_{i+1} \geq O_i$ ($1 \leq i \leq n$) then*

$\forall t' \geq t$ the partial schedule $\sigma_{t'}$, only depends on the requests of τ_j ($1 \leq j \leq n$) from time X'_j , inductively defined by $X'_{n+1} = t'$, $X'_i = O_i + \left\lfloor \frac{X'_{i+1} - O_i}{T_i} \right\rfloor T_i$, for $i = n, n-1, \dots, 1$.

Proof. The following property holds: $X'_j \geq X_j$, by descending induction. The property is true in the trivial case: $t' = X'_{n+1} \geq X_{n+1} = t$. Assume that the property is true until $j+1$: $X'_{j+1} \geq X_{j+1}$. It is easy to see that the last request of τ_j which occurs before or at time X_{j+1} occurs before or at the last request of τ_j before or at X'_{j+1} , so that $X_j \leq X'_j$. Therefore $X'_{j+1} \geq O_j$ ($1 \leq j \leq n$) and the corollary follows from Lemma 2.52. ■

Corollary 2.55 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be a feasible asynchronous general deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$, and t be a time instant. Let X_j be inductively defined by $X_{n+1} = t$, $X_i = O_i + \left\lfloor \frac{X_{i+1} - O_i}{T_i} \right\rfloor T_i$, for $i = n, n-1, \dots, 1$. Let S_i be inductively defined by $S_1 = O_1$, $S_i = \max\{O_i, O_i + \lceil \frac{S_{i-1} - O_i}{T_i} \rceil T_i\}$ ($i = 2, 3, \dots, n$). If $t \geq S_n$ then the partial schedule σ_t , only depends on the requests of τ_j ($1 \leq j \leq n$) from time X_j .*

Proof. The following property holds: $X_{j+1} \geq S_j \geq O_j$ ($1 \leq j \leq n$). By descending induction: the property is true in the initial case, where $j = n$: $t = X_{n+1} \geq S_n$ by hypothesis and $S_n \geq O_n$ according to the properties of the values S_i (see Theorem 2.48). Assume that the property is true until $j+1$: $X_{j+1} \geq S_j \geq O_j$. S_j is the first request of τ_j after or at S_{j-1} ; hence X_j , the last request of τ_j before or at X_{j+1} , is certainly after or at S_{j-1} , which is itself after or at O_{j-1} . Therefore, $X_{j+1} \geq O_j$ ($1 \leq j \leq n$) and the corollary follows from Lemma 2.52. ■

We shall now consider a property similar to Lemma 2.52, but the fact that the instants X_1, X_2, \dots, X_n (called Y_1, Y_2, \dots, Y_n here) are not necessarily growing.

Lemma 2.56 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be a feasible asynchronous general deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$, and t be a time instant. Let Y_j inductively defined by $Y_{n+1} = t$, $Y_i = O_i + \left\lfloor \frac{(\min_{k=i+1}^{n+1} Y_k - O_i)^+}{T_i} \right\rfloor T_i$, for $i = n, n-1, \dots, 1$ (where $x^+ = \max\{x, 0\}$). Then the partial schedule σ_t only depends on the requests of τ_j ($1 \leq j \leq n$) occurring after (or at) time Y_j .*

Proof. We shall prove the property by descending induction on j . The property is true in the initial case, where $j = n$: in the schedule σ_t we have

only to consider the requests of τ_n from time $Y_n = O_n + \left\lfloor \frac{t-O_n}{T_n} \right\rfloor T_n$ if $t \geq O_n$, otherwise we consider the requests from the first one ($Y_n = O_n$). Indeed, the requests of τ_n which occur strictly before time Y_n (if any) are terminated at time Y_n (the schedule is feasible) and do not have any impact on σ_t (in particular, they have no impact on higher priority tasks). Assume that the property is true for the tasks $\tau_n, \tau_{n-1}, \dots, \tau_{j+1}$ and let us consider the requests of task τ_j . We have to distinguish between two cases: (a) $O_j > \min_{k=j+1}^{n+1} Y_k$ or (b) $O_j \leq \min_{k=j+1}^{n+1} Y_k$.

- a) This is the trivial case: the schedule and consequently the partial schedule σ_t only depends on the request of τ_j from time $O_j = Y_j$.
- b) Let $p : Y_p = \min_{k=j+1}^{n+1} Y_k$, the requests of τ_j which occur strictly before (if any) time $Y_j = O_j + \left\lfloor \frac{\min_{k=j+1}^{n+1} Y_k - O_j}{T_j} \right\rfloor T_j$ (the time of the last request of τ_j which occurs before or at Y_p) are terminated before Y_j (the schedule is feasible), hence before Y_p , and do not impact on the requests of τ_k after time Y_k ($k = j+1, \dots, n$), since $Y_p \leq Y_k$ ($k = j+1, \dots, n$), nor of course on the schedule of higher priority tasks. Hence, dropping some or all of those useless requests (from σ_t 's point of view) may not render the schedule unfeasible, and the schedule σ_t only depends on the requests of τ_j which occur after (or at) time Y_j .

■

Theorem 2.57 *Let X_i be inductively defined by $X_n = S_n$, $X_i = O_i + \left\lfloor \frac{X_{i+1} - O_i}{T_i} \right\rfloor T_i$ ($i = n-1, n-2, \dots, 1$) and let S_i be inductively defined by $S_1 = O_1$, $S_i = \max\{O_i, O_i + \left\lceil \frac{S_{i-1} - O_i}{T_i} \right\rceil T_i\}$ ($i = 2, 3, n$); then $[X_1, S_n + P]$ is a feasibility interval. Moreover, for each τ_i one only has to check the deadlines in the interval $[S_i, S_i + \text{lcm}\{T_j | j \leq i\}]$.*

Proof. Lemma 2.50 means that the load before S_n is certainly less than or equal to the one during the period $[S_n, S_n + P]$; hence, if no deadline is missed between S_n and $S_n + P$, this will also be the case before S_n , since the requests are fulfilled earlier there. As a consequence, the load before S_n is only necessary to lead the system to its periodic behavior from S_n (or earlier). By Corollary 2.55 we have that the periodic behavior of the system (σ_{S_n}), only depends on the requests of τ_j from time X_j . Hence the first part of the property. The second part immediately follows from Theorem 2.48. ■

It is important to note that the various feasibility intervals given in this section in general present a major improvement in comparison with feasibility intervals

issued from the literature, and in particular with the interval $[O^{max}, O^{max} + 2P)$ given by Leung and Whitehead [LW82]; indeed our feasibility interval has a maximal length of $P + \sum_{i=1}^{n-1} (T_i - 1)$ and the second term is generally by far lower than the first one.

2.6 Feasibility interval for asynchronous arbitrary deadline systems

We consider here a larger sub-class of periodic task sets:

- Asynchronous systems: the offsets are fixed by the constraints of the system and may be different (the tasks are not necessarily started at the same time).
- Arbitrary deadline systems: the deadline of each task τ_i may be less ($D_i \leq T_i$) or greater ($D_i > T_i$) than the period.

For this sub-class of periodic task sets the rate/deadline monotonic scheduler is certainly not weakly optimal: Example 2.27 considered in section 2.4 shows also the non-optimality of the rate/deadline monotonic priority assignment in the present case, since synchronous systems are special cases of asynchronous systems.

We suppose to have a fixed priority assignment ($\tau_1 > \tau_2 > \dots > \tau_n$) and we study the feasibility problem; in particular we are concerned by feasibility intervals.

Lehoczky has only considered arbitrary deadline in the synchronous case; in particular he has indicated that the synchronous case is the worst case and he has studied feasibility intervals for it. It is interesting however to consider more general and optimistic cases than the worst case. We shall here extend the theory to handle arbitrary deadlines in asynchronous systems.

We define g_i as the maximal number of active requests of τ_i at the same time. It is not difficult to see that, if the system is feasible, $g_i \leq \lceil \frac{D_i}{T_i} \rceil$ (otherwise the oldest active request already missed its deadline). Let δ_i^k be the k^{th} request ($k = 1, 2, \dots$) of task τ_i , which occurs at time $R_i^k = O_i + (k - 1)T_i$. The number h_i of requests of task τ_i which occur during a hyper-period P (after O_i) is given by: $h_i = \frac{P}{T_i}$ ($\in \mathbb{N}$). We first extend the definition of the function $\epsilon_i(t)$ in this context (there may be many active requests of task τ_i at time t).

Definition 2.58 We define $\epsilon_i^k(t)$ as the amount of processor time used by the request δ_i^k in the interval $[R_i^k, t)$, if no deadline has been missed before, otherwise it is left undefined. In particular, $\epsilon_i^k(t) = 0$ if $t \leq R_i^k$ unless a deadline has been missed before. ■

Remark that, at any time t there is a maximum of g_i active requests of τ_i say: $\delta_i^k, \delta_i^{k+1}, \dots, \delta_i^{k+q_i}$ ($q_i < g_i$). Since the various requests of the same task are served in a FIFO basis, we have $\epsilon_i^k(t) < C_i$ and $\epsilon_i^{k+p}(t) = 0$ for $p = 1, \dots, q_i$. For this reason we shall only consider the value $\epsilon_i^k(t)$ in the extended configuration of the system in arbitrary deadline situation.

Definition 2.59 We define the configuration of the schedule at time t as

$$C_S(R, t) = ((\gamma_1(t), \alpha_1(t), \beta_1(t)), (\gamma_2(t), \alpha_2(t), \beta_2(t)), \dots, (\gamma_n(t), \alpha_n(t), \beta_n(t))).$$

where

- $\gamma_i(t)$ is the time elapsed since the last request of τ_i , if $t \geq O_i$ and no deadline was missed before. $\gamma_i(t) = t - O_i$ if $t < O_i$ and no deadline was missed before. (In particular, $\gamma_i(0) = -O_i$.)
- $\alpha_i(t)$ is the number of active requests of τ_i at time t , if no deadline was missed before. (In particular, $\alpha_i(0) = 1$ if $O_i = 0$, 0 otherwise.)
- $\beta_i(t)$ is the amount of processor time used at time t by the oldest active request of τ_i , if any, and if no deadline was missed before. If $\alpha_i(t) = 0$, $\beta_i(t) = 0$. (In particular, $\beta_i(0) = 0$.)

If a request was missed before time t , $\gamma_i(t), \beta_i(t)$ as well as $\alpha_i(t)$ are undefined. ■

With the definition of $C_s(R, t)$, Theorem 2.47 may be generalized as follows.

Theorem 2.60 *Any feasible schedule of an asynchronous arbitrary deadline system is finally periodic, i.e., periodic from some point.*

Proof. For any integer instant time t , we consider the configuration of the schedule

$$C_S(R, t) = ((\gamma_1(t), \alpha_1(t), \beta_1(t)), (\gamma_2(t), \alpha_2(t), \beta_2(t)), \dots, (\gamma_n(t), \alpha_n(t), \beta_n(t))).$$

For static priority preemptive schedulers, the configuration at time $t + 1$ is univocally determined by the configuration at time t . This can be shown by

considering the following algorithm which computes $C_S(R, t+1)$ from $C_S(R, t)$ and the task characteristics (i.e., O_i, D_i, C_i, T_i).

```

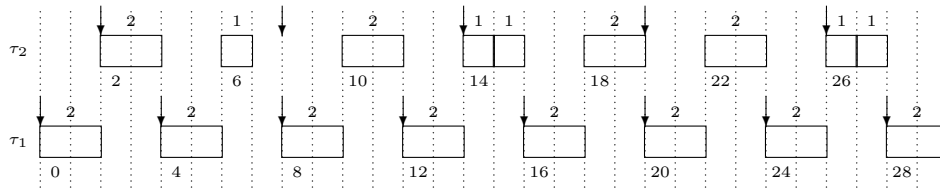
If  $\forall i \in [1, n]$   $\alpha_i(t), \gamma_i(t)$  as well as  $\beta_i(t)$  are undefined Then
     $\forall i \in [1, n]$   $\alpha_i(t+1), \gamma_i(t+1)$  as well as  $\beta_i(t+1)$  are undefined;
    {a deadline was missed before time  $t$ }
Else
     $\forall i \in [1, n]$  Do
        If  $\gamma_i(t) < 0$  Then  $\gamma_i(t+1) := \gamma_i(t) + 1$ ; {one progresses to  $O_i$ }
        Else  $\gamma_i(t+1) := (\gamma_i(t) + 1) \bmod T_i$ ;
            {from  $O_i, \gamma_i(t)$  progresses cyclically}
        EndIf
         $\alpha_i(t+1) := \alpha_i(t)$ ;
         $\beta_i(t+1) := \beta_i(t)$ ; {in general  $\alpha_i$  and  $\beta_i$  are unchanged}
        If  $(\gamma_i(t+1) = 0)$  Then {a new request occurs at time  $t+1$ }
             $\alpha_i(t+1) := \alpha_i(t) + 1$ ; {if  $\alpha_i(t) = 0, \beta_i(t) = \beta_i(t+1) = 0$  too}
        EndIf
    Od
     $j := 1$ ;
    While  $(j \leq n$  and  $\alpha_j(t) = 0)$  Do
         $j := j+1$ ;
    Od { $\tau_j$  is the highest priority active task at  $t$ , if any; otherwise,  $j = n+1$ }
    If  $(j \leq n)$  Then
         $\beta_j(t+1) := \beta_j(t) + 1$ ;
        {execution of the request of  $\tau_j$  during one time unit}
        If  $(\exists k : \alpha_k(t+1) > 0$  and  $(\alpha_k(t+1) - 1)T_k + \gamma_k(t+1) = D_k$  and
             $\beta_k(t+1) < C_k)$  Then { $\tau_k$  misses its deadline}
             $\forall i \in [1, n] : \alpha_i(t+1), \gamma_i(t+1)$  as well as  $\beta_i(t+1)$  are undefined;
        Else
            If  $(\beta_j(t+1) = C_j)$  Then
                 $\alpha_j(t+1) := \alpha_j(t) - 1$ ; {the request becomes inactive}
                 $\beta_j(t+1) := 0$ ;
            EndIf
        EndIf
    EndIf
EndIf

```

Since $\forall i \in [1, n], 0 \leq \alpha_i(t) \leq g_i, 0 \leq \beta_i(t) < C_i$ and $-O_i \leq \gamma_i(t) < T_i$, there are finitely many possible configurations and we may find two instants t_1, t_2 ($t_1 < t_2$) with the same configuration. From t_1 , the schedule will repeat periodically (with a period dividing $t_2 - t_1$). ■

| | T_i | D_i | C_i | O_i |
|----------|-------|-------|-------|-------|
| τ_1 | 4 | 4 | 2 | 0 |
| τ_2 | 6 | 7 | 3 | 2 |

Table 2.2: Characteristics of a periodic task set.

Figure 2.13: Response time of τ_2 , $t_2 - t_1 = 2$.

Again, if the schedule is unfeasible the system is also finally periodic, since from the first deadline failure the configuration is constantly undefined.

Theorem 2.48 (defining an instant S_n where the schedule repeats) does not remain valid for arbitrary deadlines, this can be seen with the following example.

Example 2.61 Consider the scheduling of the system given by Table 2.2, with the static priority assignment $\tau_1 > \tau_2$. We get $S_1 = 0$ and $S_2 = 2$, but the schedule (see Figure 2.13) is not periodic from time S_2 , since at time $t = S_2 + P = 14$, when the third request of τ_2 occurs, the schedule differs (in comparison with time S_2), due to the fact that at time $S_2 + P$ the second request of τ_2 is still active. ■

In this arbitrary deadline situation, the scheduling of a request of τ_i (say δ_i^k) may also depend on several preceding requests of task τ_i ($\delta_i^{k-1}, \delta_i^{k-2}, \dots$). Each preceding request of task τ_i (say δ_i^{k-j}) may also depend on several preceding requests of τ_i ($\delta_i^{k-j-1}, \delta_i^{k-j-2}, \dots$), and so on. For this reason, an extension of the Lemma 2.52 in this situation does not seem easy. We shall however define a feasibility test for the arbitrary deadline and asynchronous systems based on another approach.

First, we show that again, from a schedulability point of view, the first part of the schedule is more favorable.

Lemma 2.62 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be an asynchronous arbitrary deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$. For each task τ_i , for any instant $t \geq O_i$ and any k such that $R_i^k \leq t < R_i^k + D_i$, if*

the schedule is feasible up to time $t + P$ (or if the first deadline failure occurs exactly at $t + P$) we have $\epsilon_i^k(t) \geq \epsilon_i^{k+h_i}(t + P)$, with $h_i = \frac{P}{T_i}$.

Proof. We prove the lemma by contradiction and assume that some first instant (the time is discrete in our model of computation) t exists such that there is some j, k with $R_j^k \leq t \leq R_j^k + D_j$ and $\epsilon_j^k(t) < \epsilon_j^{k+h_j}(t + P)$. From the definition of $\epsilon_i^k(t)$, it follows that $\epsilon_i^k(t)$ is a non-decreasing discrete step function with $0 \leq \epsilon_i^k(t) \leq C_i$. The function increases at time instants where the corresponding request of τ_i is executing, otherwise the function is constant. Moreover, $\epsilon_i^k(R_i^k) = 0 = \epsilon_i^{k+h_i}(R_i^k + P)$. Then there must be some time $R_j^k \leq t' < t$ such that $\delta_j^{k+h_j}$ is executing at $t' + P$ while δ_j^k is not executing at time t' . This can only occur if there is a task (say task τ_i , $1 \leq i \leq j$) executing a higher priority request $\delta_i^{k'}$ (i.e., $i < j$ or ($i = j$ and $k' < k$)) at time t' while the request $\delta_i^{k'+h_i}$ is not executing at $t' + P$, nor any previous request of τ_i , nor any request of τ_h with $h < i$. But this means that $\epsilon_i^{k'}(t') < \epsilon_i^{k'+h_i}(t' + P) = C_i$ (otherwise $\delta_i^{k'+h_i}$ is active at time $t' + P$ and $\delta_j^{k+h_j}$ cannot execute at time $t' + P$), contradicting the fact that t is the first instant with this property. ■

Corollary 2.63 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be an asynchronous arbitrary deadline system with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$. For each task τ_i , for any instant $t \geq O_i$, if the schedule is feasible up to time $t + P$ (or if the first deadline failure occurs exactly at $t + P$) we have $(\alpha_i(t) < \alpha_i(t + P))$ or $[(\alpha_i(t) = \alpha_i(t + P))$ and $(\beta_i(t) \geq \beta_i(t + P))]$.*

Proof. If $\alpha_i(t) = 0$, then either $\alpha_i(t + P) > 0$ or $\alpha_i(t + P) = 0 = \beta_i(t + P) = \beta_i(t)$. Otherwise, $\alpha_i(t) = n_i(t) - m_i(t)$ where $n_i(t) = \#\{k | R_i^k \leq t\} = \max\{k | R_i^k \leq t\}$ is the number of requests of τ_i started till time t and $m_i(t) = \#\{k | \epsilon_i^k(t) = C_i\} = \max\{k | \epsilon_i^k(t) = C_i\}$ is the number of completed requests of τ_i till time t ; $n_i(t + P) = n_i(t) + h_i$ and from Lemma 2.62 it occurs that $m_i(t + P) \leq m_i(t) + h_i$, hence $\alpha_i(t + P) \geq \alpha_i(t)$; and if $\alpha_i(t) = \alpha_i(t + P)$ then $m_i(t + P) = m_i(t) + h_i$ and, $\beta_i(t) = \epsilon_i^{m_i(t)+1}(t) \geq \epsilon_i^{m_i(t)+1+h_i}(t + P) = \beta_i(t + P)$. ■

The periodic part of the schedule can now be made more explicit.

Lemma 2.64 *Let S be a feasible schedule of an asynchronous and arbitrary deadline periodic task set R for a fixed priority assignment and let $C_S(R, t)$ be the configuration of the schedule at time t ($t \geq O^{\max}$), then $C_S(R, t_1) = C_S(R, t_1 + P)$ with $t_1 = O^{\max} + P$.*

Proof. Assume $C_S(R, t_1) \neq C_S(R, t_1 + P)$. First remark that $\forall i, \gamma_i(t_1) = \gamma_i(t_1 + P)$. From Corollary 2.63 it follows that there is a number j such that

either $(\alpha_j(t_1) < \alpha_j(t_1 + P))$ or $[(\alpha_j(t_1) = \alpha_j(t_1 + P)) \text{ and } (\beta_j(t_1) > \beta_j(t_1 + P))]$. In both case there must be a natural k such that: $\epsilon_j^k(t_1) > \epsilon_j^{k+h_j}(t_1 + P)$ and $\epsilon_i^r(t) \geq \epsilon_i^{r+h_i}(t + P) \forall i = 1, \dots, n, \forall r : R_i^r \leq t < R_i^r + D_i$. We first show that the schedule has no idle time slots in $[t_1, t_1 + P)$ (i.e., the CPU remains busy in $[t_1, t_1 + P)$). Suppose there is some idle slot at time $t_1 + \delta, 0 \leq \delta < P$. This implies that no task request is active at that time; that is, for all $i \in \{1, \dots, n\}$, and for all k such that $R_i^k \leq t_1 + \delta \leq R_i^k + D_i$ we have: $\epsilon_i^k(t_1 + \delta) = C_i$. By Lemma 2.62, this implies $C_S(R, O^{max} + \delta) = C_S(R, t_1 + \delta)$. Since the task requests in the intervals $[O^{max} + \delta, t_1 + \delta)$ and $[t_1 + \delta, t_1 + P + \delta)$ are the same ($t_1 + \delta = O^{max} + \delta + P$) and, at time $O^{max} + \delta$, we have that for all $i \in \{1, \dots, n\}$ and for all $k : R_i^k \leq t_1 + \delta \leq R_i^k + D_i \Rightarrow \epsilon_i^k(O^{max} + \delta) = C_i$, i.e., all the previous requests are terminated, the schedules in these intervals are identical. This means that $C_S(R, t_1) = C_S(R, t_1 + P)$, a contradiction. Therefore, S has no idle time slot in $[t_1, t_1 + P)$. At time t_1 there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n \alpha_q(t_1)C_q - \beta_q(t_1)$; at time $t_1 + P$ the total remaining demand is larger: $\sum_{q=1}^n \alpha_q(t_1 + P)C_q - \beta_q(t_1 + P)$. But the additional demand in $[t_1, t_1 + P)$, i.e., the total demand which occurs in the interval $[t_1, t_1 + P)$, is equal to $P \cdot U$ and the CPU availability is P , it follows that the additional demand is larger than P which implies that $U > 1$, contradicting the feasibility of the schedule. ■

Lemma 2.65 *Let S be some schedule constructed with a fixed priority assignment applied to an asynchronous and arbitrary deadline periodic task set R . S is feasible iff (1) all deadlines in the interval $[0, O^{max} + 2P)$ are met in the schedule S , and (2) $C_S(R, O^{max} + P) = C_S(R, O^{max} + 2P)$.*

Proof.

(only if part). If S is feasible, all deadlines in the interval $[0, O^{max} + 2P)$ are met in S and, by Lemma 2.64, we have $C_S(R, O^{max} + P) = C_S(R, O^{max} + 2P)$.

(if part). Let $t_1 = O^{max} + P$; since $C_S(R, t_1) = C_S(R, t_1 + P)$, from the proof of Lemma 2.60 where we showed that $C_S(R, t + 1)$ is univocally determined by $C_S(R, t)$, the schedule S repeats every P units of time, starting from t_1 . Since all deadlines in the interval $[0, t_1 + P)$ are met in S (in particular, $C_S(R, t_1)$ is not undefined), the deadlines of all task computations must also be met in S . Hence the property. ■

Both conditions of Lemma 2.65 are clearly necessary for the feasibility of the system. The following example shows that condition (1) does not imply condition (2).

Example 2.66 Consider the following system:

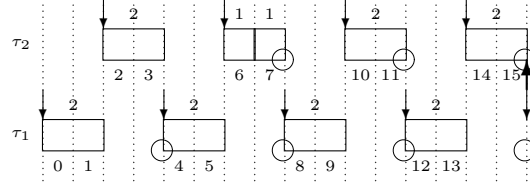


Figure 2.14: In arbitrary deadline situation the (first) deadline failure may occur at time t_2 .

| | T | D | C | O |
|----------|-----|-----|-----|-----|
| τ_1 | 4 | 4 | 2 | 0 |
| τ_2 | 4 | 6 | 3 | 2 |

We have $P = 4$, $O^{max} = 2$ and $t_1 = 6$; moreover, as illustrated in Figure 2.14, all deadlines in the interval $[0, 10)$ are met, but $C_S(R, 6) \neq C_S(R, 10)$, since $\beta_2(6) = \epsilon_2^1(6) = 2$ and $\beta_2(10) = \epsilon_2^2(10) = 1$. And τ_2 misses its deadline at time 16. This example also shows that the interval $[0, O^{max} + 2P)$ is not a feasibility interval in the arbitrary deadline case. ■

Instead of trying to find a longer feasibility interval (depending on the various D_i 's), we shall take advantage that we already know that a system may only be feasible if $U \leq 1$ (notice that this is not the case in the Example 2.66). In that case we have:

Theorem 2.67 *The interval $[0, O^{max} + 2P)$ is a feasibility interval for asynchronous arbitrary deadline periodic task sets for a fixed priority assignment if $U \leq 1$.*

Proof. The property is trivial if a deadline failure occurs in the interval $[0, O^{max} + 2P)$. Hence, we suppose that no deadline failure occurs in the interval $[0, O^{max} + 2P)$. Assume $C_S(R, t) \neq C_S(R, t + P)$, with $t = O^{max} + P$. As in the proof of Lemma 2.64, there must be a task j and a natural k : $\epsilon_j^k(t) > \epsilon_j^{k+h_j}(t + P)$ and $\epsilon_i^r(t) \geq \epsilon_i^{r+h_i}(t + P)$ for $1 \leq i \leq n$ and $R_i^r \leq t$ (this also holds if a deadline failure occurs at $t + P$). We first show that the schedule has no idle time slot in $[t, t + P)$ (i.e., the CPU remains busy in $[t, t + P)$). Suppose there is some idle time slot at time $t + \delta$, $0 \leq \delta < P$. This implies that no task request is active at that time; consequently $\forall i, k : t + \delta \geq R_i^k \Rightarrow \epsilon_i^k(t + \delta) = C_i$. By Lemma 2.62, this implies $\forall i, k : O^{max} + \delta \geq R_i^k \Rightarrow \epsilon_i^k(O^{max} + \delta) = C_i$ and $C_S(R, O^{max} + \delta) = C_S(R, t + \delta)$ and $C_S(R, t + \delta) = ((\gamma_1(t + \delta), 0, 0), \dots, (\gamma_n(t + \delta), 0, 0))$. From the proof of Lemma 2.60, which

shows that $C_S(R, t+1)$ is univocally determined by $C_S(R, t)$, it follows that in the intervals $[O^{max} + \delta, t + \delta)$ and $[t + \delta, t + P + \delta)$ the schedules are identical. This means that $C_S(R, t) = C_S(R, t+P)$, a contradiction. Therefore, S has no idle time slot in $[t, t+P)$. At time t there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n C_q - \epsilon_q^{r_q}(t_1)$; at time $t+P$, from Lemma 2.62 and the fact that $\epsilon_j^k(t) > \epsilon_j^{k+h_j}(t+P)$ the remaining demand: $\sum_{q=1}^n C_q - \epsilon_q^{r_q+h_q}(t+P)$ is larger. But the additional demand in the interval $[t, t+P)$ is equal to $P \cdot U$. In the interval $[t, t+P)$ the CPU availability is P , it follows that the additional demand is larger than P which implies that $U > 1$, a contradiction with the hypothesis. Hence $C_S(R, t) = C_S(R, t+P)$, and the property follows from Lemma 2.65. ■

2.7 Stability

We shall consider in this section the scheduling of a system composed of two periodic task sets (say S_1 and S_2), the task set S_1 , called the *critical task set*, is composed of a schedulable periodic task set with hard deadlines (as considered in the previous sections), the task set S_2 is composed of *non-critical* periodic tasks with soft deadlines. Moreover, we shall relax a constraint concerning the characteristics of the non-critical tasks (i.e., in S_2): the computation time of the requests of non-critical tasks may vary and is unbounded. For instance, the computation time may be stochastic and the given characteristic C_i only specifies the average execution time (or some percentile) of a request of τ_i in our computation model. Then, some requests of non-critical tasks (say τ_j) may have a larger execution time than C_j and cause a deadline failure. In this case, when a request of task τ_j misses its deadline, we assume that the scheduler continues the execution of the various requests according to the priority rule. In this kind of systems, we can require to preserve the schedulability the task set S_1 , in other words: have a *stable* scheduler.

Definition 2.68 A scheduler \mathcal{A} is said to be *stable* if all requests of tasks in the set S_1 meet their deadlines in the system composed by $S_1 \cup S_2$, even if the actual execution times of requests of tasks in S_2 (the non-critical requests) are unbounded. ■

Theorem 2.69 A static priority scheduler is stable iff the critical tasks are the highest priority tasks.

Proof. (*if part*). If the critical tasks are the highest priority tasks it is not difficult to see that whatever the execution times of the non-critical requests,

all critical requests meet their deadlines, since the schedulability of the highest priority tasks is not altered by the requests with lower priority.

(*only if part*). Suppose that the condition does not hold: $\exists k, r : \tau_k \in S_2, \tau_r \in S_1$ and $\tau_k > \tau_r$. Since the execution times of the requests of τ_k are unbounded and can be arbitrarily large, it may exist a time instant such that both tasks are active and such that the remaining processing time of τ_k is larger than the time needed by τ_r to reach its deadline: consequently τ_r misses its deadline. ■

2.7.1 Stability of the rate monotonic rule

From Theorem 2.69, the rate monotonic scheduler is stable iff the critical tasks have the smallest periods. If a critical task has a longer period than a non-critical one, a period transformation may be useful to artificially decrease or increase the priority of a task. A systematic procedure for period transformation can be found in [SLR86]; this method is based on the following period transformation: consider the critical task τ_i with the following attributes: $\tau_i = \{C_i = 4, T_i = D_i = \mathbf{24}, O_i = 0\}$, we can replace task τ_i by task: $\tau'_i = \{C'_i = 2, T'_i = D'_i = \mathbf{12}, O'_i = 0\}$. In general with this transformation the deadline constraints are more demanding since the first request of τ_i must complete half of its computation before or at half of its deadline. Remark that this transformation preserves the utilization factor; hence, if the utilization factor is less than 69 %, according to Theorem 2.22 the critical task set remains schedulable.

We shall see in Chapter 4 that contrary to what happens for static priority rules, the stability cannot be guaranteed for popular dynamic scheduling rules.

2.8 Conclusion

In this chapter we have presented static priority schedulers. We have first studied the rate monotonic scheduler; we have reviewed the literature and we have completed/corrected the theory, in particular concerning the optimality result and the feasibility test based on the utilization factor. Then we have examined the deadline monotonic scheduler defined for synchronous and general deadline systems. We have also considered arbitrary deadlines systems in the synchronous case with the results of Lehoczky. We have studied then asynchronous and general deadline systems, we have extended the theory by considering feasibility intervals for this kind of systems. We have also extended the theory to asynchronous and arbitrary deadlines systems, particularly concerning the periodicity and the feasibility intervals. We have shown alongside

our study that we must be very careful, that (our) intuition may lead to incorrect reasonings, for the kind of systems we consider in this work: even in very “simple” cases (e.g., synchronous and late deadline systems with 2 periodic tasks), it is difficult to anticipate their behavior.

Interesting questions for further research issued from this chapter include: the study of a sufficient condition based on the utilization factor for general deadline systems and the deadline monotonic scheduler; the study of optimal (or pseudo-optimal, i.e., heuristic) static priority assignments for asynchronous systems or synchronous arbitrary deadline systems.

Bibliography

- [ABD⁺95] N. C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *The Journal of Real-Time Systems*, 8, 1995.
- [ABRT93] N. C. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.
- [ABRW92] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Welling. Deadline monotonic scheduling theory. In Boullard and Puente, editors, *Proc. IFAC/IFIP WRTP'92*, pages 55–60, Bruges, Belgium, 1992.
- [Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, University of York, England, 1991.
- [BF97] A. A. Bertossi and A. Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research*, pages 429–443, 1997.
- [BW95] Alan Burns and Andy Wellings. A computational model for fixed priority scheduling. M.S. in parallel computer and computation, Warwick University, March 1995.
- [HaL94] Michael Gonzalez Harbour and Mark H. Klein and John Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transaction on Software Engineering*, 20(1), January 1994.
- [Leh90] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1990*, pages 201–213, Lake Buena Vista, Florida, USA, December 1990.

- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- [LM80] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.
- [LSD89] John Lehoczky, Liu Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium*, pages 166–171, 1989.
- [LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [Ser72] Omri Serlin. Scheduling of time critical processes. In *the 1972 Spring Joint Computer Conference*, volume 40 of *AFIPS Conference Proceedings*, 1972.
- [SLR86] Liu Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solution for some practical problems in prioritized preemptive scheduling. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium*, 1986.
- [Tin94a] K. W. Tindell. An extensible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 1994.
- [Tin94b] Kenneth William Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, England, 1994.

Chapter 3

Response times for static schedulers

Trouver d'abord. Chercher après
— Jean Cocteau, *Journal d'un inconnu* (Grasset).

Contents

| | | |
|------------|--|------------|
| 3.1 | Introduction | 64 |
| 3.2 | 1st request for synchronous general deadlines . . . | 65 |
| 3.3 | k^{th} request for asynchronous general deadlines . . | 67 |
| 3.3.1 | Computation of ρ_i^k | 72 |
| 3.4 | The worst case response time | 83 |
| 3.5 | The best case response time | 88 |
| 3.6 | Feasibility tests for general deadline systems . . . | 94 |
| 3.6.1 | Synchronous case | 94 |
| 3.6.2 | Asynchronous case | 95 |
| 3.7 | k^{th} request with arbitrary deadlines | 97 |
| 3.7.1 | Computation of ρ_i^k | 101 |
| 3.8 | Schedulability tests | 103 |
| 3.8.1 | The worst case response time for arbitrary systems . | 103 |
| 3.8.2 | Synchronous systems | 104 |

| | |
|--|------------|
| 3.8.3 Asynchronous systems | 104 |
| 3.9 Comparison on the various feasibility tests | 105 |
| 3.10 Conclusion | 106 |
| Bibliography | 106 |

3.1 Introduction

In this chapter we shall study the response time notion. For each request of τ_i ($1 \leq i \leq n$) we define the *response time* as the time between the arrival of the request and the completion of its processing. The computation of the response time can be useful to determine if a request of τ_i ($1 \leq i \leq n$) meets its deadline; indeed, a request of τ_i meets its deadline iff its response time (say r_i) respects the deadline (i.e., iff $r_i \leq D_i$). It follows from the definition of the feasibility interval (see Definition 2.28) that the feasibility problem (defined in the previous chapter) can be resolved by checking the response time of each request in the feasibility interval.

In the literature a single special case of response time was considered, i.e., the response time for the first request of τ_i in the synchronous and general deadline systems. We shall see the interest to consider this special case with respect to the feasibility of periodic task sets. However, we shall exhibit the interest to consider also more general and optimistic cases (e.g., asynchronous systems, arbitrary deadlines, etc.). We shall study the response time notion for the various sub-classes of periodic task sets presented in the previous chapter. For each sub-class we shall give a feasibility test based on our response time computation. Moreover, the study of the response time computation in asynchronous situations with general deadlines will be used to prove the Conjecture 2.9, i.e., the fact that the largest response time occurs for the first request of τ_i in the synchronous situation.

The remainder of the chapter is as follows: in section 3.2 we consider the simplified case of the response time computation for the 1st request in the synchronous and general deadline case; in section 3.3 we extend the computation to asynchronous situations (and general deadlines); in section 3.4 we use this generalized notion to show the fact that the synchronous case is the worst case; in section 3.5 we study the dual property of the worst case response time, i.e., the best case response time notion; in section 3.6, from the response time notion we define feasibility tests for general deadline systems (synchronous and asynchronous); in section 3.7 we extend the computation of the response time for systems with arbitrary deadlines, and in section 3.8 we define schedulability

tests for this kind of systems and we give a more quantitative approach for the proof of Lehoczky concerning the worst case response time in the first busy period.

3.2 The response time of the first request in the synchronous case

We have seen in the previous chapter that for synchronous systems (with late or general deadlines) the feasibility of the system depends only on the first request for the priority assignment given by the deadline/rate monotonic rule. Hence the interest of this response time computation. We shall consider here a synchronous and general deadline system with a static priority assignment ($\tau_1 > \dots > \tau_n$), not necessarily the deadline/rate monotonic priority assignment (even if they are optimal in this case): the formulas are given whatever the static priority rule.

Audsley and Tindell [Aud91, Tin93b] have determined this response time. In this special case, the response time is the smallest value r_i^1 such that r_i^1 is exactly equal to the total interference from higher priority tasks, plus the computation due to τ_i :

$$r_i^1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{r_i^1}{T_j} \right\rceil C_j. \quad (3.1)$$

Indeed, the computation time of τ_i is C_i and in the interval $[0, r_i^1)$ there are $\lceil \frac{r_i^1}{T_j} \rceil$ requests of τ_j ($j < i$) with a higher priority. Each of these requests of τ_j delays the request of τ_i during C_j time units.

If $r_i^1 = C_i + \sum_{j=1}^{i-1} \lceil \frac{r_i^1}{T_j} \rceil C_j$ that means that in the interval $[0, r_i^1)$ all requests of task $\tau_1, \dots, \tau_{i-1}$ and the first request of τ_i are satisfied without leaving idle times.

Equation (3.1) may have several solutions; for example (see Figure 3.1), if we consider a synchronous system $S = \{\tau_1 = \{T_1 = 2, C_1 = 1, O_1 = 0\}, \tau_2 = \{T_2 \geq 2, C_2 = 1, O_2 = 0\}\}$, Equation (3.1) has 2 solutions: $r_2^1 = 2$ and $r_2^1 = 3$. The response time of τ_i is the first instant such that all requests with a higher priority than τ_i and the first request of τ_i are satisfied (unless a deadline failure is encountered previously).

Notice that r_i^1 occurs on both sides of Equation (3.1). The minimal value for r_i^1 can be found by iteration:

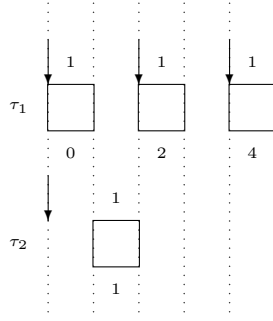


Figure 3.1: Multiplicity of the solution for Equation 3.1.

$$\begin{cases} w_0 = C_i, \text{ (initialization)} \\ w_{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_k}{T_j} \right\rceil C_j \text{ (iteration)}. \end{cases}$$

The iteration proceeds until $w_{k+1} = w_k = r_i^1$, or w_k exceeds D_i ; in the latter case, the iteration may be stopped because the first request of τ_i misses its deadline and the system is then deemed unschedulable.

Theorem 3.1 ([Tin93b]) *The iteration stops.*

Proof. We only need to show that the successive approximations w_k to r_i are monotonically increasing. By induction, we have that:

$$w_k \geq w_{k-1} \Rightarrow w_{k+1} \geq w_k$$

since, if $w_k \geq w_{k-1}$ then $w_{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_k}{T_j} \right\rceil C_j \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_{k-1}}{T_j} \right\rceil C_j = w_k$ and $C_i = w_0 \leq w_1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{C_i}{T_j} \right\rceil C_j$. Hence, after finitely many iterations either the series w_k stabilizes or exceeds D_i . ■

Tindell has shown that the iteration stops; we refine this property by showing that the iteration converges to the minimal solution, if any.

Theorem 3.2 $w_0 < w_1 < \dots < w_k \implies r_i^1 \geq w_k$.

Proof. By induction on k . The property is true initially: $C_i = w_0$ while $r_i^1 \geq C_i$. Suppose that the property is true up to k and we have $w_0 < w_1 < \dots < w_k < w_{k+1}$; by induction hypothesis $r_i^1 \geq w_k$. As a consequence, r_i^1 is at least equal to C_i plus the interference of higher priority tasks in the interval $[0, w_k)$. In other words, we have that $r_i^1 \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_k}{T_j} \right\rceil C_j = w_{k+1}$. ■

Tindell has considered the case of general deadlines; it may be noticed that his formulas remain valid for arbitrary deadlines, since we consider the response time of the first request of τ_i , so that there are no previous requests of τ_k ($k = 1, \dots, n$) to be considered, even for arbitrary deadline systems. Remark however that for synchronous and arbitrary deadlines, r_i^1 is not necessarily the largest response time of τ_i and other response times must be considered (see Conjecture 2.37); we shall consider this point in more details in section 3.8.

3.3 Response time of the k^{th} request in asynchronous and general deadline systems

Tindell and Audsley have only considered the synchronous case and the first request of a task τ_i . We shall now consider a more general case, i.e., the response time ρ_i^k for the k^{th} request of task τ_i (which occurs at time $R_i^k = O_i + (k-1)T_i$) for asynchronous and general deadline systems. Remark that the extension concerns not only the rank of the request but also the fact that the situation is asynchronous; consequently, ρ_i^1 is more general than r_i^1 , and for this reason we use a new notation.

The generalization to the asynchronous case for the k^{th} request of task τ_i is not direct since we have to consider the requests which occur before time R_i^k and which impact on the response time of ρ_i^k .

Again, the response time for the k^{th} request of τ_i is the smallest value ρ_i^k such that ρ_i^k is equal to the total interference from higher priority tasks, plus the computation due to τ_i .

Theorem 3.3 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a periodic asynchronous task set with general deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. If no deadline was missed before, the response time of the k^{th} request of task τ_i , ρ_i^k , is the smallest solution of the equation:*

$$\rho_i^k = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j$$

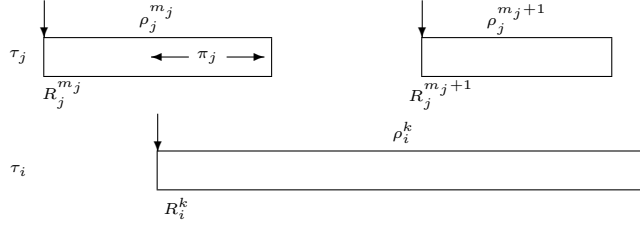


Figure 3.2: Interference of requests which occur strictly before time R_i^k .

where

$$\begin{aligned}
 R_j^p &= O_j + (p - 1)T_j \\
 m_j &= \begin{cases} \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil & \text{if } O_j < R_i^k \\ 1 & \text{otherwise} \end{cases} \\
 \pi_j &= \begin{cases} 0 & \text{if } j = 0 \text{ or } O_j \geq R_i^k \text{ or } R_j^{m_j} + D_j \leq R_i^k \\ (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{otherwise} \end{cases} \\
 z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall 1 \leq j < i \\ \max\{j \mid 1 \leq j < i \text{ and } \pi_j > 0\} & \text{otherwise} \end{cases} \\
 \tilde{m}_j &= \begin{cases} \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1 & \text{if } j > z \\ \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil + 1 & \text{otherwise} \end{cases} \\
 x^+ &= \max\{x, 0\}
 \end{aligned}$$

Proof. We suppose that the task subset $\{\tau_1, \dots, \tau_{i-1}\}$ is schedulable and that the first $(k - 1)$ requests of τ_i met their deadline. R_j^p denotes the arrival time of the p^{th} request for τ_j . The interference from higher priority tasks can be computed from the response time of some requests of higher priority tasks (so that the formula may be recursive). For a higher priority task τ_j (with $j < i$), we only have to consider the requests from the m_j^{th} , the last one that precedes strictly R_i^k (see Figure 3.2), if any (i.e., if $O_j < R_i^k$), otherwise we take $m_j = 1$ and consider the requests from the first one. Indeed, if the subset $\{\tau_1, \dots, \tau_{i-1}\}$ is schedulable, the requests of τ_j that occur before the m_j^{th} , if any, are completed (we assume that the schedulability of previous higher priority requests has already been verified) and have no direct interference with the response time

ρ_i^k (but they may have an indirect interference, through their impact on some $\rho_k^{m_k}$ with $i > k > j$). It is easy to see that $m_j = \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil$ if $O_j < R_i^k$; otherwise $m_j = 1$, hence the formula above. The k^{th} request of τ_i may be directly delayed by a part (or all) of the m_j^{th} request of τ_j if $O_j < R_i^k$, $R_j^{m_j} + D_j > R_i^k$ (otherwise the request of τ_j is certainly completed before or at time R_i^k since we assumed that the task set $\{\tau_1, \dots, \tau_{i-1}\}$ is schedulable) and the corresponding response time $\rho_j^{m_j}$ is greater than $R_i^k - R_j^{m_j}$ (and in this case the interference is equal to $\pi_j = R_j^{m_j} + \rho_j^{m_j} - R_i^k$). Suppose that two such requests of higher priority tasks (say $\tau_a > \tau_b$) have an interference in the response time of the k^{th} request of τ_i ; in that case the interference of the higher priority one is included in the interference of the lower one: since at time R_i^k both tasks (τ_a and τ_b) are active, the request of τ_b ends its execution after the request of τ_a and its response time includes the impact of the request of τ_a . Hence, $\pi_b > \pi_a > 0$ and the total interference of all requests which precede strictly the k^{th} request of τ_i is equal to π_z with $z = \max\{j | 1 \leq j < i \text{ and } \pi_j > 0\}$ ($z = 0 = \pi_z$ if no request occurring strictly before time R_i^k delays the k^{th} request of τ_i).

Let us consider now the interference of requests which occur after or at time R_i^k . For each task τ_j ($j = 1, \dots, i-1$) we shall consider the requests from the \tilde{m}_j^{th} , i.e., the first one which is not included in the term π_z .

We have to distinguish between two kinds of tasks: (i) the tasks up to τ_z ($\tau_1, \tau_2, \dots, \tau_z$), with $z > 0$, and (ii) the tasks with a priority between τ_z and τ_i ($\tau_{z+1}, \dots, \tau_{i-1}$), with $z \geq 0$.

- (i) The interference of requests of τ_q ($1 \leq q \leq z$) which occur strictly before the end of the m_z^{th} request of τ_z are included in $\rho_z^{m_z}$ (it is the case of the $(m_q + 1)^{\text{th}}$ request of τ_q in Figure 3.3). We then only have to consider the requests of τ_q which occur after or at time $R_z^{m_z} + \rho_z^{m_z} = R_i^k + \pi_z$; the first request which satisfies this condition has the rank $\left\lceil \frac{(R_i^k + \pi_z - O_q)^+}{T_q} \right\rceil + 1$. Notice that, since all requests of τ_z are schedulable, there is no request of τ_z occurring in $[R_i^k, R_i^k + \pi_z)$.
- (ii) For τ_l with $z < l < i$, we have to consider the interference of all the requests from R_i^k , i.e., from the one with the rank $\tilde{m}_l = \left\lceil \frac{(R_i^k - O_l)^+}{T_l} \right\rceil + 1$ (see Figure 3.3). It may be noticed that we may have at most one such request (the \tilde{m}_l^{th}) of τ_l which occurs between R_i^k and $R_i^k + \pi_z$. Indeed, this request cannot be terminated before the m_z^{th} request of τ_z since $l > z$.

Hence, we have shown that the interference of higher priority tasks than τ_i in

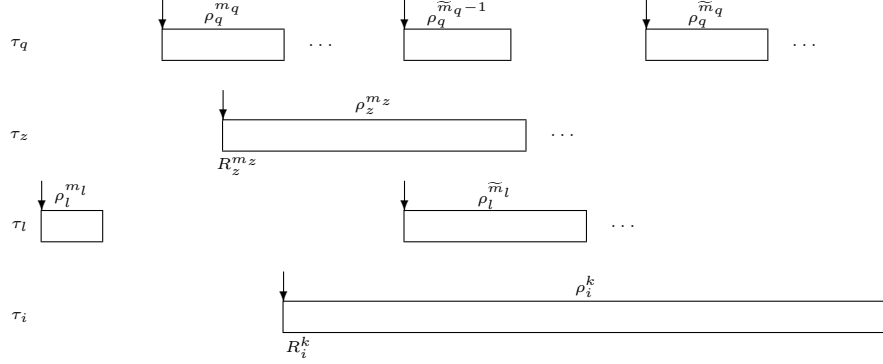


Figure 3.3: Interference of requests which occur after (or at) time R_i^k .

the interval $[R_i^k, R_i^k + \omega)$ plus the computation of τ_i is

$$I(\omega) = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \omega - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j.$$

The k^{th} request of τ_i ends its computation at the first instant $R_i^k + \omega$ such that the equality $\omega = I(\omega)$ is satisfied. \blacksquare

It may be noticed that Equation (3.1) (the response time for the first request of τ_i in the synchronous case, as defined by Audsley *et al*) is a special case of the formula in Theorem 3.3, if we choose $k = 1$ and $O_1 = O_2 = \dots = O_i = 0$, since then $\forall j : m_j = \tilde{m}_j = 1, \pi_j = 0, R_j^1 = O_j = 0$ and $z = 0 = \pi_z$.

Other interesting cases may be derived as special cases of Theorem 3.3.

Corollary 3.4 *Let S be a synchronous system ($\forall j : O_j = 0$). In this special case, ρ_i^k is the smallest solution of the equation:*

$$\begin{aligned} \rho_i^k &= C_i \\ &+ \pi_z \\ &+ \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j \end{aligned}$$

where

$$\begin{aligned}
 R_j^p &= (p-1)T_j \\
 m_j &= \begin{cases} \left\lceil \frac{R_i^k}{T_j} \right\rceil & \text{if } k > 1 \\ 1 & \text{otherwise} \end{cases} \\
 \pi_j &= \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 1 \text{ or } R_j^{m_j} + D_j \leq R_i^k \\ (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{otherwise} \end{cases} \\
 z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall 1 \leq j < i \\ \max\{j \mid 1 \leq j < i \text{ and } \pi_j > 0\} & \text{otherwise} \end{cases} \\
 \tilde{m}_j &= \begin{cases} \left\lceil \frac{R_i^k}{T_j} \right\rceil + 1 & \text{if } j > z \\ \left\lceil \frac{R_i^k + \pi_z}{T_j} \right\rceil + 1 & \text{otherwise} \end{cases} \\
 x^+ &= \max\{x, 0\}
 \end{aligned}$$

Proof. From Theorem 3.3, since $O_j = 0$ for each j and $x^+ = x$ when $x \geq 0$. ■

Corollary 3.5 *Let S be a system satisfying the condition: $\forall j : O_j \leq O_{j+1}$. In this special case of a growing offsets system, ρ_i^k is the smallest solution of the equation:*

$$\begin{aligned}
 \rho_i^k &= C_i \\
 &\quad + \pi_z \\
 &\quad + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j
 \end{aligned}$$

where

$$\begin{aligned}
 R_j^p &= O_j + (p-1)T_j \\
 m_j &= \begin{cases} \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil & \text{if } O_j < R_i^k \\ 1 & \text{otherwise, i.e., if } k = 1 \text{ and } O_j = O_i \end{cases} \\
 \pi_j &= \begin{cases} 0 & \text{if } j = 0 \text{ or } R_j^{m_j} + D_j \leq R_i^k \\ & \text{or } (k = 1 \text{ and } O_j = O_i) \\ (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{otherwise} \end{cases}
 \end{aligned}$$

$$\begin{aligned}
z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall 1 \leq j < i \\ \max\{j | 1 \leq j < i \text{ and } \pi_j > 0\} & \text{otherwise} \end{cases} \\
\tilde{m}_j &= \begin{cases} \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil + 1 & \text{if } j > z \\ \left\lceil \frac{R_i^k + \pi_z - O_j}{T_j} \right\rceil + 1 & \text{otherwise} \end{cases} \\
x^+ &= \max\{x, 0\}
\end{aligned}$$

Proof. From Theorem 3.3, since $R_i^k \geq R_i^1 = O_i \geq O_j$ ($\forall j \leq i$), and $x^+ = x$ when $x \geq 0$. \blacksquare

We shall see in section 5.3 that we can restrict offset free systems to growing offset systems, hence the interest of this special case.

3.3.1 Computation of ρ_i^k

We shall divide the computation in two parts: the computation of the term π_z (and the related numbers: $\rho_j^{m_j}, z, \tilde{m}_j$) and the computation of the lowest positive solution of the equation:

$$\rho_i^k = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j.$$

We consider first the second part of the computation; based on the term π_z , the lowest solution can be computed by iteration, in the same way than for r_i^1 .

$$w_0 = C_i + \pi_z \quad (\text{initialization}), \quad (3.2)$$

$$w_{k+1} = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + w_k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j \quad (\text{iteration}). \quad (3.3)$$

The iteration proceeds until $w_{k+1} = w_k = \rho_i^k$ or w_k exceeds D_i ; in the latter case, the iteration may be stopped because the request of τ_i misses its deadline and the system is then deemed unschedulable.

Theorem 3.6 *The number of iterations of the iterative process for the computation of ρ_i^k is bounded by $\left\lceil \frac{(D_i - C_i - \pi_z)^+}{\min_{j < i} C_j} \right\rceil + 1$.*

Proof. The property follows from the facts that $w_0 = C_i + \pi_z$, the process stops in the worst case when $w_{k-1} \leq D_i$ and $w_k > D_i$, and at each iteration w_k is increased by at least C_j time units for at least one request of some τ_j , unless the solution has been reached. ■

Remark that this maximal number of iterations is very pessimistic; in practice the number of iterations is by far lower since the iterative process can stop with $w_k < D_i$ and w_k can be increased by several C_j 's (and not necessarily the minimal one). It is difficult to estimate the actual number of iterations in all generality, since it depends in a non obvious way on the many parameters of the real-time system, in particular on the (distribution of the) system characteristics, i.e., the number of tasks, the period values, the load of the system, etc. It is not possible of course to consider all distributions of hard real-time periodic task sets. Moreover, it is hard to determine which distributions are (possibly) realistic. Such information is uncommon in the literature, and generally confidential. Notice however that it seems (from some informal sources) that one hundred is a realistic upper bound for the number n of tasks. We shall in this work limit our study by considering uniform variables (say in an interval $[a, b]$), the variation domain (i.e., a and b) being chosen in order to model realistic systems with a reasonable amount of computations, in order to simulate a large number of systems and perform statistical analyses. Of course these two requirements are antagonistic since in general the simulation of more realistic systems (e.g., larger variation domains) induces larger computations. As a consequence we shall restrict our study on limited hard real-time periodic task sets: our goal here is simply to have an indication on their behavior. More sophisticated statistical analyses, including other random variables, confidence intervals, etc., remain for further researches. These remarks concern not only the simulation results we shall give next, but more generally the various simulation results considered in this work.

We have implemented the iterative process above and we have compared the actual numbers of iterations with the bound given by Theorem 3.6 on a large set of simulations. We have applied this experimentation on randomly chosen task sets; n was chosen randomly in the interval $[20, 100]$, the periods T_i in the interval $[50, 1000]$, the deadlines D_i in the interval $[\frac{T_i}{2}, T_i]$ and the computation times C_i in order to have a large utilization factor (i.e, near 1). Before comparing the average number of iterations needed by the iterative process to compute the minimal solution and the average number of iterations given by the bound, we shall justify that such a comparison has a sense. For this purpose we shall first consider the distributions of these variables for $n \in [50, 60]$. Figures 3.4 and 3.5 show the frequency of the actual number of iterations and those given by the bound, respectively. In these histograms (and the followings) a box represents the number of individuals (say y) with a number of

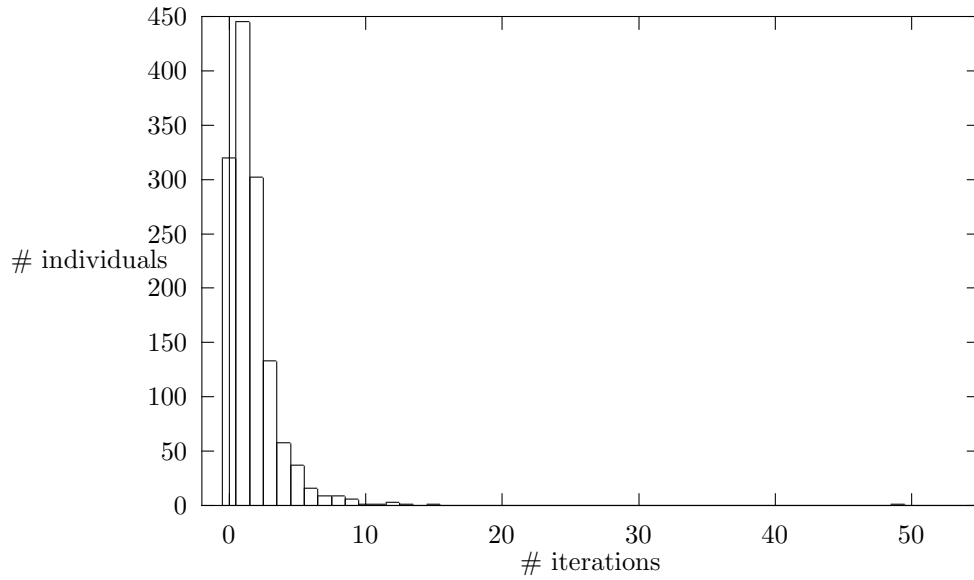


Figure 3.4: Frequency of the actual number of iterations.

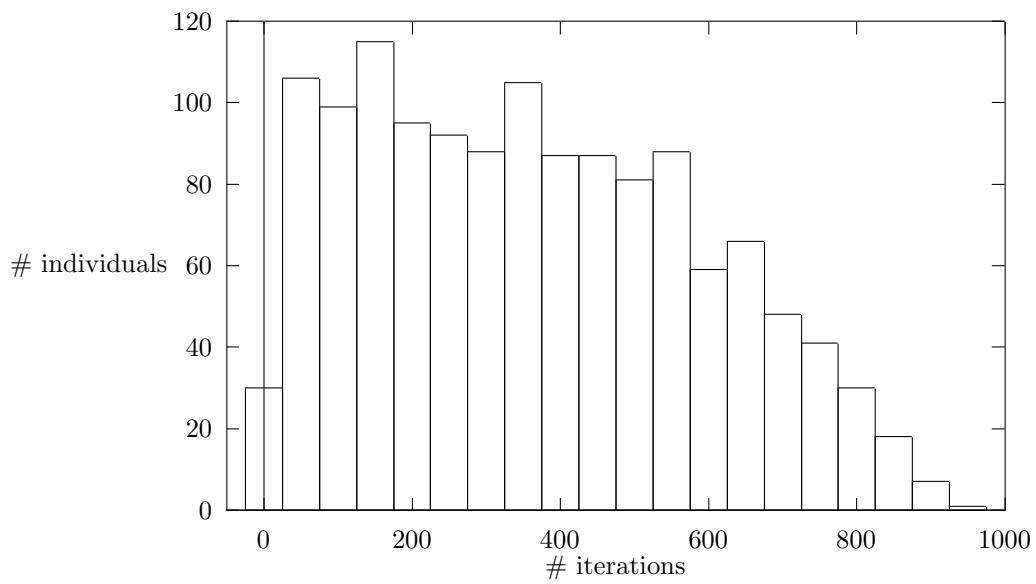


Figure 3.5: Frequency of the number of iterations given by the bound.

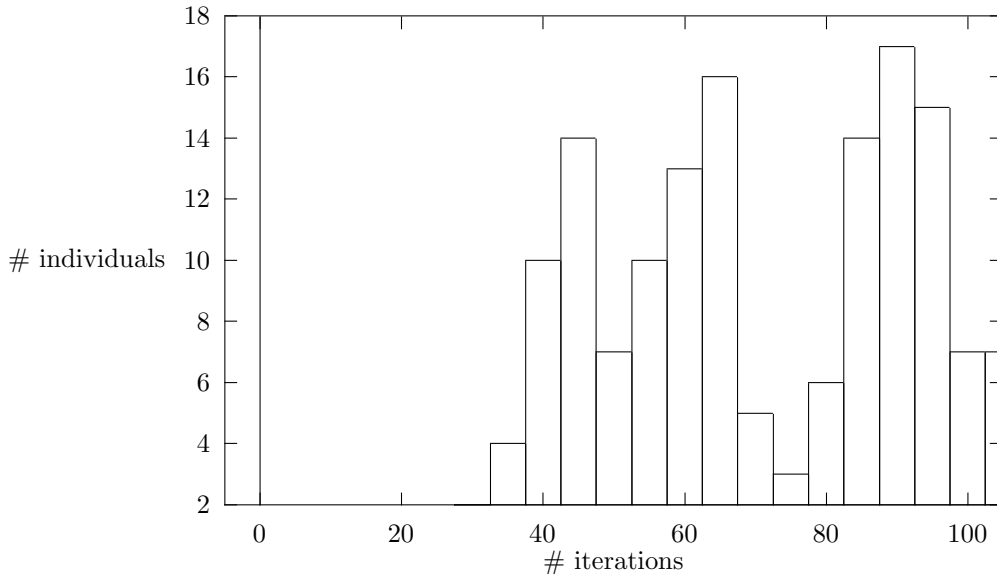


Figure 3.6: Frequency of the number of iterations given by the bound (zoom in the interval $[0, 100]$).

iterations in $(x - h, x]$ where x is the abscissa of the center of the box, y is the ordinate of the top of the box and h the width of the box. Figure 3.4 shows that the main part of the probability mass is situated in $[0, 10]$, the maximal number of actual iterations is 49 (for a single individual among a population of more than 1000). Figure 3.5 shows that the number of iterations given by the bound is significantly larger: the main part of the probability mass is situated in $[30, 800]$. Figure 3.6 shows more precisely the distribution for the number of iterations in $[0, 100]$ (with a finer precision) and shows that the minimal number of iterations given by the bound is 38. Since both distributions have the shape of a (skew) bell curve with a rather clean decrease and more particularly since the right queues of the distributions are short, it seems judicious to compare the average of both distributions. Figure 3.7 shows the distribution of the ratio $\frac{b}{\left(\frac{D_n - C_n - \pi z}{\min_{j < n} C_j}\right)}$ where b is the actual number of iterations of the iterative

process for the computation of ρ_n^k and shows that the main part of the probability mass is less than 0.01; there are very few individuals for which the ratio is greater than 0.02 and the maximum is less than 0.08. Considering the shape of this histogram, it then seems judicious to consider only the average ratio. Remark that we have observed the same kind of behavior (such as illustrated by Figures 3.4 to 3.7) for the other variation domains of n in our simulations.

