ULB

PUB
PRESSES UNIVERSITAIRES DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES

Real-Time Operating Systems

Joël GOOSSENS

Yannick MOLINGHEN

D/2024/0098/089
2 e édition – Tirage 2024-25/1
INFO-F-404_Z

9 782500 015708

« C'est l'éducation, l'instruction qui décideront de l'émancipation de la femme et de l'homme, de leur tolérance et de leur capacité à dialoguer. »

Lucia de Brouckère (1904-1982)
Docteur en chimie de l'ULB, première femme à enseigner dans une faculté de sciences en Belgique, militante de la laïcité et du libre-examen.

UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F404

Real-Time Operating Systems

Instructor:
Joël GOOSSENS

Authors:
Joël GOOSSENS,
Yannick MOLINGHEN,
Pascal TRIBEL

24th May 2024

L'article premier des statuts de notre Université proclame que son enseignement a pour principe le *libre examen*. Celui-ci postule, en toute matière, le rejet de l'argument d'autorité et l'indépendance de jugement.

Based on the principle of free enquiry that postulates independent reasoning and the rejection of all dogma, the University has retained its original ideals as a free institution that is firmly engaged in the defence of democratic and human values. (ulb.be/en/about-ulb).

CONTENTS

I Introduction to this curriculum guide **1**

1 Preamble **3**

II Uniprocessor scheduling **5**

2 Introduction to real-time systems **7**

2.1 Real-time systems 7
2.2 Examples 9
2.3 Real-time, embedded or Cyber-Physical Systems? 9
2.4 A an introductory example: adaptive cruise control in a car 10
2.5 Model of computation and assumptions 12
2.5.1 Periodic task 12
2.5.2 Diagrams conventions 13
2.5.3 Tasks vs. jobs 14
2.5.4 Tasks produce jobs 14
2.5.5 Sporadic tasks 15
2.5.6 System utilisation 15
2.5.7 Different kinds of tasks 16
2.5.8 Scheduling 17
2.5.9 Assumptions 18
2.5.10 Job response time 18
2.5.11 Earliest Deadline First (EDF) 18
2.5.12 Schedulability conditions 19

3 Fixed Task Priority (FTP) assignment **21**

3.1 Introduction . . . 21

3.2 Proof by induction . . . 21

3.3 Rate Monotonic (RM) . . . 22

3.3.1 Definitions . . . 22

3.3.2 Response time of the first job . . . 26

3.3.3 Schedulable utilisation of Rate Monotonic (RM) . . . 28

3.4 Deadline Monotonic (DM) . . . 28

3.4.1 Fixed Task Priority (FTP) optimality of Deadline Monotonic (DM) . . . 29

3.5 Arbitrary deadlines . . . 29

3.5.1 Feasibility interval . . . 30

3.6 Asynchronous systems . . . 31

3.6.1 Feasibility interval . . . 32

3.7 AUDSLEY FTP assignment . . . 35

3.7.1 AUDSLEY algorithm . . . 36

4 Fixed Job Priority (FJP) Assignment **39**

4.1 Earliest Deadline First (Earliest Deadline First (EDF)) . . . 39

4.1.1 Optimality . . . 39

4.1.2 Schedulability tests . . . 40

5 Dynamic Priority (DP) assignment **45**

5.1 Least Laxity First (LLF) . . . 45

5.1.1 Least Laxity First (LLF) Optimality . . . 45

5.1.2 Least Laxity First (LLF) vs. EDF . . . 46

III Multiprocessor scheduling **47**

6 Introduction & motivation **49**

6.1 MOORE's law . . . 49

6.2 Scheduling problem . . . 49

6.2.1 Limitations . . . 49

6.2.2 Taxonomy of multiprocessor platforms . . . 50

6.2.3 Taxonomy of multiprocessor schedulers . . . 50

6.3 Illustrating partitioning . . . 51

6.4 Illustrating global scheduling . . . 51

6.4.1 Partitioned and global scheduling are incomparable 52

7 Partitioning **55**

7.1 Introduction 55

7.1.1 Heuristics 55

7.2 Partitioned Rate Monotonic 56

7.3 First Fit Decreasing Utilisation (FFDU) 56

8 Global Scheduling **59**

8.1 Negative results 59

8.1.1 No online optimal scheduler exists 59

8.1.2 Scheduling anomalies 60

8.2 Positive results 61

8.2.1 Periodic implicit-deadline Systems 61

8.2.2 Global EDF 63

8.2.3 EDF$^{(k)}$ scheduling 64

IV Parallel Algorithms **67**

9 Parallel algorithms **69**

9.1 The speed-up factor 69

9.2 Cost of parallelisation 70

9.3 Maximum speed-up factor: AMDAHL's law 70

9.4 Parallel sorting algorithms 71

9.4.1 Rank sort 71

9.4.2 Bitonic sort 72

V Concurrency **75**

10 Introduction **77**

10.1 Concurrent execution 77

10.2 State diagrams 78

10.3 Scenario 78

10.4 Atomic statements 79

11 Critical sections with 2 processes **81**

11.1 Problem definition 81
11.2 First protocol attempt 82
11.3 Second protocol attempt 83
11.4 Third protocol attempt 83
11.5 A fourth protocol attempt 85
11.6 DEKKER's algorithm 86

12 Critical sections with *N* processes **87**
12.1 Bakery algorithm 87

13 Semaphores **89**
13.1 Introduction 89
13.2 Process state 89
13.3 The semaphore data type 90
13.4 Critical section problem with semaphores 91
13.4.1 Critical section with two processes 91
13.4.2 Critical section with *N* processes 91
13.5 Synchronisation 92
13.5.1 Merge sort 92
13.5.2 Producer-consumer problem 92

14 Distributed algorithms **95**
14.1 Introduction 95
14.2 Distributed critical section problem 95
14.3 Consensus 97
14.3.1 Byzantine generals problem 99
14.3.2 One round algorithm 99
14.3.3 Byzantine generals algorithm 100

VI Exercises **103**

15 Uniprocessor scheduling exercises **105**
15.1 Rate Monotonic (RM) 105
15.2 Deadline Monotonic (DM) 106
15.3 Systems with arbitrary deadlines 107
15.4 AUDSLEY 107

15.5 Earliest Deadline First (EDF) . . . 107
15.6 Least Laxity First (LLF) . . . 108

16 Simulation exercises **109**
16.1 Introduction . . . 109
16.2 Constant step simulation . . . 109
16.3 Exercises . . . 110

17 Multiprocessor scheduling exercises **113**
17.1 Global and partitioned RM . . . 113
17.2 Partitioned EDF . . . 113
17.3 EDF$^{(k)}$ . . . 114

18 Concurrency exercises **115**
18.1 Protocol 1 . . . 115
18.2 Protocol 2 . . . 116
18.3 Protocol 3 . . . 116
18.4 Protocol 4 . . . 117
18.5 Protocol 5 . . . 118

Appendices **119**

A Additional proofs **121**

B Additional algorithms **123**

Index **125**

Glossary **127**

List of symbols **129**

Bibliography **129**

PART I

INTRODUCTION TO THIS CURRICULUM GUIDE

CHAPTER 1

PREAMBLE

This document constitutes the courses notes for INFO-F404 “Real-Time Operating Systems”, which is compulsory for the students following the Master’s in Computer Science, Faculty of Sciences, Université libre de Bruxelles. The goal of this course is to give an introduction to the key theoretical foundations and techniques needed to design real-time and embedded systems, as well as to provide an introduction to parallel and distributed programming.

Throughout the text, two symbols are used:

This is a particularly important concept.

This is a question to make you think and to activate your critical thinking.

Acknowledgement of Joël GOOSSENS. I would like to thank Nell FOSTER for her contribution to the English language quality of this curriculum guide within the TEA project [1]. I would like to thank my mentors who introduced me to the scientific rigour and discipline of scheduling theory: Profs. Raymond DEVILLERS (ULB) and Sanjoy BARUAH (Washington University in Saint Louis, USA). Finally, I would like to dedicate this curriculum guide to the memory of my late colleague Prof. Laurent GEORGE (1967–2021).

1

PART II

UNIPROCESSOR SCHEDULING

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

CHAPTER 2
INTRODUCTION TO REAL-TIME SYSTEMS

In this chapter we will introduce the notion real-time systems as well as the abstract model of computation that is used throughout this curriculum guide.

2.1 Real-time systems

We begin by discussing the specificities of real-time systems and the scheduling problems associated with their use.

Real-time systems are computing systems which have timing constraints. Thus, the correctness of such a system depends not only on its logical result of computations, 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 settings, for example:

- to control of engines,
- to run chemical and nuclear plants,
- traffic control systems,
- time-critical packet communications,
- flight control systems,
- railway switching systems,
- robotics,
- military systems,
- space missions, and
- virtual reality.

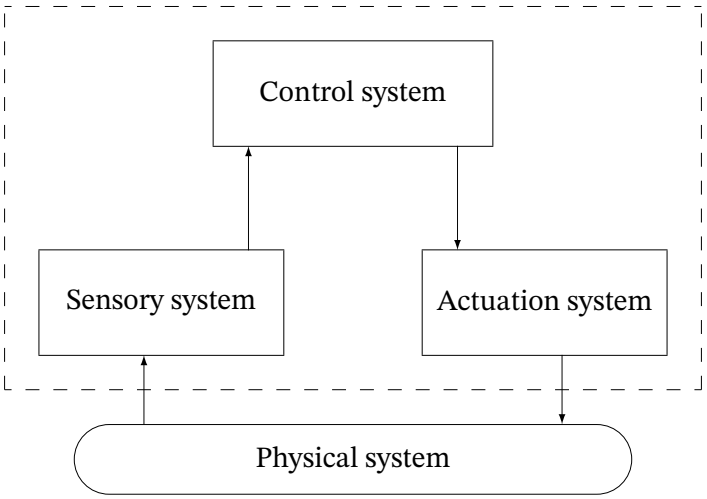


Figure 2.1: Real-time control system

Figure 2.1 shows the architecture of a typical real-time system for controlling a *physical system*.This physical system might be a plant, a car, a robot, or any physical device that has to exhibit a desired behaviour. 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 is also important to note that a real-time system evolves in interaction with the physical system. As a consequence, the system time 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 know as *online* systems. Note that online systems are not necessarily real-time systems, since online systems do not necessarily have (hard) timing constraints.

In many real-time systems the consequences of *failure* (e.g. if a timing constraint is not respected) can be catastrophic and lead to serious damage or even loss of human life. 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, keeping the time for computation to a minimum helps to meet 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 about the important principle of *predictability* [23], i.e., the ability to predict, a priori, whether the system can meet all hard (also known as *critical*) timing requirements.

We can define the term *job* as a certain activity or computation. Thus a task is a set of specific jobs. In real-time systems, each job has a certain deadline, meaning that there is a

moment at which the job should be completed, and a Worst-Case Execution Time (WCET).

We can use the concept of deadlines to already make a distinction between three different kinds of (real-time) systems:

Definition 1 (*Hard real-time*)

Absolutely no deadline can be missed (it could have health, financial or ecological consequences).

Definition 2 (*Firm real-time*)

Ideally no deadline should be missed. However, some deadline misses may be tolerated, in which case we define a Quality of Service (QoS). For example, in video conferencing or live streaming, the system must ensure that video frames are transmitted and displayed within a specific time frame to maintain a smooth and uninterrupted video stream. Any delay in transmitting or displaying video frames can result in jitter, buffering, or even loss of data.

Definition 3 (*Soft real-time*)

Deadlines can be missed.

2.2 Examples

A good way to understand the need of real-time constraints is to look at examples.

The weather forecast: The computation of such forecasts might be too demanding, but must be completed in the time available i.e. before the deadline which is the moment at which we want to know the weather.

Adaptive Cruise Control: This is a system that adapts a car speed in order to keep an safe distance with the vehicles driving in front of the car. As such it is an example of more critical deadline.

2.3 Real-time, embedded or Cyber-Physical Systems?

"Cyber-Physical System (CPS) has emerged as a unifying name for systems where the cyber parts, i.e., the computing and communication parts, and the physical parts are tightly integrated, both at design time and during operation. It covers a very wide range of application domains, ranging from extremely light-weight medical devices to nation-scaled power grids. Cyber-Physical System (CPS) often involves multiple disciplines, such as real-time systems, sensor networks, embedded systems, control, information processing, and signal processing".[1]

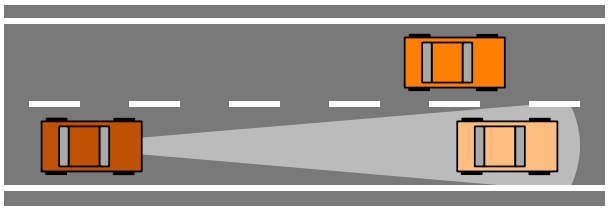
Thus, CPSs are a more general class of techniques that includes Real-Time Systems and Embedded Systems.

[1] Tei-Wei Kuo, ACM Transactions on Cyber-Physical Systems Inaugural Issue, February 2017.

Which of these applications/systems are real-time?

- Pacemaker
- Excel
- Airbus A320
- Nuclear power plant management system
- Gmail

2.4 A an introductory example: adaptive cruise control in a car



Adaptive cruise control has multiple characteristics:

- A critical constraint: keeping the car a safe distance from the vehicle in front;
- A sensor *periodically* determines the distance and the speed of the vehicle in front.

In order to work, the system has to:

- τ_1: Update the information on the driver's dashboard;
- τ_2: Automatically adjust the speed;
- τ_3: Disengage the cruise control if the driver brakes.

Why is adaptive cruise control "real-time"?

We can model this application using three real-time *periodic* tasks: τ_1, τ_2 and τ_3 (see Table 2.1). Every 20 units of time the task τ_1 (re-)executes the same piece of code (i.e., a job of τ_1 is released) in order to refresh the information displayed on the driver's dashboard. This operation takes at *most* 4 units of time but can be preempted. Every 10 units of time, a new job of τ_2 is released in order to adjust the speed. This operation takes at most 2 units of time. Finally, every 5 units of time, τ_3 releases a new job. This operation may consist of disabling the system (i.e., if the driver braked), and takes at most 3 units of time. For the sake of

Table 2.1: AAC model

Task	Execution time	Period (& deadline)	Load
τ_1	4	20	20%
τ_2	2	10	20%
τ_3	3	5	60%
		Total	100%

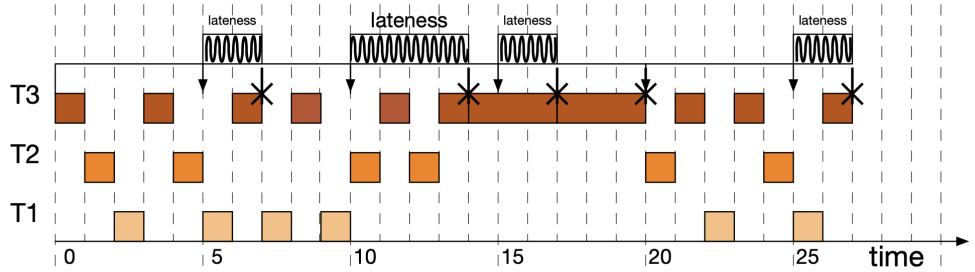


Figure 2.2: Round-Robin execution

simplicity we assume that for the three tasks, the deadline for each job corresponds to the next arrival of the task (that is, the deadline equals the period). For instance the deadline of the first job of τ_1 is 20. The load (or the utilisation — to be explained in the next chapter) is the fraction of the Central Processing Unit (CPU) that is required by the task. For instance $20\% = 4/20$ for τ_1. The total load (utilisation) is 100% (that is, $1 = 4/20 + 2/10 + 3/5$). Which scheduler might be relevant in this real-time context in order to share the CPU and meet the deadlines. A popular scheduler for *non* real-time (general purpose operating) systems is Round-Robin.

Round-Robin [24]. The Round Robin illustrated in Figure 2.2 is a technique where each task periodically receives an equal share of the processing resource — e.g. the CPU — in order to provide a good *average* process response time. The key idea is to assign time slices in equal portions to each process in a *circular order*. This technique is a typical scheduling scheme for most general purpose operating systems (unix, linux, windows). Figure 2.2 shows that in our set of tasks, the Round Robin is deadly for τ_3.

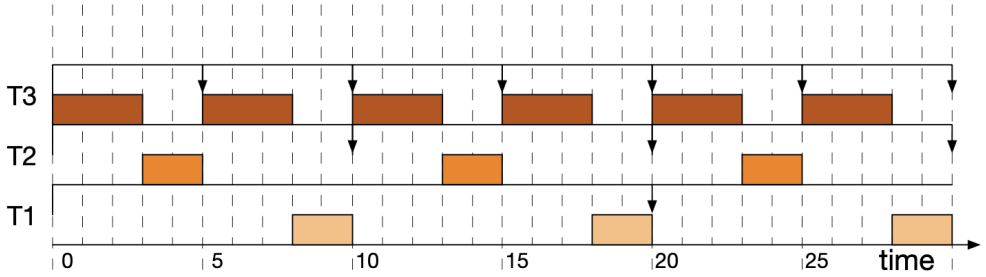


Figure 2.3: Rate Monotonic execution

Rate Monotonic [18]. Rate Monotonic is a preemptive scheduling algorithm used in Real-Time Operating System (RTOS), where each task τ receives a static priority P_τ. This priority is assigned to the task by looking at its periodicity: the shorter the period, the higher the priority. In our example system, τ_3 receives the highest priority, τ_2 the middle priority and τ_1 the lowest priority. The schedule produced is shown in Figure 2.3. In our example, we observe that with the RM scheduler, all tasks are processed on time. Note that the schedule repeats from time marker 20 (the least common multiple of the task's periods).

2.5 Model of computation and assumptions

We now have the material to provide a first and more formal definition of real-time systems.

Definition 4 (*Real-time systems*)

Real-time systems *are defined as systems in which the correctness of the system depends not only on the logical result of computations, but also on the* time *at which the results are produced.*

Definition 5 (*Job*)

A real-time job *is characterized by the tuple* $\langle a, e, d \rangle$*, where:*

- *a corresponds to the release time.*
- *e corresponds to the maximal execution time —which is known as the Worst-Case Execution Time (WCET).*
- *d corresponds to the* absolute *deadline with the interpretation that in the interval* $[a, d)$ *the job must receive e CPU units.*

A real-time instance is a collection (usually infinite) of jobs $J = \{j_1, j_2, j_3, ...\}$.

Definition 6 (*Active job*)

A job is said to be active *from its release time to its completion time. In other words an active job is a job already released but not completed.*

2.5.1 Periodic task

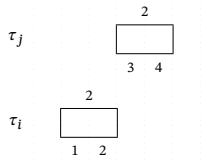


Figure 2.4: Graphical representation of the execution of two tasks

Definition 7 (*Periodic task*)

A periodic task τ_i *is characterized by the tuple* $\langle O_i, C_i, D_i, T_i \rangle$*, where:*

- O_i (the offset): *corresponds to the release time of the* first *job of the task.*
- C_i (the maximal computation time): *corresponds to the Worst-Case Execution Time (WCET).*
- D_i (the *relative* deadline): *corresponds to the time-delay between a job release and its corresponding deadline. A job released at time t must be completed before or at time $t + D_i$.*
- T_i (the period): *corresponds to the* exact *duration between two consecutive task releases.*

2.5.2 Diagrams conventions

In this section we introduce the graphical conventions used in this curriculum guide to represent the execution of the various systems. The system execution starts at time $t = 0$ and the diagrams show what happens for each time unit from time unit 0. The time units are numbered sequentially, starting from value 0.

Consider a simple case where the system is composed of a single task. If the task is executing during the interval $[a, b]$ (and consequently during $c = b - a + 1$ consecutive time units), this execution is represented by the picture: 

In the special case where $a = b$ (and consequently the task is executing during 1 time unit), b is omitted, this execution is represented by the picture: 

Imagine we now have several tasks (say τ_i and τ_j). In order to distinguish between the execution of τ_i and τ_j, we show the execution of each task on a separate level. The time units of each level coincide (see Figure 2.4 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 active (released but not completed) job of each task at each time instant; later, we shall refine the conventions to represent more general behaviours.

Moreover, we represent a *job release* which occurs at time t by the picture  where the

symbol ↓ is placed at the beginning (i.e., on the left) of the time unit number t. We represent the *job deadline* which occurs at time t by the picture ○ 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 💣 which is placed at the beginning of the time unit number t.

2.5.3 Tasks vs. jobs

A *task*:

- is an offline concept;
- has parameters known at design time (e.g.: the *relative* deadline);
- is recurring;
- induces a (theoretically) infinite set of jobs
- there is a fixed number (n) of tasks in the system.

A *Job*:

- is a runtime concept;
- has parameters only known at runtime (e.g. the *absolute* deadline);
- is not recurring (executed once);
- can be executed;
- exists independently of the notion of task
- there is a potentially infinite number of jobs.

At runtime, what does the system schedule?

- Tasks?
- Jobs?
- Both?

2.5.4 Tasks produce jobs

A job could be induced by a task, and we define $J_{i,k}$ as the k^{th} job of τ_i ($k \in \mathbb{N}$). $J_{i,k}$ has a duration of C_i, is released at $O_i + (k-1)T_i$, and has an absolute deadline of $O_i + (k-1)T_i + D_i$ (illustrated at Figure 2.5 for jobs $J_{1,1}$ and $J_{1,2}$ for $T_1 = 10, C_1 = 3, D_1 = 5, O_1 = 0$).

Figure 2.5: An example of how the jobs of a periodic task can be represented

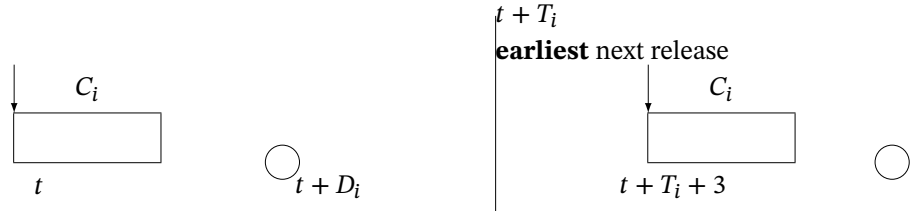


Figure 2.6: Example of a sporadic task. Two subsequent jobs of τ_i are separated by a delay *larger* than the period T_i

2.5.5 Sporadic tasks

Sporadic tasks are similar to periodic tasks, there is an important difference in the meaning of the "period" parameter.

Definition 8 (*Sporadic task*)

Sporadic tasks *are similar to periodic tasks, the only difference being that the period of a sporadic task denotes the* minimum *inter-arrival time instead of the exact one. An example example of the way this is represented graphically is given in Figure 2.6.*

The scheduling problem of each category is thus different:

- The scheduling of periodic tasks is an *offline* problem (we know all the characteristics of the jobs at design time).
- The scheduling of sporadic tasks is an *online* problem (we don't know all the characteristics of the jobs at design time, the release time in particular).

What is the main difference between sporadic and periodic tasks?

2.5.6 System utilisation

The utilisation of a task is the proportion of a shared resource — typically the CPU — that this task utilises. The utilisation of a task τ_i is defined as[2] by definition:

$$U(\tau_i) \stackrel{\text{def}}{=} \frac{C_i}{T_i}$$

A system is composed of a *finite* set of tasks (n in this curriculum guide): $\tau \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_n\}$. The system utilisation is the sum all of the task utilisations.

[2]The notation $\stackrel{\text{def}}{=}$ means equal

Definition 9 (*System utilisation*)

$$U(\tau) \stackrel{\text{def}}{=} \sum_{i=1}^{n} U(\tau_i) = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

What is the system utilisation?

