145071 - Real time operating systems and middleware

Marco Peressutti
230403
marco.peressutti@studenti.unitn.it

January 27, 2023

Contents

Introduction to the Course

Material

- Slides available from moodle
- Interested students can have a look at: *Giorgio Buttazzo*, **HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications**

Exam

Written Exam

- 3 questions, 30 minutes per question
- Each answer gets a score from 0 to 30
- (Optional) project

Oral Exam

- Discussion of the written exam
- Open Questions or Discussion on a project

Prerequisites

- Programming skills: C, maybe C++.
  You must know how ot code in C (optionally C++). This is not about knowing the C syntax, it is about writing good and clean C code.
  To help overcome this lack of prerequisites please consider reading the book *Kerrigan & Ritchie*,**The C Programming Language**
- Knowledge about Operating Systems.
  This prerequisites is met if you have taken the course *Sistemi Operativi 1* or similar exams.
  Alternatively please refer to a good Operating Systems book (e.g. Stallings,...).
  This includes how to use a shell, basic POSIX commands, `make`, how to compile, ...

Overview of the Course

The course will cover 6 main macro areas of real time operating systems and middleware.

1. Real Time Systems:

   - Real-Time Computing and temporal constraints.
     Real time systems are software and hardware systems (hence computing systems), that have to comply with temporal constraints.
   - Definitions and task model
     We will make things much clearer and better defined by introducing a sequence of definitions and mathematical models that will allow us to given this notion of temporal constraint a well founded meaning.

- Real-Time Scheduling
  We will also study solutions that allow us to enforce these real time constraints and this solution will have much to do on how we schedule shared resources.

2. Real-Time programming, RT-POSIX, pthreads,...
   We will move to a concrete ground and see what is the exact shape that these notions take once they are moved in a computer program.
3. Real-Time Scheduling algorithms:

   - Fixed Priority scheduling, RM, DM
   - EDF and dynamic priorities
   - Resource Sharing (Priority Inversion, ...)

   As regards the Real-Time scheduling we will see many interesting policies, but since this is not a course on Real Time scheduling what we will do is provide the knowledge of real time scheduling so that the reader will be able to understand the mechanism of real time operating systems and thereby make best use of these technologies in future projects.
4. Operating System Structure

   - Notes about traditional kernel structures
     In order to keep latencies in check, we need proper technological solutions that make our operating systems differ quite a bit from standard operating systems.
   - Sources of kernel latencies
   - Some approaches to real-time kernels (e.g. dual kernel approach, interrupt pipes, micro-kernels, monolithis kernels and RT)

5. Real-Time Kernels and OSs.
6. Developing Real-Time applications

Real-Time Operating Systems

In order to discuss about the Real-Time systems we need to provide some basic definitions:

Definition 1: Real-Time Operating Systems (RTOS)

Operating Systems that provide support to Real-Time Applications

Definition 2: Real-Time application

the correctness depends not only on the output values, but also on the time when such values are produced

Definition 3: Operating Systems (OS)

- Set of computer programs, of critical programs to be precise: because they have to be written efficiently, otherwise the hardware resources get disrupted, hence the system cannot operate correctly.
- Interface between applications and hardware.
  Whenever an application interacts with an hardware, it is not of the developer interest to directly control the hardware. The Operating System provides an API that enables you to open a connection to a peripheral and takes care of all the low level interactions. On this regard, understanding the notion of interrupt will be of fundamental importance, because it is, essentially, what gave rise to concurrent programming: in the case we would like to interact with a peripheral, rather than continuously check if the peripheral has ended what it is supposed to do, you can tell the peripheral to communicate when

it has completed the given task.
Anyway the Operating systems acts as an interface towards the hardware and hides away all these complex details.
- Control the execution of application programs
- Manage the hardware and software resources

Since the Operating System is something that lies in-between the user application and the hardware resources we can summarize the aforementioned interpretation of

- **Service Provider** for user programs (i.e. exports a programming interface). Service Provider
This concept looks at the OS from the perspective of the software application, in the sense that the Operating Systems provides to the application a series of services:

  – Process Synchronization mechanism
  – Inter-Process Communication (IPC)
  – Process/Thread Scheduling, i.e. ways to create and schedule tasks
  – Input/Output
  – Virtual Memory

  And all these services are accessible through an API.
- **Resource Manager** Resource Manager
If you think at the Operating System as a Resource Manager, then it is something that takes care of many things:

  1. **Process Management** Process Management
  The fact that multiple applications can run at the same time on a PC, even though there is a small amount of processor available to manage these applications. (generally 2,4 or 8).

  The number of application that you are likely to create is often on the hundreds, hence it is necessary to make an appropriate sharing of the limited resources that you have in order for all the applications to live correctly.

  2. **Memory Management** Memory Management
  Supposing one is using a 64-bit architecture, what will happen is that a space of memory is addressable with 64 bit. As a consequence we can imagine that the addressable memory is space has $2^{64} - 1$ memory locations available.

  And each application sees, these much space available for its execution. But however large the space can be in a machine, it will never match the aforementioned size. It could potentially for one task, but in the case a machine is hundreds of tasks and each of them wants to use that much memory, there is no way that the hardware can provide enough physical memory to satisfy all of them.

  To counteract this problem, it is common practice to schedule the memory as well, because you take advantage of the fact that an application CAN use $2^{64} - 1$ memory locations, but at a given time it uses a tiny portion of these locations. It is only that tiny portion of memory locations that needs to be made available to the running task.

  In this scenario, the OS makes it possible to accommodate within the physical memory of the computer these small slices of the available space that the application uses. So somehow it operates as a resource manager for the memory as well.

  3. **File Management** File Management
  4. **Networking**, **Device Drivers**, **Graphical Interface** Networking Device Drivers Graphical Interface

  The important thing is that all of these resources, like the processor, the memory, the drivers etc..., are shared between all the tasks. All these resource managers have to be distributed among all the spectrum of tasks in such a way that the tasks behave properly, i.e. if you do not provide frequently enough these resources they would not be able to deliver the result on time (the OS manages this problem on its own).

In the case we decide to look at the Operating System as a Resource Manager, we need to think of a structure for the OS that makes this resource management effective, effective in the sense that we believe it is the most relevant for our specific range of application.
The way OSs handles devices, interrupt, etc. can be very different (and optimized in very different ways) depending on the type of application one is looking at. However, the type of optimizations we are interested in are those that allow our application to have time-limited execution.

Real-Time Systems

A **Real-Time application** is an application of which the time when a result is produced matters. Real-Time application
In particular:

- a correct result produced too late is equivalent to a wrong result, or to no result.
- it is characterized by temporal constraints that have to be respected.