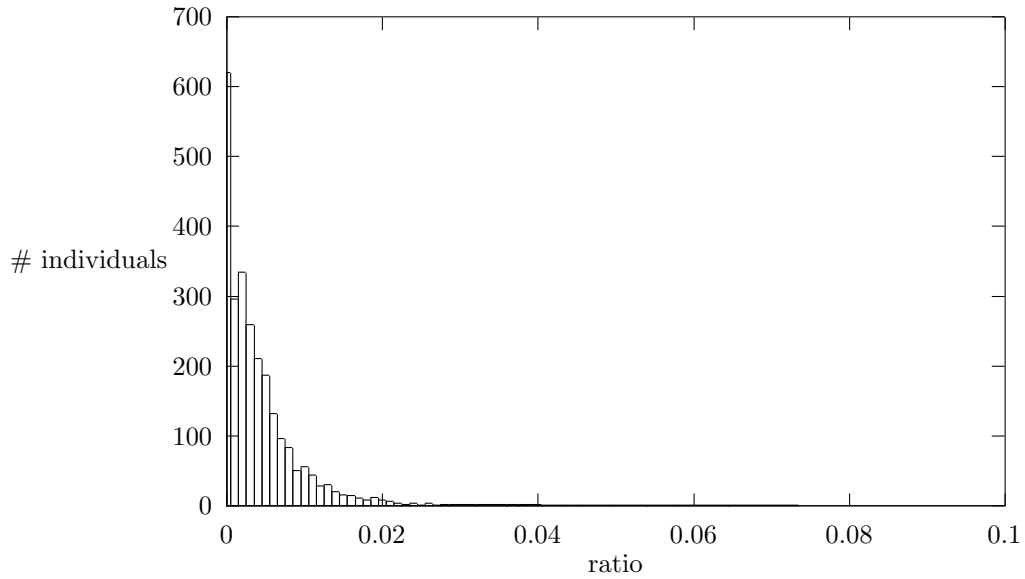
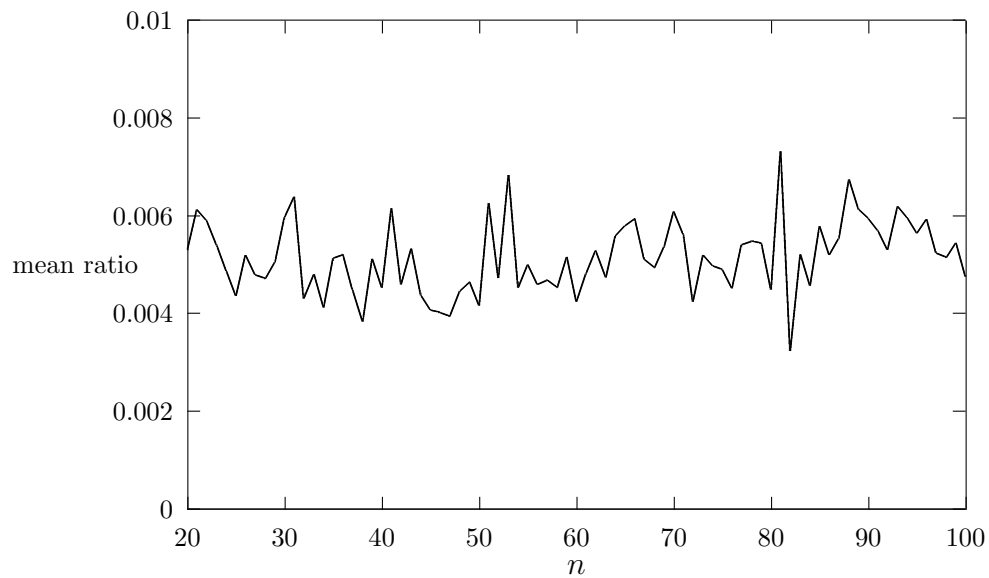


Figure 3.7: Frequency of the ratio.

Figure 3.8: Mean ratio $\frac{b}{\left(\frac{D_n - C_n - \pi_z}{\min_{j < n} C_j}\right)}$ in function of the number n of tasks.

For each simulated system we have computed the ratio $\frac{b}{\left(\frac{D_n - C_n - \pi_z}{\min_{j < n} C_j}\right)}$ where b is the actual number of iterations of the iterative process for the computation of ρ_n^k . We have then computed the average ratio: this one does not vary much with the number n of tasks, and is around 0.5 % (as exhibited in Figure 3.8). It appears therefore that the bound given by Theorem 3.6 is very pessimistic. We consider now the computation of the term π_z , together with the associated values $\rho_j^{m_j}$, z and \tilde{m}_j ; we can distinguish between 3 different methods:

- Method 1: recursively;
- Method 2: by computing all response times (say ρ_j^m) in the interval $[0, R_i^k)$ by increasing value of R_j^m , and then by computing ρ_i^k ;
- Method 3: by computing only the response times which have an impact on the term π_z .

Method 1 consists to define a function (say $f(i, k)$) which computes the response time ρ_i^k , this function needs for its computations the value of the response times $\rho_j^{m_j}$ ($j < i$ and $R_j^{m_j} < R_i^k$) so that $f(i, k)$ is recursive and before applying the iterative process for the computation of ρ_i^k , the function $f()$ recalls itself in order to determine some $\rho_j^{m_j}$'s. It is not difficult to see that the recursive calls stop in the worst case with the computation of ρ_1^p (for some p) which does not depend on other response times, consequently the maximal number of nested recursive calls is $i - 1$. Notice that in the recursive process, it may happen that $f()$ is re-evaluated many times with the same parameters.

Theorem 3.7 *The maximal time complexity of the computation of ρ_i^k using method 1 is $\beta_i + \sum_{j=1}^{i-1} \beta_j \cdot 2^{i-1-j}$, where $\beta_1 = O(1)$ and $\beta_j = O\left(\frac{D_j - C_j}{\min\{C_p | p=1, \dots, j-1\}} \times (j-1)\right)$ ($j > 1$).*

Proof. Let \mathcal{G}_i be the maximal time complexity of the function $f(i, k)$; it can be defined recursively: $\mathcal{G}_1 = \beta_1 = O(1)$, $\mathcal{G}_i = \beta_i + \sum_{j=1}^{i-1} \mathcal{G}_j$ since we may need to call $f(1, m_1), \dots, f(i-1, m_{i-1})$ before applying the iterative process for the k^{th} request of τ_i . β_i is the maximal time complexity of the iterative process for the k^{th} request of τ_i (see Theorem 3.6, each iteration being proportional to the number of terms in the sum (3.3)). We shall show by induction on i , that $\mathcal{G}_i = \beta_i + \sum_{j=1}^{i-1} \beta_j \cdot 2^{i-1-j}$. The property is true in the trivial case $\mathcal{G}_1 = \beta_1$. Suppose the property true till index $i - 1$ and consider the case of \mathcal{G}_i . By definition $\mathcal{G}_i = \beta_i + \sum_{j=1}^{i-1} \mathcal{G}_j$, and by induction hypothesis $\mathcal{G}_i = \beta_i + \sum_{j=1}^{i-1} (\beta_j + \sum_{r=1}^{j-1} \beta_r \cdot 2^{j-1-r}) = \beta_i + \sum_{j=1}^{i-1} \beta_j \cdot 2^{i-1-j}$. ■

Corollary 3.8 *The maximal number of recursive calls necessary to compute ρ_i^k is 2^{i-1} .*

Proof. Immediately follows from Theorem 3.7, by taking $\beta_j = 1 \forall j$, since $1 + \sum_{j=1}^{i-1} 2^{i-1-j} = 2^{i-1}$. ■

Remark that the time complexity analysis of the method 1 assumes to have a mono-processor system, while the processing can be parallelized which speeds up the processing; a time complexity analysis of the parallelization of method 1 remains for further researches.

It may be noticed that for k sufficiently large, the actual time complexity is close to the maximal one, since the number of direct recursive calls in the function $f(i, k)$ is near (and less than or equal to) $i - 1$, unless $R_j^{m_j} = R_i^k$ for most $j < i$, which is not very common. The maximal space complexity of the computation of ρ_i^k is $O(i)$, the maximal number of nested recursive calls (the depth of the stack). Method 1 is not well suited to handle “real size” problems (especially for large i 's) due to its huge time complexity; the interest of method 1 (and its implementation) lies mainly in the “verification” of our formulas.

Method 2 is based on the observation that the computation of ρ_i^k will generally be done in the framework of a feasibility check (see section 3.6, where all the response times in a feasibility interval like $[0, O^{max} + 2P)$ are needed); it consists in computing all response times of requests occurring in the feasibility interval, by increasing values of their arrival time. In this way, whenever the method computes a response time (say ρ_p^r) each needed $\rho_j^{m_j}$ is already computed (since we need $\rho_j^{m_j}$ iff $R_j^{m_j} < R_p^r$); for this reason we have at each instant only to know the response time of the last request of τ_j ($j = 1, \dots, i - 1$), if any. The maximal space complexity of this method 2 for the computation of ρ_i^k is $O(i)$. The maximal time complexity is $\sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot \beta_j$, or simply $O(1)$ if we consider that the previous response time computations were due anyway. Method 2 can be improved in some situations: for R_i^k sufficiently large (i.e., from $R_i^k > S_i + P_i$), according to Theorem 2.48, the schedule (and then the response times) repeats from time S_i ; hence, in the worst case we have to compute $\sum_{j=1}^{i-1} \frac{S_j - O_j}{T_j}$ response times to reach the periodic part of the schedule (according to Theorem 2.48) and $\sum_{j=1}^{i-1} \frac{P_j}{T_j}$ to compute the response times $\rho_j^{k_j}$ corresponding to $\rho_j^{m_j}$ (i.e., $k_j = \min\{k | k = m_j \bmod \frac{P_j}{T_j} \text{ and } R_j^k \geq S_j\}$). The maximal time complexity of the computation of ρ_i^k is $\sum_{j=1}^{i-1} \frac{P_j + S_j - O_j}{T_j} \cdot \beta_j$. The actual complexity is close to the worst case, for the same reason as before.

Method 3 consists in computing only the response times which have an impact on the term π_z in the computation of (a single) ρ_i^k . Let Y_j be inductively

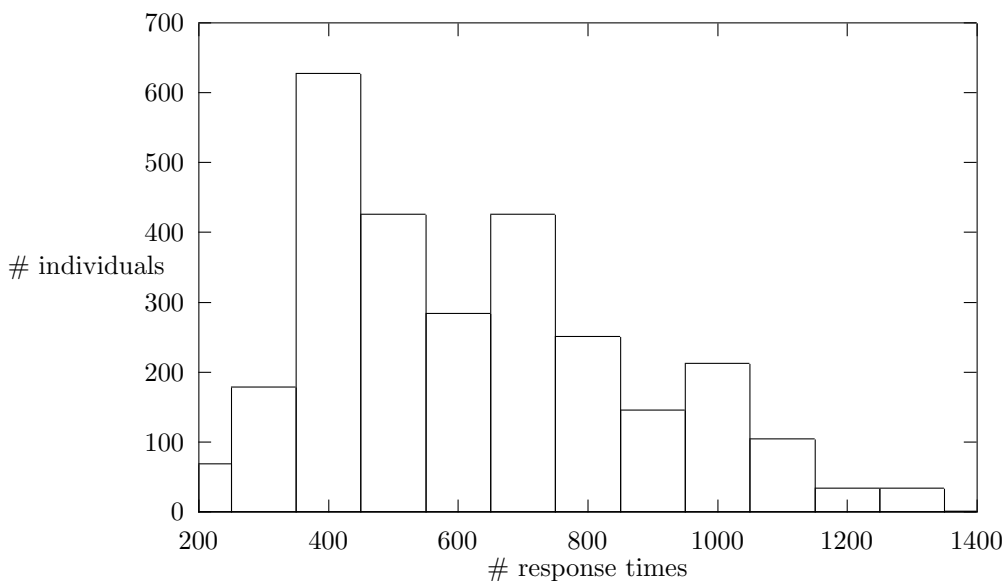


Figure 3.9: Actual number of response times considered by method 3; the average is 671 and the standard deviation is 249.

defined by $Y_i = R_i^k$, $Y_j = O_j + \left\lfloor \frac{(\min_{k=j+1}^i Y_k - O_j)^+}{T_j} \right\rfloor T_j$, for $j = i-1, i-2, \dots, 1$. By Lemma 2.56, ρ_i^k only depends on the requests of τ_j from time Y_j . Method 3 consists in computing first the values Y_i, Y_{i-1}, \dots, Y_1 , and then the response times for the requests occurring from Y_p for τ_p ($p = 1, \dots, i-1$) (before applying the iterative process for ρ_i^k) by increasing values of their arrival time. In this way, whenever method 3 computes a response time (say ρ_p^r) all needed $\rho_j^{m_j}$'s are already computed (since we need $\rho_j^{m_j}$ iff $R_j^{m_j} < R_p^r$); moreover, like before, we only have at each instant to know the response time of the last request of τ_j ($j = 1, \dots, i-1$), if any. Hence, the maximal space complexity of the method 3 of the computation of ρ_i^k is again $O(i)$. From the definition of the value Y_j it follows that method 3 computes at most $\sum_{r=1}^{i-1} \left\lfloor \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rfloor$ different response times; hence the maximal time complexity of the computation of ρ_i^k using method 3 is bounded by $\sum_{r=1}^{i-1} \left\lfloor \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rfloor \cdot \beta_r$.

Table 3.1, summarizes the complexity of the 3 methods. The superiority of the method 3 is obvious for a single ρ_i^k computation, since the maximal number of different response times computed by the method is pseudo-polynomial³

³We use here the terminology of Garey and Johnson [GJ79]: an algorithm is called a *pseudo-polynomial time algorithm* if its complexity function is bounded above by a poly-

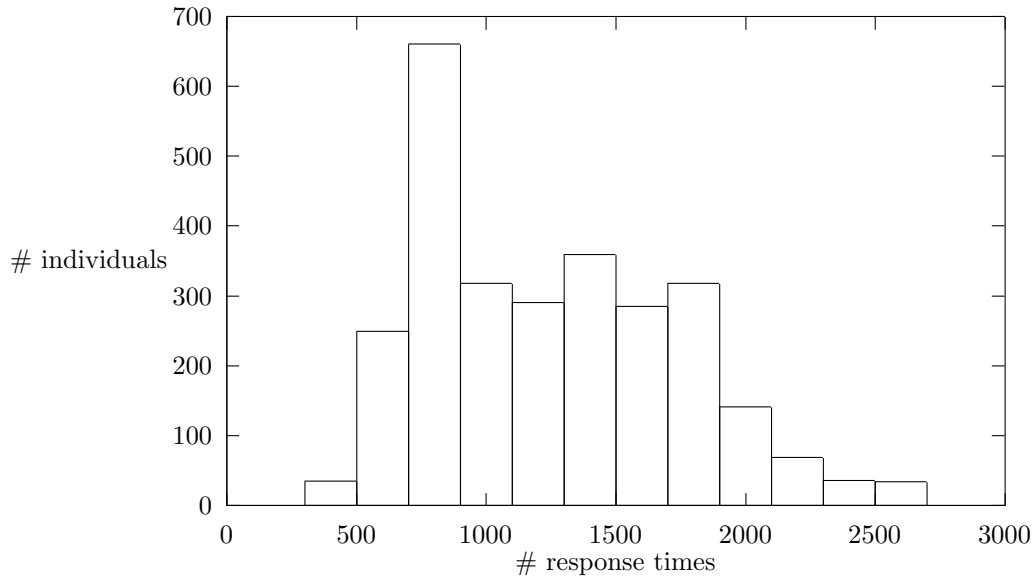


Figure 3.10: Number of response times given by the bound; the average is 1353 and the standard deviation is 493.

| Method | Space | # Response times | Actual # of response times |
|----------|-------|--|--|
| Method 1 | i | 2^{i-1} | $\approx 2^{i-1}$ |
| Method 2 | i | $\sum_{j=1}^{i-1} \left\lfloor \frac{R_i^k}{T_j} \right\rfloor$ or 1 | $\approx \sum_{j=1}^{i-1} \left\lfloor \frac{R_i^k}{T_j} \right\rfloor$ or 1 |
| Method 3 | i | $\sum_{r=1}^{i-1} \left\lfloor \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rfloor$ | $\frac{1}{2} \sum_{r=1}^{i-1} \left\lfloor \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rfloor$ |

Table 3.1: Space and time complexities of method 1, method 2 and method 3

in terms of the system characteristics (i.e., n, T_i, D_i, C_i), while this number grows exponentially for method 1 and method 2; moreover their actual number of response times computed are close to the maximal one. We have already shown that the maximal time complexity of the iterative process is very pessimistic; we shall now show that the maximal number of different response times considered by method 3 is also pessimistic. Indeed, while $\min\{Y_p | p = j, \dots, i-1\} - Y_{j-1} < T_{j-1}$, in the average we may expect that the actual difference is about half that value (if $O_{j-1} \leq \min\{Y_p | p = j, \dots, i-1\}$). In order to check that, we have applied method 3 on randomly chosen task sets; n was chosen randomly in the interval $[20, 100]$, the periods in the interval $[50, 1000]$. We consider first the distribution of the actual number of response times considered by the method 3 and of the number given by the bound for $n \in [30, 50]$. Figures 3.9 and 3.10 show the frequency of the actual number of response times and those given by the bound, respectively. Figure 3.11 shows the frequency of the ratio $\frac{b}{\sum_{r=1}^{i-1} \left\lceil \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rceil}$ where $b = \sum_{r=1}^{i-1} \left\lceil \frac{(R_i^k - Y_r)^+}{T_r} \right\rceil$ is the ac-

tual number of response times considered by method 3. For similar reasons than those introduced for the estimation of the actual number of iterations of the iterative process which compute the response time, it seems again judicious to compare the average of both distributions and the average ratio. Again, we have observed the same behavior for the other variation domains of n in our simulations. For each simulated system we have computed the ratio $\frac{b}{\sum_{r=1}^{i-1} \left\lceil \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rceil}$. We have then computed the average ratio: this one does

not vary much with the number n ($n \geq 20$) of tasks and is around 50 % (see Figure 3.12), as expected. Hence, the bound $\sum_{r=1}^{i-1} \left\lceil \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rceil$ is pessimistic by a factor 2.

mial function of $Length[Input]$ or $Max[Input]$. The $Input$ is in this case the characteristics of the system (i.e., n, T_i, D_i, C_i).

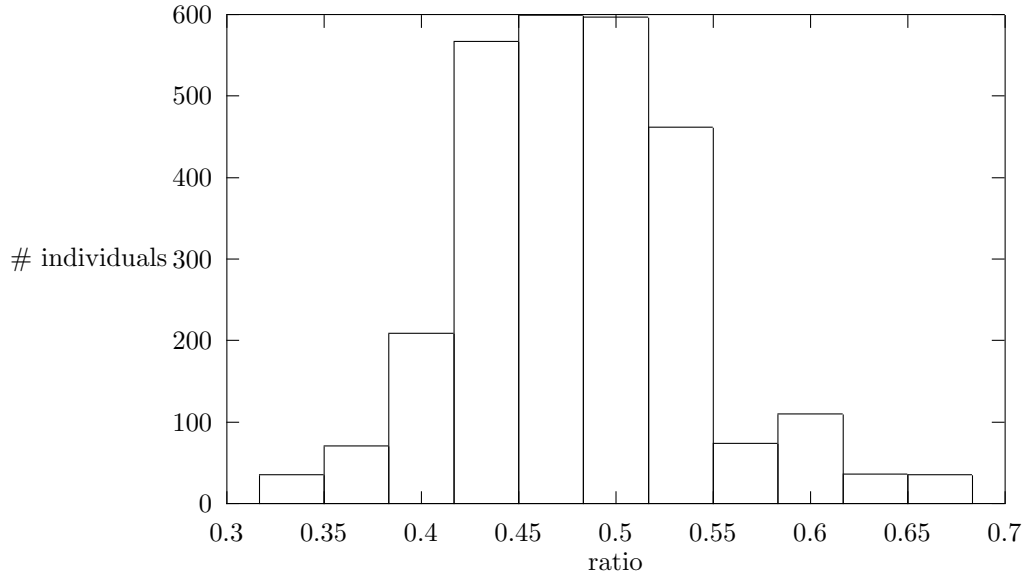


Figure 3.11: Frequency of the ratio; the average is 0.5 and the standard deviation is 0.06.

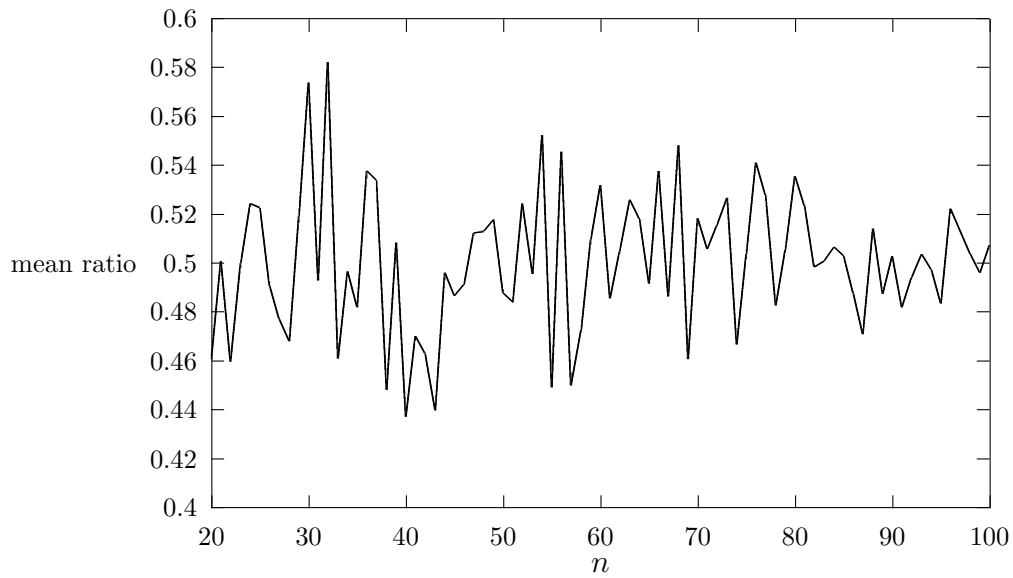
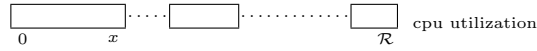


Figure 3.12: Mean ratio $\frac{b}{\sum_{r=1}^{i-1} \left\lceil \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rceil}$ in function of n .

Figure 3.13: Definition of $\tilde{\rho}_i^1$.

3.4 The worst case response time for general deadline system

We consider now a simplified case of asynchronous response time computation, instrumental to prove that the synchronous case leads to the worst response time (we consider here general deadline systems): the response time for the first request of τ_i in the special asynchronous case where $O_i = \min_{j \leq i} \{O_j\} = 0$, is the smallest value of $\tilde{\rho}_i^1$ such that.

$$\tilde{\rho}_i^1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(\tilde{\rho}_i^1 - O_j)^+}{T_j} \right\rceil C_j. \quad (3.4)$$

Equation (3.4) follows from Theorem 3.3 when $k = 1$ and $O_j \geq O_i$ ($j = 1, \dots, i$), since $R_i^1 = 0$, $z = 0$, $\pi_z = 0$.

Another preliminary result states that, if there exists an interval $[0, \mathcal{R})$ larger than the demand occurring during this interval, it follows that $\tilde{\rho}_i^1$ is less than \mathcal{R} . More formally we have:

Lemma 3.9 $\mathcal{R} \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(\mathcal{R} - O_j)^+}{T_j} \right\rceil C_j \Rightarrow \tilde{\rho}_i^1 \leq \mathcal{R}$

Proof. We can assume that $\mathcal{R} > C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(\mathcal{R} - O_j)^+}{T_j} \right\rceil C_j$, otherwise the property is immediate. $\mathcal{R} > C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(\mathcal{R} - O_j)^+}{T_j} \right\rceil C_j$ means that the interval $[0, \mathcal{R})$ has idle time unit(s). Let x be the time that precedes the first idle time unit (see Figure 3.13). By construction, in the interval $[0, x)$ all requests of $\tau_1, \dots, \tau_{i-1}$ and the first request of τ_i are exactly satisfied (we consider only the first request of τ_i , e.g., $T_i = \infty$), so that $x = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(x - O_j)^+}{T_j} \right\rceil C_j$ and $\tilde{\rho}_i^1 \leq x < \mathcal{R}$. ■

Lemma 3.10 *Let $S = \{\tau_1, \dots, \tau_n\}$ be an asynchronous and arbitrary deadline system feasible for some scheduling algorithm such that the schedule is periodic from time t with a period of P . Then there is necessarily an idle point (in the periodic part of the schedule) each P time units.*

Proof. If the utilization factor is strictly less than 1 the property is obvious since there is necessarily an idle time unit (and consequently an idle point) in the interval $[t, t + P)$. If $U = \sum_{i=1}^n \frac{C_i}{T_i} = 1$, we define r as the first time instant such that from time r the CPU remains busy; since $U = 1$ such a time instant exists and $0 \leq r \leq t$. The requests occurring strictly before time r do not impact the schedule⁴ after or at time r . For this reason, we shall only consider the requests from time r . Let x_i be the time instant of the first request of τ_i after or at time r : $x_i = \min\{O_i + a \cdot T_i \mid O_i + a \cdot T_i \geq r\}$. We shall also change the time origin: $x'_i = x_i - r$. The schedule is periodic from time $t - r$; let $k = \min\{j \mid j \cdot P \geq t - r\}$. We shall show that time $k \cdot P$ is an idle point. The CPU is permanently utilized in the interval $[0, kP)$; it follows that time instant $k \cdot P$ is an idle point iff

$$\sum_{i=1}^n \left\lceil \frac{k \cdot P - x'_i}{T_i} \right\rceil C_i = k \cdot P \quad (3.5)$$

Since $U = 1$, it follows that

$$\sum_{i=1}^n \left\lceil \frac{k \cdot P - x'_i}{T_i} \right\rceil C_i \leq \sum_{i=1}^n \left\lceil \frac{k \cdot P}{T_i} \right\rceil C_i = k \cdot P \sum_{i=1}^n \frac{C_i}{T_i} = k \cdot P.$$

and since the CPU is permanently utilized in this period we have:

$$\sum_{i=1}^n \left\lceil \frac{k \cdot P - x'_i}{T_i} \right\rceil C_i \geq k \cdot P$$

Hence, the relation (3.5) is satisfied, $k \cdot P$ is an idle point, and since $k \cdot P \geq t - r$ this idle point repeats every P time units. ■

Note that the previous Lemma is valid for a large class of scheduling algorithms, including static priority schedulers (considered in this chapter), and dynamic ones considered in Chapter 4.

⁴Here we assume that the decision of the scheduling algorithm at each time instant (say time q) only depends on the active requests at time q .

We have seen in Chapter 2 that major properties concerning the rate/deadline priority assignment are based on the fact that the synchronous case is the worst case. We have also exhibited the fact that the classical proof (in the literature) of this main property is not convincing; for this reason we have considered this property as a conjecture. We shall here show this property using our response time computation. It may be noticed that we shall show a more general property than Conjecture 2.9, since we shall show that if the first request of task τ_i meets its deadline in the synchronous case then all requests of τ_i for all general asynchronous situations (i.e., whatever the O_i 's) meet their deadline.

We shall first show this property for asynchronous situations where the task sub-set $\tau_1, \dots, \tau_{i-1}$ is schedulable (Lemma 3.11) and then we shall relax this assumption (Theorem 3.13).

Lemma 3.11 *If the first request of τ_i meets its deadline in the general synchronous case (which implies that the first requests of the tasks $\tau_1, \dots, \tau_{i-1}$ have also met their deadlines) then for all asynchronous situations (i.e., whatever the O_i 's) such that $\tau_1, \dots, \tau_{i-1}$ is a schedulable task set, we have that any request of τ_i meets its deadline if the previous requests of τ_i met their deadlines.*

Proof. The first request of τ_i meets its deadline in the synchronous case, hence:

$$\exists r_i^1 : r_i^1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{r_i^1}{T_j} \right\rceil C_j \text{ and } r_i^1 \leq D_i. \quad (3.6)$$

Consider now an asynchronous situation and a request of τ_i which occurs at time v . We have only to consider higher priority tasks and from an argument similar to the one used in the proof of Lemma 5.7, among all requests of $\tau_1, \dots, \tau_{i-1}$ we can ignore the first ones, which do not impact on the computation of the response time of τ_i . Indeed, let t_i be the last instant before v ($0 \leq t_i \leq v$) where all requests of τ_j ($j < i$) occurring strictly before t_i (if any) are terminated before or at time t_i : the requests of τ_j ($j < i$) strictly before t_i do not have any impact on the response time of τ_i (see Figure 3.14; t_i is the last idle point before or at v for the task sub-set $\{\tau_1, \dots, \tau_{i-1}\}$).

Ignoring the request(s) of τ_j before t_i is equivalent to refine the offsets:

$$O'_j = \min_{k \geq 0} \{O_j + k \cdot T_j \mid O_j + k \cdot T_j \geq t_i\} \quad (\forall j : 1 \dots i-1).$$

We may also refine the last offset: $O'_i = v$, since we assumed that the previous (general) deadlines were met, so that the requests of τ_i which occur before time v have no impact of the request on τ_i at time v .

Figure 3.14: Definition of t_i .

We can also change the time origin:

$$O_j'' = O_j' - \min\{O_1', \dots, O_i'\} \quad (\forall j : 1 \dots i).$$

By construction of the new task set, in the interval $[0, O_i'')$ the cpu is permanently utilized by the task set $\{\tau_1, \dots, \tau_{i-1}\}$. In this case, we can compute the response time of the first request of τ_i (say x_i) by assuming that the request of τ_i arrives at time 0, since in this case we have that $x_i = \tilde{\rho}_i^1 - O_i''$, $\tilde{\rho}_i^1$ being the response time for the new request. In this special asynchronous case, from Equation 3.4, $\tilde{\rho}_i^1$ is the smallest solution of:

$$\tilde{\rho}_i^1 = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(\tilde{\rho}_i^1 - O_j'')^+}{T_j} \right\rceil C_j. \quad (3.7)$$

From Equation (3.6) we have that $r_i^1 \geq C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(r_i^1 - O_j'')^+}{T_j} \right\rceil C_j$ and by Lemma 3.9 we have that the Equation (3.7) has a solution $\tilde{\rho}_i^1$ with $0 < \tilde{\rho}_i^1 \leq r_i^1$. By construction, we have that $\tilde{\rho}_i^1 > O_i''$, and the response time x_i of τ_i in the original asynchronous case satisfies the relations $0 \leq x_i = \tilde{\rho}_i^1 - O_i'' \leq r_i^1 \leq D_i$, where r_i^1 is the response time in the synchronous case. ■

It may be noticed that this proof also shows that the worst case response time of task τ_i occurs when all higher priority tasks are released at the same time than τ_i .

Corollary 3.12 *If the first request of τ_i meets its deadline in the general synchronous case (which implies that the first requests of the tasks $\tau_1, \dots, \tau_{i-1}$ have also met their deadlines) then for all asynchronous situations (i.e., whatever the O_i 's) such that $\tau_1, \dots, \tau_{i-1}$ is a schedulable task set, we have that any request of τ_i meets its deadline.*

Proof. From Lemma 3.11 the property is true for the first (asynchronous) request of τ_i , then for the second one,... ■

$$r_i^1 \leq D_i \implies r_i^1 = \max_{\substack{O_1, O_2, \dots, O_n \in \mathbb{N} \\ k \in \mathbb{N}}} \rho_i^k. \quad (3.8)$$

Theorem 3.13 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a general deadline task set. If this task set is schedulable in the synchronous case, then this task set is schedulable in all asynchronous cases.*

Proof. By induction on n . The property is true in the trivial case, where $n = 1$. Suppose that the property is true up to n , and consider $\{\tau_1, \tau_2, \dots, \tau_n, \tau_{n+1}\}$: if it is a schedulable task set in the synchronous case, by induction hypothesis, the task set $\{\tau_1, \tau_2, \dots, \tau_n\}$ is schedulable in all asynchronous cases and we are able to apply Corollary 3.12: any request of τ_{n+1} meets its deadline, so that the task set $\tau_1, \tau_2, \dots, \tau_n, \tau_{n+1}$ is schedulable in all asynchronous cases. ■

Corollary 3.14 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a general deadline task set. If $r_i^1 \leq D_i$ ($i = 1, \dots, n$), then this task set is schedulable in all asynchronous cases.*

Proof. Immediately follows from Theorem 3.13 and Lemma 3.11. ■

From Corollary 3.12 another interesting property may be derived.

Theorem 3.15 *Let $S = \{\tau_1, \dots, \tau_n\}$ be a feasible and general deadline synchronous system ($O_i = 0$, $i = 1, \dots, n$) with the priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$ and t be any time instant. Then, the interval $[0, t)$ maximizes the CPU usage by tasks τ_1, \dots, τ_n among all intervals of length t in all schedules (i.e., among all offset combinations).*

Proof. We can assume that in the interval $[0, t)$ there is at least one idle time unit, otherwise the property is immediate. The CPU usage in the interval $[0, t)$ is $t - x$, where x the number of idle time units left by tasks τ_1, \dots, τ_n in the interval $[0, t)$; it may also be viewed as the interference of tasks τ_1, \dots, τ_n on an additional task τ_{n+1} with the lowest priority, with $T_{n+1} = D_{n+1} = \infty$, $C_{n+1} = x$ and $O_{n+1} = 0$. Hence, τ_{n+1} is running at time $y \in [0, t)$ iff the original system was idle at time y . Let r_{n+1}^1 be the response time of the first (and only) request of τ_{n+1} : by construction we have that $r_{n+1}^1 \leq t$ and there is no idle time units in $[r_{n+1}^1, t)$. Consider now the schedule of an asynchronous system in the interval $[\delta, \delta + r_{n+1}^1)$ with $T_{n+1} = D_{n+1} = \infty$, $C_{n+1} = x$ and $O_{n+1} = \delta$; let ρ_{n+1}^1 be the response time of the first (and only) request of τ_{n+1} . From Equation (3.8) we have that $r_{n+1}^1 \geq \rho_{n+1}^1$, which implies that the number of idle time units in $[\delta, \delta + r_{n+1}^1)$, and a fortiori in $[\delta, \delta + T)$, in the original asynchronous schedule is greater than or equal to the number x of idle time units in $[0, r_{n+1}^1)$ in the synchronous schedule. ■

We shall see the interest of this property in the next section, where we consider the notion of best response time.

3.5 The best case response time for general deadline systems

In the previous sections we have studied the response time of the k^{th} request of task τ_i (ρ_i^k), and we have seen that the response time of the first request of τ_i in the synchronous case (r_i^1) is the worst response time among all asynchronous situations. We shall here present the dual property of the worst case response time (Theorem 3.3): the *best case response time* ρ_i^* . We shall see in Chapter 5 the interest to determine the best case response time for the feasibility of offset free systems; we shall not give details here; we are concerned in this section only with the value of this best case response time and the configuration regarding the task requests which leads to this response time.

The best case response time notion can be understood as the dual notion with respect to the worst case response time r_i^1 , which occurs when all other higher priority tasks are synchronized with the request of τ_i . We suppose that the task sub-set $\{\tau_1, \tau_2, \dots, \tau_{i-1}\}$ is schedulable in the synchronous case and we shall exhibit the configuration of the task requests which leads to the minimal response time for a request of τ_i (assuming that the previous requests of τ_i meet their deadline). In the same way that r_i^1 can be larger than D_i so that the system is not schedulable in the synchronous case, the best case response time can be larger than D_i (it may even be infinite, if $\sum_{j=1}^{i-1} \frac{C_j}{T_j} = 1$) and consequently the system is then unschedulable in all asynchronous situations. Remark that if the best case response time is less than D_i nothing can be inferred regarding the feasibility of the system; we shall see however in Chapter 5 the interest of this notion.

It is not difficult to see that the best case response time for a task τ_i is C_i (the computation time of task τ_i), if we choose for example $O_i = 0$ and $O_j = C_i$ ($\forall j < i$). However it is more relevant from a schedulability point of view to determine the situation where the response time is minimum in the periodic part of the schedule (assuming that the previous requests of τ_i meet their deadline) or more precisely during a period of this periodic part (e.g., in the interval $[S_i, S_i + P_i]$ with $P_i = \text{lcm}\{T_j | j \leq i\}$, S_i being inductively defined by $S_1 = O_1$, $S_i = O_i + \lceil \frac{(S_{i-1} - O_i)^+}{T_i} \rceil T_i$ ($i = 2, 3, \dots, n$) according to Theorem 2.48).

Example 3.16 Consider the system $\Gamma = \{\tau_1 = \{T_1 = D_1 = 5, C_1 = 3, O_1 = 0\}, \tau_2 = \{T_2 = D_2 = 9, C_2 = 3, O_2 = 0\}\}$; in this case, the periodic behavior starts at time $t = 0$ and its period is $\text{lcm}\{5, 9\} = 45$. The interval $[0, 45)$ contains 5 requests of τ_2 with the following response times: $\rho_2^1 = 9, \rho_2^2 = 6, \rho_2^3 = 6, \rho_2^4 = 7$ and $\rho_2^5 = 8$ (see Figure 3.15). We have that the best response time of requests of task τ_2 occurs during the second and the third request of τ_2

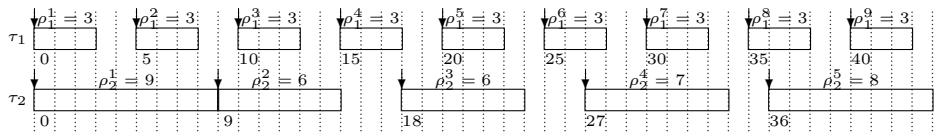


Figure 3.15: Response times in the periodic part of the schedule.

with $\rho_1^* = \rho_2^2 = \rho_2^3 = 6$; moreover, in this schedule, all relative phasings between τ_1 and τ_2 occur, due to the fact that T_1 and T_2 are relatively prime. Hence for this task set all offset assignments define equivalent asynchronous systems: they define the same periodic behavior. In Chapter 5 we shall rigorously study this notion of equivalence and its implications. We shall not give more details here. As a consequence the best periodic response time of τ_2 is 6 for all choices of offsets, and not only for the one exhibited here. ■

Remark the duality of the worst/best response time:

- If the worst case response time of task τ_i is less or equal than D_i , the system is schedulable in the synchronous case (and consequently in all asynchronous situations).
- If the best case response time of task τ_i is greater than D_i , the system is unschedulable in all asynchronous situations, i.e., whatever the O_i 's.

We shall characterize which situation, regarding the relative phasings between task requests, is the more favorable for a request of τ_i (there are possibly several such situations, like in Example 3.16). We are looking for a systematic method which determines this situation, without considering all requests of τ_i during a period of the schedule for all (non equivalent) offset assignments.

We shall show with Theorem 3.19 that the best case in terms of response time in the periodic behavior occurs when all requests of higher priority tasks than τ_i are synchronized with the completion of the corresponding request of τ_i , i.e., each task τ_j ($1 \leq j < i$) makes a new request at time t , where t is the completion time of the request of τ_i . This may be checked in the previous example, where the second request of τ_2 arrives at time $t = 9$ and completes its execution at time $t = 15$, when τ_1 makes a new request (but there may be other “accidental” solutions, like the third request of τ_2).

In order to prove this property, we introduce a preliminary result.

Lemma 3.17 *Let $\Gamma = \{\tau_1, \dots, \tau_n\}$ be a feasible asynchronous and general deadline periodic task set. Let S_i be inductively defined by $S_1 = O_1$, $S_i =$*



Figure 3.16: Periodic behavior of the schedule.

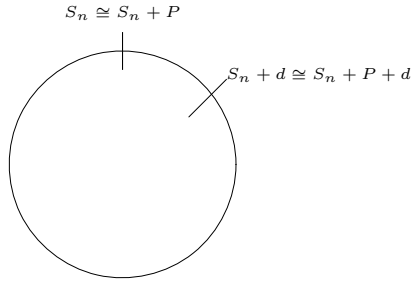


Figure 3.17: Cyclical interpretation on the periodic behavior of the schedule.

$O_i + \lceil \frac{(S_{i-1} - O_i)^+}{T_i} \rceil T_i$ ($i = 2, 3, \dots, n$). The CPU usage of tasks $\{\tau_1, \dots, \tau_n\}$ in the interval $[S_n, S_n + P)$ is exactly equal to $\sum_{i=1}^n \frac{P}{T_i} C_i$.

Proof. Consider a task τ_j ($1 \leq j \leq n$); we shall show that in the interval $[S_n, S_n + P)$ the CPU usage of τ_j is exactly equal to $\frac{P}{T_j} C_j$. If a request of τ_j occurs at time S_n the property is trivial. In the opposite case, we can define the instant $S_n + d$ to be the instant of the first request of τ_j after time S_n (by definition of S_n we have that $d < T_j$). The interference of τ_j in the interval $[S_n + d, S_n + P + d)$ is exactly equal to $\frac{P}{T_j} C_j$ (T_j divides P , hence $\frac{P}{T_j} = \lceil \frac{P}{T_j} \rceil$). The schedule is periodic from S_n , with a period of P , hence the intervals $[S_n, S_n + P)$ and $[S_n + d, S_n + d + P)$ are identical if we consider the CPU usage during one period of the periodic behavior of the system (see Figure 3.16): in this case we can represent this part of the schedule by a “circle” (see Figure 3.17) and consider the CPU usage on it. ■

We have shown that the CPU usage of tasks $\{\tau_1, \dots, \tau_n\}$ on the circle is exactly equal to $\sum_{i=1}^n \frac{P}{T_i} C_i$. More generally, for a feasible schedule σ (of the task set $\{\tau_1, \dots, \tau_n\}$), we define $\varphi(\sigma, a, b)$ as the schedule (still of the task set $\{\tau_1, \dots, \tau_n\}$) constructed by repeating the slice $[a, b)$ of the schedule σ for $t \in (-\infty, a)$ and $t \in [b, +\infty)$. It may be noticed that the Figure 3.17 is a finite representation of $\varphi(\sigma, S_n, S_n + P)$; σ and $\varphi(\sigma, S_n, S_n + P)$ have thus the same periodic behavior (the second one is infinitely periodic to the left as to the right) and are both feasible (assuming that σ is feasible).

Lemma 3.18 *Let $\Gamma = \{\tau_1, \dots, \tau_n\}$ be a feasible synchronous general deadline task set and $l \in \mathbb{N}$. The interval $[P - l, P)$ minimizes the CPU usage by tasks τ_1, \dots, τ_n among all intervals of length l in all periodic parts of all asynchronous schedules (i.e., among all offset combinations).*

Proof. This property is in a way the dual property of the worst case response time (the synchronous case), according to the Lemma 3.17. During a period of the periodic behavior of schedule the interference is exactly equals to $\sum_{i=1}^n \frac{P}{T_i} C_i$. We can divide the interval in two parts: $I_1 = [0, P - l)$ and $I_2 = [P - l, P)$; Theorem 3.15 shows that I_1 in the synchronous case maximizes the CPU usage by task τ_1, \dots, τ_n among all intervals of length $P - l$ in the periodic part of all asynchronous schedules; by Lemma 3.17 it follows that I_2 in the synchronous case minimizes this CPU usage among all intervals of length l in the periodic part of all asynchronous schedules. ■

Theorem 3.19 *Let $\Gamma = \{\tau_1, \dots, \tau_{i-1}\}$ be a general deadline task set schedulable in the synchronous case. The best case in terms of response time for a request of τ_i (assuming that the previous requests of τ_i meet their deadline) included in the periodic behavior of the system occurs when the requests of tasks $\tau_1, \dots, \tau_{i-1}$ are synchronized with the completion of the corresponding request of τ_i (i.e., each task τ_j ($1 \leq j < i$) makes a new request at time t , where t is the completion time of the request of τ_i).*

Proof. Determining the best case in terms of response time for a request of τ_i , during the periodic behavior, is equivalent to find the smaller interval $[t_1, t_2)$ which contains exactly C_i free time units, in the periodic part of all schedules (i.e., for all offset combinations). From Lemma 3.18, we have that the upper bound of this interval t_2 corresponds to the synchronization with the arrivals of requests of $\tau_1, \dots, \tau_{i-1}$. ■

In order to determine this best response time, without loss of generality, we can choose $t_2 = P_{i-1}$ in the schedule $\varphi(\sigma, 0, P_{i-1})$ where σ is the schedule of the synchronous task sub-set $\{\tau_1, \dots, \tau_{i-1}\}$. Then the lower bound is the largest t_1 such that the interval $[t_1, P_{i-1})$ contains exactly C_i idle time units (see Figure 3.18). In this case the response time of the request of τ_i which occurs at time t_1 is $\rho_i^* = P_{i-1} - t_1$. If $\rho_i^* = P_{i-1} - t_1 > D_i$, the best response time of τ_i is larger than the deadline D_i and consequently all requests of τ_i fail in all asynchronous situations. Notice that τ_i is executing at time t_1 , otherwise t_1 could be chosen further, $U_{i-1} = \sum_{j=1}^{i-1} \frac{C_j}{T_j} < 1$, otherwise there is no idle time unit in $\varphi(\sigma, 0, P_{i-1})$ and $\rho_i^* = \infty > D_i$, and $\rho_i^* \leq \lceil \frac{C_i}{k_{i-1}} \rceil \cdot P_{i-1}$ with $k_{i-1} = (1 - U_{i-1}) \cdot P_{i-1}$, since there are k_{i-1} idle time units in each hyperperiod

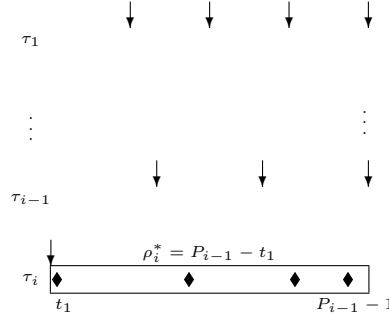


Figure 3.18: Phasing configuration which leads to the best response time (the C_i last idle time units are marked \blacklozenge).

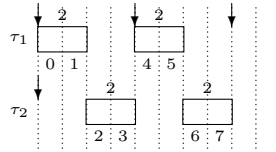
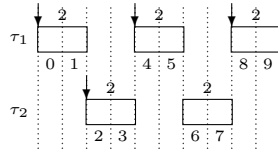


Figure 3.19: Phasing configuration for $w = 4$.

P_{i-1} . Remark also that the time unit $P_{i-1} - 1$ is an idle time unit since from Lemma 3.18 the interval $[P_{i-1} - 1, P_{i-1})$ minimizes the CPU usage among all interval of length 1 and the task set $\{\tau_1, \dots, \tau_{i-1}\}$ leaves at least one idle time unit in the interval $[0, P_{i-1})$. Consequently τ_i executes at times t_1 and $P_{i-1} - 1$ as exhibited in Figure 3.18.

Having identified the configuration which leads to the minimal response time, we are now interested by the computation of this value. According to Theorem 3.19 and the remarks above, ρ_i^* is the smallest value w such that if a request of τ_i occurs at time $\lceil \frac{C_i}{k_{i-1}} \rceil \cdot P_{i-1} - w$, or $\lceil \frac{w}{P_{i-1}} \rceil \cdot P_i - w$, the response time of the corresponding request is equal to w ; the possible values of w are in the interval $[C_i, \min\{\lceil \frac{C_i}{k_{i-1}} \rceil P_{i-1}, D_i\}]$. The computation of this response time is equivalent to compute ρ_i^1 by choosing $O_1 = O_2 = \dots = O_{i-1} = 0$ and $O_i = \lceil \frac{w}{P_{i-1}} \rceil P_{i-1} - w$, by an iterative computation issued from Equation (3.2) and (3.3), until $\rho_i^1 = w$ and this does not hold for a smaller w ; then $\rho_i^1 = \rho_i^*$. We have investigated the possibility to define an iterative process for the computation of ρ_i^* , similar to the one used for the computation of ρ_i^k ; such an iterative process seems difficult to define: the problem is illustrated with the next example.

Figure 3.20: Best response time configuration of τ_2 .

Example 3.20 An iterative process for the computation of ρ_i^* could be the following:

$$w_0 = C_i,$$

$$w_k = \rho_i^1 \text{ with } O_1 = \dots = O_{i-1} = 0 \text{ and } O_i = \lceil \frac{w_{k-1}}{P_{i-1}} \rceil P_{i-1} - w_{k-1}$$

But this iterative process (besides the fact that it is not necessarily monotonic) does not always converge to ρ_i^* . Consider the task set $\{\tau_1 = \{T_1 = D_1 = 4, C_1 = 2\}, \tau_2 = \{T_2 = D_2 = 6, C_2 = 4\}\}$; it occurs that $w_0 = C_2 = 4$, $w_1 = 8 = w_2$ (see Figure 3.19), while $\rho_2^* = 6 < w_2$ as exhibited in Figure 3.20. ■

We suggest to compute ρ_i^* by dichotomy:

LowerBound = C_i ;

UpperBound = $\min\{\lceil \frac{C_i}{k_{i-1}} \rceil P_{i-1}, D_i\}$;

Continue = true;

While (LowerBound \leq UpperBound and Continue) Do

$w = \lfloor \frac{\text{LowerBound} + \text{UpperBound}}{2} \rfloor$;

Compute $\rho = \rho_i^1$, with all the intermediate values as specified in Theorem 3.3, when $O_1 = \dots = O_{i-1} = 0$ and $O_i = \lceil \frac{w}{P_{i-1}} \rceil P_{i-1} - w$;

If ($\rho = w$) Then

If $\pi_z = 0$ and $j < i \Rightarrow R_j^{\tilde{m}_j} > O_i$ Then

Continue = false;

{ w is the smallest solution}

Else { w is a solution, but not the smallest}

UpperBound = $w - \pi_z - \sum_{1 \leq j \leq i-1, R_j^{\tilde{m}_j} \leq O_i + \pi_z} C_j$;

{since in the interval $[O_i, O_i + \pi_z + \sum_{1 \leq j \leq i-1, R_j^{\tilde{m}_j} \leq O_i + \pi_z} C_j$ the CPU is permanently used by higher priority requests than δ_i^1 }

Endif

Else-If ($\rho < w$) Then

UpperBound = $w - \max\{1, \pi_z + \sum_{1 \leq j \leq i-1, R_j^{m_j} \leq O_i + \pi_z} C_j\}$;
 {for the same reason as above}

Else

LowerBound = $w + 1 + \max(\{0\} \cup \{O_i - R_j^{m_j} | 1 \leq j \leq i-1, R_j^{m_j} < O_i, R_j^{m_j} + \rho_j^{m_j} \geq O_i\})$;
 {since in the interval $[R_j^{m_j}, O_j)$ ($j < i$) the CPU is permanently used by higher priority requests than δ_i^k if $R_j^{m_j} < O_i$ and $R_j^{m_j} + \rho_j^{m_j} \geq O_i$ }

EndIf

Od.

{If Continue then $\rho_i^* > D_i$ and the system is always unschedulable}

The maximal number of attempts is close to $\log(\min\{\lceil \frac{C_i}{k_{i-1}} \rceil P_{i-1}, D_i\} - C_i)$ and consequently the maximal time complexity of the computation of ρ_i^* is close to $\log(\min\{\lceil \frac{C_i}{k_{i-1}} \rceil P_{i-1}, D_i\} - C_i) \cdot \sum_{r=1}^{i-1} \left\lceil \frac{\sum_{j=r}^{i-1} T_j}{T_r} \right\rceil \cdot \beta_r$ (we consider here general deadline systems, hence method 3 can be applied).

3.6 Feasibility tests for general deadline systems

We shall present feasibility tests based on our response time computation. Indeed, we can check the feasibility of a system by considering the response time of all requests which occur in the feasibility interval.

3.6.1 Synchronous case

For synchronous systems with general deadlines we have seen that we have only to check the first request with the optimal rule of deadline monotonic assignment (see Corollary 2.14 and 2.26).

Theorem 3.21 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic synchronous task set with general deadlines. The system S is feasible with a static scheduler iff $r_i^1 \leq D_i$ ($\forall i = 1, \dots, n$) for the deadline monotonic priority assignment.*

Proof. Immediately follows from Theorem 3.13 and the fact that the deadline monotonic is optimal for synchronous and general deadline systems (see Theorem 2.25). ■

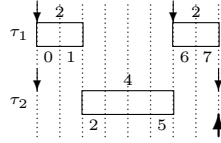


Figure 3.21: τ_1 and τ_2 are started at the same time: τ_2 misses its first deadline.

The feasibility test consists in computing n response times r_i^1 ($i = 1, \dots, n$) hence the maximal time complexity of the feasibility test is $\sum_{i=1}^n \beta_i$ if we compute each response time independently. Based on the fact that $r_j^1 \geq r_{j-1}^1 + C_j$, the computation can be improved however; indeed if we compute these response times in sequence $(r_1^1, r_2^1, \dots, r_n^1)$ the iterative process for the computation of r_j^1 ($j > 1$) can be speeded up by the initialization $w_0 = r_{j-1}^1 + C_j$.

3.6.2 Asynchronous case

For asynchronous systems, the computation of r_i^1 gives only a sufficient condition.

Corollary 3.22 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic asynchronous task set with general deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The system S is feasible if $r_i^1 \leq D_i \forall i = 1, \dots, n$.*

Proof. Immediately follows from Theorem 3.13. ■

The reciprocal is false since from a schedulability point of view the synchronous case is worst than any asynchronous cases. A task set can be unschedulable in the synchronous case while being schedulable if we consider different task start times for each task.

Example 3.23 Consider two tasks τ_1 and τ_2 with $T_1 = D_1 = 6$, $C_1 = 2$ and $T_2 = D_2 = 8$, $C_2 = 5$. The priorities of tasks τ_1 and τ_2 are given by the rate monotonic scheduler, hence τ_1 has a higher priority than τ_2 ($\tau_1 > \tau_2$). If all tasks are started at the same time, this task set is not schedulable (see Figure 3.21). But, the task set is schedulable with $O_1 = 1$ and $O_2 = 0$ (see Figure 3.22). ■

We have shown that $[X_1, S_n + P)$ is a feasibility interval for asynchronous and general deadline systems, and for the task sub-set $\{\tau_1, \dots, \tau_i\}$ the schedule

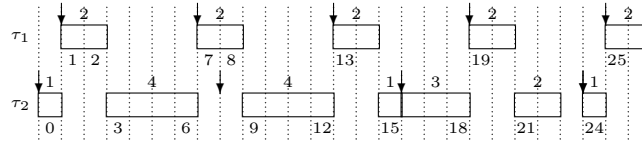


Figure 3.22: The task set is schedulable with $O_1 = 1$ and $O_2 = 0$; from $t = 24$, the schedule repeats.

repeats from S_i with a period of P_i . Hence, a feasibility test can be formulated as follows:

Theorem 3.24 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic asynchronous task set with general deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The system is schedulable iff $\rho_i^k \leq D_i \forall i = 1, \dots, n$ and for all k such that $S_i \leq R_i^k < S_i + P_i$.*

Proof. Immediately follows from Theorem 2.48. ■

Note the difference between Theorem 3.21, Corollary 3.22 and Theorem 3.24; in Theorem 3.21 we have a necessary and sufficient condition for the feasibility of a synchronous system, while with Corollary 3.22 and Theorem 3.24 we have conditions for the schedulability of asynchronous systems and for a particular priority assignment. This is due to the fact that, for asynchronous systems, the monotonic priority assignment is not optimal.

The maximal time complexity increases considerably in this case (in comparison with the synchronous case), since we have to consider at most $\sum_{i=1}^n \frac{P_i}{T_i}$ response times. This number may grow exponentially with the number of tasks (see [Mac98] for experimental results). Note that in this case the interest of the sufficient condition, i.e., to consider the synchronous case (and then the sufficient condition given by Corollary 3.22) first, since this condition requires to compute the utilization factor and only n response times, if $1 \geq \sum_{i=1}^n \frac{C_i}{T_i} > n(\sqrt[2]{2} - 1)$. If the sufficient condition is not satisfied, we have to check the response times given by Theorem 3.24. Remark however that the feasibility of the system is checked previously (off-line), possibly on a more powerful (and parallel) system than the one used for the task set; for this reason the computations may be acceptable. Remark also that if the system is not schedulable it could be interesting to consider other priority assignments, since in this case monotonic priority assignments are not optimal.

3.7 Response time of the k^{th} request in asynchronous and arbitrary deadline systems

We shall here extend the computation of the response time to arbitrary deadline and asynchronous systems. Recall that the formulas of Tindell et al. (introduced in section 3.2) remain valid for the 1st request of task τ_i in the synchronous case with arbitrary deadlines. But the formulas given in section 3.3 have to be modified. In this case at time R_i^k , we have to consider several requests of τ_j ($j \leq i$), since several requests of the same task may be active at time R_i^k , including previous requests of τ_i . Hence, the generalization is far from obvious, and the formulas have to be adapted to handle this more general case.

Theorem 3.25 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic asynchronous task set with arbitrary deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The response time of the k^{th} request of task τ_i , ρ_i^k , is the smallest solution of the equation:*

$$\rho_i^k = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j$$

where

$$\begin{aligned} R_j^p &= O_j + (p-1)T_j \\ m_j &= \begin{cases} \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil & \text{if } O_j < R_i^k \\ 1 & \text{otherwise} \end{cases} \\ \pi_j &= \begin{cases} 0 & \text{if } j = 0 \text{ or } O_j \geq R_i^k \text{ (or } R_j^{m_j} + D_j \leq R_i^k, \\ & \text{if we assume that all higher priority} \\ & \text{requests met their deadlines)} \\ (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{otherwise} \end{cases} \\ z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall j \leq i \\ p : \pi_p = \max\{\pi_j \mid \pi_j > 0 \text{ and } j \leq i\} & \text{otherwise} \end{cases} \\ \tilde{m}_j &= \begin{cases} \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1 & \text{if } j > z \\ m_z + 1 & \text{if } j = z \\ \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil + 1 & \text{otherwise} \end{cases} \\ x^+ &= \max\{x, 0\} \end{aligned}$$

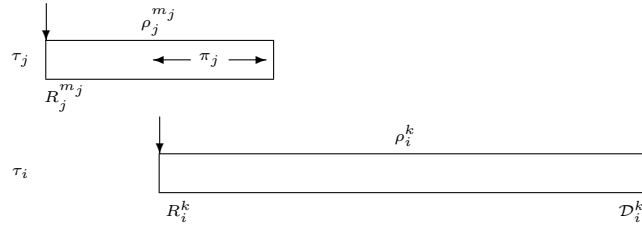
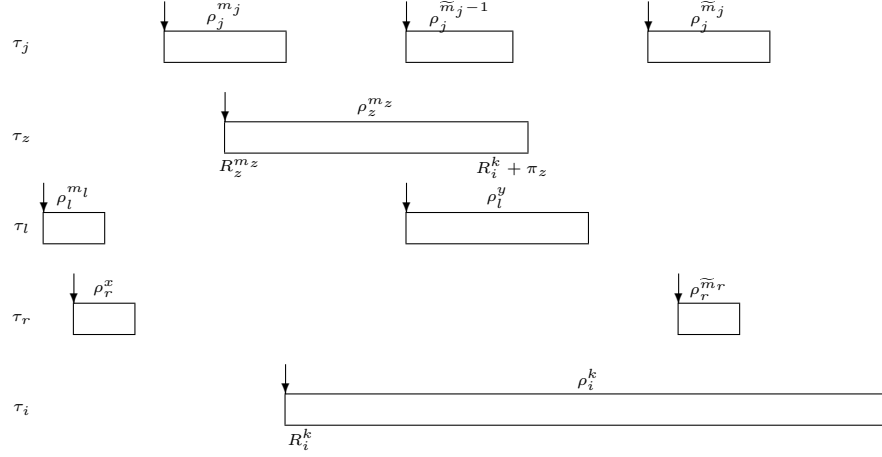


Figure 3.23: Interference of requests which occur strictly before time R_i^k .

Proof. We may assume that all higher priority requests (than the k^{th} of τ_i) which occur before time $R_i^k + \rho_i^k$ meet their deadlines, but this is not essential here: we may consider the extended schedules with soft deadlines introduced in section 2.4. R_j^p denotes the arrival time of the p^{th} request for τ_j . The interference from higher priority requests can be computed from the response time of some higher priority requests (so that the formula may be recursive). For a task τ_j (with $1 \leq j \leq i$), we only have to consider the requests from the m_j^{th} , the last one that precedes strictly R_i^k (see Figure 3.23) if any, (i.e., if $O_j < R_i^k$), otherwise we take $m_j = 1$ and consider the requests from the first one.

Indeed, the interference of ρ_j^r ($r < m_j$, $j \leq i$), if any, is included in $\rho_j^{m_j}$ since $R_j^r < R_j^{m_j} < R_i^k$ and we use the FIFO discipline between requests of a same task. It is easy to see that $m_j = \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil$ if $O_j < R_i^k$; otherwise $m_j = 1$, hence the formula above. The k^{th} request of τ_i is delayed by a part (or all) of the m_j^{th} request of τ_j if $O_j < R_i^k$ (then we also have $R_j^{m_j} < R_i^k$) and the response time $\rho_j^{m_j}$ is greater than $R_i^k - R_j^{m_j}$; if we assume that all higher priority requests which occur before time R_i^k meet their deadlines, we may save the computation of $\rho_j^{m_j}$ if $R_j^{m_j} + D_j \leq R_i^k$, since if $\rho_j^{m_j} > R_i^k - R_j^{m_j}$ that would mean that the corresponding deadline was missed. The interference is equal to $\pi_j = R_j^{m_j} + \rho_j^{m_j} - R_i^k$. Remark that, in the arbitrary deadline situation, the previous request of task τ_i (if any, i.e., if $k \geq 2$) may also have an interference on ρ_i^k ; it follows that the term π_i must be considered (in this case, we have $m_i = k - 1$). Suppose that two such requests of higher priority (say $\tau_a > \tau_b$) have an interference in the response time of the k^{th} request of τ_i ; in that case the interference of the higher priority one is included in the interference of the lower one: since at time R_i^k both requests are active, the request of τ_b ends its execution after the request of τ_a . Hence, $\pi_b > \pi_a > 0$ and the total interference of all requests which precede strictly the k^{th} request of τ_i is equal to $\pi_z = \max\{\pi_j | \pi_j > 0 \text{ and } j \leq i\}$ ($z = 0 = \pi_z$ if no request occurring strictly before time R_i^k delays the k^{th} request of τ_i).

