145071 - Real time operating systems and middleware

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

February 6, 2022

Contents

Chapter 1

Introduction to the course

1.1 Overview of the course

- **Real time systems**
  About the topic we will cover:
  - 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.
- **POSIX API**
  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.
- **Real-Time Scheduling**
  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.
- **Operating System structure**
  Since it will be important to keep in check the latencies, we will cover:
  - 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, monolithic kernels and RT).
- **Real-Time Kernels and OSs**
- **Developing Real-Time applications**

Real time systems

Real-Time Computing

temporal constraints

Real Time Scheduling

POSIX API

Real-Time Scheduling

Operating System structure

Real-Time Kernels and OSs

Developing Real-Time applications

1.2 Real-Time Systems

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

- Real-Time Operating Systems (RTOS) is an operating system providing support to Real Time applications
- Real-Time Application the correctness of the application does not only depends on the output values/results that the application produces, but also on the time when such values are delivered
- Operating System an operating systems can be looked at from many different perspective:

  1. 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.
  2. 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.
  3. Control the execution of application programs.
  4. Manage the hardware and software resources.

Real-Time Operating Systems (RTOS)

Real-Time Application

Operating System (OS)

interrupt

1.2.1 Real-Time Operating Systems

Since the OS is something that lies in-between the user application and the hardware resources we can summarize the aforementioned interpretation of the OS, as:

- Service Provider for user programs (i.e. exports a programming interface).
  This concept looks at the OS from the perspective of the software application.
- Resource Manager (i.e. implements schedulers)
  This concept looks at the OS from the perspective of the hardware.

Service Provider

Resource Manager

OS as a Service Provider

One way of looking at the OS is as a Service Provider, in the sense that it provides:

- 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.

OS as a 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.

Process Management

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.

Memory Management

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.

1.2.2 Real-Time Concepts and Definitions

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: 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, breaks, ...
4. Sens 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: it 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.

Examples of temporal constraints could be:

- must react to external events in a predictable time
- must repeat a given activity at a precise rate
- 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

Real-Time & Determinism

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.

Throughput vs Real-Time

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.

Processes, Threads and Tasks

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

- Algorithm logical procedure used to solve a problem

  Algorithm
- Program formal description of an algorithm, using a programming language
- Process Instance of a program (program in execution)

  Program
- 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.

  Process
- Task process or thread

  Thread

  Task

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.

1.3 Real-Time Tasks

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.

1.3.1 Mathematical Model of a Task

We can define a Real-Time task τ_i as a stream of jobs or instances $J_{i,k}$. In other terms, it is a sequence of activity that is activated periodically or aperiodically.

Real-Time task

Each job $J_{i,k} = (r_{i,k}, c_{i,k}, d_{i,k})$, that is characterized by a tuple of four values:

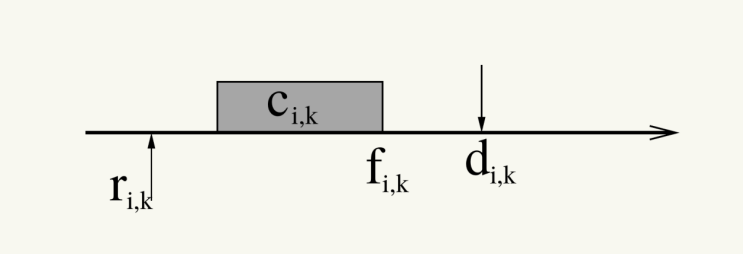
- $r_{i,k}$ instant when the job is activated
- $c_{i,k}$ computation time
- $d_{i,k}$ absolute deadline
- $f_{i,k}$ finishing time (of the computation), which has to be smaller than the absolute deadline

computation time

absolute deadline

finishing time

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.