Example 1: Mobile vehicle

Let us consider a mobile vehicle with a software module that

1. Detects obstacles
2. Computes a new trajectory to avoid them
3. Computes the commands for engine, brakes,...
4. Sends the commands

If you decide to steer to the left or to the right there is a limited amount of time in which the operation has to be carried out. Hence if one can find an extremely effective strategy for steering the wheels but the strategy amounts to setting the values for the motors after one second, it is completely useless, since the vehicle is most likely to crash.
Hence a time violation in executing a task is a critical problem: it means that the developed application is useless and also dangerous.
But then, what is a reasonable time frame for completing the steering operation?
Depends on the speed in which the vehicle is traveling. But no matters if the vehicle is traveling at high or low speed the timing constraint is there, and if it is violated, the vehicle will eventually crash against the obstacle.
As a consequence: when a constraint is set, that constraint needs to be respected. And this is one of the core concept of Real Time.

Hence, a Real-Time is not necessarily synonym of fast execution, but rather of **predictable** execution. predictable
Real time computing has much more to do with predictability than of being quick.
Some examples of temporal contraints are:

- The program must react to external events in a predictable time
- The program must repeat a given activity at a precise rate
- The program must end an activity before a specified time

In this case, we can clearly notice that the temporal constraints can be either one shot events or periodic events, but in both cases, a common characteristic, there is a need of being predictability.
Temporal constraints are modeled using the concept of **deadline**. deadline
Please notice that a Real-Time system is not just a *fast system*, because the speed is always relative to a specific environment, i.e. the steering commands temporal constraint is set by the velocity of the vehicle.
Running faster is good, but does not guarantee the correct behavior. In fact, it is far more valuable to that temporal constraints are always respected; in other terms Real time systems prefer to run fast enough to respect the deadlines, to be reliable.

Hence, the type of analysis that is necessary to perform is not an analysis based on of average/typical cases but rather an analysis of worst case: I have to prove that even in the worst-case scenario, there is not deadline violation.

This predictability creates a wide gap between what a Real Time system is and what a general purpose system is, because general purpose systems are optimised for the average case, but a real time system only cares about the worst case. As a consequence, the way one designs a Real Time system is very different from the way a general purpose system is designed.
In fact:

- When one optimize for the average case, what one would look at is the number of times that an application completes a task every second, and this is called **Throughput**. Throughput
- When one have a worst case requirement, the notion of throughput is not relevant anymore, and the analysis focuses in every single instance the maximum delay will be bounded.

Let us introduce some notion and general terms that we will extensively using during the course

Definition 4: Algorithm

Logical procedure used to solve a problem

Definition 5: Program

Formal description of an algorithm, using a *programming language*

Definition 6: Process

Instance of a program (program in execution)

Definition 7: Thread

Flow of execution, something that is able to execute using your processor along with other threads. These threads can be part of the same program and they can be executed in parallel.

Definition 8: Task

Process or thread

Hence there are two different ways of sharing resources: one are threads in which your share computing resources and memory space, and processes in which you share computer resources but each of the processes has its own memory space.
Unfortunately, there is no common definition of a task: somebody use the terms with the same meaning as a thread and sometimes it is used with the same meaning as a process. In this class we will refer to threads.
Henceforth, when we talk about a task we will refer to a program that it is running and they share the same memory space with other programs.

Chapter 1

Basic Concepts

A task can be seen as a sequence of actions and a deadline must be associated to each one of them. We, therefore, are after is a definition of a formal model that identifies what these tasks or actions are and associate deadlines with them.

1.1 Real-Time Tasks

Definition 9: Real-Time Task (τ_i)

stream of jobs (or instances) $J_{i,k}$, or, in other terms, a sequence of activities that is activated periodically or aperiodically

Each job $J_{i,k} = (r_{i,k}, c_{i,k}, d_{i,k})$ is characterised by the following quantities:

- $r_{i,k}$ activation time — `activation time`
  It is the time at which a task becomes ready for execution; it is also referred as *request time* or *release time*.
- $c_{i,k}$ computation time — `computation time`
  Time necessary to the processor for executing the job without interruption.
- $d_{i,k}$ absolute deadline — `absolute deadline`
  time before which a job should be completed to avoid damage to the system.
- $f_{i,k}$ finishing time — `finishing time`
  The time at which a job finishes its execution
- $\rho_{i,k}$ response time — `response time`
  The time at which a job finishes its execution. Formally this quantity is the difference between the finishing time and the activation time.

$$\rho_{i,k} = f_{i,k} - r_{i,k}$$

Furthermore, since each task i is a sequence of jobs, we need to differentiate between them. That is why each job $J_{i,k}$ is uniquely identified by its task index i and the k-th activation of the i-th task. In addition, we will say that job $J_{i,k}$ respects its deadline if $f_{i,k} \leq d_{i,k}$.

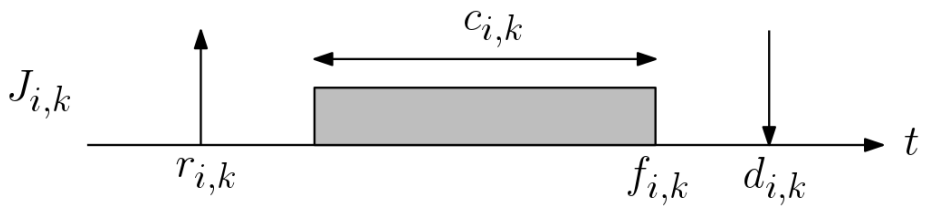

Figure 1.1: Graphical representation of Mathematical model of a Task

This mathematical definition of a job in a real-time task holds regardless of the nature of the task itself. In fact, we can identify three different types of tasks: Periodic tasks, Aperiodic Tasks and Sporadic Tasks. Each of them holds different properties and a different mathematical representation.

1.1.1 Periodic Tasks

Definition 10: Periodic Task

A periodic task $\tau_i = (C_i, D_i, T_i)$ is a stream of jobs $J_{i,k}$, with:

$$\begin{aligned} r_{i,k+1} &= r_{i,k} + T_i \\ d_{i,k} &= r_{i,k} + D_i \\ C_i &= \max_k \{c_{i,k}\} \end{aligned}$$

where:

- T_i **Period**
- D_i **Relative Deadline**
- C_i **Worst-Case Execution Time (WCET)**
- R_i **Worst-Case Response Time (WCRT)**

$$R_i = \max_k \{\rho_{i,k}\} = \max_k \{f_{i,k} - r_{i,k}\}$$

For the task to be correctly scheduled, it must be $R_i \leq D_i$

A periodic task has a regular structure (called **cycle**), in the sense that:

- it is activated periodically with a period of T_i
- it executes a computation
- when the computation terminates, it suspends waiting for the next period

Hence, its fundamental implementation can be represented as:

```
void *PeriodicTask(void *arg)
{
  <initialization>;
  <start periodic timer, period = T>;
  while (condition)
  {
    <read sensors>;
    <update outputs>;
    <update state variables>;
    <wait next activation>;
  }
}
```

1.1.2 Aperiodic Tasks

Definition 11: Aperiodic Task

Aperiodic tasks are not characterised by periodic arrivals, meaning that:

- A minimum interarrival time between activations does not exist
- Sometimes, aperiodic tasks do not have a particular structure

Aperiodic tasks can model tasks responding to events that occur rarely (e.g. a mode change) or tasks responding to events with irregular structure (e.g. bursts of packets from the network,...).