Figure 3.24: Interference of requests which occur after (or at) time R_i^k .

Let us consider now the interference of requests which occur after or at time R_i^k . For each task τ_j ($j = 1, \dots, i-1$, since next requests of τ_i have a lower priority than the k^{th}) we shall consider the requests from the \tilde{m}_j^{th} , i.e., the first one which is not included in the term π_z . We have to distinguish three kinds of tasks: (i) the tasks up to τ_{z-1} ($\tau_1, \tau_2, \dots, \tau_{z-1}$), (ii) task τ_z and (iii) the tasks with a priority between τ_{z+1} and τ_{i-1} ($\tau_{z+1}, \dots, \tau_{i-1}$).

- (i) In this case, all the requests of τ_j which occurred strictly before time $R_i^k + \pi_z = R_z^{m_z} + \rho_z^{m_z}$ were already considered in the term π_z ; hence we have to consider the requests from the number $\left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil + 1$ (this is the case of the request number \tilde{m}_j in Figure 3.24); notice that if $O_j \geq R_i^k + \pi_z$, we consider the requests from the first one.
- (ii) The first request of τ_z which is not in the term π_z has the rank $\tilde{m}_z = m_z + 1$.
- (iii) In this case we have necessarily that $\pi_j = 0$ (otherwise z would not be the maximal index such that $\pi_z > 0$) and we have to consider the requests from the first request of τ_j which occurs after or at time R_i^k : $\tilde{m}_j = \min\{r | R_j^r \geq R_i^k\} = \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1$.

The term $\sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j$ denotes the interference of requests of τ_j ($j < i$) from the one of rank \tilde{m}_j in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$.

Hence, we have shown that the interference of higher priority tasks than τ_i in the interval $[R_i^k, R_i^k + \rho_i^k)$, plus the previous requests of τ_i , is

$$I(\rho_i^k) = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j.$$

The k^{th} request of τ_i ends its computation at the first instant $R_i^k + \rho_i^k$ such that the equality is satisfied: $\rho_i^k = I(\rho_i^k)$. ■

It may be noticed that the synchronous case is a (rather interesting) special case of Theorem 3.25.

Corollary 3.26 *Let $\tau_1, \tau_2, \dots, \tau_n$ be a synchronous periodic task set with arbitrary deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. In this special case, ρ_i^k is the smallest solution of the equation:*

$$\rho_i^k = C_i + \pi_z + \sum_{j=1}^{i-1} \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil C_j$$

where

$$\begin{aligned} R_j^p &= (p-1)T_j \\ m_j &= \begin{cases} \left\lceil \frac{R_i^k}{T_j} \right\rceil & \text{if } k > 1 \\ 1 & \text{otherwise} \end{cases} \\ \pi_j &= \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 1 \text{ (or } R_j^{m_j} + D_j \leq R_i^k \\ & \text{if we assume that all higher priority} \\ & \text{requests met their deadlines)} \\ (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{otherwise} \end{cases} \\ z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall j \leq i \\ p : \pi_p = \max\{\pi_j | \pi_j > 0 \text{ and } j \leq i\} & \text{otherwise} \end{cases} \\ \tilde{m}_j &= \begin{cases} \left\lceil \frac{R_i^k}{T_j} \right\rceil + 1 & \text{if } j > z \\ m_z + 1 & \text{if } j = z \\ \left\lceil \frac{R_i^k + \pi_z}{T_j} \right\rceil + 1 & \text{otherwise} \end{cases} \\ x^+ &= \max\{x, 0\} \end{aligned}$$

Proof. From Theorem 3.25, since $O_j = 0$, $R_j^p = (p-1)T_j$, and $x^+ = x$ when $x \geq 0$. ■

3.7.1 Computation of ρ_i^k

In the same way than in section 3.3.1, we shall present several methods for the computation of ρ_i^k . Again, the computation can be divided in two parts: the computation of the term π_z (and the related numbers: $\rho_j^{m_j}, z, \tilde{m}_j$) and the computation of the lowest solution of the equation:

$$\rho_i^k = C_i + \pi_z + \sum_{j=1}^{i-1} \left[\frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right] C_j.$$

The second part of the computation can be resolved by an iterative process similar to the one exhibited in section 3.3.1.

$$\begin{aligned} w_0 &= C_i + \pi_z, \\ w_{k+1} &= C_i + \pi_z + \sum_{j=1}^{i-1} \left[\frac{(R_i^k + w_k - R_j^{\tilde{m}_j})^+}{T_j} \right] C_j. \end{aligned}$$

Theorem 3.6 remains valid in this case and the maximal number of iterations is bounded by $\frac{D_i - C_i - \pi_z}{\min_{j < i} C_j}$. For similar reasons than those considered in section 3.3.1 this bound is very pessimistic. It may also be noticed that, in comparison with general deadline systems:

- the term π_z may represent the interference of several previous requests of τ_j , including τ_i ; as a consequence, π_z will generally be larger here than in the general deadline situation, which speeds up the iterative process;
- the value $D_i - C_i$, in general, is larger in this case, which slows down the iterative process.

Experimental results (similar to the ones exhibited in section 3.3.1) show that the actual number of iterations represents 1 % of the bound given by Theorem 3.6; hence again, the bound is very pessimistic and the iterative process converges much more quickly.

We consider now the computation of the term π_z . Method 1 defined for general deadline systems can be adapted here, but the maximal time complexity and in particular the total number of calls is (significantly) larger. We shall not estimate this number as exactly as exhibited in Theorem 3.7, since the time complexity of $f(i, k)$ depends more deeply on the parameter k and the T'_i s. Let

$\mathcal{H}_i(k)$ be the maximal time complexity of the function $f(i, k)$ for the computation of ρ_i^k in arbitrary deadline situation. $\mathcal{H}_i(k)$ is approximatively equal to $\beta_i + \mathcal{H}_i(k-1) + \sum_{j < i} (\beta_j + \mathcal{H}_j(\frac{k \cdot T_i}{T_j}))$. It follows that $\mathcal{H}_i(k) \geq \mathcal{G}_i$ since we have also to consider the previous request of τ_i . In conclusion the total number of calls is (significantly) larger. The maximal depth of recursive calls (and consequently the maximal space complexity of the method) is also larger in this situation. In the recursion we shall distinguish between two kinds of calls: for the same task, i.e., $f(j, k)$ calls the function $f(j, k-1)$ (type 1) and for higher priority tasks, i.e., $f(j, k)$ calls the function $f(r, k')$ with $r < j$ and $R_r^{k'} < R_j^k$ (type 2). The recursion stops with the computation of ρ_j^1 with $O_j = \min_{k \leq i} O_k$. In this case the depth of recursive calls is bounded by $\left\lceil \frac{R_i^k - O_j}{\min\{T_k | k \leq i\}} \right\rceil + i - j$; the first term (i.e., $\left\lceil \frac{R_i^k - O_j}{\min\{T_k | k \leq i\}} \right\rceil$) represents the maximal number of type 1 calls and the second term (i.e., $i - j$) represents the maximal number of type 2 calls. The worst case is $O_1 = 0$ and $T_1 = \min\{T_k | k \leq i\}$: in this case the maximal depth of recursive calls is bounded by $\left\lceil \frac{R_i^k}{T_1} \right\rceil + i - 1$.

Method 1 is not suited to handle “real size” problems; the interest of this method (and its implementation) lies (mainly) in the “verification” of our formulas.

Method 2 can easily be adapted in this case, the only difference in comparison with the general deadline situation is that we need also to compute all the response times of τ_i till time R_i^k ; at each instant we have only to know the response time of the last request of τ_i , if any. Hence the maximal time complexity is $\sum_{j=1}^i \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot \beta_j$, or simply β_i if we consider that the previous response time computations were due anyway for a feasibility test. The maximal space complexity is $O(i)$. If the schedule is feasible, for R_i^k sufficiently large (i.e., from $R_i^k > O^{max} + 2P_i$), according to Lemma 2.64, the schedule (and then the response times) repeats from time $O^{max} + P_i$; hence in the worst case we have to compute $\sum_{j=1}^i \left\lceil \frac{O^{max} - O_j + P_j}{T_j} \right\rceil$ response times to reach the periodic part of the schedule and $\sum_{j=1}^i \frac{P_j}{T_j}$ to compute the response times $\rho_j^{k_j}$ which are equal to the $\rho_j^{m_j}$'s but occur in the first period of the periodic behavior of the schedule (i.e., $k_j = \min\{k | k = m_j \bmod \frac{P_j}{T_j} \text{ and } R_j^k \geq S_j\}$). As a consequence, the maximal time complexity of the computation of ρ_i^k is $\sum_{j=1}^{i-1} \left\lceil \frac{O^{max} - O_j + 2P_j}{T_j} \right\rceil \cdot \beta_j$.

Method 3 cannot be applied with efficiency here; the computation of the term π_z needs the value of the response times of $\delta_1^{m_1}, \dots, \delta_i^{m_i}$. It is possible however that such a request (say $\delta_j^{m_j}$) does not have an explicit impact on ρ_i^k , but answering this question requires the value of π_j , in other words the value of $\rho_j^{m_j}$, unless $R_j^{m_j} \geq R_i^k$. In arbitrary deadline situation, method 3 essentially

amounts to method 2. Note that if we use the response time computation to check the schedulability of a system, we have anyway (in the “worst” case, i.e., if the system is schedulable) to compute all the response times in the feasibility interval, as developed in the section 3.8.3.

3.8 Schedulability tests for arbitrary deadline systems

Based on our response time computation we shall present schedulability tests for arbitrary and asynchronous systems. Again we can check the feasibility of a system by considering the response time of all requests which occur in the feasibility interval.

3.8.1 The worst case response time for arbitrary systems

We shall in this section give a “quantitative approach” of Theorem 2.37.

Another proof of Theorem 2.37 Let $[a, b)$ be a level- i busy period in some synchronous or asynchronous system built from $\{\tau_1, \dots, \tau_i\}$, and let us denote $a + \Delta_j$ the starting time of the first request of τ_j after a ($\Delta_j \geq 0$ and $1 \leq j \leq i$); from definitions above we have that $\Delta_i < b - a$ and $\Delta_k = 0$ for at least one k . Suppose first that $\Delta_i > 0$. Only tasks having a higher priority than τ_i are processed during $[a, a + \Delta_i)$; consequently, if Δ_i were changed to any value in $[0, \Delta_i)$, each request of task τ_i in $[a, b)$ would finish at the same time as before (each request of τ_i occurs strictly before than in the original situation and uses the same free slots left by the task subset $\{\tau_1, \dots, \tau_{i-1}\}$ since there was no idle time unit left by τ_i), hence increasing each of their response time: the maximum response time occurs when $\Delta_i = 0$. Moreover, no idle point occurs before b since it is still true that each request of τ_i in the interval, but the first one, starts strictly before the previous one is completed (see Lemma 2.36); it could even happen that b is no longer an idle point, if the first request of τ_i which occurred after or at b now starts strictly before b : the level- i busy period is lengthened. If $\Delta_j > 0$ ($j < i$), then reducing Δ_j leads to increase (or is left unchanged) the processing requirement $r_j(t)$ of τ_j during $[0, t)$ for every $t \in (0, b]$, where $r_j(t) = \left\lceil \frac{(t - \Delta_j - a)^+}{T_j} \right\rceil C_j$. Notice that if Δ_j is decreased it follows that $r_j(t)$ increases (or is left unchanged). Now it may be seen that the first request of τ_i after a finishes (if ever) at the first time instant t such that $(t - a) = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{(t - \Delta_j - a)^+}{T_j} \right\rceil C_j$, and more

generally the k^{th} request of τ_i in the interval $[a, b)$ finishes at the first instant t such that $(t - a) = kC_i + \sum_{j=1}^{i-1} \left\lceil \frac{(t - \Delta_j - a)^+}{T_j} \right\rceil C_j$: if $r_j(t)$ increases, this will delay accordingly each request of τ_i ; this may also enlarge the level- i busy period, and possibly blend it with the next one(s). Hence, the largest response times, and the largest level- i busy period, are achieved by setting each Δ_j to its smallest value: $\Delta_1 = \Delta_2 = \dots = \Delta_i = 0$. This configuration corresponds to the first level- i busy period in the synchronous case. ■

3.8.2 Synchronous systems

For synchronous systems with arbitrary deadlines, we have seen that $[0, \lambda_n)$ is a feasibility interval (see Theorem 2.37).

Theorem 3.27 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic synchronous task set with arbitrary deadlines. The system is schedulable for the static priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$ iff $\rho_i^k \leq D_i \forall i = 1, \dots, n$ and for all k such that $R_i^k < \lambda_n$.*

Proof. Immediately follows from Theorem 2.37. ■

The maximal time complexity increases considerably in this case, in comparison with synchronous and general deadline systems, since we have to consider a number of response times proportional to P (λ_n is bounded by P but may reach it if $\sum_{i=1}^n \frac{C_i}{T_i} = 1$).

For asynchronous systems, the computation of the response time in the interval $[0, \lambda_n)$ gives only a sufficient condition.

Corollary 3.28 *Let $\tau_1, \tau_2, \dots, \tau_n$ a periodic asynchronous task set with arbitrary deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The system is schedulable if the corresponding synchronous system is schedulable, i.e., if $\rho_i^k \leq D_i < \lambda_n \forall i = 1, \dots, n$ and for all k such that $R_i^k < \lambda_n$ (ρ_i^k and λ_n are computed in the synchronous case).*

Proof. Immediately follows from Theorem 2.37. ■

Remark that this is only a sufficient condition: a system can be unschedulable in the synchronous case while being schedulable if we consider different task start times for each task.

3.8.3 Asynchronous systems

We have shown that $[0, O^{\text{max}} + 2P)$ (see Theorem 2.67) is a feasibility interval for asynchronous and arbitrary deadline systems if $U \leq 1$. Hence, a feasibility

test can be formulated as follows.

Theorem 3.29 *Let $\{\tau_1, \tau_2, \dots, \tau_n\}$ be a periodic asynchronous task set with arbitrary deadlines and a static priority assignment: $\tau_1 > \tau_2 > \dots > \tau_n$. The system is schedulable iff $\rho_i^k \leq D_i \forall i = 1, \dots, n$ and for all k such that $R_i^k < O^{max} + 2P$ and $U \leq 1$.*

Proof. Immediately follows from Theorem 2.67. ■

The maximal time complexity of the test is comparable to the one for general deadline asynchronous systems. The situation is different with respect to the synchronous case where for general deadline systems the maximal time complexity is pseudo-polynomial and increases considerably for arbitrary deadline case, since we have to consider the interval $[0, \lambda_n)$, the maximum value of λ_n being P (if $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$).

3.9 Comparison on the various feasibility tests

During our study we have given a comparison of the time complexity of the various sub-classes of periodic task sets. We shall in this section summarize these comparisons.

First let us consider the sufficient conditions, where according Theorem 3.13 for late/general deadlines and Theorem 3.13 for arbitrary deadlines, synchronous and asynchronous systems can be amalgamated.

| late deadline | general deadline | arbitrary deadline |
|---|------------------|--|
| $\sum_{i=1}^n \frac{C_i}{T_i} < n(\sqrt[n]{2} - 1)$ | $r_i^1 \leq D_i$ | $\rho_i^k \leq D_i, R_i^k < \lambda_n$ |

We consider now sufficient and necessary conditions; here, synchronous and asynchronous cases must be distinguished.

Let us consider first synchronous systems.

| late deadline | general deadline | arbitrary deadline |
|------------------|------------------|---|
| $r_i^1 \leq D_i$ | $r_i^1 \leq D_i$ | $\rho_i^k \leq D_i,$ $R_i^k < \lambda_n$ |

Then we consider asynchronous systems.

| late deadline | general deadline | arbitrary deadline |
|--|--|---|
| $\rho_i^k \leq D_i,$ $R_i^k \in [X_1, S_n + P)$ | $\rho_i^k \leq D_i,$ $R_i^k \in [X_1, S_n + P)$ | $\rho_i^k \leq D_i,$ $R_i^k \in [0, O^{max} + 2P)$ |

3.10 Conclusion

In this chapter we have studied the response time notion for static schedulers. We have extended the computation (and consequently the theory) to handle more general and optimistic cases than those considered in the literature, including asynchronous systems with arbitrary deadlines. We have shown the interest of considering these response time computations regarding the feasibility problems of these more general systems. We have also considered the problem of the computation of these response times. For the various kinds of task sets considered in this work, we have proposed several methods for these computations. We have studied the analytical and experimental (time and space) complexity of our algorithms. In particular for asynchronous and general deadline we propose a method which computes the response time of the k^{th} request of τ_i with a (pseudo-)polynomial time complexity and a linear space complexity in terms of the system characteristics. For arbitrary deadline system we have shown that the situation is less attractive and the computation grows exponentially with the system characteristics. The study of the response time computation in asynchronous situation has also provided the material to prove the property “stated” by Liu and Layland concerning the worst case response time and to refine the proof of Lehoczky concerning the first busy period. Hence, we feel to have justified the interest of our general response time computation.

Interesting questions for further research related to response time computation include: statistical analysis of the actual performance of the various methods and algorithms proposed in this chapter with other random variables or with “real” systems; analysis of the parallelization of method 1 and the study of its time/space complexity; the computation of the best case response time would be improved by considering an iterative process rather than a dichotomy.

Bibliography

- [ABRW92] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Welling. Deadline monotonic scheduling theory. In Boullard and Puente, editors, *Proc. IFAC/IFIP WRTP'92*, pages 55–60, Bruges, Belgium, 1992.
- [Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, University of York, England, 1991.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, a guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
- [JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.
- [Mac98] Christophe Macq. Etude pratique des principaux algorithmes d'ordonnancement de tâches apériodiques en présence de tâches périodiques dans un système temps réel. Master's thesis, Université Libre de Bruxelles, Belgique, 1998.
- [Tin93] K. W. Tindell. Using offset information to analysis static priority pre-emptively scheduled task sets. Technical report, University of York, England, 1993.

Chapter 4

Dynamic Schedulers

*Une accumulation de faits n'est pas plus une science
qu'un tas de pierres n'est une maison.*
— Henri Poincaré, *La Science et l'hypothèse (Flammarion)*.

Contents

| | | |
|------------|--|------------|
| 4.1 | Introduction | 110 |
| 4.2 | Simplified model of computation | 111 |
| 4.3 | The deadline driven scheduler | 111 |
| 4.4 | Optimality | 114 |
| 4.5 | Feasibility intervals | 124 |
| 4.5.1 | Synchronous systems | 138 |
| 4.6 | Response times | 139 |
| 4.6.1 | Introduction | 139 |
| 4.6.2 | 1 st request for synchronous systems | 140 |
| 4.6.3 | k^{th} request for asynchronous general deadlines | 146 |
| 4.6.4 | Computation of ρ_i^k | 150 |
| 4.6.5 | k^{th} request for asynchronous arbitrary deadlines | 153 |
| 4.6.6 | Computation of ρ_i^k | 156 |
| 4.7 | Feasibility tests for asynchronous systems | 161 |
| 4.8 | Feasibility tests for synchronous systems | 164 |

| | | |
|-------------|---|------------|
| 4.8.1 | Feasibility of bounded general deadline synchronous task sets | 164 |
| 4.8.2 | Worst case response time computation | 170 |
| 4.9 | The Least Laxity First scheduling algorithm . . . | 184 |
| 4.10 | The (non-)stability of dynamic priority rules . . . | 190 |
| 4.11 | Conclusion | 190 |
| | Bibliography | 192 |

4.1 Introduction

We shall consider in this chapter the second family of scheduling algorithms, i.e., the dynamic schedulers. In this case, the scheduling algorithm computes the priorities during the execution of the system. The priority of each active request depends on the system state (e.g., the current time), and on the request characteristics (e.g., the remaining processing time, the time before reaching the deadline, etc.). Unlike static schedulers, the priority of a task or request may change with time. We consider again the scheduling of periodic and independent task sets for preemptive and mono-processor systems.

In this chapter we shall mainly study the most popular optimal dynamic scheduling algorithm: the deadline driven scheduler. We shall first present basic properties, in particular the optimality of this priority rule. We shall then extend our previous results to the dynamic case (e.g., feasibility intervals, response time computation, etc.). We shall outline in this study the differences between static and dynamic scheduling algorithms regarding the feasibility problem and the response time computation.

The remainder of the chapter is as follows: in section 4.3 we present the deadline driven scheduler; in section 4.4 we show the optimality of this scheduling rule and we complete the results given by Liu and Layland; in section 4.5 we study feasibility intervals, and we extend the results to the arbitrary deadline case; in section 4.6 we extend the notion of the response time to the dynamic deadline driven scheduler and we exploit these computations in sections 4.7 and 4.8 for the feasibility test of the various kinds of periodic task sets considered in this work; in section 4.9 we consider another optimal dynamic scheduling rule and we extend major properties to it; in section 4.10 we consider the stability of dynamic priority rules. But first, we shall refine our model of computation.

4.2 Simplified model of computation

For this family of scheduling algorithms, we can restrict the schedule, i.e., the function $\sigma(t)$, to value changes at natural time instants: $\sigma : \mathbb{N} \rightarrow \mathbb{N}$. Consequently, $\sigma(t)$ is an integer function and $\sigma(t) = j$, with $j > 0$, means that a request of task τ_j is executing at time t during one time unit, while $\sigma(t) = 0$ means that the CPU is idle at time t (during one time unit). Notice that, in the static case, the same property holds from the very definition of static schedulers (changes may only occur at natural time instants –see page 20). Here we could allow to change priorities, hence the executing request, at any time; however, it may be shown that this would not add true benefits to the framework considered here. We shall not give the proof of this property now: we need more material on dynamic scheduling algorithms. For this reason, we shall assume the property here (and use its consequence) and we shall show it later, in section 4.4. In the arbitrary deadline case (or in the late/general deadline case with soft deadlines), where many requests of the same task may be active simultaneously, we shall assume that the oldest active request of τ_j receives the CPU; if another policy is desired, we shall then extend the notation to $\sigma(t) = (j, k)$ to represent the fact that at time t the k^{th} request of τ_j is executing.

4.3 The deadline driven scheduler

We shall present here the more popular dynamic (and optimal) scheduler: the deadline driven scheduler. The names *earliest deadline scheduler* (EDS) and *earliest deadline first* (EDF) also occur in the recent literature to denote this dynamic priority algorithm based on the nearest deadline. This algorithm was basically defined by Liu and Layland [LL73] and termed the *deadline driven scheduling algorithm*; it is a dynamic scheduling algorithm which gives (at any instant) the highest priority (and then the CPU) to the active request with the nearest deadline. The tie, if any, may be broken in an arbitrary way. We shall use in this work the original term (i.e., deadline driven scheduler) to denote this scheduler.

Example 4.1 Consider the following system composed of two tasks τ_1 and τ_2 with $\tau_1 = \{C_1 = 1, D_1 = 3, T_1 = 4, O_1 = 0\}$ and $\tau_2 = \{C_2 = 2, D_2 = 3, T_2 = 5, O_2 = 0\}$. Figure 4.1 corresponds to the schedule of this set using the deadline driven scheduler. From this schedule several remarks can be done.

□ At time $t = 0$ two requests are active and their deadlines coincide (at

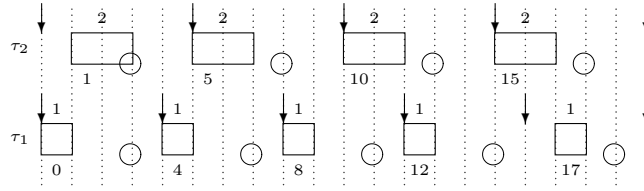


Figure 4.1: Schedule with the deadline driven scheduler.

time $t = 3$): the tie is broken here by giving a higher priority (and then the CPU) to the request of τ_1 .

- The priority of the various requests of the same task changes with time: for instance at time $t = 0$ the first request of τ_1 has a higher priority than the first request of τ_2 , while at time $t = 16$, the fifth request of τ_1 has a lower priority than the (end of the) fourth request of task τ_2 .
- At time $t = 20$ the schedule repeats.

■

We shall first exhibit a very general property instrumental to prove that with the deadline driven scheduler the tie (if any) can be broken arbitrarily; first we consider the following notations (inspired from [BHR93]): $\eta_i(t, t')$ denotes, for $t \leq t'$, the number of values $k \in \mathbb{N}$ such that

1. $t \leq O_i + k \cdot T_i$, and
2. $O_i + k \cdot T_i + D_i \leq t'$.

That is, $\eta_i(t, t')$ is the number of requests of task τ_i which occur in the interval $[t, t')$ with a deadline less than or equal to t' . Any feasible scheduling algorithm must give at least $\eta_i(t, t') \cdot C_i$ CPU time units to τ_i in this interval; this is a necessary condition for the schedulability of the system in this interval.

Lemma 4.2 *An asynchronous and arbitrary deadline system R is feasible up to time t with a deadline driven scheduler iff $\sum_{i=1}^n \eta_i(t', t'') \cdot C_i \leq t'' - t'$ for all $0 \leq t' < t'' \leq t$.*

Inspired from [BHR93] The condition is clearly necessary. Suppose the condition holds, and R is not feasible up to time t using a deadline driven scheduler. Let $t'' (\leq t)$ be the time of the (first) deadline failure. Let t' be the

last time before t'' such that at time $t' - 1$ either the system is idle or a request with a deadline strictly greater than t'' is scheduled. Since t'' corresponds to a deadline, $t'' > 0$; so t' is well-defined (it may be 0). Furthermore, since the deadline at t'' is not met, there is an active task (with deadline t'') scheduled at $t'' - 1$ and $t' < t''$. It follows that there is a task scheduled at every time in $[t', t'')$ with its deadline not later than time t'' . Since no task having a deadline less than or equal to t'' is scheduled at $t' - 1$, every task scheduled in $[t', t'')$ must have been released not earlier than t' . Since there is a task scheduled at every time in $[t', t'')$ and the deadline at t'' is not met, $\sum_{i=1}^n \eta_i(t', t'') \cdot C_i > t'' - t'$: a contradiction. ■

Corollary 4.3 *An asynchronous and arbitrary deadline system R is feasible iff $\sum_{i=1}^n \eta_i(t, t') \cdot C_i \leq t' - t$ for all $0 \leq t < t'$.* ■

The way to resolve the ties has an impact on the response time of some requests, and we shall fix a strategy when we come to this point (in section 4.6), but as far as schedulability is concerned, we have a very general property which shows that the way chosen to resolve ties has no impact on the feasibility.

Corollary 4.4 *Let S be an asynchronous and arbitrary deadlines system. If S is not schedulable with the deadline driven scheduler for a particular way of resolving the ties, and a deadline failure occurs at time t , this is also the case for all other ways of resolving ambiguities.*

Proof. Immediately follows from Lemma 4.2. ■

Corollary 4.5 *Let S be an asynchronous and arbitrary deadlines system. If S is schedulable with the deadline driven scheduler for a particular way to resolve the ambiguities, i.e., the ties, this is also the case for all other ways of resolving ambiguities.*

Proof. Immediately follows from Lemma 4.2, or from Corollary 4.4. ■

Remark that this property does not hold for the rate/deadline monotonic priority assignment as exhibited with Example 2.27, and the example used in the proof of Lemma 2.41.

When there is no tie (i.e., when all requests have different deadlines, or when requests with a same deadline are never active simultaneously), the deadline driven scheduler amounts to assigning different priorities to task requests (instead of tasks, like in the static case): a request δ_i^k has a higher priority than δ_j^r if its deadline occurs earlier. If there is a tie, in principle, the choice of

the running request may be changed at any (integer) instant; however, from Corollary 4.5, from a schedulability point of view, we may decide to choose a simpler way of proceeding: in particular we may, for the uniformity of the procedure, assign (more or less) arbitrarily different priorities to requests with the same deadline (the relative priority of a request with respect to other ones with the same deadline will not change with time). In the following we shall adopt this way of applying the deadline driven scheduler.

Notice also that, even in the arbitrary deadline situation, there will never be a tie between requests of a same task (contrary to what happened in the static case, where we had to introduce an extra FIFO rule to lift ambiguities): they always have different priorities.

4.4 Optimality

Liu and Layland have exhibited several major properties concerning this scheduling algorithm; their main result concerns the optimality of this scheduler. The results given by Liu and Layland are based on the fact that the synchronous case is the worst case from a schedulability point of view. But this property is not fully proved in their paper; more precisely the authors use an incorrect argument to justify a preliminary property.

Liu and Layland have only considered in their work [LL73] the optimality of the deadline driven scheduler in the late deadline case for synchronous systems. First, let us define what we mean by optimality. As usual we have to distinguish between two kinds of optimality:

Definition 4.6 A dynamic priority rule is *strongly optimal* for a family of task sets if, when a feasible dynamic priority assignment exists for some task set of the family, any schedule given by the rule is also feasible for that task set, whatever the way in which the ambiguities are resolved. ■

Definition 4.7 A dynamic priority rule is *weakly optimal* for a family of task sets if, when a feasible dynamic priority assignment exists for some task set of the family, some schedule given by the rule is also feasible for that task set, for a particular way of resolving the ambiguities. ■

From Corollary 4.5, it occurs that strong and weak optimality collapse for the deadline driven scheduler: if it is weakly optimal (and it is, see Theorem 4.16) it is also strongly optimal.

Liu and Layland have shown that a task set is schedulable iff $U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ in the late deadline (and synchronous) case. They first state a property of

the schedules produced by the deadline driven scheduler prior to an *overflow*, i.e., just before a deadline failure. We have use the term “state” because the property is true (see Theorem 4.8 and our proof) but Liu and Layland have used an incorrect reasoning to show this property.

Theorem 4.8 ([LL73]) *When the deadline driven scheduler is used to schedule a synchronous set of tasks with late deadlines on a processor, there is no processor idle time prior to an overflow.*

Proof from Liu and Layland [LL73]. Consider some schedule given by the deadline driven scheduler when starting all tasks at the same instant $t = 0$. We suppose there is an overflow at time t_3 , a processor idle time in the interval $[t_1, t_2)$ and no processor idle time in the interval $[t_2, t_3)$ with⁵ $0 \leq t_1 < t_2 \leq t_3$ (see Figure 4.2). Let us consider in this schedule only the requests which occur after the idle period (i.e., from time t_2): we have in this case an asynchronous⁶ system where all tasks are started after or at time t_2 . From this asynchronous system, we shall construct a synchronous system (where all tasks are started at time t_2) with an overflow and no processor idle time prior to it. In this asynchronous system, $O_1 = t_2 + \delta_1$ is the instant of the first request of task τ_1 ; since there is no processor idle time in the interval $[t_2, t_3)$, this will also be the case in the asynchronous system based on the previous one but where τ_1 starts at time t_2 . Indeed, in the interval $[t_2, t_3)$ the number of the τ_1 's requests increases and the overflow occurs either at or before time t_3 . Repeating the same argument for all tasks, we construct a schedule for the deadline driven scheduler where all tasks are started at time t_2 without processor idle time prior the overflow. This leads to a contradiction, since from⁷ Corollary 4.4 the schedule from t_2 should be equivalent to the one from 0, with an overflow at or before $t_3 - t_2$. ■

With the following example we show that the argument of Liu and Layland is incorrect. They state that, from an asynchronous situation with a deadline failure in t_3 and no idle time before, if τ_1 starts at time t_2 instead of time $t_2 + \delta_1$ an overflow occurs at or before time t_3 . This is not the case in the following system:

⁵It is easy to see that $t_1 > 0$ (initially the first request of each task is active) and $t_2 < t_3$, since in $t_3 - 1$ there was an active request (the one missing its deadline, or each one if there are many of them).

⁶If the situation were to be synchronous, from Corollary 4.4, the schedule from t_2 would be equivalent to the one from 0, and the overflow in the latter would occur at least in $t_3 - t_2$.

⁷To tell the truth, Liu and Layland did not explicitly used this property but implicitly assumed it.

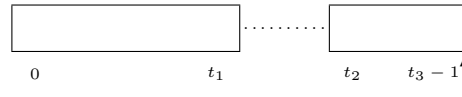
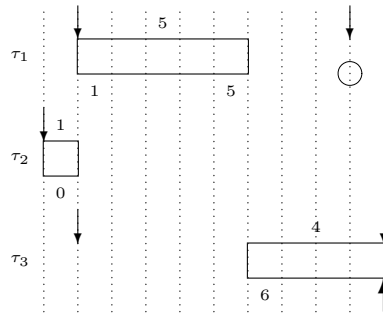
Figure 4.2: CPU utilization before the overflow at time t_3 .

Figure 4.3: Overflow at time 10.

Example 4.9 Consider the following system:

| | C_i | $T_i = D_i$ | O_i |
|----------|-------|-------------|-------|
| τ_1 | 5 | 8 | 1 |
| τ_2 | 1 | 100 | 0 |
| τ_3 | 5 | 9 | 1 |

At time $t = 10$, τ_3 misses its deadline and an overflow occurs (see Figure 4.3). If we start τ_1 at time $t = 0$ (i.e., if $O_1 = 0$), there is no overflow before time $t = 19$ (see Figure 4.4). This example contradicts the main argument in the proof of Liu and Layland. ■

Again, we see that we must be very careful: incorrect reasonings can be constructed from a wrong intuition, and even in very “simple” cases (e.g., regarding the number of tasks) it is not obvious at all to have a good idea on the exact behavior of our systems.

Here we complete and correct the proof.

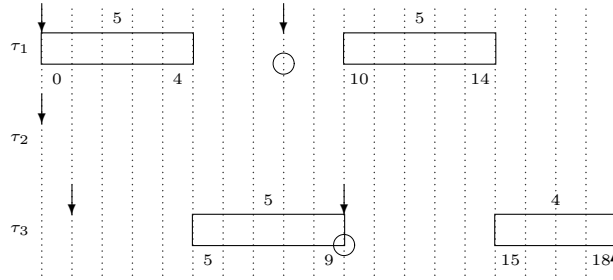


Figure 4.4: No overflow before time 19.

Proof of Theorem 4.8. We show the theorem by contradiction and we consider some schedule given by the deadline driven scheduler when starting all tasks at the same time $t = 0$. We suppose there is an overflow at time t_3 , a processor idle interval $[t_1, t_2)$ and no processor idle time in the interval $[t_2, t_3)$ with⁵ $0 < t_1 < t_2 < t_3$. At time t_3 (and not before) a request (at least one) misses its deadline, hence a deadline corresponds with time t_3 , and at time $t_3 - 1$ the CPU was busy for a request with deadline at time t_3 . Consider now time t_4 defined as follows: t_4 is the smallest instant less than t_3 such that in the interval $[t_4, t_3)$ the CPU remains busy for requests with deadline less than or equal to t_3 . From the previous remark, it is not difficult to see that such an instant exists and since during the interval $[t_1, t_2)$ the CPU is idle: $t_2 \leq t_4 < t_3$. At $t_4 - 1$, either all tasks are inactive, or all active requests have a deadline after (strictly) t_3 , so that they have no impact on the requests occurring between t_4 and t_3 . Let us consider in this schedule only the requests which occur after or at time t_4 : we have an asynchronous system where all tasks are started after or at time t_4 . From this asynchronous system, we shall construct a synchronous system (where all tasks are started at time t_4) with an overflow and no processor idle time prior it. In this asynchronous system, $O_1 = t_4 + \delta_1$ is the instant of the first request of task τ_1 ; since in the interval $[t_4, t_3)$ the processor remains busy for requests whose deadlines occur before or at time t_3 , this is also the case if we start τ_1 at time t_4 and the number of such requests is greater than or equal to the one in the previous situation. More precisely, $\sum_{i=1}^n k_i \cdot C_i > t_3 - t_4$ (where k_i represents the number of requests of τ_i occurring in the interval $[t_4, t_3)$ with a deadline less than or equal to t_3) if $O_1 = t_1 + \delta_1$ and hence also if $O_1 = t_1$: for this reason an overflow occurs before or at time t_3 in both situations. From a schedulability point of view, the situation is worst and in this new asynchronous system an overflow occurs before or at time t_3 . Repeating the same argument for all tasks, we have a system where all tasks are started at time t_4 without processor idle time before the overflow which occurs at time t_3 (whatever the way chosen to resolve the

ties since $\sum_{i=1}^n k_i \cdot C_i > t_3 - t_4$). This leads to a contradiction, since there is a way to resolve the ties such that the new schedule from t_4 should be the same as the old one from 0. ■

Remark that Liu and Layland have only considered the case of late deadline systems, but the proof of Theorem 4.8 does not rely on this assumption and holds for arbitrary deadline systems. Moreover, the property can be extended by considering idle points (cf Definition 2.30) instead of idle intervals.

It may be also noticed that Example 4.9 does not contradict our argument since in this case $t_4 = 1, \delta_1 = 0, \delta_2 = 100, \delta_3 = 0$, and an overflow still occurs at time $t_3 = 10$, even if τ_2 starts at time $t_4 = 1$ (i.e., the synchronous system is constructed by shifting requests left to time 1, instead of time 0 like in Example 4.9).

We shall see in section 4.5.1 the interest of considering this more general property based on idle points instead of idle intervals.

Theorem 4.10 *When the deadline driven scheduler is used to schedule a synchronous set of tasks with arbitrary deadlines on a processor, there is no processor idle point prior to an overflow but the origin.*

Proof. We show the theorem by contradiction and we consider some schedule given by the deadline driven scheduler when starting all tasks at the same time $t = 0$. We suppose there is an overflow at time t_3 , and a processor idle point in the interval $(0, t_3]$. Let t_2 be the last idle point before time t_3 , with⁸ $0 < t_2 < t_3$. We may drop all requests occurring before t_2 without modifying the schedule from time t_2 . At time t_3 (and not before) a request (at least one) misses its deadline, hence a deadline corresponds with time t_3 , and at time $t_3 - 1$ the CPU was busy for a request with deadline at time t_3 . Consider now time t_4 defined as follows: t_4 is the smallest instant less than t_3 such that in the interval $[t_4, t_3)$ the CPU remains busy for requests with deadline less than or equal to t_3 . From the previous remark, it is not difficult to see that such an instant exists and since time t_2 is an idle point: $t_2 \leq t_4 < t_3$. At $t_4 - 1$, either all tasks are inactive, or all active requests (which have not been dropped) have a deadline after (strictly) t_3 , so that they have no impact on the requests occurring between t_4 and t_3 . Let us consider in this schedule only requests which occur after or at time t_4 : we have an asynchronous system where all tasks are started after or at time t_4 . From this asynchronous system, we shall construct a synchronous system (where all tasks are started at time t_4) with an overflow and no processor idle point before it. Hence, we consider an asynchronous system where each task τ_i

⁸ $t_2 < t_3$, since in t_3 there is an active request (the one missing its deadline) which is not terminated.

starts its execution at time $O_i = t_4 + \delta_i$. The first request of τ_1 occurs at time $O_1 + \delta_1$; since in the interval $[t_4, t_3)$ the processor remains busy for requests whose deadline occurs before or at time t_3 , this is also the case if we start τ_1 at time t_4 and the number of such requests is greater than or equal to the one in the previous situation. More precisely, $\sum_{i=1}^n k_i \cdot C_i > t_3 - t_4$ (where k_i represents the number of request of τ_i occurring in the interval $[t_4, t_3)$ with a deadline less than or equal to t_3) if $O_1 = t_1 + \delta_1$ as well as if $O_1 = t_1$ and for this reason an overflow occurs before or at time t_3 in both situations. From a schedulability point of view, the situation is worst and in this new asynchronous system an overflow occurs before or at time t_3 . We have also to show that if $O_1 = t_4 + \delta_1$ as well as $O_1 = t_4$ there are no processor idle point in $[t_4, t_3)$. We know that if $O_1 = t_4 + \delta_1$, the interval $[t_4, t_3)$ has no processor idle point; it follows that: $t \in [t_4, t_3) \Rightarrow \sum_{i=1}^n \left\lceil \frac{(t-t_4-\delta_i)^+}{T_i} \right\rceil C_i > t - t_4$. Since $\left\lceil \frac{t-t_4}{T_1} \right\rceil \geq \left\lceil \frac{(t-t_4-\delta_1)^+}{T_1} \right\rceil$, it follows that if we start τ_1 at time t_4 the interval $[t_4, t_3)$ does not contain any processor idle point either. Repeating the same argument for all tasks, we construct a schedule for the deadline driven scheduler where all tasks are started at time t_4 without processor idle point before the overflow before or at t_3 (whatever the way chosen to resolve the ties since $\sum_{i=1}^n k_i \cdot C_i > t_3 - t_4$). This leads to a contradiction, since there is a way to resolve the ties such that the new schedule from t_4 should be the same as the old one from 0. ■

Liu and Layland have not explicitly proved that the synchronous case is the worst case from a schedulability point of view for late deadline systems. They have assumed that this property follows from Theorem 4.8. This main property can be proved in a similar way than for Theorem 4.8: we shall give this proof for completeness, and we shall also by the way handle the more general case of arbitrary deadline systems.

Theorem 4.11 *Let $S = \{\tau_i = \{C_i, D_i, T_i\} | i = 1, \dots, n\}$ with arbitrary deadlines. If S is schedulable in the synchronous case using the deadline driven scheduler this is also the case in all asynchronous situations.*

Proof. Let us first recall that from Corollary 4.5 the way ties are resolved has no impact on the schedulability of a task set by the deadline driven scheduler. We show the theorem by contradiction and we assume that the system S is unschedulable with the offset assignment O_1, \dots, O_n ($\min\{O_i | i = 1, \dots, n\} = 0$) and schedulable in the synchronous case. Since the asynchronous system is not schedulable, it follows that a request misses its deadline (say at time t_1). Hence a deadline corresponds with time t_1 , and at time $t_1 - 1$ the CPU was used by a request with its deadline in t_1 . Consider now time t_2 defined as follow: t_2 is the smallest instant less than t_1 such that in the interval $[t_2, t_1)$ the CPU remains busy for requests with deadline less than or equal to t_1 . From the previous

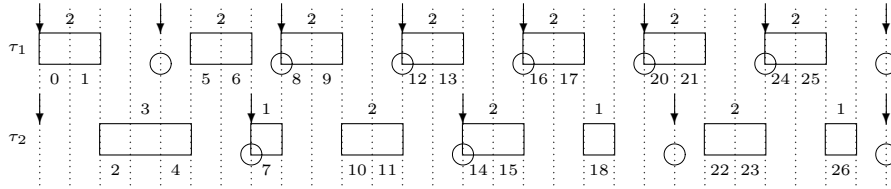


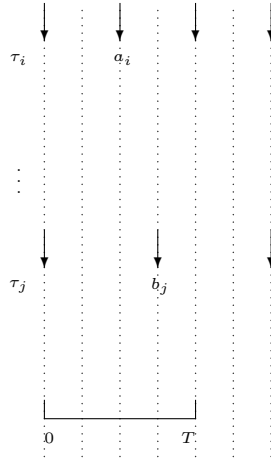
Figure 4.5: The system is schedulable, from time $t = 28$ the schedule repeats.

remark, it is not difficult to see that such an instant exists: $0 \leq t_2 < t_1$. Let us consider in this schedule only requests which occur after or at time t_2 : we have an asynchronous system where all tasks are started after or at time t_2 with the same schedule from t_2 to t_1 as before. From this asynchronous system, we shall construct a synchronous system (where all tasks are started at time t_2) with an overflow. In this asynchronous system $O_1 = t_2 + \delta_1$ is the instant of the first request of task τ_1 ; since in the interval $[t_2, t_1)$ the processor remains busy for requests whose deadlines occur before or at time t_1 , this is also the case if we start τ_1 at time t_2 and the numbers of such requests is greater than or equal to the one in the previous situation. More precisely, $\sum_{i=1}^n k_i \cdot C_i > t_1 - t_2$ (where k_i represents the number of request of τ_i occurring in the interval $[t_2, t_1)$ with a deadline less than or equal to t_1) if $O_1 = \delta_1$ as well if $O_1 = 0$ and for this reason an overflow occurs before or at time t_1 in both situations. From a schedulability point of view the situation is worst and in this new asynchronous system an overflow occurs before or at time t_1 . Repeating the same argument for all tasks, we construct a schedule for the deadline driven scheduler where all tasks are started at time t_2 , without processor idle time, with an overflow. This leads to a contradiction, since the new schedule from 0 is a deadline driven schedule, for which there is no overflow. ■

Notice that Theorem 4.11 expresses only the fact that for the dynamic deadline driven scheduler and from a schedulability point of view the synchronous case is the worst case. But for the dynamic deadline driven scheduler the response time of the first request of τ_i is not necessarily the worse, even for late deadline systems as illustrated in the following example.

Example 4.12 Consider the system $S = \{\tau_1 = \{C_1 = 2, D_1 = T_1 = 4\}, \{\tau_2 = \{C_2 = 3, D_2 = T_2 = 7\}\}$, the response time of the first requests of τ_2 in the synchronous case are respectively: 5, 5, 5, 6 (see Figure 4.5). ■

Theorem 4.8 is instrumental in proving the optimality of the deadline driven scheduler in the late deadline case; the following proof is based on Liu and Layland's one; we have however lifted some lacks of precision.

Figure 4.6: Overflow at time T .

Theorem 4.13 *For a given set of n synchronous tasks with late deadlines, the deadline driven scheduling algorithm is feasible (whatever the way to resolve the ties) iff $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$.*

Proof inspired from [LL73] The necessity is trivial: a task set is certainly not schedulable if the (long term) demand exceeds the available processor time (i.e., if $\sum_{i=1}^n \frac{C_i}{T_i} > 1$). To show the sufficiency, we assume that $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ and yet the scheduling is not feasible: there is an overflow at time T ; by Theorem 4.8 there is no idle time in $(0, T]$. For each task τ_i , let us consider the last request occurring strictly before time T ; among them, we may distinguish two kinds of requests (see Figure 4.6): those with a deadline at T (occurring at a_i , $i \in I_1$) and those with a deadline beyond T (occurring at b_j , $j \in I_2$). Notice that $I_1 \cap I_2 = \emptyset$, $I_1 \cup I_2 = \{1, \dots, n\}$, and $I_1 \neq \emptyset$ since the request(s) missing its deadline at T belongs to I_1 . We shall consider two cases:

Case 1. None of the requests at b_j for τ_j ($j \in I_2$) has received any CPU unit before T . In this case the scheduler had to serve $\sum_{i=1}^n \left\lfloor \frac{T}{T_i} \right\rfloor C_i$ CPU time units in the interval $[0, T)$ since $\left\lfloor \frac{T}{T_i} \right\rfloor$ is the number of periods of τ_i completely included in $[0, T]$, the tasks in I_1 have an exact number of periods in this interval, and the ones in I_2 did not use the CPU for their last incomplete period; since there is no processor idle time before the overflow at T , we have $\sum_{i=1}^n \left\lfloor \frac{T}{T_i} \right\rfloor C_i > T$, hence $\sum_{i=1}^n \frac{C_i}{T_i} \geq \sum_{i=1}^n \left\lfloor \frac{T}{T_i} \right\rfloor \frac{C_i}{T} > 1$.

Case 2. Some of the computations requested at b_j for τ_j ($j \in I_2$) have received

CPU units before T . We define T' as the time instant terminating the last time unit in the interval $[0, T)$ used by a request at b_j for τ_j ($j \in I_2$): $T' < T$ since at $T - 1$ the processor was used by a request with deadline T . In the interval $[T', T)$ the scheduler served only requests of task τ_i with $i \in I_1$, and the requests of τ_i ($i \in I_1$) occurring strictly before T' are completed at T' , since they have a higher priority than the requests of τ_j at b_j ($j \in I_2$). Since an overflow occurs at T , this demand exceeds the available CPU: $\sum_{i \in I_1} \left\lfloor \frac{T-T'}{T_i} \right\rfloor C_i > T - T'$ (the formula follows from the fact that for each task $\tau_i \in I_1$ a request corresponds with T and we only have to consider periods entirely in the interval). Since $\sum_{i=1, \dots, n} \left\lfloor \frac{T-T'}{T_i} \right\rfloor C_i > \sum_{i \in I_1} \left\lfloor \frac{T-T'}{T_i} \right\rfloor C_i$, we have that $\sum_{i=1}^n \frac{C_i}{T_i} > 1$ for the same reason as before. ■

Corollary 4.14 *For a given set of n asynchronous tasks with late deadline, the deadline driven scheduler algorithm is feasible (whatever the way to resolve the ties) iff $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$.*

Proof. By Theorem 4.13 we have that if $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ the system is schedulable in the synchronous case and by Theorem 4.11 this is also the case for all asynchronous systems. ■

It may be noticed that in the case of the deadline driven scheduler, the condition $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ is necessary and sufficient for the schedulability (of late deadline systems), while for the static case it is only a necessary condition. It follows immediately from Theorem 4.13 that the deadline driven scheduler is strongly optimal in the late deadline case.

Corollary 4.15 *The deadline driven scheduler is strongly optimal for asynchronous and late deadline systems.*

Proof. Immediately follows from Corollary 4.14. ■

We see that the strong optimality of the dynamic deadline driven scheduler remains for asynchronous and late deadline systems. We shall also see in this chapter that the strong optimality of the dynamic deadline driven scheduler persists for asynchronous and general or arbitrary deadline systems, contrary to what happened with the static deadline monotonic scheduler.

Liu and Layland have not considered the optimality in more general cases, where the deadlines may be less or greater than the periods, nor the asynchronous case. On the other hand Labetoulle has considered the case of asynchronous and general deadline systems for the *relative urgency* algorithm (RU) [Lab74a, Lab74b]. The RU algorithm is a scheduling algorithm defined for

systems where the task characteristics are real numbers (i.e., $T_i, D_i, C_i, O_i \in \mathbb{R}$) and the CPU allocation for the time instant t ($t \in \mathbb{R}$) is a set of functions $\{\mu_i(t) \in \mathbb{R} \mid 1 \leq i \leq n\}$ which associate a real number $\mu_i(t)$ at time t for each task τ_i , with the constraint $0 \leq \sum_{i=1}^n \mu_i(t) \leq 1, \forall t$, where $\mu_i(t)$ is the fraction of CPU instantaneously allocated to the task τ_i at the instant t . The relative urgency algorithm is defined as follows: at each instant of time ($t \in \mathbb{R}$) the processor is allocated to the (one of) active task whose deadline is closest. We shall not give more details here since the results of Labetoulle concern only asynchronous and general deadline systems in a different computational model than ours.

Dertouzos [Der74] has considered the optimality of the deadline driven scheduler in our computational model for asynchronous and general deadline systems; we shall extend this result to handle asynchronous arbitrary deadline systems and to show the strong optimality of this rule. It may be noticed that the strong optimality of the deadline driven scheduler for asynchronous and arbitrary deadline systems follows from Lemma 4.2, we shall however present an explicit proof which is instrumental to justify our simplified model of computation.

Theorem 4.16 *The deadline driven scheduler is strongly optimal for asynchronous and arbitrary deadline systems.*

Proof. Let us assume the existence of a scheduling algorithm A so that the system is schedulable with it. Let σ be the feasible schedule produced by the scheduler A on the system. We shall show that any schedule σ' produced by the deadline driven scheduler is also feasible. We shall show the following property by induction on t : we can always transform the schedule σ in such a way that in the interval $[0, t)$ the resulting schedule σ^t is the same as σ' , and σ^t remains feasible for the same task set. This is trivially true for $t = 0$. Suppose the hypothesis holds for t . If $\sigma^t(t) = \sigma'(t)$, we may take $\sigma^{t+1} = \sigma^t$. If $\sigma^t(t) = 0 \neq \sigma'(t) = (i, k)$, let e be the first instant after t such that $\sigma^t(e) = (i, k)$ (this instant exists since δ_i^k is started but not completed at time t in σ^t as in σ'): we may take $\sigma^{t+1}(t) = (i, k), \sigma^{t+1}(e) = 0$ and $\sigma^{t+1} = \sigma^t$ elsewhere; σ^{t+1} fulfills the requested condition. It is not possible to have $\sigma^t(t) \neq 0 = \sigma'(t)$ since $\sigma'(t) = 0$ means then there is no active request at t in σ' as in σ^t (the deadline driven scheduler is expedient). Suppose that a request of task τ_i (say δ_i^k) is executing at time t in σ^t (i.e., $\sigma^t(t) = (i, k)$) while there is another active request of task τ_j (say the request⁹ $\delta_j^{k'}$) with a nearest (or the same) deadline, executing in σ' (i.e., $\sigma'(t) = (j, k')$). Let d_i and d_j be the deadlines

⁹Remark that for arbitrary deadline systems we can have $j = i$, but then $k' < k$).

of the requests δ_i^k and $\delta_j^{k'}$, respectively; by definition we have: $d_j \leq d_i$. Notice that the request $\delta_j^{k'}$ must be executing at least one time unit before time unit d_j , and consequently before time unit d_i , in σ^t . We can execute the request $\delta_j^{k'}$ at time t (i.e., $\sigma^{t+1}(t) = (j, k')$) and the request of δ_i^k at time e_j (i.e., $\sigma^{t+1}(e_j) = (i, k)$), where e_j is the first time unit after time t in σ^t assigned to the request $\delta_j^{k'}$ (this is a valid definition, since $\delta_j^{k'}$ is still active at time t in σ^{t+1} , and $e_j < d_j \leq d_i$). The resulting schedule ($\sigma^{t+1} = \sigma^t$ elsewhere) remains feasible and in the interval $[0, t + 1)$ the schedule is the one produced by σ' . Consequently, the feasible schedule σ can be transformed into σ' , while remaining feasible, and σ' itself is feasible. ■

Notice that the optimality of the deadline driven scheduler holds even if we consider schedules defined at time instants with a finer granularity than the task characteristics (e.g., if the schedule changes occur at instants multiple of $\frac{1}{m}$): in the proof of Theorem 4.16 we can replace the induction step of 1 by an induction step of $\frac{1}{m}$. Since the schedule produced by the deadline driven scheduler changes at natural instants multiple of $\text{gcd}\{O_i, T_i, C_i\}$ and since it remains optimal even if we consider schedules with finer granularities, it follows that (possibly through a time scaling) we can restrict our study to schedules with value changes at natural time instants, with $\text{gcd}\{O_i, T_i, C_i\} = 1$. This justifies the assumptions made in section 4.2 concerning our simplified model of computation.

Theorem 4.13 and Theorem 4.16 show the (strong) optimality of the deadline driven priority assignment, in the late deadline case for synchronous systems and in the arbitrary deadline case for asynchronous systems, respectively. It may be noticed that the optimality of the deadline driven scheduler holds for asynchronous systems, contrary to the static priority assignment case where the optimality of the rate/deadline monotonic priority assignment is only valid in the synchronous case (see Chapter 2).

Notice however that it is not said that the deadline driven scheduler is the only optimal dynamic priority rule: we shall see in section 4.9 that the least laxity first algorithm is another strongly optimal priority rule.

4.5 Feasibility intervals

We shall now consider the feasibility problem, i.e., how to decide if a system is feasible (or not). Since the deadline driven priority rule is optimal for the various classes of periodic task sets considered in this work, we can restrict this question by considering the schedulability of the system using the deadline driven priority rule. Leung and Merrill [LM80] have “shown” that the schedule

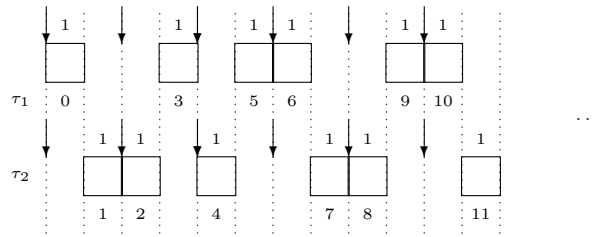


Figure 4.7: Non-periodic schedule using the deadline driven scheduler.

produced by the deadline driven scheduler is periodic and they have defined a feasibility interval based on this property. However, this property is only valid with an additional assumption concerning the way chosen to resolve the ambiguities, which is not considered in their work and which is illustrated by the following example.

Example 4.17 Consider the synchronous system $\tau_1 = \{T_1 = D_1 = 2, C_1 = 1, O_1 = 0\}, \tau_2 = \{T_2 = D_2 = 2, C_2 = 1, O_2 = 0\}$: if we make no assumption on the way chosen to resolve the ambiguities, it is possible to construct a non-periodic (feasible) schedule using the deadline driven scheduler, as exhibited in Figure 4.7; indeed, for any k , the k^{th} request of τ_1 and the k^{th} request of τ_2 have the same deadline, and the priority may be given to δ_2^k only when k is prime. ■

The problem arises from the fact that the way chosen here to resolve the ties is not periodic. We shall then correct the results of Leung and Merrill, and in particular the property which states that the schedule given by the deadline driven scheduler is periodic with a period of P . For this reason we shall add the (not too strong constraint) that the way chosen to resolve the ties is itself periodic with a period of P ; we could be a bit more general however, and assume that this period is a multiple of P (say $k \cdot P$) but this implies that the period of the schedule would also be $k \cdot P$, while under our assumption we shall show that the schedule has a period of P , as claimed by Leung and Merrill. This is the case for instance if the way chosen to resolve the tie at any time (say time t) depends only on the configuration of the requests at time t (i.e., on the quantities: $(t - O_1) \bmod T_1, \dots, (t - O_n) \bmod T_n$). More formally we shall assume to have *request-dependent* scheduling rules.

First let us introduce the following notation: $\delta_i^k(t) \succ \delta_j^h(t)$ which means that the request δ_i^k has a higher priority than the request δ_j^h at time t . Of course, a higher priority will only be effective if the two requests indeed compete, i.e., if there exists a time instant such that both requests are active.

Definition 4.18 A scheduling rule is said to be request-dependent if $\forall i, j, k, h$:

$$\delta_i^{k+h_i}(t+P) \succ \delta_j^{h+h_j}(t+P) \Leftrightarrow \delta_i^k(t) \succ \delta_j^h(t)$$

with $h_i = \frac{P}{T_i}$ and $h_j = \frac{P}{T_j}$. ■

For instance the deadline driven scheduler is request-dependent if we break the tie (i.e., when $R_i^k + D_i = R_j^h + D_j$) by giving a higher priority to the request with the smaller index (i.e., $\min\{i, j\}$). Remark that the previous definition consider also situations where the relative priority between two requests changes with time: $\delta_i^k(t) \succ \delta_j^h(t) \not\Rightarrow \delta_i^k(t+1) \succ \delta_j^h(t+1)$. This would occur for instance if we break the ties by giving the highest priority to the oldest active request of the first active task in the sequence $\tau_{(t \bmod n)+1}, \dots, \tau_{((t+n-1) \bmod n)+1}$. Remark that the period of the schedule of this deadline driven scheduler is $\text{lcm}\{P, n\}$.

The results of Leung and Merrill concern the general deadline case and do not remain valid for arbitrary deadlines, however. For this reason we shall first consider their original (but corrected) results and we shall then consider the extension to the arbitrary deadline case.

Some concepts were already defined for static scheduling algorithms; we shall however replicate some of them for convenience and uniformity of the presentation. Notice also that some definitions have to be refined in the dynamic case, in particular the functions $\sigma(t)$ and $\epsilon_i(t)$. Indeed, some of the results from the literature which we shall use and extend consider the evolution of the system after deadline failures. This means that we must define rigorously the behavior of the system whenever a request misses its deadline; remark that this is not always done in the literature, even when results on these extended schedules are presented (e.g., in [LM80, BRH90]). There are various ways to do so; we shall consider a first kind of behavior for general deadline systems, and later another one for arbitrary deadline ones (which may also be applied to general and late deadlines as a special case). We first redefine the scheduling rule (and consequently the function $\sigma(t)$) whenever a request (say δ_i^k) misses its deadline at time $(R_i^k + D_i)$ thus: the execution of the system continues with the rule given by deadline driven scheduler up to the next request, i.e., a request can miss its deadline and remains active till its completion or till the next request of it occurs (i.e., δ_i^{k+1}) at time R_i^{k+1} : at this instant the request δ_i^k is dropped and the request δ_i^{k+1} becomes active. For convenience, let us call this kind of schedule: *partially extended schedule*. From this new system behavior it follows that the function $\epsilon_i(t)$, which gives the amount of processor time used by the last request of τ_i in the interval $[0, t)$, is defined $\forall t \geq O_i$, even if a request of τ_i misses its deadline. Indeed, if the request δ_i^k misses its deadline, the function $\epsilon_i(t)$ remains defined for $t > R_i^k + D_i$, and $\epsilon_i(t)$ is growing till the next request

of τ_i occurs (at time R_i^{k+1}): at this instant $\epsilon_i(R_i^{k+1}) = 0$. Notice also that if a schedule is feasible, it is identical to its partial extension.

Leung and Merrill have “shown” that in the dynamic case the first part of a schedule may be neglected: we may consider only the schedule from its periodic part, i.e., in steady state situation. This result is the dynamic version of a result of Leung and Whitehead [LW82] for static schedulers (see Lemma 2.50). However the property has to be completed, by assuming a request-dependent scheduler.