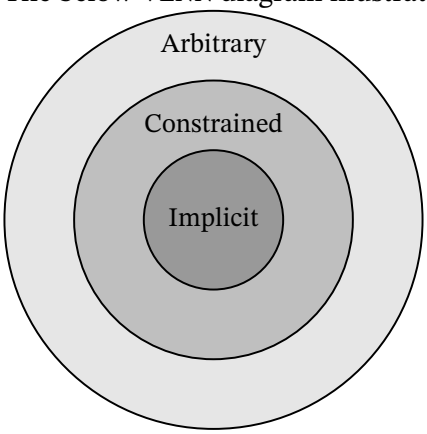
- The exact utilisation of the CPU at any time?
- The exact utilisation of the CPU for an arbitrary long period of time?
- Just an indication of the CPU utilisation?

2.5.7 Different kinds of tasks

Regarding the deadlines

- *Implicit-deadline:* the deadline corresponds to the period, i.e., $D_i = T_i; \forall i$. Each job must finish before the next release of the task.
- *Constrained-deadline:* the deadline is explicit and not greater than the period, i.e. $D_i \leqslant T_i, \ \forall i$.
- *Arbitrary-deadline,* the general case: no constraint exists between the period and the deadline.

Implicit deadline systems are also one kind of constrained deadline systems. Constrained deadline systems are also one kind of arbitrary deadline systems. The below VENN diagram illustrates this relationship:



Regarding the offsets

- *Synchronous:* the first job of every task is released simultaneously at time origin, i.e., $O_i = 0, \ \forall i$.

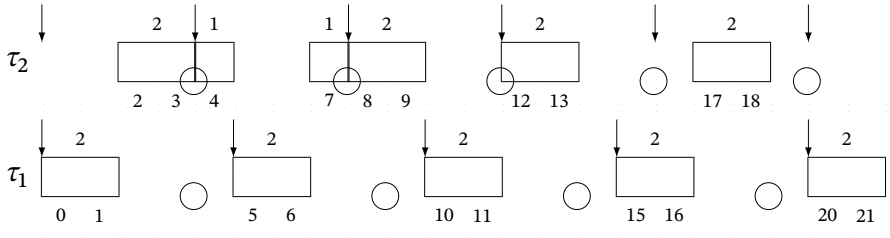


Figure 2.7: Example of FTP scheduling with $\tau = \{\tau_1 = (T_1 = 5, D_1 = 4, C_1 = 2), \tau_2 = (T_2 = 4, D_2 = 4, C_2 = 2)\}$ assuming that τ_1 receives the highest priority

- *Asynchronous,* the general case: no constraint exists on the offsets.

2.5.8 Scheduling

The main goal of the scheduling algorithm (the scheduler) in General Purpose Operating Systems (GPOS) is to minimise resource starvation and to ensure fairness (like Round-Robin) amongst the parties utilising the resources. Scheduling deals with the problem of deciding which of the program is to be allocated resources. There are many different scheduling algorithms.

In real-time environments, such as embedded systems for automatic control in industry (e.g. robotics), the scheduler must also ensure that processes meet deadlines; this is crucial for keeping the system stable or safe.

The CPU scheduler decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt (an I/O interrupt, an operating system call or another form of signal).

Definition 10 (*Preemptive*)

A scheduler can be preemptive*, meaning that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process.*

In this curriculum guide, we distinguish three kinds of scheduling algorithms:

- Fixed Task Priority (FTP) a fixed and unique priority is assigned to each task at design time. During system execution, the priority of each job is inherited from its task. An example is given in Figure 2.7. The highest priority task (τ_1) is executed *asap* without preemption, while task τ_2 can progress only when task τ_1 is not active. The task τ_2 can be preempted, this is the case for instance at time instant 5, the second job of τ_2 is resumed at time instant 7.
- Fixed Job Priority (FJP) a fixed and unique priority is assigned to each *job* during system execution. Several jobs of a same task may have distinct priorities.
- Dynamic Priority (DP) no restrictions are placed on the priorities that may be assigned to jobs. A job may have different priority levels over the duration of its execution.

2.5.9 Assumptions

Unless we explicitly state the opposite, we make the following assumptions in the framework of uniprocessor systems.

- The time space is discrete;
- We consider hard real-time systems;
- We consider preemptive tasks/jobs;
- Preemption delays are negligible;
- Context switch delays are negligible;
- The scheduler is work-conserving: the CPU cannot be idle while active jobs exist (see Definition 26, page 24 for a formal definition);
- Tasks are independent: except for the CPU, there is no other common resource shared among tasks, nor any kind of precedence constraint between tasks.

2.5.10 Job response time

Definition 11 (*Job response time*)

The response time of a job is the completion time minus the release time.

For instance, in Figure 2.7, response time of

- job $J_{1,1}$ is 2 because it was released at $t = 0$ and completed at $t = 2$;
- job $J_{2,1}$ is 4 because it was released at $t = 0$ and completed at $t = 4$;
- job $J_{2,2}$ is 4 because it was released at $t = 4$ and completed at $t = 8$

2.5.11 Earliest Deadline First (EDF)

While the study of the EDF scheduler will be covered in Chapter 4, we introduce it here to present a second kind of scheduler.

Definition 12 (*Earliest Deadline First (EDF)*)

EDF is an Fixed Job Priority (FJP) (Fixed Job Priority) scheduler, where the priority is based on the absolute *job deadline: the lower the absolute deadline, the higher the priority.*

An example of EDF scheduling is given in Figure 2.8 for System 13. This schedule illustrates that the priorities are *dynamic* at task level. For instance in the interval $[0, 3]$ task τ_1 has the highest priority, but in the interval $[4, 7]$ task τ_2 receives the highest priority. Of course at *job* level the priorities are constant (static).

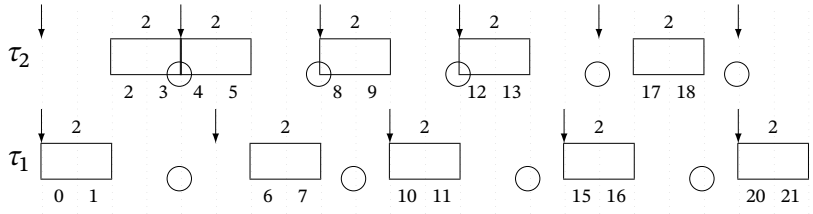


Figure 2.8: Example of EDF, a FJP scheduler, scheduling the System 13

System 13

τ_i	C_i	D_i	T_i
τ_1	2	4	5
τ_2	2	4	4

2.5.12 Schedulability conditions

In this curriculum guide we will study schedulability conditions. It is important to note that there are 3 types of conditions.

Necessary condition. Suppose that A and B are two statements. We say that statement A is *necessary* for the statement B if B cannot be true unless A is also true. In other words, B requires A. However it is possible for A to be true even if B is not true. The notation is $B \Rightarrow A$. (A is necessary for B.)

Sufficient condition. Statement A is said to be a *sufficient* condition for the statement B if knowing that A is true guarantees that B is also true. However knowing that B is true does not guarantee that A is true. That is, B does nor need to be a sufficient condition for A. The notation is $A \Rightarrow B$. (A is sufficient for B.)

Necessary and sufficient condition. We say that statement A is a *necessary and sufficient* condition for statement B when B is true if and only if A is also true. That is, either A and B are both true, or they are both false. Note that if A is necessary and sufficient for B, then B is necessary and sufficient for A. The notation is $A \Leftrightarrow B$. (A and B are equivalent.)

The very first necessary condition, regarding the schedulability of any uniprocessor system is the condition $U(\tau) \leqslant 1$, more formally:

Lemma 14 (*A necessary condition*)

Let τ be a periodic task-set, then $U(\tau) \leqslant 1$ is a necessary *condition for system schedulability.*

Do you have an idea of how to prove the Lemma 14?

CHAPTER 3
Fixed Task Priority (FTP) ASSIGNMENT

3.1 Introduction

This chapter explores Fixed Task Priority (FTP) scheduling algorithms and in particular how fixed priorities can be assigned to periodic and sporadic tasks.

Definition 15 (*FTP ($\tau_i \succ \tau_j$)*)

The notation $\tau_i \succ \tau_j$ means that τ_i has a higher priority than τ_j.

Unless we explicitly state the opposite, we assume to have the following *strict total* order:

$$\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n.$$

3.2 Proof by induction

Before studying FTP schedulers, it is important to remember the notion of *proof by induction* essential in scheduling theory (in particular for the FTP family).

The main idea is to identify an *induction axiom* with the following definition.

Definition 16 (*Induction axiom*)

Let $\mathbb{P}(n)$ be the a predicate (a symbol which represents a property). If $\mathbb{P}(0)$ is true and $\forall n \in \mathbb{N}\ (\mathbb{P}(n) \Rightarrow \mathbb{P}(n+1))$ is true, then $\mathbb{P}(n)$ is true $\forall n \in \mathbb{N}$.

If $\mathbb{P}(0)$ is true, $\mathbb{P}(0) \Rightarrow \mathbb{P}(1)$, $\mathbb{P}(1) \Rightarrow \mathbb{P}(2)$, ... then $\mathbb{P}(0), \mathbb{P}(1), \mathbb{P}(2)$, ... are true. By domino effect the property is true for any n.

Consider an example to illustrate the proof technique.

Theorem 17 (*Triangular numbers*)

$$\forall n \geqslant 0, \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof. By induction. Let $\mathbb{P}(n)$ be proposition that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Base case. $\mathbb{P}(0)$ is true: $\sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$.

Inductive step. For $n \geqslant 0$ show that $\mathbb{P}(n) \Rightarrow \mathbb{P}(n+1)$ is true. Assume $\mathbb{P}(n)$ is true for purposes of induction (i.e., assume $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$) need to show $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$.

I.e., $1 + 2 + \cdots + n + n + 1 = \frac{n(n+1)}{2} + n + 1 = \frac{n^2+n+2n+2}{2} = \frac{(n+1)(n+2)}{2}$. □

3.3 Rate Monotonic (RM)

Originally, RM [18] was defined for *synchronous* systems (the first job of every task is released simultaneously at time origin, i.e., $O_i = 0;\ \forall i$) and *implicit deadline* systems (the jobs deadlines are the same as the periods).

Priority assignment. It assigns, at design time, the priority to the task according to the rate/period. The lower the period, the higher the priority. If two tasks have the same period (i.e., a tie), one is allocated the higher priority; this remains fixed during the analysis and at runtime.

Complexity. The algorithm is fundamentally similar to sorting tasks according to their periods. Then its complexity is $\mathcal{O}(n \log n)$).

3.3.1 Definitions

Definition 18 (*Feasible priority assignment*)

A priority assignment is feasible *for a task-set τ if it schedules τ such that no deadline is ever missed.*

Definition 19 (*FTP-feasibility*)

A set of tasks is said to be FTP-feasible *if a feasible FTP-priority assignment exists.*

Definition 20 (*FTP-optimality*)

An FTP-priority assignment (an FTP-scheduler) is FTP-optimal if the assignment schedules all *FTP-feasible sets of tasks.*

Critical instant

Definition 21 (*Scenario*)

A scenario *is a set of release times of jobs allowing to characterise a busy period of the CPU (i.e., an interval where the CPU is executing jobs).*

3.3. RATE MONOTONIC (RM)

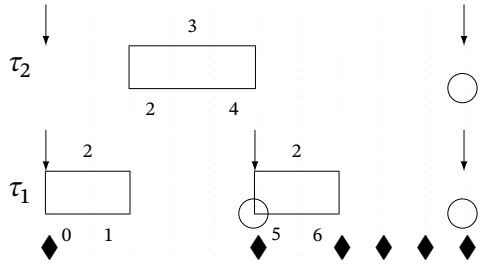


Figure 3.1: Execution of System 24

Definition 22 (*Worst-case scenario*)

For each task τ_i, a worst-case scenario is a scenario which maximises the response time of a job $J_{i,k}$.

Definition 23 (*Idle point*)

We say that $x \in \mathbb{N}$ is an idle *processor* point *if all the jobs released* strictly *before x been fully executed before or at time x.*

If the CPU is idle in the interval $[a, b)$, all the points in that interval are idle points, and we say that $[a, b)$ is an idle period.

As an example, let us take System 24:

System 24

τ_i	C_i	T_i
τ_1	2	5
τ_2	3	10

System 24 is represented in Figure 3.1, where the idle points are represented with the symbol ⧫.

This system (in Figure 3.1) is idle in the interval $[7, 10)$.

For the scheduling of a constrained[1] deadline system, the critical instant is known and corresponds to the synchronous case, more formally:

Lemma 25 (*Critical instant [19]*)

For a constrained deadline system, a critical instant *of any task τ_i occurs when one of its jobs $J_{i,k}$ is released at the same time as the jobs of all higher priority task (synchronously).*

Proof. Let t_0 be the release time of $J_{i,k}$.

Let t_{-1} be the most recent idle point at or before t_0.

Let t_R be the instant where $J_{i,k}$ completes.

The processor is busy from t_{-1} up to t_R (according to the definition of an idle point).

If we move the arrival of higher priority jobs/tasks to the *left*, the response time of the job in question ($J_{i,k}$) can't decrease but it can increase if the interference is larger. The maximal

[1] In this case the property is stated for a more general case (constrained deadline systems) than the one considered in this section. But, by definition, it also holds for implicit deadline systems.

value of the response time of job $J_{i,k}$ is found when the first job of all higher priority tasks are moved left to correspond to the idle point t_{-1}, which means that the synchronous case give the critical instant for $J_{i,k}$. Using the same argument, if the other jobs are released as soon as possible (i.e., we consider periodic job releases not to be sporadic) we get the worst-case response time. □

Lemma 25 means that checking the schedulability of sporadic systems is *equivalent* to checking the schedulability of the corresponding *synchronous* periodic system, because this synchronous periodic system is the worst-case scenario.

Work conservation

Definition 26 (*Work-conserving scheduler*)

A scheduler is said to be work-conserving *if it does not leave resources (e.g. the processor) idle when there are unfinished released jobs.*

Lemma 27

Let τ be a periodic task system, and let S_1 and S_2 be two schedules of work-conserving *schedulers for τ. The schedule S_1 is idle at instant t* if and only if *the schedule S_2 is idle at instant t, $\forall t \geqslant 0$.*

Proof. By induction. Let $\mathbb{P}(n)$ be proposition that $\forall t \leqslant n, S_1(t) =$ idle if and only if $S_2(t) =$ idle.

Base case. $\mathbb{P}(0)$ is true: since $t = 0$ is the time origin of the system.

Inductive step. For $n \geqslant 0$ show that $\mathbb{P}(n) \Rightarrow \mathbb{P}(n+1)$ is true. Assume $\mathbb{P}(n)$ is true for purposes of induction.

Let t_1 be the last idle unit (Without Loss of Generality (WLoG)[2] in S_1) strictly before $n+1$ ($t_1 = 0$ if there is no idle unit strictly before $n+1$).

Case a. $S_1(n+1) =$ busy. Since S_1 is work-conserving there is no idle unit in $(t_1, n+1]$, but since S_2 is also work-conserving and needs to serve the same workload in $(t_1, n+1]$ (possibly in a different order) at time $n+1$ the cpu must be busy as well in S_2. $\mathbb{P}(n+1)$ is true.

Case b. $S_1(n+1) =$ idle. Since S_1 is work-conserving we know that all jobs released in $(t_1, n+1]$ have completed their execution at time $n+1$. Since S_2 is work-conserving as well, we must have that the same work is executed in $(t_1, n+1]$ (possibly in a different order). $\mathbb{P}(n+1)$ is true. □

This property (Lemma 27) forms the basis for proving a collection of other properties as the optimality of RM (Theorem 28).

[2] "Without Loss of Generality (WLoG) is a frequently used expression in mathematics. The term is used to indicate the assumption that follows is chosen arbitrarily, narrowing the premise to a particular case, but does not affect the validity of the proof in general." —Wikipedia, September 2022.

3.3. RATE MONOTONIC (RM)

Optimality

Rate monotonic is popular especially because it is *optimal* in the family of FTP schedulers.

Theorem 28 (*Optimality of RM*)

RM is an FTP-optimal priority assignment for synchronous and implicit deadline tasks.

Proof. Let $\tau = \{\tau_1, \dots, \tau_n\}$ be a periodic implicit deadline synchronous task-set. We must prove that if a feasible FTP-priority assignment exists, then the rate monotonic priority assignment is feasible as well.

Suppose the feasible FTP-assignment corresponds to $\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$. Let τ_i and τ_{i+1} be two tasks with *adjacent* priorities and $T_i \geqslant T_{i+1}$. Let us exchange the priorities of a such pair of tasks τ_i and τ_{i+1} to match the priority ordering of rate monotonic. If the task-set is still schedulable, we may deduce that any rate monotonic priority assignment is also feasible. This is because Bubble Sort (see Appendix B, page 123) sorts the entire array by repeatedly swapping two *adjacent* elements a *finite* number of times. Therefore, without loss of generality, we show that the swap $(\tau_i \leftrightarrow \tau_{i+1})$ does not jeopardize the schedulability of the system.

To show that such a priority exchange is feasible, we analyse the schedulability of all tasks and split down the tasks into 4 groups: τ_i, τ_{i+1}, all tasks with a higher priority than τ_i, and all tasks with a lower priority than τ_{i+1}.

1. The priority exchange of τ_i and τ_{i+1} does not modify the priority (hence the schedulability) of the tasks with a higher priority than τ_i (i.e. the task subset $\tau_1, \dots, \tau_{i-1}$).
2. Task τ_{i+1} remains schedulable after the exchange, since its priority has increased. It may now use all the free slots left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}\}$ instead of only those left by $\{\tau_1, \tau_2, \dots, \tau_{i-1}, \tau_i\}$.
3. Assuming that τ_i remains schedulable (demonstrated below), Lemma 27 shows that the scheduling of tasks $\{\tau_{i+1}, \tau_{i+2}, \dots, \tau_n\}$ is not altered since the idle periods left by higher priority tasks are the same.
4. Hence, we only need to verify that τ_i also remains schedulable. Using Lemma 25, we can restrict this question to the *first* job of task τ_i. Let r_{i+1} be the response time of the first job of τ_{i+1} before the priority exchange. Feasibility implies that $r_{i+1} \leqslant D_{i+1}$. It is not difficult to see that during the interval $[0, r_{i+1})$, the CPU (when left free by higher priority tasks) is assigned first to the (first) job of τ_i then to the (first) job of τ_{i+1} (the latter is not interrupted by subsequent jobs of τ_i since $T_i > T_{i+1} = D_{i+1} \geqslant r_{i+1}$). Hence, after the priority exchange, the CPU allocation is exchanged between τ_i and τ_{i+1}, and it follows that τ_i ends its computation at time r_{i+1} and thus meets its deadline since $r_{i+1} \leqslant D_{i+1} = T_{i+1} \leqslant T_i = D_i$.

□

FTP-optimality of RM means that RM can be used to schedule *any* FTP-feasible system for synchronous and implicit deadline systems.
The optimality of EDF (Section 4.1, page 39) is different as it concerns an *arbitrary* set of jobs.

3.3.2 Response time of the first job

From Lemma 25, for synchronous constrained[3] deadline systems, only the response time of the *first* job of each task must be considered in order to determine the schedulability of the system. This is because the system is synchronous, and Lemma 25 gives that this response time is maximal.

In the following, r_i^1 denotes the response time of the first job of τ_i.

Theorem 29 (*Synchronous constrained deadline system schedulability*)

A synchronous constrained deadline system is FTP-schedulable iff:

$$r_i^1 \leqslant D_i \ \forall i$$

Note. "if and only if" (shortened as *"iff"*) is a biconditional logical connective between statements, where either both statements are true or both are false.

This r_i^1 is the smallest positive solution to this equation (details in [4]):

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

The *ceiling* function ($\lceil x \rceil$ which maps x to the smallest integer greater than or equal to x) is important since $\left\lceil \frac{r_i^1}{T_j} \right\rceil$ is the number of job(s) of τ_i released in the time interval $[0, r_i^1)$, i.e., an natural number.

System 30

τ_i	C_i	T_i	D_i
τ_1	2	8	8
τ_2	3	11	11
τ_3	5	15	15

Illustration. Consider the System 30 with $\tau_1 \succ \tau_2 \succ \tau_3$ and the response time of the first job of τ_3:

$$r_3^1 = C_3 + \mathbf{2} \cdot C_1 + \mathbf{2} \cdot C_2 = 5 + 4 + 6 = 15$$

As illustrated by Figure 3.2.

Indeed, the interval $\left[0, r_i^1\right)$ includes $\left\lceil \frac{r_i^1}{T_j} \right\rceil$ higher priority jobs of τ_j ($j < i$).

Because the term r_i^1 occurs on *both* sides of the Equation 3.1, the *minimal* solution has to be computed *exactly* by fixed-point iteration:

[3] Again, the property is stated for a more general case (constrained deadline systems) than the one considered in this section. But, by definition, it holds for implicit deadline systems.

3.3. RATE MONOTONIC (RM)

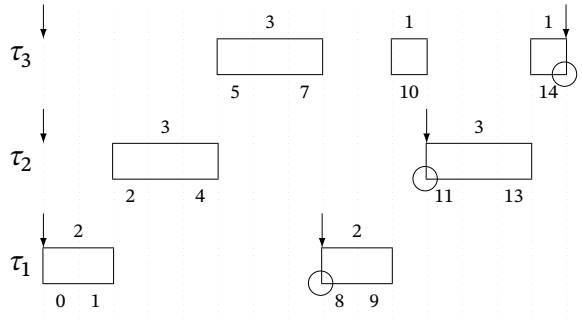


Figure 3.2: The response time r_3^1 for the task-set $\{\tau_1 = \{T_1 = D_1 = 8, C_1 = 2\}, \{T_2 = D_2 = 11, C_2 = 3\}, \{T_3 = D_3 = 15, C_3 = 5\}\}$ with $\tau_1 \succ \tau_2 \succ \tau_3$

$$\begin{cases} w_0 & \stackrel{\text{def}}{=} C_i & (initialisation) \\ w_{k+1} & = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{w_k}{T_j} \right\rceil C_j & (iteration) \end{cases} \tag{3.2}$$

We can stop this computation when $w_k = w_{k+1}$ (the value is exact) or when $w_k > D_i$ (one job will not finish on time), in the second case we are not interested by the exact value of the response time.

Find a system with 3 tasks ($\tau_1 \succ \tau_2 \succ \tau_3$) where Equation 3.1 has several (positive) solutions for r_3^1.

A corollary of the Theorem 29 is that *only* things that happen in the interval $[0, \max\{D_i \mid i = 1, \ldots, n\})$ are required to determine whether the (whole) system is schedulable or not. This type of interval, here $[0, \max\{D_i \mid i = 1, \ldots, n\})$, is known as a *feasibility interval* with the following definition:

Definition 31 (*Feasibility interval*)

A feasibility interval *is a* finite *interval of time such that, if no deadline is missed considering* only *the jobs released within this interval, then we can conclude that no deadline will be missed during the system's lifetime.*

Corollary 32 (*The feasibility interval* $[0, D^{\max})$)

For synchronous constrained deadline tasks,

$$[0, \max\{D_i \mid i = 1, \ldots, n\})$$

is a feasibility interval.

Proof. This is a direct consequence of Theorem 29. □

3.3.3 Schedulable utilisation of RM

For RM (see Lemma 14, page 19) $U(\tau) \leqslant 1$ is necessary, not sufficient. The following theorem provides a *sufficient* condition for RM-schedulability.

Theorem 33 (*RM schedulability test with* $U(\tau)$ *[16, 10]*)

A set of n implicit deadline synchronous periodic tasks can be feasibly scheduled with RM if

$$U(\tau) \leqslant n(\sqrt[n]{2} - 1)$$

The proof of Theorem 33 will not be explored in this curriculum guide, as it is not immediately relevant.

The original proof of Theorem 33 was published in a seminal article by LIU and LAYLAND [18] and has been cited around 14,000 times[4]. However, it is interesting to note that even if the results are correct, the proof itself is flawed. It is therefore necessary to take a critical approach even with published, seminal, quoted documents, or even those by prestigious authors.