1.1.3 Sporadic Tasks

Sporadic tasks are aperiodic tasks characterised by a **Minimum Interarrival Time (MIT)** between jobs. In this sense they are similar to periodi tasks, but while a periodic task is activated by a periodic timer, a sporadic task is activated by an external event. (e.g. the arrival of a packet from the network)

Hence, its fundamental implementation can be represented as:

```
void *SporadicTask(void *arg)
{
    <initialization>;
    while (condition)
    {
        <computation>;
        <wait events>;
    }
}
```

Formally:

Definition 12: Sporadic Task

A sporadic task $\tau_i = (C_i, D_i, T_i)$ is a stream of jobs $J_{i,k}$, with:

$$r_{i,k+1} \geq r_{i,k} + T_i$$
$$d_{i,k+1} = r_{i,k} + D_i$$
$$C_i = \max_k \{c_{i,k}\}$$

where:

- T_i **Minimum Interarrival Time (MIT)**
- D_i **Relative Deadline**
- C_i **Worst-Case Execution Time (WCET)**
- R_i **Worst-Case Response Time (WCRT)**

Minimum Interarrival Time (MIT)

Relative Deadline

Worst-Case Execution Time (WCET)

Worst-Case Response Time (WCRT)

$$R_i = \max_k \{\rho_{i,k}\} = \max_k \{f_{i,k} - r_{i,k}\}$$

For the task to be correctly scheduled, it must be $R_i \leq D_i$.

1.2 Task Criticality

A deadline is said to be *hard* if a deadline miss causes a critical failure in the system, whereas a task is said to be a **hard real-time task** if all its deadlines are hard, which means that all the deadlines must be guaranteed before starting the task, i.e.

hard real-time task

$$\forall j, \rho_{i,j} \leq D_i \quad \Rightarrow \quad R_i \leq D_i$$

Example 2: Hard Real-Time Task

The controller of a mobile robot, must detect obstacles and react within a time dependent on the robot speed, otherwise the robot will crash into the obstacles

A deadline is said to be *soft* if a deadlien miss causes a degradation in the **Quality of Service (QoS)**, but is not a catastrophic event, whereas a task is said to be a **soft real-time task** if it has soft deadlines.

Quality of Service (QoS)

soft real-time task

In other terms, some deadlines can be missed without compromising the correctess of the system, but the number of missed deadlines must be kept under control, because the *quality* of the results depend on the number of missed deadlines.

Unline the hard real-time task, soft real-time tasks can be difficult to characterize, particularly:

- What's the tradeoff between *non compromising the system correctness* and *not considering missed deadlines*?
- Moreover, some way to express the QoS experienced by a soft real-time task is needed

Examples of QoS definitions could be

- no more than X consecutive deadlines can be missed
- no more that X deadlines in an interval of time T can be missed
- the **deadline miss probability** must be less than a specified value, i.e. deadline miss probability

$$P\{f_{i,j} > d_{i,j}\} \leq R_{max}$$

- the **deadline miss ratio** must be less than a specified value, i.e. deadline miss ratio

$$\frac{\text{number of missed deadlines}}{\text{total number of deadlines}} \leq R_{max}$$

- the maximum **tardiness** must be less than a specified value, i.e. tardiness

$$\frac{R_i}{D_i} < L$$

- ...

Example 3: Audio and Video players

Assuming a framerate of 25 fps, which imply a frame period of 40 ms, if a frame is played a little bit too late, the user might even be unable to notice any degration in the QoS, however, skipped frames can be disturbing.
In fact missing a lot of frames by 5 ms can be better than missing only a few frames by 40 ms.

Example 4: Robotic Systems

Some actuations can be delayed with little consequences on the control quality.

In any case, soft real-time constraints does not mean no guarantee on dealines, given that tasks can have variable execution times between different jobs.
These execution times might depend on different factors:

- Input data
- HW issues (cache effects, pipeline stalls, ...)
- The internal state of the task
- ...

1.3 Schedulability analysis

Schedulability analysis tries to answer the question: Given a task set $\mathcal{T}$, how can we guarantee if it is schedulable or not?

1.3.1 Simulating the hyperperiod

The first possibility is to simulate the system to check that no deadline is missed. The execution time of every job is set equal to the WCET of the corresponding task.

In the case of periodic tasks with no offsets it is sufficient to simulate the schedule until the **hyperperiod** (H = $lcm\{T_i\}$). hyperperiod
In the case of offsets $\phi_i = r_{i,0}$ it is sufficient to simulate until $2H + \phi_{max}$.
If tasks periods are prime numbers the hyperperiod can be very large!

In the case of sporadic tasks, we can assume them to arrive at the highest possible rate, so we fall back to the case of periodic tasks with no offsets.

1.3.2 (Worst-Case) Response Time Analysis

According to the methods proposed by Audsley et al., the longest response time R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its computation time and the interference I_i of the higher priority tasks:

$$R_i = C_i + I_i$$

where:

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

Hence,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1.1}$$

Definition 13: Critical instant

The Critical instant for task τ_i occurs when job $J_{i,j}$ is released at the same time with a job in every high priority task

It is straighforward to notice that if all the offsets of the task set are 0, the first job of every task is released at the **critical instant**. critical instant

A job $J_{i,j}$ released at the critical instant experiences the maximum response time for τ_i:

$$\forall k, \quad \rho_{i,j} \geq \rho_{i,k}$$

No simple solution exists for this equation since R_i appears on both sides of the equation. Thus, the worst-case response time of task τ_i is given by the smallest value of R_i that satisfies equation 1.1. Notice, however, that only a subset of points in the interval $[0, D_i]$ need to be examined for feasibility. In fact, the interference on τ_i only increases when there is a release of a higher-priority task.

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

$$I_i^{(k)} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j \tag{1.2}$$

Then the calculation of R_i is performed as follows:

1. Iteration starts with $R_i^{(0)} = \sum_{j=1}^{i} C_j$, which is the first point in time that τ_i could possibly complete
2. The actual interference I_i^k in the interval $[0, R_i^{(k)}]$ is computed by equation 1.2
3. If $I_i^{(k)} + C_i = R_i^{(k)}$, then $R_i^{(k)}$ is the actual worst-case response time of task τ_i; that is, $R_i = R_i^{(k)}$. Otherwise, the next estimate is given by

   $$R_i^{(k+1)} = I_i^{(k)} + C_i$$

   and the iteration continues from step 2.

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

The response time analysis is an efficient algorithm: in the worst case, the number of steps N for the algorithm to converge is exponential and it depends on the total number of jobs of higher priority tasks in the interval $[0, D_i]$:

$$N \propto \sum_{h=1}^{i-1} \left\lceil \frac{D_h}{T_h} \right\rceil$$

If s is the minimum granularity of the time, then in the worst case $N = \frac{D_i}{s}$. However, such worst case is very rare, usually the number of steps is low.