Lemma 4.19 *For any request-dependent deadline driven partially extended scheduler applied to general deadline asynchronous periodic task sets, for each task τ_i and instant $t \geq O_i$, we have $\epsilon_i(t) \geq \epsilon_i(t + P)$ with $P = \text{lcm}\{T_i | i = 1, \dots, n\}$.*

Proof inspired from [LM80] We prove the lemma by contradiction and assume that there is some task τ_j and some time instant $t_1 \geq O_j$ such that $\epsilon_j(t_1) < \epsilon_j(t_1 + P)$. Then there must be some time $t'_1 \in [R_j^h, t_1)$, with $h = \lfloor \frac{t_1 - O_j}{T_j} \rfloor + 1$, such that τ_j is active at both t'_1 and $t'_2 = t'_1 + P$, and τ_j is executing at t'_2 but not at t'_1 . This can only occur if there is another task τ_i (say the k^{th} request of τ_i), which is active at t'_1 but not at t'_2 , such that $\delta_i^k(t'_1) \succ \delta_j^h(t'_1)$. But this means that $\epsilon_i(t'_2) < \epsilon_i(t'_2) = C_i$ since $\delta_i^{k+h_i}(t'_2) \succ \delta_j^{h+h_j}(t'_2)$ from the request-dependency assumption, and repeating the above argument, we have an infinite descending chain of time instants ($t_1 > t'_1 > t''_1 > \dots$) contradicting the fact that every task τ_i in the system has an initial request time O_i and the time is discrete in our model of computation. ■

Definition 4.20 We define the configuration of the partially extended schedule S at time t for the system R as

$$C_S(R, t) = ((\gamma_1(t), \epsilon_1(t)), \dots, (\gamma_n(t), \epsilon_n(t)))$$

where

- $\gamma_i(t) = (t - O_i) \bmod T_i$ is the time elapsed since the last request of τ_i , if $t \geq O_i$; $\gamma_i(t) = t - O_i$ if $t < O_i$.
- $\epsilon_i(t)$ gives the amount of processor time used by the last request of τ_i in the interval $[0, t)$, if $t \geq O_i$; $\epsilon_i(t) = 0$ if $t < O_i$.

■

In particular, $\gamma_i(0) = -O_i$ and $\epsilon_i(0) = \epsilon_i(O_i) = 0$.

Lemma 4.21 *The configuration of the partially extended schedule S at time $t+1$ is univocally determined by the configuration at time $t \geq 0$ for any request-dependent scheduler.*

Proof. Consider first the case of the quantities $\gamma_i(t+1)$:

If $\gamma_i(t) < 0$, then $\gamma_i(t+1) = \gamma_i(t) + 1$

otherwise $\gamma_i(t+1) = (\gamma_i(t) + 1) \bmod T_i$.

Before considering $\epsilon_i(t)$, let us first define, if $t \geq O_i$ (i.e., if $\gamma_i(t) \geq 0$), $\Omega_i(t) = \lfloor \frac{t-O_i}{T_i} \rfloor \bmod h_i + 1$; $\Omega_i(t)$ is the rank (modulo $h_i = \frac{P}{T_i}$) of the last request of τ_i before or at t ; from the request-dependency assumption, the actual rank of the request is not necessary to determine its priority: only the rank $\Omega_i(t)$ is needed; this quantity $\Omega_i(t)$ is univocally determined by the configuration of the partially extended schedule S at time t : if $\exists \gamma_j(t) < 0$, the current time and consequently $\Omega_i(t)$ can be determined (i.e., $t = O_j + \gamma_j(t)$); if $\gamma_j(t) > 0 \forall j$, the current time can be determined (modulo P) from the Generalized Chinese Remainder Theorem (and more particularly from its constructive proof, see Yih-hing [Dic19, Mat81] and Knuth [Knu69], pp. 513, for details) and the knowledge of the various $\gamma_j(t)$'s.

Consider now the quantities $\epsilon_i(t)$:

If $\gamma_i(t+1) = 0$, then $\epsilon_i(t+1) = 0$ (i.e., a new request arrives).

Else-if $\epsilon_i(t) < C_i$ and $\gamma_i(t) \geq 0$ (i.e., the last request of τ_i is active) and $\nexists j \neq i$ such that $\epsilon_j(t) < C_j$, $\gamma_j(t) \geq 0$ and $\delta_j^{\Omega_j(t)}(t) \succ \delta_i^{\Omega_i(t)}(t)$ (i.e., there is no higher priority active request than $\delta_i^{\Omega_i(t)}$), $\epsilon_i(t+1) = \epsilon_i(t) + 1$ (i.e., the request of τ_i runs at time t).

Otherwise $\epsilon_i(t+1) = \epsilon_i(t)$ (if τ_i is not executing at time t , $\epsilon_i(t+1)$ is unchanged). ■

Remark the improvement of the proof in comparison with the static situation (see Lemma 2.46): here we consider partially extended schedules which lead to some simplifications. It may also be noticed that deadline failures may be detected while observing the successive configurations, but this may be a bit complicated due to the partial extension: τ_i misses a deadline at time t if $\epsilon_i(t) < C_i$ and $\gamma_i(t) \geq D_i$, or if $\gamma_i(t) = 0$, $D_i = T_i$ and $\epsilon_i(t-1) < C_i - 1$ or if $\gamma_i(t) = 0$, $D_i = T_i$, $\epsilon_i(t-1) = C_i - 1$ and $\exists j \neq i$ with $\epsilon_j(t-1) < C_j$, $\gamma_j(t-1) \geq 0$ and $\delta_j^{\Omega_j(t-1)} \succ \delta_i^{\Omega_i(t-1)}$.

Lemma 4.22 *Let S be a feasible schedule constructed by some request-dependent deadline driven scheduler applied to a general deadline asynchronous periodic task set R . Then $C_S(R, t_1) = C_S(R, t_2)$, where $t_1 = O^{max} + P$ and $t_2 = O^{max} + 2P$.*

Proof inspired from [LM80] Suppose $C_S(R, t_1) \neq C_S(R, t_2)$. First remark that $\forall i, \gamma_i(t_1) = \gamma_i(t_1 + P)$. From Lemma 4.19, there must be a task τ_j for which $\epsilon_j(t_1) > \epsilon_j(t_2)$ and $\epsilon_i(t_1) \geq \epsilon_i(t_2)$ for $i = 1, \dots, n$. We first show that the schedule has no idle time slot in $[t_1, t_2)$. Suppose there is an idle slot at time $t = t_1 + \delta$, $0 \leq \delta < P$. This implies that no task is active at that time: $\forall i \in \{1, \dots, n\}$, $\epsilon_i(t_1 + \delta) = C_i$, its maximum value. By Lemma 4.19, we have that $C_S(R, O^{max} + \delta) = C_S(R, t_1 + \delta = O^{max} + \delta + P) = ((\gamma_1(t), C_1), \dots, (\gamma_n(t), C_n))$. From Lemma 4.21 it follows that the schedules in the intervals $[O^{max} + \delta, t_1 + \delta)$ and $[t_1 + \delta, t_2 + \delta)$ are identical. This means that $C_S(R, t_1) = C_S(R, t_2)$, a contradiction. Therefore, S has no empty time slot in $[t_1, t_1 + P)$. At time t_1 , there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n (C_q - \epsilon_q(t_1))$; at time $t_1 + P$, from Lemma 4.19 and the fact that $\epsilon_j(t_1) > \epsilon_j(t_2)$, the remaining demand: $\sum_{q=1}^n (C_q - \epsilon_q(t_1 + P))$ is strictly larger. But the additional demand in the interval $(t_1, t_1 + P]$ is $P \cdot U$, since in the interval $(t_1, t_1 + P]$ there are $\frac{P}{T_j}$ new requests of τ_j . In the interval $(t_1, t_1 + P]$ the CPU availability is P ; it follows that the additional demand is larger than P which implies that $U > 1$, a contradiction with the feasibility of the schedule. ■

Theorem 4.23 (inspired from [LM80]) *Let S be the schedule constructed by some request-dependent deadline driven and partially extended scheduler applied to a general deadline asynchronous periodic task set R . R is feasible iff (1) all deadlines in the interval $[0, O^{max} + 2P)$ are met and (2) $C_S(R, O^{max} + P) = C_S(R, O^{max} + 2P)$.*

Proof.

(only if part). If R is feasible on a single processor, then any schedule S constructed by the deadline driven scheduler must be a valid schedule. Hence, all deadlines in the interval $[0, t_2)$ are met in S , and by Lemma 4.22 we have $C_S(R, O^{max} + P) = C_S(R, O^{max} + 2P)$.

(if part). For each nonnegative integer j , let us define t_j as the time instant $O^{max} + j \cdot P$. For each $j \geq 2$, the requests made by all tasks in R in the interval $[t_j, t_{j+1}]$ must be identical to those made in the interval $[t_1, t_2]$. Since $C_S(R, t_1) = C_S(R, t_2)$, from Lemma 4.21, the schedule S repeats every P time units, starting from t_1 . Since all deadlines in the interval $[0, t_2)$ are met in S , the deadlines of all task computations must also be met in S . Hence, R is feasible on a single processor. ■

Theorem 4.23 uses two conditions; both of them are clearly necessary for the feasibility of the system. The following example shows that condition (1) does not imply condition (2).

Example 4.24 Consider the following system:

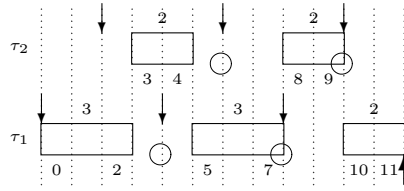


Figure 4.8: Both conditions of Theorem 4.23 are necessary.

| | T | D | C | O |
|----------|-----|-----|-----|-----|
| τ_1 | 4 | 4 | 3 | 0 |
| τ_2 | 4 | 4 | 2 | 2 |

We have $P = 4$, $O^{\max} = 2$, $t_1 = 6$ and $t_2 = 10$; moreover, as illustrated in Figure 4.8, all deadlines in interval $[0, 10)$ are met; but $C_S(R, 6) \neq C_S(R, 10)$, since $\epsilon_1(6) = 1$ and $\epsilon_1(10) = 0$, and the system is not feasible. ■

Baruah, Howell and Rosier [BRH90] have shown that, if $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, the first condition only needs to be considered; this can be shown with a reasoning similar to the one used in the proof of Lemma 4.22 (according to the new definitions of the functions $\sigma(t)$ and $\epsilon_i(t)$, i.e., for a partially extended schedule).

Lemma 4.25 *Let S be a partially extended schedule of a general deadline periodic task system R constructed by a request-dependent deadline driven scheduler. If $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, then $C_S(R, t_1) = C_S(R, t_2)$, where $t_1 = O^{\max} + P$ and $t_2 = t_1 + P$.*

Proof inspired from [BRH90] Suppose $C_S(R, t_1) \neq C_S(R, t_2)$. First remark that $\forall i, \gamma_i(t_1) = \gamma(t_1 + P)$. From Lemma 4.19, there must be a task τ_j for which $\epsilon_j(t_1) > \epsilon_j(t_2)$ and $\epsilon_i(t_1) \geq \epsilon_i(t_2)$ for $i = 1, \dots, n$. We first show that the schedule has no idle time slot in $[t_1, t_2)$. Suppose there is an idle slot at time $t = t_1 + \delta$, $0 \leq \delta < P$. This implies that no task is active at that time: $\forall i \in \{1, \dots, n\}$, $\epsilon_i(t_1 + \delta) = C_i$, its maximum value. By Lemma 4.19, we have that $C_S(R, O^{\max} + \delta) = C_S(R, t_1 + \delta = O^{\max} + \delta + P) = ((\gamma_1(t), C_1), \dots, (\gamma_n(t), C_n))$. From Lemma 4.21 it follows that the schedules are identical in the intervals $[O^{\max} + \delta, t_1 + \delta)$ and $[t_1 + \delta, t_2 + \delta)$. This means that $C_S(R, t_1) = C_S(R, t_2)$, a contradiction. Therefore, S has no empty time slot in $[t_1, t_1 + P)$. At time t_1 there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n (C_q - \epsilon_q(t_1))$; at time $t_1 + P$ the remaining demand is strictly larger: $\sum_{q=1}^n C_q - \epsilon_q(t_1 + P)$. But the additional demand in the interval $(t_1, t_1 + P]$ is less than or equal to $P \cdot U$, since in the interval $(t_1, t_1 + P]$ there are $\frac{P}{T_j}$ new requests of τ_j . In the interval

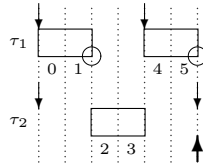


Figure 4.9: If $D_2 < 6$, or $D_2 = 6$ and the tie is broken by giving a higher priority to the request of τ_1 , the system is unschedulable.

$(t_1, t_1 + P]$ the CPU availability is P , it follows that the additional demand is larger than P which implies that $U > 1$, contradicting the hypothesis. ■

Notice that Lemma 4.25 holds whether or not the system is feasible, but we use partially extended schedules.

It follows from Lemmata 4.25 and 4.23 that $[0, O^{max} + 2P)$ is a feasibility interval only if $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$.

Corollary 4.26 *For the deadline driven scheduler applied to general deadline asynchronous periodic task sets, the system is feasible iff $U \leq 1$ and all deadlines in the interval $[0, O^{max} + 2P)$ are met.*

Proof. Immediate from Lemmata 4.25, 4.23, 4.21 and Corollary 4.5. ■

We shall now extend the results of Leung and Merrill [LM80] and those of Baruah, Howell and Rosier [BRH90] on the feasibility interval to asynchronous systems with arbitrary deadlines. We consider systems where the deadline may be larger than the period; in such systems several requests of the same task may be active simultaneously.

We shall first examine the interest of the arbitrary deadline systems in comparison with general deadline systems; i.e., is it possible to obtain schedulable sets by increasing the deadlines of unschedulable task sets?

There is no “interest” to modify the deadlines of a late deadline system, since if such a system is not schedulable it follows from Theorem 4.13 that $\sum_{i=1}^n \frac{C_i}{T_i} > 1$ and increasing/decreasing deadlines leaves the system unschedulable. However arbitrary deadlines may have an interest in comparison with the general deadline case.

Example 4.27 Consider the following synchronous general deadline system: $\{\tau_1 = \{T_1 = 4, D_1 = 2, C_1 = 2, O_1 = 0\}, \tau_2 = \{T_2 = 6, D_2 \leq 6, C_2 = 3, O_2 = 0\}\}$. We suppose that the value of D_1 is fixed by the constraints of the system ($D_1 = 2$), but not D_2 . If we consider general deadlines ($D_2 \leq 6$), the system is not schedulable (notice that a tie occurs if $D_2 = 6$) as illustrated in Figures 4.9

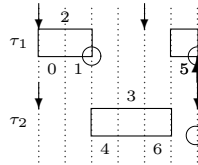


Figure 4.10: If $D_2 = 6$ and the tie is broken by giving a higher priority to the request of τ_2 , the system is also unschedulable (as predicted by Corollary 4.5).

and 4.10. This is not the case if we allow D_2 to be larger than T_2 : for instance if we choose $D_2 = 7$, the system becomes schedulable (see Figure 4.11). ■

We now come back to the extension of the results on the feasibility intervals for arbitrary deadline systems. In this framework we shall use another kind of extension of the schedule in case of deadline failure. We redefine the scheduling rule (and consequently the functions $\sigma(t)$ and $\epsilon_i^k(t)$) thus: whenever a request (say δ_i^k) misses its deadline, the execution of the system continues and the request δ_i^k remains active till its completion, whether or not new requests of the same task occur; this is not tedious here, since we consider systems where there may be several active requests of the same task at the same time. For convenience, let us call this kind of schedule: *extended schedules*. Extended schedules may also be applied to late or general deadline systems, but in case of deadline failure(s) it may happen that many requests of a same task are active simultaneously, contrary to what happened with partially extended schedules. Remark that, if the system is feasible, the extended schedule is the same as the normal one and the relation $g_i \leq \left\lceil \frac{D_i}{T_i} \right\rceil$ still holds¹⁰, but for unfeasible systems with $U > 1$, some g_i are unbounded. It follows from this new system behavior that the function $\epsilon_i^k(t)$ differs from its previous definition: $\epsilon_i^k(t)$ is the amount of processor time used by the request δ_i^k in the interval $[R_i^k, R_i^k + t)$, whatever the period and the deadline; $\epsilon_i^k(t)$ is defined for any t (contrary to the corresponding definition for static schedulers).

Lemma 4.19 may be generalized as follows.

Lemma 4.28 *For any request-dependent deadline driven extended scheduler applied to arbitrary deadline asynchronous periodic task sets, for each task τ_i , for any instant $t \geq O_i$ and any k such that $R_i^k \leq t$, we have $\epsilon_i^k(t) \geq \epsilon_i^{k+h_i}(t+P)$, with $h_i = \frac{P}{T_i}$.*

Proof. We prove the lemma by contradiction and assume that there is some first instant (the time is discrete in our model of computation) such that there

¹⁰ g_i is the maximal number of simultaneous active requests of τ_i in a feasible schedule.

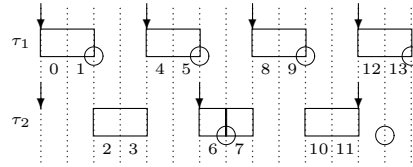


Figure 4.11: With arbitrary deadlines the system may be schedulable (from time $t = 12$ the schedule repeats).

is some $j, k, t \geq R_j^k$, with $\epsilon_j^k(t) < \epsilon_j^{k+h_j}(t + P)$. From the definition of $\epsilon_i^k(t)$ it follows that $\epsilon_i^k(t)$ is a non-decreasing discrete step function with $0 \leq \epsilon_i^k(t) \leq C_i$. The function increases at time instants where the corresponding request of τ_i is executing, otherwise the function is constant. We have $\epsilon_i^k(R_i^k) = 0 = \epsilon_i^{k+h_i}(R_i^{k+h_i})$. Then there must be some time $R_j^k < t' < t$ such that $\delta_j^{k+h_j}$ is executing at $t' + P$ while δ_j^k is not executing at time t' . This can only occur if there is a task (say task $\tau_i, 1 \leq i \leq n, i = j$ is possible) with a request $\delta_i^{k'}$ which is executing at time t' while the request $\delta_i^{k'+h_i}$ is not executing at $t' + P$, implying that the request $\delta_i^{k'}$ has a higher priority than δ_j^k (and $\delta_i^{k'+h_i}$ has a higher priority than $\delta_j^{k+h_j}$). But this means that $\epsilon_i^{k'}(t') < \epsilon_i^{k'+h_i}(t' + P) = C_i$, contradicting the fact that t is the first instant with this property. ■

We extend the definition of $C_S(R, t)$:

Definition 4.29 We define the configuration of the extended schedule S at time t for the system R as:

$$C_S(R, t) = ((\gamma_1(t), \alpha_1(t), \beta_1(t)), \dots, (\gamma_n(t), \alpha_n(t), \beta_n(t))).$$

where

- $\gamma_i(t)$ is the time elapsed since the last request of τ_i , if $t \geq O_i$; $\gamma_i(t) = t - O_i$ if $t < O_i$.
- $\alpha_i(t)$ is the number of active requests of τ_i at time t .
- $\beta_i(t)$ is the amount of processor time used at time t by the oldest active request of τ_i (if any). If $\alpha_i(t) = 0$, we define $\beta_i(t) = 0$.

■

In particular, $\gamma_i(0) = -O_i$; $\gamma_i(R_i^k) = 0 \forall k$; $\alpha_i(0) = 1$ if $O_i = 0$, 0 otherwise; $\gamma_i(t) < 0 \Rightarrow \alpha_i(t) = 0$; $\alpha_i(O_i) = 1$; $\beta_i(0) = 0 = \beta_i(O_i)$.

We present here a result very similar to Corollary 2.63 considered for static schedulers; remark that in the present case the property holds even if the system is unfeasible.

Lemma 4.30 *Let $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ be an asynchronous arbitrary deadline system with the deadline driven scheduler. For each task τ_i , for any instant $t \geq O_i$, we have either $(\alpha_i(t) < \alpha_i(t+P))$ or $[(\alpha_i(t) = \alpha_i(t+P))$ and $(\beta_i(t) \geq \beta_i(t+P))]$.*

Proof. If $\alpha_i(t) = 0$, then either $\alpha_i(t+P) > 0$ or $\alpha_i(t+P) = 0 = \beta_i(t+P) = \beta_i(t)$. Otherwise, $\alpha_i(t) = n_i(t) - m_i(t)$ where $n_i(t) = \#\{k | R_i^k \leq t\} = \max\{k | R_i^k \leq t\}$ is the number of requests of τ_i started till time t and $m_i(t) = \#\{k | \epsilon_i^k(t) = C_i\} = \max\{k | \epsilon_i^k(t) = C_i\}$ is the number of completed requests of τ_i till time t ; $n_i(t+P) = n_i(t) + h_i$ and from Lemma 4.28 it occurs that $m_i(t+P) \leq m_i(t) + h_i$, hence $\alpha_i(t+P) \geq \alpha_i(t)$; and if $\alpha_i(t) = \alpha_i(t+P)$ then $m_i(t+P) = m_i(t) + h_i$ and, $\beta_i(t) = \epsilon_i^{m_i(t)+1}(t) \geq \epsilon_i^{m_i(t)+1+h_i}(t+P) = \beta_i(t+P)$. ■

Lemma 4.31 *The configuration of the extended schedule S at time $t+1$ is univocally determined by the configuration at time $t \geq 0$ for any request-dependent scheduler.*

Proof. This can be shown by considering the following algorithm which computes $C_S(R, t+1)$ from $C_S(R, t)$, the task characteristics (i.e., O_i, D_i, C_i, T_i) and the rule used to break the ties.

```

forall  $i \in [1, n]$  Do
  If  $\gamma_i(t) < 0$  Then  $\gamma_i(t+1) := \gamma_i(t) + 1$ ; {one progresses to  $O_i$ }
  Else  $\gamma_i(t+1) := (\gamma_i(t) + 1) \bmod T_i$ ;
  {from  $O_i$ ,  $\gamma_i(t)$  progresses cyclically}
EndIf
 $\alpha_i(t+1) := \alpha_i(t)$ ;
 $\beta_i(t+1) := \beta_i(t)$ ; {in general  $\alpha_i$  and  $\beta_i$  are unchanged }
If  $(\gamma_i(t+1) = 0)$  Then {a new request occurs at time  $t+1$ }
   $\alpha_i(t+1) := \alpha_i(t) + 1$ ; {if  $\alpha_i(t) = 0$ ,  $\beta_i(t) = \beta_i(t+1) = 0$  too}

```


Endlf

Od
 $j := n + 1$;
 {in this algorithm the $\Omega_i(t) = (\lfloor \frac{t-O_i}{T_i} \rfloor - (\alpha_i(t) - 1)) \bmod h_i + 1$ is the rank (modulo h_i) of the oldest active request of τ_i (if any); it is univocally determined by the configuration of the extended schedule S at time t .}
For $p := 1$ To n Do
 If $\alpha_p(t) > 0$ and ($j = n + 1$ or $\delta_p^{\Omega_p(t)}(t) \succ \delta_j^{\Omega_j(t)}(t)$) Then
 {from the request-dependency assumption the actual rank of the requests is not necessary, only the rank $\Omega_i(t)$ is sufficient }
 $j := p$
 Endlf
Od
If ($j \leq n$) Then
 $\beta_j(t + 1) := \beta_j(t) + 1$;
 {execution of the request of τ_j during one time unit}
 If ($\beta_j(t + 1) = C_j$) Then
 $\alpha_j(t + 1) := \alpha_j(t) - 1$; {the request becomes inactive}
 $\beta_j(t + 1) := 0$;
 Endlf
Endlf ■

Remark the improvement of the proof in comparison with the static situation (see Theorem 2.60): here we consider extended schedules, which lead to some simplifications. Moreover, deadline failures may be easily detected from the configuration: τ_i misses a deadline at time t (or before) if $\alpha_i(t) > 0$ and $(\alpha_i(t) - 1)T_i + \gamma_i(t) \geq D_i$. (Compare this with the partially extended schedule).

Now we have the material to extend Lemma 4.22.

Lemma 4.32 *Let S be the feasible schedule constructed by a request-dependent deadline driven scheduler applied to an arbitrary deadline asynchronous periodic task set R . Then $C_S(R, t_1) = C_S(R, t_2)$, where $t_1 = O^{max} + P$ and $t_2 = t_1 + P$.*

Proof. Assume $C_S(R, t_1) \neq C_S(R, t_1 + P)$. First remark that $\forall i, \gamma_i(t_1) = \gamma_i(t_1 + P)$. From Lemma 4.30 it follows that there is a number j such that either $(\alpha_j(t_1) < \alpha_j(t_1 + P))$ or $[(\alpha_j(t_1) = \alpha_j(t_1 + P)) \text{ and } (\beta_j(t_1) > \beta_j(t_1 + P))]$. In both cases there must be a natural k such that: $\epsilon_j^k(t_1) > \epsilon_j^{k+h_j}(t_1 + P)$ and $\epsilon_i^r(t) \geq \epsilon_i^{r+h_i}(t + P) \forall i = 1, \dots, n, \forall r : R_i^r \leq t < R_i^r + D_i$. We first show that the schedule has no idle time slot in $[t_1, t_1 + P)$ (i.e., the CPU remains busy in $[t_1, t_1 + P)$).

Suppose there is some idle time slot at time $t_1 + \delta, 0 \leq \delta < P$. This implies that no task request is active at that time; that is, for all $i \in \{1, \dots, n\}$, and for all k such that $R_i^k \leq t_1 + \delta \leq R_i^k + D_i$ we have: $\epsilon_i^k(t_1 + \delta) = C_i$. By Lemma 4.28, the fact that $\gamma_i(t) = \gamma_i(t + P) \forall t \geq O^{max}$ and $\alpha_i(O^{max} + \delta) = \alpha_i(t_1 + \delta) = 0 \forall i$, this implies $C_S(R, O^{max} + \delta) = C_S(R, t_1 + \delta)$. Hence, from Lemma 4.31, the schedules in the intervals $[O^{max} + \delta, t_1 + \delta)$ and $[t_1 + \delta, t_1 + P + \delta)$ are identical. This means that $C_S(R, t_1) = C_S(R, t_1 + P)$, a contradiction. Therefore, S has no idle time slot in $[t_1, t_1 + P)$. At time t_1 there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n (\alpha_q(t_1) \cdot C_q - \beta_q(t_1))$; at time $t_1 + P$, from Lemma 4.30 and the relations above, the remaining demand: $\sum_{q=1}^n (\alpha_q(t_1 + P) \cdot C_q - \beta_q(t_1 + P))$ is larger. But the additional demand in the interval $[t_1, t_1 + P)$ is equal to $P \cdot U$. In the interval $[t_1, t_1 + P)$ the CPU availability is P ; it follows that the additional demand is larger than P which implies that $U > 1$, a contradiction with the feasibility of the schedule. ■

In the same way, the Theorem 4.23 can be extended to arbitrary deadline systems.

Theorem 4.33 *Let S be the extended schedule of a task system R constructed by a request-dependent deadline driven scheduler applied to an asynchronous and arbitrary deadline periodic task set. R is feasible iff (1) all deadlines in the interval $[0, O^{max} + 2P)$ are met in the schedule S , and (2) $C_s(R, O^{max} + P) = C_s(R, O^{max} + 2P)$.*

Proof.

(only if part). If R is feasible on a single processor, then the schedule S constructed by any deadline driven scheduler must be a valid schedule. Consequently, all deadlines in the interval $[0, O^{max} + 2P)$ are met in S , and, by Lemma 4.32, we have $C_s(R, O^{max} + P) = C_s(R, O^{max} + 2P)$.

(if part). Since $C_s(R, t_1) = C_s(R, t_2)$, from Lemma 4.31, the schedule S repeats every P units of time, starting from t_1 . Since all deadlines in the interval $[0, t_2)$ are met in S , the deadlines of all task computations must also be met in S , since deadline failures may be read from C_S . Hence, R is feasible on a single processor. ■

Both conditions of Theorem 4.33 are clearly necessary for the feasibility of the system, The following example shows that condition (1) does not imply condition (2).

Example 4.34 Consider the following system:

| | T | D | C | O |
|----------|-----|-----|-----|-----|
| τ_1 | 4 | 4 | 2 | 0 |
| τ_2 | 4 | 7 | 3 | 2 |

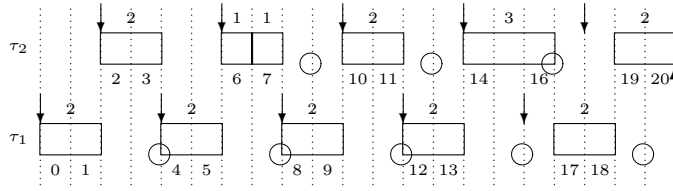


Figure 4.12: Both conditions of Theorem 4.33 are necessary for the feasibility of the system.

We have $P = 4$, $O^{max} = 2$, $t_1 = 6$ and $t_2 = 10$; moreover as illustrated in Figure 4.12, all deadlines in interval $[0, 10)$ are met, but $C_S(R, 6) \neq C_S(R, 10)$, since $\epsilon_2^1(6) = 2$ and $\epsilon_2^2(10) = 1$. And τ_2 misses its deadline at time 21. ■

Again, if $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, only the first condition needs to be considered; Lemma 4.25 can be adapted for arbitrary deadline systems.

Lemma 4.35 *Let S be the extended schedule constructed by a request-dependent deadline driven scheduler applied to an arbitrary deadline asynchronous periodic task set R . If $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$, then $C_S(R, t) = C_S(R, t+P)$ when $t = O^{max} + P$.*

Proof. Assume $C_S(R, t) \neq C_S(R, t+P)$. As in the proof of Lemma 4.32, there must be a task j and a natural k : $\epsilon_j^k(t) > \epsilon_j^{k+h_j}(t+P)$ and $\epsilon_i^r(t) \geq \epsilon_i^{r+h_i}(t+P)$ for $1 \leq i \leq n$ and $R_i^r \leq t$. We first show that the schedule has no idle time slot in $[t, t+P)$ (i.e., the CPU remains busy in $[t, t+P)$). Suppose there is some idle time slot at time $t + \delta$, $0 \leq \delta < P$. This implies that no task request is active at that time; consequently $C_S(R, t + \delta) = ((\gamma_1(t + \delta), 0, 0), \dots, (\gamma_n(t + \delta), 0, 0))$. By Lemma 4.28, this implies $C_S(R, O^{max} + \delta) = C_S(R, t + \delta)$. Hence, from Lemma 4.31, it follows that in the intervals $[O^{max} + \delta, t + \delta)$ and $[t + \delta, t + P + \delta)$ the schedules are identical. This means that $C_S(R, t) = C_S(R, t + P)$, a contradiction. Therefore, S has no idle time slot in $[t, t + P)$. At time t_1 there may be remaining demands to satisfy, amounting to $\sum_{q=1}^n (\alpha_q(t_1) \cdot C_q - \beta_q(t_1))$; at time $t_1 + P$, from Lemma 4.30 and the relations above, the remaining demand: $\sum_{q=1}^n (\alpha_q(t_1 + P) \cdot C_q - \beta_q(t_1 + P))$ is larger. But the additional demand in the interval $[t_1, t_1 + P)$ is equal to $P \cdot U$. In the interval $[t_1, t_1 + P)$ the CPU availability is P , it follows that the additional demand is larger than P which implies that $U > 1$, a contradiction with the hypothesis. ■

It follows from Lemma 4.35 and Theorem 4.33 that $[0, O^{max} + 2P)$ is a feasibility interval for asynchronous and arbitrary deadline systems only if $U \leq 1$:

Corollary 4.36 *For the deadline driven scheduler applied to arbitrary deadline asynchronous periodic task sets, the system is feasible iff $U \leq 1$ and all deadlines in the interval $[0, O^{max} + 2P)$ are met.*

Proof. Immediately follows from Lemmata 4.35, and 4.31 and Corollary 4.5. ■

4.5.1 Synchronous systems

For synchronous and arbitrary deadline systems, feasibility intervals can be improved: from Theorem 4.10, we know that we only need to check the feasibility of the system from time $t = 0$ till the first idle processor point, i.e., in the *first busy period*. This concept has already been exploited in the literature for static priority schedulers [LL73, ABRT93, Leh90]. In section 2.4 we have used a similar notion of level- i busy period for the feasibility of arbitrary deadline and static systems. Spuri [Spu96] has considered this notion for dynamic priority schedulers and we shall consider this point in details in section 4.8.2. Hence, we consider the schedule from time $t = 0$ to the first instant $L > 0$ such that all requests started strictly before L have completed their processing before or at time L , which amounts to consider the static level- n busy period (see Definition 2.34); consequently we have that $L = \lambda_n$ (defined in section 2.4), which can be computed by iteration from Theorem 2.40.

Theorem 4.37 *The interval $[0, L)$ is a feasibility interval for synchronous and arbitrary deadline systems with the deadline driven scheduler.*

Proof. Immediately follows from the fact that L is the first idle point in the schedule of the system. Consequently, from Theorem 4.10 if no deadline was missed before L the system is feasible. ■

We see here the interest of the Theorem 4.10 with respect to Theorem 4.8 (which consider idle points and idle time units, respectively) since the first idle point occurs before (or at) the first idle time unit (if any), which gives a better (i.e., smaller) feasibility interval. Moreover, Theorem 4.10 is applicable if $U = 1$ while Theorem 4.8 is not applicable in this case. Remark that if $U > 1$ Theorem 4.37 is correct but useless since in this case $L = \lambda_n = \infty$. On the other hand, if $U \leq 1$ we have from Lemma 2.38 that $L = \lambda_n \leq P$.

4.6 Response times

4.6.1 Introduction

We shall now extend the notion of the response time, and its computation, to the deadline driven scheduler. We shall exploit it in section 4.7 for the feasibility problem; we shall show that, for synchronous and asynchronous systems, the maximal time complexity of our computation based on the response time notion (while remaining exponential for asynchronous systems), exhibits an exponential improvement in comparison with previous results issued from the literature.

For static priority assignments, the response time computation consists mainly to determine the interference of higher priority tasks during the execution of a request. With dynamic priority rules, it is meaningless to compare the priority of tasks, since it changes with time. However, for the (dynamic priority) deadline driven scheduler, it is judicious to compare the priority of requests, which only depends on the corresponding deadline and does not change¹¹ with the time. As usual, δ_i^k is the k^{th} request ($k = 1, 2, \dots$) of task τ_i , which occurs at time $R_i^k = O_i + (k - 1)T_i$. The request δ_i^k has a higher priority than the request δ_j^r if the deadline of δ_i^k occurs before the one of δ_j^r . When the deadlines are equal, we break the tie by giving a higher priority to the request with the smaller index (i.e., $\tau_h : h = \min\{i, j\}$). Remark that in this manner the scheduling rule is request-dependent. Remark also that the relative priority between two requests does not change with time: $\delta_i^k(t) \succ \delta_j^r(t) \Leftrightarrow \delta_i^k(t+1) \succ \delta_j^r(t+1)$. For this reason we shall omit t in our notation and consider the following definition:

Definition 4.38 The request δ_i^k has a *higher priority* than the request δ_j^r ($\delta_i^k \succ \delta_j^r$) iff

$$\begin{cases} O_i + (k - 1)T_i + D_i \leq O_j + (r - 1)T_j + D_j & \text{if } i < j \\ O_i + (k - 1)T_i + D_i < O_j + (r - 1)T_j + D_j & \text{otherwise} \end{cases}$$

■

Of course, a higher priority will only be effective if the two requests indeed compete, i.e., if there exists a time instant such that both requests are active.

Remark that if we choose another way to resolve the tie but the relative priority between two requests does not change with time, it is always possible to re-index the various tasks in order that the previous definition is valid, but this

¹¹recall that this is a property of the deadline driven scheduler when there is no tie, and a choice we made when there are ties.

re-indexing may depend on the time. If we want to allow to change the priority of a request δ_i^k with time, it will be possible to obtain a lower bound and an upper bound for its response time by re-indexing the tasks in order to place τ_i in the first ($i = 1$) then in the last ($i = n$) position.

4.6.2 The response time of the first request in the synchronous case

We shall first consider the simplified case of the response time of the first request of a task in the synchronous case. As usual, this response time is the smallest value r_i^1 such that r_i^1 is exactly equal to the total interference from higher priority requests, plus the computation due to τ_i :

Theorem 4.39 *Let $\gamma = \{\tau_1, \dots, \tau_n\}$ be a synchronous task set with arbitrary deadlines; for our deadline driven scheduler, r_i^1 is the smallest solution (if it is not greater than D_i and the higher priority requests meet their deadline too, or if we adopt the extended schedule) of the equation:*

$$r_i^1 = C_i + \sum_{j < i} W_j^i(r_i^1)C_j + \sum_{j > i} w_j^i(r_i^1)C_j \quad (4.1)$$

where

$$\begin{aligned} W_j^i(\omega) &= \min \left(\left\lfloor \frac{(D_i + T_j - D_j)^+}{T_j} \right\rfloor, \left\lceil \frac{\omega}{T_j} \right\rceil \right), \\ w_j^i(\omega) &= \min \left(\left\lceil \frac{(D_i - D_j)^+}{T_j} \right\rceil, \left\lceil \frac{\omega}{T_j} \right\rceil \right), \\ x^+ &= \max\{x, 0\}. \end{aligned}$$

Proof. We assume that either all requests of higher priority than δ_i^1 meet their deadline and $r_i^1 \leq D_i$, or the extended schedule is considered. We compute the interference of all higher priority requests in the interval $[0, r_i^1)$. We distinguish between the requests with a lesser index ($j < i$, case (a)) and those with a greater index ($j > i$, case (b)). It may be noticed that the other requests of τ_i , (i.e., δ_i^k with $k > 1$) have a lower priority than δ_i^1 and do not interfere.

- (a) $W_j^i(\omega)$ is the number of requests of τ_j with a higher priority than the first request of τ_i ($j < i$) which occur in the interval $[0, \omega)$. This number is equal to the number of requests of τ_j which occur in the interval $[0, \omega)$

with a deadline less than or equal to D_i . $\lceil \frac{\omega}{T_j} \rceil$ denotes the number of requests of τ_j in the interval $[0, \omega)$ and $k = \left\lfloor \frac{(D_i + T_j - D_j)^+}{T_j} \right\rfloor$ denotes the number of all requests of τ_j in the interval $[0, D_i)$ with a deadline less than or equal to D_i . Indeed k is the largest natural integer such that $(k - 1)T_j + D_j \leq D_i$ if $D_j \leq D_i$, $k = 0$ otherwise. The formula follows.

- (b) $w_j^i(\omega)$ is the number of requests of τ_j with a higher priority than the first request of τ_i ($j > i$) which occur in the interval $[0, \omega)$. This number is equal to the number of requests of τ_j which occur in the interval $[0, \omega)$ with a deadline strictly less than D_i . $\lceil \frac{\omega}{T_j} \rceil$ denotes the number of requests of τ_j in the interval $[0, \omega)$ and $k = \left\lceil \frac{(D_i - D_j)^+}{T_j} \right\rceil$ denotes the number of all requests of τ_j in the interval $[0, D_i)$ with a deadline strictly less than D_i . Indeed k is the largest natural integer such that $k < \frac{D_i - D_j + T_j}{T_j} = 1 + \frac{D_i - D_j}{T_j}$ if $(D_j < D_i)$, $k = 0$ otherwise. The formula follows.

If the interval r_i^1 is equal to the interference of higher requests than δ_i^1 plus the computation time of τ_i (C_i) that means that the interval $[0, r_i^1)$ is fully utilized and that δ_i^1 can complete its execution. The response time of δ_i^1 is the first such instant. ■

Notice that, even if the system is not feasible (e.g., if $U > 1$) and we consider the extended scheduling, r_i^1 is upper bounded¹² since so are W_j^i and w_j^i ; consequently

$$r_i^1 \leq B_i = C_i + \sum_{j < i} \left\lfloor \frac{(D_i + T_j - D_j)^+}{T_j} \right\rfloor C_j + \sum_{j > i} \left\lceil \frac{(D_i - D_j)^+}{T_j} \right\rceil C_j.$$

Remark that Theorem 4.39 is valid for synchronous systems with arbitrary deadlines, and not only for late or general deadlines, since there is no previous (active) request.

Equation (4.1) may have several solutions; for example, if we consider the following synchronous system:

| | C | T | D |
|----------|-----|-----|-----|
| τ_1 | 1 | 2 | 2 |
| τ_2 | 1 | 4 | 4 |

¹²Recall that this is not the true for static priority schedulers: if $C_1 \geq T_1, r_2^1 = \infty$.

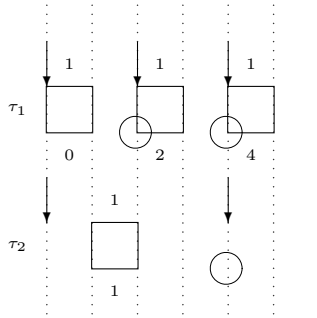


Figure 4.13: Multiplicity of the solutions for Equation (4.1).

Equation (4.1) has 2 solutions: $r_2^1 = 2$ and $r_2^1 = 3$ (see Figure 4.13); only the first one is relevant.

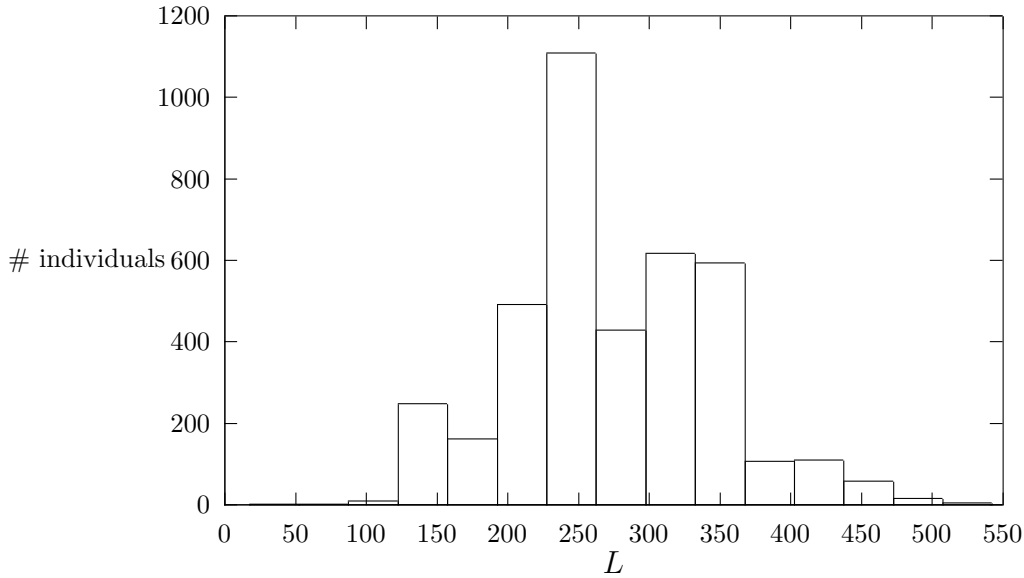
Notice that r_i^1 occurs on both sides of the Equation (4.1). The minimal value for r_i^1 can be found by iteration:

$$\begin{cases} z_0 &= C_i \\ z_{k+1} &= C_i + \sum_{j < i} W_j^i(z_k)C_j + \sum_{j > i} w_j^i(z_k)C_j \end{cases}$$

The iteration is growing and proceeds until $z_{k+1} = z_k = r_i^1$. If we do not consider the extended scheduling, the iteration may be stopped if z_k exceeds D_i because τ_i is then deemed unschedulable. Hence the maximal number of iterations is $\lfloor \frac{D_i - C_i}{\min_{j \neq i} C_j} \rfloor + 1$. Like in the static case, it is not difficult to see that this maximal number of iterations is very pessimistic, in practice the number of iterations is by far lower since the iterative process can stop with $w_k < D_i$ and w_k can be increased by several C_j 's (and not necessarily the minimal one). We shall see experimental results concerning the exact number of iterations in sections 4.6.4 and 4.6.6, for more general cases of response time computations.

But the iteration also converges if we do not stop the system on the first deadline failure, i.e., with extended schedules, since W_j^i and w_j^i are upper bounded: with the same remark as above on the pessimistic nature of this bound, the maximal number of iterations is $\lfloor \frac{B_i - C_i}{\min_{j \neq i} C_j} \rfloor + 1$.

Theorem 4.40 *The iteration converges.*

Figure 4.14: Frequency of the value L

Proof. We only need to show that the successive approximations z_k to r_i are monotonically increasing. By induction, we have that:

$$z_k \geq z_{k-1} \Rightarrow z_{k+1} \geq z_k$$

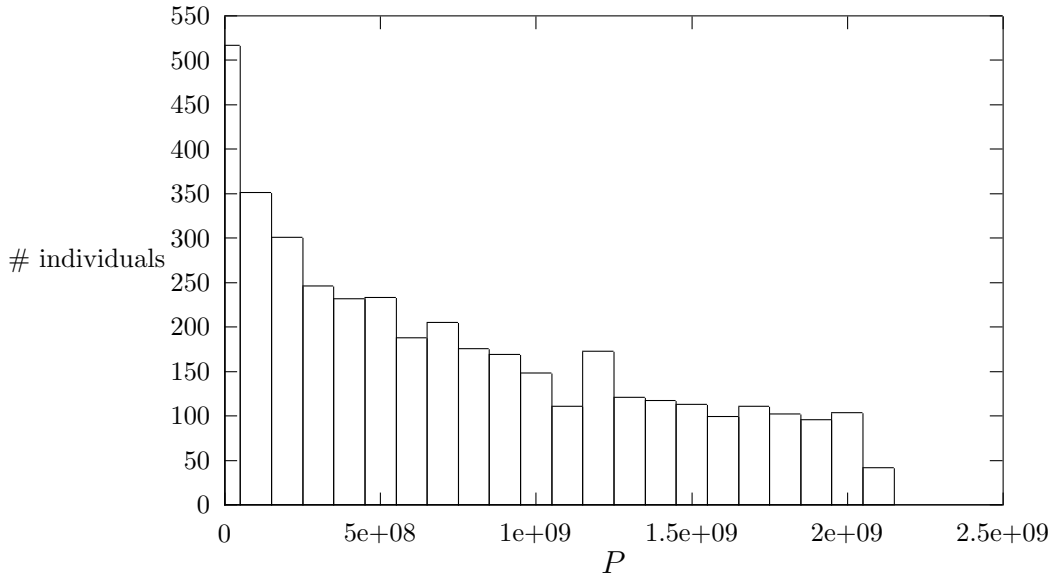
if $z_k \geq z_{k-1}$ then

$$\begin{aligned} z_{k+1} &= C_i + \sum_{j<i} W_j^i(z_k)C_j + \sum_{j>i} w_j^i(z_k)C_j \\ &\geq C_i + \sum_{j<i} W_j^i(z_{k-1})C_j + \sum_{j>i} w_j^i(z_{k-1})C_j = z_k, \end{aligned}$$

since in an interval of length z_k the number of requests which occur in this interval with deadline less than (or equal to) D_i increases monotonically with the length of the interval. The property then results from the fact that $z_0 = C_i \leq z_1$. ■

Theorem 4.41 *The iteration converges to the minimal solution*

Proof. We shall show by induction on k that $z_0 < z_1 < \dots < z_k \Rightarrow r_i^1 \geq z_k$. The property is true initially: $C_i = z_0 < C_i + \sum_{j<i} W_j^i(C_i)C_j + \sum_{j>i} w_j^i(C_i)C_j = z_1 \Rightarrow r_i^1 \geq z_1$. Suppose that the property is true up to k and we have $z_0 < z_1 < \dots < z_k < z_{k+1}$; by induction hypothesis $r_i^1 \geq z_k$.

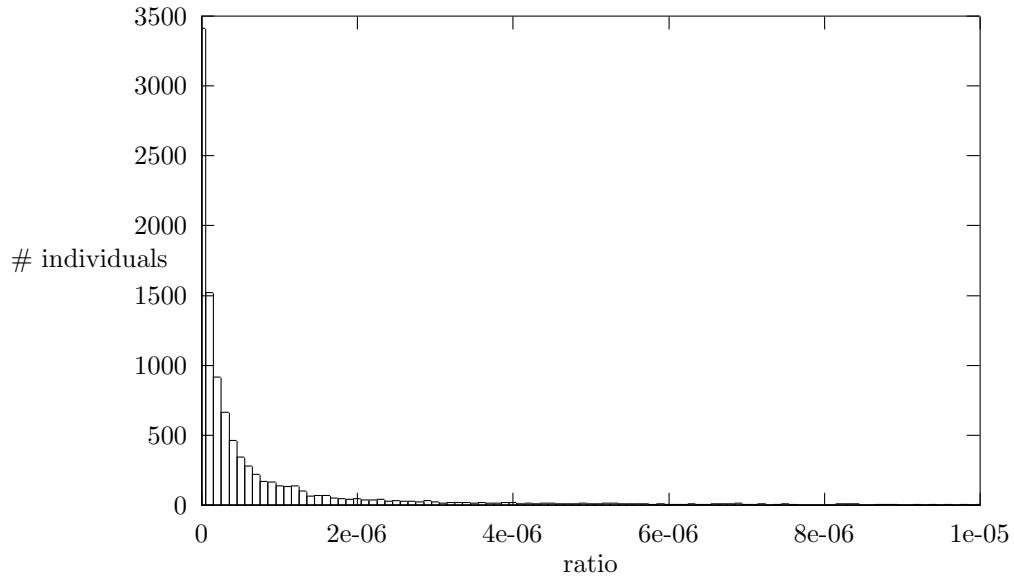
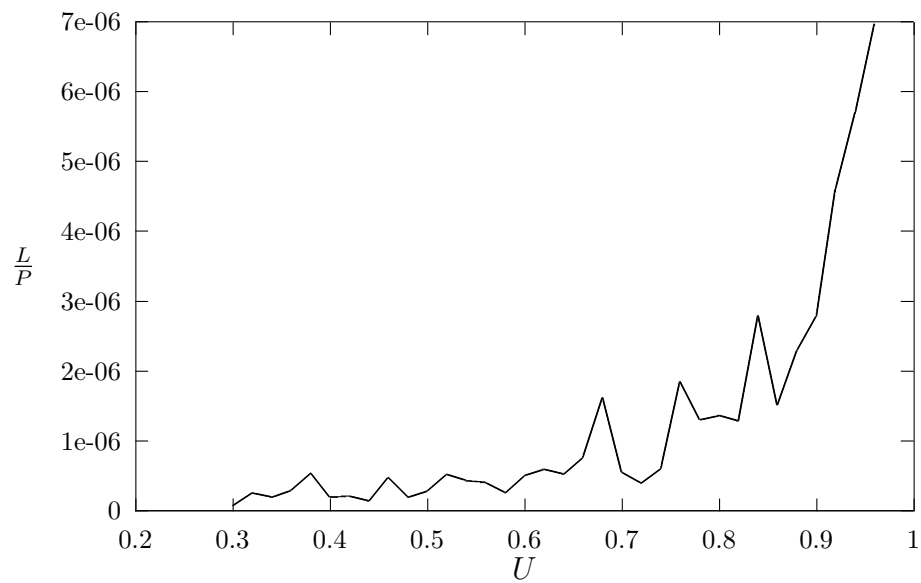
Figure 4.15: Frequency of the value P .

As a consequence, r_i^1 is at least equal to C_i plus the interference of higher priority requests in the interval $[0, z_k)$. In other words, we have that $r_i^1 \geq C_i + \sum_{j < i} W_j^i(z_k)C_j + \sum_{j > i} w_j^i(z_k)C_j$. ■

It may be noticed that, from a schedulability point of view, the response time of the first request of τ_i in the synchronous case is less instructive than the one for static schedulers. We have shown in Chapter 3 that this response time is the worst case for static schedulers with general deadlines. From a schedulability point of view this property is important since the system is then schedulable iff $r_i^1 \leq D_i$ for $i = 1, \dots, n$. We have shown in Example 4.12 that this property does not hold for the deadline driven scheduler.

We have considered first this case for its potential simplifications and for didactic purposes, not for its practical interest.

In the current state of our knowledge there is no polynomial time algorithm for the feasibility of synchronous and general (or arbitrary) deadline systems with the deadline driven scheduler. We propose to consider all response times in the interval $[0, L)$, according to Theorem 4.10, i.e., by considering the system from time 0 until the first idle point (beside 0). But in the “worst case” (i.e., if the utilization factor is 1 and the system is schedulable) we have to consider an interval of length $L = P = \text{lcm}\{T_i | i = 1, \dots, n\}$, which may grow exponentially with the number n of tasks. Remark that in practical cases (even if the utilization factor is near 1) the bound P is very pessimistic and

Figure 4.16: Frequency of the ratio $\frac{L}{P}$.Figure 4.17: The average ratio $\frac{L}{P}$ in function of U .

L is significantly smaller. In order to illustrate the fact that the bound P is very pessimistic for L , we shall consider these values on randomly chosen late deadline task sets for $n = 20$, the periods are randomly chosen in $[50, 100]$, and the computation times are chosen in order to have utilization factors in the interval $[0.3, 1)$; the systems are not necessarily feasible but the extended schedules are considered. As usual we shall first consider the distributions of these variables; Figures 4.14 and 4.15 show the distributions of the values L and P , respectively, for systems where the utilization factor $U \in [0.85, 0.95]$. Figure 4.16 shows the distribution of the ratio $\frac{L}{P}$ (it may be noticed that the minimal value for P in our simulations is near 10^6). It seems again judicious to consider the average of the distributions; remark that we have observed the same behavior for other variation domains of U . Figure 4.17 shows the average ratio $\frac{L}{P}$ in function of the utilization factor: it increases with U and but remains less than $7 \cdot 10^{-6}$ (for $U \leq 0.95$), exhibiting the fact that the bound P is very pessimistic for L , even if U is near 1. It could be also interesting to study the ratio $\frac{L}{P}$ in the neighborhood of 1, but this question remains for further research.

The situation is less attractive in comparison with static priority schedulers, however. In the next section we shall extend the formulas for the computation of the k^{th} request of a task. We shall show that the maximal time complexity of our feasibility check based on the response time notion, while remaining exponential, exhibits an exponential improvement in comparison with previous results issued from the literature.

4.6.3 Response time of the k^{th} request in asynchronous and general deadline systems

We have seen in section 4.5 that the feasibility interval for general deadline systems (synchronous and asynchronous cases) contains many requests of the task set, more precisely the length of the interval is proportional to L and to P for synchronous and asynchronous system, respectively. Hence the interest of the general response time computation, i.e., the response time ρ_i^k for the k^{th} request of task τ_i for synchronous or asynchronous systems. Again, the response time for the k^{th} request of τ_i is the smallest value ρ_i^k such that ρ_i^k is equal to the total interference from higher priority requests, plus the computation due to τ_i :

Theorem 4.42 *Let $\gamma = \{\tau_1, \dots, \tau_n\}$ be an asynchronous task set with general deadlines; for our deadline driven scheduler, ρ_i^k is the smallest solution of the equation:*

$$\rho_i^k = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j$$

where

$$\begin{aligned} R_j^p &= O_j + (p-1)T_j \\ m_j &= \begin{cases} \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil & \text{if } O_j < R_i^k \\ 1 & \text{otherwise} \end{cases} \\ \pi_j &= \begin{cases} (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{if } j > 0 \text{ and } R_j^{m_j} < R_i^k < R_j^{m_j} + D_j \\ & \text{and } \delta_j^{m_j} \succ \delta_i^k \\ 0 & \text{otherwise} \end{cases} \\ z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall j \\ p : \pi_p = \max\{\pi_j | \pi_j > 0\} & \text{otherwise} \end{cases} \\ \hat{m}_j &= \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil \\ \tilde{m}_j &= \begin{cases} \hat{m}_j & \text{if } \hat{m}_j > 0 \text{ and } z \neq 0 \text{ and } \delta_j^{\hat{m}_j} \prec \delta_z^{m_z} \text{ and } R_j^{\hat{m}_j} \geq R_i^k \\ \hat{m}_j + 1 & \text{otherwise} \end{cases} \\ n_j &= \begin{cases} \min \left(\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} + T_j - D_j)^+}{T_j} \right\rceil \right) & \text{if } j < i \\ \min \left(\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} - D_j)^+}{T_j} \right\rceil \right) & \text{otherwise} \end{cases} \\ x^+ &= \max\{x, 0\}. \end{aligned}$$

Proof. We assume that all higher priority requests than δ_i^k meet their deadline. We cannot consider here partially extended schedules, since they sometimes drop requests before satisfaction and this is not taken into account in the present formulas. The case of extended schedules will be taken into account in section 4.6.5 with arbitrary deadline systems since then we may have many requests of a same task which are simultaneously active, even with general deadlines.

R_j^p denotes the arrival time of the p^{th} request for τ_j . The interference from higher priority requests can be computed from the response time of some higher priority requests (so that the formula may be recursive). For a task τ_j (with $j \neq i$), we only have to consider the requests from the m_j^{th} , the last one that

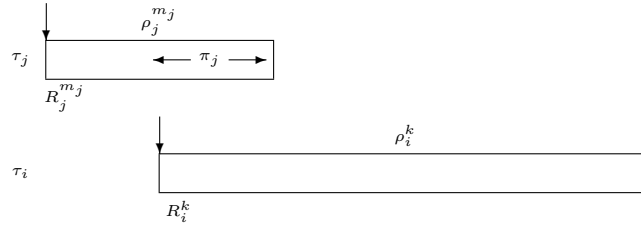


Figure 4.18: Interference of requests which occur strictly before time R_i^k .

precedes strictly R_i^k (see Figure 4.18) if any, i.e., if $O_j < R_i^k$, otherwise we take $m_j = 1$ and consider the requests from the first one.

Indeed the requests of τ_j that occur before the m_j^{th} , if any, are completed before time R_i^k and have no direct¹³ interference on the response time ρ_i^k . It is easy to see that $m_j = \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil$ if $O_j < R_i^k$; otherwise $m_j = 1$, hence the formula above. The k^{th} request of τ_i may be directly delayed by a part (or all) of the m_j^{th} request of τ_j if $R_j^{m_j} < R_i^k$, $R_j^{m_j} + D_j > R_i^k$, the corresponding response time $\rho_j^{m_j}$ is greater than $R_i^k - R_j^{m_j}$ and the request $\delta_j^{m_j}$ has a higher priority than δ_i^k ($\delta_j^{m_j} \succ \delta_i^k$). The interference is then equal to $\pi_j = R_j^{m_j} + \rho_j^{m_j} - R_i^k$. Suppose that two such requests of higher priority (say $\delta_a^{m_a} \succ \delta_b^{m_b} \succ \delta_i^k$) have an interference in the response time of the k^{th} request of τ_i ; in that case the interference of the higher priority one is included in the interference of the lower one: since at time R_i^k both requests are active, the request $\delta_b^{m_b}$ ends its execution after the request $\delta_a^{m_a}$. Hence, $\pi_b > \pi_a > 0$ and the total interference of all requests which precede strictly δ_i^k is equal to $\pi_z = \max\{\pi_j | j \neq i \text{ and } \pi_j > 0\}$ ($z = 0 = \pi_0$ if no request occurring strictly before time R_i^k delays the k^{th} request of τ_i ; notice also that $\pi_i = 0$).

Let us consider now the interference of requests which occur after or at time R_i^k . For each task τ_j ($j \neq i$) we have to consider the requests from the \tilde{m}_j^{th} , i.e., the first one which is not included in the term π_z . Let \hat{m}_j denote the rank of the last request of τ_j which occurs strictly before $R_i^k + \pi_z$ ($\hat{m}_j = 0$ if no request occurs before this time); it can be seen that $\hat{m}_j = \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil$. We have to distinguish four cases: (i) $\pi_z = 0$, (ii) $\hat{m}_j = 0$, (iii) the request $\delta_j^{\hat{m}_j}$ has a higher priority than $\delta_z^{m_z}$, or is $\delta_z^{m_z}$, and (iv) the request $\delta_j^{\hat{m}_j}$ has a lower priority than $\delta_z^{m_z}$.

- (i) If $\pi_z = 0$, we have to consider the requests from the rank $\hat{m}_j + 1$, i.e.,

¹³They may have an indirect interference, by delaying a request with priority between it and δ_i^k which is not finished at R_i^k .

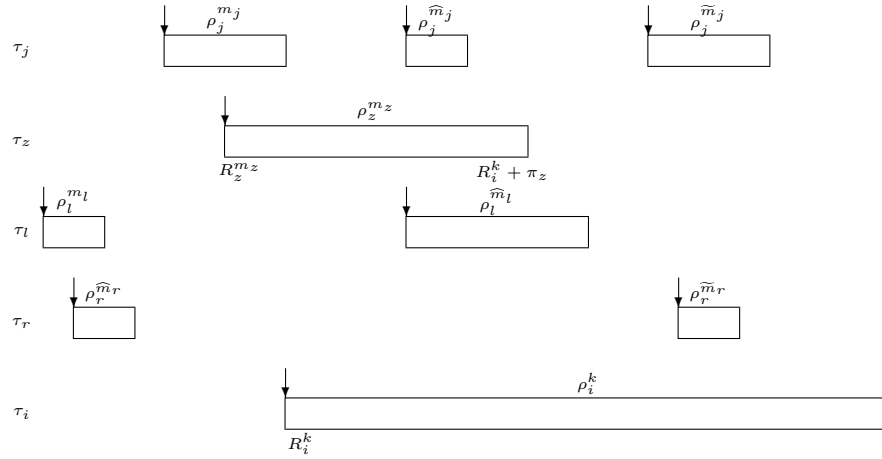


Figure 4.19: Interference of requests which occur after (or at) time R_i^k .

from the first one which occurs after or at R_i^k .

- (ii) If $\pi_z \neq 0$ and $\widehat{m}_j = 0$, we have to consider the requests from the first one and consequently $\widetilde{m}_j = \widehat{m}_j + 1 = 1$.
- (iii) In this case, all requests of τ_j up to $\delta_j^{\widehat{m}_j}$ have a higher priority than $\delta_z^{m_z}$, or is $\delta_z^{m_z}$, and were already considered in the term π_z (or were terminated so early that they have no direct impact on π_z nor on ρ_i^k); hence we have to consider the requests from the next one, i.e., $\widetilde{m}_j = \widehat{m}_j + 1$ (this is the case of the request $\delta_j^{\widetilde{m}_j}$ in Figure 4.19).
- (iv) In this case, we have to distinguish two sub-cases: (iva) the request $\delta_j^{\widetilde{m}_j}$ occurs strictly before time R_i^k or (ivb) after or at R_i^k .
 - (a) In this case we have necessarily that $\pi_j = 0$ (otherwise $\pi_j > \pi_z$ and the latter would not be maximal) and we have to consider the requests from the rank $\widehat{m}_j + 1$ (this is the case of the request $\delta_r^{\widehat{m}_r}$ in Figure 4.19).
 - (b) The interference of this request is not included in the term π_z , nothing has been executed for it before $R_i^k + \pi_z$ (this is the case of the request $\delta_l^{\widehat{m}_l}$ in Figure 4.19) and we must consider the request of τ_j from rank $\widetilde{m}_j = \widehat{m}_j$. This is the only case where $\widetilde{m}_j \neq \widehat{m}_j + 1$, hence the formula above.

n_j denotes the number of higher priority requests (than δ_i^k) of τ_j which occur in the interval $[R_j^{\widetilde{m}_j}, R_i^k + \rho_i^k)$.