Theorem 33 can be considered for huge n, this leads to the corollary:

Corollary 34

A set of n implicit deadline synchronous periodic tasks can be feasibly scheduled with RM if

$$U(\tau) \leqslant 0.69$$

or

$$U(\tau) \leqslant \ln 2$$

Provide a synchronous system with implicit deadlines and more than 2 tasks that is *not* schedulable with $0.69 < U(\tau) \leqslant 1$.

To summarise: for RM-schedulability (implicit deadline, synchronous periodic/sporadic tasks) we can distinguish between 3 situations:

- $U(\tau) \leqslant 0.69$: τ is definitely RM-schedulable;
- $0.69 < U(\tau) \leqslant 1$: τ is *maybe* RM-schedulable;
- $U(\tau) > 1$: τ is definitely *not* schedulable.

3.4 Deadline Monotonic (DM)

Deadline monotonic is an FTP assignment algorithm that is defined and used for *synchronous constrained deadline* task-sets. The lower the relative deadline is, the higher the priority. Similar to Rate Monotonic, ties can occur but have to be *deterministically* resolved.

In fact, RM is a particular case of DM, where the deadline is exactly the period.

[4]Source: Google Scholar, March 2024.

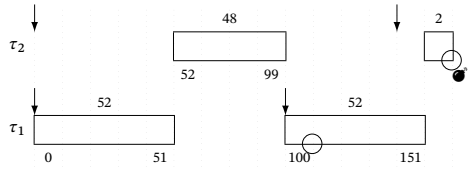


Figure 3.3: The execution of an arbitrary deadline task-set with RM/DM assignment (System 36)

3.4.1 FTP optimality of DM

Theorem 35 (*FTP optimality of DM*)

DM is an FTP-optimal priority assignment for synchronous and constrained deadline tasks.

Proof. We want to prove that if a feasible FTP assignment exists for a synchronous and constrained deadline system, then DM priority assignment is feasible as well.

Let us assume the FTP assignment $\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$ is feasible and that τ_i and τ_{i+1} are two adjacent tasks.

If $D_i \geqslant D_{i+1}$, then we can exchange τ_i and τ_{i+1} and the assignment is still feasible. Consequently, DM can be obtained by applying a bubble sort (see Appendix B) on the tasks according to their deadlines D_i, and this assignment will also be feasible.

In terms of Rate Monotonic optimality, you have to check if the swap of τ_i and τ_{i+1} is allowed. The four cases presented in the proof of Theorem 28 remain valid here, with T_i now being evaluated as D_i. □

3.5 Arbitrary deadlines

Until now, we only have techniques for implicit deadlines (RM), and for constraint deadlines (DM). We now consider the general case: *arbitrary* deadlines.

RM/DM is not optimal. First notice that neither RM nor DM are optimal for arbitrary systems. To prove this, consider a (counter-)example:

System 36 (*An arbitrary deadline system*)

τ_i	C_i	T_i	D_i
τ_1	52	100	110
τ_2	52	140	154

In this situation, the priorities assigned by RM and DM are identical: $\tau_1 \succ \tau_2$. But this assignment leads to $J_{2,1}$ missing a deadline at instant 154 as shown in Figure 3.3.

But if we had assigned priorities in the other order possible ($\tau_2 \succ \tau_1$), we would be successfully executed (see Figure 3.4).

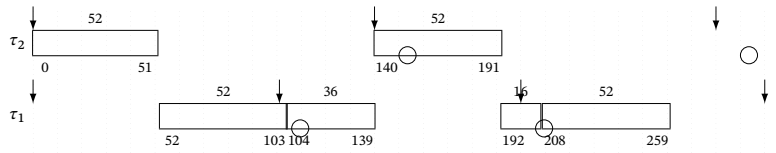


Figure 3.4: FTP-feasible priority assignment for System 36

System 36 requires that two jobs of the *same* task are active *simultaneously* (see time interval $[100, 104)$ on Figure 3.4). More generally, for an arbitrary deadline (since D_i can be larger than T_i), several jobs of the *same* task can be active simultaneously. In that case, the scheduler considers only the *older* active job, i.e., First In First Out (FIFO) is used at task level.

The response time of the first job. Note that for arbitrary deadline systems, the response time of the *first* job is *not* always maximum: in Figure 3.4, the first job of τ_1 starts on 0, and ends on 104, and has a response time of 104. Its second job starts on $t = 100$ and ends on 208, and has a response time of $108 > 104$.

Consequently we need to check the schedulability of *several* jobs of a given task. The question is how many jobs do we need to consider? For this reason, the notion of a feasibility interval is important.

For arbitrary deadline systems, the response time of the first job is *not* significant to conclude schedulability of the task.

3.5.1 Feasibility interval

For synchronous *arbitrary* deadline systems, the first busy period (see Definition 37) is a feasibility interval (see Definition 31, page 27).

Definition 37 (*Elementary busy period*)

An elementary busy period *is a time-interval* $[a, b)$ *such that a and b are idle points and the interval* (a, b) *does not contain any idle points.*

Definition 38 (*Level-i busy period*)

A level-i busy period *is an elementary busy period considering the scheduling of tasks* $\{\tau_1, \ldots, \tau_i\}$ *where* τ_i *executes at least one job.*

Theorem 39 (*Largest level-i busy period*)

The largest response time for a job of task τ_i *occurs during the* first *level-i busy period* $[0, \lambda_i)$ *in the synchronous case.* λ_i *is the first idle point (after* 0*) in the synchronous schedule of the task subset* $\{\tau_1, ..., \tau_i\}$*, and* $[0, \lambda_i)$ *is the* largest level-i busy period.

Proof. Let $[a, b)$ be a level-i busy period considering the scheduling of the task-set $\{\tau_1, \ldots, \tau_i\}$.

Let $a + \Delta_j$ be the instant of the first job of τ_j after instant a ($\Delta_j \geqslant 0$). By definition, $\Delta_k = 0$ for at least one k. Suppose first that $\Delta_i > 0$. Only tasks with a higher priority than τ_i execute in $[a, a + \Delta_i)$.

Thus, if we decrease Δ_i, each job of τ_i in the interval $[a, b)$ will be completed at the same instant, thus increasing the response time. Therefore, the maximum case corresponds to $\Delta_i = 0$.

Now, if $\Delta_j > 0$ ($j < i$), then decreasing Δ_j increases (or does not change) the demand of higher priority jobs in $[a, b)$. This fact increases (or does not change) the length of the busy period.

Consequently, the worst-case corresponds to $\Delta_1 = \Delta_2 = \cdots = \Delta_i = 0$. □

Theorem 40 (*The feasibility interval* $[0, \lambda_n)$)

For synchronous arbitrary deadline systems, $[0, \lambda_n)$ *is a feasibility interval. (Remember,* λ_n *is the first idle point after* 0 *in the synchronous case, here for the whole n-tasks set).*

Proof. This is a direct consequence of $\lambda_1 < \lambda_2 < \cdots < \lambda_n$. □

The computation is similar to Theorem 29. λ_n is the smallest solution to the equation: $\lambda_n = \sum_{i=1}^{n} \left\lceil \frac{\lambda_n}{T_i} \right\rceil C_i$

This value can be computed exactly by iteration:

$$\begin{cases} \lambda_0 & \stackrel{\text{def}}{=} \sum_{i=1}^{n} C_i & (initialisation) \\ \lambda_{k+1} & = \sum_{i=1}^{n} \left\lceil \frac{\lambda_k}{T_i} \right\rceil C_i & (iteration) \end{cases}$$

3.6 Asynchronous systems

Periodicity of the schedules. The schedulability of *asynchronous* systems relies on the *periodicity properties* of their schedules. A first very general property is the fact that *any* deterministic scheduler of periodic tasks produces a *periodic schedule* (see Figure 3.5). Most of the time[5], the period of the schedule is a multiple of the hyper-period (the least common multiple of the task periods; see Definition 41 for a formal definition). This is illustrated in Figure 3.5, with X and k as unknown constants and P as the hyper-period. We are interested in finding exact values for X and k or at least an upper-bound for $X + k \cdot P$.

We can see in Figure 3.5 that the exact schedulability test needs the simulation of the system in the interval $[0, X + k \cdot P)$, with X being the transient phase duration and $k \cdot P$ being the period length. This requires the time to be discrete and the scheduler to be deterministic.

By studying schedule periodicity, it is possible to design feasibility intervals.

[5] In particular cases, it is possible to define schedulers whose periodicity does not depend on the arrival of tasks, but they are irrelevant in practice.

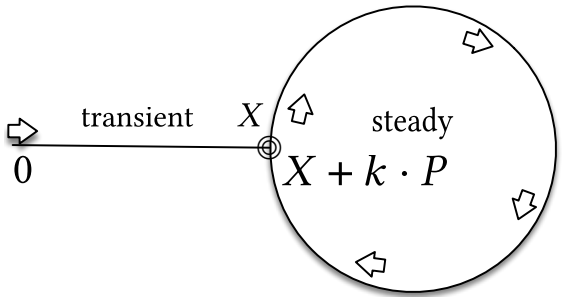


Figure 3.5: Periodic schedule

3.6.1 Feasibility interval for constrained deadline asynchronous systems

The hyper-period, i.e. the period of the period, is defined as follows:

Definition 41 (*Hyper-period*)

The hyper-period *P of a system is defined as:*

$$P \stackrel{\text{def}}{=} \text{lcm}(T_i \mid i = 1, \dots, n)$$

We also note the maximal offset $O_{\max}$ as:

$$O_{\max} \stackrel{\text{def}}{=} \max(O_i \mid i = 1, \dots, n)$$

Consider a system whose tasks have periods 5, 10 and 15. What is its hyper-period (often the periodicity length of the schedule)?

- The product of the task periods (750)
- The maximal task period (15)
- The least common multiple of task periods (30)

Definition 42 (*Deterministic and memoryless scheduler*)

A scheduler is deterministic and memoryless *if and only if the scheduling decision at time t is* unique *and* univocally *defined by the state of the system at time t.*

Theorem 43 (*Periodicity of the schedules*)

For deterministic and memoryless schedulers the feasible schedules of periodic tasks are periodic.

Proof. For any natural instant, time $t \geqslant O_{\max}$, we denote the configuration of a feasible schedule by the tuple $\{(\epsilon_i, \delta_i) \mid 1 \leqslant i \leqslant n\}$, where $\epsilon_i \in \mathbb{N}$ is the cumulative CPU time used by the current job of τ_i and $\delta_i \in \mathbb{N}$ is the time elapsed since its last release.

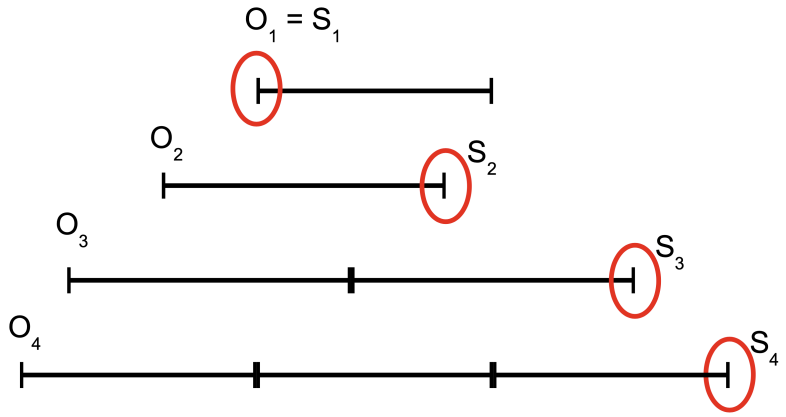


Figure 3.6: S_i values informally represented

Since the configuration of the schedule S at time $t+1$ is univocally determined by the configuration at time $t \geqslant 0$ (since the scheduler is assumed to be deterministic) and the fact that:

$$\forall i \in [1,n] \begin{cases} 0 & \leqslant \epsilon_i(t) & \leqslant C_i \\ 0 & \leqslant \delta_i(t) & < T_i \end{cases}$$

there is a finite number of *different* configurations. Therefore, there exist two instants t_1 and t_2 with $t_1 < t_2$ where the configurations are identical. Hence, the schedule will repeat periodically with a period dividing $t_2 - t_1$. □

The question is to determine *when* the cyclical behaviour of the system begins as well as the period of this cyclic behaviour (respectively X and P in Figure 3.5). One seminal result by LEUNG and MERRILL is the following:

Theorem 44 (*The feasibility interval* $[O_{\max}, O_{\max}+2P)$*, details [17]*)

For asynchronous constrained deadline systems, if the schedule is feasible up to $O_{\max}+2P$, it is feasible and periodic from $O_{\max}+P$ with a period of P.

A better result is the following:

Theorem 45 (*The feasibility interval* $[0, S_n+P)$*, details [12]*)

For any asynchronous constrained deadline system ordered by decreasing priorities ($\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$), let S_i be inductively defined by:

$$S_i = \begin{cases} S_1 \stackrel{\text{def}}{=} O_1 & \text{(initial case)} \\ S_i \stackrel{\text{def}}{=} O_i + \left\lceil \dfrac{(S_{i-1}-O_i)^+}{T_i} \right\rceil T_i & (i>1) \end{cases}$$

with $x^+ \stackrel{\text{def}}{=} \max(x,0)$. Then, if the schedule is feasible up to S_n+P_n, it is feasible and periodic from S_n with a period of P. Where $P_k \stackrel{\text{def}}{=} \operatorname{lcm}(T_i \mid i=1,\ldots,k)$.

We can illustrate S_i as the first release of τ_i which occurs not before S_{i-1} (see Figure 3.6).

Proof. By induction. We first assume that, WLoG, the FTP scheduler corresponds to $\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$. Let $\mathbb{P}(i)$ be the proposition: if the task-set $\{\tau_1, \tau_2, \ldots, \tau_i\}$ is FTP-schedulable then the schedule is periodic from S_i with a period of P_i.

Base case: $\mathbb{P}(1)$ is true: if feasible the schedule for τ_1 is periodic of period $T_1 = S_1$ from the first release of τ_1 ($S_1 = O_1$) with $P_i = \text{lcm}\{T_j \mid j = 1, \ldots, i\}$.

Inductive step: For $i > 1, \mathbb{P}(i-1) \Rightarrow \mathbb{P}(i)$ is true. Assume $\mathbb{P}(i-1)$ is true for purpose of induction. We need to show that $\mathbb{P}(i)$ is true. $\mathbb{P}(i-1)$ means that if the task-set $\{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ is FTP-schedulable then the schedule of $\{\tau_1, \ldots, \tau_{i-1}\}$ is periodic from S_{i-1} with a period of P_{i-1}.

Now we consider the scheduling of $\{\tau_1, \ldots, \tau_i\}$, since τ_i is the lowest priority task the schedule and thus the periodicity of $\{\tau_1, \ldots, \tau_{i-1}\}$ is unchanged by the releases of task τ_i. S_i is the first release of task τ_i after (or at) S_{i-1} (i.e., $S_i \geqslant S_{i-1}$), combining the periodicity of the schedule of $\{\tau_1, \ldots, \tau_{i-1}\}$ and the periodicity of τ_i (from S_i with a period of T_i) we conclude that $\mathbb{P}(i)$ is true: if $\{\tau_1, \ldots, \tau_i\}$ is schedulable, its FTP schedule is periodic from S_i with a period of $\text{lcm}\{P_{i-1}, T_i\} = \text{lcm}\{T_j \mid j = 1, \ldots, i\} = P_i$. □

We have now the material to design a shorter feasibility interval:

Corollary 46

$[0, S_n + P)$ is a feasibility interval for asynchronous constrained deadline systems and FTP schedulers.

Indeed, in comparison with Theorem 44, the computation time is halved.

Consider the system:

System 47

τ_i	O_i	C_i	$D_i = T_i$
τ_1	0	7	10
τ_2	0	1	16
τ_3	4	3	15

We see that the utilisation of this System 47 is $U(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i} = 0.9625$

Consider a FTP assignment: $\tau_1 \succ \tau_2 \succ \tau_3$

The schedule of System 47 is given in Figure 3.7. If you compute S_n for System 47:

$$S_1 = 0$$
$$S_2 = 0 + 0$$
$$S_3 = 0 + 4 = 4$$

The feasibility interval is therefore $[0, 244)$ (since $P = 240$). As illustrated (see Figure 3.7) the periodicity starts at $t = 4$ with a period of 240, thus $[0, 244)$ is a feasibility interval.

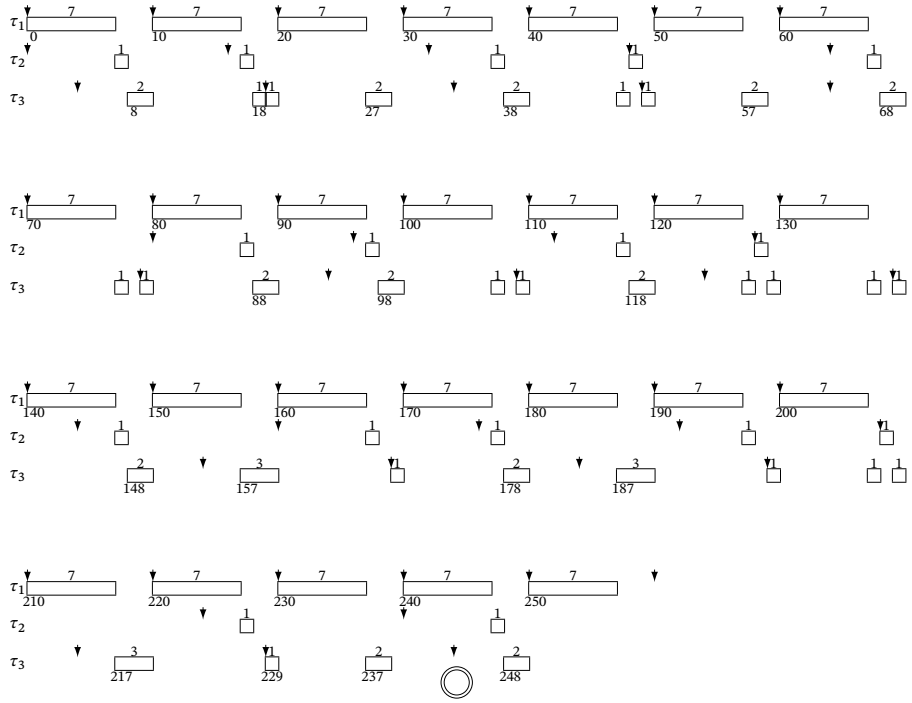


Figure 3.7: Periodic schedule of System 47

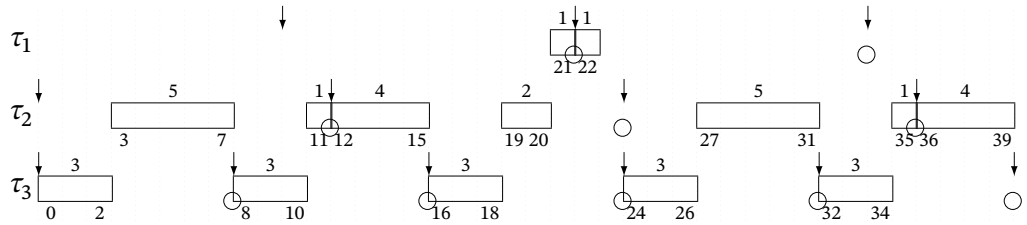


Figure 3.8: Feasible schedule of System 48 (the schedule repeats from time instant 24)

3.7 Optimality for constrained deadline asynchronous systems

The previous section provided a feasibility interval for asynchronous and constrained deadline systems. In this section we focus on schedulers for such systems for which neither RM nor DM are optimal. To illustrate this, let us define System 48.

System 48

τ_i	O_i	C_i	$D_i = T_i$
τ_1	10	1	12
τ_2	0	6	12
τ_3	0	3	8

This system is schedulable, with priorities $\tau_3 \succ \tau_2 \succ \tau_1$ (see Figure 3.8), while DM and RM ($\tau_3 \succ \tau_1 \succ \tau_2$) fail (see Figure 3.9).

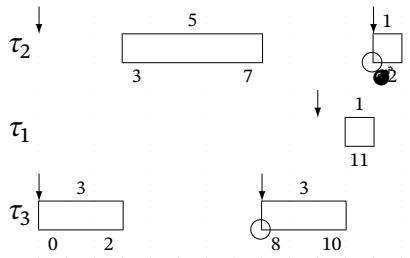


Figure 3.9: Unfeasible schedule of System 48 with RM/DM ($\tau_3 \succ \tau_1 \succ \tau_2$). A deadline is missed at time 12

3.7.1 AUDSLEY algorithm

A first naive but optimal priority assignment consists of considering *all* possible $n!$ FTP-priority assignments. AUDSLEY [3] proposed an efficient algorithm which considers, in the worst-case, $\mathcal{O}(n^2)$ FTP-priority assignments. The idea is based upon the following notion and property.

Definition 49 (*Lowest-priority viable*)

A task τ_i is said to be lowest-priority viable *iff all its jobs meet their deadline when:*

- *The task τ_i has the lowest priority.*
- *The others tasks have higher priority.*
- *When scheduling task τ_j ($j \neq i$), the scheduler considers deadlines as soft (i.e., it continues to schedule until completion).*

Theorem 50 (*Feasibility when τ_i is lowest-priority viable*)

Suppose that τ_i is lowest-priority viable. Then, there exists a feasible FTP-assignment for τ iff there exists an FTP-assignment for $\tau \setminus \{\tau_i\}$.

Proof. First we denote the task/priority assignment with $p(\tau_a) = b$ meaning that the task τ_a has a priority b.

Suppose that the following priority assignment p is feasible for τ:

$$p(\tau_1) = 1, \ldots, p(\tau_i) = i, p(\tau_{i+1}) = i+1, \ldots, p(\tau_n) = n$$

If τ_i is lowest-priority viable, then a second priority assignment p' where τ_i is given the lowest priority is feasible as well:

$$p'(\tau_1) = 1, \ldots, p'(\tau_{i-1}) = i-1, p'(\tau_{i+1}) = i, \ldots, p'(\tau_n) = n-1, p'(\tau_i) = n$$

The tasks assigned to levels $i+1, \ldots, n$ are promoted and remain schedulable because time slots they were occupying before their promotion remain available. Obviously, the tasks assigned to levels $1, \ldots, i-1$ remain also schedulable. □

3.7. AUDSLEY FTP ASSIGNMENT

AUDSLEY's algorithm

Theorem 50 suggests a procedure to assign task priorities:

- First, determine a lowest-priority viable task;
- Recursively apply the same technique to the subset $\tau \setminus \{\tau_i\}$ of cardinality $n-1$.

This leads to Algorithm 1.

Algorithm 1 AUDSLEY(τ)

```
if there is no lowest-priority viable task then
    return Infeasible
else
    τ_i ← lowest-priority viable task in τ
    p(τ_i) ← lowest-priority
    AUDSLEY(τ \ {τ_i})
end if
```

For AUDSLEY's algorithm, what is the maximal number priority assignments considered? (as a function of n the number of tasks)

- $\mathcal{O}(n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(n!)$

CHAPTER 4
Fixed Job Priority (FJP) ASSIGNMENT

The previous chapter introduced different ways to assign priorities at *task level*. This chapter focuses on allocating priorities at *job level* — fixed job priority (FJP) assignments — and in particular on Earliest Deadline First (EDF) and on the analysis of the resulting schedule.

4.1 Earliest Deadline First (EDF)

The principle of *EDF* scheduling (see Definition 12, page 18) is to prioritise jobs at runtime according to the *absolute* deadline of each job: the earlier the absolute deadline, the higher the job priority. A first important property of EDF is related to its optimality.