1.3.3 Processor Demand Analysis

Another necessary and sufficient test for checking the schedulability of fixed priority systems with constrained deadlines was proposed by Lehoczky, Sha and Ding. The test is based on the concept of Level-i workload, defined as follows

Definition 14: Level-i workload

The Level-i workload $W_i(t)$ is the cumulative computation time requested in the interval $(0, t]$ by task τ_i and all the tasks with priority higher than p_i

The basic idea is very simple: in any interval, the computation demanded by all tasks in the set must never exceed the available time.
The problem is: how to compute the time demanded by a tast set $\mathcal{T}$?
Since we have to look only at jobs released at the critical instant, we can consider all offsets equal to zero and only consider the first job of each task...

Definition 15: Processor Demand

Given an interval $[t_1, t_2]$, let $\mathcal{J}_{t_1,t_2}$ be the set of jobs started after t_1 and with deadline lower than or equal to t_2:

$$\mathcal{J}_{t_1,t_2} = \{J_{i,j} : r_{i,j} \geq t_1 \wedge d_{i,j} \leq t_2\}$$

The processor demand in $[t_1, t_2]$ is defined as:

$$W(t_1, t_2) = \sum_{J_{i,j} \in \mathcal{J}_{t_1,t_2}} c_{i,j}$$

Worst case: use C_i instead of $c_{i,j}$

Guaranteeing a task set $\mathcal{T}$ based on $W(t_1, t_2)$ can take a long time.
In fact, it must hold

$$\forall (t_1, t_2) \quad W(t_1, t_2) \leq t_2 - t_1$$

This means that the test requires to check all the (t_1, t_2) combinations in a hyperperiod.
However, we only need to check the first job of every task τ_i.

The quantity $W_i(t_1, t_2)$ is the time demanded in $[t_1, t_2]$ by all tasks τ_j with $p_j \geq p_i$ ($\Rightarrow j \leq i$)

We can consider only $W_i(0, t)$.
For task τ_i only check $W_i(0, t)$ for $0 \leq t \leq D_i$.
Change $\forall$ into $\exists$: consider worst case for $W_i()$
The number of jobs in $[0, t]$ is $\left\lfloor \frac{t}{T_i} \right\rfloor$
Use $\lceil\rceil$ instead

We already have hints about computing an upper bound for $W_i(0, t)$...

$$W_i(0, t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t}{T_h} \right\rceil C_h$$

Task τ_i is schedulable if and only if $\exists t : 0 \leq t \leq D_i \wedge W_i(0, t) \leq t$.
A task set $\mathcal{T}$ is schedulable if and only if

$$\forall \tau_i \in \mathcal{T}, \quad \exists t : 0 \leq t \leq D_i \wedge W_i(0, t) \leq t$$

Sometimes, different notations in literature:

$$W_i(0, t) \rightarrow W_i(t) - \sum_{h=1}^{i} \left\lceil \frac{t}{T_h} \right\rceil C_h$$

This is equivalent, because $0 \leq t \leq T_i$.
Someone defines

$$L_i(t_1, t_2) = \frac{W_i(t_1, t_2)}{t_2 - t_1}$$

$$L_i = \min_{0 \le t \le D_i} L_i(0,t) \quad ; \quad L = \max_{\tau_i \in \mathcal{T}} L_i$$

The guarantee tests then becomes:

- Task τ_i is schedulable iff $L_i \le 1$
- $\mathcal{T}$ is schedulable iff $L \le 1$

The test might still be long (need to check many values of $L(0,t)$ to find the minimum)...
The number of points to check for computing W_i or L_i can be reduced:

$$S_i = \left\{ k\,T_h | h \le i; 1 \le k \le \left\lfloor \frac{T_i}{T_h} \right\rfloor \right\}$$

multiples of T_h for $h \le i$

$$L_i = \min_{t \in S_i} L_i(0,t)$$

1.3.4 Processor Utilization Factor test

The feasibility of a task set with contrained deadlines could be guaranteed using the utilization based test, by reducing tasks' periods to relative deadlines:

$$U_{lub} = \sum_{i=1}^{n} \frac{C_i}{D_i} \le n(2^{1/n-1})$$

However, such a test would be quite pessimistic, since the workload on the processor would be overestimated.
For this reason this test is **sufficient but not necessary**.

Nonetheless, in many cases it is useful to have a very simple test to see if a task set is schedulable. This sufficient test is based on the **Utilisation bound**.

Utilisation bound

Definition 16: Utilisation Least Upper Bound

The utilisation least upper bound for a scheduling algorithm $\mathcal{A}$ is the smallest possible utilisation U_{lub} such that, for any task set $\mathcal{T}$, if the task set's utilisation U is not greater than U_{lub} ($U \le U_{lub}$), then the task set is schedulable by algorithm $\mathcal{A}$

In other terms, we can consider that each task uses the processor for a fraction of time

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

The total processor utilisation is

$$U = \sum_i \frac{C_i}{T_i}$$

which we will consider as a measure of the processor's load.

Given these definition, the necessary condition for the schedulability of a task set is:

- If $U > 1$ the task set is surely not schedulable
- If $U \le U_{lub}$, the task set is schedulable
- If $U_{lub} < U \le 1$ the task set may or may not be schedulable

Ideally a value of $U_{lub} = 1$ would be optimal.

In general, given that the tasks might not always have relative deadline equals to the period the formulation of the total processor utilisation considers the relative deadline:

$$U' = \sum_{i=1}^{n} \frac{C_i}{D_i}$$

This approach considers the worst case for a task... hence if the task set is guaranteed using the relative deadlines, it must hold that the test holds even when considering the period.

The bound is very pessimistic: most of the times, a task set with $U > U_{lub}$ is schedulable. A particular case is when tasks have periods that are harmonic.

Definition 17: Harmonic task set

A task set is harmonic if, for every two tasks τ_i, τ_j either T_i is multiple of T_j or T_j is multiple of T_i

For a harmonic task set, the utilisation bound is $U_{lub} = 1$. (Foreshadowing: Rate Monotonic is an optimal algorithm for harmonic task sets)

Real-Time Scheduling

Chapter 2

Periodic Task Scheduling

The term task is used to indicate a schedulable entity (either a process or a thread), in particular:

- A thread represents a flow of execution (it executes with shared resources, multi thread within the same process)
- A process represents a flow of execution + private resources (it executes with its own resources), such as address space, file table, ...

Tasks do not run on bare hardware, but then how can multiple tasks execute on one single CPU? The OS kernel is a piece of the operating system that takes care of multi-programming and somehow it is able to create the illusion that each CPU/processor has its own space, whereas in fact it is sharing the same resources with other processes.
In the end the kernel provides the mechanism that enable multiple tasks to execute in parallel; in a sense tasks have the illusion of executing concurrently on a dedicated CPU per task.

On this regard, with the term concurrency we refer to the simultaneous execution of multiple threads/processes in the same PC.
Concurrency is implemented by multiplexing tasks on the same CPU. Tasks are alternated on a real CPU and the task scheduler decides which task executes at a given instant in time. In other terms, in order to implement the concurrency mechanism it is necessary to introduce this new component (i.e. the task scheduler), since it makes sure that the time of your pc is shared between the different processes or tasks that compete for the reosurces at that time.