If $j < i$ we have to consider the number of requests of τ_j which occur in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$ with a deadline before or at time $R_i^k + D_i$; $\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil$ denotes the number of requests of τ_j in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$ and $x = \left\lfloor \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} + T_j - D_j)^+}{T_j} \right\rfloor$ denotes the number of requests of τ_j from $R_j^{\tilde{m}_j}$ with a deadline less than or equal to $R_i^k + D_i$. Indeed x is the largest natural integer such that $R_j^{\tilde{m}_j} + (x - 1)T_j + D_j \leq R_i^k + D_i$ if $R_j^{\tilde{m}_j} + D_j \leq R_i^k + D_i$, 0 otherwise; hence the formula.

If $j > i$ we have to consider the number of requests of τ_j which occur in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$ with a deadline strictly before time $R_i^k + D_i$; $\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil$ denotes the number of requests of τ_j in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$ and $x = \left\lfloor \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} - D_j)^+}{T_j} \right\rfloor$ denotes the number of requests of τ_j from $R_j^{\tilde{m}_j}$ with a deadline strictly less than $R_i^k + D_i$. Indeed, x is the largest natural integer such that $R_j^{\tilde{m}_j} + (x - 1)T_j + D_j < R_i^k + D_i$ if $R_j^{\tilde{m}_j} + D_j < R_i^k + D_i$, 0 otherwise; hence the formula.

We have shown that the interference of higher priority tasks than τ_i in the interval $[R_i^k, R_i^k + \rho_i^k)$ plus the computation of τ_i is

$$I(\rho_i^k) = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j.$$

The k^{th} request of τ_i ends its computation at the first instant $R_i^k + \rho_i^k$ such that the equality is satisfied: $\rho_i^k = I(\rho_i^k)$. \blacksquare

4.6.4 Computation of ρ_i^k

In the same way than in sections 3.3.1 and 3.7.1 we shall here present several methods for the computation of ρ_i^k . Again the computation can be divided into two parts: the computation of the term π_z (and the related numbers: $\rho_j^{\tilde{m}_j}, z, \tilde{m}_j$) and the computation of the lowest solution of the equation:

$$\rho_i^k = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j.$$

The second part of the computation can be resolved by an iterative process.

$$\begin{aligned} w_0 &= C_i + \pi_z, \\ w_{k+1} &= C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j. \end{aligned}$$

Theorem 4.43 *The number of iterations of the iterative process for the computation of ρ_i^k is bounded by $\lfloor \frac{(D_i - C_i - \pi_z)^+}{\min_{j \neq i} C_j} \rfloor + 1$.*

Proof. The property follows from the fact that $w_0 = C_i + \pi_z$, the process stops in the worst case when $w_{k-1} \leq D_i$ and $w_k > D_i$, and at each iteration w_k is increased by at least C_j time units for at least one request of τ_j , unless the solution has been reached. ■

Experimental results (similar to the one exhibited in sections 3.3.1 and 3.7.1, including the analysis of distributions) show that the actual number of iterations represents 4 % of the bound given by Theorem 4.43; again the bound is very pessimistic and the iterative process converges much more quickly. We have not distinguished in this study feasible and unfeasible systems; it could be interesting however to consider feasible and unfeasible systems separately, but this question remains open for further research, together with the reason why the acceleration ratio seems so stable (it does not depend much on n in our experiments).

We consider now the computation of the term π_z . Method 1, already defined for static priority schedulers, which computes recursively ρ_i^k (the recursive function $f(i, k)$ can be adapted here). The time complexity and in particular the total number of calls is quite different, however, since each response time depends on $n - 1$ other response times in the general case, and the recursion stops in the worst case with the computation of the response time of the first request of the system, i.e., ρ_j^1 such that $O_j = 0$.

We shall now estimate the maximal time complexity of the computation of ρ_i^k using method 1; the following theorem gives an upper bound for this time complexity.

Theorem 4.44 *The maximal time complexity of the computation of ρ_i^k using method 1 is $O(\frac{(n-1)^{R_i^k+1}}{n} \cdot \gamma)$, where $\gamma = (\frac{\max\{D_j - C_j | j=1, \dots, n\}}{\min\{C_j | j=1, \dots, n\}} \times n)$.*

Proof. We shall count the maximal number of calls of the function $f()$ induced by the function $f(i, k)$. This number of calls (say $\mathcal{Q}()$) can be defined recursively: $\mathcal{Q}(0) = 0$ (no request occurs before time 0), $\mathcal{Q}(u = R_i^k) = \sum_{j \neq i} (1 + \mathcal{Q}(u -$

1)) = $(n-1)(1+\mathcal{Q}(u-1))$ when $(u \geq 1)$; the first term corresponds to the direct calls of the functions $f(1, m_1), \dots, f(i-1, m_{i-1}), f(i+1, m_{i+1}), \dots, f(n, m_n)$ and the second term corresponds to the numbers of recursive calls induced by the latter, which in the worst case occur at time $R_i^k - 1$ (this is rather pessimistic, of course, and this scenario cannot repeat at each step, but we are only interested here by an upper bound for the number of calls of the function $f()$). We shall show by induction on u , that $\mathcal{Q}(u) = \sum_{r=1}^u (n-1)^r \approx \frac{(n-1)^{u+1}}{n}$. The property is true in the trivial case $\mathcal{Q}(0) = 0$. Suppose the property true till index u and consider $\mathcal{Q}(u+1)$. By definition $\mathcal{Q}(u+1) = (n-1)(1+\mathcal{Q}(u))$ and from the induction hypothesis $\mathcal{Q}(u+1) = (n-1)(1+\sum_{r=0}^u (n-1)^r) = \sum_{r=1}^{u+1} (n-1)^r$. Consequently, the maximal time complexity of the computation of ρ_i^k using method 1 is $\mathcal{Q}(R_i^k) \cdot \gamma$, where γ is the maximal time complexity of the iterative process. ■

It may be noticed that the actual number of recursive calls is by far less than $\sum_{r=1}^{R_i^k} (n-1)^r$ since in the average case the previous request of τ_j occurs $\frac{T_j}{2}$ time units before (and, more marginally, some of the $f(j, m_j)$'s do not have to be computed, if it is sure that they have no impact when the deadlines are fulfilled). Nevertheless, the total number of recursive calls remains unreasonable to handle “real size” problems. The maximal space complexity of the computation of ρ_i^k using method 1 is $O(R_i^k) = O(k)$, since the recursion stops in the worst case with the computation of ρ_j^1 such that $O_j = 0$. Method 1 is not suited to handle “real size” problems: the interest of this method (and its implementation) lies in the “verification” of our formulas.

Method 2 consists to compute all response times of requests occurring in the interval $[0, R_i^k)$ (before applying the iterative process) by increasing values of their arrival time. In this way, whenever the method computes a response time (say ρ_p^r) all $\rho_j^{m_j}$ needed are already computed (since we need $\rho_j^{m_j}$ if $R_j^{m_j} < R_p^r < R_j^{m_j} + D_j$ and $\delta_j^{m_j} > \delta_p^k$), for this reason we have at each instant only to know the response time of the last request of τ_j ($j \neq i$), if any, so that the recursive aspect of the formulas is not a drawback (but we need to store n previously computed response times). Consequently, the maximal space complexity of the method 2 for the computation of ρ_i^k is $O(n)$. The maximal time complexity is $\sum_{j=1}^n \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot \gamma$, or simply γ if we consider that this method will be used in the analysis of a feasibility interval $[0, X]$, so that the previous response time computations were due anyway (and that the corresponding requests met their deadline). Method 2 can be improved in some situations: for R_i^k sufficiently large (i.e., from $R_i^k > O^{max} + 2P$), the schedule and then the response times repeat from time $O^{max} + P$; hence in the worst case we have to compute $\sum_{j=1}^n \left\lfloor \frac{O^{max} + P}{T_j} \right\rfloor$ response times to reach the periodic part of the

schedule and $\sum_{j=1}^n \frac{P}{T_j}$ to compute the response times $\rho_j^{k_j}$ corresponding to $\rho_j^{m_j}$ (i.e., $k_j = \min\{k | k = m_j \bmod \frac{P}{T_j} \text{ and } R_j^k \geq O^{max} + P\}$). The maximal time complexity of the computation of ρ_i^k using method 2 is then $\sum_{j=1}^n \lfloor \frac{O^{max} + 2P}{T_j} \rfloor \cdot \gamma$. Method 3 cannot be applied efficiently here: the computation of the term π_z needs in general the value of $n - 1$ other response times. It is possible however that such a request (say $\delta_j^{m_j}$) does not have an impact on ρ_i^k but, in general, answering this question requires the value of π_j , in other words the value of $\rho_j^{m_j}$. Again, in this case, method 3 amounts to method 2.

4.6.5 Response time of the k^{th} request in asynchronous and arbitrary deadline systems

We shall now consider the more general response time computation: the response time of the k^{th} request in asynchronous and arbitrary deadline systems for our dynamic deadline driven scheduler. The Theorem 4.42 does not hold in this more general case: for arbitrary deadline systems, we have to consider several requests of τ_j since several requests of the same task may be active at the same time, including previous requests of task τ_i . Remark that we may perform the computations either while assuming that the previous requests met their deadline, or with the extended schedules defined on page 133.

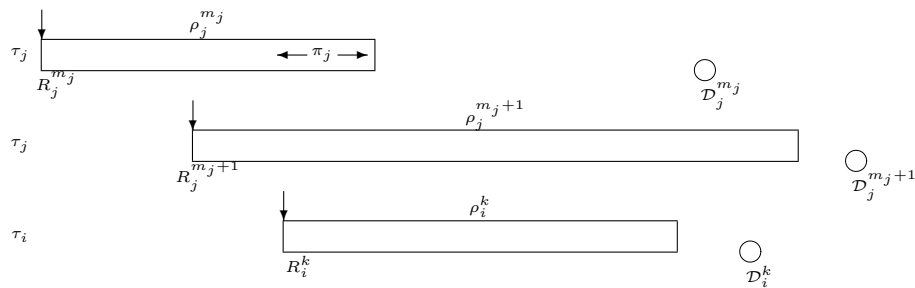


Figure 4.20: Interference of requests which occur strictly before time R_i^k .

Theorem 4.45 *Let $\gamma = \{\tau_1, \dots, \tau_n\}$ be an asynchronous task set with arbitrary deadlines, for our deadline driven scheduler, ρ_i^k is the smallest solution of the equation:*

$$\rho_i^k = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j$$

where

$$\begin{aligned}
R_j^p &= O_j + (p-1)T_j \\
m_j &= \begin{cases} \min \left\{ \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil, \left\lceil \frac{R_i^k + D_i + T_j - O_j - D_j}{T_j} \right\rceil \right\} & \text{if } (j < i) \wedge (O_j < R_i^k) \\ & \wedge (\delta_j^1 \succ \delta_i^k) \\ \min \left\{ \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil, \left\lceil \frac{R_i^k + D_i - O_j - D_j}{T_j} \right\rceil \right\} & \text{if } (j \geq i) \wedge (O_j < R_i^k) \\ & \wedge (\delta_j^1 \succ \delta_i^k) \\ 0 & \text{otherwise} \end{cases} \\
\pi_j &= \begin{cases} (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+ & \text{if } j > 0 \wedge m_j > 0 \\ & (\wedge R_j^{m_j} + D_j > R_i^k) \\ & \text{if feasibility is assumed} \\ 0 & \text{otherwise} \end{cases} \\
z &= \begin{cases} 0 & \text{if } \pi_j = 0 \forall j \\ p : \pi_p = \max\{\pi_j \mid \pi_j > 0\} & \text{otherwise} \end{cases} \\
\tilde{m}_j &= \begin{cases} \max \left\{ \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1, \right. \\ \left. \min \left\{ \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil + 1, \left\lceil \frac{R_z^{m_z} + D_z - O_j - D_j + T_j}{T_j} \right\rceil + 1 \right\} \right\} & \text{if } j < i \\ \max \left\{ \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1, \right. \\ \left. \min \left\{ \left\lceil \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rceil + 1, \left\lceil \frac{R_z^{m_z} + D_z - O_j - D_j + T_j}{T_j} \right\rceil \right\} \right\} & \text{otherwise} \end{cases} \\
n_j &= \begin{cases} \min \left(\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} + T_j - D_j)^+}{T_j} \right\rceil \right) & \text{if } j < i \\ \min \left(\left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} - D_j)^+}{T_j} \right\rceil \right) & \text{otherwise} \end{cases} \\
x^+ &= \max\{x, 0\}.
\end{aligned}$$

Proof. We may assume that all higher priority requests than δ_i^k meet their deadlines, but this is not essential here: we may consider instead the extended schedules with soft deadlines introduced before. R_j^p denotes the arrival time of the p^{th} request for τ_j . The interference of higher priority requests can be computed from the response time of some higher priority requests (so that the formula may be recursive).

For a task τ_j (with $1 \leq j \leq n$), we only have to consider the requests from the m_j^{th} , the last one occurring strictly before R_i^k (see Figure 4.20) with a higher priority than δ_i^k , if any, (i.e., if $O_j < R_i^k$ and $\delta_j^1 \succ \delta_i^k$), otherwise we take $m_j = 0$.

Indeed, the interference of δ_j^r for $r < m_j$, if any, is included in $\rho_j^{m_j}$ since $R_j^r < R_j^{m_j}$ and $\delta_j^r \succ \delta_j^{m_j}$. Let us assume that $O_j < R_i^k$ and $\delta_j^1 \succ \delta_i^k$: in this case $m_j = \max\{q | (R_j^q < R_i^k) \text{ and } \delta_j^q \succ \delta_i^k\}$. The rank (say x) of the last request of τ_j which occurs strictly before time R_i^k is $x = \left\lfloor \frac{R_i^k - O_j}{T_j} \right\rfloor$. If $j \leq i$, the rank (say y) of the last request of τ_j with a higher priority than δ_i^k is the largest natural integer such that $(y - 1)T_j + O_j + D_j \leq R_i^k + D_i$, i.e., $y = \left\lfloor \frac{R_i^k + D_i + T_j - O_j - D_j}{T_j} \right\rfloor$. If $j > i$, the rank (say y) of the last request of τ_j with a higher priority than δ_i^k is the largest natural integer such that $(y - 1)T_j + O_j + D_j < R_i^k + D_i$, i.e., $y = \left\lfloor \frac{R_i^k + D_i - O_j - D_j}{T_j} \right\rfloor$. Hence the formulas above.

The interference of the request $\delta_j^{m_j}$ (if $m_j > 0$) is equal to $\pi_j = (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+$. We may save this computation if feasibility is assumed and $R_j^{m_j} + D_j \leq R_i^k$. Remark that, in arbitrary deadline systems, it is not relevant to consider only the previous request of τ_j (with respect to δ_i^k) since this request may have a lower priority than δ_i^k and consequently no impact on ρ_i^k while a previous request of τ_j may be active at time R_i^k with a higher priority than δ_i^k (this is the case in Figure 4.20, where \mathcal{D}_r^p denotes the deadline of the p^{th} request of τ_r : $\rho_j^{m_j}$ has an impact on ρ_i^k while $\rho_j^{m_j+1}$ has no impact and $R_j^{m_j} < R_j^{m_j+1} < R_i^k$). Remark also that $m_i = k - 1$ and, here, the m_i^{th} request of task τ_i may have an interference on ρ_i^k ; it follows that the term π_i must be considered. Suppose that two such requests of higher priority (say $\delta_a^{m_a} \succ \delta_b^{m_b} \succ \delta_i^k$) have an interference in the response time of the k^{th} request of τ_i ; in that case the interference of the higher priority one is included in the interference of the lower one: since at time R_i^k both requests are active, the request $\delta_b^{m_b}$ ends its execution after the request $\delta_a^{m_a}$. Hence, $\pi_b > \pi_a > 0$ and the total interference of all requests which precede strictly the k^{th} request of τ_i is equal to $\pi_z = \max\{\pi_j | \pi_j > 0\}$ ($z = 0 = \pi_z$ if no request occurring strictly before time R_i^k delays the k^{th} request of τ_i).

Let us now consider the requests of τ_j from the first one, say the \tilde{m}_j^{th} , whose impact is not included in the term π_z . We have $R_j^{\tilde{m}_j} \geq R_i^k$ since a request occurring strictly before R_i^k either is completed before R_i^k or is included in the term π_z , or has a lower priority than δ_i^k and has no impact. We may assume $j \neq i$ since the next request of τ_i from R_i^k is δ_i^k itself, and the following ones have a lower priority. If $\pi_z = 0$, there is no other constraint and $\tilde{m}_j = x_1 = \left\lfloor \frac{(R_i^k - O_j)^+}{T_j} \right\rfloor + 1$ is the rank of the first request of τ_j after or at R_i^k . If $\pi_z \neq 0 \neq z$, $\tilde{m}_j \leq x_2 = \left\lfloor \frac{(R_i^k + \pi_z - O_j)^+}{T_j} \right\rfloor + 1$, the rank of the first request of τ_j after or at $R_i^k + \pi_z$. But if between $\delta_j^{x_1}$ and $\delta_j^{x_2}$ there are requests with a lower priority than $\delta_z^{m_z}$, they are not included in π_z and must also be considered. Let x_3 be

the rank of the first request of τ_j with a priority strictly lower than $\delta_z^{m_z}$: if $j < i$, x_3 is the least natural such that $O_j + (x_3 - 1)T_j + D_j > R_z^{m_z} + D_z$, so that $x_3 = \left\lfloor \frac{R_z^{m_z} + D_z - O_j - D_j + T_j}{T_j} \right\rfloor + 1$; if $j > i$, x_3 is the least natural such that $O_j + (x_3 - 1)T_j + D_j \geq R_z^{m_z} + D_z$, so that $x_3 = \left\lceil \frac{R_z^{m_z} + D_z - O_j - D_j + T_j}{T_j} \right\rceil$. Hence, $\tilde{m}_j = \max\{x_1, \min\{x_2, x_3\}\}$. Hence the formulas.

n_j denotes the number of higher priority requests (with respect to δ_i^k) of τ_j which occur in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$. $\left\lfloor \frac{(R_i^k + \rho_i^k - R_j^{\tilde{m}_j})^+}{T_j} \right\rfloor$ denotes the number of requests of τ_j in the interval $[R_j^{\tilde{m}_j}, R_i^k + \rho_i^k)$. If $j < i$, $x = \left\lfloor \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} + T_j - D_j)^+}{T_j} \right\rfloor$ denotes the number of requests of τ_j from $R_j^{\tilde{m}_j}$ with a deadline less than or equal to $R_i^k + D_i$; indeed x is the largest natural integer such that $R_j^{\tilde{m}_j} + (x - 1)T_j + D_j \leq R_i^k + D_i$ if $R_j^{\tilde{m}_j} + D_j \leq R_i^k + D_i$, 0 otherwise. If $j > i$, $x = \left\lfloor \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} - D_j)^+}{T_j} \right\rfloor$ denotes the number of requests of τ_j from $R_j^{\tilde{m}_j}$ with a deadline strictly less than $R_i^k + D_i$; indeed, x is the largest natural integer such that $R_j^{\tilde{m}_j} + (x - 1)T_j + D_j < R_i^k + D_i$ if $R_j^{\tilde{m}_j} + D_j < R_i^k + D_i$, 0 otherwise. Hence the formulas.

We have shown that the interference of higher priority tasks than τ_i in the interval $[R_i^k, R_i^k + \rho_i^k)$ plus the computation of τ_i is

$$I(\rho_i^k) = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j$$

The k^{th} request of τ_i ends its computation at the first instant $R_i^k + \rho_i^k$ such that the equality is satisfied: $\rho_i^k = I(\rho_i^k)$. ■

Notice that, if the system is feasible, $\rho_i^k \leq D_i$, and we may save the computation of π_j if $R_j^{m_j} + D_j \leq R_i^k$. Otherwise, with the extended behavior described on page 133, n_j is upper bounded by $\left\lfloor \frac{(R_i^k + D_i - R_j^{m_j} + T_j - D_j)^+}{T_j} \right\rfloor$, but π_z , while being finite, may indefinitely increase with k as exhibited, for instance, with the system $S = \{\tau_1 = \{T_1 = 2, C_1 = 3, O_1 = 0\}\}$: the terms π_z in the computation of ρ_1^k (for increasing values of k) are: 0, 1, 2, ...

4.6.6 Computation of ρ_i^k

We shall here present several methods for the computation of ρ_i^k , and we shall see that the computation of the response time for arbitrary deadline systems

(or general deadline with extended behavior) is quite different, particularly for method 2 which must be revisited here. As usual, the computation can be divided into two parts: the computation of the term π_z (and the related numbers: $\rho_j^{m_j}, z, \tilde{m}_j$) and the computation of the least solution of equation:

$$\rho_i^k = C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j.$$

The second part of the computation can be resolved by an iterative process.

$$\begin{aligned} w_0 &= C_i + \pi_z, \\ w_{k+1} &= C_i + \pi_z + \sum_{j \neq i} n_j \cdot C_j. \end{aligned}$$

Theorem 4.43 remains valid here: the number of iterations of the iterative process for the computation of ρ_i^k is bounded by $\lfloor \frac{D_i - C_i - \pi_z}{\min_{j \neq i} C_j} \rfloor + 1$, if we stop the iteration when w_k exceeds D_i . But here, contrary to what happened with static schedulers, if we consider the extended schedule, even if $U > 1$, ρ_i^k is finite and the number of iteration is upper bounded since n_j is upper bounded by $\left\lfloor \frac{(R_i^k + D_i - R_j^{\tilde{m}_j} + T_j - D_j)^+}{T_j} \right\rfloor$.

From this point, however, since those computations will essentially be used to check the feasibility of the schedule, we shall only consider limited schedules, i.e., the schedule is undefined from the first deadline failure (if any), and the iterative process stops when w_k exceeds D_i .

Experimental results (similar to the ones exhibited in sections 3.3.1, 3.7.1 and 4.6.4, including the analysis of the distributions which have a very similar behavior than the one described for static schedulers) show that the actual number of iterations represents 0.5 % of the bound given by Theorem 4.43; consequently again the bound is very pessimistic and the iterative process converges much more quickly. We have not distinguished in this study feasible and unfeasible systems, it could be however interesting to consider feasible and unfeasible system separately, but this question remains open for further research.

We consider now the computation of the term π_z . Method 1 already defined for static priority schedulers and dynamic general deadline systems, which computes recursively ρ_i^k (with the recursive function $f(i, k)$) can be applied here. In comparison with dynamic and general deadline systems, the maximal

number of calls is quite similar here: each response time depends on n other response times.

Theorem 4.46 *The maximal time complexity of the computation of ρ_i^k using method 1 is $O(n^{R_i^k} \cdot \gamma)$, where $\gamma = \frac{\max\{D_j - C_j | j=1, \dots, n\}}{\min\{C_j | j=1, \dots, n\}} \times n$.*

Proof. We shall count the maximal number of calls of the function $f()$ induced by the function $f(i, k)$. This number of calls (say $\mathcal{Q}()$) can be defined recursively: $\mathcal{Q}(0) = 0$ (no request occurs before time 0), $\mathcal{Q}(u = R_i^k) = \sum_{j=1}^n (1 + \mathcal{Q}(u - 1)) = n(1 + \mathcal{Q}(u - 1))$ if $(u \geq 1)$; the first term corresponds to the direct calls of the function $f(1, m_1), \dots, f(n, m_n)$ and the second term corresponds to the numbers of recursive calls induced by the latter, which in the worst case occurs at time $R_i^k - 1$ (this is rather pessimistic, of course, since this scenario cannot repeat at each step, but we are only interested here by an upper bound for the maximal number of calls of the function $f()$). We shall show by induction on u , that $\mathcal{Q}(u) = \sum_{r=1}^u n^r \equiv \frac{n^{u+1} - 1}{n - 1}$. The property is true in the trivial case $\mathcal{Q}(0) = 0$. Suppose the property true till index u and consider the case of $\mathcal{Q}(u + 1)$. By definition $\mathcal{Q}(u + 1) = n + n\mathcal{Q}(u)$ and from induction hypothesis $\mathcal{Q}(u + 1) = n + \sum_{r=1}^u n^{r+1} = \sum_{r=1}^{u+1} n^r$. Consequently the maximal time complexity of the computation of ρ_i^k using method 1 is $\mathcal{Q}(R_i^k) \cdot \gamma$, where γ is the maximal time complexity of the iterative process. ■

The maximal space complexity of the computation of ρ_i^k using method 1 is $O(R_i^k)$, since the recursion stops in the worst case with the computation of ρ_j^1 such that $O_j = 0$. The actual space and time complexity of method 1 is of course by far lower, but our experimentation shows that method 1 is not suited to handle “real size” problem (e.g., for $n = 10$, method 1 needs in general several hours of CPU time¹⁴); the interest of this method (and its implementation) lies in the “verification” of our formulas.

Method 2 consists in computing all response times of requests occurring in the interval $[0, R_i^k)$ (before applying the iterative process for ρ_i^k) by increasing values of their arrival time. In this way, whenever the method computes a response time (say ρ_p^r) all needed $\rho_j^{m_j}$ are already computed (since we need $\rho_j^{m_j}$ if $R_j^{m_j} < R_p^r$). However, in the arbitrary deadline case, the request $\rho_j^{m_j}$ is not necessarily the previous request of τ_j ; for this reason method 2 must know (or “store”) more than a single (previous) response time for each task. We shall however show that we have at most to know $\left\lceil \frac{D^{max} - D^{min} + T^{min}}{T^{min}} \right\rceil$ response times for each task.

¹⁴Simulation on a ULTRA-SPARC (140 MHZ) with 64 MBYTES of RAM.

Theorem 4.47 *The maximal space complexity of the computation of ρ_i^k using method 2 is $O(n \cdot \left\lceil \frac{D^{max} - D^{min} + T^{min}}{T_j} \right\rceil)$.*

Proof.

For each task τ_j , m_j is the rank of the last request of τ_j which occurs strictly before R_i^k with a higher priority than δ_i^k (if any); suppose $j > i$ (the worst case, since then we have to consider the last request of τ_j with a deadline strictly less than $R_i^k + D_i$): m_j is the rank of the last request of τ_j which occurs strictly before R_i^k such that $(m_j - 1)T_j + O_j + D_j < R_i^k + D_i$; between time $R_j^{m_j}$ and R_i^k , there are at most $\left\lceil \frac{(D_j - D_i)^+ + T_j}{T_j} \right\rceil$ requests of τ_j . To show this, let us call x this number of requests of τ_j ; consider first the simplified case where $\delta_j^1 \succ \delta_i^k$, a request of τ_j occurs at time R_i^k and $D_j > D_i$: x is in this case the smallest integer such that $D_j - x \cdot T_j < D_i$. If $D_j \leq D_i$, the previous request of τ_j has a higher priority than δ_i^k ; and if no request of τ_j corresponds to time R_i^k , we have to consider in the best case one request less than the simplified case; the formula follows. Consequently for each task τ_j we have at most to know the $\left\lceil \frac{(D_j - D_i)^+ + T_j}{T_j} \right\rceil$ last response times. It follows that method 2 computes all response times in the interval $[0, R_i^k)$ and needs for each ρ_i^k at most the value of the last $\left\lceil \frac{D^{max} - D^{min} + T^{min}}{T_j} \right\rceil$ response times (for each task). ■

The space complexity of method 2 is larger in the arbitrary deadline case but remains polynomial in terms of the system characteristics and the recursive aspect of the formulas is not a drawback.

The maximal time complexity of method 2 is $O(\sum_{j=1}^n \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot \gamma)$, or simply γ if we consider that the previous response time computations were due anyway. Method 2 can be again improved in some situations: for R_i^k sufficiently large (i.e., from $R_i^k > O^{max} + 2P$), the schedule and then the response times repeat from time $O^{max} + P$ if the system is feasible; in the worst case we have to compute $\sum_{j=1}^n \left\lceil \frac{O^{max} + P}{T_j} \right\rceil$ response times to reach the periodic part of the schedule and $\sum_{j=1}^n \frac{P}{T_j}$ response times to compute the $\rho_j^{k_j}$ corresponding to $\rho_j^{m_j}$ (i.e., $k_j = \min\{k | k = m_j \bmod \frac{P}{T_j} \text{ and } R_j^k \geq O^{max} + P\}$). Hence, the maximal time complexity of the computation of ρ_i^k using method 2 is $O(\sum_{j=1}^n \frac{P}{T_j} \cdot \gamma)$.

Method 3 cannot be applied efficiently here: the computation of the term π_z needs the value of n other response times. It is possible that such a request (say $\delta_j^{m_j}$) does not have an impact on ρ_i^k , but answering this question requires in general the value of π_j , in other words the value of $\rho_j^{m_j}$. Again in this case method 3 amounts to method 2.

Table 4.1 summarizes the time and the space complexities of the three meth-

| Method | Scheduler | Deadline | Space | # Resp. times |
|--------|-----------|-----------|---|--|
| M 1 | static | general | i | 2^{i-1} |
| M 2 | static | general | i | $\sum_{j=1}^{i-1} \frac{P_j}{T_j}$ |
| M 3 | static | general | i | $\sum_{r=1}^{i-1} \left\lfloor \frac{\sum_{j=1}^{i-1} T_j}{T_r} \right\rfloor$ |
| M 1 | static | arbitrary | $\left\lfloor \frac{R_i^k - \min_{k \leq i} O_k}{\min\{T_k k \leq i\}} \right\rfloor + i - 1$ | $\gg 2^i$ |
| M 2 | static | arbitrary | i | $\sum_{j=1}^{i-1} \frac{P_j}{T_j}$ |
| M 3 | static | arbitrary | n.a. | n.a. |
| M 1 | DDS | general | k | $\frac{(n-1)R_i^{k+1}}{n}$ |
| M 2 | DDS | general | n | $\sum_{j=1}^n \frac{P}{T_j}$ |
| M 3 | DDS | general | n.a. | n.a. |
| M 1 | DDS | arbitrary | k | nR_i^k |
| M 2 | DDS | arbitrary | $n \cdot \left\lfloor \frac{D^{max} - D^{min} + T^{min}}{T^{min}} \right\rfloor$ | $\sum_{j=1}^n \frac{P}{T_j}$ |
| M 3 | DDS | arbitrary | n.a. | n.a. |

Table 4.1: Space and time complexities of method 1, method 2 and method 3 for static priority assignments and for the deadline driven scheduler (DDS), expressed in terms of $O(\cdot)$.

ods considered in this work for the computation of ρ_i^k , the complexities are classified according to: the scheduler (static priority assignments or the (dynamic) deadline driven scheduler), and the relation between the period and the deadline (general and arbitrary deadline systems). From our study several concluding remarks can be raised:

- Method 3 has only an interest for static priority schedulers and general deadline systems. In comparison with other methods, schedulers or arbitrary deadline systems, method 3 is significantly superior for its time complexity as well as for its space complexity, for the computation of a single response time. Otherwise method 2 can be used for the computation of all response times of requests which occur before a given time.
- The total number of calls (and consequently the total number of iterative processes) induced by method 1 is in all cases unreasonable for “real-size” systems. The interest of this method lies in the “verification” of our formulas.
- Method 2 can be applied in all the cases with quite a similar space and time complexity, i.e., $\sum_{j=1}^n \frac{P}{T_j}$ iterative processes and $O(n)$ space

complexity (for the dynamic and arbitrary deadline case where the space complexity is bigger but remains pseudo-polynomial however). Method 2 is well suited to compute all response times of requests which occur before a given time, e.g., during a feasibility interval in order to check the feasibility of the system.

4.7 Feasibility tests for asynchronous systems

We shall here consider the feasibility problem, i.e., deciding if a system is feasible (or not), i.e., if there exists a priority rule which makes the system schedulable. Since the deadline driven priority rule is optimal, we can restrict this question by considering the schedulability of the system using the deadline driven priority rule.

We shall exploit our general response time computation for the feasibility problem. We shall show that the maximal time complexity of our computation, while remaining exponential, exhibits an exponential improvement in comparison with previous results issued from the literature.

Baruah, Howell and Rosier have defined a feasibility test for asynchronous systems, first for the general deadline case [BRH90], then for the arbitrary deadline situation [BHR93]. As in section 4.3, we shall denote by $\eta_i(t, t')$ the number of requests of task τ_i which occur in the interval $[t, t')$ with a deadline less than or equal to t' . Any feasible scheduling algorithm must give at least $\eta_i(t, t') \cdot C_i$ CPU time units to τ_i in this interval; this is a necessary condition for the schedulability of the system in this interval.

In [BRH90] for general deadlines and in [BHR93] for arbitrary deadlines the authors give the following formula for $\eta_i(t, t')$:

$$\eta_i(t, t') = \max \left\{ 0, \left\lfloor \frac{t' - O_i - D_i}{T_i} \right\rfloor - \max \left\{ 0, \left\lceil \frac{t - O_i}{T_i} \right\rceil \right\} + 1 \right\}. \quad (4.2)$$

Baruah, Howell and Rosier [BRH90] have also shown that we can restrict the analysis of the functions $\eta_i(t, t')$ to time instants included in the feasibility interval defined by Leung and Merrill.

Theorem 4.48 ([BRH90]) *An asynchronous general deadline system R is feasible iff*

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

$$2. \sum_{i=1}^n \eta_i(t, t') \cdot C_i \leq t' - t \text{ for all } 0 \leq t < t' \leq O^{max} + 2P.$$

Proof. Both conditions are clearly necessary. Suppose both conditions hold, and R is not feasible. Let S be a (partially extended) schedule of R constructed by the deadline driven scheduler (the way ties are resolved is not specified, since from Corollary 4.5, this has no impact on the schedulability, but we assume request-dependency). From Lemma 4.25, $C_S(R, O^{max} + P) = C_S(R, O^{max} + 2P)$. From Theorem 4.23, some deadline in $(0, O^{max} + 2P]$ must be missed. Let t' be the time of this deadline. Let t be the last time before t' such at time $t - 1$ either the system is idle or a request with a deadline strictly greater than t' is scheduled. Since t' corresponds to a deadline, $t' > 0$; so t is well-defined (it may be 0). Furthermore, since the deadline at t' is not met, there is an active task (with deadline t') scheduled at $t' - 1$ and $t < t'$. It follows that there is a task scheduled at every time in $[t, t')$ with its deadline not later than time t' . Since no task having a deadline less than or equal to t' is scheduled at $t - 1$, every task scheduled in $[t, t')$ must have been released not earlier than t . Since there is a task scheduled at every time in $[t, t')$ and the deadline at t' is not met, $\sum_{i=1}^n \eta_i(t, t') \cdot C_i > t' - t$: a contradiction. ■

For the arbitrary deadline case, this result can be immediately adapted, by considering the feasibility interval given by Lemma 4.32 instead of Lemma 4.22.

Theorem 4.49 *An asynchronous arbitrary deadline system R is feasible iff*

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

$$2. \sum_{i=1}^n \eta_i(t, t') \cdot C_i \leq t' - t \text{ for all } 0 \leq t < t' \leq O^{max} + 2P.$$

■

From Theorem 4.48, Baruah, Howell and Rosier [BRH90] proposed a nondeterministic algorithm to verify if a task system (say R) is not feasible in the general deadline case. The algorithm first determines whether $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$. If so, the algorithm guesses t and t' subject to the constraints of Theorem 4.48, and checks if $\sum_{i=1}^n \eta_i(t, t') > t' - t$. This nondeterministic algorithm is instrumental in the paper of Baruah, Howell and Rosier [BRH90] in order to prove an important property concerning the complexity of the feasibility problem. We shall give the result for completeness, although we are concerned here by deterministic algorithms.

Baruah, Howell and Rosier have shown that the feasibility problem is co-NP-complete¹⁵ in the strong sense¹⁶.

Theorem 4.50 ([BRH90, BHR93]) *The feasibility problem for asynchronous periodic task sets is co-NP-complete in the strong sense in the general deadline case as well as in the arbitrary deadline case.* ■

We now come back to the feasibility test of asynchronous systems in the arbitrary deadline case; from the result of Baruah, Howell and Rosier, we derive the following deterministic algorithm:

Algorithm 4.51

```

If  $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$  Then
     $\forall t \in \mathbb{N}, 0 \leq t < O^{max} + 2P$  Do
         $\forall t' \in \mathbb{N}, t < t' < O^{max} + 2P$  Do
            If  $\sum_{i=1}^n \eta_i(t, t') \cdot C_i > t' - t$  Then
                Return(false);
            Fi
        Od
    Od
    Return(true)
Else
    Return(false)
Fi.

```

It follows that the maximal time complexity of this algorithm is $O(P^2 \times n)$.

Another feasibility test for the general (or arbitrary) deadline case, based on our response time computation, consists in the computation of all response times in the feasibility interval defined by Corollary 4.26 (or 4.36 for arbitrary deadline systems), i.e., by $O(\sum_{i=1}^n \frac{P}{T_i})$ response times. We have already seen in section 4.6.4 that with method 2 the maximal time complexity of these computations is $O(\sum_{i=1}^n \frac{P}{T_i} \times n \times \frac{\max\{D_i - C_i | i=1, \dots, n\}}{\min\{C_i | i=1, \dots, n\}})$ and the space complexity is n (or $n \cdot \frac{D^{max} - D^{min} + T^{max}}{T^{min}}$ in the arbitrary case); the time complexity of our

¹⁵The complementary problem (i.e., the problem with reversed answer) is NP (see [GJ79] pp. 156 for details).

¹⁶The problem cannot be solved by a pseudo-polynomial algorithm in terms of the characteristics of the system (i.e., n, T_i, C_i, D_i, O_i) unless $P = NP$ (see [GJ79], pp. 95 for details).

computations exhibits an exponential improvement in comparison with the method derived from the results of Baruah, Howell and Rosier.

We have implemented the Algorithm 4.51 and we have applied it on randomly chosen asynchronous systems: the actual time complexity of the algorithm is very large in comparison with the response time computation approach, even for very small size problems (i.e., regarding the number of tasks and the number T^{max}): the algorithm takes an unreasonable time to check the feasibility of the system. For instance, for $n = 4$, each T_i being randomly chosen in $[5, 20]$, D_i randomly chosen in $[\frac{T_i}{2}, T_i]$ and $C_i (\leq D_i)$ in order to have a large utilization factor. Algorithm 4.51 needs in general several hours of CPU time¹⁴ while our computations take a few minutes, and can also handle “real-size” problems (e.g., $n = 100$) with a reasonable actual time complexity if P is not too high.

4.8 Feasibility tests for synchronous systems

We shall consider in this section synchronous systems. More exactly, we shall allow that the offsets are fixed by the constraints of the system but lead to a synchronization of all task requests, i.e., $\exists t \in \mathbb{N}$ such that $\forall i : t = O_i + k_i \cdot T_i$ for some $k_i \in \mathbb{N}$. Since the first requests of each task may be dropped (we shall see this property in details in Chapter 5, see Theorem 5.8) without modifying the feasibility of the task set, without loss of generality, we can assume that the synchronization occurs at time 0, i.e., $O_1 = O_2 = \dots = O_n = 0$. It may be noticed that in section 5.3 we shall study the problem of verifying if a synchronization of all task requests occurs (or not); we shall not give details here.

For this sub-class of periodic task sets (i.e., synchronous systems) some simplified feasibility tests can be defined. It is important to note that these simplified tests are necessary and sufficient conditions for synchronous systems and are only sufficient for asynchronous systems, according to Theorem 4.11.

Of course, the feasibility tests defined for asynchronous systems remain necessary and sufficient for synchronous systems, since synchronous systems are special cases of asynchronous systems. But, for an interesting sub-class of synchronous systems (i.e., when $\sum_{i=1}^n \frac{C_i}{T_i} < 1$), Theorem 4.48 can be simplified and the feasibility test derived from Baruah, Howell and Rosier for asynchronous systems can be improved. This is the purpose of the next section.

4.8.1 Feasibility of bounded general deadline synchronous task sets

For the sub-class of general deadline synchronous task sets with bounded utilization factor, in Theorem 4.48, we can choose $t = 0$ and decrease the upper bound for t' :

Theorem 4.52 ([BRH90]) *The feasibility problem for synchronous general deadline systems is solvable in $O(n \cdot \frac{U}{1-U} \cdot \max\{T_i - D_i | i = 1, \dots, n\})$ time if $\sum_{i=1}^n \frac{C_i}{T_i} < 1$.*

Proof.

Let R be a synchronous general deadline task system with $U = \sum_{i=1}^n \frac{C_i}{T_i} < 1$. From Theorem 4.48, R is feasible iff for all $0 \leq t < t' \leq O^{max} + 2P = 2P$, $\sum_{i=1}^n \eta_i(t, t') \cdot C_i \leq t' - t$. We will show that if there exist t and t' such that this inequality does not hold, then t may be chosen to be 0, and t' may be chosen to be less than

$$\frac{U}{1-U} \max\{T_i - D_i | i = 1, \dots, n\}.$$

Remark that if $T_i = D_i \forall i$, i.e., in the late deadline case, the theorem is equivalent to Theorem 4.13: the condition $U < 1$ is sufficient.

For the family of systems such that $U \leq c < 1$, the above value is linear in $\max\{T_i - D_i | i = 1, \dots, n\}$.

We first show that t may be chosen to be 0. Suppose we have $\sum_{i=1}^n \eta_i(t, t') \cdot C_i > t' - t$. From the definition of $\eta_i(t, t')$, we have that for $1 \leq i \leq n$,

$$\begin{aligned} \eta_i(0, t' - t) &= \max \left\{ 0, \left\lfloor \frac{t' - t - D_i}{T_i} \right\rfloor + 1 \right\} \\ &\geq \max \left\{ 0, \left\lfloor \frac{t' - D_i}{T_i} \right\rfloor - \left\lceil \frac{t}{T_i} \right\rceil + 1 \right\} \\ &= \eta_i(t, t'). \end{aligned}$$

Consequently, $\sum_{i=1}^n \eta_i(0, t' - t) \cdot C_i \geq \sum_{i=1}^n \eta_i(t, t') \cdot C_i > t' - t$.

We will now show that if $t = 0$, then t' cannot be too large. Suppose $\sum_{i=1}^n \eta_i(0, t') \cdot C_i > t'$ and $t' \geq D_i - T_i$. We have

$$\begin{aligned}
 t' &< \sum_{i=1}^n \eta_i(0, t') \cdot C_i \\
 &= \sum_{i=1}^n \left(\left\lfloor \frac{t' - D_i}{T_i} \right\rfloor + 1 \right) C_i \\
 &\leq \sum_{i=1}^n \frac{t' + T_i - D_i}{T_i} C_i \\
 &= \sum_{i=1}^n \left(\frac{t' \cdot C_i}{T_i} + \frac{(T_i - D_i)C_i}{T_i} \right) \\
 &\leq U \cdot t' + U \cdot \max\{T_i - D_i\}.
 \end{aligned}$$

Solving for t' , we get

$$t' \leq \frac{U}{1 - U} \max\{T_i - D_i\}.$$

■

From this property, the authors define an algorithm which consists in computing $\sum_{i=1}^n \eta_i(0, t)$ for increasing t 's while the condition is satisfied and $t \leq \frac{c}{1-c} \max\{T_i - D_i | i = 1, \dots, n\}$. For the family of systems such that $U \leq c < 1$ the maximal time complexity of this algorithm is pseudo-polynomial in terms of the task characteristics. Remark that the ratio $\frac{c}{1-c}$ is a constant, for this reason only the term $\max\{T_i - D_i | i = 1, \dots, n\}$ is considered in the analysis of Baruah, Howell and Rosier, but this constant can be very large in comparison with the term $\max\{T_i - D_i | i = 1, \dots, n\}$, and can even be larger than P .

We shall now compare the actual time complexity of the feasibility test of Baruah et al and the one issued from Theorem 4.37 for synchronous systems. We have implemented the computations of Baruah et al and ours, based on the computation of the response time of all requests in the interval $[0, L)$, and we have compared their performances on randomly chosen systems (feasible or unfeasible). n was chosen randomly in the interval $[20, 100]$, the periods in the interval $[50, 1000]$, the deadlines (e.g., D_i) in the interval $[\frac{T_i}{2}, 2 \cdot T_i]$ and the computation times in order to have an utilization in $[0.45, 0.97]$. Figures 4.21 and 4.22 show the distribution of the actual time of the Baruah et al method and ours respectively (we have observed similar behaviors for different variations domains for n and U). It may be also noticed that, in the Baruah et al method the minimal value is greater than 200. Figures 4.23 and 4.24 show the

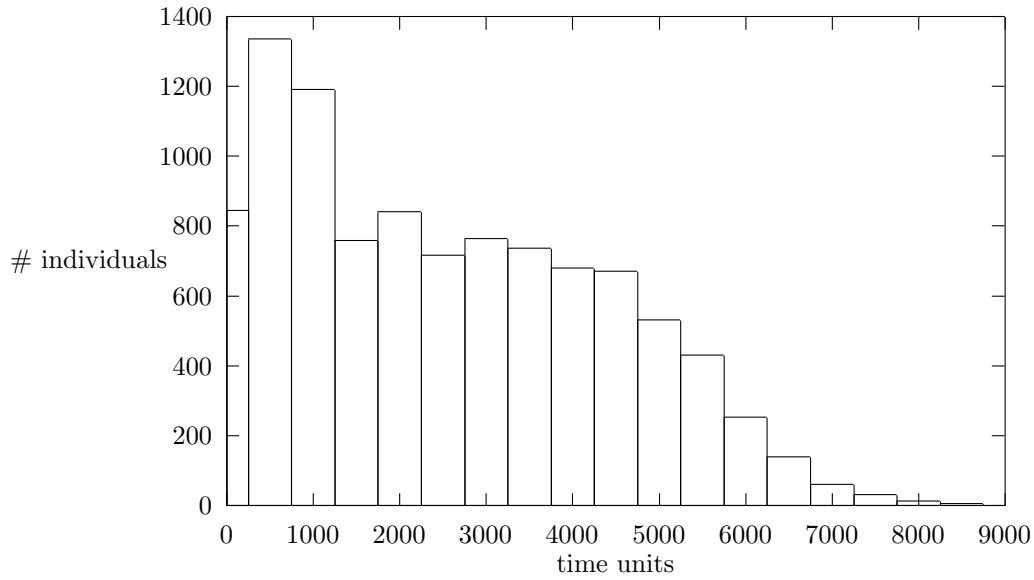


Figure 4.21: Frequency of the time complexity of the Baruah et al method.

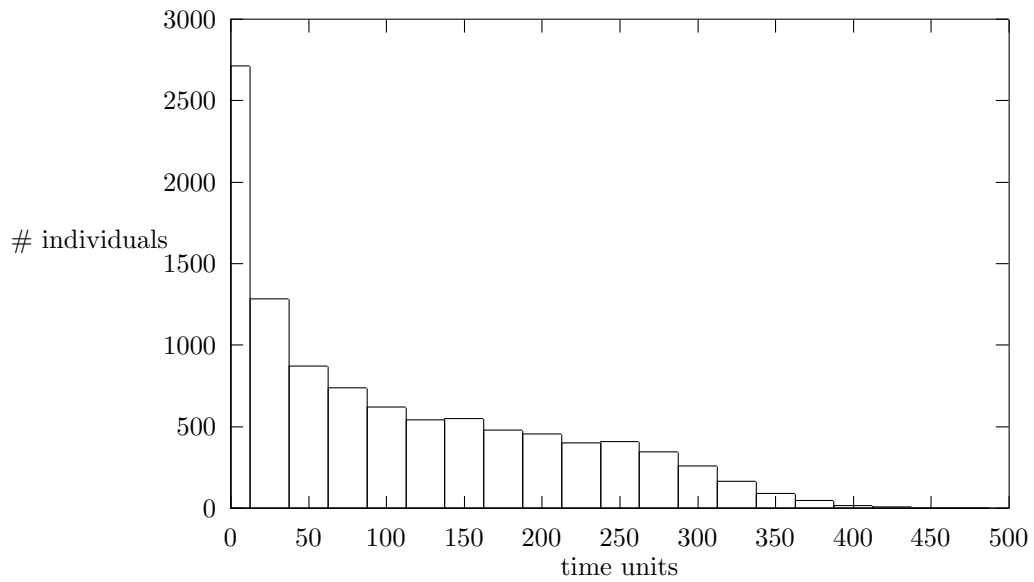


Figure 4.22: Frequency of the actual time complexity of our method.

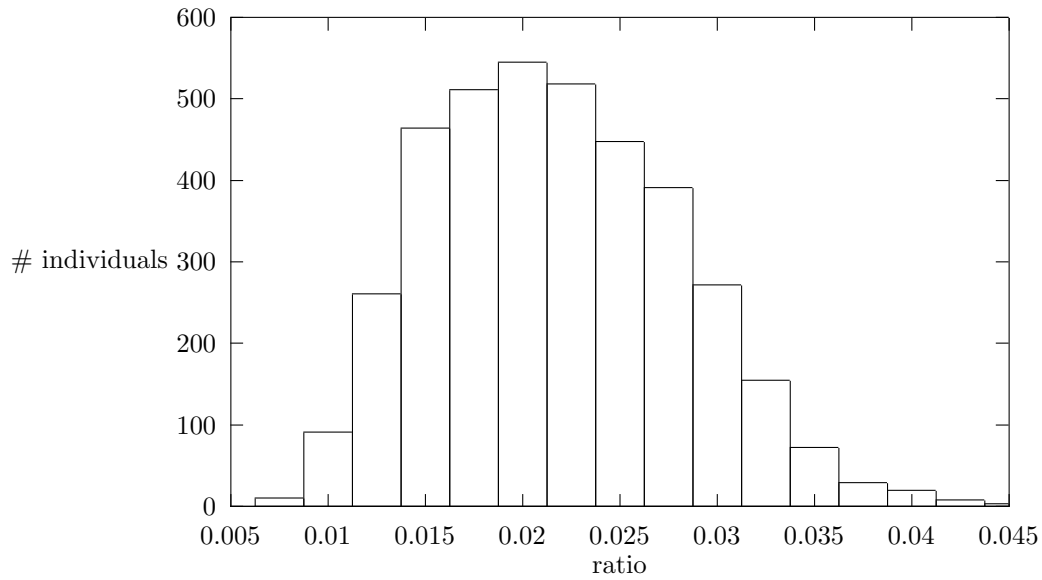


Figure 4.23: Frequency of the ratio between our method and Baruah et al's for $n \in [30, 60]$; the standard deviation is 0.006.

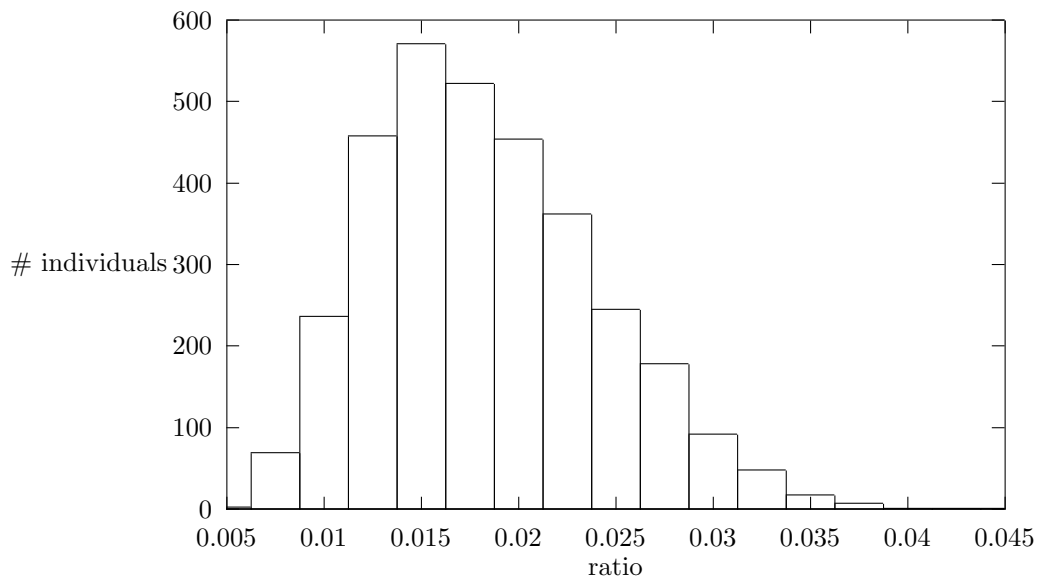


Figure 4.24: Frequency of the ratio between our method and Baruah et al's for $U \in [0.7, 0.9]$; the standard deviation is 0.006.

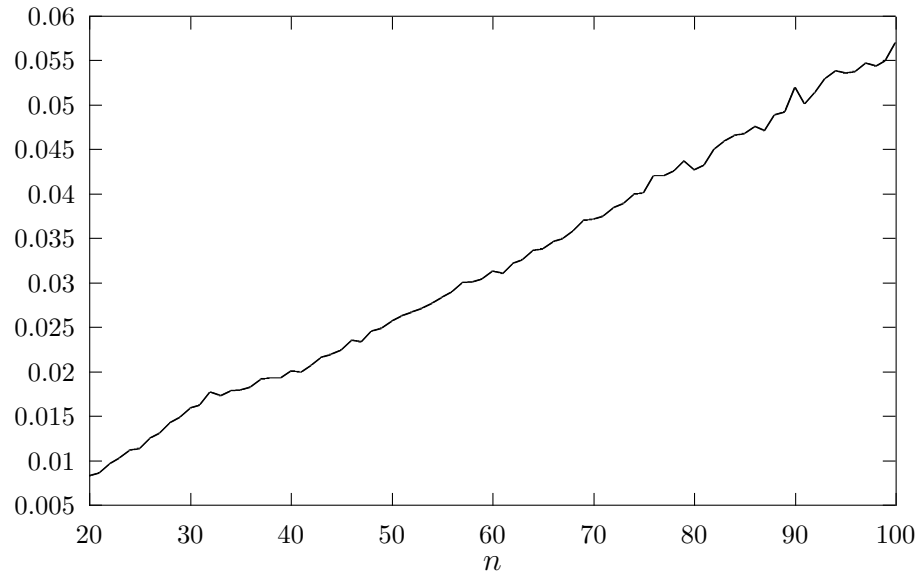


Figure 4.25: Actual time complexity of our approach in comparison with Baruah et al's, in function of n .

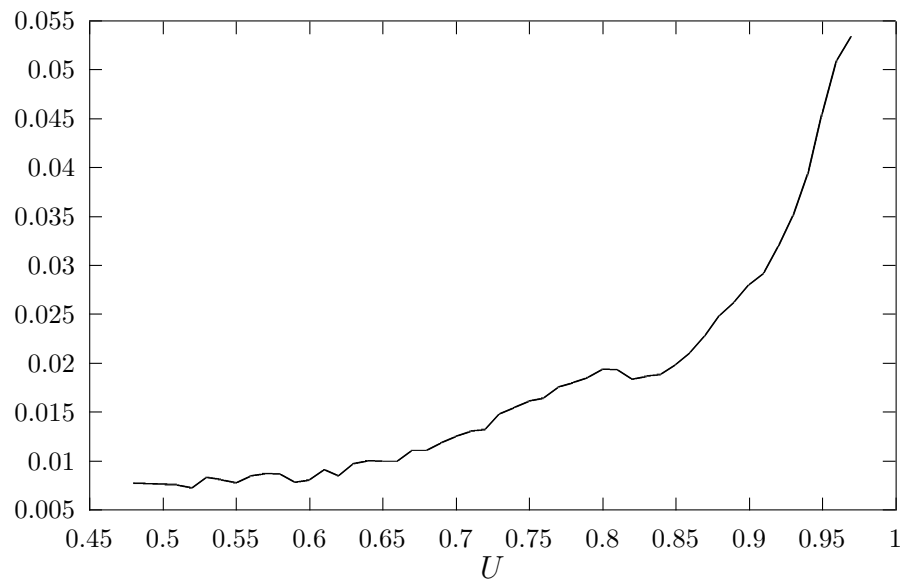


Figure 4.26: Actual time complexity of our approach in comparison with Baruah et al's, in function of U .

distribution of the ratio between the actual time of the Baruah et al method and our for $n \in [50, 60]$ and $U \in [0.7, 0.9]$ respectively. Figure 4.25 shows the ratio between the actual CPU time complexity of our method and Baruah's, in function of the number of tasks. The ratio increases with n but remains in all cases less than 0.06, which shows the benefit of our approach. The same phenomenon is described with Figure 4.26, but in function of the utilization factor: the ratio increases with U but remains in all cases less than 0.06.

In recent works [Spu96], another approach was adopted in order to check the feasibility of synchronous systems, based on the worst response time computation; this is the aim of the next section.

4.8.2 Worst case response time computation

Recently, Spuri [Spu96] has extended the computation of the worst case response time, already considered for static priority rules (see sections 3.2 and 3.4, see also Theorem 2.37), to the dynamic case (more exactly to the deadline driven scheduler). It is important at this point to specify that the work of Spuri concerns arbitrary asynchronous (or offset free) systems, but its main result provides a necessary and sufficient condition for the feasibility of arbitrary and synchronous systems. We shall also present here the main results concerning the corresponding feasibility test. We shall meanwhile lift some imprecisions in the work of Spuri. Let us recall that all results given by Spuri concern the arbitrary deadline situation.

Spuri does not assume any particular way for breaking deadline ties and considers the worst case, i.e., the request δ_i^k of the considered task τ_i has a higher priority than δ_j^p iff $R_i^k + D_i < R_j^p + D_j$. Remark that this is rather pessimistic and leads to a difficulty if we want to consider the worst case for all the tasks simultaneously.

Definition 4.53 For a task τ_i , we define the *worst case response time* (say r_i) as the largest response time of τ_i which occurs in the schedule of the synchronous system. ■

The computation of the worst case response time is interesting from a schedulability point of view, since the worst case response time is less than or equal to the deadline ($r_i \leq D_i$) iff the system is schedulable in the synchronous case.

Finding r_i is not trivial with the deadline driven scheduler, since the worst case response time does not always occur in the first busy period, while the latter is a feasibility interval (see section 4.5.1 for details).

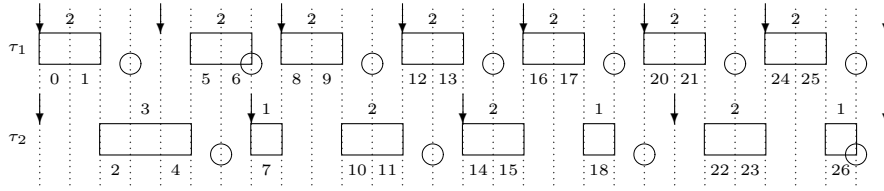


Figure 4.27: The worst case response time does not always occur in the first busy period.

Example 4.54 Consider for example the synchronous system $S = \{\tau_1 = \{C_1 = 2, D_1 = 3, T_1 = 4\}, \{\tau_2 = \{C_2 = 3, D_2 = 6, T_2 = 7\}\}$. The length L of the first busy period is in this case 7, the response time of the first request of τ_2 is 5 while the response time of the 4th request of τ_2 , starting at time 21, is 6 (see Figure 4.27). ■

However the busy period concept still has an interest in this case: Spuri has “stated” that the worst case response time occurs in the busy period of a request δ_i^k (see Definition 4.55) so that all other tasks are released at the beginning of the period. But this property was not really proved in his paper, so that his argument is not fully satisfactory; we shall complete here the proof.

The results of Spuri are based on a adaptation of the first busy period: *the busy period for a request δ_i^k* , but the author has not defined explicitly this concept. We shall consider this point here for completeness. Our definition is based on the concept of the elementary busy period, already defined in Chapter 2, but instead of limiting the schedule to a task sub-set $\{\tau_1, \dots, \tau_i\}$ we shall limit the schedule to higher priority requests:

Definition 4.55 We define *the busy period of the request of δ_i^k* as the elementary busy period $[a, b)$ in the schedule of all the requests with a higher (or equal) priority than δ_i^k , which includes the request δ_i^k (i.e., $a \leq R_i^k < b$). ■

From this definition it follows that if $[a, b)$ is the busy period of the request δ_i^k , b is the completion time $R_i^k + \rho_i^k$ of the request δ_i^k and a is the arrival time of a request $\delta_j^h \succeq \delta_i^k$. Definition 4.55 can be understood as the extension to the dynamic case of the level- i busy period considered in Chapter 2.

Theorem 4.56 *The worst case response time of a request of τ_i occurs at the end of the busy period of a request of τ_i in which all other tasks are released asap, i.e., they are released synchronously at the beginning of the period.*

Proof. Consider an asynchronous system and a request of τ_i (say the k^{th}) with arrival time¹⁷ R_i^k and deadline $d = R_i^k + D_i$. Let $[t_1, t_2)$ be the busy period of the request δ_i^k . Since $[t_1, t_2)$ is the busy period of the k^{th} request of τ_i , the requests of τ_j ($j = 1, \dots, n$) which occur strictly before t_1 either have a lower priority than δ_i^k or are completed not later than t_1 ; in both cases, they do not impact on the request δ_i^k nor on higher priority ones occurring after or at time t_1 . Hence, since we are only interested in the response time of δ_i^k , without loss of generality we can only consider the system from time t_1 , change the time origin ($t'_1 = 0, t'_2 = t_2 - t_1$), assume that each task τ_j makes its first request at time $\Delta_j \geq 0$, consider the response time of the request $\delta_i^{k'}$ ($k' = k - q$, where q is the number of requests of task τ_i which occur in the interval $[0, t_1)$) with a deadline $d' = R_i^{k'} + D_i$. We shall now prove that if $\Delta_j > 0$ (for $j \neq i$), reducing Δ_j , i.e., considering that the first request of τ_j occurs at time $0 \leq \Delta'_j < \Delta_j$ does not decrease the response time of the request $\delta_i^{k'}$. Since $[t'_1 = 0, t'_2)$ is the busy period of the request $\delta_i^{k'}$, we have:

$$\forall t \in (0, t'_2) \quad t < \sum_{j=1}^n k_j(\Delta_j, t) \cdot C_j$$

where $k_j(\Delta_j, t)$ represents the number of requests of τ_j in $[0, t)$ with a deadline less than or equal to d' (recall that, in case of a tie, $\delta_i^{k'}$ has the lower priority). Consequently we have:

$$\forall t \in (0, t'_2) \quad t < \sum_{j=1}^n k_j(\Delta'_j, t) \cdot C_j \quad (4.3)$$

since $k_j(\Delta'_j, t) \geq k_j(\Delta_j, t)$.

