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.

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

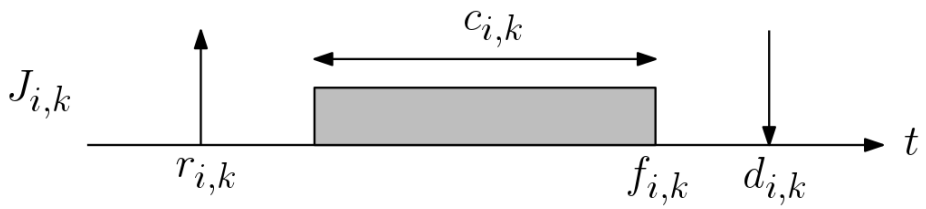

Figure 1.1: Graphical representation of Mathematical model of a Task

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

1.3.1 Worst-Case Response Time Analysis

1.3.2 Processor Demand Analysis

1.3.3 Processor Utilization Factor test

Real-Time Scheduling

Chapter 2

Periodic Task Scheduling

Chapter 3

Aperiodic Servers

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