Tasks are associated to temporal constraints (a.k.a. deadlines), hence the scheduler must allocate the CPU to tasks so that their deadlines are respected.

2.1 Real Time Scheduling

Definition 18: Scheduler

A scheduler generates a schedule from a set of tasks

1. In the case of Unicore processor system (UP) (simpler definition), a schedule $\sigma(t)$ is a function mapping time t into an executing task.

   $$\sigma : t \to \mathcal{T} \cup \tau_{idle}$$

   where $\mathcal{T}$ is the taskset and τ_{idle} is the idle task

2. For a Symmetric Multipprocessor System (SMP) (m CPUs), $\sigma(t)$ can be extended to map t in vectors $\tau \in (\mathcal{T} \cup \tau_{idle})^m$

Hence a scheduler is responsible for selecting the task to execute at time t.

Definition 19: Scheduling algorithm

Algorithm used to select for each time instant t a task to be executed on a CPU among the ready task

Given a task set $\mathcal{T}$, a scheduling algorithm $\mathcal{A}$ generates the schedule $\sigma_{\mathcal{A}}(t)$.
A task set is schedulable by an algorithm $\mathcal{A}$ if $\sigma_{\mathcal{A}}$ does not contain missed deadlines.
To verify that no missed deadlines occur, a **Schedulability test** checks if $\mathcal{T}$ is schedulable by $\mathcal{A}$.

Schedulability test

2.2 Cyclic Executive Scheduling

Timeline Scheduling, also known as **Cyclic Executive Scheduling**, is one of the most used approaches to handle periodic tasks in defense military systems and traffic control systems.
The methods consists of dividing the tmeporal axis into slots of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements. A timer synchronizes the activation of the tasks at the beginning of each time slot.

Timeline Scheduling

Cyclic Executive Scheduling

Cyclic Executing Scheduling is a **static scheduling algorithm** where **jobs are not preemptable** (i.e. A scheduled job executes until termination).
The slots are statically allocated to the tasks using a **scheduling table**.
In this Scheduling algorithm two quantities are considered:

static scheduling algorithm

scheduling table

- **Major Cycle**: least common multiple of all the tasks' periods (a.k.a. **hyperperiod**)
- **Minor Cycle**: greatest common divisor of all the tasks' periods

Major Cycle

hyperperiod

Minor Cycle

The period timer fires every Minor Cycle Δ.
Hence the implementation of the scheduling algorithm performs as follow:

1. The periodic timer fires every minor cycle
2. Read the scheduling table and execute the appropriate tasks
3. Sleep until next minor cycle

The main advantage of timeline scheduling is its simplicity. The method can be implemented by programming a timer to interrupt with a period equal to the minor cycle and by writing a main program that calls the tasks in the order given in the major cycle, inserting a time synchronization point at the beginning of each minor cycle. Since the task sequence is not decided by a scheduling algorithm in the kernel, but it is triggered by the calls made by the main program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same, can be easily visualized, and it is not affected by jitter (i.e., task start times and response times are not subject to large variations).

In spite of these advantages, timeline scheduling has some problems. For example, it is very fragile during overload conditions. If a task does not terminate at the minor cycle boundary, it can either be continued or aborted. In both cases, however, the system may run into a critical situation. In fact, if the failing task is left in execution, it can cause a domino effect on the other tasks, breaking the entire schedule (timeline break). On the other hand, if the failing task is aborted while updating some shared data, the system may be left in an inconsistent state, jeopardizing the correct system behavior.
Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch.
Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence. The problems outlined above can be solved by using priority-based scheduling algorithms.

2.3 Fixed Priority Scheduling

Fixed Priority Scheduling is a very simple preemptive scheduling algorithm.

To each task τ_i is assigned a fixed priority p_i as an integer number: the higher the number the higher the priority. In the research literature sometimes, authoers use the opposite convention: the lowest the number, the highest the priority.

The active task with the highest priority is scheduled.

Fixed Priority Scheduling has the following priority:

- The response time of the task with the highest priority is minimum and equal to its WCET
- The reposnse time of the other tasks depends on the interference of the higher priority tasks
- The priority assignment may influence the schedulability of a task set
  Problem: how to assign tasks' priorities so that a task set is schedulable?

There are two main approaches to assigning priorities to the task set:

- **Schedulability**, i.e. find the priority assignment that makes all tasks schedulable — Schedulability
- **Response time (optimization)**, i.e. find the priority assignment that minimise the response time of a subset of tasks — Response time (optimization)

By now we consider the first objective only, hence we will investigate the **optimal priority assignment (Opt)**. — optimal priority assignment (Opt)

2.3.1 Rate Monotonic Scheduling

The Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates (that is, with shorter periods) will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: a priority p_i is assigned to the task before execution and does not change over time. Moreover, RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with a shorter period.

In 1973, Liu and Lyland showed that RM is **optimal** among all fixed-priority assignments (with deadline equals to the period and offset equal to 0) in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM.
In addition, RM is an optimal algorithm for harmonic task sets. This holds also for sporadic tasks.

2.3.2 Deadline Monotonic Scheduling

The Dealine Monotonic (DM) priority assignemt weakens the *periodi equals dealine* contraint within a static priority scheduling scheme. This algorithm was first proposed in 1982 by Leung and Whitehead as an extension of Rate Monotonic, where tasks can have relative dealines less than or equal to their period (i.e. *constrained deadlines*).

According to the DM algorithm, each task is assigned a fixed priority p_i inversely proportional to its relative deadline D_i. Thus, at any instant, the task with the shorter relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As RM, DM is normally used in a fully preemptive mode, that is the currently executing task is preempted by a newly arrived task with shorter relative deadline.

The DM priority assignment is **optimal**, meaning that, if a task set is schedulable by some fixed priority assignment (with deadline different from the period and offset equal to 0), then it is also schedulable by DM.
This holds also for sporadic tasks.

2.4 Dynamic Priority Scheduling

RM and DM are optimal fixed priority assignments. Maybe we can improve schedulability by using **dinamic priorities**? Assumption: priorities change from job to job (a job $J_{i,j}$ always has the same priority $p_{h,k}$) — dinamic priorities

2.4.1 Earliest Deadline First (EDF)

The Earliest Deadline First (EDF) algorithm is a dynamic scheduling rule that selects tasks according to their absolute deadlines. Specifically, tasks with realier deadlines will be executed at higher priorities. Since the absolute deadline of a periodic task depends on the current jth instance as

$$d_{i,j} = (j-1)T_i + D_i$$

EDF is a dynamic priority assignment. Moreover, it is typically executed in preemptive mode, thus the currently executing task is preempted whenever another periodic instance with realier deadline becomes active.

Note that EDF does not make any specific assumption on the periodicity of the tasks; hence, it can be used for scheduling periodc as well as aperiodic and sporadic tasks.