Since $[0, t'_2)$ is the busy period of the request $\delta_i^{k'}$, this one ends its computation at time t'_2 , and t'_2 is the smallest solution to the equation:

$$t = \sum_{j=1}^n k_j(\Delta_j, t) \cdot C_j.$$

We shall show that the request $\delta_i^{k'}$ completes its execution after or at time t'_2 with the starting times Δ'_j ($j \neq i$). Suppose the property false, the request of $\delta_i^{k'}$ ends its computation at time $t' < t'_2$ with

¹⁷Time a is not necessarily the time of the first request of τ_i , i.e., $k \geq 1$.

$$t' = \sum_{j=1}^n k_j(\Delta'_j, t') \cdot C_j$$

which contradicts Equation (4.3).

Hence, since $R_1^{k'}$ is not modified by the decrease of Δ_j , the largest response time is achieved by setting Δ_j to its smallest value: $\Delta_1 = \Delta_2 = \dots = \Delta_{i-1} = \Delta_{i+1} = \dots = \Delta_n = 0$, i.e., in an asap configuration. ■

This proof completes the original one, which did not formally prove that reducing Δ_j leads to increase (or leave unchanged) the response time of the request $\delta_i^{k'}$.

Theorem 4.57 ([Spu96]) *The length L_i^k of the busy period of any request δ_i^k of τ_i in an asynchronous system is less than or equal to the length L of the first busy period in the synchronous case.*

Proof. Consider the busy period $[a, b)$ for the request δ_i^k . We have:

$$\begin{aligned} \forall t \in (a, b) \quad t - a &< \sum_{j=1}^i k_j(\Delta_j) \cdot C_j \\ &< \sum_{j=1}^n \left\lceil \frac{t-a}{T_j} \right\rceil C_j. \end{aligned}$$

where $k_j(\Delta_j)$ represents the number of requests of τ_j in $[a, t)$ with a deadline less than or equal to $R_i^k + D_i$, which is bounded by the maximum number of requests of τ_j started in an interval of length $t - a$, i.e., $\lceil \frac{t-a}{T_j} \rceil$. The length of the first busy period in the corresponding synchronous case is the smallest solution L to the Equation (2.1)

$$L = \sum_{j=1}^n \left\lceil \frac{L}{T_j} \right\rceil C_j$$

so that $L \geq (b - a)$. ■

We present here the computation of Spuri; however, we shall use a slightly different notation for convenience and uniformity of this work.

The author suggests to compute the response time of all possible requests of τ_i in the situation defined by Theorem 4.56, i.e., all tasks but τ_i are released

at time $t = 0$, task τ_i makes a new request at time a (say the k^{th} request of τ_i) and $[0, a + \rho_i^k)$ is the busy period of the request δ_i^k .

From Theorem 4.57 we can restrict the values of a as follows: $0 \leq a \leq L - C_i$, but a is not necessarily the time of the first request of τ_i ; it may be preceded by other requests of τ_i , the first one occurring at time

$$s_i(a) = a - \left\lfloor \frac{a}{T_i} \right\rfloor T_i = a \pmod{T_i}.$$

The computation of the response time $r_i(a)$ of the request δ_i^k of τ_i is based on the computation of the length $L_i(a)$ of the *first busy period* of a limited schedule: $L_i(a)$ is the first idle point (beside 0) from time $t = 0$ by considering only requests with a deadline less than or equal to $a + D_i$.

Hence, $L_i(a)$ is the smallest positive solution ($L_i(a) > 0$) of the equation:

$$L_i(a) = \sum_{j=1}^n k_j(L_i(a)) C_j$$

where $k_j(t)$ is the number of requests of τ_j which occur in the interval $[0, t)$ with a deadline less than or equal to $a + D_i$; $k_j(t) = \min \left(\left\lfloor \frac{(a+D_i+T_j-D_j)^+}{T_j} \right\rfloor, \left\lfloor \frac{t}{T_j} \right\rfloor \right)$ and $k_i(t) = \left\lfloor \frac{(t-s_i(a))^+}{T_i} \right\rfloor$.

The minimal solution can be found by iteration:

$$L_i^{(0)}(a) = \sum_{j \neq i, D_j \leq a + D_i} C_j + \begin{cases} C_i & \text{if } s_i(a) = 0, \\ 0 & \text{otherwise} \end{cases} \quad (4.4)$$

$$L_i^{(k+1)}(a) = \sum_{j=1}^n k_j(L_i^{(k)}) C_j. \quad (4.5)$$

The iteration is growing and proceeds until $L_i^{(k+1)}(a) = L_i^{(k)}(a) = L_i(a)$ or $L_i^{(k)}(a)$ exceeds $a + D_i$; in the latter case, the iteration may be stopped because τ_i is then deemed unschedulable (remark that the iterative process converges anyway, i.e., even if we continue the iteration in this case).

If $L_i(a) \leq a$ it follows that $[0, a + \rho_i^k)$ is not the busy period of the request of τ_i which occurs at time a , and from Theorem 4.56 we can take the convention that $r_i(a) = C_i$.

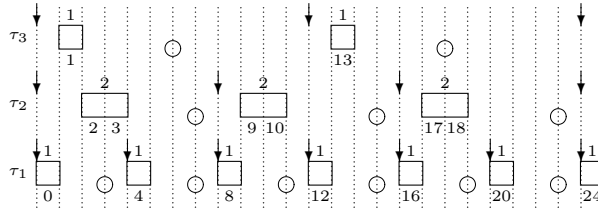


Figure 4.28: The synchronous schedule (from time $t = 24$ it repeats).

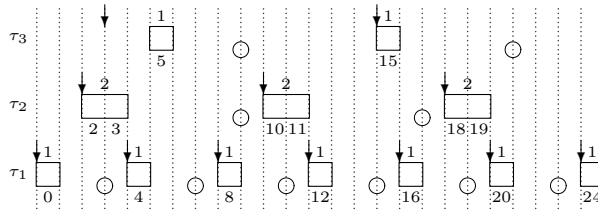


Figure 4.29: An asynchronous schedule (from time $t = 24$ it repeats).

If $a < L_i(a) \leq a + D_i$, the response time of the request of τ_i which occurs at time a is $L_i(a) - a$, otherwise the request of τ_i misses its deadline and we take the convention that $r_i(a) = \infty$. It follows that the worst response time $r_i = \max_{L-C_i \geq a \geq 0} \{r_i(a)\}$.

We have presented the results concerning the worst case response time issued from the work of Spuri [Spu96]. We shall now address two criticisms: the first one concerns the computation of the worst case response time itself and the second one concerns the complexity of this computation in comparison with our general response time computation.

Exact computation of the worst case response time:

The value of the worst response time r_i computed by the formulas of Spuri does not necessarily occur in the schedule of the synchronous case, and consequently does not match exactly with Definition 4.53.

Example 4.58 Consider the following system:

| | T_i | D_i | C_i | r_i |
|----------|-------|-------|-------|-------|
| τ_1 | 4 | 3 | 1 | 1 |
| τ_2 | 8 | 7 | 2 | 4 |
| τ_3 | 12 | 6 | 1 | 3 |

In the synchronous case, the largest response time of a request of τ_3 is 2 (see Figure 4.28) while the worst case response time of a request of τ_3 (given by the formulas of Spuri) is 3: the response time 3 only corresponds to asynchronous situations (e.g., $O_1 = 0, O_2 = 2, O_3 = 3$ – see Figure 4.29). It may be also noticed that the worst case response time occurs in asap situations as well as in other situations, as illustrated in Figure 4.29, for the first request of τ_3 , since this one is not an asap situation. ■

Contrary to our intuition the worst case response time does not necessary occur in the synchronous case, nor in the first busy period (see Example 4.54 while the synchronous case is the worst case regarding the feasibility of the system and $[0, L)$ is a feasibility interval for synchronous systems. We shall see why this is not contradictory. Regarding the feasibility of the system, from Corollary 4.3, an asynchronous system is feasible iff $\sum_{i=1}^n \eta_i(t, t') \cdot C_i \leq t' - t$ for all $0 \leq t < t'$, and it may be shown that the function $\frac{\sum_{i=1}^n \eta_i(t, t')}{t' - t}$, which combines characteristics of the various tasks, is maximum in the synchronous case (see Theorem 4.11) and for some $t \leq t' \leq L$ (see Theorem 4.10). The response time of a particular request depends on the relative priorities between the various requests, which depend on the request/deadline configurations. Since the synchronous case (or the interval $[0, L)$ in the synchronous case) does not necessary include all the configurations, it follows that the worst case response time for a particular request does not necessarily occur in the synchronous schedule (nor in the interval $[0, L)$ in the synchronous case).

Spuri computes in fact the worst case response time among all requests of τ_i and among all asynchronous situations.

On the other hand, all worst case response times r_1, r_2, \dots, r_n do not necessarily occur in the same schedule, i.e., in the same asynchronous situation.

Example 4.59 Consider the following system:

| | T_i | D_i | C_i | r_i |
|----------|-------|-------|-------|-------|
| τ_1 | 4 | 5 | 2 | 4 |
| τ_2 | 6 | 4 | 2 | 3 |

We shall show that the worst case responses of τ_1 and τ_2 never occur in the same asynchronous situation. A priori there is an infinite number of asynchronous situations. We shall show in Chapter 5 that we can consider only $\frac{\prod_{i=1}^n T_i}{\text{lcm}\{T_i | i=1, \dots, n\}}$ non-equivalent asynchronous situations; in our case there are 2 non-equivalent asynchronous situations given by (for instance): $\{O_1 = 0, O_2 = 0\}$, and $\{O_1 = 0, O_2 = 1\}$ (we shall not give more details here, this question is studied

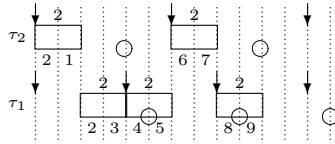


Figure 4.30: First (a)synchronous situation, from time $t = 12$ the schedule repeats.

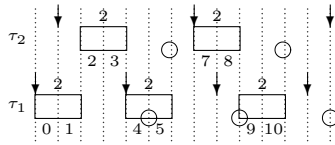


Figure 4.31: Second synchronous situation, from time $t = 12$ the schedule repeats.

extensively in Chapter 5). The largest response time for a request of τ_1 occurs in the first (a)synchronous situation (see Figure 4.30: $\rho_1^1 = 4 = D_1 - 1$) but does not occur in the second one (see Figure 4.31); symmetrically the largest response time of a request of τ_2 occurs in the second asynchronous situation ($\rho_2^1 = 3$) but does not occur in the first one. Remark that we have considered here an arbitrary deadline system, but it is not difficult to see that the phenomenon holds also for general deadline systems, since if in the previous system we subtract 1 from each D_i we get the same schedule since the relative priority ordering is not modified and the response times remain less than or equal to their deadline. ■

Although the value r_i computed by Spuri does not match Definition 4.53, the feasibility test based on the computations of Spuri is indeed necessary and sufficient for the feasibility of arbitrary and synchronous systems.

Theorem 4.60 *A synchronous system with arbitrary deadlines is feasible iff the worst case response time (computed by Spuri) of each task (τ_i) is less than or equal to the deadline: $r_i \leq D_i$ for $i = 1, \dots, n$.*

Proof. (*only if part*). If the system is schedulable in the synchronous case, it follows from Theorem 4.11 that this is also the case in all asynchronous situations (i.e., for all offset assignments); consequently the response time of any request of τ_i in any asynchronous situations respects the deadline and $r_i \leq D_i$.

(if part). We show the contraposition. Suppose the system is not schedulable in the synchronous case; there must exist i such that a request (say the k^{th}) of τ_i misses its deadline. It follows that its response time (say ρ_i^k) is greater than its deadline (D_i). Since r_i is the largest response time of a request of τ_i we have: $\rho_i^k \leq r_i$; it follows that $r_i > D_i$. ■

Corollary 4.61 *An offset free and arbitrary deadline system is schedulable in all asynchronous situations iff the worst case response time (computed by Spuri) of each task (τ_i) is less than or equal to the deadline: $r_i \leq D_i$ for $i = 1, \dots, n$.*

Proof. Immediately follows from Theorems 4.60 and 4.11. ■

We have seen in section 4.5.1 that $[0, L)$ is a feasibility interval for synchronous and arbitrary deadline systems; hence our general response time computation can be used in order to check the requests in the first busy period.

Theorem 4.62 *A synchronous system with arbitrary deadlines is feasible using the deadline driven scheduler iff*

$$\forall i : \max_{k \in \{1, \dots, \lceil \frac{L}{T_i} \rceil\}} \rho_i^k \leq D_i$$

with the deadline driven priority rule.

Proof. From Theorem 4.37 we have only to check the feasibility interval $[0, L)$. If the largest response time is less than or equal to the deadline this is also the case for all response times and each request of task τ_i meets its deadline in the feasibility interval. It follows that the system is schedulable. ■

Using our general response time computation we have defined a necessary and sufficient feasibility test for synchronous systems. Moreover, even in cases where Spuri computes the correct worst response times, the maximal time complexity of our test is an improvement in comparison with Spuri's. This is the purpose of our next criticism concerning the result of Spuri.

Complexity of the feasibility test

In view of the pessimism of the situations considered by Spuri, his method considers a too large number of situations: Theorem 4.62 shows that only the response times in the synchronous case and in the first busy period need be considered. It follows that the time complexity of the feasibility test of Spuri

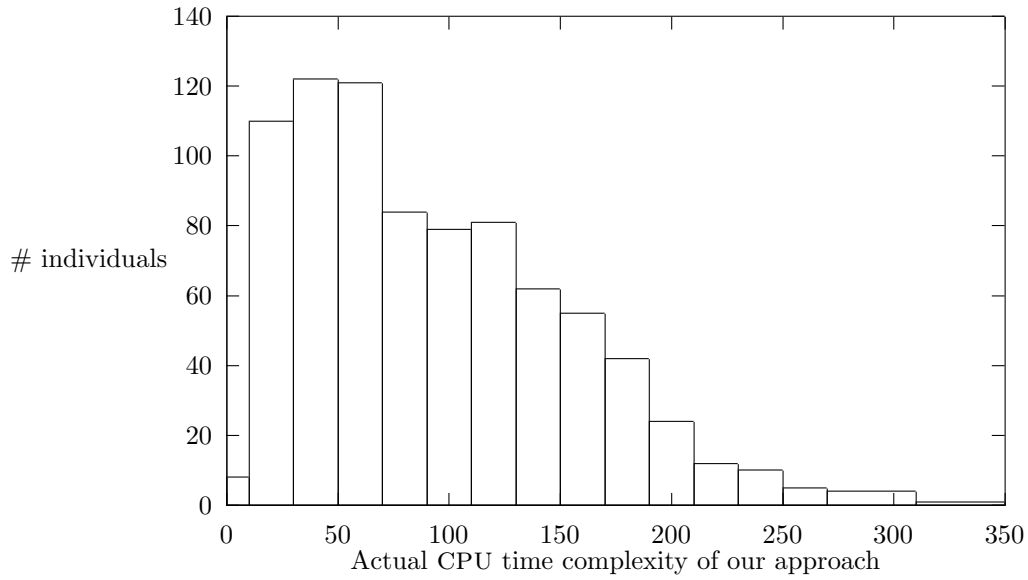


Figure 4.32: Frequency of the actual time complexity of our approach for feasible systems.

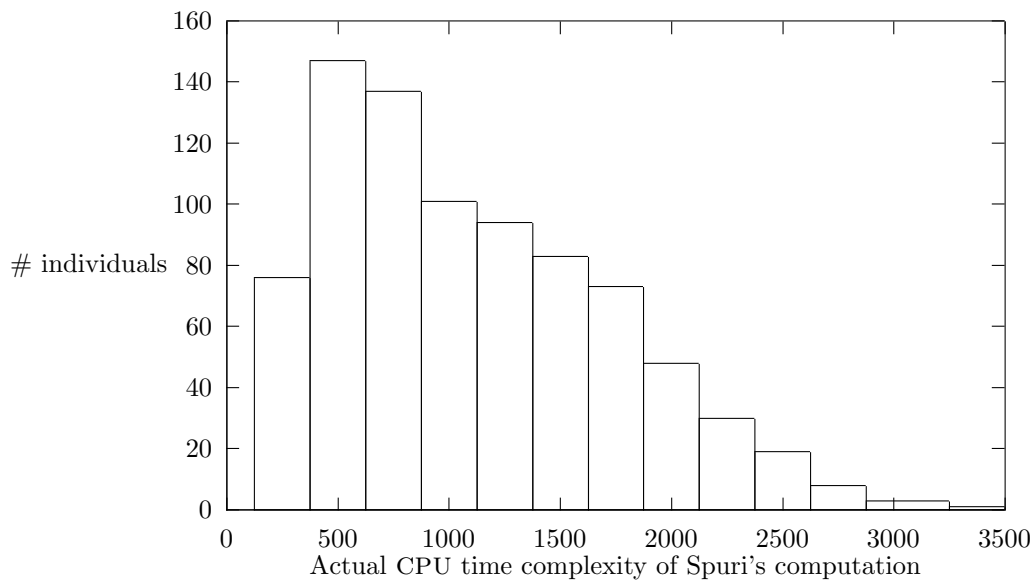


Figure 4.33: Frequency of the actual time complexity of Spuri's computation for feasible systems.

is too large. We shall now compare the time complexity of the feasibility test of Spuri, and the one issued from Theorem 4.62 for synchronous systems.

For a task τ_i ($i = 1, \dots, n$) the maximal time complexity of the computation of r_i given by Spuri's formulas is¹⁸ $O(L^2)$, since the iterative process given by Equations (4.4) and (4.5) must be applied at most L times ($a \in [0, L - C_i]$) and each iterative process converges after a maximum of $\frac{L}{\min_{j=1, \dots, n} C_j}$ steps (this bound is probably pessimistic for similar reasons than the ones utilized for our iterative process which computes the response time). For the whole task system, the maximal time complexity is thus $O(n \cdot L^2)$. The maximal time complexity of the computation of the response times of the requests which occur in the interval $[0, L)$ using method 2 is $O(\sum_{i=1}^n \frac{L}{T_i} \cdot n \cdot \frac{\max\{D_j - C_j | j=1, \dots, n\}}{\min\{C_j | j=1, \dots, n\}})$ (see section 4.6.6 for details); this complexity exhibits an exponential improvement in comparison with Spuri's.

In order to check if the actual performances of these two algorithms follow the same shape than the worst cases, we have implemented the computations of Spuri and ours, based on the computation of the response time of all requests in the interval $[0, L)$, and we have compared their performances on randomly chosen systems on a large set of simulations. We have applied this experimentation on randomly chosen task sets; n was chosen randomly in the interval $[20, 100]$, the periods T_i in the interval $[50, 1000]$, the deadlines D_i in the interval $[\frac{T_i}{2}, \frac{4T_i}{3}]$ and the computation times C_i in order to have a large utilization factor (i.e, near 1). We shall distinguish between feasible and unfeasible systems in the presentation of our simulation results, we shall see that indeed the performance of both algorithms are very related to the feasibility of the system.

We consider first feasible systems, and as usual we consider first the distributions: Figures 4.32 and 4.33 show the frequency of the actual time complexity of our approach and the Spuri's computation, respectively, for $n \in [50, 60]$. Figure 4.34 shows the distribution of the average ratio between the actual time complexity of our approach and those of Spuri. From the latter, it seems judicious to compare the average of both distributions, or to compute the average ratio. We have observed the same behavior for other variation domains of n in our simulations. Figure 4.35 shows the average ratio between the actual CPU time complexity of our method and Spuri's, in function of the number of tasks. The average ratio increases with n but remains in all cases below 0.095, which shows the actual pessimism of the Spuri's approach and the benefit of our approach.

¹⁸Of course it is possible to improve the computation suggested by Spuri, e.g, by using the computation of $L_i(a)$ in the one of $L_i(a + T_i)$, but we do not consider this point here, since we propose a (by far) better and simpler method.

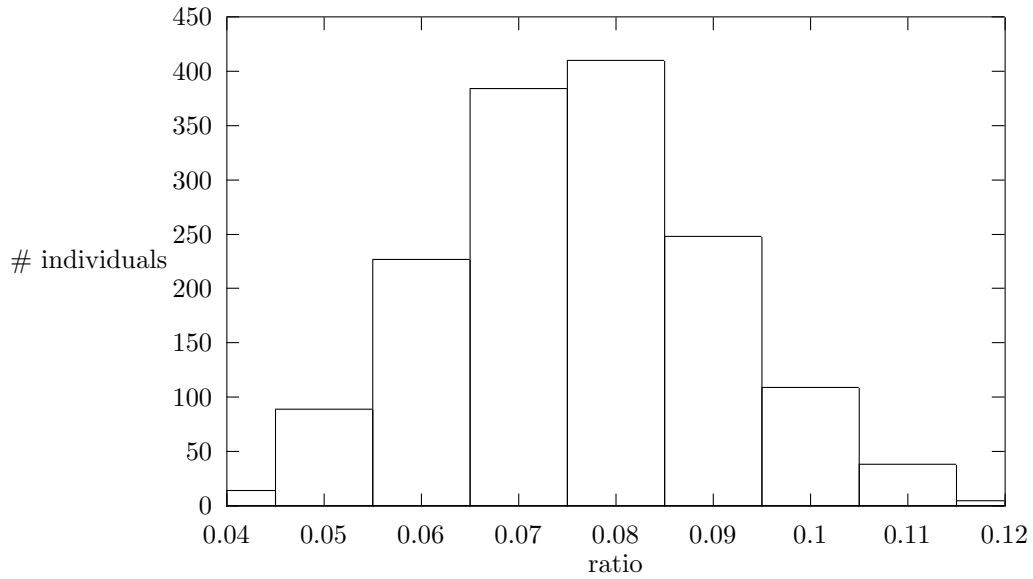


Figure 4.34: Frequency of the ratio for feasible systems.

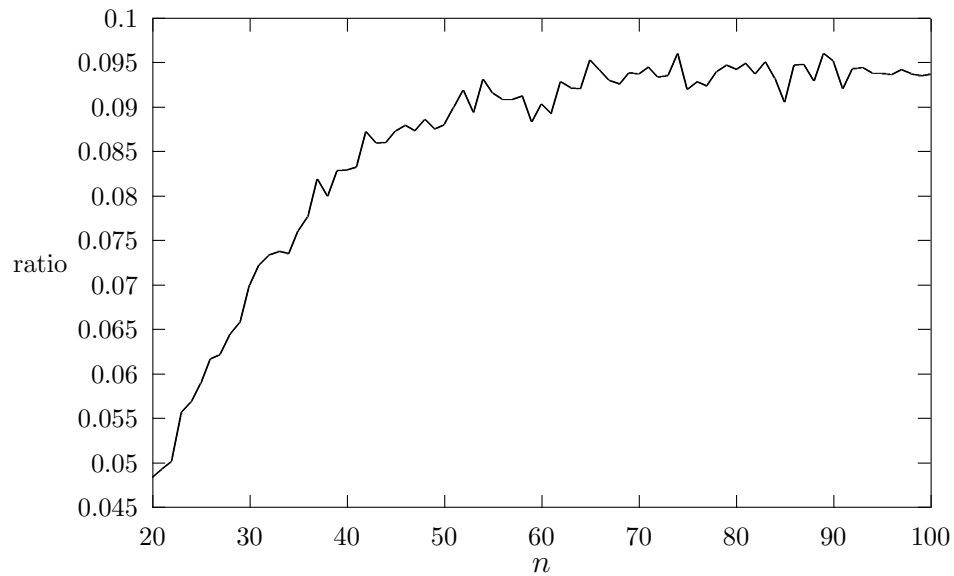


Figure 4.35: Actual CPU time complexities of our approach in comparison with Spuri's.

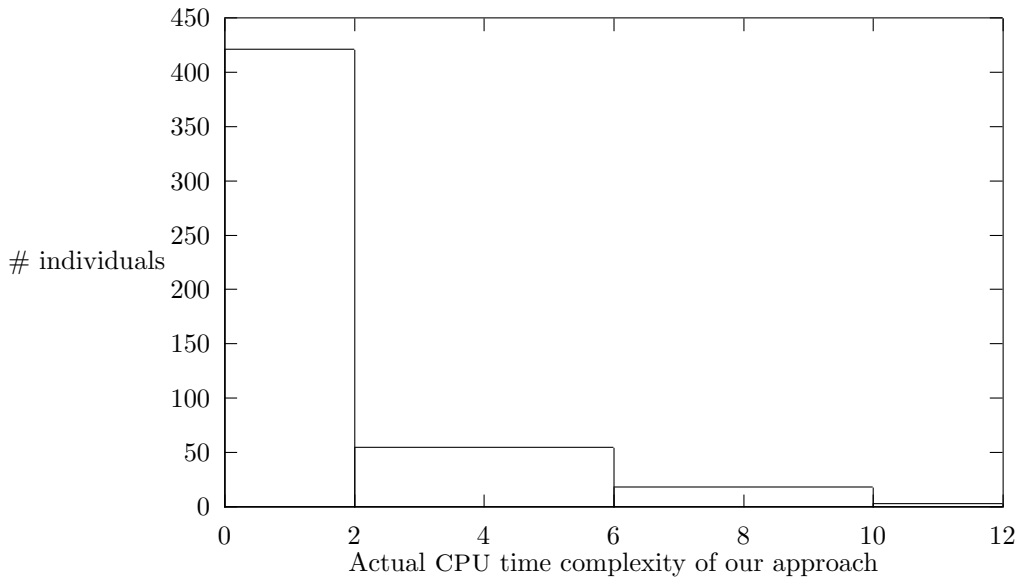


Figure 4.36: Frequency of the actual time complexity of our approach; for unfeasible systems.

Consider now the case of unfeasible systems. Figures 4.36 and 4.37 show the frequency of the actual time complexity of our approach and the Spuri's computation, respectively (for $n \in [50, 60]$). Figure 4.38 shows the distribution of the ratio between the actual time complexity of our approach and the one of Spuri. In this case both algorithms work much more quickly, and in a large proportion of cases, the algorithms have a comparable time complexity. Figure 4.39 shows the average ratio in function of n . The ratio depends less clearly on n , but the average ratio is in the interval $[0.2, 0.8]$, which shows again the (slight) actual improvement of our approach.

For static and general deadline systems, we have exhibited the situation which leads to the best case response time; for dynamic schedulers the problem seems much more difficult: recall that the computation of the best response time is based on the (dual) property which states that the worst case response time occurs for the first request and in the synchronous case (for all tasks); this property does not hold for dynamic schedulers since the worst response time occurs in asynchronous situations and not necessarily for the first request for each task. The determination of the situation which leads to the best response time is a relevant problem, however, and remains for further research.

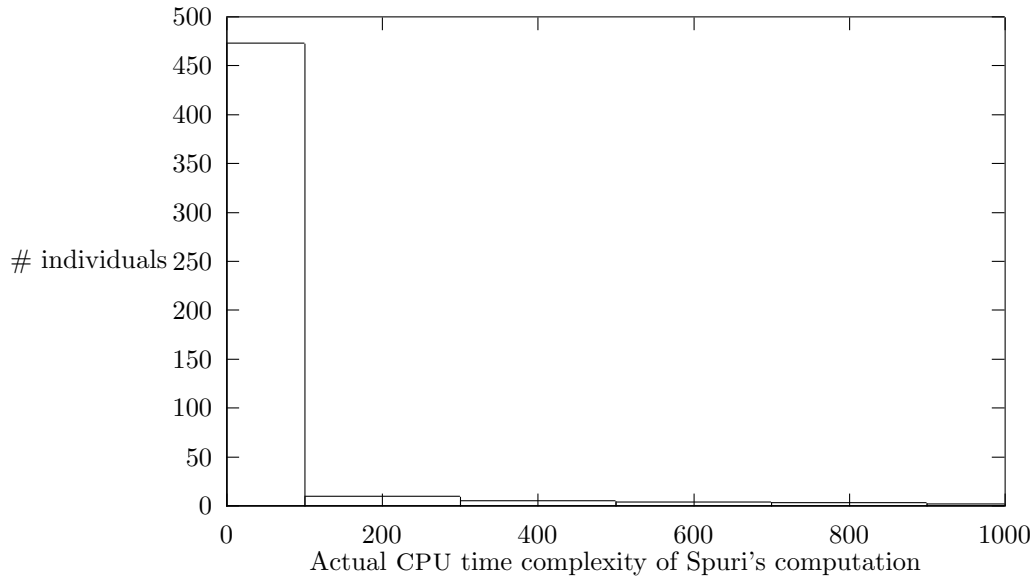


Figure 4.37: Frequency of the actual time complexity of Spuri's computation; for unfeasible systems.

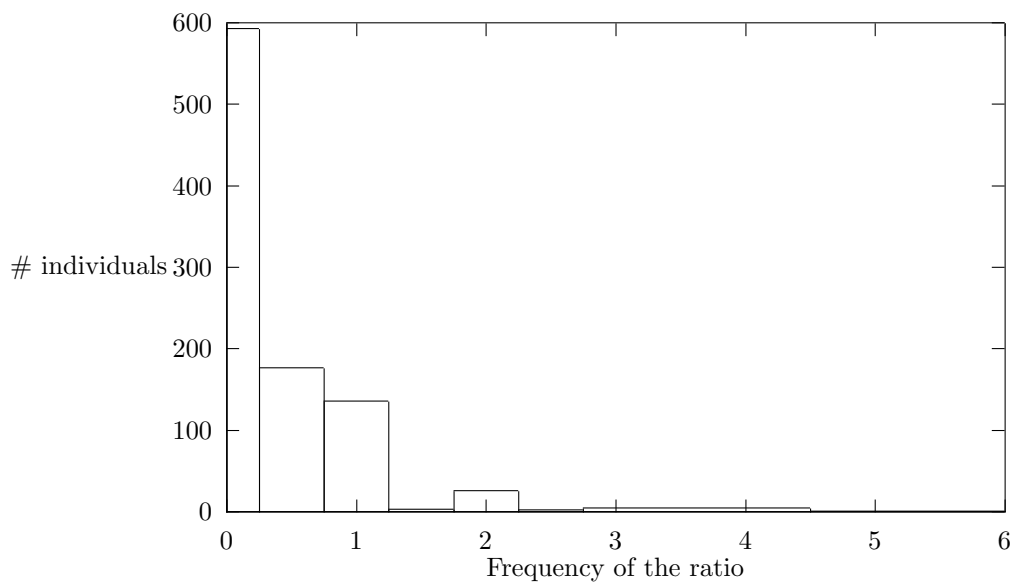


Figure 4.38: Frequency of the ratio; for unfeasible systems.

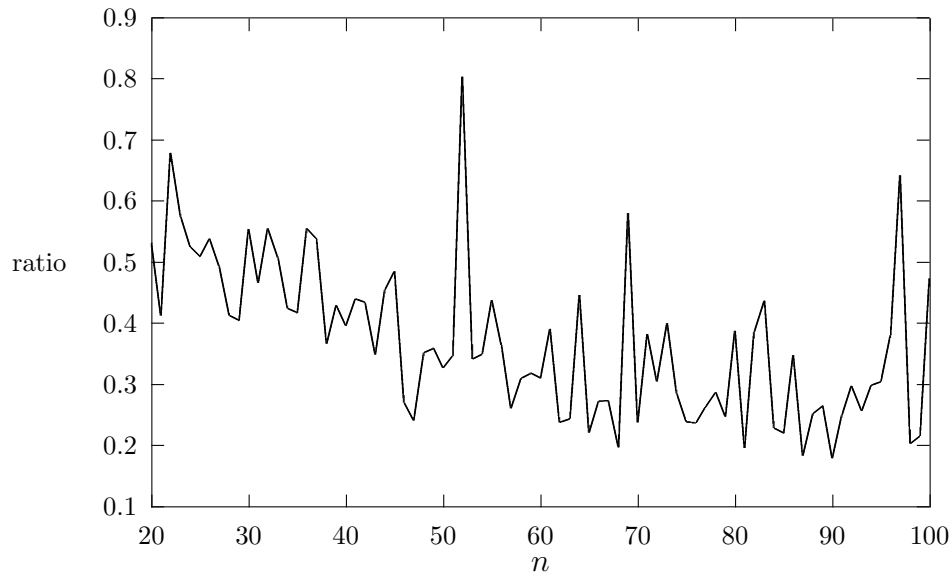


Figure 4.39: Actual CPU time complexities of our approach in comparison with the Spuri's computation; unfeasible systems.

4.9 The Least Laxity First scheduling algorithm

We have considered in this chapter dynamic priority algorithms, particularly the most popular dynamic scheduling algorithm: the deadline driven scheduler. We shall now consider another optimal dynamic scheduler for asynchronous and arbitrary deadline systems: the *least laxity first* (LLF) algorithm. This dynamic priority rule was basically defined by Mok and Dertouzos [MD78] and termed the least laxity first algorithm; in the recent literature [Mok83, Leu89] the term *slack-time algorithm* denotes the same scheduler. We shall use in this work the original term (i.e., the least laxity first) to denote this scheduler. The *laxity* of a request is defined as the maximal amount of time that the request can wait and still meets its deadline. More precisely, at time $t \geq R_i^k$ we define $e_{i,k}^{(t)}$ as the remaining processing time of the request δ_i^k and \mathcal{D}_i^k as its deadline instant ($\mathcal{D}_i^k = R_i^k + D_i$ and $e_{i,k}^{(t)} = C_i - \epsilon_i^k(t)$); the laxity at time t for the request δ_i^k (say $\lambda_{i,k}^{(t)}$) is $\mathcal{D}_i^k - t - e_{i,k}^{(t)}$. Hence, the laxity of a request is a measure of its urgency. The least laxity first scheduler gives the CPU to the active request with the smallest laxity. The tie may be broken arbitrarily (as exhibited by Theorem 4.65). In this work we break the tie by giving a higher priority to the task with the smaller index; if two requests of the same task are

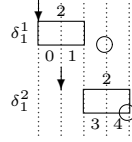


Figure 4.40: Two requests of the same task with the same laxity.

active simultaneously and have the same (smallest) laxity, the tie is broken by applying the first-in-first-out rule if the index is the smallest with this laxity.

Example 4.63 Consider the system composed by a single task $\tau_1 = \{T_1 = 1, D_1 = 3, C_1 = 2, O_1 = 0\}$; Figure 4.40 shows that, at time $t = 1$, two requests of τ_1 are active and have the same laxity, i.e., $\lambda_{1,1}^{(1)} = \lambda_{1,2}^{(1)} = 1$ and the CPU is given to the first request of τ_1 ■

However, this can only occur if $C_i > T_i$, which implies unfeasibility; otherwise, the relative priorities between successive requests of a same task respects the FIFO rule:

Theorem 4.64 *If the requests δ_i^k and δ_i^{k+1} are active at time t and $C_i \leq T_i$, then $\lambda_{i,k}^{(t)} < \lambda_{i,k+1}^{(t)}$.*

Proof. We prove the property by contradiction and suppose that at time t the requests δ_i^k and δ_i^{k+1} are active and

$$\lambda_{i,k}^{(t)} \geq \lambda_{i,k+1}^{(t)}. \quad (4.6)$$

$\lambda_{i,k}^{(t)} = (k-1)T_i + D_i - t - e_{i,k}^{(t)}$ and $\lambda_{i,k+1}^{(t)} = kT_i + D_i - t - e_{i,k+1}^{(t)}$. From Equation (4.6) we get

$$T_i \leq e_{i,k+1}^{(t)} - e_{i,k}^{(t)} = \epsilon_i^k(t) - \epsilon_i^{k+1}(t)$$

which contradict the fact that $C_i \leq T_i$ since $0 \leq \epsilon_i^k(t) < C_i$ and $0 \leq \epsilon_i^{k+1}(t) < C_i$ when both requests are active. ■

By definition, the laxity of an executing request is constant and the laxity of a request waiting for CPU has a linearly decreasing laxity in function of the waiting time. Remark also that, if at time t a request is active and its laxity is negative, the deadline is missed before or at time t ; if the laxity of a request reaches 0, the latter gets an absolute priority (nothing else may be

scheduled till the completion of the request), but another request with a null laxity (which leads to a deadline failure).

The least laxity first rule is optimal for independent, periodic, asynchronous and arbitrary deadline systems.

Theorem 4.65 *The least laxity first algorithm is strongly optimal for asynchronous systems with arbitrary deadlines.*

Proof inspired from [Mok83] Let us assume the existence of a scheduling algorithm A so that the system is schedulable with it. Let σ be the feasible schedule σ' produced by the scheduler A on the system. We shall show that the schedule produced by the least laxity first algorithm is also feasible, for any particular way to resolve ambiguities. We shall show the following property by induction on t : we can always transform the schedule σ in such a way that in the interval $[0, t)$ the resulting schedule σ^t is same as σ' , and σ^t remains feasible for the same task set. This is trivially true for $t = 0$. Suppose the hypothesis holds for t . If $\sigma^t(t) = \sigma(t)$, we may take $\sigma^{t+1} = \sigma^t$. If at time t the CPU is idle in σ^t and $\sigma'(t) = (i, k)$ we can take $\sigma^{t+1}(t) = (i, k)$, $\sigma^{t+1}(e) = 0$, where e is the first instant after t such that $\sigma^t(e) = (i, k)$, and $\sigma^{t+1} = \sigma^t$ elsewhere. Remark that if at time t the CPU is not idle in σ' this is also the case in σ^t since there is no active request at time t in both schedules. Suppose that a request of task τ_i (say δ_i^k) is executing at time t in σ^t (i.e., $\sigma^t(t) = (i, k)$) while $\sigma'(t) = (j, p)$ with $\lambda_{j,p}^{(t)} \leq \lambda_{i,k}^{(t)}$.

Notice that the request δ_j^p must be executing during at least one time unit before \mathcal{D}_i^k , the current deadline of task τ_i in the feasible schedule σ^t . Otherwise, τ_j misses its deadline since its laxity at time \mathcal{D}_i^k is negative:

$$\begin{aligned} \lambda_{j,p}^{(\mathcal{D}_i^k)} &= \lambda_{j,p}^{(t)} - (\mathcal{D}_i^k - t + 1) \\ &\leq \lambda_{i,k}^{(t)} - (\mathcal{D}_i^k - t + 1) \\ &\leq \mathcal{D}_i^k - e_{i,k}^{(t)} - t - (\mathcal{D}_i^k - t + 1) \\ &= -e_{i,k}^{(t)} - 1 < 0. \end{aligned}$$

We can execute the request δ_j^p at time t (i.e., $\sigma^{t+1}(t) = (j, p)$) and the request of δ_i^k at time e_j (i.e., $\sigma^{t+1}(e_j) = (i, k)$), where e_j is the first time unit after time t in σ^t assigned to the request δ_j^p (this is a valid definition, since δ_j^p is still active at time t in σ^t). The resulting schedule remains feasible and in the interval $[0, t + 1)$ the schedule is the one produced σ' . Consequently, the feasible schedule σ can be transformed into the schedule σ' , while remaining feasible, and σ' is feasible. ■

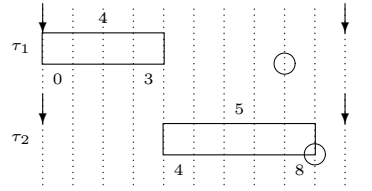


Figure 4.41: Schedule produced by the deadline driven scheduler (from time $t=10$, the schedule repeats).

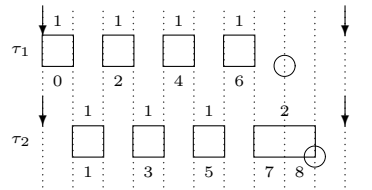


Figure 4.42: Schedule produced by the least laxity first (from time $t=10$, the schedule repeats).

The least laxity first and the deadline driven scheduler are both optimal dynamic priority rules, but the schedules produced by these algorithms are quite different, in particular if we focus our attention on the number of preemptions.

Example 4.66 Consider the following system:

| | C | T | D |
|----------|-----|-----|-----|
| τ_1 | 4 | 10 | 8 |
| τ_2 | 5 | 10 | 9 |

The schedule produced by the deadline driven scheduler contains no preemption (see Figure 4.41). On the contrary, the schedule produced by the least laxity first algorithm contains 6 preemptions every 10 time units (see Figure 4.42). ■

More generally suppose that at time t two active requests, say δ_i^k and δ_j^p have the same minimum laxity (i.e., $\lambda_{i,k}^{(t)} = \lambda_{j,p}^{(t)}$ and $\lambda_{q,r}^{(t)} > \lambda_{i,k}^{(t)} \forall (q,r) \notin \{(i,k), (j,p)\}$); suppose that the tie is broken in favor of δ_i^k ; at time $t + 1$ we have that $\lambda_{i,k}^{(t+1)} = \lambda_{j,p}^{(t+1)} + 1$, the scheduler preempts the request δ_i^k and gives the CPU to request δ_j^p ; at time $t + 2$ the laxity of both requests is identical again. Hence, we can reproduce this scenario while δ_i^k and δ_j^p are active and no

other higher priority request interferes. This trashing situation is cumbersome if we consider the switching times (up to now, we always considered them as negligible, but we know this is not exactly true, especially if there are many preemptions) and seems to be a drawback of this scheduling algorithm in comparison with the deadline driven scheduler. However, for the scheduling on multiprocessor systems, Leung [Leu89] has shown the superiority of the least laxity first rule in comparison with the deadline driven scheduler for a model quite close to ours. This superiority concerns the number of processors needed by the least laxity first rule in order to schedule the system. We shall not give more details on the subject here, since this concerns the scheduling on multiprocessor systems, which is not the purpose of this work.

Both priority rules are optimal, hence the following properties follow:

Lemma 4.67 *Let S be an asynchronous system with arbitrary deadlines. S is schedulable with the least laxity first rule iff S is schedulable with the deadline driven scheduler.*

Proof. Immediate according to the definition of the optimality (Definition 4.6) and the fact that these priority rules are both optimal. ■

Theorem 4.68 *For a given set of n asynchronous tasks with late deadlines, the least laxity first algorithm is feasible iff $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$.*

Proof. We prove the theorem by contradiction: suppose there is some late deadline system S with $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ which is not schedulable with the least laxity first rule. Since $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$ the system is schedulable with the deadline driven scheduler (see Theorem 4.11) and by Lemma 4.67 it follows that S is schedulable with the least laxity first rule, this is a contradiction and proves the theorem. ■

We have seen in section 4.6 the notion of response time and its interest. For the least laxity first rule, the computation of the response time, even in the synchronous case, seems difficult without a full simulation. This lack arises from the fact that, contrary to what happens with our deadline driven scheduler and with the static priority rules, the priority of a request changes with the time. It follows that the interference of requests of a task τ_j on the response time of a request of τ_i does not have a form of the kind: $n_j \cdot C_j$ ($n_j \in \mathbb{N}$) even if the system is synchronous. Indeed, τ_i may be delayed by a fraction of C_j with a strange looking form.

Example 4.69 Let us consider for instance the following synchronous system:

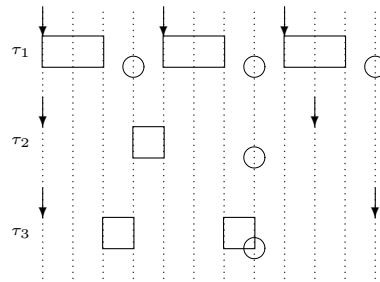


Figure 4.43: least laxity first schedule.

| | C | T | D |
|----------|-----|-----|-----|
| τ_1 | 2 | 4 | 3 |
| τ_2 | 1 | 9 | 7 |
| τ_3 | 2 | 11 | 7 |

the response time of the first request of τ_2 is 4 and the interference of task τ_3 is $\frac{C_3}{2}$ (see Figure 4.43). ■

Hence, the problem of computing the response time for the least laxity first rule seems tricky and remains open for future work. It may be noticed that the schedulability of a system with the least laxity first rule can be resolved without the knowledge of this response time. Indeed, from Lemma 4.67 it follows that we may simply check if the system is schedulable with the deadline driven scheduler, by checking the response time in the schedulability interval for the latter.

For the least laxity first rule, the synchronous case is again the worst case from a schedulability point of view.

Theorem 4.70 *Let S be a task set. If this task set is schedulable in the synchronous case with the least laxity first rule, then this task set is schedulable in all asynchronous situations with the least laxity first rule.*

Proof. We prove the theorem by contradiction: suppose there is some system S which is schedulable in the synchronous case while being not schedulable in an asynchronous situation with the least laxity first rule. Since the system S is schedulable in the synchronous situation with the least laxity first rule, by Lemma 4.67 again it follows that this is also the case with the deadline driven scheduler. By Theorem 4.11 we have also that S is schedulable in all asynchronous cases with the deadline driven scheduler. By Lemma 4.67 it follows that S is schedulable in all asynchronous cases with the least laxity first rule. This is a contradiction and proves the theorem. ■

4.10 The (non-)stability of dynamic priority rules

In this section we shall consider the (non-)stability of dynamic priority rules. We shall show that neither the deadline driven scheduler nor the least laxity first scheduler is stable.

First, we consider the case of the deadline driven scheduler.

Theorem 4.71 *The deadline driven scheduler is not stable.*

Proof. With the deadline driven scheduler there is necessarily a time instant t such that a non-critical request is active and has the “nearest” deadline among all active requests (say a deadline at time d ; notice that this deadline can be less than the current time instant since non-critical requests have a soft deadline). With the rule given by the deadline driven scheduler this non-critical request remains the highest priority one till the completion of its processing which is not bounded; consequently the schedulability of the critical requests from time t cannot be guaranteed. ■

In a similar way, we show that the least laxity first algorithm is not stable.

Theorem 4.72 *The least laxity first scheduler is not stable.*

Proof. With the least laxity first scheduler there is necessarily a time instant t such that a non-critical request is active with a minimal and negative laxity (e.g., if the computation time of this request exceeds its deadline); it can remain active and the highest priority request as long as desired; consequently, the feasibility of active and critical requests cannot be guaranteed. ■

In all generality we cannot claim of course that dynamic priority schedulers are not stable: consider for instance the case of the (dynamic¹⁹) rate monotonic scheduler which is stable under some assumptions and/or period transformations (as exhibited in section 2.7). It could be interesting to investigate the existence of an optimal stable and dynamic priority rule; however this question remains open for future works.

4.11 Conclusion

In this chapter we have presented dynamic priority schedulers. We have first studied the deadline driven scheduler, reviewed the literature, and we have

¹⁹Dynamic priority rules include static priority ones by definition.

completed/corrected the theory in particular concerning the optimality result. We have considered the results with arbitrary deadlines in mind. We have shown that several results remain for this more general class (than those generally considered in the literature). We have also extended the theory to this class, in particular for the feasibility interval given by Leung and Merrill. Then we have extended the computation of the response time to dynamic priority rules (i.e., formulas, iterative process and implementation methods) and for the various task sub-sets considered in this work, we have studied the interest of our general response time computation in comparison with previous results for synchronous and asynchronous systems (i.e., regarding the worst and the actual complexity of the various approaches). For both cases, we have shown the advantage of our approach.

First, for asynchronous systems we have shown that the maximal time complexity of our computation exhibits an exponential improvement in comparison with the computation suggested by Baruah, Howell and Rosier. For synchronous systems, we studied the worst case response time computation of Spuri, we have shown the pessimism of this approach, i.e., the value of the worst response time computed by Spuri does not necessarily occur in the schedule of the synchronous case. Moreover, we have shown that all response times computed by Spuri do not occur in the same schedule. Then we have shown the interest of our general response time computation to compute the worst case response time in a particular schedule, e.g., in the synchronous case. Lastly, based on the general response time computation we have defined a feasibility test for synchronous systems, which considers the system from time 0 till the first idle point (beside 0). The maximal time complexity of this test exhibits again an exponential improvement in comparison with the approach of Spuri.

Hence, we feel to have justified the interest of our general response time computation concerning the feasibility test of synchronous/asynchronous systems for arbitrary deadlines.

We have shown alongside our study that we must be very careful, that (our) intuition may lead to incorrect reasonings (e.g., it was the case of the incorrect argument used by Liu and Layland): for the kind of systems we consider in this work, even in very “simple” cases (e.g., synchronous and late deadline systems with 2 periodic tasks), it is difficult to anticipate their behavior.

Interesting questions for further research issued from this chapter include: statistical analysis of the actual performance of the various methods and algorithms proposed in this chapter with other random variables or with “real” systems; analysis of the parallelization of method 1 and the study of its time/space complexity; the determination of the situation which leads to the best response time; the computation of the response time using the least lax-

ity first algorithm; investigating the existence of optimal stable and dynamic priority rule,...

Bibliography

- [ABRT93] N. C. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.
- [BHR93] S. K. Baruah, R. R. Howell, and L. E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoret. Comput. Sci.*, 118:3–20, 1993.
- [BHR93] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoret. Comput. Sci.*, 1(118), 93.
- [BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *The Journal of Real-Time Systems*, 2:301–324, 1990.
- [Der74] M. Dertouzos. Control robotics: the procedural control of physical processes. In *Proceedings of the IFIP Congress*, 1974.
- [Dic19] Leonard Eugene Dickson. *History of the Theory of Numbers*, volume II. Chelsea Publishing Company, 1919.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, a guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
- [Knu69] Donald E. Knuth. *The Art of Computer Programming*, volume 2 of *Seminumerical Algorithms*. Addison-Wesley, 1969.
- [Lab74a] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps réel. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, B-1:11–17, février 1974.

- [Lab74b] Jacques Labetoulle. Some theorems on real time scheduling. *Computer Architectures and Networks*, 1974.
- [Leh90] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1990*, pages 201–213, Lake Buena Vista, Florida, USA, December 1990.
- [Leu89] Joseph Y.-T. Leung. An new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- [LM80] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.
- [LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [Mat81] L. Matthiessen. Le problème des restes dans l’ouvrage chinois swang-king de sun-tsze et dans l’ouvrage ta-yen-lei-schu de yih-hing. *Comptes rendus de l’Académie de Paris*, 92:291–294, 1881.
- [MD78] A. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Seventh Texas Conference on Computing Systems*, 1978.
- [Mok83] Aloysius Ka-Lau Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.
- [Spu96] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, 1996.

Chapter 5

Offset free systems

Douter de tout ou tout croire, ce sont deux solutions également commodes, qui l'une et l'autre nous dispensent de réfléchir.
— Henri Poincaré, *La Science et l'hypothèse* (Flammarion).

Contents

| | | |
|------------|--|------------|
| 5.1 | Introduction | 196 |
| 5.2 | Offset granularity | 199 |
| 5.3 | Non-equivalent asynchronous systems | 204 |
| 5.4 | Non-optimality of monotonic schedulers | 210 |
| 5.4.1 | Definitions and properties | 212 |
| 5.4.2 | Optimality in special cases of offset free systems | 213 |
| 5.4.3 | Non-optimality of monotonic schedulers | 217 |
| 5.5 | Optimality of dynamic schedulers | 221 |
| 5.6 | Practical interest of offset free systems | 221 |
| 5.7 | Optimal offset assignment | 224 |
| 5.7.1 | Two tasks | 226 |
| 5.7.2 | n tasks | 228 |
| 5.8 | Dissimilar offset assignment | 229 |
| 5.9 | Conclusion | 237 |
| | Bibliography | 237 |

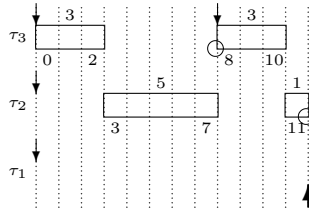


Figure 5.1: The task set is unschedulable in the synchronous case.

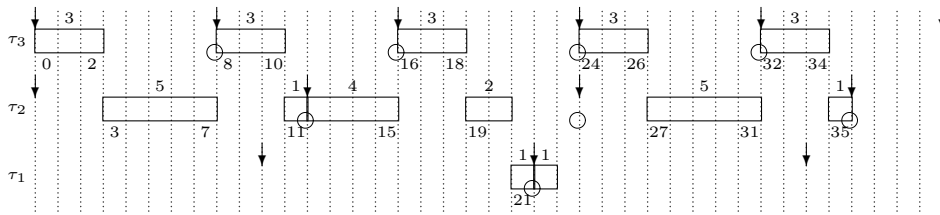


Figure 5.2: The task set is schedulable; at $t = 34$ the situation is the same as at $t = 10$ and the schedule repeats.

5.1 Introduction

In the previous chapters we have studied the feasibility problem of various sub-classes of periodic task sets, using static and dynamic priority scheduling algorithms. From this study a general remark can be raised:

From a schedulability point of view the synchronous case is the worst case, i.e., if the system is schedulable in the synchronous case it follows that this is also the case in all asynchronous situations.

More precisely for (late and) general deadline systems with monotonic priority assignments the largest response time occurs for the first request of τ_i in the synchronous case (see Theorem 3.13); for arbitrary deadline with monotonic priority assignments the largest response time occurs in the interval $[0, \lambda_n)$ in the synchronous case (see Theorem 2.37). Moreover, for dynamic priority rules and especially for the deadline driven scheduler, the largest response time does not necessarily occur in the first busy period, nor in the synchronous schedule, but the synchronous case remains the worst case from a schedulability point of view (see Theorem 4.11), whatever the sub-class of periodic task sets considered in this work. We have finally noticed that, for the least laxity first scheduling algorithm, the synchronous case is also the worst case (see Theorem 4.70).

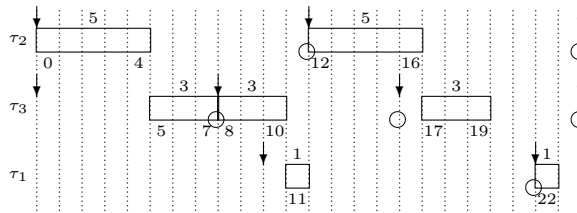


Figure 5.3: The task set is schedulable; at $t = 24$ the situation is the same as at $t = 0$ and the schedule repeats.

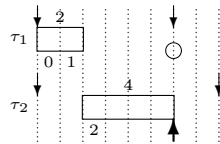


Figure 5.4: The synchronous task set is unschedulable.

Consequently, for static as well as dynamic schedulers, it is pessimistic to consider only the synchronous case, since a system can be unschedulable in the synchronous case, while being schedulable in a particular asynchronous situation. We present two such systems, first for the static case, then for the dynamic one.

Example 5.1 Consider the task set composed of three tasks τ_1, τ_2 and τ_3 ; $\tau_1 = \{C_1 = 1, T_1 = D_1 = 12, O_1 = 10\}$, $\tau_2 = \{C_2 = 6, T_2 = D_2 = 12, O_2 = 0\}$, $\tau_3 = \{C_3 = 3, T_3 = D_3 = 8, O_3 = 0\}$. In the synchronous case the system is statically unschedulable, even with the optimal priority assignments given by the rate monotonic rule: $\tau_3 > \tau_2 > \tau_1$ (see Figure 5.1) or $\tau_3 > \tau_1 > \tau_2$: the first request of task τ_1 misses its deadline. But with the offset $O_3 = O_2 = 0$ and $O_1 = 10$, the system is schedulable with the priority assignment $\tau_3 > \tau_2 > \tau_1$ (see Figure 5.2) and with the priority assignment $\tau_2 > \tau_3 > \tau_1$ (see Figure 5.3). ■

Hence an interesting problem arises: if the offsets are not fixed by the constraints of the problem and the task set is unschedulable in the synchronous case, is there an assignment of the offsets and priorities such that it becomes schedulable?

Example 5.2 Let us consider the following system with the deadline driven scheduler.

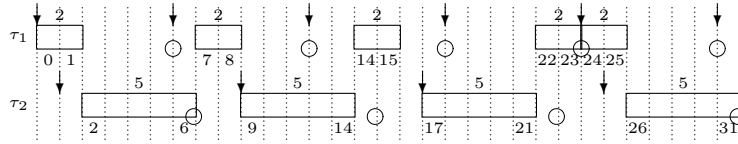


Figure 5.5: The asynchronous task set is schedulable (from time $t = 24$, the schedule repeats).

| | C | T | D |
|----------|-----|-----|-----|
| τ_1 | 2 | 6 | 6 |
| τ_2 | 5 | 8 | 6 |

If we choose $O_1 = O_2 = 0$, the system is unschedulable: the first request of τ_2 misses its deadline (see Figure 5.4). However, the system is schedulable with $O_1 = 0$ and $O_2 = 1$ (see Figure 5.5). ■

As a consequence, if the real-time system for which the scheduling is computed does not have definite requirements about the task start times (offsets), considering only the synchronous case is too pessimistic. We consider in this chapter hard real-time systems which have no definite requirements about the task start times. In such systems the offsets will be chosen beforehand by the scheduling algorithm. We call this kind of system: *offset free system*.

We shall see in section 5.5 that the deadline driven scheduler and the least laxity first scheduler remain optimal for offset free systems. For these reasons, the practical interest of offset free systems for the deadline driven scheduler (and the least laxity first rule) is even more important than in the static case, since in the static case no optimal priority assignment is known, but of course the problem of scheduling offset free systems with static priority assignment is also relevant.

The remainder of this chapter is as follows: in section 5.2 we show that we can restrict the offsets to have the same granularity than the task characteristics, in section 5.3 we study the notion of non-equivalent asynchronous systems, in particular we show that only $\frac{\prod_{i=1}^n T_i}{P}$ different offset assignments need be considered; in section 5.4 and 5.5 we study the optimality problem of popular priority rules for offset free systems; we first consider the case of monotonic priority assignments and then the case of the deadline driven scheduler; in section 5.6 we show the interest to consider systems where the offsets can be chosen by the scheduling algorithm; in section 5.7 we propose an optimal offset assignment which considers only the non-equivalent asynchronous systems, and in section 5.8 we present a pseudo-polynomial time and nearly optimal heuristic offset assignment rule.

5.2 Offset granularity

We have assumed in our model of computation that the fixed characteristics of the various tasks (i.e., T_i, D_i, C_i for offset free systems) are natural numbers. In this chapter we consider real-time systems which have no requirement on the offset values. In particular the offsets may have a different granularity than the fixed characteristics. A priori it could be interesting to choose the offsets with a finer granularity. If a system is unschedulable for all natural offset assignments (i.e., $O_i \in \mathbb{N}$ for $i = 1, \dots, n$), it is not obvious that this will still be the case if we allow, for example, the offsets to be multiples of $\frac{1}{2}$ (i.e., $O_i \in \{\frac{p}{2} | p \in \mathbb{N}\}$ for $i = 1, \dots, n$). It may be noticed that allowing the offsets to be multiple of $\frac{1}{2}$ for an offset free system $S = \{\tau_i = \{T_i, C_i, D_i\} | i = 1, \dots, n\}$ with $T_i, C_i, D_i \in \mathbb{N}$ is equivalent to allow the offsets to be natural numbers for the offset free system $S' = \{\tau'_i = \{T'_i = 2T_i, C'_i = 2C_i, D'_i = 2D_i\} | i = 1, \dots, n\}$ where the fixed characteristics are multiplied by 2. We shall show that we may restrict the offsets to have the same granularity than the fixed characteristics (i.e., natural numbers in our model of computation, or still better: multiples of the greatest common divisor of the fixed characteristics): from a schedulability point of view, for offset free systems, it is not relevant to have a finer granularity for the offsets.

Definition 5.3 Let $S = \{\tau_i = \{T_i, C_i, D_i\} | i = 1, \dots, n\}$ with $T_i, C_i, D_i \in \mathbb{N}$ be an offset free system. An offset assignment is said to have a *granularity of m* iff m is the smallest positive integer such that $O_i \in \{\frac{p}{m} | p \in \mathbb{N}\}$ ($i = 1, \dots, n$). ■

First, we consider the case of static scheduling algorithms.

Theorem 5.4 Let $S = \{\tau_i = \{T_i, C_i, D_i\} | i = 1, \dots, n\}$ with $T_i, C_i, D_i \in \mathbb{N}$ be an offset free system with arbitrary deadlines. If S is not schedulable with the static priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$ for all natural offset assignments (i.e., $O_i \in \mathbb{N}$ for $i = 1, \dots, n$), then this is also the case for all offset assignments with a granularity of m , for all m ($m \in \mathbb{N}_0 = \mathbb{N} \setminus \{0\}$).

Proof. Let O_1, \dots, O_n be an offset assignment with a granularity of m . We shall show that, under our assumptions, this asynchronous system is not schedulable. Let $O'_i = \lfloor O_i \rfloor$ ($i = 1, \dots, n$); we have: $O_i = O'_i + \varepsilon_i$ with $\varepsilon_i \in \{\frac{p}{m} | p \in \mathbb{N}\}$ and $0 \leq \varepsilon_i < 1$ ($i = 1, \dots, n$). Let S' be the asynchronous system with the same task characteristics than S and the natural offsets O'_1, \dots, O'_n : from our assumptions, we have that S' is unschedulable. We shall show that ‘shifting’ right the tasks by ε_j cannot make the system schedulable. The system

S' is not schedulable, hence there is at least one task (say τ_i) which misses its deadline at some time $t = O'_i + (k - 1) \cdot T_i + D_i$ ($k > 0$); t is assumed to be the first deadline failure instant. It may be noticed that the tasks $\tau_{i+1}, \dots, \tau_n$ do not impact on the schedulability of τ_1, \dots, τ_i . Since a request of task τ_i misses its deadline we have only to consider the task subset: τ_1, \dots, τ_i . Let t' be the smallest time instant such that the CPU remains busy for tasks in the subset τ_1, \dots, τ_i , in the interval $[t', t)$ ($0 \leq t' < t$ since at time t task τ_i misses its deadline and had an active request at time $t - 1$). The requests occurring strictly before t' do not impact on the schedule in the interval $[t', t)$. Ignoring the requests before t' is equivalent to refine the offsets and consider the system S'' with the same task characteristics than S and the offsets:

$$\begin{aligned} O''_j &= \min_{k \geq 0} \{O'_j + k \cdot T_j \mid O'_j + k \cdot T_j \geq t'\} \\ &= O'_j + \left\lceil \frac{(t' - O'_j)^+}{T_j} \right\rceil \cdot T_j \text{ for } j = 1, \dots, i \end{aligned}$$

We can also change the time origin and consider the system S''' with the same task characteristics than S and the offsets:

$$\forall j : O'''_j = O''_j - \min\{O''_1, \dots, O''_i\} \text{ and } t'' = t - \min\{O''_1, \dots, O''_i\}$$

By construction, the CPU is permanently busy in the interval $[0, t'')$ by tasks in the set $\{\tau_1, \dots, \tau_i\}$. Hence, if τ_i misses its deadline in t'' , it follows that the demand of τ_1, \dots, τ_i in the interval $[0, t'')$ is greater than the length of this interval:

$$\sum_{j=1}^i \left\lceil \frac{(t'' - O'''_j)^+}{T_j} \right\rceil \cdot C_j > t''$$

At best the demand needs one extra unit of CPU:

$$\sum_{j=1}^i \left\lceil \frac{(t'' - O'''_j)^+}{T_j} \right\rceil \cdot C_j \geq t'' + 1 > t''$$

We shall now consider the system \tilde{S} with the same task characteristics than S and the offset $\tilde{O}_1, \dots, \tilde{O}_n$ with a granularity of m :

$$\tilde{O}_j = O_j''' + \varepsilon_j \text{ with } 0 \leq \varepsilon_j < 1 \text{ (} j = 1, \dots, i \text{)}$$

Remark that, with the offset assignment O_1, \dots, O_n the requests of τ_j ($j = 1, \dots, i$) occurring strictly before $\tilde{O}_j + \min\{O_1'', \dots, O_i''\}$ may have an impact²⁰ on the schedule after them; we shall however neglect this impact, which anyway reinforces our argument, since we shall prove that the demand in the schedule from the offsets \tilde{O}_j is already too large and induces a deadline failure.