4.1.1 Optimality

Definition 51 (*Feasibility*)

A job set J is said to be feasible *if there exists a schedule which meets all the job deadlines of J.*

Note the difference between the previous definition and feasibility of FTP (see Definition 19, page 22.)

Theorem 52 (*EDF optimality*)

If a job set J is feasible, then EDF schedules that set.

Proof. In the following $\sigma^{(i)}$ ($i \in \mathbb{N}$) is a schedule of the set of jobs J and $\sigma^{(i)}(t)$ corresponds to the job executed at time instant t. $\sigma^{(i)}$ is defined *step by step*. Initially, $\sigma^{(0)}$ corresponds to a *feasible* schedule of the J which exists since J is feasible. Then, $\sigma^{(i)}$ is obtained by modifying $\sigma^{(i-1)}$ (see details in the inductive step). At the end, we obtain $\sigma^{(\infty)}$ which corresponds to EDF (by the domino effect).

By induction. Let $\mathbb{P}(i)$ be the proposition: the schedule $\sigma^{(i)}$ (defined above) is feasible and $\forall t \in \{0, 1, \ldots, i-1\}, \quad \sigma^{(i)}(t) = \text{EDF}(t)$. Intuitively, for all $t < i$, $\sigma^{(i)}(t)$ *mimics* EDF while $\sigma^{(i)}(t)$ can schedule the jobs in a different order for $t \geqslant i$.

Base case: $\mathbb{P}(0)$ is true: $\sigma^{(0)}$ being a feasible schedule for J which exists since J is feasible.

Inductive step: For $i \geqslant 0, \mathbb{P}(i) \Rightarrow \mathbb{P}(i+1)$ is true. Assume $\mathbb{P}(i)$ is true for the purpose of induction. We need to show that $\mathbb{P}(i+1)$ is true.

If $\sigma^{(i)}(i+1) = \text{EDF}(i+1)$ we define $\sigma^{(i+1)} = \sigma^{(i)}$. Consequently $\mathbb{P}(i+1)$ is true.

Otherwise, $\sigma^{(i)}(i+1) \neq \text{EDF}(i+1)$. During the time slot $i+1$, we have $j_1 = \sigma^{(i)}(i+1)$ and $j_2 = \text{EDF}(i+1)$ with $j_1 = (a_1, e_1, d_1)$ and $j_2 = (a_2, e_2, d_2)$. Since $\sigma^{(i)}$ meets all the deadlines, it successfully schedules j_2 in $[a_2, d_2($ and j_2 completes before d_2. This implies that $\sigma^{(i)}$ schedules j_2 for at least one slot before d_2 in $[i+1, d_2($.
By definition of EDF, $d_2 \leqslant d_1$; thus $\sigma^{(i)}$ schedules j_2 for at least one slot before d_1 as well. The schedule $\sigma^{(i+1)}$ is obtained by *swapping* the executions of j_1 and j_2 in $\sigma^{(i)}$. Note that j_1 after the swap is still executed before d_1.

By construction $\mathbb{P}(i+1)$ is true. □

You could also look at another proof of Theorem 52 for *non* discrete (dense) time space (see Appendix A for details).

The strength of Theorem 52 lies in the fact that it proves optimality of EDF *at job level*. This means that EDF is also optimal for asynchronous and arbitrary deadline systems (giving better results than FTP schedulers).

4.1.2 Schedulability tests

This section describes schedulability tests, beginning with simple cases and moving on to more complex ones.

Synchronous and implicit deadlines

We have seen in Lemma 14 (page 19) that $U(\tau) \leqslant 1$ is a *necessary* condition for feasibility. We will see that for EDF periodic implicit deadline tasks this condition is also *sufficient*.

Theorem 53 (*Necessay and sufficient condition of schedulability*)

> $U(\tau) \leqslant 1$ *is a necessary and sufficient condition for the* schedulability *of periodic implicit deadline tasks.*

Proof. Let τ be a task-set of n implicit deadline periodic tasks $\tau_1, \ldots, \tau_n$.

Consider a *"processor sharing"* schedule S obtained by partitioning the timeline into *infinitesimal* slots and by scheduling the task τ_i for a fraction of $U(\tau_i)$ in each slot.

Since $U(\tau) \leqslant 1$, such a schedule can be constructed.

For each job of task τ_i this schedule contains exactly $U(\tau_i) \times T_i = C_i$ CPU units between the job's arrival and its deadline. □

4.1. EARLIEST DEADLINE FIRST (EDF)

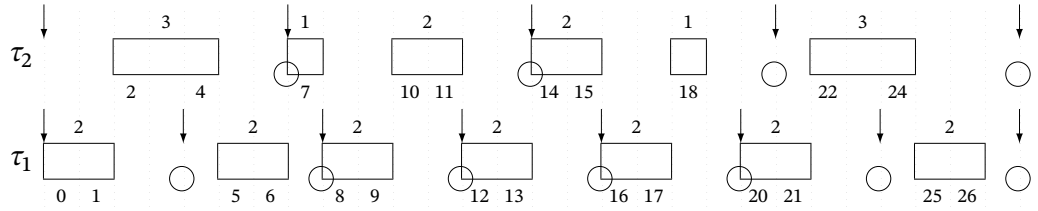


Figure 4.1: Execution of System 56 with EDF

Why is it not possible to implement the processor sharing schedule (referred to in the previous proof)?

Corollary 54

An implicit deadline periodic task-set is EDF-feasible iff $U(\tau) \leqslant 1$.

Proof. This is a direct consequence of EDF optimality (even with non discrete time space, see Appendix A) and Theorem 53. □

Synchronous and arbitrary deadlines

First, note that the synchronous periodic case remains the worst-case (as shown for FTP family in Lemma 25, page 23).

Theorem 55 (*EDF-feasibility*)

Let τ *be a synchronous periodic arbitrary deadline system. If* τ *is EDF-feasible, then all corresponding asynchronous periodic task-sets (the very same task, the same parameters except for the offsets) and all corresponding sporadic task-sets are also EDF-feasible.*

The property is stated for arbitrary deadline systems and therefore also holds for implicit and constrained ones by definition.

However, for EDF-scheduling, the response time of the *first* job in the synchronous case is *not* the greatest any more.

Let us take the following system:

System 56

τ_i	C_i	$D_i = T_i$
τ_1	2	4
τ_2	3	7

If it is scheduled with EDF, you obtain the schedule represented in Figure 4.1. Notice that the response time of the first job of τ_2 is 5, whilst the response time of the fourth job is 6. The key question here is what feasibility interval for EDF can be used?

Liu & Layland [18] have "shown" that we cannot miss a deadline *after* an idle unit. Once again, it is interesting to note that even if the results are correct, the proof itself is flawed (see [10] for details). It is therefore necessary to take a critical approach even with published, seminal, quoted documents, or even those by prestigious authors.

Theorem 57 (*[18]*)

When EDF schedules a synchronous periodic constrained deadline system, there are no idle units in the schedule before a deadline is missed.

Theorem 57 is extended with the notion of *idle point* (Definition 23, page 23).

Theorem 58 (*Idle point in EDF [11]*)

When EDF schedules a synchronous periodic arbitrary deadline systems, there are no idle points ((except for the origin) in the schedule before a deadline is missed.

Based on the previous property we can design a feasibility interval for synchronous periodic systems with arbitrary deadlines scheduled with EDF.

Corollary 59 (*The feasibility interval* $[0, L)$)

For EDF and synchronous periodic arbitrary deadline systems,

$$[0, L)$$

is a feasibility interval, where L is the first idle point.

Proof. This is a direct consequence of Theorem 57. □

L is the smallest positive solution to this equation:

$$L = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil C_i$$

We can compute this value in the same way as in Theorem 40 of FTP schedulers:

$$\begin{cases} w_0 & \stackrel{\text{def}}{=} \sum_{i=1}^{n} C_i & (initialisation) \\ w_{k+1} & = \sum_{i=1}^{n} \left\lceil \frac{w_k}{T_i} \right\rceil C_i & (iteration) \end{cases}$$

Note that L and λ_n are exactly the same thing; it's the context that differs. λ_n pertains to the FTP family, whereas L pertains to EDF.

Asynchronous and arbitrary deadlines

In this scenario, a feasibility interval is designed based on the schedule periodicity. But first the notion of system configuration at time t is formalised.

4.1. EARLIEST DEADLINE FIRST (EDF)

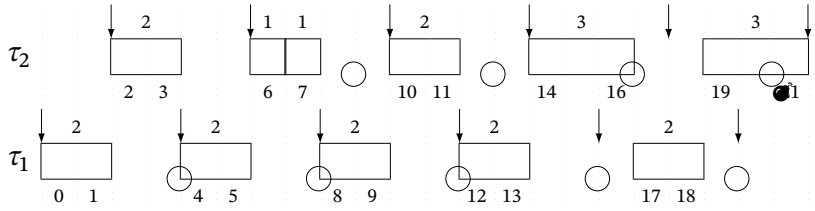


Figure 4.2: Execution of System 62 with EDF. τ_2 misses a deadline at $t = 21$

Definition 60 (*Configuration*)

Considering a periodic system R and a schedule S, we define the configuration *at instant t as follows:*

$$C_S(R,t) \stackrel{\text{def}}{=} ((\gamma_1(t),\alpha_1(t),\beta_1(t)),\ldots,(\gamma_n(t),\alpha_n(t),\beta_n(t)))$$

where:

- $\gamma_i(t) \stackrel{\text{def}}{=} (t - O_i) \mod T_i$ *is the time elapsed since the last release of τ_i, if $t \geqslant O_i$; $\gamma_i(t) \stackrel{\text{def}}{=} t - O_i$ otherwise.*
- $\alpha_i(t)$ *is the number of active jobs of τ_i;*
- $\beta_i(t)$ *is the cumulative CPU time used by the* oldest *active job of τ_i. If $\alpha_i(t) = 0$, we let $\beta_i(t) = 0$.*

Theorem 61

Let S be the EDF-schedule for the periodic asynchronous arbitrary deadline system τ. We say that τ is feasible iff:

- *Every deadline occurring in $[0, O_{\max} + 2P)$ is met*
- $C_S(R, O_{\max} + P) = C_S(R, O_{\max} + 2P)$

Both conditions of Definition 61 are necessary, as shown by the scheduling of System 62 in Figure 4.2following system:

System 62

τ_i	C_i	T_i	D_i	O_i
τ_1	2	4	4	0
τ_2	3	4	7	2

The hyper-period $P = 4 = \text{lcm}(T_1, T_2)$ and $O_{\max} = 2$. All of the deadlines in the interval $[0, 10)$ are met but $C_S(R, 6) \neq C_S(R, 10)$, since $\beta_2(6) = 2$ and $\beta_2(10) = 1$. In fact, τ_2 misses its deadline at time 21 as shown in Figure 4.2.

Formally, according to Definition 31, $[0, O_{\max} + 2P)$ is *not* a feasibility interval. Notice that the condition $U(\tau) \leqslant 1$ is not even satisfied by our system: $U(\tau) = \frac{5}{4} > 1$.

But if $U(\tau) \leqslant 1$ holds, then only the first condition of Theorem 61 must be satisfied in order to conclude feasibility.

Corollary 63 (*The feasibility interval* $[0, O_{\max} + 2P)$)

$[0, O_{\max} + 2P)$ *is a feasibility interval for periodic asynchronous arbitrary deadline systems where* $U(\tau) \leqslant 1$.

CHAPTER 5
Dynamic Priority (DP) ASSIGNMENT

We conclude this review of uniprocessor systems with the most general class of schedulers: dynamic priority (at job level). No restrictions are placed on the priorities that may be assigned to jobs. A job may have different priority levels during its lifetime. This is the case of the most popular DP scheduler: Least Laxity First (LLF). The scheduler LLF assigns priorities to jobs at runtime depending on their *laxity*.

Definition 64 (*Laxity*)

Let J be a job characterized by the tuple (a, e, d) *(see Definition 5). Let* $\epsilon_J(t)$ *represent, at time t, the cumulative CPU time used by J since its release. The* laxity *at instant t is defined as:*

$$\ell_J(t) \stackrel{\text{def}}{=} d - t - (e - \epsilon_J(t))$$

We can intuitively define the laxity of a job as its degree of urgency. If it is positive, the job can be delayed (by a maximum of $\ell_J(t)$ units). If it is null, the job should be immediately and continuously scheduled until its deadline in order to avoid missing a deadline. And of course, if it is negative, the deadline will definitely be missed.

This definition leads to the idea of LLF.

5.1 Least Laxity First (LLF)

Definition 65 (*Least Laxity First (LLF)*)

At each instant, LLF assigns the CPU to the job with the smallest laxity. Ties have to be solved in a deterministic manner.

5.1.1 Least Laxity First (LLF) Optimality

Theorem 66 (*LLF Optimality*)

If a job set J is feasible, then LLF schedules J.

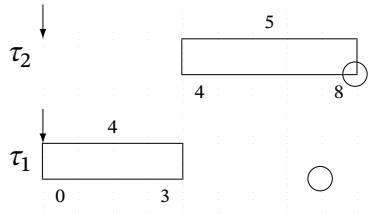


Figure 5.1: Execution of System 67 with EDF

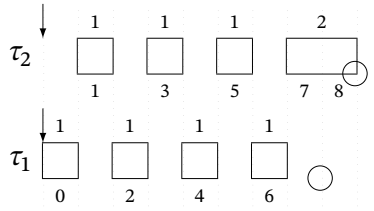


Figure 5.2: Execution of System 67 with LLF

5.1.2 LLF vs. EDF

Both priority assignment techniques are optimal. However, they differ in the number of preemptions required by the scheduler.

Let us consider the system:

System 67

τ_i	C_i	D_i	T_i
τ_1	*4*	*8*	*10*
τ_2	*5*	*9*	*10*

With an EDF schedule, there are no preemption (see Figure 5.1). But with LLF, there are 6 preemptions every 10 units (see Figure 5.2).

In fact, a *thrashing* situation can occur (i.e., where large amounts of computer resources are used to do a minimal amount of work). Suppose that at time t, two active jobs (J_1 and J_2) have the same and minimal laxity. Assuming that LLF schedules J_1 first, then at instant $t+1$, $\ell_{J_1}(t+1) = \ell_{J_1}(t)$ and $\ell_{J_2}(t+1) = \ell_{J_2}(t) - 1 < \ell_{J_1}(t+1)$. LLF must then preempt J_1 and schedule J_2. This scenario repeats itself.

To conclude, for the uniprocessor scheduling problem of periodic real-time tasks, there are not advantages to considering the DP scheduler class since they induce a lot of preemptions with no additional properties (optimality exists for a *simpler* class — FJP). On the other hand, the DP class of schedulers has real value for other types of problems than those studied in this curriculum guide. E.g. in situations where efficient resource utilisation is crucial and where time constraints are less stringent.

PART III

MULTIPROCESSOR SCHEDULING

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

Part III concerns more general hardware platforms, i.e., motherboards with multiple processors/cores which are able to execute tasks and jobs *simultaneously*.

CHAPTER 6
INTRODUCTION & MOTIVATION

6.1 Moore's law

In 1965, Gordon E. Moore (1929–2023) defined Moore's law based on his observation that the number of transistors per square inch on integrated circuits had doubled every year (see Figure 6.1) and predicted that this would continue in a similar way in the future. However, around 2002–2006, a limit was reached.

This limit could be explained by two factors:

- Physical limits: there is a maximum possible temperature, as well as a minimum (atomic) dimension.
- Financial considerations: manufacturing such chips was (and still is) expensive.

To face the need of more and more computing power, the electronic industry designed *parallel and multicore/multiprocessor architectures*, and these are the focus of the following chapters.

In the next section, we will examine the units which are strongly coupled, and which share a common time reference, and a common shared memory.

6.2 Scheduling problem

6.2.1 Limitations

A multiprocessor platform is built according to a parallel architecture but this parallelism is *limited*:

- At any given moment, each processor is idle or is executing *one* single job;
- At any given moment, a job is either: waiting to be executed; it is being executed on *one* single processor; or it has been completed, i.e. job parallelism is impossible.

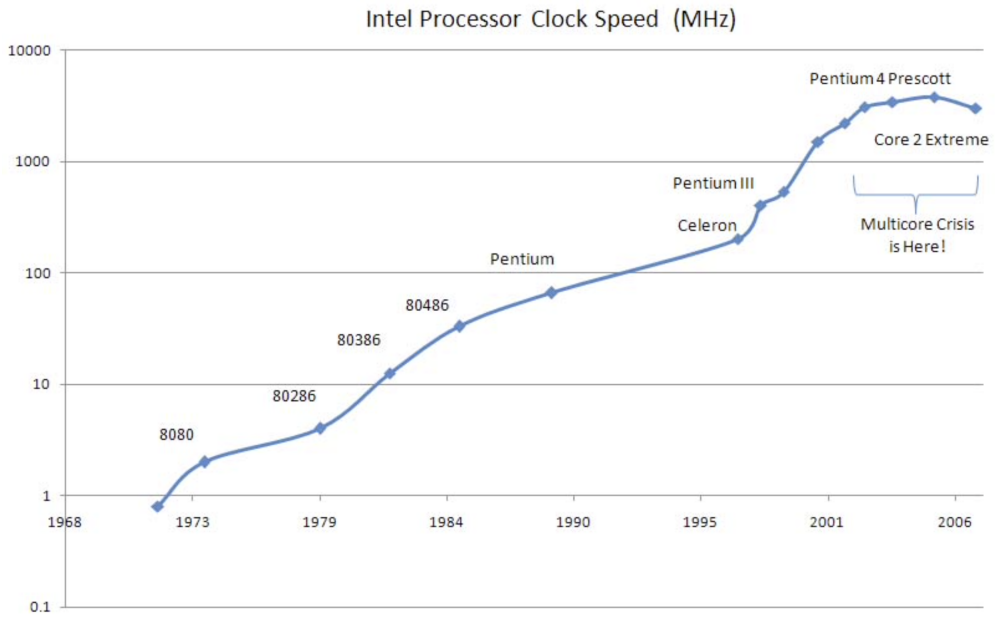


Figure 6.1: Moore's law, with a *logarithmic* scale for clock speed

6.2.2 Taxonomy of multiprocessor platforms

There are three kinds of platforms:

Definition 68 (*Identical parallel machines*)

Platforms on which all of the processors are identical in the sense that they have the same computing power.

Definition 69 (*Uniform parallel machines*)

In contrast, each processor in a uniform parallel machine is characterized by its own computing capacity. A job that is executed on a processor π_i with computing capacity s_i for t time units will be completed after $s_i \times t$ units of execution.

Definition 70 (*Unrelated parallel machines*)

In unrelated parallel machines, a rate is associated with each pair (i, j), where i is the index of a job and j is the index of a processor. A job J_i that is executed on a processor π_j for t time units will execute $s_{i,j} \times t$ units.

Note that Identical $\subset$ Uniform $\subset$ Unrelated.

The following section deals with the situation of *identical* multiprocessors, composed of m processors: $\pi_1, \pi_2, ..., \pi_m$.

6.2.3 Taxonomy of multiprocessor schedulers

We distinguish between two scheduling families.

Definition 71 (*Partitioned scheduling*)

Task partitioning involves statically dividing the set of tasks among the processor cores. Each task is assigned to a specific processor core, and the scheduling of tasks within each core follows a local *scheduling policy, such as earliest deadline first (EDF).*
Each processor core has its own local queue of tasks assigned to that particular core. The local queue represents the set of tasks that can be scheduled and executed on that core. The scheduling decisions are made independently *for each core, based on the local scheduling policy and the tasks in its local queue. The tasks in one core's queue are not visible to or considered by the other cores.*

Definition 72 (*Global scheduling*)

Global scheduling does not statically assign tasks to specific processor cores. Instead, the scheduling decisions are made dynamically and globally *based on the current state of the system, including the job priorities, and availability of processors.*
At the system level there is single shared queue *that contains all the jobs in the system, regardless of the processor core to which they are assigned. The system queue represents the set of jobs eligible for scheduling on any available processor core. The scheduling decisions are made globally for all jobs in the system, taking into account the priorities of jobs and the availability of processor cores.*

6.3 Illustrating partitioning

Consider the system:

System 73

τ_i	C_i	T_i	D_i
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

A feasible partition could be to statically assign τ_1 and τ_2 upon the processor π_1 (since $U(\tau_1)+U(\tau_2) = 1/4 + 3/5 \leqslant 1$) and τ_3 upon processor π_2.

6.4 Illustrating global scheduling

If we schedule System 73 using the *global* deadline monotonic, we obtain that $\tau_1 \succ \tau_2 \succ \tau_3$. The resulting schedule is shown in Figure 6.2. Notice that τ_3 *migrates* from π_1 to π_2 and back, for each job.

A job/task migration is the fact that a job/task is paused on a first processor and then resumed on a second (different) processor.

Look at Figure 6.2 in the interval $[0, 20)$. How many migrations occur?

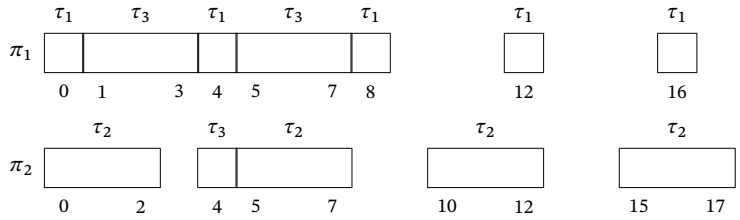


Figure 6.2: Execution of System 73 scheduled with Global DM on 2 processors

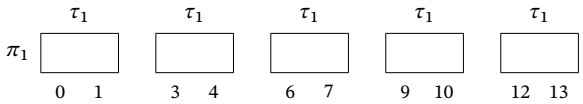


Figure 6.3: Example schedule

It is important to note one main difference between global and partitioned scheduling. In partitioned scheduling, it is not possible to migrate tasks. This means that once the tasks are assigned to a processor, the uniprocessor techniques seen in Part II are used. Global scheduling *is not* a form of partitioned scheduling.

6.4.1 Partitioned and global scheduling are incomparable

Lemma 74 (*[5]*)

There exist task-sets that are schedulable with global FJP algorithms but not with partitioned FJP ones.

Proof. Take an example system, scheduled on two processors:

System 75

τ_i	C_i	D_i	T_i
τ_1	2	2	3
τ_2	3	3	4
τ_3	5	12	12

Obviously all partitions fail: $U(\tau_i) + U(\tau_j) > 1$, $\forall i \neq j$.

We can schedule the tasks with a global FJP algorithm which assigns the lowest priority to τ_3's jobs. Notice that since $C_1 = D_1$ and $C_2 = D_2$ we *must* execute τ_1 and τ_2 immediately. Since there are two processors, τ_1 and τ_2 meet all deadlines. For τ_3, over any 12 contiguous time-slots, τ_1 may execute during at most 8 time-slots (see Figure 6.3). If τ_1 executes for fewer than 8 slots, τ_3 is schedulable. If τ_1 executes for exactly 8 time-slots, τ_1's jobs must be arriving 3 time-slots apart. Based upon τ_2's parameters, we know that it's impossible that τ_2's jobs execute in parallel with τ_1's jobs. Consequently, we have at least one extra time-slot for τ_3 on the remaining processor. □

Although it seems somewhat counter-intuitive, note that there are task systems which are scheduled using partitioned FJP algorithms that global FJP algorithms cannot schedule.

6.4. ILLUSTRATING GLOBAL SCHEDULING

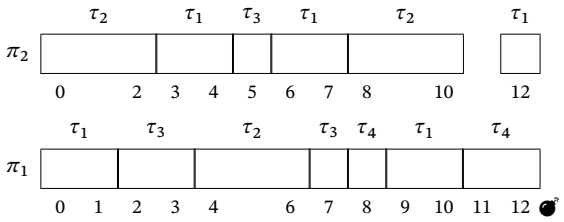


Figure 6.4: Global FJP (here $J_3 \succ J_4$) fails to schedule System 77. τ_4 misses a deadline at $t = 12$

Lemma 76 (*[5]*)

There are task systems that are schedulable using partitioned FJP algorithms but not with global FJP algorithms.