Chapter 3

Aperiodic Servers

The scheduling algorithms treated in the previous chapter deals with homogeneous sets of tasks, where all computational activities are periodic. Many real-time control applications, however, require both aperiodic and periodic processes, which may also differ for their criticality. Tipically, periodic tasks are time-driven and execute critical control activities with hard timing contraints aimed at guaranteeing regular activation rates. Aperiodic tasks are usually event-driven and may have hard, soft, or non real-time requirements depending on the specific applications.

When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities. Off-line guarantee of event-driven aperiodic tasks with critical timing contraints can be done only by making proper assumptions on the environment; that is, by assuming a maximum arrival rate for each critical event. This implies that aperiodic tasks associated with critical events are characterized by a minimum interarrival time between consecutive instances, which vounds the aperiodic load. Aperiodic tasks characterized by a minimum interarrival time are called sporadic. They are guaranteed under peak-load situations by assuming their maximum arrival rate.

3.1 Background Execution

The simplest method to handle a set of soft aperiodic activities in the presence of periodic tasks is to schedule them in background; that is, when there are not periodic instances ready to execute. The major problem with this technique is that, for high periodic loads, the response time of aperiodic requests can be too long for certain applications. For this reason, background scheduling can be adopted only when the aperiodic activities do not have stringest timing constraints and the periodic load is not high.

The major advantage of background scheduling is its simplicity. In general, only two queues are needed to implement the scheduling mechanism: one (with a higher priority) dedicated to periodic tasks and the other (with a lower priority) reserved for aperiodic requests. The two queueing strategies are independent and can be realized by different algorithms. Tasks are taken from the aperiodic queue only when the periodic queue is empty. The activation of a new periodic instance causes any aperiodic tasks to be immediately preempted.

3.2 Immediate Execution

Contrary to the Background Execution, aperiodic tasks are served with the highest priority as soon as they come. This however, might cause deadline misses among the periodic tasks.

Aperiodic Servers are the solution to the problem. Normally we associate two parameters with a server:

- C_s: capacity
- T_s: server period

Roughly speaking, the idea is that the served tasks receive no more that C_s time units every T_s. How this is done depends on the specific server technology.

The server is scehduled as any periodic tasks. Priorities are manipulated in favour of the server. Tasks inside the server can be queued with an arbitrary discipline.

3.3 Polling Servers (PS)

The average response time of aperiodic tasks can be improved with respect to background scheduling through the use of a **server**, that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. Like any periodic task, a server is characterized by a **server period** T_s and a computation time C_s, called **server capacity**, or **server budget**. In general, the server is scheduled with the same algorithm used for the periodic tasks, and once active, it serves the aperiodic requests within the limit of its budget. The ordering of aperiodic requests does not depend on the scheduling algorithm used for periodic tasks, and it can be done by arrival time, computation time, deadline or any other parameter.

server

server period

server capacity

server budget

The **Polling Server (PS)** is an algorithm based on such an approach. At regular intervals equal to the period T_s, PS become active and serves the pending aperiodic requests withing the limit of its capacity C_s. If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the budget originally allocated for aperiodic service is discharged and given periodi tasks.

Polling Server (PS)

Note that if an aperiodic request arrives just after the server has suspended, it must wait until beginning of the next period, when the server capacity is replenished at its full value.

3.4 Deferrable Servers (DS)

The **Deferrable Server (DS)** algorithm is a service technique introduced by Lehoczky, Sha, and Strosnider to improve the average response time of aperiodic requests with respect to polling service. As the Polling Server, the DS algorithm creates a periodic task (usually having a high priority) for servicing aperiodic requests. However, unlike polling, DS preserves its capacity if no requests are pending upon the invocation of the server. The capacity is maintained until the end of the period, so that aperiodic requests can be services at the same server's priority at anytime, as long as the capacity has not been exhausted. At the beginning of any server period the capacity is replenished at its full value.

Deferrable Server (DS)

DS provides much better aperiodic responsiveness than polling, since it preserves the capacity until is needed. Shorter response times can be achieved by creating a Deferrable Server having the highest priority among the periodic tasks.

3.5 Sporadic Servers (SS)

The **Sporadic Server (SS)** algorithm is another technique which allow the enhancement of the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set.

Sporadic Server (SS)

The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way it replenishes its capacity. Whereas DS periodically replenish their capacity to full value at the beginning of each server period, SS replenishes its capacity only after it has been consumed by aperiodic task execution.

In order to simplify the description of the replenishment method used by SS, the following terms are defined:

- P_{exe} It denotes the priority level of the task that is currently executing
- P_s It denotes the priority level associated with SS
- **Active** SS is said to be active when $P_{exe} \geq P_s$
- **Idle** SS is said to be idle when $P_{exe} < P_s$
- **RT** It denotes the replenishment time at which the SS capacity will be replenished

- **RA** It denotes the replenishment amount that will be added to the capacity at time RT

Using this terminology, the capacity C_s consumed by aperiodic requests is replenished according to the following rules:

- The replenishment time RT is set as soon as SS becomes active and $C_s > 0$. Let t_a be such a time. The value of RT is set equal to T_a plus the server period

$$RT = t_a + T_s$$

- The replenishment amount RA to be done at time RT is computed when SS becomes idle or C_s has been exhausted. Let t_I be such ta time. The value of RA is set equal to the capacity consumed withing the interval $[t_a, t_I]$

3.6 Constant Bandwidth Servers (CBS)

In this section we present a novel service mechanism, called **Constant Bandwidth Server (CBS)**, which efficiently implements a bandwidth reservation strategy. The Constant Bandwidth Server guarantees that, if U_s is the fraction of processor time assigned to a server (i.e. its bandwidth), its contribution to the total utilization factor is no greater thatn U_s, even in the presence of overloads.

Constant Bandwidth Server (CBS)

The basic idea behind the CBS mechanism can be explained as follows: when a new job enters the system, it is assigned a suitable scheduling deadline (to keep its demand within the reserved bandwidth) and it is inserted in the EDF ready queue. If the job tries to execute more than expected, its deadline is postponed (i.e. its priority is decreased) to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work conserving algorithm, exploiting the available slack in an efficient (deadline-based) way, thus priviging better responsiveness with respect to non-work conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background.

If a subset of tasks is handled by a single server, all the tasks in that subset will share the same bandwidth, so there is no isolation among them. Novertheless, all the other tasks in the system are protected agains overruns occurring in the subset.

In order not to miss any hard deadline, the deadline assignment rules adopted by the server must be carefully designed.

Definition 20: Constant Bandwidth Server

A CBS is characterized by three main quantities:

- an ordered pair (Q_s, T_s) assigned by the user.
  Where Q_s is the maximum budget and T_s is the period of the server. The ratio

$$U_s = \frac{Q_s}{T_s}$$

  is denotes as the server bandwidth.
- The current budget q_s (initialized to 0) managed by the server.
- The scheduling deadline d_s (initialized to 0) managed by the server.

Each served job J_k is assigned a dynamic deadline equal to the current server deadline. Whenever a served job executes, the server budget q_s is decreased by the same amount.