Let $\tilde{t} = t'' + \varepsilon_i$; \tilde{t} corresponds to the deadline of the request of τ_i which missed its deadline at time t'' in S' . It may be noticed that $\tilde{O}^{min} = \min_{j=1, \dots, i} \{\tilde{O}_j\}$ is not necessarily 0, and in the interval $[0, \tilde{O}^{min}]$ the CPU remains idle. We shall show that in the interval $[\tilde{O}^{min}, \tilde{t})$ the demand of tasks τ_1, \dots, τ_i is greater than or equal to the one in the previous situation, hence greater than the length of the interval and consequently the set remains unschedulable.

By construction of \tilde{t} , each task (say τ_j , $1 \leq j \leq i$) in \tilde{S} has a greater or equal number of requests in the interval $[\tilde{O}^{min}, \tilde{t})$ than in the interval $[0, t'')$ in S''' . To show this, let t_j be the time of the last request of τ_j which occurs before or at time t'' ($t_j \leq t''$) in S''' (if $O_j''' \geq t''$, t_j is not defined but there is no request of τ_j in $[0, t'')$ in S''' and this may only increase). We have to distinguish between two cases:

1. $t_j = t''$: in this case, if the first request of τ_j starts at time \tilde{O}_j instead of O_j''' the number of requests of τ_j in the interval $[\tilde{O}^{min}, \tilde{t})$ is the same if $\varepsilon_i \leq \varepsilon_j$ and is increased by one unit otherwise.
2. $t_j < t''$: in this case we have necessarily that $t'' - t_j \geq 1$. If the first request of τ_j starts at time \tilde{O}_j instead of O_j''' the number of requests of τ_j in the interval $[\tilde{O}^{min}, \tilde{t})$ is the same since $|\varepsilon_i - \varepsilon_j| < 1$.

Hence the demand of tasks τ_1, \dots, τ_i is greater or equal than $t'' + 1$ but the CPU availability is $t'' + \varepsilon_i - \tilde{O}^{min} < t'' + 1$. It follows that the system remains unschedulable: the deadline of τ_i in $t'' + \varepsilon_i$ is missed. \blacksquare

Now, we extend the result to the dynamic deadline driven scheduler.

Theorem 5.5 *Let $S = \{\tau_i = \{T_i, C_i, D_i\} | i = 1, \dots, n\}$ with $T_i, C_i, D_i \in \mathbb{N}$ be an offset free system with arbitrary deadline. If S is not schedulable with*

²⁰e.g., with the system $S = \{\tau_1 = \{T_1 = 2, C_1 = 1 = D_1\}, \tau_2 = \{T_2 = 6, C_2 = 3 = D_2\}\}$, $O_1 = \frac{1}{3}$ and $O_2 = 1$; $O_1' = 0, O_2' = 1, t' = 1, t = 4$; $O_1'' = 2$ and $O_2'' = 1$; $O_1''' = 1$ and $O_2''' = 0$; $\tilde{O}_1 = \frac{4}{3}$ and $\tilde{O}_2 = 0$ with $\min\{O_1'', O_2''\} = 1$, but the request of τ_1 at $\frac{1}{3}$ delays τ_2 from 1 to $\frac{4}{3}$.

the deadline driven scheduler for all natural offset assignments (i.e., $O_i \in \mathbb{N}$ for $i = 1, \dots, n$), then this is also the case for all offset assignments with a granularity of m , for all m ($m \in \mathbb{N}_0$).

Proof. Let O_1, \dots, O_n be an offset assignment with a granularity of m . We shall show that, under our assumptions, this asynchronous system is not schedulable. Let $O'_i = \lfloor O_i \rfloor$ ($i = 1, \dots, n$); we have: $O_i = O'_i + \varepsilon_i$ with $\varepsilon_i \in \{\frac{p}{m} | p \in \mathbb{N}\}$ and $0 \leq \varepsilon_i < 1$ ($i = 1, \dots, n$). Let S' be the asynchronous system with the same task characteristics than S and the natural offsets O'_1, \dots, O'_n : from our assumptions, we have that S' is unschedulable. We shall show that 'shifting' right the tasks with ε_j cannot make the system schedulable. The system S' is not schedulable, hence there is at least one request (say a request of task τ_i) which misses its deadline at some time t ($t > 0$), t is assumed to be the first deadline failure instant. Let t' be the smallest time instant such that in the interval $[t', t)$ the CPU remains busy for requests with deadline less than or equal to t ($t' < t$). The requests occurring strictly before t' do not impact on the schedule in the interval $[t', t)$. Ignoring the requests before t' is equivalent to refine the offsets and consider the system S'' with the same task characteristics than S and the offsets:

$$\begin{aligned} O''_j &= \min_{k \geq 0} \{O'_j + k \cdot T_j | O'_j + k \cdot T_j \geq t'\} \\ &= O'_j + \left\lceil \frac{(t' - O'_j)^+}{T_j} \right\rceil \cdot T_j \quad (j = 1, \dots, n) \end{aligned}$$

We can also change the time origin and consider the system S''' with the same task characteristics than S and the offsets:

$$O'''_j = O''_j - \min\{O''_1, \dots, O''_n\} \quad (j = 1, \dots, n)$$

$$t'' = t - \min\{O''_1, \dots, O''_n\}$$

The CPU is permanently busy by requests with deadline less than or equal to t'' and, at time t'' , at least one request misses its deadline. Hence, the demand that the deadline driven scheduler had to consider in the interval $[0, t'')$ exceeds the available CPU. This demand is

$$\sum_{j=1}^n \left\lceil \frac{(t'' + T_j - O'''_j - D_j)^+}{T_j} \right\rceil C_j.$$

Indeed, the demand of task τ_j is $n_j \cdot C_j$ where n_j denotes the number of requests of τ_j with a deadline less than or equal to t'' ; hence n_j is the largest natural integer such that $(n_j - 1)T_j + D_j + O_j''' \leq t''$: the formula follows.

Hence, this demand is strictly greater than the length t'' of the interval; at best the demand needs one extra unit of CPU:

$$\sum_{j=1}^n \left\lfloor \frac{(t'' + T_j - O_j''' - D_j)^+}{T_j} \right\rfloor C_j \geq t'' + 1 > t''.$$

We shall now consider the system \tilde{S} with the same task characteristics than S and the offset $\tilde{O}_1, \dots, \tilde{O}_n$ with a granularity of m :

$$\tilde{O}_j = O_j''' + \varepsilon_j \text{ with } 0 \leq \varepsilon_j < 1 \text{ (} j = 1, \dots, n \text{)}$$

Remark that, with the offset assignment O_1, \dots, O_n the requests of τ_j ($j = 1, \dots, i$) occurring strictly before $\tilde{O}_j + \min\{O_1'', \dots, O_n''\}$ may have an impact²⁰ on the schedule after them; we shall however neglect this impact, which anyway reinforces our argument, since we shall prove that the demand in the schedule from the offsets \tilde{O}_j is already too large and induces a deadline failure.

Let $\tilde{t} = t'' + \varepsilon''$ with $\varepsilon'' = \max\{\varepsilon_i | \tau_i \text{ has a deadline at time } t\}$: \tilde{t} corresponds to the largest deadline of the requests with a deadline at t'' in S'' . It may be noticed that $\tilde{O}^{min} = \min_{j=1, \dots, n} \{\tilde{O}_j\}$ is not necessarily 0: in the interval $[0, \tilde{O}^{min}]$ the CPU remains idle. We shall show that in the interval $[\tilde{O}^{min}, \tilde{t})$ the demand that the deadline driven scheduler has to satisfy before \tilde{t} is equal to the one in the previous situation between $[0, t'')$, i.e., is greater than the length of the interval and consequently the set remains unschedulable.

By construction of \tilde{t} , each task (say τ_j , $1 \leq j \leq i$) has the same number of requests in the interval $[0, \tilde{t})$ for \tilde{S} with a deadline not later than \tilde{t} , than in the interval $[0, t'')$ for S''' with a deadline not later than t'' . To show this, consider the first time d_j strictly after than t'' which corresponds to a deadline of task τ_j in S''' . It follows from our definitions that $d_j - t'' \geq 1$ and consequently the number of requests of τ_j with a deadline less than or equal to \tilde{t} with the offset $\tilde{O}_1, \dots, \tilde{O}_n$ is identical since $|\varepsilon'' - \varepsilon_j| < 1$.

Hence the demand that the deadline driven scheduler has to satisfy before \tilde{t} is at least $t'' + 1$ and the CPU availability is $t'' + \varepsilon'' - \tilde{O}^{min} < t'' + 1$. It follows that the system remains unschedulable. ■

It may be noticed that Baruah, Howell and Rosier [BRH90] have shown a similar result for general deadline systems. They have shown that if a feasible offset assignment exists where $O_i \in \mathbb{R}$ then there exists a feasible offset assignment where $O_i \in \mathbb{N}$. The proof uses an elaborate reasoning to assert

this property. We have shown that we can restrict the schedule to be discrete (see the discussion after Theorem 4.16); for this reason we have considered Theorem 5.5 rather than the result of Baruah, Howell and Rosier. Moreover, we have considered the case of arbitrary deadline systems: among all offsets assignments, i.e., for all granularity, we may always restrict the offsets to be natural numbers.

Corollary 5.6 *Let $S = \{\tau_i = \{T_i, C_i, D_i\} | j = 1, \dots, n\}$ with $T_i, C_i, D_i \in \mathbb{N}$ be an offset free system. If S is not schedulable with the least laxity first algorithm for all natural offset assignments (i.e., $O_i \in \mathbb{N}, i = 1, \dots, n$), then this is also the case for all offset assignments with a granularity of m , for all m ($m \in \mathbb{N}_0$).*

Proof. We prove the theorem by contradiction: suppose there is some offset free system S schedulable with the least laxity first algorithm and an offset assignment O_1, \dots, O_n with a granularity of m ($m > 1$) while all natural offset assignments make the system unschedulable. By Lemma 4.67, we have that S is also schedulable with the deadline driven scheduler and the offsets O_1, \dots, O_n . From Theorem 5.5 there must exist a natural offset assignment (say O'_1, \dots, O'_n) for which the system S remains schedulable with the deadline driven scheduler. If we apply Lemma 4.67 again we have a contradiction, since S is schedulable with the least laxity first algorithm for the natural offset assignment O'_1, \dots, O'_n . ■

Consequently, in this chapter we shall restrict without loss of generality the offsets to have the same granularity than other task characteristics, i.e., to be integer numbers.

5.3 Non-equivalent asynchronous systems

We shall in this section introduce the notion of (non-)equivalent asynchronous systems; this notion (and its interest) does not depend on the scheduling algorithm, not even on the scheduling family (i.e., static or dynamic). For this reason we shall present here results without considering a specific scheduling algorithm, but in particular circumstances. We only suppose that the schedule is periodic with a period of P time units, that only the periodic behavior is significant regarding the feasibility of the system and that the periodic behavior only depends on the relative phasing of task requests, which is true for all the schedulings techniques we considered in this work.

Indeed, we have studied in Chapters 2 and 4 the feasibility problem of various sub-classes of periodic task sets (e.g., synchronous late deadline system,

asynchronous general deadline system, etc.): in all the cases and particularly in the more general ones, i.e., asynchronous and arbitrary deadline systems, the feasibility of the system only depends on the periodic part of the schedule (consequently the first part of the schedule may be neglected). It remains to show that suppressing the request of τ_i at time O_i does not alter the periodic behavior of the system; we shall first consider the case of static priority assignments.

Lemma 5.7 *Let S be feasible asynchronous and arbitrary deadline system with $U \leq 1$, the periodic part of the schedule using a static priority assignment is not altered²¹ by suppressing the request of τ_i at time O_i .*

Proof. From the proof of Theorem 2.67, we know that the schedule is finally periodic when $U \leq 1$. Let $t \geq O_i + T_i$ be a time instant in the periodic part of the schedule; from Lemma 3.10 there must exist an idle point $t' \geq t$. It follows that at time t' all previous task requests have completed their execution, the schedule from time t' does not depend on requests before time t' and the schedule repeats from time t' . The property follows. ■

Now, we extend the result to the dynamic deadline driven scheduler.

Lemma 5.8 *Let S be an asynchronous and arbitrary deadline system with $U \leq 1$, the periodic part of the schedule using a request-dependent deadline driven scheduler is not altered²¹ by suppressing the request of τ_i at time O_i .*

Proof. From Lemma 4.35, we know that the schedule is finally periodic when $U \leq 1$. Let $t \geq O_i + T_i$ be a time instant in the periodic part of the schedule; from Lemma 3.10 there must exist an idle point $t' \geq t$. It follows that at time t' all previous task requests have completed their execution, the schedule from time t' does not depend on requests before time t' and the schedule repeats from time t' . The property follows. ■

Several asynchronous systems may lead to the same periodic behavior, and can be considered as equivalent in terms of feasibility according to the previous remark.

Definition 5.9 Let S and S' be two asynchronous arbitrary deadline systems: $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ and $S' = \{\tau'_i = \{O'_i, C'_i = C_i, D'_i = D_i, T'_i = T_i\} | i = 1, \dots, n\}$. Let σ_S and $\sigma_{S'}$ their schedule; S and S' are said to be *equivalent* ($S \equiv S'$) if they have the same periodic behavior, i.e.,

²¹In this context, by “not altered” we mean: $\exists t_0, a \in \mathbb{N} : \forall t \geq t_0 : C_S(t) = C_{S'}(t + a)$ where S' is the modified system.

$\exists t_1, a \in \mathbb{N} \forall t \geq t_1 : \sigma_S(t) = (i, k) \Leftrightarrow \sigma_{S'}(t + a) = (i, k + k_i)$ with $k_i \in \mathbb{Z}$ for $i = 1, \dots, n$. ■

Theorem 5.10 *Let S and S' be two asynchronous arbitrary deadline systems: $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ and $S' = \{\tau'_i = \{O'_i, C'_i = C_i, D'_i = D_i, T'_i = T_i\} | i = 1, \dots, n\}$; S and S' are equivalent iff*

$$\exists k_1, \dots, k_n, A \in \mathbb{Z} : O_i = O'_i + k_i \cdot T_i + A \quad (1 \leq i \leq n).$$

Proof.

(Only if part.) From Definition 5.9, at time $t \geq t_1$ the schedules are identical, including the relative phasing between task requests; it follows that there must exist $k_i \in \mathbb{Z}$ ($1 \leq i \leq n$), $A \in \mathbb{Z} : O_i = O'_i + k_i \cdot T_i + A$ ($1 \leq i \leq n$).

(If part.) The difference between system S and S' lies in the offsets. The schedule of the system S can be obtained from the one of S' by adding or subtracting requests of task τ_i at times O_i and by changing the time origin. These transformations do not change the periodic behavior of the systems S' from our assumptions. ■

We have seen that the feasibility of the synchronous case is a simpler problem in terms of time complexity, especially for static scheduling algorithms applied to general deadline systems. Hence, it is interesting to use feasibility tests defined for synchronous systems if the considered asynchronous system is equivalent to the synchronous one. Definition 5.9 can be simplified in this case.

Definition 5.11 Let S be an asynchronous and arbitrary deadline systems: $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$; S is said to be *equivalent to its synchronous case* if

$$\exists t, k_1, \dots, k_n \in \mathbb{N} \text{ such that } \forall i : t = O_i + k_i \cdot T_i \quad (5.1)$$

It is not necessarily obvious to check if an asynchronous system matches Equation (5.1); first, let us remark that Equation (5.1) may reformulated as follows:

$$\exists t \in \mathbb{N} \text{ such that } \forall i : t \equiv O_i \pmod{T_i}.$$

If the values T_i are pairwise prime this problem is known as the *Chinese Remainder Theorem* or the *Chinese Lemma*.

Theorem 5.12 (Chinese Remainder Theorem [CS47]) *Let T_1, T_2, \dots, T_n be positive integers which are relatively prime pairwise, i.e.,*

$$\gcd(T_i, T_k) = 1 \text{ when } i \neq k.$$

Let $P = T_1 \times T_2 \times \dots \times T_n$, then the congruence system:

$$\begin{aligned} t &\equiv O_1 \pmod{T_1} \\ t &\equiv O_2 \pmod{T_2} \\ &\vdots \\ t &\equiv O_n \pmod{T_n} \end{aligned}$$

has exactly one solution (modulo P). ■

Consequently, if the periods are relatively prime (pairwise), the asynchronous system is always equivalent to the synchronous case. In the framework of this work, we are only interested in the existence of a solution, but its construction can be found in [Knu69], pp. 250. In order to verify if the periods are pairwise relatively prime, we can apply the Euclid's algorithm to each pair $(T_i, T_j), j \neq i$, hence the time complexity of this procedure is $O\left(\binom{n}{2} \times \log T^{\max}\right) = O(n^2 \times \log T^{\max})$. It may be noticed that there may exist other methods to check the pairwise primality of the periods; we leave this question to the perspicacity of the mathematicians.

We shall now consider the case where the periods are not relatively prime. Knuth gave a generalization of Theorem 5.12 to this case (see exercise 3, section 4.3.2 of [Knu69]).

Theorem 5.13 (Generalized Chinese Remainder Theorem) *Let T_1, T_2, \dots, T_n be positive integers. Let P be the least common multiple of T_1, T_2, \dots, T_n and let a, O_1, O_2, \dots, O_n be any integers. There is exactly one integer t which satisfies the conditions*

$$a \leq t < a + P, \quad t \equiv O_j \pmod{T_j} \quad 1 \leq j \leq n,$$

provided that

$$O_i \equiv O_j \pmod{\gcd(T_i, T_j)} \quad 1 \leq i < j \leq n; \quad (5.2)$$

and there is no such integer t when the latter condition fails. ■

In the framework of this work, we are only interested in the existence of a solution, but its construction can be found in Yih-hing [Dic19, Mat81] and Knuth [Knu69], pp. 513.

From Theorem 5.13 it follows that $O_i \equiv O_j \pmod{\gcd(T_i, T_j)}$ is a necessary and sufficient condition for an asynchronous system to be equivalent to the corresponding synchronous one. Checking if an asynchronous system matches Equation (5.2) can be resolved by applying the Euclid's algorithm to each pair (i, j) . The maximal time complexity of this procedure is again $O(n^2 \times \log T^{max})$. It may be noticed that there may exist other methods to check the $\frac{n(n-1)}{2}$ conditions simultaneously; we leave again this question to the perspicacity of the mathematicians.

We shall now show that there are $\frac{\prod_{i=1}^n T_i}{P}$ different classes of equivalent asynchronous systems (for the same values of T_1, T_2, \dots, T_n), based on the relationship given by Definition 5.9. If we fix the periods there is an infinite number of asynchronous systems. However, we may first remark that without loss of generality we can restrict the offsets as follows.

Theorem 5.14 *We may restrict the offsets in such a way that*

$$\begin{cases} O_1 = 0, \\ O_i \in [0, T_i) \end{cases} \quad i = 2, \dots, n.$$

without emptying any equivalence class of asynchronous systems.

Proof. This results immediately from our assumptions, in particular from the fact that the periodic part of the schedule is not altered²¹ by suppressing the request of τ_i at time O_i . ■

Theorem 5.15 *We may restrict the offsets in such a way that they fulfill the limited growing offset property*

$$\begin{cases} O_1 = 0, \\ O_i \in [O_{i-1}, O_{i-1} + T_i) \end{cases} \quad i = 2, \dots, n.$$

without emptying any equivalence class of asynchronous systems.

Proof. This results immediately from our assumptions, in particular from the fact that the periodic part of the schedule is not altered by suppressing the request of τ_i at time O_i . ■

It follows that the number of classes of equivalent asynchronous systems is finite and not greater than $\prod_{i=2}^n T_i$. In order to identify this number exactly, we shall base our study on the request separation time notion.

Definition 5.16 Let $\Gamma = \{\tau_i = \{C_i, D_i, T_i\} | i = 1, \dots, n\}$ be an offset free task set. For the offset assignment $\vec{O} = \langle O_1, \dots, O_n \rangle$ and for all $k > 0$ such that R_1^k is in the periodic part of the systems (e.g., $R_1^k \geq O^{max} + P$), we define the *request separation* for the k^{th} request of τ_1 as $\vec{\Delta}(k, \vec{O}) = \langle \Delta_2(k, \vec{O}), \dots, \Delta_n(k, \vec{O}) \rangle$, where $\Delta_j(k, \vec{O})$ is the delay between R_1^k and the first request of τ_j which immediately follows time R_1^k , i.e., $\Delta_j(k, \vec{O}) = (O_j - R_1^k) \bmod T_j$ (notice that $\Delta_1(k, \vec{O}) = 0$). ■

The request separation time is used here to compare asynchronous systems in terms of their relative phasings and check their equivalence.

Theorem 5.17 Let S and S' be two asynchronous arbitrary deadline systems: $S = \{\tau_i = \{O_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$ and $S' = \{\tau'_i = \{O'_i, C_i, D_i, T_i\} | i = 1, \dots, n\}$; $S \equiv S'$ iff $\exists k_1, k_2 \in \mathbb{N} : \vec{\Delta}(k_1, \vec{O}) = \vec{\Delta}(k_2, \vec{O}')$.

Proof.

(if part). If $\vec{\Delta}(k_1, \vec{O}) = \vec{\Delta}(k_2, \vec{O}')$, it follows that $\forall j \geq 1$ we have $(O_j - R_1^{k_1}) \bmod T_j = (O'_j - R_1'^{k_2}) \bmod T_j$, hence $\exists r_j : O_j = O'_j + r_j T_j + (R_1^{k_1} - R_1'^{k_2})$ and $S \equiv S'$ from Theorem 5.10.

(only if part). If $S \equiv S'$ it follows that the periodic behavior of S and S' are identical: there must exist t_1 which corresponds with a new request of τ_1 (say the k_1^{th}) in S and t_2 which corresponds to a new request of τ_1 (say the k_2^{th}) in S' such that $(t_1 - O_i) \bmod T_i = (t_2 - O'_i) \bmod T_i$ (for all $i = 1, \dots, n$). It follows that $\vec{\Delta}(k_1, \vec{O}) = \vec{\Delta}(k_2, \vec{O}')$. ■

Definition 5.18 Let \vec{O}_1 and \vec{O}_2 be two offset assignments. \vec{O}_1 and \vec{O}_2 are equivalent ($\vec{O}_1 \equiv \vec{O}_2$) iff $\exists k_1, k_2 \in \mathbb{N} : \vec{\Delta}(k_1, \vec{O}_1) = \vec{\Delta}(k_2, \vec{O}_2)$. ■

It follows from Definition 5.18 and Theorem 5.17 that equivalent offset assignments define equivalent asynchronous systems (and inversely).

Lemma 5.19 Let $\Gamma = \{\tau_i = \{C_i, D_i, T_i\} | i = 1, \dots, n\}$. There are $\prod_{i=2}^n T_i$ different request separations for any request (say the k^{th}) of τ_1 , when the task periods are fixed and the offsets are free.

Proof. Since $0 \leq \Delta_i(k, \vec{O}) < T_i$, we have that the number of different request separations is the number of different tuples $\langle x_2, x_3, \dots, x_n \rangle$ with $0 \leq x_i < T_i$ and $x_i \in \mathbb{N}$. ■

Lemma 5.20 *Let $\Gamma = \{\tau_i = \{C_i, D_i, T_i\} | i = 1, \dots, n\}$. The offset assignment \vec{O} defines $\frac{P_n}{T_1}$ equivalent and different request separations for any request of τ_1 , where $P_n = \text{lcm}\{T_j | j = 1, \dots, n\}$.*

Proof. The behavior of the system is periodic with a period of P_n (moreover, P_n is the smallest such period). Hence, the successive request separations for the requests of τ_1 $\vec{\Delta}(k, \vec{O}), \vec{\Delta}(k+1, \vec{O}), \dots$ are also periodic, with a period $\frac{P_n}{T_1}$ (since the requests of τ_1 are separated by T_1 time units) and the interval $[R_1^k, R_1^k + P_n)$ contains $\frac{P_n}{T_1}$ request separations. We have also to prove that in the interval $[R_1^k, R_1^k + P_n)$ all the request separations for the requests of τ_1 are different. Suppose that this is not true: there exists $t_1 = R_1^k + p_1 T_1$ and $t_2 = R_1^k + p_2 T_1$ with $R_1^k \leq t_1 < t_2 < R_1^k + P_n$ such that

$$O_j - (R_1^k + p_1 T_1) \equiv O_j - (R_1^k + p_2 T_1) \pmod{T_j}$$

which implies that $p_1 T_1$ and $p_2 T_1$ are multiples of T_j ($j = 1, \dots, n$), hence of $\text{lcm}\{T_j | j = 1, \dots, n\}$, but $0 < t_2 - t_1 < P_n$, a contradiction with the fact that $P_n = \text{lcm}\{T_j | j = 1, \dots, n\}$. ■

Theorem 5.21 *Let $\Gamma = \{\tau_i = \{C_i, D_i, T_i\} | i = 1, \dots, n\}$. There are $\frac{\prod_{i=1}^n T_i}{P_n}$ different equivalence classes of offset assignments according to the equivalence relation given by definition 5.18.*

Proof. Let x be the number of such classes. By Lemma 5.19 and Lemma 5.20, we have that $x \cdot \frac{P_n}{T_1} = \prod_{i=2}^n T_i$. Hence, $x = \frac{\prod_{i=1}^n T_i}{P_n}$. ■

5.4 Non-optimality of monotonic priority assignments

For offset free systems, it is not necessarily true that the rate/deadline monotonic priority assignment still gives the optimal static priority assignment. We have already considered the optimality of the rate/deadline monotonic scheduler for systems where the tasks are not started at the same times, but where the offsets are fixed beforehand (contrary to our offset free systems where the offsets can be chosen by the scheduling algorithm itself), i.e., for asynchronous systems. The synchronous systems studied by Liu and Layland are special cases of asynchronous systems (the case where $O_i = 0$ for all $1 \leq i \leq n$), but the asynchronous systems and the offset free systems are of different natures.

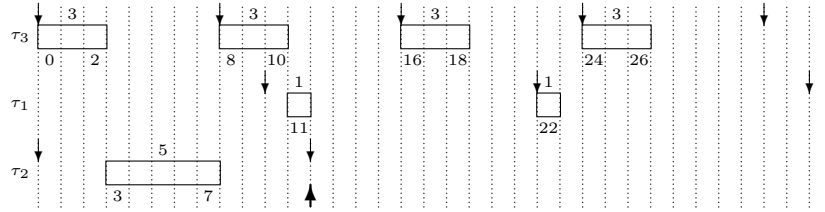


Figure 5.6: The task set is not schedulable with $\tau_3 > \tau_1 > \tau_2$: the first request of τ_2 fails.

We have seen that for asynchronous systems (with general and arbitrary deadlines) the monotonic priority assignment is not weakly optimal. Leung and Whitehead have shown the non (strong) optimality by considering the following system (already considered in the introduction of this chapter).

Example 5.22 $\tau_1 = \{C_1 = 1, T_1 = D_1 = 12, O_1 = 10\}$, $\tau_2 = \{C_2 = 6, T_2 = D_2 = 12, O_2 = 0\}$, $\tau_3 = \{C_3 = 3, T_3 = D_3 = 8, O_3 = 0\}$. This system can be scheduled with priority assignment $\tau_3 > \tau_2 > \tau_1$ (see Figure 5.2) while the rate monotonic priority assignment $\tau_3 > \tau_1 > \tau_2$ is not feasible (see Figure 5.6). ■

This example, largely used in the literature [LW82, Aud91] to illustrate the non-optimality of the rate monotonic priority assignment for asynchronous systems raises two points.

1. Both priority assignments $\tau_3 > \tau_2 > \tau_1$ and $\tau_3 > \tau_1 > \tau_2$ are rate monotonic priority assignments. In this case, the non-optimality of the rate monotonic priority assignment is due to the choice made to resolve the tie between T_1 and T_2 . Hence, the example shows more precisely the non strong optimality for asynchronous systems; but nothing can be inferred at that point for weak optimality. Subsequently, for this reason, we shall only consider non-ambiguous situations, where all periods/deadline are distinct.
2. We cannot conclude from it that the rate/deadline monotonic priority assignment is not strongly optimal for the offset free systems: if we choose in the previous example $O_1 = O_2 = 2$ and $O_3 = 0$, with both rate monotonic priority assignments $\tau_3 > \tau_2 > \tau_1$ and $\tau_3 > \tau_1 > \tau_2$ the system becomes schedulable (see figure 5.7). The optimality analysis of asynchronous systems cannot be transferred directly to the offset free systems.

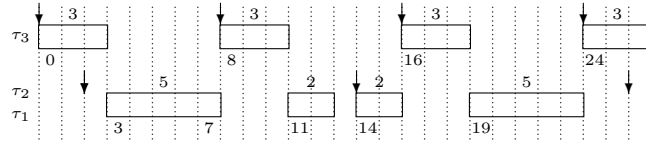


Figure 5.7: Since $O_1 = O_2$ and $T_1 = T_2 = D_1 = D_2$, the tasks τ_1 and τ_2 are like a simple task $\tau' = \{C' = C_1 + C_2 = 7, T' = D' = T_1 = T_2 = 12, O' = O_1 = O_2 = 2\}$; at time $t = 26$ the situation is like at $t = 2$ and the schedule repeats.

Hence, the optimality problem for offset free systems is of a different nature in comparison to asynchronous systems, due to the fact that the offsets can be chosen in order to schedule these systems.

5.4.1 Definitions and properties

Definition 5.23 A priority assignment rule is *strongly optimal* for a family of offset free systems if when a feasible priority assignment (μ) and offset assignment (ρ) exist for some offset free task set of the family, any priority assignment (μ') given by the rule, whatever the way in which the ambiguities are resolved, leads to a feasible schedule for some offset assignment (ρ'). ■

Definition 5.24 A priority assignment rule is *weakly optimal* for a family of offset free systems if when a feasible priority assignment (μ) and offset assignment (ρ) exist for some offset free task set of the family, there is a priority assignment (μ') given by the rule, for a particular way to resolve the ambiguities, which leads to a feasible schedule for some offset assignment (ρ'). ■

The definitions of the optimality for offset free systems and for asynchronous systems are close; in fact the optimality for offset free systems is less demanding than the optimality for asynchronous systems. Indeed, if in the definition of the optimality for offset free systems we required that both offset assignments ρ and ρ' must be the same, we get the definition of the optimality for asynchronous systems and in this case the offsets given by the assignment $\rho = \rho'$ are the offsets of the corresponding asynchronous system. Hence, the optimality for asynchronous systems implies the optimality for offset free systems, in the general case; and conversely the non-optimality of a rule for offset free systems implies it also for asynchronous systems. We have already seen the non-(strong)-optimality of the rate/deadline monotonic assignments for (ambiguous) asynchronous systems, but in some special cases we may get optimality results, both for asynchronous and offset free systems.

5.4.2 Optimality in special cases of offset free systems

First, remark that if a priority rule is optimal for a subclass of asynchronous systems (e.g., systems composed by a single task) the same property holds for offset free systems. More formally:

Lemma 5.25 *If a priority assignment rule is optimal for a subclass of asynchronous systems and the definition of the subclass does not rely on special restrictions about the offsets, the same assignment rule is also optimal for the corresponding subclass of offset free systems.*

Proof. It follows from definition 5.23 that any counter-example for the optimality of a priority rule for offset free systems is also a counter-example for the same priority rule for asynchronous systems. ■

Leung and Whitehead [LW82] have identified two special cases where the deadline monotonic priority assignment is strongly optimal for asynchronous systems:

1. systems with late deadlines having only two tasks,
2. systems with late deadlines satisfying the conditions that each T_i is an exact multiple or sub-multiple of each T_j ($T_i = m_{i,j}T_j$, $m_{i,j} \in \mathbb{N}$ when $T_i > T_j$).

From Lemma 5.25 in these special cases, the deadline and the rate monotonic priority assignment are optimal for offset free systems:

Corollary 5.26

1. *The deadline monotonic priority assignment is strongly optimal for offset free systems with late deadline having only two tasks.*
2. *The rate monotonic priority assignment is strongly optimal for offset free systems with late deadlines having only two tasks.*
3. *The deadline monotonic priority assignment is strongly optimal for offset free systems with late deadlines satisfying the conditions that each T_i is an exact multiple or sub-multiple of each T_j ($T_i = m_{i,j}T_j$, $m_{i,j} \in \mathbb{N}$ when $T_i > T_j$).*

4. The rate monotonic priority assignment is strongly optimal for offset free systems with late deadlines, satisfying the conditions that each T_i is an exact multiple or sub-multiple of each T_j ($T_i = m_{i,j}T_j$, $m_{i,j} \in \mathbb{N}$ when $T_i > T_j$).

But we may also devise another special case.

Theorem 5.27 *The rate monotonic priority assignment is strongly optimal for asynchronous systems with late deadlines satisfying the condition that all the periods are distinct and $\frac{T_i}{T_j} \geq 2$ whenever $T_i > T_j$.*

Proof. We must prove that if a feasible priority assignment exists for some task set satisfying the given condition, the rate monotonic priority assignment is also feasible for that task set. Let τ_1, \dots, τ_n be a set of n such tasks with a feasible priority assignment $\tau_1 > \tau_2 > \dots > \tau_i > \tau_{i+1} > \dots > \tau_n$. Let τ_i and τ_j be two tasks of adjacent priorities ($j = i + 1$). Suppose that $T_i > T_j$. Let us exchange the priorities of τ_i and τ_j : if the task set is still schedulable, since the (unique) rate monotonic priority assignment can be obtained from any priority ordering by a sequence of such priority exchanges, we may deduce that the rate monotonic priority assignment is schedulable.

The priority exchange does not modify the schedulability of the tasks with a higher priority than τ_i (i.e., $\tau_k \forall k < i$). The task τ_j remains of course schedulable after the priority exchange, since it may use all the free slots left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}\}$ instead of only those left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}, \tau_i\}$. Assuming that the requests of τ_i remain schedulable, from Lemma 2.12 the scheduling of each task τ_k ($k = i + 2, i + 3, \dots, n$) is not altered since the idle periods left by higher priority tasks are identical. Consequently, we must only verify that τ_i also remains schedulable. We shall show by induction on r that the r^{th} request of τ_i (at time x) remains schedulable after the priority exchange, assuming that all the previous requests of τ_i remain schedulable after the priority exchange. The property is true in the trivial case for $r = 0$. Let us consider the r^{th} request of τ_i (at time x) and the previous request of τ_j (at time y ; we can assume that this request exists since from Lemma 5.7, without loss of generality, we can assume that: $O_1 \geq O_2 \geq \dots \geq O_n$). Since $\frac{T_i}{T_j} \geq 2$, we have necessarily at least one τ_j 's request completely included in the τ_i 's request: $y \leq x \leq y + T_j \leq y + 2T_j \leq x + T_i$ (see Figure 5.8). Before the priority exchange, either the τ_j request at time y is not completed at time x (case 1) or this request is completed at time x (case 2).

1. The τ_j request at time y is not completed at time x ; let \widetilde{C}_j be the remaining process time at time x for this request of τ_j . In this case in

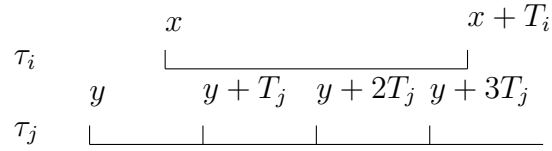


Figure 5.8: Relative task phasing between τ_i and τ_j if $\frac{T_i}{T_j} \geq 2$.

the interval $[x, y + T_j]$ the $\widetilde{C}_j + C_i$ first free units of CPU left by the higher priority tasks (if any) are consumed by τ_i (first) and (then) τ_j , and since the schedule is feasible both requests are fulfilled at $y + T_j$. In the interval $[y + T_j, x + T_i]$ τ_j is not preempted by τ_i since the last request of τ_i is completed. Remark that $x - T_i \leq y - T_j$ so that, if there is a request of τ_i at $x - T_i$, since $O_2 \leq O_1$ there is also a request of τ_j at $y - T_j$, which must be completed at time y , and the request of τ_i at $x - T_i$ must also be completed at that time. After the priority exchange, let \widetilde{C}'_j be the remaining process time at time x for the request of τ_j at time y : $\widetilde{C}'_j = \widetilde{C}_j$ since the request of τ_i at time $x - T_i$ was completed at time y so that the situation is the same in the interval $[y, x)$ before and after the priority exchange (τ_j utilizes all the free slots between y and x left by higher priority tasks); as a consequence, between x and $y + T_j$, the first \widetilde{C}_j free slots are used by τ_j , the next C_i ones are used by τ_i and the latter is completed before $y + T_j$, at the completion time of τ_j before the priority exchange. Notice that in the interval $[y + T_j, x + T_i]$ the situation is the same as before since τ_j was not preempted by τ_i there. Hence the property.

2. The request of τ_j at time y is completed when reaching time x ; this will still be true after the priority exchange. Before the priority exchange either the request of τ_i at time x is completed before or at time $y + T_j$ (case a) or not (case b).
 - a) In the interval $[x, y + T_j]$ the C_i first free units of CPU are consumed by τ_i . In the interval $[y + T_j, y + 2T_j)$ the C_j first free units of CPU are consumed by τ_j . After the priority exchange, since at time x the request of τ_j at time y is terminated, the situation is the same in the interval $[x, y + T_j)$: the first C_i free units of CPU are consumed by τ_i and the request of τ_i is terminated largely in due time (before $y + T_j$ instead of before $x + T_i$). Notice that in the interval $[y + T_j, y + 2T_j)$ the C_j first free units of CPU are consumed by τ_j . In the interval $[y + T_j, x + T_i]$ the situation is the same as before, since τ_j was not preempted by τ_i there. Hence the property.

- b) In the interval $[x, y + T_j)$, all the free units of CPU are consumed by τ_i , the request of τ_j at y is completed before x and in the interval $[y + T_j, y + 2T_j)$ the first free units of CPU are consumed by τ_i (first) and (then) by τ_j ; both requests are fulfilled at time $y + 2T_j$. After the priority exchange, the request of τ_j at y is terminated at x , in the interval $[x, y + T_j)$ the first units of CPU are consumed by τ_i and in the interval $[y + T_j, y + 2T_j)$ the first free units of CPU are consumed by τ_j (first) and by τ_i ; at time $y + 2T_j$ both requests are again fulfilled. Notice that in the interval $[y + T_j, x + T_i]$ the situation is the same as before, since τ_j was not preempted by τ_i there. Hence the property. ■

Corollary 5.28 *The rate monotonic priority assignment is strongly optimal for offset free systems with late deadlines satisfying the condition that all the periods are distinct and $\frac{T_i}{T_j} \geq 2$ whenever $T_i > T_j$.*

Proof. Immediate from Theorem 5.27 and Lemma 5.25. ■

Remark that some asynchronous systems are equivalent to their synchronous case; it follows that the deadline monotonic remains optimal for this kind of system.

Theorem 5.29 *The deadline monotonic priority rule is strongly optimal for asynchronous systems which are equivalent to their synchronous case (i.e., matching Definition 5.11).*

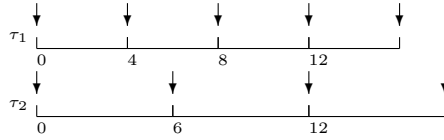
Proof. Immediately follows from Lemma 5.7 and Theorem 2.25.. ■

Corollary 5.30 *The deadline monotonic priority rule is strongly optimal for asynchronous and general deadline systems where the periods are relatively prime pairwise, i.e., if $\gcd(T_i, T_j) = 1 \forall i \neq j$.*

Proof. From the Chinese Remainder Theorem 5.12 there is a synchronization point, there is 1 class of offset assignments, all offset assignments are equivalent to the synchronous case, and from Theorem 5.29 the property results. ■

Corollary 5.31 *The deadline monotonic priority rule is strongly optimal for asynchronous and general deadline systems satisfying the condition: $O_i \equiv O_j \pmod{\gcd(T_i, T_j)}$ for any $i \neq j$.*

Proof. Immediately follows from Theorem 5.29 and the Generalized Chinese Remainder Theorem 5.13. ■

Figure 5.9: Relative phasings between τ_1 and τ_2 .

5.4.3 Non-optimality of monotonic priority assignments in the general case of offset free systems

In order to prove our next result, we have to introduce the notion of the relative phasing between two requests, a notion similar but somewhat different from the *request separation* for the k^{th} request of τ_1 (Definition 5.16) since we consider requests of τ_i and τ_j .

Definition 5.32 Let τ_i and τ_j be two tasks with $\tau_i > \tau_j$. For the k^{th} request of τ_j (which occurs at time $O_j + (k-1)T_j$), we define $\Delta_{i,j}(k)$, the *relative phasing between τ_i and the k^{th} request of τ_j* , as the difference between $O_j + (k-1)T_j$ and the time of the last request of τ_i which occurs before or at time $O_j + (k-1)T_j$ (assuming there is one, i.e., $O_j + (k-1)T_j \geq O_i$). ■

It may be checked that

$$\begin{aligned}\Delta_{i,j}(k) &= (O_j - O_i + (k-1)T_j) \bmod T_i, \\ \Delta_{i,j}(k+1) &= (\Delta_{i,j}(k) + T_j) \bmod T_i.\end{aligned}$$

Hence, the quantities $\Delta_{i,j}(k)$, for successive values of k , form a cycle of length q : $\langle \Delta_{i,j}(k), \Delta_{i,j}(k+1), \dots, \Delta_{i,j}(k+q-1) \rangle$, with $\Delta_{i,j}(k+d) = \Delta_{i,j}(k+d \bmod q)$ and $q = \frac{T_i}{\gcd(T_i, T_j)}$.

Example 5.33 Consider the task set composed of two tasks $S = \{\tau_1 = \{T_1 = 4, O_1 = 0\}, \tau_2 = \{T_2 = 6, O_2 = 0\}\}$; the relative phasings between τ_1 and τ_2 form the cycle of length 2: $\langle \Delta_{1,2}(1) = 0, \Delta_{1,2}(2) = 2 \rangle$ (see Figure 5.9). ■

Now, we are able to solve the problem of the optimality of the rate monotonic priority assignment for offset free systems.

Theorem 5.34 *The rate monotonic priority assignment is not even weakly optimal for the class of offset free systems with late deadlines.*

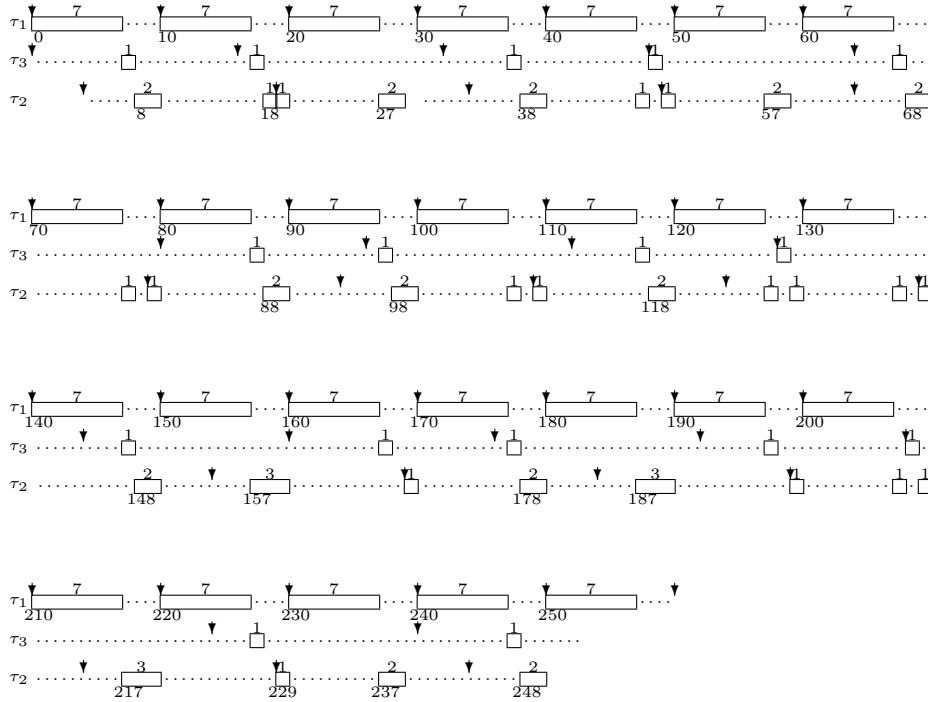
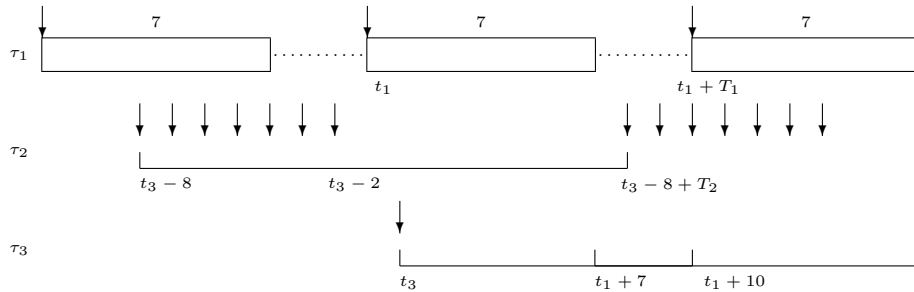


Figure 5.10: The system is schedulable in the interval $[0, 244]$; at time $t = 244$ the situation is like at time $t = 4$ and the schedule repeats.

Proof. To show this, we consider a task set with distinct periods which is a counter-example for the definition of the weak optimality of a priority assignment rule for offset free systems: $S = \{\tau_1, \tau_2, \tau_3\}$ with $\tau_1 = \{T_1 = D_1 = 10, C_1 = 7\}$, $\tau_2 = \{T_2 = D_2 = 15, C_2 = 3\}$, $\tau_3 = \{T_3 = D_3 = 16, C_3 = 1\}$.

1. With the priority assignment ($\tau_1 > \tau_3 > \tau_2$) and the offsets $O_1 = O_3 = 0$ and $O_2 = 4$, the system is schedulable. To verify this, from Theorem 2.57, we only have to check the interval $[4, 244]$ in a simulation from 0 (see Figure 5.10; $X_1 = 0, S_3 = 4, P = 240$).
2. For the unique rate monotonic priority assignment ($\tau_1 > \tau_2 > \tau_3$) and for all offsets (according to Lemma 5.14, i.e., $O_1 = 0, 0 \leq O_2 < 15, 0 \leq O_3 < 16$ – we shall later see that we can in fact only consider $\frac{\prod_{i=1}^n T_i}{P} = 10$ asynchronous situations) this system is unschedulable. This may be checked by a small computer program, but may also be proved formally as follows:

If we apply Theorem 2.57, we know that for τ_3 we only have to check the interval $[S_3, \text{lcm}\{T_1, T_2, T_3\} + S_3] = [S_3, 240 + S_3]$, in a simulation from X_1 (or from 0; notice that S_3 and X_1 depend on the choice of these offsets);

Figure 5.11: $\Delta_{1,3}(h) = 1$.

this interval contains 15 releases of τ_3 . We shall show that, for all offsets, there is a release of τ_3 which misses its deadline in this interval. We can see that the relative phasings between τ_1 and τ_3 ($\Delta_{1,3}(k)$) are either in the cycle $\langle 0, 6, 2, 8, 4 \rangle$ or in the cycle $\langle 1, 7, 3, 9, 5 \rangle$ and the relative phasings between τ_2 and τ_3 ($\Delta_{2,3}(k)$) are in the cycle $\langle 0, 1, 2, \dots, 14 \rangle$. Hence, after at most 4 releases of τ_3 from S_3 , we have a relative phasing between τ_1 and τ_3 equal to 0 or 1. Let r be the rank of the request of τ_3 at time S_3 , t_3 be the time of the request of τ_3 with phasing 1 or 0, h be the rank of this request of τ_3 and t_1 be the time of the last request of τ_1 before or at t_3 .

- (a) If the relative phasing between τ_1 and τ_3 is 1 ($\Delta_{1,3}(h) = 1$, $r \leq h \leq r + 4$), i.e., $t_3 = t_1 + 1$, we see that τ_3 is schedulable only if some CPU unit is free in the interval $[t_1 + 7, t_1 + 10]$; this may only be the case if there is no request for τ_2 arriving between t_1 and $t_1 + 7$, i.e. $\Delta_{2,3}(h) \in \{2, 3, 4, 5, 6, 7, 8\}$ (otherwise, the 3 units before $t_1 + T_1$ are used by τ_2 , see Figure 5.11).

If $\Delta_{2,3}(h) \geq 4$ then 5 releases of τ_3 later, the relative phasing between τ_1 and τ_3 is necessarily the same ($\Delta_{1,3}(h+5) = 1$) and the relative phasing between τ_2 and τ_3 satisfies the condition: $\Delta_{2,3}(h+5) = (\Delta_{2,3}(h) + 5 \times 16) \bmod 15 \in \{9, 10, 11, 12, 13\}$; hence τ_3 misses its deadline since in this configuration τ_1 left 3 free CPU units for lower priority requests in the interval $[R_3^k, R_3^k + D_3)$ (k is the rank of the request of τ_3 we consider) and a request of τ_2 occurs after time R_3^k and before the availability of the 3 free CPU units which are assigned to this request of τ_2 . If $\Delta_{2,3}(h) < 4$ then 10 releases of τ_3 later, the relative phasing between τ_1 and τ_3 is necessarily the same ($\Delta_{1,3}(h+10) = 1$) and the relative phasing between τ_2 and τ_3 satisfies the condition: $\Delta_{2,3}(h+10) = (\Delta_{2,3}(h) + 10 \times 16) \bmod 15 \in \{12, 13\}$; hence τ_3 still misses its deadline.

- (b) If the relative phasing between τ_1 and τ_3 is 0 ($\Delta_{1,3}(h) = 0$, $r \leq h \leq r + 4$), for similar reasons τ_3 is schedulable only if the relative phasing between τ_2 and τ_3 satisfies the condition: $\Delta_{2,3}(h) \in \{1, 2, 3, 4, 5, 6, 7\}$. If $\Delta_{2,3}(h) \geq 3$ then 5 releases of τ_3 later, the relative phasing between τ_1 and τ_3 is necessarily the same ($\Delta_{1,3}(h+5) = 0$) and the relative phasing between τ_2 and τ_3 satisfies the condition: $\Delta_{2,3}(h+5) \in \{8, 9, 10, 11, 12, 13\}$; hence τ_3 misses its deadline. If $\Delta_{2,3}(h) < 3$ then 10 releases of τ_3 later, the relative phasing between τ_1 and τ_3 is necessarily the same ($\Delta_{1,3}(h+10) = 0$) and the relative phasing between τ_2 and τ_3 satisfies the condition: $\Delta_{2,3}(h+10) \in \{11, 12\}$; hence τ_3 still misses its deadline.

We have shown that in any cases, before the $(r+15)^{\text{th}}$ release of τ_3 , the latter misses a deadline. ■

Consequently we also have that:

Corollary 5.35 *The deadline monotonic priority assignment is not weakly optimal for the offset free systems with general deadlines.*

Proof. This results immediately from Theorem 5.34 and the fact that when $D_i = T_i \quad \forall i$, deadline monotonicity coincides with rate monotonicity. ■

Corollary 5.36 *The deadline monotonic priority assignment is not weakly optimal for the offset free systems with arbitrary deadlines.*

Proof. Any example for the non-optimality for offset free systems for late deadline, like in Theorem 5.34, may also serve for the non-optimality for offset free systems and arbitrary deadlines. ■

The task set introduced in the proof of Theorem 5.34 shows that the non-optimality of the rate monotonic scheduler for asynchronous and offset free systems is not due an unfortunate choice to resolve the tie between tasks with same period.

Corollary 5.37 *The rate monotonic priority assignment is not optimal for asynchronous systems with distinct periods.*

Proof. If the periods are distinct, strong and weak optimality collapse. The example in the proof of Theorem 5.34 shows the non-optimality for asynchronous systems with distinct periods. ■

5.5 Optimality of dynamic schedulers for offset free systems

Theorem 4.16 shows the optimality of the deadline driven scheduler for the more general class of periodic task sets considered in this work (i.e., asynchronous and arbitrary deadline systems). If we compare this to the case of static priority assignments and in particular to the deadline monotonic priority assignment, the situation is completely different; we have seen that the latter is not optimal for asynchronous systems (while it is for synchronous ones). Hence, contrary to static priority schedulers, the optimality for dynamic priority schedulers is stronger: it does not depend on the offset values.

We shall show in this section that the deadline driven scheduler and the least laxity first scheduler still give optimal dynamic priority assignments for offset free systems.

Let us consider first the case of the deadline driven scheduler.

Theorem 5.38 *The deadline driven scheduler is strongly optimal for offset free and arbitrary deadline systems.*

Proof. Immediate from Lemma 5.25 and Theorem 4.16. ■

We have already shown that the least laxity first is optimal whatever the offset granularity. It follows that the least laxity first algorithm is optimal for offset free and arbitrary deadline systems.

Theorem 5.39 *The least laxity first scheduler is strongly optimal for offset free systems.*

Proof. Immediate according to the fact that the least laxity first algorithm is optimal (Theorem 4.65) for asynchronous systems and Lemma 5.25. ■

Hence, the optimality of the deadline driven scheduler (and of the least laxity first one) holds for offset free systems, contrary to what happened for static monotonic priority assignments.

5.6 Practical interest of offset free systems

We have seen in the introduction of this chapter the interest to consider systems where the offsets can be computed by the scheduling algorithm, and we

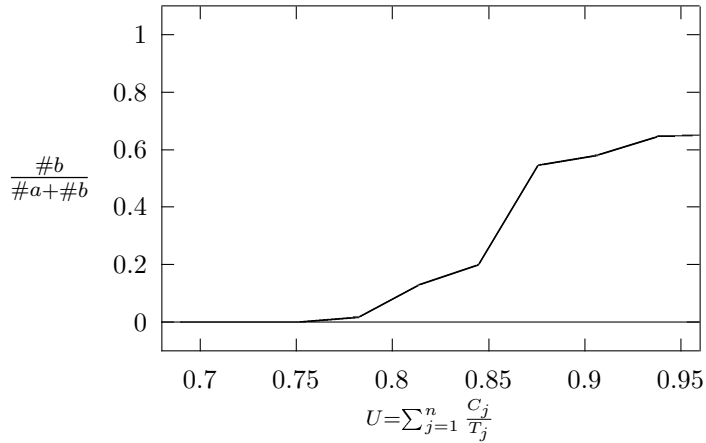


Figure 5.12: Ratio between the number of task sets in the case b and in the case a in function of the utilization factor.

have presented systems which are only schedulable with a judicious choice of the offsets (see Examples 5.1 and 5.2). In this section we are concerned with the practical interest to compute the offsets. For this purpose, we shall consider the ratio between systems which are schedulable in the synchronous case and systems which are only schedulable in an asynchronous case, i.e., if we get some (randomly chosen) system, what is the probability to be in the first or in the second case? It is difficult to answer this question in all generality, since it depends on the real-time system itself, in particular on the system characteristics, i.e., the distribution of the number of tasks, the period values, the load of the system, etc. It is not possible of course to consider all distributions of hard real-time periodic task sets. Moreover, it is hard to determine which distributions are (possibly) realistic.

However, the study of special cases of hard real-time periodic task sets can give a good indication on the interest of offset free systems in practical cases. For this reason we have studied a special case: we consider the late deadline case and we suppose that the priority assignment is resolved with the rate monotonic scheduler. In this special case of hard real-time periodic task sets, the offset free systems have an interest only if the utilization factor ($U = \sum_{i=1}^n \frac{C_i}{T_i}$) is less than 1 but greater than $n(\sqrt[n]{2} - 1)$, since in the opposite case the system is certainly schedulable in the synchronous case (Theorem 3.3). Hence, we only consider heavily loaded systems, with $n(\sqrt[n]{2} - 1) < U \leq 1$. Each of these systems can be either schedulable in the synchronous case (case a), only in an asynchronous case (case b) or unschedulable in all asynchronous cases (case c). This “classification” for a set of n tasks has a maximal time complex-

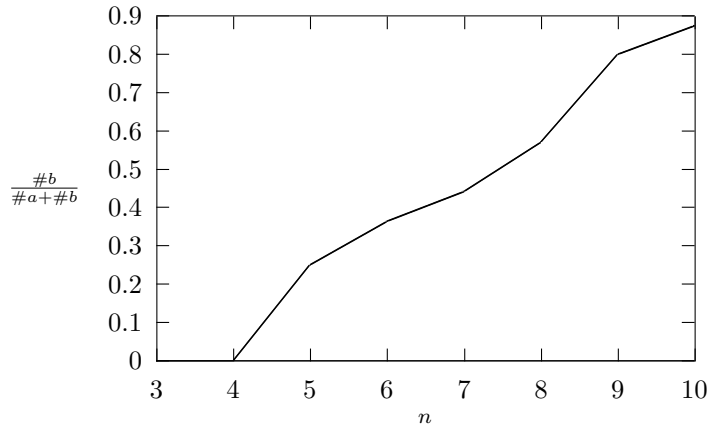


Figure 5.13: Ratio between the number of task sets in the case b and in the case a in function of the number of tasks.

ity of $O\left(\frac{\prod_{i=1}^n T_i}{P} \cdot \mathcal{R}\right)$ where $\frac{\prod_{i=1}^n T_i}{P}$ represents the number of (non-equivalent) offset assignments to be considered (it is also the number of non-equivalent asynchronous systems, see Theorem 5.21) and \mathcal{R} is the time complexity of the schedulability test (in section 5.7 we shall present a method to construct only those $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent asynchronous systems). We have seen that we can restrict the schedulability test to a finite interval but the length of this interval is proportional to P ($P = \text{lcm}\{T_i | i = 1, \dots, n\}$), which may grow exponentially with n . Hence, the classification has an exponential complexity in n and in the periods; consequently, we have limited these values in our study. We present in this section simulation results of this classification applied to a large number of task sets. The task sets are generated with a pseudo-random algorithm; the number of tasks (n) is chosen with equal probability in an interval $[3, 10]$, the periods (T_j) in an interval $[10, 25]$ then the computation times (C_j) are chosen randomly in the interval $[1, T_j]$ and we only consider systems where the utilization factor is greater than $n(\sqrt[n]{2} - 1)$ and less than 1. For each task set (our simulation includes more than 2000 task sets) we have identified its class (case a, case b or case c). Figures 5.12 and 5.13 only concern schedulable systems (i.e., systems in the class a or in the class b), the graph of Figure 5.12 represents the ratio between the task sets in case b and in case a+b in function of the utilization factor, and the graph of Figure 5.13 shows the same ratio in function of the number of tasks.

From the simulation results, it can be noticed that the practical interest of offset free systems is obvious, especially when the utilization factor is large or the number of tasks is “large”. Indeed, the more the utilization factor is close to 1, the more the corresponding systems are only schedulable with a judicious

choice of the offsets. For example, for a utilization factor of 0.9 we have about 60% of schedulable systems which are schedulable only in an asynchronous situation. From Figure 5.13, if we except marginal cases where the number of tasks is very small (i.e., $n < 5$); it occurs that the interest of offset free systems is still more obvious, and this should be also the case for “real-size” systems.

However, when the utilization factor grows, or when the number of tasks grows, the proportion of schedulable systems diminishes. Figures 5.14 and 5.15 are made of 3 graphs: each graph represents the ratio between the number of task sets in a class (a, b or c) and the total number of task sets (e.g. $\frac{\#a}{\#(a \cup b \cup c)}$) in function of the utilization factor and the number of tasks for Figures 5.14 and 5.15, respectively. Figure 5.14 shows that when the utilization factor is large the proportion of unschedulable task sets (case c) is large too, but the interest to compute the offsets remains relevant in that case in order to schedule a larger number of systems. Remark that from a load of 0.87 there is a larger number of asynchronous schedulable systems than synchronous schedulable ones. Notice also that the shape of Figure 5.14 is a bit chaotic and that the maximum of the number of b coincides with the number of a . It seems that a finer analysis could be interesting but this remains for further research. Figure 5.15 shows other interesting phenomena.

- The proportion of unschedulable task sets is more influenced by the utilization factor than by the number of tasks.
- The proportion of schedulable synchronous systems decreases significantly with the number of tasks, and symmetrically the proportion of schedulable offset free systems increases (significantly) with the number of tasks. Again, beside marginal cases (where the number of tasks is very small, i.e., $n < 5$) the interest of offset free systems is obvious.

We shall see in section 5.8 the practical interest of offset free systems for the deadline driven scheduler, but first let us try to determine the optimal offsets for an offset free system.