Proof. Take the example of System 77, scheduled on two processors π_1 and π_2:

System 77

τ_i	C_i	D_i	T_i
τ_1	2	2	3
τ_2	3	3	4
τ_3	4	12	12
τ_4	3	12	12

This system may be partitioned: τ_1 and τ_3 on π_1 and the remaining tasks on π_2, with each processor scheduled using EDF.

To show that no global FJP priority assignment can meet all deadlines, consider the synchronous arrival over the period $[0, 12)$. For feasibility's sake, we need to first serve all jobs of τ_1 and τ_2 ($C_i = D_i$). Jobs of τ_3 and τ_4 have a lower priority and end up missing their deadline (be it for the assignment $J_3 \succ J_4$ or the only other possible one ($J_4 \succ J_3$)) while one processor goes idle over the period $[11, 12)$ (see Figure 6.4 where we consider the case $J_3 \succ J_4$). □

Recap. Due to Lemma 74, there are partitioned algorithms that fail where global algorithms succeed. But due to Lemma 76, the opposite is also possible. Hence, global and partition scheduling are *incomparable*.

CHAPTER 7
PARTITIONING

7.1 Introduction

This chapter focuses on implicit deadline tasks.

The partitioning problem can be related to the *bin-packing* problem [20]. k objects have to fit in a finite number of bins of capacity C in a way that minimizes the number of bins used. In our case we can say that the objects correspond to the tasks, and that the bins are the cores. The bin size is the largest system utilisation possible for the scheduler: $\ln 2$ for RM (see Corollary 34, page 28), or 1 for EDF (see Theorem 53, page 40).

This problem is NP-complete. Many heuristics have been developed but they often lead to suboptimal solutions.

7.1.1 Heuristics

In order to partition the system in an efficient manner we use heuristics. The principle is to define *greedy algorithms*, i.e., algorithms that build a solution piece by piece. It is possible to pre-sort the tasks by increasing or decreasing system utilisation.

There are multiple ways to assign the current task to a processor:

- First-fit: assign it to the first eligible processor (i.e., where the task will *fit*, where there is enough spare capacity). See First-Fit with decreasing utilisation (FFD) Figure 7.1.
- Best-fit: assign it to an eligible processor with the maximum load ($U(\tau)$).
- Worst-fit: assign it to an eligible processor with the minimum load. See Worst-Fit-Decreasing-Utilisation (WFD) Figure 7.1.
- Next-fit: assign it to the *current* processor being considered, and if it cannot fit, it moves to the next available processor. It can never be assigned to the previous processors. See Next-Fit-Decreasing-Utilisation (NFD) Figure 7.1.

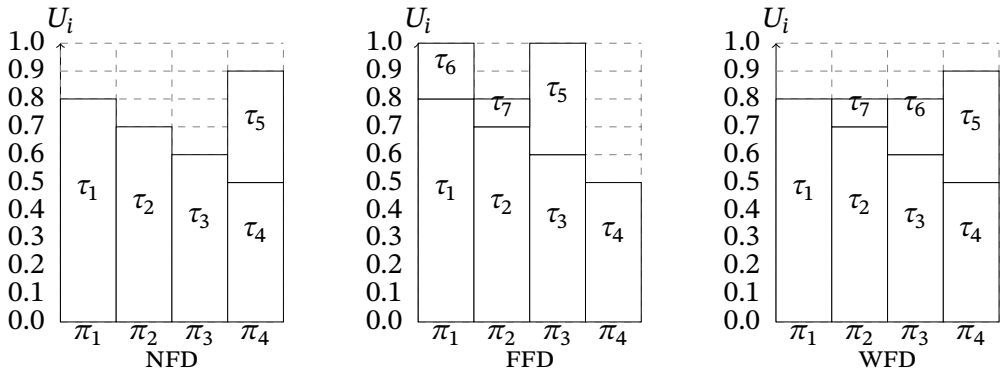


Figure 7.1: Bin-packing heuristics

7.2 Partitioned Rate Monotonic

Remember from Theorem 33, page 28 that in the uniprocessor case, an implicit deadline synchronous periodic system can be scheduled with RM when $U(\tau) \leqslant n(\sqrt[n]{2} - 1)$.
Similarly, in the multiprocessor case with partitioned Rate Monotonic, a sufficient schedulability test on processor π is:

$$U(\tau) \leqslant n_\pi(2^{\frac{1}{n_\pi}} - 1)$$

where n_π is the number of tasks on processor π.

Note that the bound is not tight enough to successfully partition the system. In the worst-case, we can lose about 50% of the platform capacity.

Consider $m + 1$ tasks with $T_i = 1$ and $C_i = \sqrt{2} - 1 + \epsilon$. Assume the platform size is m. There must be a processor π_ℓ which schedules two tasks (because we have $m + 1$ tasks for m processors). The total utilisation is $2(\sqrt{2} - 1 + \epsilon)$, which is (obviously) larger than $2(\sqrt{2} - 1)$. Consequently, no partition can be found with the chosen bound.

Here, there are two tasks, the limit is $U(\tau) \leqslant 2(\sqrt{2} - 1)$. But we know that these tasks have a WCET of $\sqrt{2} - 1 + \epsilon$. So the total WCET of both tasks is greater than the limit for schedulability with RM.

If we consider the limit case ($\epsilon \to 0$ and $m \to \infty$), we cannot guarantee schedulability above a utilization of $\sqrt{2} - 1$, i.e. about 41%.

7.3 Partitioned EDF: First Fit Decreasing Utilisation

For EDF, the bound is optimal since we know that $U(\tau) \leqslant 1$ is a necessary and sufficient condition for uniprocessor schedulability. The idea of First Fit Decreasing Utilisation (FFDU) is to assign tasks (ordered by decreasing utilisation factor) by the use of first-fit.

Theorem 78 provides a *sufficient* schedulability test for First Fit Decreasing Utilisation:

7.3. FFDU

Theorem 78 (*Sufficient EDF-FFDU schedulability test[6]*)

A sporadic implicit deadline task-set τ is schedulable using FFDU if:

$$U(\tau) \leqslant \frac{m+1}{2}$$

and

$$U_{\max} \leqslant 1$$

where $U_{\max} \stackrel{\text{def}}{=} \max\{U(\tau_i) \mid i = 1, \ldots, n\}$.

The proof of this property is not the subject of this curriculum guide.

CHAPTER 8
GLOBAL SCHEDULING

This chapter is about *global scheduling* where migrations are allowed. It first examines negative results, followed by positive results.

8.1 Negative results

If we don't know the characteristics of the jobs in advance, then no optimal decision exists.

8.1.1 No online optimal scheduler exists

Definition 79 (*Online scheduler*)

Online schedulers take their scheduling decisions at runtime, based on the characteristics of the jobs already released and without any knowledge of when future jobs will be released.

A first negative result is that no online optimal scheduler exists.

Theorem 80 (*[14]*)

For any $m > 1$, no online and optimal scheduling algorithm exists.

Proof. To show the property we exhibit an (counter-)example where $m = 2$ and the scheduling of real-time jobs. Also, we consider *non*-discrete (dense or contiguous) time space.

At instant 0 three jobs J_1, J_2, J_3 are released with $d_1 = d_2 = 4$, $d_3 = 8$, $e_1 = e_2 = 2$ and $e_3 = 4$. The algorithm schedules the jobs on two processors starting at instant 0. There is an infinite number of different online schedules; here we examine 2 classes of schedules: *(i)* the job J_3 executes for a strictly positive amount of time in the interval $[0, 2)$ xor *(ii)* the job J_3 does not execute at all.

- J_3 executes during the interval $[0, 2)$. In this case, at least one of the other jobs will not complete at instant 2. Suppose that this is the case of J_2. If two new jobs J_4, J_5

are released at instant 2, with $d_4 = d_5 = 4$ and $e_4 = e_5 = 2$, the system fails to schedule despite the fact that the system is feasible (we need to start the execution of J_3 at instant 4).

- J_3 does not execute during the interval $[0, 2)$. In this case, at instant 4, the remaining J_3 requires at least two units of time for execution. Now, consider that two other new jobs J'_4, J'_5 are released at instant 4 with $d'_4 = d'_5 = 8$ and $e'_4 = e'_5 = 4$. Once again, the online schedule fails to schedule a feasible system (we need to schedule J_3 in $[0, 2)$).

□

To summarise we need *clairvoyance*[1] in order to take an optimal decision.

8.1.2 Scheduling anomalies

This section demonstrates a counter-intuitive phenomenon specific to multiprocessors: the scheduling anomalies.

Definition 81 (*Scheduling anomaly*)

A scheduling anomaly occurs when a change of the system parameters that intuitively seems positive in fact jeopardises the system's schedulability.

Definition 82 (*Intuitively positive change*)

We say that a change is intuitively positive if it reduces the utilisation factor of tasks.

The period might be increased, the WCET might be decreased, the processor speed might be increased.

For example:

System 83

τ_i	C_i	D_i	T_i
τ_1	1	2	4
τ_2	3	3	5
τ_3	7	8	10

If you schedule it with global DM on two processors, it gives $\tau_1 \succ \tau_2 \succ \tau_3$. The schedule is given by Figure 8.1.

But if we increase the period of τ_1 from 4 to 5, the scheduled execution leads to a missed deadline for τ_3 at instant 8 (see Figure 8.2).

Scheduling anomalies exist because parallelism in the execution of jobs is *not* allowed. Although this does not seem logical at the outset, in fact it is perfectly normal!

[1] Clairvoyance is a term often used in the context of online scheduling to refer to the ability of a scheduling system to accurately predict future events or demands.

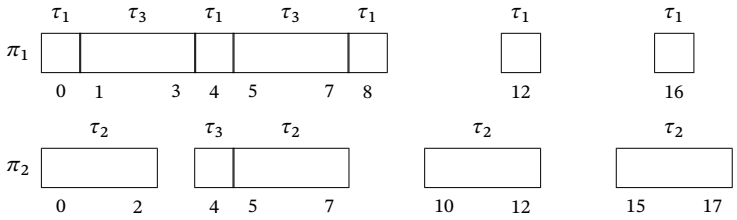


Figure 8.1: Execution of System 83

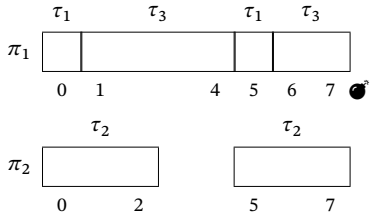


Figure 8.2: Execution System 83 with $T_1 = 5$

Do you think that this type of anomaly can occur in a *uni*processor context?

Consequence of the scheduling anomaly on the periods. If you want to check the schedulability of *sporadic tasks*, it is not enough to check the synchronous periodic subcase. In fact, this remains an open question: in the context of multiprocessor scheduling we have no idea of when the critical instant is.

8.2 Positive results

The next section addresses global scheduling for multiprocessors with *positive* results (schedulers and schedulability tests).

8.2.1 Periodic implicit-deadline Systems

Theorem 84

Any periodic implicit-deadline system is feasible iff

$$U(\tau) \leqslant m$$

and

$$U_{\max}(\tau) \leqslant 1$$

Proof. The argument used for feasibility, under those conditions, is to partition the timeline into infinitesimal slots and by scheduling a task τ_i for a fraction proportional to $U(\tau_i)$ in each

slot. □

Why is the condition $U_{\max}(\tau) \leqslant 1$ required for $m > 1$ processors?

PFAIR

This section presents an optimal scheduler (PFAIR), for multiprocessors in the particular case of periodic and implicit-deadline task-sets.

It is important to first understand the notion of an *ideal fair* schedule for non discrete (dense) time space.

Definition 85 (*Ideal fairness*)

Ideal fairness *requires us to schedule each task at a* constant rate*. For periodic tasks, each task τ_i progresses* exactly *at a rate of $U(\tau_i)$.*

In an ideal fair schedule, each task receives the processor for an *exact* duration of $U(\tau_i) \times t$ on the interval $[0, t]$. This implies that all the deadlines are met, but is *impossible* to implement in practice: such implementation might require a large number of preemptions, moreover we need preemption *within* the time slots, which is not possible in a discrete-time situation.

In the context of *discrete* time space we will consider PFAIR; the time is discrete since the timeline is divided into quanta (or time slices). The scheduler is called once every quantum, to choose the jobs to run (at most m jobs). PFAIR scheduling is a *"discrete"* and practical implementation of ideal fair scheduling.

A schedule can be formalised as a function:

Definition 86 (*Schedule* $S(\tau_i, \ell)$)

$$S : \tau \times \mathbb{N} \mapsto \{0, 1\}$$

where:

- τ *is a periodic task-set*
- $S(\tau_i, t) = 1$ *means that task τ_i is scheduled in* $[t, t + 1)$
- $S(\tau_i, t) = 0$ *means the opposite*

The actual difference between the ideal fair solution and the PFAIR solution is the lag:

Definition 87 (*Lag*)

The lag is defined as the function:

$$\text{lag}(\tau_i, t) \stackrel{\text{def}}{=} U(\tau_i) \cdot t - \sum_{\ell=0}^{t-1} S(\tau_i, \ell)$$

The first part of the difference is the ideal fair schedule, and the second part is the sum of the time units scheduled until time t.

At any time, we require PFAIR to have an absolute lag lower than 1. This means that at any time τ_i has to receive either $\lfloor U(\tau_i) \cdot t \rfloor$ or $\lceil U(\tau_i) \cdot t \rceil$.

Formally speaking,

Definition 88 (*PFAIR Schedule*)

A schedule is said to be PFAIR iff:

$$-1 < \text{lag}(\tau_i, t) < 1 \; \forall \tau_i \in \tau, \forall t \in \mathbb{N}$$

This leads to a second interesting result:

Theorem 89

PFAIRness implies that all the deadlines are met.

Proof. Each periodic task τ_i must receive C_i processor units in each interval $[\ell \cdot T_i, (\ell+1) \cdot T_i)$, with $\ell \in \mathbb{N}$. I.e., each job is on time.

At time $t = \ell \cdot T_i$, we have $U(\tau_i) \cdot t = \frac{C_i}{T_i} \cdot \ell \cdot T_i = \ell \cdot C_i$, a natural number. From Definition 88, we know that at time $t = \ell \cdot T_i$, the allocation of PFAIR corresponds to the ideal fair schedule. Since all the deadlines are met in the ideal fair schedule, it is also the case in PFAIR schedule. □

And we have a second interesting result:

Theorem 90 (*PFAIR Optimality*)

Let τ be a periodic synchronous implicit-deadline system. A PFAIR schedule exists for τ on m processors iff:

$$U(\tau) \leqslant m$$

and

$$U_{\max} \leqslant 1$$

This theorem is great because it shows that PFAIR schedules *all* schedulable systems. Thus, *PFAIR is optimal*. Note that this does not contradict Theorem 80 since we only consider *periodic* tasks, where we know all of the release times of the future jobs.

Algorithms to implement PFAIR can be found in research papers PF [7], PD [8] and PD^2 [2].

8.2.2 Global EDF

A natural generalisation of EDF upon multiprocessors is *global* EDF for which a *sufficient* schedulability condition is known.

Theorem 91 (*[22]*)

Any sporadic implicit-deadline system is schedulable using global EDF on m processors if

$$U(\tau) \leqslant m - (m-1) \cdot U_{\max}$$

Note that this test is very pessimistic[2] if $U_{\max} \approx 1$. In this case, the test is almost equivalent to the $U(\tau) \leqslant 1$ condition, which means using only one processor with EDF. However, it is possible to use a scheduler with a better schedulability test: EDF$^{(k)}$.

8.2.3 EDF$^{(k)}$ scheduling

For simplicity, we assume that $U(\tau_1) \geqslant U(\tau_2) \geqslant \cdots \geqslant U(\tau_n)$. $\tau^{(i)}$ is used to denote the set of the $(n-i+1)$ tasks with the lowest utilisation factor of τ: $\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_i, \tau_{i+1}, ..., \tau_n\}$.

We can derive from Theorem 91 that:

$$m \geqslant \left\lceil \frac{U(\tau) - U(\tau_1)}{1 - U(\tau_1)} \right\rceil$$

The goal now is to define a way to adapt EDF so it requires a smaller number of processors than $\left\lceil \frac{U(\tau) - U(\tau_1)}{1 - U(\tau_1)} \right\rceil$ (if any). In the following, the parameter k is an integer such that $k \in [1, \dots, n]$.

Definition 92 (*EDF$^{(k)}$*)

- *For all $i < k$, the jobs of τ_i receive* maximum *priority, which can be done by* modifying *their absolute deadlines to $-\infty$.*
- *Otherwise ($i \geqslant k$), the jobs of τ_i are assigned priorities according to EDF (the unmodified absolute deadline).*

Note that even if the RTOS is only based on the pure global EDF; by modifying the absolute deadlines it is easy to implement EDF$^{(k)}$ without modifying the scheduler, i.e., the real-time kernel.

A simpler way to explain EDF$^{(k)}$ is to say that it gives the highest priority to the jobs of the $k-1$ first tasks of τ (to the $k-1$ most utilising tasks), and that it schedules the others with global EDF.

It is in fact a generalisation of EDF, where global EDF corresponds to EDF$^{(1)}$.

The EDF$^{(k)}$ schedulability test (for k determined) is:

Theorem 93 (*[13]*)

A sporadic implicit-deadline system is schedulable on m processors using EDF$^{(k)}$ if

$$m \geqslant (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1 - U(\tau_k)} \right\rceil$$

Put more simply: EDF$^{(k)}$ requires $k-1$ processor(s) for the $k-1$ first task(s) (sorted), and

[2]The test is based on a *sufficient* condition, which is inherently pessimistic. However, in many cases, the test is not satisfied even though the system is schedulable.

8.2. POSITIVE RESULTS

uses global EDF on the $\left\lceil \frac{U(\tau^{(k+1)})}{1-U(\tau_k)} \right\rceil$ other processor(s).

We have to choose k in order to minimise the required number of processors, more formally:

Corollary 94 (*[13]*)

A sporadic implicit-deadline system τ is schedulable on $m_{\min}$ processors using EDF$^{(m_{\min})}$ with

$$m_{\min}(\tau) = \min_{k=1}^{n} \left\{ (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1-U(\tau_k)} \right\rceil \right\}$$

For example:

System 95

τ_i	C_i	T_i	$U(\tau_i)$	$U(\tau^{(i)})$
τ_1	9	10	0.9	2.456
τ_2	14	19	0.737	1.556
τ_3	1	3	0.333	0.819
τ_4	2	7	0.286	0.486
τ_5	1	5	0.2	0.2

In Corollary 94, we want to identify the value of k ($k_{\min}$) for which the equation below is minimized

$$m_{\min}(\tau) = (k_{\min}(\tau) - 1) + \left\lceil \frac{U(\tau^{(k_{\min}(\tau)+1)})}{1-U(\tau_{k_{\min}})} \right\rceil$$

- $k = 1 \rightarrow m = 16$
- $k = 2 \rightarrow m = 5$
- $\mathbf{k = 3 \rightarrow m = 3}$
- $k = 4 \rightarrow m = 4$
- $k = 5 \rightarrow m = 5$

We see that the value of m is minimal for $k = 3$, so the minimum number of processors required is 3. We see also the improvement in comparison with pure EDF ($k = 1$).

PART IV

PARALLEL ALGORITHMS

« Writing parallel programs is like trying to untangle spaghetti.
It's a mess, but when you get it right, it's fast. »
— Leslie LAMPORT

CHAPTER 9
INTRODUCTION TO PARALLEL ALGORITHMS

Motivation. Nowadays commercial off-the-shelf (COTS) on-board computers (OBCs) are composed of a parallel *hardware* architectures (multiprocessor or multicore). To exploit this parallelism it is essential to design *software* that is itself parallel. This is important for *efficiency* but perhaps more importantly nowadays with regard to *energy* consumption. In the research community, there is evidence that multi-core machines are said to provide high performance at *low-energy* cost via parallelism (e.g., multi-threading).

Objective of the chapter. The aim of this chapter is to show how parallelism reduces the time complexity of algorithms (and therefore the power consumption). To do so, we will revisit a well-known algorithmic problem: sorting an array. Unfortunately, we will also report a negative result: the limits of parallelism which is characterised by AMDAHL's law.

9.1 The speed-up factor

Let's introduce a new notion: the speed-up factor:

Definition 96 (*Speed-up factor*)

The speed-up factor $S(n)$ is a measure of the relative performance difference between the sequential and parallel versions of a program. It is defined as:

$$S(n) \stackrel{\text{def}}{=} \frac{t_s}{t_p(n)}$$

where

- *t_s denotes the execution time on a uniprocessor system, and*
- *$t_p(n)$ denotes the execution time on a multiprocessor system using n cpus.*

When we compare sequential and parallel algorithms, we usually consider the fastest known

version of the sequential program: we want to already have an ideal scheme on the uniprocessor system before trying to speed it up on a multiprocessor system.

In principle, the maximum speed-up factor is n: each parallel algorithm which is solving a problem in time $t_p(n)$ with n processors can in principle be simulated by a sequential algorithm in $t_s = n \cdot t_p(n)$ time on a single processor. (However, simulation may require some execution overhead.)

9.2 Cost of parallelisation

The parallelisation of a program can produce several (additional) costs:

- times where not *all* processors are busy. This includes the times where only one processor is busy due to the sequential nature of a step in the program.
- Additional computations in the parallel version (from merging operations for instance).
- Delays in communication and synchronisation (rendez-vous and critical section).

We also have to take into account that the design of a parallel algorithm is generally more difficult, even far from trivial.

9.3 Maximum speed-up factor: AMDAHL's law

AMDAHL's law, defined by Gene AMDAHL in 1967, is a law defining the maximum speed-up factor possible with parallelisation.

The execution of the parallel program can be modelled by distinguishing two phases: (i) *sequential phase* which corresponds to time periods where only a *single* cpu is working (the others are assumed to be blocked), and (ii) *parallel phase* which corresponds to time periods where *all* the cpus are working. Let f be the fraction of sequential computations of a parallel program. $1-f$ is thus the fraction of parallelisable instructions. If t_s is the total computation time required by the sequential program then the total computation time required by the parallel program is:

$$f \cdot t_s + (1-f)\frac{t_s}{n}.$$

Therefore,

Definition 97 (*AMDAHL's law*)

The maximum speed-up factor by parallelisation is

$$S(n) = \frac{t_s}{f \cdot t_s + (1-f) \cdot \frac{t_s}{n}} = \frac{n}{1 + (n-1) \cdot f}$$

This law has a negative consequence: there is an upper limit for the speed-up factor:

$$\lim_{n\to\infty} S(n) = \frac{1}{f}$$

For example, if only 5% of the computations are sequential, the maximum speed-up factor we can hope to achieve is 20, regardless of the number of cores.

Amdahl's Law states that the maximum speedup that can be achieved by a parallel algorithm is limited by the fraction of the program that cannot be parallelized.
In other words, if a program has a portion that cannot be parallelized, then increasing the number of processors will not improve the performance of that portion of the program. Therefore, it is important for programmers to identify the portions of the program that cannot be parallelized and optimize them as much as possible to ensure maximum performance.

9.4 Parallel sorting algorithms

Ordering a list of items (an array) is a very common task in computer science and is solved by a sorting algorithm. The section below presents two parallel solutions.

9.4.1 Rank sort

For each item of the array, the Rank Sort (Algorithm 2) is computing the number of elements that are smaller than the (current) item. This is the *rank* of the item. The rank determination requires to compare the item with *all* other values of the array. Finally, the algorithm uses the rank of each item to place it in its correct (final) sorted position.