The CBS acts considering the following procedures:

1. When the server budget is exhausted (i.e. $q_s = 0$), the server budget is recharged at the maximum value Q_s and a new server deadline is generated as $d_s = d_s + T_s$. Note that there are no finite intervals of time in which the budget is equal to zero.

2. When a job J_k arrives and the server is active the request is enqueued in a queue of pending jobs according to a given (arbitrary) discipline.

3. When a job J_k arrives and the server is idle, if $q_s \geq (d_s - r_k)U_s$ the server generates a new deadline $d_s = r_k + T_s$ and q_s is recharged at the maximum value Q_s, otherwise the job is served with the last server deadline d_s using the current budget.

4. When a job finishes, the next pending job, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle.

Hence, the server behaviour can be described by the algorithm:

At arrival of job J_k at time r_k $\rightarrow$ Assign d_s
if $\exists$ pending aperiodic request **then**
  enqueue J_k
else
  if $(q_s \geq (d_s - r_k)\, U_s)$ **then**
    $q_s \leftarrow Q_s$
    $d_s \leftarrow r_k + T_s$
  else
    Continue to use the budget q_s with deadline d_s
  end if
end if

3.6.1 CBS Properties

The proposed CBS service mechanism presents some interesting properties that make it suitable for supporting applications with highly variable computation times. The most important one, the **temporal isolation property**, is formally expressed as follows:

temporal isolation property

Theorem 1

The CPU utilization of a CBS with parameters (Q_s, T_s) is

$$U_s = \frac{Q_s}{T_s}$$

independently from the computation times and the arrival pattern of the served jobs.

Lemma 1

Given a set of n periodic hard tasks with processor utilization U_p and a set of m CBSs with processor utilization

$$U_s = \sum_{i=1}^{m} U_{si}$$

the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

The temporal isolation property allows us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most imporant consequence of this result is that soft tasks can be scheduled together with hard tasks without affecting the a priori guarantee, even in the case in which the execution time of the soft tasks are not known or the soft requests exceed the expected load.
Another general technique used in real-time systems for limiting the effects of overruns in tasks with variable computation times is the **resource reservation paradigm**. According to this method, each task is assigned a fraction of the processor bandwidth, just enough to satisfy its timing containts. The kernel, however, must prevent each task from consuming more that the requested amount to protect the other tasks in the systems (**temporal protection**). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor

resource reservation paradigm

temporal protection

with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the behavior of the other tasks.

A simple and effective mechanism for implementing resource reservation in a real-time system is to reserve each task τ_i a specified amount of CPU time Q_i in every reservation period T_s.

Chapter 4

Resource Access Protocols

4.1 Introduction

A **resource** is any software structure that can be used by a process to advance its executino. resource
Tipically, a resource can be a data structure, a set of variables, a main mamory area, a file, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*. A shared resource protexted against concurrent accesses is called an *exclusive resource*.

To ensure consistency of the data structures in exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A piece of code executed under mutual exclusion contraints is called a **critical section**. critical section

Any task that needs to enter a critical section must wait until no other task is holding the resource. A task waiting for exclusive resource is said to be *blocked* on that resource, othertwise it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the resource associated with the critical section becomes *free*, and it can be allocated to another waiting task, if any.

In this chapter, we describe the main problems that may arise in a uniprocessor system when concurrent tasks use shared resources in exclusive mode, and we present some resource access protocols designed to avoid such problems and bound the maximum blocking time of each task. We then show how such blocking times can be used in the schedulability analysis to extend the guaranteee tests derived for periodic task sets.

4.1.1 Atomicity

So far we have assumed that all tasks that run and compete for a processor are independent, which means that they do not interact with one another. But there are several occasions in which this assumption cannot be made (e.g. when two tasks need to share information, exchange variables,...). The other is when you have to compete for shared resources.

In this section we will introduce this problem and in particular we will introduce the notion of atomicity.

Definition 21: Atomic Instruction

An atomic instruction is an instruction whose execution cannot be interleaved with the execution of other instructions

In this sense atomic operations are always sequentialized since they cannot be interrupted. Under these conditions they are safe operations.
On the other hand, non atomic operations can be interrupted, and as such they are not *safe* operations.

Usually, it is preferrable to have, whenever possible, non atomic operations, because they allow you to exploit in full the possibility of scheduling the processor to activities having higher priority

via preemption.

Example 5: Non atomic operations

Conside a simple operation like

$$x = x + 1$$

The variable is stored in a memory address that we call x. Hence, whenever the variable is incremented using this operation, we would have to:

- load the variable x into a register $R0$

```
LD R0, x
```

- increment the register

```
INC R0
```

- store back the value contained in the register into x

```
ST x, R0
```

If the same operation is executed inside an interrupt handler an inconsistency may arise.

Example 6: Interrupt on non-atomic operations

Let us consider that the increment operation is both applied in the normal code and in an interrupt handler code (routine executed in response to an interrupt).
In both cases, the operation is translated into the assembly language using three instructions (load, increment and store).
The program starts executing as follows:

1. The normal code starts executing: the value of x is loaded from memory to the register
2. At some point during this operation something triggers the execution of the interrupt
3. The interrupt handler creates a copy of all the registers
4. The interrupt handler load (once again) x from memory, increments the register and stores the result in memory (the value of x in memory has changed to $x + 1$)
5. Upon the interrupt handler has completed, the saved registers are restored. Hence, the old value of the register (i.e. x) is loaded back, its value is incremented to $x + 1$ and stored into memory at the address of x

The problem is that even though the code should have performed two increments (i.e. the final value should have been $x + 2$), it yields the incorrect result (i.e. $x + 1$).
From a logical point of view two increment operations should have taken place, but in effect one of them not successfully completed because while i was incrementing the variable, I was allowed to be interrupted and I was left with a state that was not up to date.

The nasty problem about this phenomenon is that it does not always happen in this way, because sometimes the function successfully completes before the interrupt is fired.
The example provided is the description of a condition called **critical race**, because you can have multiple execution of your code that interleave the operation in slightly different ways and you obtain different results. `critical race`

This is a nasty problem in computer science because it might not materialize for years.

This is so because you cannot make assumption about the speed of the hardware and on the exact moment when certain events take place (we do not know the order of execution of the hardware instructions).

The case studies proposed are a perfect example of a not atomic operation that should be atomic.

The same behaviour occurs not only in the case of interrupts but also on interleaving tasks: so you could have two tasks running in parallel, both of which are sharing the variable x.

We can give a few definitions that are important for the follow up of our discussion:

Definition 22: Shared Object

An object where the conflict may happen.

Definition 23: Critical section

A critical section is a sequence of operations that cannot be interleaved with other operations on the same resource

Definition 24: Mutual exclusion

Two critical sections cannot be active at the same time (they must be sequentialized). Either one or the other needs to stand by while the other executes

There are three ways to obstain mutual exclusion:

1. Implementing the critical section as an atomic operation.
   This that interrupt are disabled before the execution of the critical section and then are restored upon termination of the operation. The problem with this is that it is really tough because disabling the interrupts the I/O system of the machine is no longer allowed to work properly. (pressing an emergency button will have no effect and the execution of the code will continue),
   Moreover if the critical section is long, no interrupt can arrive during the critical section. In the case of a timer interrupt that arrives every 1ms, if a critical section lasts more than 1ms, a timer interrupt could be lost!

