145071 - Real time operating systems and middleware

Marco Peressutti

230403

marco.peressutti@studenti.unitn.it

January 26, 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.

1

- 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).
  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**
  If you think at the Operating System as a Resource Manager, then it is something that takes care of many things:

  1. **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**
     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**

  4. **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. 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.

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**.

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 $f_{i,k} \leq d_{i,k}$.
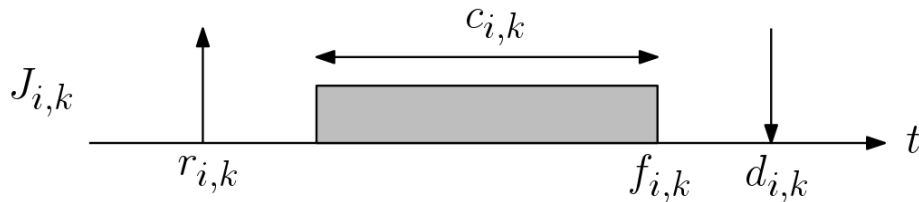

Figure 1.1: Graphical representation of Mathematical model of a Task

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

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:

```c
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:

```
1    void *SporadicTask(void *arg)
2    {
3        <initialization>;
4        while (condition)
5        {
6            <computation>;
7            <wait events>;
8        }
9    }
```

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

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

`Worst-Case Execution Time (WCET)`

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

`Worst-Case Response Time (WCRT)`

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 `hard real-time task` must be guaranteed before starting the task, i.e.

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

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** `Quality of Service (QoS)` **(QoS)**, but is not a catastrophic event, whereas a task is said to be a **soft real-time task** if it has soft deadlines.

In other terms, some deadlines can be missed without compromising the correctess of the system, `soft real-time task` 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 $N = \dfrac{D_i}{s}$. 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 $\left\lfloor \dfrac{t}{T_i} \right\rfloor$

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

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

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 Timeline Scheduling 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 Cyclic Executive Scheduling 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.

Cyclic Executing Scheduling is a **static scheduling algorithm** where **jobs are not preempt-** static scheduling algorithm **able** (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: scheduling table

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

The period timer fires every Minor Cycle Δ. hyperperiod

Hence the implementation of the scheduling algorithm performs as follow: Minor Cycle

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

3.1   Background Execution

3.2   Immediate Execution

3.3   Polling Servers

3.4   Deferrable Servers

3.5   Sporadic Servers

3.6   Constant Bandwidth Servers (CBS)

Chapter 4

Resource Access Protocols

4.1 Introduction

4.1.1 Atomicity

4.1.2 Interacting Tasks

4.1.3 Priority Inversion

4.2 Non Preemptive Protocol (NPP)

4.3 Highest Locking Protocol (HLP)

4.4 Priority Inheritance Protocol (PIP)

4.5 Priority Ceiling Protocol (PCP)

4.5.1 Original Priority Ceiling Protocol (OPCP)

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