Algorithm 2 The rank sort algorithm

```
for (int i=0; i<n; i++) {
    int x = 0;
    for (int j=0; j<n; j++) {
        if(a[i]>a[j]) {
            x++;
        }
    }
    b[x] = a[i];
}
```

This algorithm has a time complexity of $\mathcal{O}(n^2)$, which is bad for a sequential sorting algorithm (in comparison with Heapsort or Quicksort for instance).

However, rank sort is ideal for the purpose of parallelisation. In particular the different ranks can be computed *independently*. Therefore, the main loop can be parallelised see Algorithm 3. This time the complexity is linear $\mathcal{O}(n)$ (if we dispose of n cores) which is better than any sequential sorting algorithm.

Algorithm 3 The parallel rank sort algorithm

```
for (int i=0; i<n; i++) {//in parallel
    int x = 0;
    for (int j=0; j<n; j++) {
        if(a[i]>a[j]) {
            x++;
        }
    }
    b[x] = a[i];
}
```

The next section shows that by taking advantage of parallelism you can do even better than linear complexity to sort an array. This is the bitonic sort, which is designed specifically for parallel architectures.

9.4.2 Bitonic sort

Before describing the algorithm we need to define a bitonic sequence:

Definition 98 (*Bitonic sequence*)

A sequence $(a_1, a_2, ..., a_{2\cdot k})$ *is said to be* bitonic *iff there is an integer* j, $1 \leqslant j \leqslant 2 \cdot k$, *such that*

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_j \geqslant a_{j+1} \geqslant a_{j+2} \geqslant \cdots \geqslant a_{2\cdot k}$$

or the sequence does not initially satisfy the previous condition, but can be cyclically shifted until the condition is satisfied.

As an example, 3 5 8 9 7 4 2 1 is bitonic.

The bitonic sort will make use of a basic statement:

Definition 99 (*Compare and swap*)

The compare and swap statement receives a pair of elements as input and produces an ordered pair (in either increasing or decreasing order) as output in one time unit.

In the following, we will assume that the length of the sequence to be sorted is a power of 2.

Lemma 100 (*[1, 21]*)

Let $(a_1, a_2, \ldots, a_{2\cdot k})$ *be a bitonic sequence, let* $d_i \stackrel{\text{def}}{=} \min(a_i, a_{k+i})$, *and* $e_i \stackrel{\text{def}}{=} \max(a_i, a_{k+i})$, *for* $i = 1, 2, \ldots, k$.

- *The sequences* $(d_1, d_2, ..., d_k)$ *and* $(e_1, e_2, ..., e_k)$ *are both bitonic,*
- $\max(d_1, d_2, ..., d_k) \leqslant \min(e_1, e_2, ..., e_k)$

9.4. PARALLEL SORTING ALGORITHMS

Sorting a bitonic sequence

Lemma 100 suggests a method to efficiently sort a bitonic sequence by means of parallelisation.

A bitonic sequence $(a_1, a_2, ..., a_{2 \cdot k})$ can be sorted into a sequence $(c_1, c_2, ..., c_{2 \cdot k})$ in non decreasing order by the following algorithm $\text{Merge}_{2 \cdot k}$:

- Step 1. The two sequences $(d_1, d_2, ..., d_k)$ and $(e_1, e_2, ..., e_k)$ are produced.
- Step 2. These two bitonic sequences are sorted *independently* and *recursively*, each by a call to Merge_k.
- Step 3. Sorted sequences $(d'_1, ..., d'_k)$ and $(e'_1, ..., e'_k)$ are concatenated and returned.

Algorithm 4 is a possible implementation of this *principle*. The sorting is done directly in the array `a[]` (the array `e[]` and `d[]` and the concatenation are actually not needed).

Assuming that there are many cores, what is the complexity of Step 1 (the for-statement of the `bitonicMerge` function)?

Algorithm 4 Merge $2k$ C-like code

```
const int ASCENDING  = 1;
const int DESCENDING = 0;

void bitonicMerge(int lo, int cnt, int dir) {
  if (cnt>1) {
    int k=cnt/2;
    int i;
    for (i=lo; i<lo+k; i++) //in parallel
      compare-and-swap(i, i+k, dir);
    // then 2 parallel and recursive calls
    bitonicMerge(lo, k, dir);
    bitonicMerge(lo+k, k, dir);
  }
}
void compare-and-swap(int i, int j, int dir) {
  if (dir==(a[i]>a[j]))
    exchange(i,j);
}
inline void exchange(int i, int j) {
  int t;
  t = a[i];
  a[i] = a[j];
  a[j] = t;
}
```

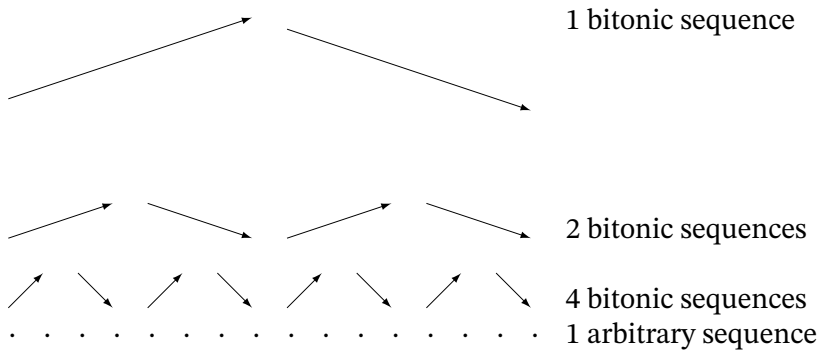


Figure 9.1: bitonic-arbitrary

Complexity. The time complexity of the bitonic sort is $\mathcal{O}((log(n)^2)$. The maximum number of *compare and swap* needed at the same time is
$\mathcal{O}(n(log(n))^2)$ (see [1, 21] for details).

Sorting an arbitrary sequence

The bitonic sort is not only useful for sorting bitonic sequences. Its value lies in the fact that the pre-processing step of constructing a bitonic sequence from an arbitrary sequence has a comparable (or even lower) time complexity ($O(\log(n)^2)$).

The idea: use Merge_{2k} as pre-processing to build a bitonic sequence as illustrated by Figure 9.1.

More formally, given an *arbitrary* sequence $(a_1, a_2, \ldots, a_n)$, consider the $n/2$ compare-and-swaps for the pairs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$.

For odd-numbered compare-and-swaps, place the smallest value first. For even-numbered compare-and-swaps, place the largest value first. At the end of the first stage, $n/4$ bitonic sequences of length 4 are obtained. These sequences can be sorted using Merge_4.

Odd-numbered instances get sorted by nondecreasing order, whereas even-numbered instances get sorted by nonincreasing order. This yields $n/8$ bitonic sequences, each of length 8.

The process repeats until a *single* bitonic sequence of length n is obtained. The time complexity of the pre-processing is identical to the one of the *bitonic* sort itself (see [1] for details).

Bitonic sequences are commonly used in computer science for designing *efficient* parallel sorting algorithms. The best known of these algorithms is the bitonic sort algorithm which sorts a bitonic sequence in $\mathcal{O}((\log(n))^2)$ time using $\mathcal{O}(n(\log(n))^2)$ processors.

PART V

CONCURRENCY

« "Sometimes" is sometimes "not never". »
— Leslie LAMPORT

CHAPTER 10
INTRODUCTION

This section is based on the excellent book by Mordechai (Moti) BEN-ARI, "Principles of Concurrent and Distributed Programming", second edition, Pearson Education, 2006 [9]. Nowadays programs are mostly concurrent or distributed and modern programming languages support concurrent and/or distributed paradigms.

An ordinary program is said to be *sequential*, i.e., after compilation a *sequence* of statements are executed in a certain (defined) order.

10.1 Concurrent execution

From now on, we assume the existence of *concurrent* programs, which are sets of sequential programs, which may be executed in parallel. Very often, the term "parallel" is associated with hardware architecture, while the term "concurrency" is associated with software. Concurrency denotes the cases where there *might* be parallelism. This concept is an abstraction, a kind of generalisation of the idea of parallelism.

Definition 101 (*Process*)

A process is a sequence of atomic statements (see Definition 105, page 79 for a formal definition of atomicity).

Definition 102 (*Concurrent program*)

A concurrent program is a set of processes that might be executed in parallel. The execution of a concurrent program corresponds to a sequence of atomic instructions. Given the potential for parallelism, this sequence is not unique. Consider the case of two concurrent instructions, A and B. There are two possible sequences: AB or BA. Therefore, we must consider all possible combinations in the execution sequence ("interleaving" in the following, see Definition 104, page 79).

The execution of a concurrent program produces a sequence of atomic statements, among all the possible permutations referred to as interleaving.

We start our study by introducing the formalism used through this part.

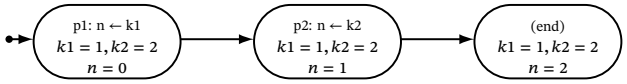


Figure 10.1: State diagram of Algorithm 5

Threads p & q	
integer n ← 0	
p	q
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

Figure 10.2: Trivial concurrent program

10.2 State diagrams

State diagrams are an effective tool to show the execution of concurrent programs. Consider the case of a trivial sequential program in Algorithm 5.

Algorithm 5 A trivial sequential program

$n \leftarrow 0$
$k_1 \leftarrow 1$
$k_2 \leftarrow 2$
$p_1 : n \leftarrow k_1$
$p_2 : n \leftarrow k_2$

The state diagram is illustrated Figure 10.1. Let's take a look at a first *concurrent* program Figure 10.2. Because the program of Figure 10.2 can be parallelised in two threads/processes, we got the state diagram of Figure 10.3.

10.3 Scenario

Definition 103 (*Scenario*)

A scenario is a sequence of states.

A scenario corresponds to a particular execution, which informally consists of taking one path on the state diagram of a concurrent program. In the following, the **bold text** shows the next state to be executed. An example is given in Table 10.1.

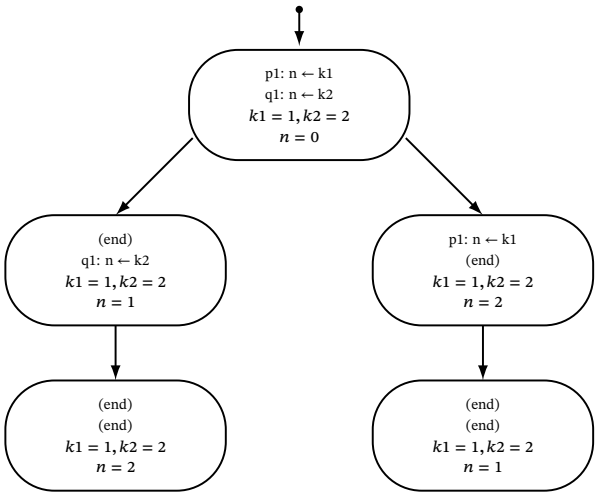


Figure 10.3: State diagram of algorithm described by Figure 10.2

Process p	Process q	n	k_1	k_2
$\mathbf{p_1 : n \leftarrow k_1}$	$q_1 : n \leftarrow k_2$	0	1	2
(end)	$\mathbf{q_1 : n \leftarrow k_2}$	1	1	2
(end)	(end)	2	1	2

Table 10.1: A scenario

10.4 Atomic statements

Definition 104 (*Interleaving*)

Interleaving *refers to the technique of executing multiple instructions from different programs in an overlapping manner. It allows multiple tasks to make progress concurrently by sharing the available computing resources, such as the CPU (Central Processing Unit), in an efficient manner.*
When multiple programs are running concurrently, the CPU divides its time and resources among them. Interleaving enables the execution of instructions from different programs in an alternating *or overlapping fashion, rather than executing all instructions from one program or thread before moving on to the next.*

Definition 105 (*Atomic statement*)

An atomic *statement is a non-divisible operation that executes without interruption.*

As a result, two (or more) atomic statements are executed "simultaneously" and therefore the resulting execution will be the same as in a sequential execution (regardless to the order). In the following, labelled statement (ex: $p_1 : n \leftarrow 0$) are assumed to be atomic.

Here, assignment statements and the evaluation of boolean conditions are considered atomic. This is unrealistic in practice, but it gives us a simple model to examine more complex cases where instruction interleaving occurs at machine code (i.e., assembly language) level.

Whilst the algorithms in this section utilise high-level pseudo-code syntax, since we consider a program as a sequence of *atomic* statements, we must view the statements as machine instructions.

Note that in the following algorithms, indentation is significant in order to express inner statements.

CHAPTER 11
CRITICAL SECTIONS WITH 2 PROCESSES

When more than one process tries to access the same code segment, that segment is known as a *critical section*. A critical section contains shared variables or resources that must be synchronised in order to maintain the *consistency* of data variables.

This chapter presents four attempts to solve the critical section problem for *two* processes and finally a correct solution, DEKKER's algorithm. The four attempts are useful in order to illustrate *typical bugs* in concurrent programs (i.e., deadlock, live-lock, starvation, violation of mutual exclusion).

11.1 Problem definition

Concurrent access to shared resources can lead to erroneous behaviour. As a result, a part of the program has to be protected to avoid concurrency-related problems. All critical sections are protected in the sense that only one process (or thread) at a time is allowed to execute the section. A solution is considered correct if it respects the following requirements:

1. *Mutual exclusion* must be enforced: the execution of the same critical sections cannot be interleaved (to some extent, this behaviour looks like an *atomic* section).
2. A process that halts in its *non* critical section must do so without interfering with other processes.
3. No *deadlock*: the execution should avoid a situation whereby two or more processes wait for each other.
4. No *starvation*: no process should wait indefinitely to enter its critical section.
5. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
6. No assumptions are made about relative process speeds or number of processors.
7. A process remains inside its critical section for a *finite* time only.

Template for two processes			
Global variables			
p		q	
loop forever		loop forever	
p1:	non-critical section	q1:	non-critical section
p2:	pre-protocol	q2:	pre-protocol
p3:	**critical section**	q3:	**critical section**
p4:	post-protocol	q4:	post-protocol

Figure 11.1: Template

First attempt			
integer turn $\leftarrow 1$			
p		q	
loop forever		loop forever	
p1:	non-critical section	q1:	non-critical section
p2:	await turn $= 1$	q2:	await turn $= 2$
p3:	critical section	q3:	critical section
p4:	turn $\leftarrow 2$	q4:	turn $\leftarrow 1$

Figure 11.2: A first attempt

8. No *live-lock*: two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

A simplistic "solution" to the critical section problem is to never allow a process to enter its critical section. However, one of the three conditions would be violated. Which one?

In practice, the solutions consist of a *pre-protocol* and *post-protocol* that go around the critical section (a piece of code) to ensure (ideally) that the previous conditions are respected. A solution must respect the template of Figure 11.1.

11.2 First protocol attempt

A first, rather intuitive way to deal with critical section is to use a variable `turn`, for which each process waits for a specific value in order to enter its critical section (see Figure 11.2). Note that the statement "await condition" corresponds to a *busy wait loop*, i.e., the statement waits until the condition is true. Note also that the variable "`turn`" is stored in a *shared* memory.

This first attempt is not correct: if p does not enter its critical section (because, for example,

Second attempt	
boolean wantp ← false, wantq ← false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp ← true	q3: wantq ← true
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Figure 11.3: A second attempt

Second attempt (abbreviated)	
boolean wantp ← false, wantq ← false	
p	q
loop forever	loop forever
p1: await wantq = false	q1: await wantp = false
p2: wantp ← true	q2: wantq ← true
p3: wantp ← false	q3: wantq ← false

Figure 11.4: Second attempt (abbreviated)

the non-critical section does not finish), then *q* suffers from *starvation*. This is because we have made no assumptions about how the program progresses in its *non*-critical section. However, it is not possible for a process to stop in the "middle" of a critical section or to never leaves its critical section.

A process is permitted to terminate in the *non*-critical section. The algorithm should be prepared to handle this specific situation.

11.3 Second protocol attempt

To avoid the problem of the first attempt, we can assign one (boolean) variable per process (`wantp` & `wantq`) as shown in Figure 11.3.

The state diagram of the abbreviated version of the second attempt (Figure 11.4) is depicted in Figure 11.5. Looking at the state diagram, we can see that mutual exclusion is *violated* in state (p_3, q_3, true, true). Indeed, with wantp=true and wantq=true, both processes can enter their critical section simultaneously.

11.4 Third protocol attempt

We suppose that the "await" statement is part of the critical section. To do so, we move this statement *after* the assignment (see Figure 11.6). However, this technique easily shows a

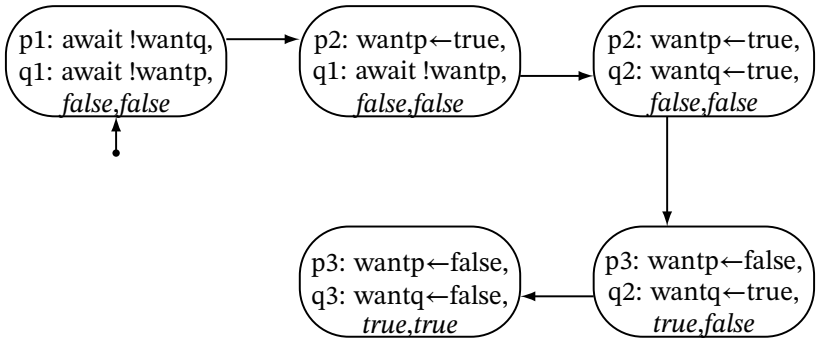


Figure 11.5: State diagram of the second attempt (abbreviated)

Third attempt	
boolean wantp ← false, wantq ← false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Figure 11.6: A third attempt

Fourth attempt	
boolean wantp ← false, wantq ← false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: wantp ← false	q4: wantq ← false
p5: wantp ← true	q5: wantq ← true
p6: critical section	q6: critical section
p7: wantp ← false	q7: wantq ← false

Figure 11.7: A fourth attempt

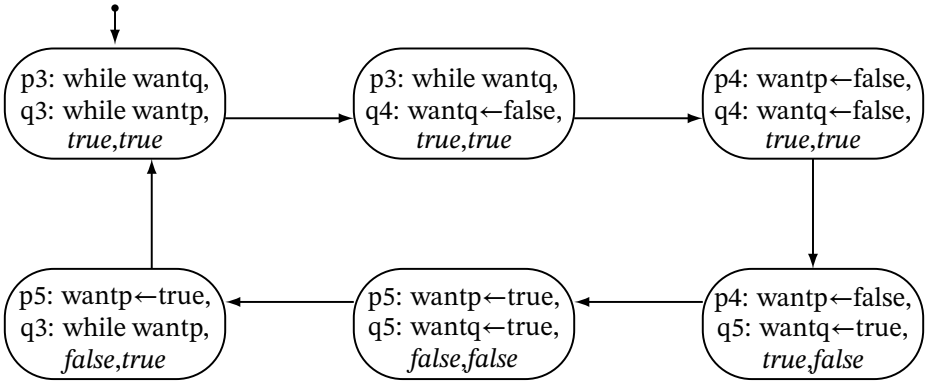


Figure 11.8: A fourth attempt where processes starve

big problem: the execution $p_1 \to q_1 \to p_2 \to q_2$ leads to a *deadlock* situation where both p and q wait for the control variable to change. Deadlocks are situations where the system is frozen (processes are in the *blocked* state). We can distinguish them from *live-locks* where processes execute statements (in the *running* state) but are otherwise stuck (i.e., nothing useful gets done).

In this deadlock scenario, two processes are obstructed at the "await" statement.

11.5 A fourth protocol attempt

We can introduce a loop in each process to allow a change in the order of execution as shown in Figure 11.7 but this leads to another problem: if the two programs execute statements *alternately* they will stay stuck in the loop and *starve* as illustrated with Figure 11.8. This is known as a "live-lock".

DEKKER's algorithm			
boolean wantp ← false, wantq ← false ; integer turn ← 1			
p		q	
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
p2:	wantp ← true	q2:	wantq ← true
p3:	while wantq	q3:	while wantp
p4:	if turn = 2	q4:	if turn = 1
p5:	wantp ← false	q5:	wantq ← false
p6:	await turn = 1	q6:	await turn = 2
p7:	wantp ← true	q7:	wantq ← true
p8:	critical section	q8:	critical section
p9:	turn ← 2	q9:	turn ← 1
p10:	wantp ← false	q10:	wantq ← false

Figure 11.9: DEKKER's algorithm

11.6 DEKKER's algorithm

Finally, we introduce (Theodorus) DEKKER's algorithm.[1] It is somehow a combination of both first and fourth attempts. See Figure 11.9. Proving that the DEKKER's algorithm is correct is not the purpose of this course. Interested students can make the link to their formal verification course(s).

The DEKKER algorithm is not perhaps the most interesting part of this chapter, but rather the four failed attempts which exemplify classic issues encountered when solving concurrent problems in computer science. These include deadlock, live-lock, and violations of mutual exclusion.

[1] DIJKSTRA generalised this algorithm for an arbitrary number of processes

CHAPTER 12

CRITICAL SECTIONS WITH N PROCESSES

The previous chapter showed an algorithm which allows you to simultaneously execute *two* processes with a critical section, without mutual exclusion, starvation, live-lock, or deadlocks. In this chapter, we will introduce the solution, the Bakery algorithm, which was designed by Leslie LAMPORT and which allows you to deal with the general case of N processes having the same critical section.

12.1 Bakery algorithm

The idea is to assign to each process a *ticket* (like at the bakery), and to allow the process with the lowest ticket number to enter the critical section. With two processes, it gives the Figure 12.1.

But of course, we want to expand this algorithm from two process to N processes. Let us generalise it in Figure 12.2. Please note that:

- the value of i can be seen as a process identifier (e.g., the process ID, the PID, in the Unix family).

Bakery algorithm (two processes)	
integer np ← 0, nq ← 0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: np ← nq + 1	q2: nq ← np + 1
p3: await nq = 0 or np ≤ nq	q3: await np = 0 or nq < np
p4: critical section	q4: critical section
p5: np ← 0	q5: nq ← 0

Figure 12.1: Bakery algorithm for two processes

Bakery algorithm (N processes)
integer array[1..n] number ← [0,...,0]

```
      loop forever
p1:       non-critical section
p2:       number[i] ← 1 + max(number)
p3:       for all other processes j
p4:           await (number[j] = 0) or (number[i] ≪ number[j])
p5:       critical section
p6:       number[i] ← 0
```

Figure 12.2: Bakery algorithm

Bakery algorithm without atomic assignment
boolean array[1..n] choosing ← [false,...,false]
integer array[1..n] number ← [0,...,0]

```
      loop forever
p1:       non-critical section
p2:       choosing[i] ← true
p3:       number[i] ← 1 + max(number)
p4:       choosing[i] ← false
p5:       for all other processes j
p6:           await choosing[j] = false
p7:           await (number[j] = 0) or (number[i] ≪ number[j])
p8:       critical section
p9:       number[i] ← 0
```

Figure 12.3: Bakery algorithm without atomic assignment

- The notation number[i] ≪ number[j] is an abbreviation of:
  - number[i]<number[j] or
  - number[i] = number[j] and $i < j$

  consequently we get a *strict total order*.

With the Bakery algorithm is it possible to have 2 processes with the same ticket number?

From a practical perspective the computation of max(number) (statement p2 in Figure 12.2) cannot be considered as atomic (because we need to scan an array). In order to solve this, we nonetheless have to consider the computation of max(number) *non*-atomic even though the solution is more complex see Figure 12.3.

This solution avoids an overlap in writing. However, there might be an overlap in reading, but LAMPORT has demonstrated that this is not a problem [15].

CHAPTER 13
SEMAPHORES

13.1 Introduction

The previous chapters looked at how to solve the critical section problem with programs written in machine language, i.e., where the atomic statements were "really" atomic at processor scale.

This chapter covers the notion of *semaphores*, which are high(er)-level constructions used in the design of concurrent programs.