2. Disabling the preemption (system-wide).
   This strategy will have some problems, because all the tasks will suffer from this suspention of the preemption even though they do not use shared resources.

3. Selectively disabling the preemption (using semaphores and mutual exclusion).
   This strategy will disable preemption only for the task that operates on the shared resource.

Hence, we should try to disable preemption rather than disabling interrupts.

Still the big issue with selectively disabling preemption the priority mechanism is no longer enforced: during the critical section might force the process to execute low priority tasks because preemption is disabled. This phenomenon is known as **Priority Inversion**.

Priority Inversion

If Priority inversion is not correctly managed it may lead to a complete violation of all timing contraints.

4.1.2 Interacting Tasks

Until now, we have considered only independent tasks, which are characterized by the fact that a job never blocks or suspends and a task only blocks on job termination.
In the real world, jobs might block for various reasons:

- Tasks exchange data throught shared memory (mutual exclusion)
- A task might need to synchronize with other tasks while waiting for some data
- A job might need a hardware resource which is currently not available.

Example 7: Control Application

Let us consider a control application composed by three periodic tasks:

- τ_1 reads the data from the sensors and applies a filter. The results are stored in memory.
- τ_2 reads the filtered data and computes some control law (updating the state and the outputs); both the state and the outputs are stored in memory

- τ_3 reads the outputs and writes on an actuator

All of the three tasks access data in shared memory. This means that there are conflicts on accessing this data concurrently with the risk that the data structures become inconsistent.

4.1.3 Priority Inversion Phenomenon

The rest of this chapter presents the following resource access protocols:

- Non-Preemptive Protocol (NPP)
- Highest Locking Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)

4.2 Non Preemptive Protocol (NPP)

A simple solution that avoids the unbounded priority inversion problem is to disallow preemption during the execution of any critical section. This method, also referred to as **Non-Preemptive Protocol (NPP)**, can be implemented by raising the priority of a task to the highest priority level whenever it enters a shared resource. In particular, as soon as a task τ_i enters a resource R_k, its synamic priority is raised to the level:

Non-Preemptive Protocol (NPP)

$$p_i(R_k) = \max_h \{p_h\}$$

The dynamic priority is then reset to the nominal value p_i when the task exits the critical section. This method solves the priority inversion phenomenon, however, is only appriopriate when tasks use short critical sections because it creates unnecessary blocking.

4.3 Highest Locking Priority (HLP)

The **Highest Locking Priority (HLP)** protocol improves NPP by raising the priority of a task that enters a resource R_k to the highest priority among the tasks sharing that resource. In particular as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the level

Highest Locking Priority (HLP)

$$p_i(R_k) = \max_h \{p_i | \tau_h \text{ uses } R_k\} \tag{4.1}$$

The dynamic priority is then reset to the nominal value p_i when the task exits the critical section. The online computation of the priority level in equation 4.1 can be simplified by assigning each resource R_k a **priority ceiling** $C(R_k)$ (computed offline) equal to the maximum priority of the tasks sharing R_k; that is:

priority ceiling

$$C(R_k) = \max_h \{p_i | \tau_h \text{ uses } R_k\}$$

Then, as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the ceiling of the resource. For this reason, this protocol is also referred to as **Immediate Priority Ceiling**.

Immediate Priority Ceiling

4.4 Priority Inheritance Protocol (PIP)

The **Priority Inheritance Protocol (PIP)** proposed by Sha, Rajkumar and Lehoczky, avoids unbounded priority inversion by modifying the priority of those tasks that cause blocking. In particular, when a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks. This prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by the higher-priority tasks.

Priority Inheritance Protocol (PIP)

The Priority Inheritance Protocol can be defined as follow:

- Tasks are scheduled based on their active priorities. Tasks with the same priority are executed in a First Come First Served discipline.
- When task τ_i tries to enter a critical section and resource R_k is already held by a lower-priority task τ_j, then τ_i is blocked. τ_i is said to be blocked by the task τ_j that holds the resource. Otherwise, τ_i enters the critical section.
- When a task τ_i is blocked, it transmits its active priority to the task τ_j that holds the semaphore/mutex. Hence, τ_j resumes adn executes the rest of its critical section with a priority $p_j = p_i$. Task τ_j is said to inherit the priority of τ_i. In general, a task inherits the highest priority of the tasks it blocks. That is, at every instant,

$$p_j(R_k) = \max\{P_j, \max_h\{P_h | \tau_h \text{ is blocked on } R_k\}\} \tag{4.2}$$

- When τ_j exits a critical section, it unlocks the mutex/semaphore, and the highest-priority task blocked, if any, is awakened. Moreover, the avtive priority of τ_j is updated as follows: if no other tasks are blocked by τ_j, p_j is set to its nominal priority P_j; otherise it is set to the highest priority of the tasks blocked by τ_j, according to equation 4.2.
- Priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2

A high priority task can experience two kinds of blocking: **Direct blocking** and **Push-through blocking**.

Direct blocking

Push-through blocking

Definition 25: Direct blocking

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

Definition 26: Push-through blocking

It occurs when a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks. Push-through blocking is necessary to avoid unbounder priority inversion

Although the Priority Inheritance Protocol bounds the priority inversion phenomenon, the blocking duration for a task can still be substantial because a chain of blocking can be formed. Another problem is that protocol does not prevent deadlocks (however, the latter problem can be solved by imposing a total ordering on the mutex accesses).

4.5 Priority Ceiling Protocol (PCP)

The **Priority Ceiling Protocol (PCP)** was introduced by Sha, Rajkumar, and Lehoczky to bound the priority inversion phenomenon and prevent the formation of deadlocks and chained blocking.

Priority Ceiling Protocol (PCP)

The basic idead of this method is to extend the Priority Inheritance Protocol with a rule granting a lock request on a free mutex. To avoid multiple blocking, this rule does not allow a task to enter a critical seciton if there are locked mutexes that could block it. This means that once a task enters its first critical section, it can never be blocked by lower-priority tasks until its completion.

In order to realize this idea, each mutex is assigned a priority ceiling equal to the highest priority of the tasks that can lock it. Then, a task τ_i is allowed to enter a critical section only if its priority is higher than all priority ceiling of the mutexes currently locked by tasks other than τ_i.

4.5.1 Original Priority Ceiling Protocol (OPCP)

The **Original Priority Ceiling Protocol** can be defined as follows:

Original Priority Ceiling Protocol

4.5.2 Immediate Priority Ceiling Protocol (IPCP)

Operating System Structure

Chapter 5

The Kernel

Chapter 6

Timer and Clock Latency

Chapter 7

The Non Preemptable Section Latency

Bibliography