5.7 Optimal offset assignment

We propose in this section an optimal method to choose the offsets. Let us first consider what optimality means in this case.

Definition 5.40 An offset assignment rule (say \mathcal{A}) is \mathcal{Q} -optimal for a task set family if, when a feasible offset assignment exists for a scheduling rule (say \mathcal{Q})

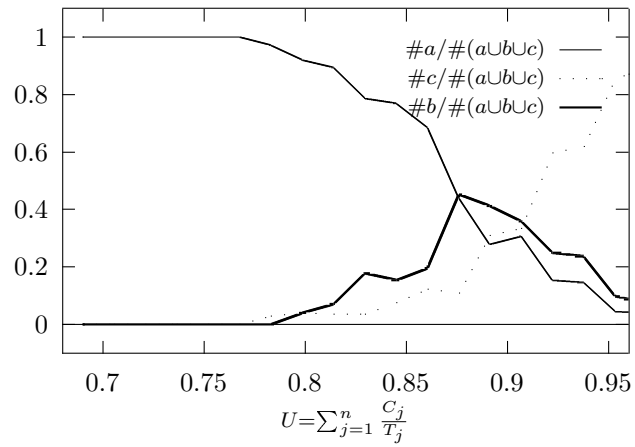


Figure 5.14: Ratio between the number of task sets in each (a or b or c) case and the total number of task sets, in function of the utilization factor.

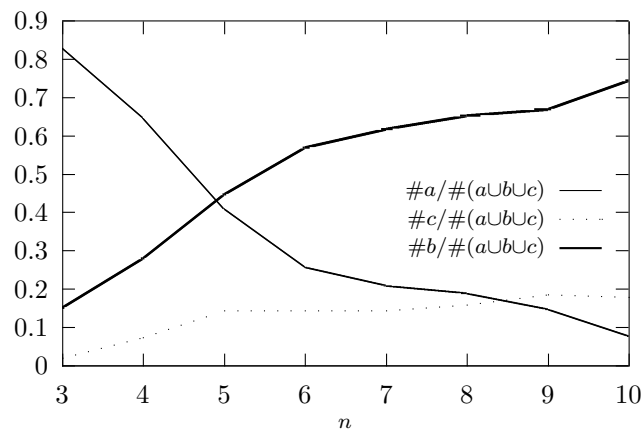


Figure 5.15: Ratio between the number of task sets in each case (a or b or c) and the total number of task sets, in function of the number of tasks.

and a task set of the family, the offset assignment given by the rule \mathcal{A} is also feasible for that task set and the scheduling rule \mathcal{Q} (we neglect here a possible distinction between strong and weak optimality to simplify the presentation). ■

Remark that, in the previous definition the optimality of the offset assignment is related to the scheduling algorithm \mathcal{Q} . Another definition, not related to a specific scheduling rule, could be:

Definition 5.41 An offset assignment rule (say \mathcal{A}) is *optimal* for a task set family if, when a feasible offset assignment exists for a scheduling rule (say \mathcal{Q}) and a task set of the family, the offsets given by the rule \mathcal{A} is also feasible for that task set and for some scheduling rule \mathcal{Q}' . ■

Despite the fact that the second definition is in a way more general, we shall only consider the Definition 5.40, i.e., the optimality of offset assignment for a “fixed” scheduling rule, without fixing the scheduling rule however.

The method proposed here is not dedicated to a particular scheduling rule, like in section 5.3; we shall present results without considering a scheduling algorithm in particular, not even a specific family of scheduling algorithms. Again, we only suppose that the schedule is periodic with a period of P time units, that only the periodic behavior is significant regarding the feasibility of the system and this periodic behavior only depends on the relative phasings between task requests.

A simple (regarding its principle) optimal offset assignment can be defined by searching a feasible offset assignment among all offset combinations. According to Theorem 5.14 we have at most T_i possible values for O_i ($i > 1$) and a single possibility for O_1 (i.e., $O_1 = 0$), consequently the total number of combinations is at most $\prod_{i=2}^n T_i = O((\max_{j=2}^n T_j)^{n-1})$, and the time complexity of the corresponding offset assignment is $O((\max_{j=2}^n T_j)^{n-1} \times \mathcal{R})$, where \mathcal{R} is the maximal time complexity of the schedulability test (e.g., $\mathcal{R} = P$ for asynchronous and general deadline systems, see Chapter 3 for more details) and depends naturally on the scheduling rule and on the kind of asynchronous task sets considered (in particular regarding the relation between the deadlines and the periods). We shall now present a method to only consider the $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent offset assignments as described in Theorem 5.21.

5.7.1 Two tasks

We introduce our method by considering the non-equivalent offset assignments for an offset free system composed of two tasks: τ_1 and τ_2 .

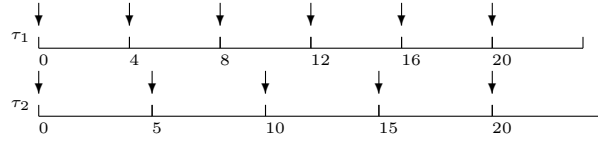


Figure 5.16: Choosing $O_2 = 0$ is equivalent to choose $O_2 = 1, \dots$

According to Theorem 5.14 and without loss of generality we can choose $O_1 = 0$. Our goal is to find $\frac{T_1 \times T_2}{\text{lcm}\{T_1, T_2\}} = \text{gcd}\{T_1, T_2\}$ non-equivalent choices for O_2 . From Theorem 5.14 we know that we can restrict our search among the values $O_2 = 0, O_2 = 1, \dots, O_2 = T_2 - 1$. First let us remark that regarding the relative phasings between the requests of τ_1 and those of τ_2 some choices for O_2 can be considered as equivalent according to Definition 5.11.

Example 5.42 Consider the following characteristics for tasks τ_1 and τ_2 : $\tau_1 = \{T_1 = 4, O_1 = 0\}, \tau_2 = \{T_2 = 5\}$ (we do not specify other task characteristics since they do not interfere in the relative phasings between task requests). We see (Figure 5.16) that choosing $O_2 = 0$ is equivalent to choose $O_2 = 1$, or $O_2 = 2$, etc.

The first occurrence of τ_2 is synchronous with τ_1 if $O_2 = 0$; the second occurrence of τ_2 then has a difference of phase equal to 1 with τ_1 and corresponds to choose $O_2 = 1$, etc. Regarding task τ_1 , choosing $O_2 = 0, O_2 = 1, O_2 = 2$ or $O_2 = 3$ is equivalent; it may be noticed that this is also the case whatever the value of O_1 , since these equivalent choices of O_2 generate the same relative phasings between requests of τ_2 and τ_1 . ■

Two choices (say v_1, v_2) are equivalent if they define the same relative phasing, more formally:

$$\exists k_1, k_2 \in \mathbb{N} : (O_1 + v_1 + k_1 \cdot T_2) \pmod{T_1} = (O_1 + v_2 + k_2 \cdot T_2) \pmod{T_1}.$$

It may be noticed that this relation does not depend on O_1 (we shall see the interest of this property in the more general case for the optimal offset assignment for a set of n tasks).

$$\exists k_1, k_2 \in \mathbb{N} : (v_1 + k_1 \cdot T_2) \pmod{T_1} = (v_2 + k_2 \cdot T_2) \pmod{T_1}. \quad (5.3)$$

Or

$$v_1 \equiv v_2 \pmod{\gcd\{T_1, T_2\}}. \quad (5.4)$$

We shall show that Equation (5.3) is equivalent to Equation (5.4); from Equation (5.3) we get:

$$\begin{cases} v_1 + k_1 T_2 = x + k T_1 \\ v_2 + k_2 T_2 = x + k' T_1 \end{cases}$$

Or

$$\begin{aligned} v_1 + k_1 T_1 &= v_2 + k_2 T_2 + k T_1 - k' T_1 \\ v_1 &= v_2 + (k - k') T_1 + (k_2 - k_1) T_2 \\ v_1 &= v_2 + \gcd\{T_1, T_2\} k'' + \gcd\{T_1, T_2\} k''' \\ v_1 &= v_2 + \gcd\{T_1, T_2\} (k'' + k''') \\ v_1 &\equiv v_2 \pmod{\gcd\{T_1, T_2\}}. \end{aligned}$$

Since $0 \not\equiv 1 \cdots \not\equiv \gcd\{T_i, T_j\} - 1 \pmod{\gcd\{T_i, T_j\}}$, it follows that the values $0, 1, \dots, \gcd\{T_1, T_2\} - 1$ are non-equivalent choices for O_2 .

The optimal offset assignment checks the feasibility of the system for all these values.

5.7.2 n tasks

We consider now the offset assignment for a set of n tasks. Our method constructs non-equivalent asynchronous systems first by considering non-equivalent choices for O_2 (O_1 is already fixed, e.g., $O_1 = 0$); for each of these non-equivalent choices for O_2 , the method considers next the non-equivalent choices for O_3 regarding the requests of τ_1 and τ_2 (O_1 and O_2 are already fixed), etc.

Suppose that the offsets O_1, \dots, O_{i-1} are fixed and consider the non-equivalent choices for O_i regarding the relative phasings between the requests of τ_i and those of τ_j ($j < i$). The request pattern limited to the requests of the task sub-set $\{\tau_1, \dots, \tau_{i-1}\}$ is periodic with a period of $\text{lcm}\{T_1, \dots, T_{i-1}\}$; from the study of the case $n = 2$ it follows that there are $\gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\}$ non-equivalent choices for O_i , e.g., all the integer values in the half-open interval $[0, \gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\})$.

This method constructs $\prod_{i=2}^n \gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\}$ non-equivalent asynchronous systems; let us now check that all the $\frac{\prod_{i=1}^n T_i}{\text{lcm}\{T_i | i=1, \dots, n\}}$ non-equivalent asynchronous systems are yielded. Consequently, it remains to show that $\prod_{i=2}^n \gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\} = \frac{\prod_{i=1}^n T_i}{\text{lcm}\{T_i | i=1, \dots, n\}}$.

Theorem 5.43

$$\prod_{i=2}^n \gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\} = \frac{\prod_{i=1}^n T_i}{\text{lcm}\{T_i | i = 1, \dots, n\}}.$$

Proof. We show the property by induction on n . The property is obvious in the trivial case, $n = 2$, since $\gcd\{T_1, T_2\} = \frac{T_1 \times T_2}{\text{lcm}\{T_1, T_2\}}$. Suppose the property is true up to $n - 1$ and consider the case of n . By induction hypothesis we have that

$$\prod_{i=2}^n \gcd\{T_i, \text{lcm}\{T_1, \dots, T_{i-1}\}\} = \frac{\prod_{i=1}^{n-1} T_i}{\text{lcm}\{T_1, \dots, T_{n-1}\}} \cdot \gcd\{T_n, \text{lcm}\{T_1, \dots, T_{n-1}\}\}.$$

and by definition of the function lcm and gcd we have that

$$\begin{aligned} \text{lcm}\{T_1, \dots, T_{n-1}, T_n\} &= \text{lcm}\{T_n, \text{lcm}\{T_1, \dots, T_{n-1}\}\} \\ &= \frac{T_n \times \text{lcm}\{T_1, \dots, T_{n-1}\}}{\gcd\{T_n, \text{lcm}\{T_1, \dots, T_{n-1}\}\}}. \end{aligned}$$

The property follows. ■

The computation of non-equivalent offsets can be resolved by applying the Euclid's algorithm to each pair $(T_i, \text{lcm}\{T_1, \dots, T_{i-1}\})$. It may be noticed that there may exist other methods; we leave again this question to the perspicacity of the mathematicians. The maximal time complexity of this procedure is $O(n \times \log P)$. Consequently the maximal time complexity of our optimal offset assignment is $O(n \times \log P + \frac{\prod_{i=1}^n T_i}{P} \times \mathcal{R})$, where \mathcal{R} is the maximal time complexity of the schedulability test. Remark that the second term dominates in general the first one.

5.8 Dissimilar offset assignment

We have studied in section 5.7 an optimal offset assignment rule which considers "all" offset assignments; more precisely, we have first shown that there are finitely many non-equivalent offset assignments to be considered (i.e., $\prod_{i=2}^n T_i$) and then we have presented a method to reduce this number and consider only the $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent offset assignments. The simplification reduces significantly the number of assignments that the optimal algorithm has to consider,

but this number remains exponential despite simplifications (for instance when $T_1 = T_2 = \dots = T_n$, but then $P = T_1$ is minimal). Moreover, this optimal algorithm is based on the feasibility test for the corresponding asynchronous task set, and we know that the maximal time complexity of such a test is in all generality proportional to P . For these reasons, it seems interesting to define a heuristic offset assignment rule which considers a single value for each offset.

We shall present here a rule to choose a single value for O_i among the non-equivalent possibilities. The optimality of the corresponding offset assignment is not preserved of course, but we shall see that this offset assignment schedules a good proportion of systems which are not schedulable in the synchronous case and that the time complexity of the offset assignment is polynomial in terms of the number of tasks and the maximal period of the system.

Remark that, before applying a sophisticated offset assignment in order to schedule an offset free system, there are some preliminary points to consider.

First, for the various classes of periodic task sets (regarding their deadlines) considered in this work, we can first check if $U \leq 1$ (which is sufficient for late deadline systems using the deadline driven scheduler) and then check if the system is schedulable in the synchronous case. The time complexity of feasibility tests for synchronous systems are in all generality simpler than for asynchronous systems. For a static scheduler in particular, except for arbitrary deadline systems, this can be checked by a pseudo-polynomial algorithm (see Chapter 3 for more details). If the system is not schedulable in the synchronous case, it may exist a judicious choice of the offsets which schedules the system if $U \leq 1$.

For general deadline systems and static scheduling algorithms, the best response time notion (studied in section 3.5) is interesting in the case where a periodic task set is unschedulable in the synchronous case (i.e., $r_i^1 > D_i$); in this case the task set may be schedulable with a judicious choice of the offsets only if the best case response time does not exceed the deadline.

Lemma 5.44 *Let $\Gamma = \{\tau_j = \{T_j, C_j, D_j\} | 1 \leq j \leq n\}$ be a general deadline and offset free task set schedulable with a judicious choice of the offsets O_1, O_2, \dots, O_n and the static priority assignment $\tau_1 > \tau_2 > \dots > \tau_n$ then $\rho_i^* \leq D_i$ where ρ_i^* is the best case response time for a request of τ_i for $i = 1, \dots, n$.*

Proof. Immediate from the definition of ρ_i^* (see section 3.5):

$$\rho_i^* \leq \min_{S_i \leq R_i^k \leq S_i + P_i} \rho_i^k \leq D_i.$$

■

It may be noticed that the condition $\rho_i^* \leq D_i$ is only a necessary condition. If we consider the example introduced previously (Figure 3.15, with $\rho_2^* = 6$) for all choices of O_1, O_2 the worst case occurs and we must have that $D_2 \geq r_2^1 = 9$ in order to have a feasible offset assignment.

If $\rho_i^* \leq D_i$ ($i = 1, \dots, n$) is satisfied, we still have to choose judiciously the offsets, if feasible. Remark that in this case, choosing the offsets randomly is already always a better choice than choosing $O_1 = O_2 = \dots = O_n$.

Remark also that the sufficient condition given by Lemma 5.44 for general deadline and offset free systems for static schedulers may be applied for other classes of periodic task sets and/or dynamic scheduling algorithms, but we do not have an expression of the best case response time in these more general cases. This problem seems difficult: recall that the computation of the best response time was based on the (dual) property that the largest response time occurs for the first request of τ_i in the synchronous case, and that this property does not hold for more general classes of periodic task sets, nor for dynamic scheduling algorithms.

We now come back to the problem of offset assignment and the presentation of our rule. The main principle of our heuristic offset assignment rule is to choose offsets in order to move away from the worst case, i.e., from the synchronous case, as much as possible. We are looking for a rule which chooses the O_i 's without inspecting the (periodic part of the) schedule but ensures that the schedule moves away, as much as possible, from the synchronous case. This problem is not obvious and we have investigated many solutions. Our heuristic is closely based on the manner to estimate if a given offset assignment is close (or not) to the synchronous case.

We estimate the proximity of an offset assignment with the synchronous case by considering the minimal distance between two requests of different tasks (in the periodic part of the schedule); we shall see later that this definition can be refined, we do not give details here. Note that this distance is 0 for synchronous systems (or for asynchronous systems which are equivalent to the synchronous case). The more this interval is large, the more the requests are dissimilar (from the synchronous case) in the whole schedule.

We shall present here our offset assignment rule, which maximizes the minimal distance between two requests of different tasks. For convenience, let us call our rule the *dissimilar offset assignment*.

We introduce our rule by considering first only the requests of τ_i and τ_j . We have seen in section 5.7 that if we consider only the requests of τ_j and those of τ_i , there are $\gcd\{T_i, T_j\}$ non-equivalent choices for O_i (whatever O_j):

$0, 1, \dots, \gcd\{T_i, T_j\} - 1$. Here we are concerned by the time which separates the requests of τ_i and those of τ_j .

Theorem 5.45 *Let $r \in [0, \gcd\{T_i, T_j\})$, If $O_j = O_i + r$ (or $O_i = O_j + r$) the minimum number of time units between a request of τ_i and a request of τ_j is $\min\{r, \gcd\{T_i, T_j\} - r\}$.*

Proof. Without loss of generality we can assume that $O_i = O_j + r$. We show the property by contradiction; suppose that the property is false, i.e., the minimum number of time units between a request of τ_i and a request of τ_j is b with $0 \leq b < \min\{r, \gcd\{T_i, T_j\} - r\}$. In this case we have to distinguish between two cases:

- (i) $\exists k_1, k_2 \in \mathbb{N} : 0 \leq O_j + r + k_1 T_i - (O_j + k_2 T_j) = b$,
- (ii) $\exists k_1, k_2 \in \mathbb{N} : 0 \leq O_j + k_2 T_j - (O_j + r + k_1 T_i) = b$.

The first relation implies that $b \equiv r \pmod{\gcd\{T_i, T_j\}}$, which leads to a contradiction since $0 \leq b < r < \gcd\{T_i, T_j\}$.

The second relation implies that $b \equiv -r \pmod{\gcd\{T_i, T_j\}}$, which leads to a contradiction since $0 \leq b < \gcd\{T_i, T_j\} - r$ or $b + r < \gcd\{T_i, T_j\}$ and $b + r \not\equiv 0 \pmod{\gcd\{T_i, T_j\}}$. ■

It follows from Theorem 5.45 that the minimum number of time units between a request of τ_i and a request of τ_j is $\left\lfloor \frac{\gcd\{T_i, T_j\}}{2} \right\rfloor$ and corresponds to the offset assignment $O_i = O_j + \left\lfloor \frac{\gcd\{T_i, T_j\}}{2} \right\rfloor$ (or $O_j = O_i + \left\lfloor \frac{\gcd\{T_i, T_j\}}{2} \right\rfloor$). Suppose that $\delta = \gcd\{T_i, T_j\} = \max\{\gcd\{T_k, T_r\} | k \neq r\}$ and consider the offset assignment $O_i = 0$ and $O_j = \left\lfloor \frac{\delta}{2} \right\rfloor$. This offset assignment maximizes the minimal distance between two requests of different tasks. Although our criterion is satisfied, we shall apply the same principle for the remaining free offsets, in order to move away from the synchronization of the remaining task requests. Since our criterion is based on the minimal distance between two requests of different tasks, this distance is maximized by considering only the requests of τ_i and τ_j , and we already fixed the offset between τ_i and τ_j , let us now consider the minimal distance between other pairs of tasks. The second offset assignment is based on the same principle, and consider the minimal distance between two requests of different tasks (but the pair (i, j)). Suppose that p, r and δ' are such that $\delta' = \gcd\{T_p, T_r\} = \max\{\gcd\{T_s, T_q\} | q \neq s \text{ and } (s, q) \neq (i, j)\}$. In the second offset assignment we have to distinguish between two cases: O_p and O_r are not already fixed or p (or r) is already fixed (e.g., $p = i$, $r \neq j$). In the first case the rule fixes two more offsets: $O_p = 0$ and $O_r =$

$O_p + \lfloor \frac{\delta'}{2} \rfloor$. The choice $O_p = 0$ is somewhat arbitrary: what is significant is $|O_p - O_r|$; a slight improvement could be choosing O_p randomly in order to avoid a synchronization of the requests of the τ_i and τ_p , or with the assignment $O_p = O_j - \lfloor \frac{\delta'}{4} \rfloor$, but the latter rule seems difficult to adapt for the following offset assignments. In the second case the rule fixes a single offset (e.g., $O_r = O_p + \lfloor \frac{\delta'}{2} \rfloor$). The next offset assignments respect the same principle until the n offsets become fixed.

Remark that this offset assignment rule does not rely on the (priority) scheduling algorithm nor on the kind of deadline (late/general/arbitrary case). The rule only depends on the periods of the system and tries as much as possible to move away from the synchronous case.

We consider now the (maximal) complexity of our offset assignment rule.

Theorem 5.46 *The maximal time complexity of the dissimilar offset assignment rule is $O(n^2(\log T^{max} + \log n))$ and the maximal space complexity is $O(n^2)$.*

Proof. The dissimilar offset assignment rule computes (and stores) first the value $\gcd\{T_i, T_j\}$ for each pair (T_i, T_j) , this can be resolved by applying the Euclid's algorithm to each pair $(T_i, T_j), j \neq i$, hence the time complexity of this procedure is $O\left(\binom{n}{2} \times \log T^{max}\right) = O(n^2 \times \log T^{max})$. After that, the rule sorts these values, with a maximal time complexity of $n^2 \log n^2$ (e.g. with Heapsort [Wei95, Sed92] which has a maximal time complexity of $O(N \log N)$ – notice that the popular Quicksort [SF96] has a better average complexity but a maximal time complexity of $O(N^2)$) and realizes n offset assignments. Hence, the maximal time complexity of the dissimilar offset assignment rule is $O(n^2(\log T^{max} + \log n))$ and the maximal space complexity is $O(n^2)$. ■

We shall present now the evaluation of our heuristic rule. Since monotonic priority assignments are not optimal for offset free systems, an evaluation in this framework would be biased by this non-optimality. We shall rather study the effectiveness of our rule for the dynamic deadline driven scheduler which remains optimal for offset free systems, and we consider general deadline offset free systems. We present here simulation results of our heuristic applied to a large number of task sets chosen randomly. For each task set we first check if the system is schedulable in the synchronous situation (case *a*), if this is not the case, we check if the system is schedulable with randomly chosen offsets (case *b*) and if the system is schedulable with the dissimilar offset assignment (case *c*). Note that if the system is neither schedulable in the synchronous situation, with random rule nor with the dissimilar offset assignment (case *d*), it may exist another offset assignment which schedules the system. But answering to

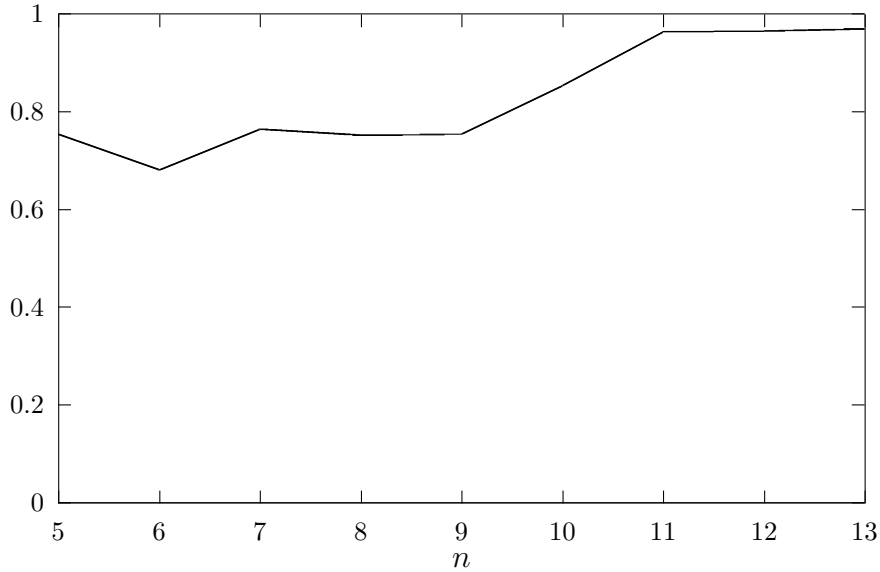


Figure 5.17: Proportion of systems in case d .

the latter question amounts to consider $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent asynchronous systems, more precisely $\frac{\prod_{i=1}^n T_i}{P}$ feasibility intervals, which leads to extremely long computations. For this reason we limit our “classification” on cases a , b and c . Consequently if the system is neither schedulable in the synchronous situation, nor with random rule nor with the dissimilar offset assignment we cannot draw any conclusion on the efficiency of our rule. Consequently, we base our study on systems covered by case a , b and c . Note that if a system is schedulable in the synchronous case, this is also the case with random and dissimilar offset assignments; consequently the verification of the schedulability using these rules is not necessary then.

The maximal time complexity of the classification of a single randomly chosen system is the complexity of the feasibility test for an asynchronous and general deadline system, i.e., $O(P + O^{max})$ as exhibited in Chapter 4, this number grows exponentially with the number of tasks. We have oriented our task set random generation in order to have “critical” systems, where the utilization factor is large (i.e., near 1.0), and the interest of choosing offsets and then our heuristic rule is relevant. In this situation, a large proportion of systems are in the case d (however, we have already seen, in section 5.6 that the interest of offset free systems remains relevant in that case); we have observed that the proportion of systems in case d (i.e., $\frac{\#d}{\#a+\#b+\#c+\#d}$) increases with n (see Figure 5.17), so that a large number of simulations is needed in order to have systems in cases a , b or c and perform our study. For this reason we have

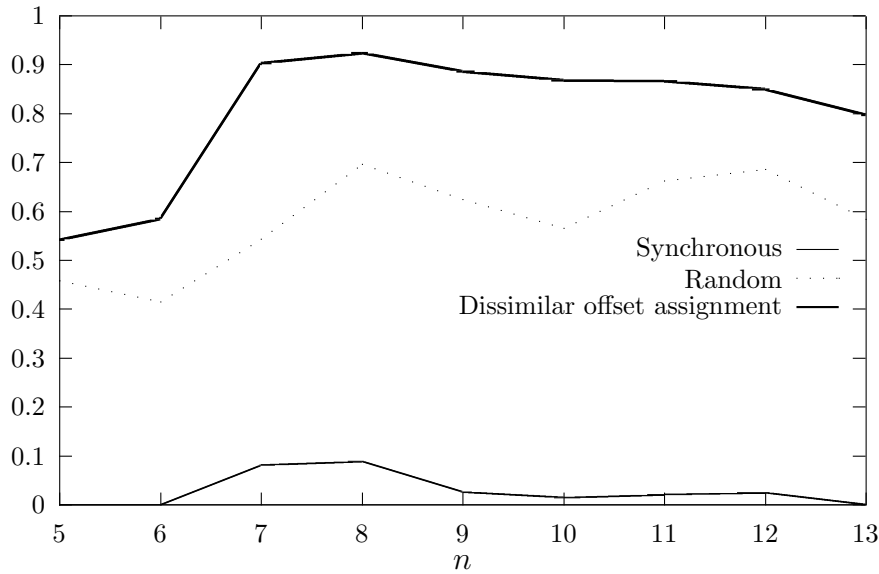


Figure 5.18: Efficiency of our offset assignment rule.

strongly limited n and the T_i 's in our simulations, the number of tasks n is chosen equiprobably in $[5, 15]$, the T_i 's in $[5, 30]$, the D_i 's in $[\frac{T_i}{2}, T_i]$ and then the computation times (C_j) are chosen randomly in the interval $[1, D_j]$ and we only consider systems where the utilization factor is near 1.

Figure 5.18 shows the proportion of systems in case a , b or c in function of the number n of tasks (i.e., $\frac{\#a}{\#a+\#b+\#c}$, $\frac{\#b}{\#a+\#b+\#c}$, $\frac{\#c}{\#a+\#b+\#c}$). From these simulations results, it can be noticed that:

- The interest of offset free systems again occurs in an obvious way, since the proportion of systems in case a is very low in comparison with those in cases b and c .
- In regard with Figures 5.15 it seems that the practical interest of offset free systems is more important for heavily loaded general deadline system using the deadline driven scheduler than for late deadline system with the rate monotonic priority assignment. A finer study of this phenomenon could be interesting but remains for further researches.
- It is very pessimistic to consider the feasibility of systems only in the synchronous case. Choosing the offsets randomly already increases considerably the number of feasible systems.
- Our heuristic offset assignment rule, like the random one, increases considerably the number of feasible systems in comparison with the pes-

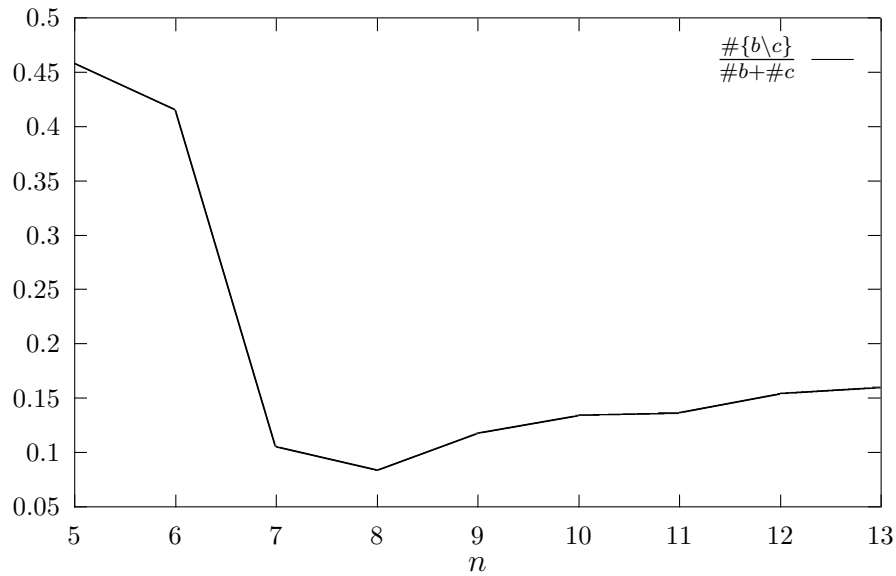


Figure 5.19: Proportion of cases where random rule is ‘better’ than the dissimilar offset assignment.

simistic synchronous case. Moreover, our rule increases nicely the number of feasible systems in comparison with the random rule. Our rule, on the average, schedules 38 % more systems than the random rule. That leads to think that our criterion is a good measure of the proximity with the synchronous situation and the efficiency of our rule.

Note that in our simulations the utilization factor is near 1: we consider “critical systems” where offset free systems and our heuristic rule is relevant. It seems interesting also to consider the phenomenon in function of the utilization factor, but this question remains for further research.

Figure 5.19 shows the proportion of systems where random is a better rule than the dissimilar offset rule (i.e., $\frac{\#\{b \setminus c\}}{\#b + \#c}$ – where $\{b \setminus c\}$ denotes the set of all the systems schedulable with the random rule and not schedulable with the dissimilar rule), except in marginal cases (i.e., $n < 7$), the proportion is about 15 %, which leads again to think that our criterion is a good measure of the proximity with the synchronous situation, and exhibits the efficiency of our rule.

5.9 Conclusion

In this chapter we have studied the scheduling problem of offset free systems. We have first shown that we can restrict the offsets to have the same granularity than the task characteristics; we have then shown that we can restrict the problem to consider $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent offset assignments and we have proposed a method to construct these values. We have also studied the optimality of the popular priority rules. We have shown that the (non) optimality of monotonic priority assignments for offset free systems cannot be reduced to similar results for asynchronous systems, but that the monotonic priority assignments are also not optimal for offset free systems. We have shown the practical interest to consider systems where the offsets can be chosen by the scheduling algorithm, and we have first proposed an optimal offset assignment which considers only the non-equivalent offset assignments; however the number of combinations remains exponential. For this reason, we have defined a rule to choose a single offset for each task, to move away from the worst case. This algorithm is nearly optimal and has a reasonable time complexity in terms of the task characteristics.

Interesting questions for further research related to offset free systems include: the study of optimal (or pseudo-optimal, i.e., heuristic) static priority assignment for offset free systems; statistical analysis of the practical interest of offset free systems with other random variables or with “real” systems; statistical analysis of the actual benefit of our dissimilar offset assignment with other random variables, with “real” systems and in function of the utilization factor; the study of other pseudo-optimal offset assignment rules for offset free systems,...

Bibliography

- [Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, University of York, England, 1991.
- [BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *The Journal of Real-Time Systems*, 2:301–324, 1990.
- [CS47] C. Chiu-Shao. *Shu Shu Chiu Chang*. C. Chiu-Shao, 1247.
- [Dic19] Leonard Eugene Dickson. *History of the Theory of Numbers*, volume II. Chelsea Publishing Company, 1919.
- [GD97] J. Goossens and R. Devillers. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems*, 13(2):107–126, September 1997.
- [Goo95] J. Goossens. Priority assignment and offset assignment for hard real-time offset free systems. In DARTS'95, *workshop on Design and Analysis of Real-Time Systems*. Université Libre de Bruxelles, November 1995.
- [Knu69] Donald E. Knuth. *The Art of Computer Programming*, volume 2 of *Seminumerical Algorithms*. Addison-Wesley, 1969.
- [LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [Mat81] L. Matthiessen. Le problème des restes dans l'ouvrage chinois swang-king de sun-tsze et dans l'ouvrage ta-yen-lei-schu de yih-hing. *Comptes rendus de l'Académie de Paris*, 92:291–294, 1881.

- [Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, 1992.
- [SF96] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley Publishing Company, 1996.
- [Tin93] K. W. Tindell. Using offset information to analysis static priority pre-emptively scheduled task sets. Technical report, University of York, England, 1993.
- [Tin94] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical report, University of York, England, 1994.
- [Wei95] Mark Allen Weiss. *Data structures and algorithm analysis*. The Benjamin/Cummings Publishing Company, Inc., 1995.

Chapter 6

Conclusion

*Tout s'anéantit, tout périt, tout passe ;
il n'y a que le monde qui reste.
Il n'y a que le temps qui dure.
— Denis Diderot.*

We have considered in this work the scheduling problem of real-time systems. More precisely we have studied the scheduling of hard real-time systems composed of periodic and independent tasks for mono-processor systems. We have considered static as well as dynamic scheduling algorithms. During our study we have shown that from a theoretical as well as practical point of view it is interesting to distinguish between three classes of periodic task sets regarding the relation between the period and the deadline (i.e., late/general and arbitrary deadline systems) and three other classes regarding the offsets (i.e., synchronous/asynchronous and offset free systems).

For the various popular scheduling algorithms (static and dynamic) we have reviewed the literature and we have completed and corrected the theory, mainly concerning the optimality and feasibility tests. We have also considered more general classes of periodic task sets than those generally considered in the literature, including asynchronous and arbitrary deadline systems. We have first shown the interest to consider these more general classes and we have extended the theory, particularly concerning the periodicity and feasibility intervals (for static and dynamic schedulers).

For the various sub-classes of periodic task sets considered in this work we have examined the response time notion. We have defined/extended this notion

to handle general classes of periodic task sets, including asynchronous and arbitrary deadline systems.

For static priority schedulers we have shown the interest of the response time computation regarding the feasibility problems of these more general systems. We have also considered the problem of the computation of these response times. For the various classes of periodic task sets considered in this work, we have proposed several methods for these computations. We have studied the analytical and experimental (time and space) complexity of our algorithms. The study of the response time computation in asynchronous situation (with general deadlines) has provided the material to prove the property “stated” by Liu and Layland concerning the worst case response time.

For the dynamic deadline driven scheduler we have extended the response time computation (not considered in the literature) for the various classes of periodic task sets considered in this work, we have studied the interest of our general response time computation in comparison with previous results for synchronous and asynchronous systems (i.e., regarding the analytical and experimental complexity of the various approaches). For both cases, we have shown the advantage of our approach. First, for asynchronous systems we have shown that the maximal time complexity of our computation exhibits an exponential improvement in comparison with the computation suggested by those of Baruah, Howell and Rosier. We have also shown that the actual time complexity of the computation of Baruah, Howell and Rosier is very large and unreasonable in comparison with the response time approach. For synchronous systems, we examined the worst case response time computation of Spuri and we have shown the pessimism of this approach. We have first corrected and generalized a result of Liu and Layland which point out that we can check the feasibility of synchronous system by checking the deadline until the first idle point. Our test amounts to calculate the response times of the requests which occur before this point. The maximal time complexity of this test exhibits again an exponential improvement in comparison with the approach of Spuri. We have also shown the actual pessimism of the Spuri’s approach and the actual benefit of ours. Hence, we feel to have justified the interest of our general response time computation concerning the feasibility test of synchronous/asynchronous systems for arbitrary deadlines.

We have shown that for static as well as for dynamic schedulers (and for the various classes of periodic task sets considered in this work) it is very pessimistic to consider only the synchronous case, since a system can be unschedulable in the synchronous case while being schedulable in a particular asynchronous situation. For this reason we have considered the scheduling problem of offset free systems. We have first shown that we can restrict the offsets to have the

same granularity than the task characteristics, we have then shown that we can restrict the problem to consider $\frac{\prod_{i=1}^n T_i}{P}$ non-equivalent offset assignments. We have also studied the optimality of the popular priority rules. We have shown that the (non) optimality of monotonic priority assignments for offset free systems cannot be reduced to similar results for asynchronous systems but we have shown that the monotonic priority assignments are again not optimal for offset free systems. We have shown the practical interest to consider systems where the offsets can be chosen by the scheduling algorithm. We have first proposed an optimal offset assignment which considers only the non-equivalent offset assignments; however the number of combinations remains exponential. For this reason, we have defined a rule to choose a single offset for each task, to move away from the worst case. This algorithm is nearly optimal and has a reasonable time complexity in terms of the task characteristics.

Of course, there is a lot of interesting questions left for further researches; we have formulated at the end of each chapter some research directions that could lead to extensions and improvements of our work. It is clear that much more (exciting) research is required to achieve this aim.

List of Symbols

In the following formulas, some letters have specific meanings:

i, j, k, t Integer-valued arithmetic expression

x, y, b Real-valued arithmetic expression

g, h Boolean-valued expression.

| | |
|-------------------------------|---|
| $g \wedge h$ | g and h |
| $g \vee h$ | g or h (inclusive) |
| $x \approx y$ | x is approximately equal to y |
| $ x $ | Absolute value of x |
| $[x, y)$ | Half-open interval: $\{x y \leq x < z\}$ |
| $\lceil x \rceil$ | Ceiling of x , least integer function: $\min_{k \geq x} k$ |
| $\binom{n}{k}$ | Binomial coefficient: $\frac{n!}{(n-k)! \cdot k!}$ |
| C_i | The worst case execution time of task number i |
| $C_S(R, t)$ | The configuration of the schedule at time t |
| δ_i^k | The k^{th} request of task number i |
| $\delta_i^k \succ \delta_j^p$ | δ_i^k has a higher priority than δ_j^p |
| $\delta_i^k \prec \delta_j^p$ | δ_i^k has a lower priority than δ_j^p |
| D_i | The deadline of task number i |
| D^{\max} | $\max\{D_i i = 1, \dots, n\}$ |
| $\epsilon_i(t)$ | The amount of processor time used by the last request of τ_i in the interval $[0, t)$ |
| $\epsilon_i^k(t)$ | The amount of processor time used by the request δ_i^k |
| $\lfloor x \rfloor$ | Floor of x , greatest integer function: $\max_{k \leq x} k$ |
| $j \equiv k \pmod{i}$ | Relation of congruence: $j \pmod{i} = k \pmod{i}$ |
| $\gcd\{j, k\}$ | Greatest common divisor of j and k |
| $\text{lcm}\{j, k\}$ | Least common multiple of j and k |
| $\log_b x$ | logarithm, base b , of x (when $x > 0$, $b > 0$, and $b \neq 1$): the y such that $x = b^y$ |
| $\ln x$ | Natural logarithm: $\log_e x$ |
| \max | Maximum |

| | |
|-------------------|---|
| \min | Minimum |
| mod | modulo function |
| n | The number of tasks |
| $k!$ | k factorial: $1 \times 2 \times \dots \times k$ |
| \mathbb{N} | Natural numbers: $0, 1, 2, 3, \dots$ |
| \mathbb{N}_0 | Strictly positive natural numbers: $1, 2, 3, \dots$ |
| O_i | The offset of task number i |
| O^{\max} | $\max\{O_i i = 1, \dots, n\}$ |
| $O(f(x))$ | Big-oh of $f(x)$ |
| P | The least common multiple of all task period |
| \mathbb{Q} | Rational numbers |
| \mathbb{R} | Real numbers |
| r_i^1 | The response time of the first request of τ_i in the synchronous case |
| ρ_i^k | The response time of the k^{th} request of τ_i |
| ρ_i^* | The best response time of task τ_i |
| R_i^k | $O_i + (k - 1) \cdot T_i$ |
| $S \equiv S'$ | S and S' are equivalent asynchronous systems |
| S | A task set |
| T_i | The period of task number i |
| T^{\max} | $\max\{T_i i = 1, \dots, n\}$ |
| τ_i | Task number i |
| $\tau_i > \tau_j$ | Task τ_i has a higher priority than task τ_j |
| U | $\sum_{i=1}^n \frac{C_i}{T_i}$, the utilization factor |
| \mathbb{Z} | Integer numbers: $\dots, -2, -1, 0, 1, 2, \dots$ |

Bibliography

- [AB93] N. C. Audsley and A. Burns. Real-time system scheduling. Technical report, University of York, England, 1993.
- [ABD⁺95] N. C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *The Journal of Real-Time Systems*, 8, 1995.
- [ABDW94] N. C. Audsley, A. Burns, R. I. Davis, and A. J. Wellings. Integrating best effort and fixed priority scheduling. In *Proceedings of the 1994 Workshop on Real-Time Programming*, June 1994.
- [ABRT93] N. C. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.
- [ABRW92] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Welling. Deadline monotonic scheduling theory. In Boullard and Puente, editors, *Proc. IFAC/IFIP WRTP'92*, pages 55–60, Bruges, Belgium, 1992.
- [ABRW93] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science & Engineering*, 2:80–89, 1993.
- [ABRW94] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Stress: a simulator for hard real-time systems. Technical report, University of York, England, 1994.
- [Alt96] Peter Altenbernd. *Timing Analysis, scheduling, and allocation of periodic tasks*. PhD thesis, Universitat-GH Paderborn, 1996.
- [Aud90] N. C. Audsley. Deadline monotonic scheduling. Technical report, University of York, England, 1990.

- [Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, University of York, England, 1991.
- [Bak74] K. R. Baker. *Introduction to sequencing and Scheduling*. John Wiley & Sons., 1974.
- [BD96] A. Burns and R. Davis. Choosing task periods to minimise system utilisation in time triggered systems. *Information Processing Letters*, 58:223–229, 1996.
- [BF97] A. A. Bertossi and A. Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research*, pages 429–443, 1997.
- [BHR93] S. K. Baruah, R. R. Howell, and L. E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoret. Comput. Sci.*, 118:3–20, 1993.
- [BHR93] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoret. Comput. Sci.*, 1(118), 93.
- [BMR90] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In IEEE Computer Society Press, editor, *11th Real-Time Systems Symposium*, pages 182–190, 1990.
- [BMS93] Ozalp Babaoglu, Keith Marzullo, and Fred B. Schneider. A formalization of priority inversion. Technical report, University of Bologna, 1993.
- [BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *The Journal of Real-Time Systems*, 2:301–324, 1990.
- [BS88] T. P. Baker and Alan Shaw. The cyclic executive and ada. IEEE Computer Society Press, pages 120–129, 1988.
- [BS90] Kenneth R. Baker and Gary D. Scudder. Sequencing with earliness and tardiness penalties: A review. *Operation Research*, 38(1):22–27, January-February 1990.

- [Bur94] A. Burns. Fixed priority scheduling with deadlines prior to completion. Technical Report YCS212, University of York, England, 1994.
- [Bur93] A. Burns. Incorporating flexibility into offline scheduling for hard real-time systems. Technical report, University of York, England, 93.
- [But97] Giorgio C. Buttazzo. *Predictable Scheduling Algorithms and Applications*. Hard Real-Time Computing System. Kluwer Academic Publishers, 1997.
- [BW95] Alan Burns and Andy Wellings. A computational model for fixed priority scheduling. M.S. in parallel computer and computation, Warwick University, March 1995.
- [CC89] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transaction on Software Engineering*, 15(10), October 1989.
- [Che87] Infan Kuok Cheong. *Scheduling Imprecise Hard Real Time Jobs with Cumulative Error*. PhD thesis, University of Illinois, 1987.
- [Che93a] Ken Chen. Scheduling in real-time systems: Problems and algorithms. In *Le salon des solutions informatiques temps-réel*, 1993.
- [Che93b] H. Chetto. Des tâches sporadiques en présence de tâches périodiques. In *Le salon des solutions informatiques temps réel*, pages 39–52, Janvier 1993.
- [CL86] Hung-Yand Chang and Miron Livny. Distributed scheduling under deadline constraints: a comparison of sender-initiated and receiver-initiated approaches. *IEEE Computer Society Press*, 1986.
- [CL88] Jen-Yao Chung and Jane W. S. Liu. Algorithms for scheduling periodic jobs to minimize average error. *IEEE Computer Society Press*, 1988.
- [CLBJM95] Claudine Chaouiya, Sophie Lefevre-Barbaroux, and Alain Jean-Marie. Real-time scheduling of periodic tasks. *Scheduling Theory and its Applications*, 1995.
- [CLL90] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transaction on Computers*, 39(9):1156–1174, 1990.

- [Cof76] E.G. Jr. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, 1976.
- [CS47] C. Chiu-Shao. *Shu Shu Chiu Chang*. C. Chiu-Shao, 1247.
- [CS88] Sheng-Chang Cheng and John A. Stankovic. Scheduling algorithms for hard real-time systems, a brief survey. In *IEEE Computer Society Press*, 1988.
- [CSB89] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *The Journal of Real-Time Systems*, 1:265–281, 1989.
- [CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *The Journal of Real-Time Systems*, 2:181–194, 1990.
- [Dav93] R. I. Davis. Approximate slack stealing algorithms for fixed priority pre-emptive systems. Technical report, University of York, England, 1993.
- [Del94] Joelle Delacroix. *Un controleur d’ordonnancement temps réel pour la stabilite du earliest deadline en surcharge: le regisseur*. PhD thesis, Université Pierre et Marie Curie, 1994.
- [Del96] Joelle Delacroix. Towards a stable earliest deadline scheduling algorithm. *The Journal of Real-Time Systems*, 1996.
- [Der74] M. Dertouzos. Control robotics: the procedural control of physical processes. In *Proceedings of the IFIP Congress*, 1974.
- [DGH96] S. De Vroey, J. Goossens, and Ch. Hernalsteen. A generic simulator of real-time scheduling algorithms. In *The 29th Simulation Symposium*, pages 242–249, April 1996.
- [dGHM98] A. de Jacquier, J. Goossens, Ch. Hernalsteen, and Th. Massart. A simulator of real-time systems, specified in et-lotos. In B. Bodnar, editor, *Applied telecommunication Symposium*, pages 147–154. The Society for Computer Simulation International, April 1998.
- [Dha77] S. K. Dhall. *Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1977.

- [Dic19] Leonard Eugene Dickson. *History of the Theory of Numbers*, volume II. Chelsea Publishing Company, 1919.
- [DL78] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January-February 1978.
- [GD97] J. Goossens and R. Devillers. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems*, 13(2):107–126, September 1997.
- [GH98] J. Goossens and Ch. Hernalsteen. A tool for statistical analysis of hard real-time systems. In IEEE Computer Society Press, editor, *Proceedings of The 31st Annual Simulation Symposium*, pages 58–65, Boston, Massachusetts, April 1998.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, a guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
- [Goo95] J. Goossens. Priority assignment and offset assignment for hard real-time offset free systems. In DARTS’95, *workshop on Design and Analysis of Real-Time Systems*. Université Libre de Bruxelles, November 1995.
- [Goo96] J. Goossens. General response time computation for hard real-time periodic task sets. Technical Report 340, Université Libre de Bruxelles, 1996.
- [Goo97a] J. Goossens. Dynamic priority schedulers for hard real-time periodic task sets. Technical Report 358, Université Libre de Bruxelles, 1997.
- [Goo97b] J. Goossens. Feasibility test for synchronous periodic task sets based on the response time computation. Technical Report 355, Université Libre de Bruxelles, 1997.
- [Goo97c] J. Goossens. Static preemptive schedulers for hard real-time periodic task sets. Technical Report 356, Université Libre de Bruxelles, 1997.
- [GS88] J. B. Goodenough and L. Sha. The priority ceiling protocol: a method for minimizing the blocking of high priority ada tasks. *Ada Letters*, VIII(7), 1988.

- [HaL94] Michael Gonzalez Harbour and Mark H. Klein and John Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transaction on Software Engineering*, 20(1), January 1994.
- [HL88] Kwang S. Hong and Joseph Y-T. Leung. On-line scheduling of real-time tasks. *IEEE Computer Society Press*, 1988.
- [HL89] Kwang Soo Hong and Joseph Y-T. Leung. Preemptive scheduling with release times and deadlines. *The Journal of Real-Time Systems*, 1, 1989.
- [Hor74] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.
- [HR96] N. Homayoun and P. Ramanathan. Dynamic priority scheduling of periodic and aperiodic tasks in hard real-time systems. *The Journal of Real-Time Systems*, 6:207–232, 1996.
- [HXT89] Jiawei Hong, Xiaonan, and Don Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Transaction on Computers*, 38(12):1736–1744, 1989.
- [Jen93] E. Douglas Jensen. A scheduling model for scaleable real-time computer systems. In *Le salon des solutions informatiques temps réel*, pages 5–21, Janvier 1993.
- [Jos85] M. Joseph. On a problem in real-time computing. *Information Processing Letters*, 20:173–177, 1985.
- [JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.
- [KAS93] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transaction on Software Engineering*, 19(9), 1993.
- [KLR94] M. H. Klein, J. P. Lehoczky, and R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. *IEEE Computer*, 27(2), 1994.
- [KMRS91] G. Koren, B. Mishra, A. Raghunathan, and D. Shasha. On-line schedulers for overloaded real-time systems. Technical report, Courant Institute, New York University, 1991.

- [Knu69] Donald E. Knuth. *The Art of Computer Programming*, volume 2 of *Seminumerical Algorithms*. Addison-Wesley, 1969.
- [Kor92] Jan Korst. *Periodic Multiprocessor Scheduling*. PhD thesis, Technische Universiteit Eindhoven, 1992.
- [KS91] G. Koren and D. Shasha. An optimal scheduling algorithm with a competitive factor for real-time systems. Technical report, Courant Institute, New York University, 1991.
- [Lab74a] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps réel. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, B-1:11–17, février 1974.
- [Lab74b] Jacques Labetoulle. Some theorems on real time scheduling. *Computer Architectures and Networks*, 1974.
- [LB92] Sophie Lefevre-Barbaroux. *Files d'attente avec arrivés atypiques: environnement aléatoire et superposition de flux périodiques*. PhD thesis, Université de Paris-Sud Centre d'Orsay, 1992.
- [Lee90] Jan Van Leeuwen. *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*. Elsevier, 1990.
- [Leh90] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1990*, pages 201–213, Lake Buena Vista, Florida, USA, December 1990.
- [Leu89] Joseph Y.-T. Leung. An new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- [LLN87] Jane W. S. Liu, Kwei-Jay Lin, and Swaminathan Natarjan. Scheduling real-time, periodic jobs using imprecise results. IEEE Computer Society Press, 1987.
- [LM80] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

- [LSD89] John Lehoczky, Liu Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium*, pages 166–171, 1989.
- [LSS87] John Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. *IEEE Computer Society Press*, pages 261–270, 1987.
- [LSsY91] Jane W. S. Liu, Wei-Kuan Shih, and Albert Chuang shi Yu. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.
- [LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.
- [Mac98] Christophe Macq. Etude pratique des principaux algorithmes d’ordonnancement de tâches aperiodiques en présence de tâches periodiques dans un système temps réel. Master’s thesis, Université Libre de Bruxelles, Belgique, 1998.
- [Mat81] L. Matthiessen. Le problème des restes dans l’ouvrage chinois swang-king de sun-tsze et dans l’ouvrage ta-yen-lei-schu de yih-hing. *Comptes rendus de l’Académie de Paris*, 92:291–294, 1881.
- [MC70] R. R. Muntz and E. G. Coffman. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the Association for Computing Machinery*, 17(2):324–338, April 1970.
- [MD78] A. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Seventh Texas Conference on Computing Systems*, 1978.
- [MF75] Graham McMahon and Michael Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, May-June 1975.
- [Moi85] Abha Moitra. Analysis of hard real-time systems. Technical report, Cornell University–Department of Computer Science, 1985.
- [Mok83] Aloysius Ka-Lau Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

- [NZM91] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & Sons, Inc., 1991.
- [Oh94] Yingfeng Oh. *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems*. PhD thesis, University of Virginia, May 1994.
- [OS93a] Yingfeng Oh and Sang H. Son. Preemptive scheduling of tasks with reliability requirements in distributed hard real time systems. Technical report, University of Virginia, 1993.
- [OS93b] Yingfeng Oh and Sang H. Son. Scheduling of hard real-time tasks with 1-processor-fault-tolerance. Technical Report CS-93-27, University of Virginia, 1993.
- [Par92] Chang Yun Park. *Predicting deterministic execution times of real-time programs*. PhD thesis, University of Washington, 1992.
- [PD93] F. Panzieri and R. Davoli. Real time systems: A tutorial. Technical Report UBLCS-93-22, University of Bologna, 1993.
- [PDP93a] F. Panzieri, L. Donatiello, and L. Poretti. Scheduling real time tasks: A performance study. Technical Report UBLCS-93-10, University of Bologna, 1993.
- [PDP93b] F. Panzieri, L. Donatiello, and L. Poretti. Scheduling real time tasks: A performance study. Technical Report UBLCS-93-10, University of Bologna, 1993.
- [PM94] Mihir Pandya and Miroslaw Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. Technical Report CS-TR-94-07, University of Texas, Austin, 1994.
- [Ram95] K. Ramamritham. Dynamic priority scheduling. M.S. in parallel computer and computation, Warwick University, March 1995.
- [RCM96] Ismael Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *The Journal of Real-Time Systems*, 11:19–39, 1996.
- [Ric92] Mike Richardson. The stress hard real-time system simulator. Technical report, University of York, England, 1992.

- [RISaL87] Ragunathan Rajkumar and Liu Sha and John Lehoczky. On countering the effects of cycle-stealing in a hard real-time environment. In *IEEE Computer Society Press*, 1987.
- [RSZ89] Krithi Ramaratham, John A. Stankovic, and Wei Zhao. Distributed scheduling of tasks with deadline and resource requirements. *IEEE Transaction on Software Engineering*, 38(8):1110–1122, August 1989.
- [Sah79] Sartaj Sahni. Preemptive scheduling with due date. *Operations Research*, 27(5):925–934, September-October 1979.
- [Sak94] Manas Chandra Saksena. *Parametric Scheduling for Hard Real-Time Systems*. PhD thesis, University of Maryland, 1994.
- [SB96] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The Journal of Real-Time Systems*, 10, 1996.
- [Sch92] Werner Schutz. On the testability of distributed real-time systems. Technical Report 3092, University of Vienna, Austria, 1992.
- [Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, 1992.
- [Ser72] Omri Serlin. Scheduling of time critical processes. In *the 1972 Spring Joint Computer Conference*, volume 40 of *AFIPS Conference Proceedings*, 1972.
- [SF96] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley Publishing Company, 1996.
- [SG90] Liu Sha and John B. Goodenough. Real-time scheduling theory and ADA. *IEEE Computer*, pages 53–62, April 1990.
- [SHH91] Alexander D. Stoyenko, V. Carl Hamacher, and Richard C. Holt. Analysing hard-real-time programs for guaranteed schedulability. *IEEE Transaction on Software Engineering*, 17(8):737–749, August 1991.
- [SLL90] Wei Kuan Shih, J. W. S. Liu, and C. L. Liu. Scheduling periodic jobs with deferred deadlines. Technical Report UIUCDCS-R-90-1593, University of Illinois at Urbana-Champaign, 1990.

- [SLR86] Liu Sha, John P. Lehoczky, and Rangunathan Rajkumar. Solution for some practical problems in prioritized preemptive scheduling. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium*, 1986.
- [SLS88] Brinkley Sprunt, John Lehoczky, and Liu Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. IEEE *Computer Society Press*, 1988.
- [Spu96] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, 1996.
- [SR] John A. Stankovic and Krithi Ramamritham. Tutorial: Hard real-time systems. IEEE.
- [SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1990.
- [SR91] John A. Stankovic and Krithi Ramamritham. The spring kernel a new paradigm for real-time systems. IEEE *Software*, 8(3):62–72, May 1991.
- [SRL90] Liu Sha, Rangunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. IEEE *Transaction on Software Engineering*, 39(9):1175–1185, September 1990.
- [SS87] John Lehoczky and Liu Sha and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environment. IEEE *Computer Society Press*, 1987.
- [SSK⁺94] Jack Stankovic, Kang Shin, Herman Kopetz, Krithi Ramamritham, and al. Real-time computing: A critical enabling technology. Technical report, University of Massachusetts, 1994.
- [SSL89] Brinkley Sprunt, Liu Sha, and John Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.
- [SSW94] David B. Shmoys, Clifford Stein, and Joel Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, June 1994.

- [ST93] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. Technical report, Cornell University, 1993.
- [Sta92] John A. Stankovic. Distributed real time computing: The next generation. Technical report, University of Massachusetts, 1992.
- [Sta93] John Stankovic. Reflective real-time system. Technical report, University of Massachusetts, June 28, 1993.
- [SW93] John A. Stankovic and Fuxing Wang. The integration of scheduling and fault tolerance in real time systems. Technical report, University of Massachusetts, 1993.
- [TaMS96] T. Tia and J. W.-S. Liu and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *The Journal of Real-Time Systems*, 10:23–43, 1996.
- [TBW92] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Real-Time Systems Symposium*, December 1992.
- [Tin93a] K. W. Tindell. An extensible approach for analysing fixed priority hard real-time tasks. Technical report, University of York, England, 1993.
- [Tin93b] K. W. Tindell. Using offset information to analysis static priority pre-emptively scheduled task sets. Technical report, University of York, England, 1993.
- [Tin94a] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical report, University of York, England, 1994.
- [Tin94b] K. W. Tindell. An extensible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 1994.
- [Tin94c] Kenneth William Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, England, 1994.
- [TLS94] Too-Seng Tia, Jane W.-S. Liu, and M. Shankar. Aperiodic request scheduling in fixed-priority preemptive systems. Technical report, University of Illinois at Urbana-Champaign, 1994.

- [Var96] T. Vardanega. Tool support for the construction of statically analysable hard real-time ada systems. In *Proceedings of the Real-Time Systems Symposium*, 1996.
- [vTK91] André M. van Tilborg and Gary M. Koob. *Scheduling and Resource Management*. Foundations of Real-Time Computing. Kluwer Academic Publishers, 1991.
- [Wan93] Fuxinf Wang. *Issues related to dynamics scheduling in real time systems*. PhD thesis, University of Massachusetts Amherst, 1993.
- [Wei95] Mark Allen Weiss. *Data structures and algorithm analysis*. The Benjamin/Cummings Publishing Company, Inc., 1995.
- [XP88] Jia Xu and David Lorge Parnas. Scheduling algorithms for hard real-time systems—a brief survey. *IEEE Transaction on Software Engineering*, 1988.
- [XP90] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadline, precedence, and exclusion relations. *IEEE Transaction on Software Engineering*, 16(3):360–369, 1990.
- [XP93] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transaction on Software Engineering*, 19(1), 1993.
- [YS91] Michal Young and Lih-Chyun Shu. Hybrid online/offline scheduling for hard real-time systems. Technical report, Purdue University, 1991.
- [Zal] Januz Zalewski. Real-time systems glossary. Glossary.
- [ZRS87] Wei Zhao, Krithivasan Ramamritham, and John A. Stankovic. Scheduling tasks with ressource requirements in hard real-time systems. *IEEE Transaction on Software Engineering*, SE-13(5):564–576, 1987.

Index

- asynchronous, 11
- busy period, 40
- cyclic executive, 9
- deadline, 4
 - arbitrary, 11, 36, 52, 65, 94, 97, 103, 135, 155
 - general, 11, 34, 43, 67, 129, 148
 - hard, 4
 - late, 11, 20
 - soft, 4
- deadline monotonic, 34
- dynamic priority, 9
- equivalent systems, 206, 207, 226
- expedient, 25
- failure, 3
- feasibility interval, 38, 42, 48, 58, 126, 131, 133, 138, 140
- feasibility test, 94, 180
- fully utilize, 28
- hyper-period, 28
- idle, 8
 - point, 39, 84, 120
- laxity, 186
- least laxity first scheduling algorithm, 186
- non-preemptive, 9
- offset
 - granularity, 201
 - limited growing, 210
- offset free systems, 11, 200
- optimality, 21, 26, 34, 36, 44, 52, 125, 212, 215, 226
 - definition, 116, 214
 - non-, 44, 219
- optimality, 117
- period, 7
- precedence constraint, 4
- predictability, 3
- preemptive, 9
- rate monotonic, 20
- real-time scheduling, 8
- relative phasing, 47, 211, 219
- relative urgency algorithm, 124
- request, 5, 7
 - active, 8
- resource requirement, 5
- response time, 22, 64
 - 1st request, 65, 142
 - k^{th} request, 67, 97, 148, 155
 - best case, 88
 - worse, 172
 - worst case, 85, 172
- schedulability test, 103
- schedulable, feasible, 21
- schedule, 12, 20, 113
 - extended, 134, 135
 - feasible, 13
 - partial, 48
 - partially extended, 128, 129

- unfeasible, 13
- separation time, 210–212
- stability, 59, 192
- static priority, 9, 19, 112
- synchronous, 11, 140, 166
- task, 3, 5
 - aperiodic, 8
 - critical, 59
 - dependent, 5
 - independent, 5
 - periodic, 7, 10
 - request, 5
 - sporadic, 7
- timing constraint, 4