13.2 Process state

A process can be described as one of those five states:

- Inactive: initial state;
- Ready: waiting to be executed by the CPU (all the necessary resources for its execution are available);
- Running: when its computations are progressing;
- Blocked: a blocked process is not eligible to run;

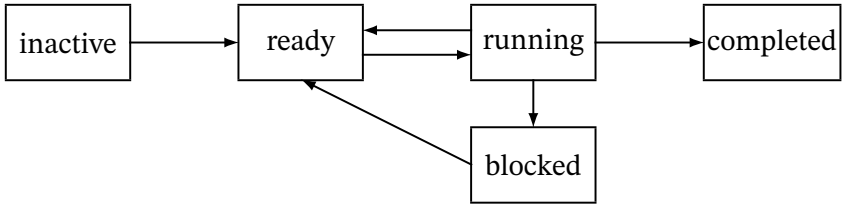


Figure 13.1: Process states

- Completed: when the last statement has been executed.

Figure 13.1 illustrates the links between those states.

13.3 The semaphore data type

A semaphore S is made of two components:

- An unsigned integer V;
- A set of processes L.

A semaphore is initialised with a positive (or null) value for V and an empty set for L:

$$S \leftarrow (k, \varnothing).$$

There are two *atomic* operations on semaphores:

- $wait(S)$, to get access to (or queue for) the shared resource (see Algorithm 6)
- $signal(S)$, to release a shared resource and potentially wake up a waiting process (see Algorithm 7)

In the literature these operations are often defined under the names $P()$ and $V()$; these terms were invented by Edsger DIJKSTRA, and come from the Dutch "Proberen" and "Verhogen", meaning "to test" and "to raise".

Algorithm 6 $wait(S)$ where p denotes the process executing the operation

if $V > 0$ **then**
  $V \leftarrow V - 1$
else
  $L \leftarrow L + p$
  $p.state \leftarrow blocked$
end if

Algorithm 7 $signal(S)$

if $L = \varnothing$ **then**
  $V \leftarrow V + 1$
else
  Let $q \in L$
  $L \leftarrow L \setminus q$
  $q.state \leftarrow ready$
end if

A *binary* semaphore is a particular semaphore where the integer part, V, can only take two values 0 and 1.

Critical section with semaphores (two processes)	
binary semaphore S ← (1, ∅)	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

Figure 13.2: Use of binary semaphores for two processes

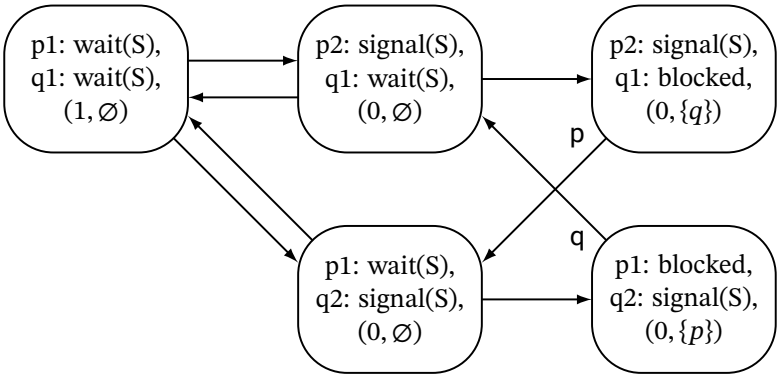


Figure 13.3: Binary semaphore state diagram

13.4 Critical section problem with semaphores

13.4.1 Critical section with two processes

With two process, we will use *binary* semaphores to lock the critical section, see Figure 13.2.

Figure 13.3 shows how it respects the concurrency rules.

13.4.2 Critical section with *N* processes

The previous algorithm can be extended to *N* processes, where all the processes respect the template of Figure 13.4. Note the existence of *global* variables indicating that usage of a shared memory.

However, this solution might lead to starvation, because the `signal(S)` function uses a unordered set (we randomly activate a process). If we change the function `signal(S)` so that it uses a queue (FIFO order), the problem is fixed.

Critical section with semaphores (N proc.)	
binary semaphore S $\leftarrow (1, \varnothing)$	
	loop forever
p1:	non-critical section
p2:	wait(S)
p3:	critical section
p4:	signal(S)

Figure 13.4: Use of semaphores for N processes

Algorithm Mergesort					
integer array A binary semaphore S1 $\leftarrow (0, \varnothing)$ binary semaphore S2 $\leftarrow (0, \varnothing)$					
sort1		**sort2**		**merge**	
p1:	sort 1st half of A	q1:	sort 2nd half of A	r1:	wait(S1)
p2:	signal(S1)	q2:	signal(S2)	r2:	wait(S2)
p3:		q3:		r3:	merge halves of A

Figure 13.5: Merge sort on three processes

13.5 Synchronisation

The critical section problem (studied in the previous chapters) represents situations where several processes compete for the *same* resource. More generally, synchronisation problems represent situations where processes must coordinate the *order of execution* of operations. In this chapter we will illustrate this kind of problem by considering two use cases.

13.5.1 Merge sort

A first example is the *merge sort* algorithm where the array is divided into two halves which can be sorted *independently* and then merged. However, the merging part cannot start before the completion of the two independent sorting operations. The use of binary semaphores can solve this, see Figure 13.5. The three processes are *synchronised* since merge cannot start its merge operation (statement $r3$) if sort1 *and* sort2 are not completed (i.e., if they reach the statement $p3$ and $q3$, respectively).

13.5.2 Producer-consumer problem

Producer-consumer problems are another example of problem category where processes have to synchronise. These problems are characterised by two *types* of processes:

- The producers: these generate data and transmit it to the consumers;
- The consumers: these make use of the data transmitted by the producers.

Producer-consumer (infinite buffer)	
infinite queue of dataType buffer ← empty queue	
semaphore notEmpty ← (0, ∅)	
producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: append(d, buffer)	q2: d ← take(buffer)
p3: signal(notEmpty)	q3: consume(d)

Figure 13.6: Producer-consumer problem with infinite buffer

Producer-consumer (finite buffer, semaphores)	
finite queue of dataType buffer ← empty queue	
semaphore notEmpty ← (0, ∅) semaphore notFull ← (N, ∅)	
producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

Figure 13.7: Producer-consumer problem with finite buffer

Such situations occur frequently in computer systems (communication lines or browsers, keyboards/OS, word processor/printer...).

Often, the communication between the two parts is *asynchronous*, and we make use of buffers (for instance) to prevent the communication line from becoming saturated.

Infinite buffers

Consider an *infinite* buffer (see Figure 13.6). The synchronisation problem is similar to the merge sort algorithm; the difference is that it happens *repeatedly* within a loop. Also note that the integer part of the semaphore is not bounded!

Bounded buffers

If the buffers are *finite* (which is the case in real life), we must make sure that data does not get written to the buffer when it is *full*. To address that issue, another semaphore, notFull, is required that is initialised to N (the size of the buffer) see Figure 13.7. Note also that the invariant notEmpty + notFull = N holds at the beginning of the loop.

CHAPTER 14
DISTRIBUTED ALGORITHMS

14.1 Introduction

The previous chapters covered the notion of concurrency for programs and processes running on the *same* machine and a shared memory.

This chapter looks at *loosely coupled distributed systems* that communicate by sending and receiving messages over a network. For example clusters, or computing grids (which are many clusters connected together by WAN). However, we use the notion of *nodes* and *processes* as abstractions.

Definition 106 (*Node*)

A node models a physical object such as a computer, whether it has one or several processors. Within a node, we assume synchronisation between processes.

The synchronisation between multiple nodes is performed by exchanging *messages*:

- The `send(type, destination, [parameters])` primitive models a message sent from one node to another.
- The `receive(type, [parameters])` primitive models a message received with optional parameters.

14.2 Distributed critical section problem

The definition of the critical section problem remains the same as in previous chapters. Updating a database or implementing a *global* counter could be examples of the problem for distributed systems. However, the solution of the *Bakery algorithm* (see Section 12.1, page 87) cannot work in itself: it is only possible to directly compare ticket numbers with a shared memory, We don't have any.

Thus we propose then the RICART-AGRAWALA algorithm in Figure 14.1. On each node, two concurrent processes are executed: a `Main` process with the pre- and post-protocol, as usual,

RICART-AGRAWALA algorithm (outline)	
integer myNum ← 0	
set of node IDs deferred ← empty set	
	main
p1:	non-critical section
p2:	myNum ← chooseNumber
p3:	for all *other* nodes N
p4:	send(request, N, myID, myNum)
p5:	await reply's from all *other* nodes
p6:	critical section
p7:	for all nodes N in deferred
p8:	remove N from deferred
p9:	send(reply, N, myID)
	receive
	integer source, reqNum
p10:	receive(request, source, reqNum)
p11:	if reqNum < myNum
p12:	send(reply,source,myID)
p13:	else add source to deferred

Figure 14.1: Outline of the RICART-AGRAWALA algorithm

while the second process `Receive` executes a piece of code when a `request` message has been received. The pre-protocol begins with the selection of a ticket number which is sent in a request to *all* other nodes. The process then waits until is has received reply messages from *all* other nodes. A node receiving a request replies immediately *if* its ticket is smaller than the requested ticket (from the sending node). Otherwise the reply is postponed using the "`deferred`" set of processes. In the post-protocol, the node finally sends their reply message(s).

One question remains open: what does the function *chooseNumber* in p_2? Ticket numbers have to be monotonic (for fairness), and each node has to keep track of the largest number used in the whole system. This gives the complete algorithm shown in Figure 14.2 where $a \ll b$ is an abbreviation of:

- $a < b$ or
- $a = b$ and $\mathrm{id}_a < \mathrm{id}_b$, where id_i represents the identifier (e.g., IP address) of the node that owns the variable i.

consequently we get a *strict total order*.

It is perfectly possible to have identical tickets in separate nodes, the algorithm remains deterministic and provides a solution to the critical section problem.

RICART-AGRAWALA algorithm	
integer myNum ← 0	
set of node IDs deferred ← empty set integer highestNum ← 0 boolean requestCS ← false	
Main	
	loop forever
p1:	non-critical section
p2:	requestCS ← true
p3:	myNum ← highestNum + 1
p4:	for all *other* nodes N
p5:	send(request, N, myID, myNum)
p6:	await reply's from all *other* nodes
p7:	critical section
p8:	requestCS ← false
p9:	for all nodes N in deferred
p10:	remove N from deferred
p11:	send(reply, N, myID)
Receive	
	integer source, requestedNum
	loop forever
p12:	receive(request, source, requestedNum)
p13:	highestNum ← max(highestNum, requestedNum)
p14:	if not requestCS or requestedNum ≪ myNum
p15:	send(reply, source, myID)
p16:	else add source to deferred

Figure 14.2: RICART-AGRAWALA algorithm

Consider the RICART-AGRAWALA algorithm:

1. Can the deferred lists of *all* nodes be nonempty?
2. What is the maximum number of entries in a *single* deferred list?
3. What is the maximum number of entries in *all* the deferred lists together (i.e. the sum of the size of the deferred lists)?

14.3 Consensus

The consensus problem in distributed systems refers to the challenge of achieving a *common* agreement among a group of nodes in the network on a particular value or decision. In a distributed system, each node has its own state and can make decisions *independently*, but as a group, nodes must agree to collectively validate certain values or to take decisions. In the avionics domain, replication of computer and sensor systems is often used to provide

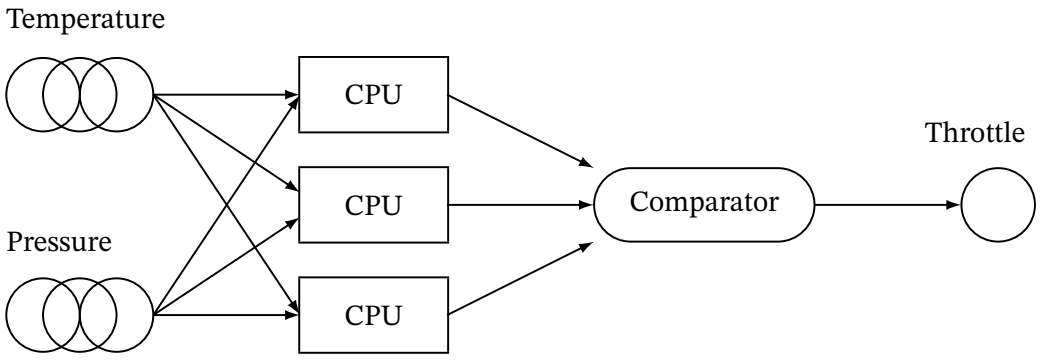


Figure 14.3: Automatic throttle system with sensor and CPU redundancy

redundancy and increase *reliability* of critical systems. Replication involves creating multiple copies of a system or component, such as a computer or sensor, and running them in parallel. These replicated systems can communicate with each other and compare their outputs to detect errors or failures. One application of replication in avionics (or automotive as well as in other safety-critical industries) is in flight control systems. These systems typically involve multiple sensors and computers working together to control the aircraft's attitude and navigation. By *replicating* these systems and comparing their outputs, errors or faults can be detected and corrected, increasing the overall *reliability* and safety of the system. In particular, replication limits (in terms of probability) dangerous situations for *critical systems*.

Automatic throttles are a good example of such problems: they rely on many sensors for temperature and pressure (see Figure 14.3). In such a situation, the sensors (and the CPUs) are duplicated: as a result, a comparator is needed (for example, a majority vote algorithm) in order to obtain a reliable output for the manipulation of the throttle.

However, if the nodes can give *different* answers, this leads to the problem of consensus.

Definition 107 (*Consensus*)

> *The* consensus *problem arises when several nodes have to make an agreement about a defined value.*

However, in distributed systems, this might result in certain failure:

- Crash failure: a node stops sending messages.
- Byzantine failure: a node sends erroneous or arbitrary messages.

Without those mistakes, the comparator would have a relatively easy task: apply a *majority vote* on the received information in a deterministic way.

To resolve this problem consider an analogy: the Byzantine generals problem.

14.3.1 Byzantine generals problem

Imagine several divisions of the Byzantine army besieging an enemy city. Each division is led by its own general. We assume generals can communicate through the use of messengers. After observing enemy activity, all generals must agree on a common plan of action: attack ("A") or retreat ("R"). However, some generals may be traitors and will try to prevent the loyal generals from finding a consensus.

The generals have to find a decision algorithm that guarantees that:

- All the loyal generals agree on a common plan of action. We know that the loyal generals will do what the algorithm tells them to do, but the traitors will do whatever they want.
- A small number of traitors cannot cause loyal generals to pick the wrong course of action.

Of course, this is an analogy. In distributed systems, the generals are an analogy for the nodes and the messengers for the communication channels. This allows us to redefine the possible failures:

- Crash failure: a traitor stops sending messages.
- Byzantine failure: a traitor sends random messages, not only the ones required by the algorithm.

It is easy to detect a crash by using of *timeouts*. However, Byzantine failures force us to consider malicious or erroneous messages. We have to design algorithms that detect them.

14.3.2 One round algorithm

This algorithm is the one suggested earlier: computing the majority of the information received (see Figure 14.4). In the case of a tie, the generals might chose "R". For example, in Figure 14.5 Basil is a traitor, but Zoe and Leo are loyal. The question is, what would happen if Basil disappeared crashed after having sent a message to Leo?

Figure 14.6 shows that if Basil disappeared, the decision made by the two loyal generals is no longer consistent.

Consensus - one-round algorithm
planType finalPlan
planType array[generals] plan
p1: plan[myID] ← chooseAttackOrRetreat p2: for all *other* generals G p3: send(G, myID, plan[myID]) p4: for all *other* generals G p5: receive(G, plan[G]) p6: finalPlan ← majority(plan)

Figure 14.4: One round algorithm

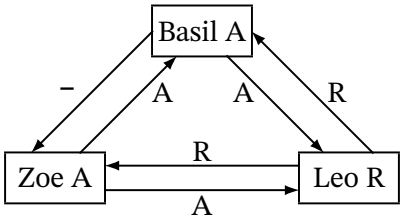


Figure 14.5: One round example

Consensus - Byzantine Generals algorithm	
planType finalPlan	
planType array[generals] plan, majorityPlan planType array[generals, generals] reportedPlan	
p1: plan[myID] ← chooseAttackOrRetreat	
p2: for all *other* generals G	// First round
p3: send(G, myID, plan[myID])	
p4: for all *other* generals G	
p5: receive(G, plan[G])	
p6: for all *other* generals G	// Second round
p7: for all *other* generals G' except G	
p8: send(G', myID, G, plan[G])	
p9: for all *other* generals G	
p10: for all *other* generals G' except G	
p11: receive(G, G', reportedPlan[G, G'])	
p12: for all *other* generals G	// First vote
p13: majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])	
p14: majorityPlan[myID] ← plan[myID]	// Second vote
p15: finalPlan ← majority(majorityPlan)	

Figure 14.7: Byzantine algorithm

Leo		Zoe	
General	Plan	General	Plan
Basil	A	Basil	-
Leo	R	Leo	R
Zoe	A	Zoe	A
Majority	A	Majority	R

Figure 14.6: One round example

14.3.3 Byzantine generals algorithm

To solve the problem above, we consider a second algorithm in Figure 14.7. During the first round, all of the generals share their plan with each other. During the second round, all the generals broadcast the data received from all the other generals. This is why (Figure 14.7, lines p9–p11) the node fills the *matrix* (array of arrays) reportedPlan[][].

Leo				
General	Plan	Reported by		Majority
		Basil	Zoe	
Basil	A		-	A
Leo	R			R
Zoe	A			A
Majority				A

Figure 14.8: Byzantine: Leo

Zoe				
General	Plan	Reported by		Majority
		Basil	Zoe	
Basil	-		**A**	A
Leo	R	-		R
Zoe	A			A
Majority				A

Figure 14.9: Byzantine: Zoe

Now, what would the decision look like for Leo and Zoe after Basil crashed? In Figure 14.8 and 14.9 the decisions made by the two loyal generals are consistent and identical.

Regarding the number of traitors/loyals, what condition is needed in order to reach a consensus?

PART VI

EXERCISES

CHAPTER 15
UNIPROCESSOR SCHEDULING EXERCISES

15.1 Rate Monotonic (RM)

Consider System 108.

System 108

τ_i	C_i	$D_i = T_i$
τ_1	1	5
τ_2	2	8
τ_3	1	10
τ_4	5	20

Exercise 1. Identify the properties of the system.

- Is this system periodic?
- Is the system synchronous or asynchronous?
- Are the deadlines implicit, constrained or arbitrary?

Exercise 2. Verify that this system is schedulable using with RM using the processor *utilisation* technique.

Exercise 3. Verify that this system can be scheduled using the *worst-case response time* technique (Theorem 29, page 26).

Exercise 4. Plot the scheduling of System 108 using RM.

Exercise 5. Find a periodic system with $U(\tau) \leqslant 1$ that can not be scheduled with RM.

Consider System 109.

System 109

τ_i	C_i	$D_i = T_i$
τ_1	0	10
τ_2	3	15
τ_3	4	20

Exercise 6. Verify that System 109 can be scheduled using RM algorithm with the following techniques:

1. Use Theorem 33, page 28 that is based on the processor *utilisation* $U(\tau)$.
2. Use Theorem 29, page 26 that is based on the *worst-case response time*.

15.2 Deadline Monotonic (DM)

Consider the System 110. These are all *periodic*, *synchronous* tasks with *constrained deadlines*.

System 110

τ_i	C_i	D_i	T_i
τ_1	1	5	5
τ_2	2	4	8
τ_3	1	3	10
τ_4	5	15	20

Exercise 7. Verify that this system could be scheduled using the Deadline Monotonic algorithm for priority assignment.

Exercise 8. Find a suitable feasibility interval and plot the scheduling of System 110 using DM.

Consider System 111 made of periodic, synchronous tasks with constrained deadlines.

System 111

τ_i	C_i	D_i	T_i
τ_1	3	5	5
τ_2	2	8	9
τ_3	1	4	11

Exercise 9. Verify whether this system can be scheduled using DM as priority assignment.

Exercise 10. Are there other algorithms that can successfully schedule τ by assigning fixed priorities to each task?

15.3 Systems with arbitrary deadlines

System 112

τ_i	C_i	D_i	T_i
τ_1	1	5	5
τ_2	2	4	8
τ_3	1	15	10
τ_4	5	22	20

Exercise 11. Find a feasibility interval for System 112.

15.4 AUDSLEY

Consider System 113. This system is asynchronous and has constrained deadlines.

System 113

τ_i	O_i	C_i	D_i	T_i
τ_1	10	1	2	3
τ_2	5	2	5	5
τ_3	0	3	10	15

Exercise 12. Which feasibility interval can we use for such system? Compute it.

Exercise 13. Using AUDSLEY's algorithm (Algorithm 1, page 37), find a feasible fixed task priority assignment.

Exercise 14. Plot the scheduling of these 3 tasks using the feasibility interval for asynchronous constrined deadline systems (Theorem 44, page 33) $[O_{\max}, O_{\max} + 2P)$.

Exercise 15. Why can we use $[0, S_n + P)$ from Theorem 45, page 33 as feasibility interval in this case? Calculate this interval.

15.5 Earliest Deadline First (EDF)

System 114

τ_i	C_i	D_i	T_i
τ_1	1	5	5
τ_2	2	4	8
τ_3	1	3	10
τ_4	5	15	20

Exercise 16. Find the feasibility interval for System 114 when using EDF to priorise jobs.

Exercise 17. Plot the scheduling of these tasks with EDF within the smallest possible feasibility interval.

Exercise 18. Find a system of periodic tasks that could be scheduled using EDF, but not using DM.

15.6 Least Laxity First (LLF)

System 115

τ_i	O_i	C_i	D_i	T_i
τ_1	0	1	5	5
τ_2	0	2	4	8
τ_3	0	1	3	10
τ_4	0	5	15	20

Exercise 19. Plot the scheduling of System 115 in the interval $[0, 20)$ using LLF. Consider a situation where priorities are recalculated every time unit.

CHAPTER 16
SIMULATION EXERCISES

16.1 Introduction

In this session, we lay down the basics of uniprocessor scheduling simulation. For some scheduling such as EDF (Section 4.1, page 39), simulation is a required step to check the feasibility of a task-set.

We introduce two types of simulators: constant step ones and event oriented ones. The former category use a constant step δt to sequentially increase the time and check for new events at each step. In contrast, the latter category of event oriented simulators pre-computes all events that can happen and goes through time from one event to the other. Note that constant step simulators can only represent discrete time while event oriented ones do not have such limitation.

In this practical session, we implement a step-based simulator. You can use your favourite programming language, but solutions are only provided in Rust and in Python.

16.2 Constant step simulation

A constant step scheduling simulator requires as input a set of tasks, a function to prioritise jobs and a time step at which to end the simulation.

Let us note the k^{th} job that task τ_i as released at time t by the tuple $\langle k, c, t, \tau_i \rangle$, where c is the remaining computation time. The notation $J.c$ refers to the remaining computation time of job J.

Algorithm 8 shows a high level view of the main simulation loop. It assumes the existence of other functions such as `deadline_missed` or `is_complete`.

Algorithm 8 Constant step simulation

```
τ ← {⟨O_1, C_1, D_1, T_1⟩ ..., ⟨O_n, C_n, D_n, T_n⟩}        ▷ The taskset
P ← a priority function (e.g.: RM, EDF, ...)
jobs ← {}                                                  ▷ The job queue
t ← 0                                                      ▷ The current time step
while t < t_max do
    jobs ← jobs ∪ release_jobs(τ, t)
    if deadline_missed(jobs, t) then return Error
    end if
    J ← P(t, jobs)                                         ▷ Elect a job to run
    if J ≠ None then
        J.c ← J.c − δt                                     ▷ Decrease the remaining computation time
        if is_complete(J) then
            jobs ← jobs ∖ {J}                              ▷ Remove from the job queue
        end if
    end if
    t ← t + δt                                             ▷ Increase the time
end while
```

16.3 Exercises

Based on Algorithm 8, the following exercises aim to guide you step by step throughout the implementation of a constant step simulator.

Exercise 20. Implement basic structures for `Task`, `Job` and `TaskSet`. What are the relevant attributes of these structures?

Exercise 21. Implement a method `release_job` for `Task` that takes as input the time t and returns a `Job` if the task releases a job at time t and nothing otherwise. Create a few tests to make sure your method works as expected.

Exercise 22. For the `Job` structure, create a method

- `deadline_missed` that takes as input the time t and returns whether the deadline has been missed.
- `schedule` to schedule a job for a given amount of time.

Exercise 23. Create a `rate_monotonic` function that takes as input a set of jobs and returns the job with the highest priority. RM is an FTP algorithm, do not hesitate to go back to Section 15.1, page 105 in order to check how priorities are assigned.

Exercise 24. Create a `schedule` function that takes as input a `TaskSet`, a scheduling function and a time step t_{max} as input. The scheduler should loop from $t = 0$ to $t = t_{max}$ with a step on 1, and at each step

1. check for deadline misses
2. check for new job releases
3. if any, schedule the job with the highest priority

If a deadline is missed, the scheduler should show a message and stop early.

Exercise 25. Once your simulator is working, implement the EDF scheduling algorithm (Section 4.1, page 39). EDF is an FJP scheduler (and not FTP).

Exercise 26. How can you improve the efficiency of your simulator? So far, we have increased the time by one step at a time, but is that really efficient? What size of step could you safely take without missing any event?

CHAPTER 17
MULTIPROCESSOR SCHEDULING EXERCISES

17.1 Global and partitioned RM

Exercise 27. Find a system that can be scheduled using global RM, but not using partitioned RM with the same hardware.

Hint: think about processor utilisation.

Exercise 28. (Optional) Find a system that can be scheduled using partitioned RM, but not using global RM (on the same hardware).

17.2 Partitioned EDF

Consider System 116. We assume that all these tasks are independent and preemptible, and a hardware system with 3 identical processors.

System 116

τ_i	C_i	$D_i = T_i$
τ_1	1	5
τ_2	3	12
τ_3	1	10
τ_4	5	20
τ_5	2	10
τ_6	5	20
τ_7	25	30
τ_8	1	20

Exercise 29. What conclusions about the schedulability of this system could you make using the FFDU test (Theorem 78, page 57)?

Exercise 30. Find the partitioning of these 8 tasks using FFDU heuristic.

17.3 EDF$^{(k)}$

Consider System 117. Assume that all these tasks are independent and preemptible.

System 117

τ_i	C_i	$D_i = T_i$
τ_1	14	19
τ_2	1	3
τ_3	2	7
τ_4	1	5
τ_5	1	10

Exercise 31. Find the number of processors required by the system in case we want to use global EDF scheduler (Section 8.2.2, page 63).

Exercise 32. Find the number of processors m required by the system (and the optimal value of k) with EDF$^{(k)}$ scheduling (Theorem 93, page 64 and Theorem 94, page 65).

CHAPTER 18
CONCURRENCY EXERCISES

18.1 Protocol 1

Let's consider the following protocol of management of critical sections.

First attempt	
integer turn ←1	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn ←2	q4: turn ←1

This protocol can be shortened to

First attempt (simplified)	
integer turn ←1	
p	q
loop forever	loop forever
p1: await turn = 1	q1: await turn = 2
p2: turn ←2	q2: turn ←1

Exercise 33. Show that this protocol ensures mutual exclusion. Use this simplified version of the protocol in order to minimize the size of diagrams.

Exercise 34. Use the state diagram in order to show that this protocol guarantees the absence of deadlock.

Exercise 35. Show that this first protocol could lead to starvation (if the time that one process passes in the non-critical section is not limited).

18.2 Protocol 2

Let's consider the following protocol of management of critical sections.

Second attempt	
boolean wantp ←false, wantq ←false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp ←true	q3: wantq ←true
p4: critical section	q4: critical section
p5: wantp ←false	q5: wantq ←false

Second attempt (simplified)	
boolean wantp ←false, wantq ←false	
p	q
loop forever	loop forever
p1: await wantq = false	q1: await wantp = false
p2: wantp ←true	q2: wantq ←true
p3: wantp ←false	q3: wantq ←false

Exercise 36. Show that this protocol can not guarantee mutual exclusion.

Exercise 37. Could we have a deadlock and/or livelock?

18.3 Protocol 3

Let's consider the following protocol of management of critical sections.

Third attempt	
wantp ←false, wantq ←false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ←true	q2: wantq ←true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ←false	q5: wantq ←false

Third attempt : simplified	
wantp ←false, wantq ←false	
p	q
loop forever	loop forever
p1: wantp ←true	q1: wantq ←true
p2: await wantq = false	q2: await wantp = false
p3: wantp ←false	q3: wantq ←false

Exercise 38. Create a state diagram of this protocol and verify that we have mutual exclusion.

Exercise 39. Show that this protocol could lead to a deadlock and describe that scenario.

18.4 Protocol 4

Let's consider the following protocol of management of critical sections. This protocol guarantees the absence of deadlocks and mutual exclusion.

Fourth attempt	
boolean wantp ←false, wantq ←false	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ←true	q2: wantq ←true
p3: while wantq	q3: while wantp
p4: wantp ←false	q4: wantq ←false
p5: wantp ←true	q5: wantq ←true
p6: critical section	q6: critical section
p7: wantp ←false	q7: wantq ←false

Exercise 40. Use a subset of the state diagram of protocol 18.4 to show ot could lead to the livelock shown in Figure 18.1.

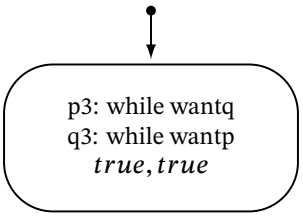


Figure 18.1: Livelock state visited over and over

Exercise 41. Why is it a livelock and not a deadlock?

18.5 Protocol 5

Consider Lamport's algorithm for mutex sections.

Bakery algorithm (2 processes)	
integer np ←0, nq ←0	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: np ←nq + 1	q2: nq ←np + 1
p3: await nq = 0 or np ≤ nq	q3: await np = 0 or nq < np
p4: critical section	q4: critical section
p5: np ←0	q5: nq ←0

Exercise 42. Build the state diagram of this algorithm and show that

- it guarantees mutual exclusion
- there is no deadlock
- there is no livelock
- there is no starvation

Appendices

APPENDIX A
ADDITIONAL PROOFS

The content of this appendix is optional for interested readers.

Additional proof of EDF optimality

The following proof of Theorem 52 consider not discrete (dense) time space, useful to show the sufficient condition $U(\tau) \leqslant 1$ (Theorem 53).

Proof. The proof is made by induction. Let $\Delta > 0$ be an infinitesimal value. Consider a feasible schedule S for J, and let $[t_0, t_0 + \Delta)$ be the first time where S differs from EDFJ (the EDF schedule of the job set J). Suppose that the job $j_1 = (a_1, e_1, d_1)$ is scheduled in S in that interval while $j_2 = (a_2, e_2, d_2)$ is scheduled in EDFJ. Since S meets all the deadlines, it feasibly schedules j_2 in $[a_2, d_2)$ (possibly preemptively) and j_2 completes before d_2. This implies that S schedules j_2 for a duration of Δ before d_2. By definition of EDF, $d_2 \leqslant d_1$; thus S schedules j_2 for a duration of Δ before d_1 as well. The schedule S^1 — obtained by swapping the executions of j_1 and j_2 for a duration of Δ in S — is identical to EDFJ in the interval $[0, t + \Delta)$. EDF optimality follows by induction on t: $S^\infty = \text{EDF}$. □

APPENDIX B

ADDITIONAL ALGORITHMS

Algorithm 9 The Bubble Sort

```
void bubbleSort(int array[], int size) {
// loop to access each array element
for (int step = 0; step < size - 1; ++step){
  // loop to compare array elements
  for (int i = 0; i < size - step - 1; ++i){
    // compare two adjacent elements
    // change > to < to sort in descending order
    if (array[i] > array[i+1]){
      // swapping occurs if elements
      // are not in the intended order
      int temp = array[i];
      array[i] = array[i+1];
      array[i+1] = temp;
    }
  }
 }
}
```

INDEX

AMDAHL's law, 69

anomaly, 60
asynchronous, 31
atomic, 90
atomic statement, 79
atomicity, 77
Audsley's algorithm, 36

bakery algorithm, 87
best fit , 55
Bitonic sort, 72
busy period
    elementary, 30
Byzantine failure, 99
Byzantine generals problem, 99

clairvoyance, 60
concurrency, 75, 81
configuration, 43
consensus, 97
consitency, 81
cost of parallelisation, 70
crash failure, 99
critical instant, 22
critical section, 81, 87, 89, 95

deadline, 10
    absolute, 12
    arbitrary, 16, 29
    constrained, 16
    implicit, 16, 55, 61
    relative, 13
Deadline Monotonic, 28
deadlock, 81
Dekker algorithm, 81
distributed algorithm, 95
distributed critical section, 95
DM, *see* Deadline Monotonic
Dynamic Priority, 45

Earliest Deadline First, 18, 39, 56
    EDF$^{(k)}$, 64
    global, 63
EDF, *see* Earliest Deadline First

fair, 62
feasibility interval, 27, 31, 42, 43
first fit , 55
FTP
    feasibility, 22
    optimality, 22, 25, 29, 35

global scheduling, 51, 59

heuristic, 55
hyper-period, 32

ideal, 62
identical multiprocessor, 50
idle point, 23
incomparability, 52
interleaving, 77, 79

job
    active, 12
    laxity, 45
    response time, 18, 26

lag, 62
laxity, 45
Least Laxity First, 45
live-lock, 81, 85
LLF, *see* Least Laxity First
load, 10

merge sort, 92

migration, 51
Moore Law, 49
multicore, 49
multiprocessor, 49
mutual exclusion, 81

necessary condition, 19
necessay and sufficient condition, 19
next fit, 55
node, 95

optimal, 62, 63

parallel, 49
parallel algorithms, 67
Parallel sorting algorithms, 71
partionned scheduling, 51
partition, 51, 55
partitioning, 55
periodic, 10
Pfair, 62
positive change, 60
post-protocol, 82
pre-protocol, 82
predictability, 8
process, 77
producer-consumer problem, 92
proof by induction, 21
protocol, 82

Rank sort, 71
Rate Monotonic, 11, 22, 28, 56
    optimality, 25
real-time
    constraint, 9
    system, 7, 12
reliability, 98
replication, 98
Ricart-Agrawala algorithm, 95
RM, *see* Rate Monotonic
Round-Robin, 11

scenario, 22, 78
schedule
    preemptive, 17
scheduler, 17
    deterministic, 32
    DP, 17
    FJP, 17, 39
    FTP, 17, 21
    offline, 15
    online, 15
    optimality, 22, 39
    work-conserving, 18, 24
scheduling, *see* scheduler
semaphore, 89
signal, 90
speed-up factor, 69
starvation, 81
state diagram, 78
sufficient condition, 19
synchronisation, 92
system
    asynchronous, 17, 31
    synchronous, 16
    utilisation, 15

task
    active, 12
    asynchronous, 31
    lowest priority viable, 36
    offset, 13, 31
    period, 13
    periodic, 10, 12
    response time, 18, 26
    sporadic, 15
trashing, 46

uniform multiprocessor, 50
unrelated multiprocessor, 50
utilisation, 10

wait, 90
WCET, *see* worst-case execution time
work-conserving, 24
worst fit, 55
worst-case execution time, 12

GLOSSARY

CPS Cyber-Physical System iii, 9

CPU Central Processing Unit 11, 12, 15–18, 22, 23, 25, 32, 40, 43, 45, 79, 89, 98

DM Deadline Monotonic iv, vi, 28, 29, 35, 36, 52, 60, 106, 108

DP Dynamic Priority iv, 17, 45, 46

EDF Earliest Deadline First iv, v, vii, 18, 19, 25, 39–44, 46, 51, 53, 55–57, 63–65, 107–109, 111, 114, 121

FFDU First Fit Decreasing Utilisation v, 56–58, 114

FIFO First In First Out 30, 91

FJP Fixed Job Priority iv, 17–19, 39, 46, 52, 53, 111

FTP Fixed Task Priority iv, 17, 21, 22, 25, 26, 28–30, 34–42, 110, 111

GPOS General Purpose Operating Systems 17

LLF Least Laxity First iv, vii, 45, 46, 108

QoS Quality of Service 9

RM Rate Monotonic iv, vi, vii, 12, 22, 24, 25, 28, 29, 35, 36, 55, 56, 105, 106, 110, 113

RTOS Real-Time Operating System 12, 64

WCET Worst-Case Execution Time 9, 12, 13, 56, 60

WLoG Without Loss of Generality 24, 34

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.

$\lvert x \rvert$	Absolute value of x
$[i, j)$	Half-open interval: $\{k \mid i \leqslant k < j\}$
$\lceil x \rceil$	Ceiling of x, least integer function: $\min_{k \geqslant x} k$
$\lfloor x \rfloor$	Floor of x, greatest integer function: $\max_{k \leqslant x} k$
$\operatorname{lcm}(j, k)$	Least common multiple of j and k
$i \mod j$	returns the remainder of a division, after i number is divided by j (called the *modulus* of the operation)
$\log_b x$	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
$k!$	k factorial: $1 \times 2 \times \cdots \times k$
$\mathbb{N}$	Natural numbers: $0, 1, 2, 3 \ldots$
$\mathcal{O}(f(x))$	Big-oh of $f(x)$
P	The least common multiple of all task period
x^+	$\max(x, 0)$
$B \setminus A$	The set difference of B and A, is the set of elements in B that are not in A
$\stackrel{\text{def}}{=}$	equal by definition (this is a definition)
$\square$	a symbol used to denote the end of a proof, in place of the traditional abbreviation "Q.E.D." for the Latin phrase *"quod erat demonstrandum"*.

BIBLIOGRAPHY

[1] AKL, S. G. *Parallel sorting algorithms*, vol. 12. Academic press, 2014.

[2] ANDERSON, J., AND SRINIVASAN, A. Early-release fair scheduling. In *12th Euromicro Conference on Real-Time Systems* (2000), pp. 35–43.

[3] AUDSLEY, N., TINDELL, K., AND BURNS, A. The end of the line for static cyclic scheduling? In *Proceedings of the EuroMicro Conference on Real-Time Systems* (1993), pp. 36–41.

[4] AUDSLEY, N. C. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Tech. rep., Department of Computer Science, University of York, 1991.

[5] BARUAH, S. Techniques for multiprocessor global schedulability analysis. In *Real-Time Systems Symposium* (2007), IEEE Computer Society, pp. 119–128.

[6] BARUAH, S. Partitioned EDF scheduling: a closer look. *Real-Time Systems 49* (2013), 715–729.

[7] BARUAH, S., COHEN, N., PLAXTON, C. G., AND VARVEL, D. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica 15* (1996), 600–625.

[8] BARUAH, S., GEHRKE, J., AND PLAXTON, C. G. Fast scheduling of periodic tasks on multiple resources. In *9th International Parallel Processing* (1995), pp. 280–288.

[9] BEN-ARI, M. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[10] DEVILLERS, R., AND GOOSSENS, J. Liu and layland's schedulability test revisited. *Information Processing Letters 73*, 5-6 (2000), 157–161.

[11] GOOSSENS, J. *Scheduling of hard real-time periodic systems with various kinds of deadline and offsets contraints*. PhD thesis, Université libre de Bruxelles, 1999.

[12] GOOSSENS, J., AND DEVILLERS, R. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. *Real-Time Systems 13*, 2 (1997), 107–126.

[13] GOOSSENS, J., FUNK, S., AND BARUAH, S. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems: The International Journal of Time-Critical Computing 25* (2003), 187–205.

[14] HONG, K., AND LEUNG, J. On-line scheduling of real-time tasks. In *Real-Time Systems Symposium* (1988).

[15] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. In *Concurrency: the works of Leslie Lamport*. ACM, 2019, pp. 171–178.

[16] LEUNG, J., Ed. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004. Chapter: Scheduling Real-time Tasks: Algorithms and Complexity.

[17] LEUNG, J., AND MERRILL, J. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters 11*, 3 (1980), 115–118.

[18] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery 20*, 1 (January 1973), 46–61.

[19] LIU, J. W. S. *Real-Time Systems*. Prentice-Hall, 2000.

[20] MARTELLO, S., AND TOTH, P. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[21] PADUA, D., Ed. *Encyclopedia of Parallel Computing*, vol. 1 A–D. Springer, 2011.

[22] SRINIVASAN, A., AND BARUAH, S. K. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters 84* (2002), 93–98.

[23] STANKOVIC, J. A., AND RAMAMRITHAM, K. What is predictability for real-time systems? *The Journal of Real-Time Systems 2* (1990), 247–254.

[24] TANENBAUM, A. *Modern Operating Systems*. Pearson, 2007.

ÉVALUATION
DES ENSEIGNEMENTS
PAR LES ÉTUDIANTS

ULB

ÉVALUATION DES ENSEIGNEMENTS

Dès le quadrimestre terminé, évaluez vos enseignements

Pour :
- **Donner une rétroaction à vos enseignants**
- **Proposer des améliorations**
- **Participer à l'évolution des enseignements**
- **Valoriser les activités d'enseignement**

Une évaluation à plusieurs dimensions

Portant sur :
- **La conception de l'enseignement**
- **Le déroulement des séances**
- **L'évaluation des apprentissages (examen)**
- **La prestation des enseignants**

VOTRE AVIS COMPTE !

→ https://www.ulb.be/fr/qualite/l-evaluation-des-enseignements-par-les-etudiants

L'évaluation institutionnelle des enseignements est organisée par l'ULB.
Elle a lieu dès que les enseignements sont terminés. Elle se déroule en deux campagnes d'enquête en ligne après les sessions de janvier et de juin.

L'étudiant répond anonymement à un questionnaire pour chaque enseignement auquel il a participé. Chaque questionnaire est analysé et les résultats sont envoyés aux enseignants et à la commission pédagogique facultaire.

La participation de l'ensemble des étudiants et des enseignants est indispensable pour l'amélioration des programmes

Le label FSC : la garantie d'une gestion responsable des forêts
Les Presses Universitaires de Bruxelles s'engagent !

Les PUB impriment depuis de nombreuses années les syllabus sur du papier recyclé. Les différences de qualité constatées au niveau des papiers recyclés ont cependant poussé les PUB à se tourner vers un papier de meilleure qualité et surtout porteur du label FSC.

Sensibles aux objectifs du FSC et soucieuses d'adopter une démarche responsable, les PUB se sont conformé aux exigences du FSC et ont obtenu en avril 2010 la certification FSC (n° de certificat COC spécifique aux PUB : SCS-COC-005219-HA

Seule l'obtention de ce certificat autorise les PUB à utiliser le label FSC selon des règles strictes. Fortes de leur engagement en faveur de la gestion durable des forêts, les PUB souhaitent dorénavant imprimer tous les syllabus sur du papier certifié FSC. Le label FSC repris sur les syllabus vous en donnera la garantie.

Qu'est-ce que le FSC ?

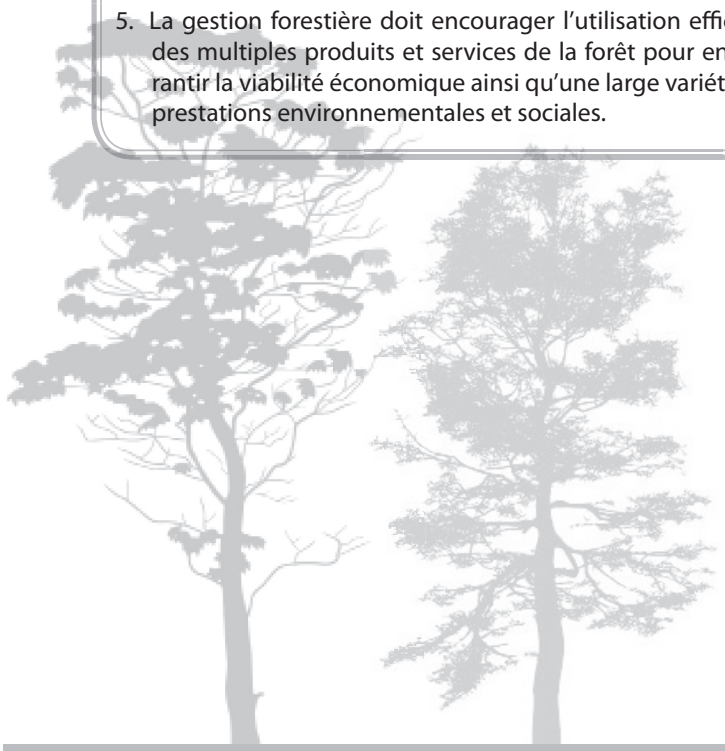
FSC signifie "Forest Stewardship Council" ou "Conseil de bonne gestion forestière". Il s'agit d'une organisation internationale, non gouvernementale, à but non lucratif qui a pour mission de promouvoir dans le monde une gestion responsable et durable des forêts.
Se basant sur dix principes et critères généraux, le FSC veille à travers la certification des forêts au respect des exigences sociales, écologiques et économiques très poussées sur le plan de la gestion forestière.

Quelles garanties ?

Le système FSC repose également sur la traçabilité du produit depuis la forêt certifiée dont il est issu jusqu'au consommateur final. Cette traçabilité est assurée par le contrôle de chaque maillon de la chaîne de commercialisation/transformation du produit (Chaîne de Contrôle : Chain of Custody – COC). Dans le cas du papier et afin de garantir cette traçabilité, aussi bien le producteur de pâte à papier que le fabricant de papier, le grossiste et l'imprimeur doivent être contrôlés.
Ces contrôles sont effectués par des organismes de certification indépendants.

Les 10 principes et critères du FSC

1. L'aménagement forestier doit respecter les lois nationales, les traités internationaux et les principes et critères du FSC.
2. La sécurité foncière et les droits d'usage à long terme sur les terres et les ressources forestières doivent être clairement définis, documentés et légalement établis.
3. Les droits légaux et coutumiers des peuples indigènes à la propriété, à l'usage et à la gestion de leurs territoires et de leurs ressources doivent être reconnus et respectés.
4. La gestion forestière doit maintenir ou améliorer le bien-être social et économique à long terme des travailleurs forestiers et des communautés locales.
5. La gestion forestière doit encourager l'utilisation efficace des multiples produits et services de la forêt pour en garantir la viabilité économique ainsi qu'une large variété de prestations environnementales et sociales.
6. Les fonctions écologiques et la diversité biologique de la forêt doivent être protégées.
7. Un plan d'aménagement doit être écrit et mis en œuvre. Il doit clairement indiquer les objectifs poursuivis et les moyens d'y parvenir.
8. Un suivi doit être effectué afin d'évaluer les impacts de la gestion forestière.
9. Les forêts à haute valeur pour la conservation doivent être maintenues (par ex : les forêts dont la richesse biologique est exceptionnelle ou qui présentent un intérêt culturel ou religieux important). La gestion de ces forêts doit toujours être fondée sur un principe de précaution.
10. Les plantations doivent compléter les forêts naturelles, mais ne peuvent pas les remplacer. Elles doivent réduire la pression exercée sur les forêts naturelles et promouvoir leur restauration et leur conservation. Les principes de 1 à 9 s'appliquent également aux plantations.



FSC®

Le label FSC apposé sur des produits en papier ou en bois apporte la garantie que ceux-ci proviennent de forêts gérées selon les principes et critères FSC.

® FSC A.C. FSC-SECR-0045

FSC, le label du bois et du papier responsable

Plus d'informations ?
www.fsc.be
A la recherche de produits FSC ?
www.jecherchedufsc.be