Formally: A job is an abstraction used to associate deadlines (temporal constraints) to activities

- $r_{i,k}$ time when job $J_{i,k}$ is activated (by an external event, a timer, an explicit activation, etc..)
- $c_{i,k}$ computation time needed by job $J_{i,k}$ to complete
- $d_{i,k}$ absolute time instant by which job $J_{i,k}$ must complete
  job $J_{i,k}$ respects its deadline if $f_{i,k} \leq d_{i,k}$
- Response time of job $J_{i,k}$

Response time

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

1.3.2 Periodic Tasks

A Periodic Task is uniquely represented by a tuple of three values in the form:

Periodic Task

$$\tau_i = (C_i, D_i, T_i)$$

and it represents 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 is the task period

period

- D_i is the task relative deadline
- C_i is the task worst-case execution time (WCET)
- R_i is the task worst-case response time (WCRT) defined as

relative
worst-case response time (WCRT)
(WCET)

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

For the task to be correctly scheduled, it must hold that:

$$R_i \leq D_i$$

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

cycle

1. it is activated periodically with a period of T_i
2. it executes a computation
3. 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>;
  }
}
```

Tasks are graphically represented by using a scheduling diagram. For instance, figure 1.1 shows a schedule of a periodic task

scheduling diagram

$$\tau_1 = (3, 6, 8)$$
$$WCET_1 = 3 \quad D_1 = 6 \quad T_1 = 8$$

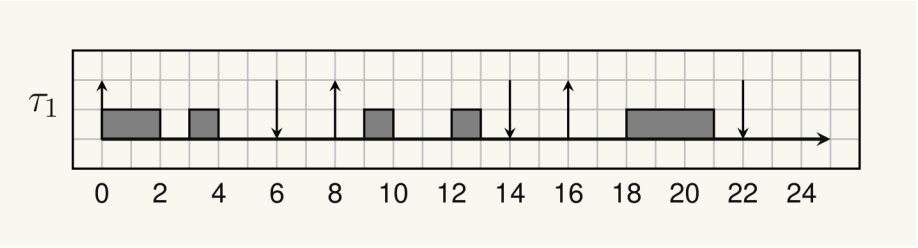


Figure 1.1: Periodic task graphical representation

Notice that, while job $J_{1,1}$ and $J_{1,3}$ execute for 3 units of time (WCET), job $J_{1,2}$ executes for only 2 units of time.

1.3.3 Aperiodic Tasks

Aperiodic Tasks are not characterised by periodic arrivals, meaning that

Aperiodic Task

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

However, aperiodic tasks are of fundamental importance given that they model:

1. Tasks responding to events that occur rarely (e.g. a mode change)
2. Tasks responding to events with irregular structure (e.g. bursts of packets from the network, ...)

A possible pattern for an aperiodic task τ_1 is proposed in figure 1.2

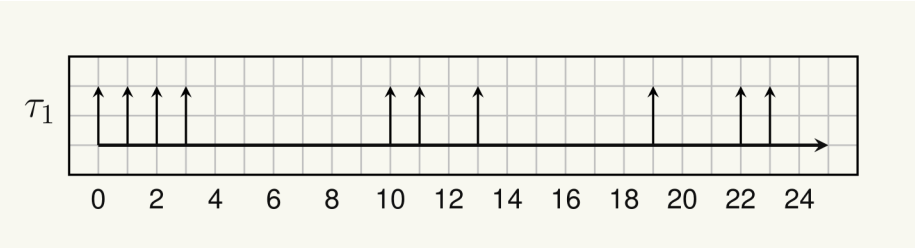


Figure 1.2: Aperiodic task graphical representation

notive that arrivals might be bursty, and there is not a minimum time between them

1.3.4 Sporadic Tasks

Sporadic Tasks are aperiodic tasks characterised by a minimum interarrival time between jobs. In this sense, they are similar to periodic 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)

`Sporadic Task`

`external event`

Hence, its fundamental implementation can be represented as:

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

Given its similarity with periodic task, a sporadic task can be represented by a tuple of three values

$$\tau_i = (C_i, D_i, T_i)$$

and it represents a stream of jobs $J_{i,k}$ with

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

where

- T_i is the task minimumm interarrival time (MIT)
- D_i is the task relative deadline
- C_i is the task worst-case execution time (WCET)

`minimumm interarrival time (MIT)`

The task is correctly scheduled if $R_i \leq D_i$

Since sporadic task can be represented by a mathematical model, they can be graphically displayed. Hence, figure 1.3 shows a possible schedule of a sporadic task $\tau_1 = (2, 5, 9)$.

Notice that

$$\begin{aligned} r_{1,2} = 12 > r_{1,1} + T_1 = 9 \\ r_{1,3} = 21 = r_{1,2} + T_1 = 21 \end{aligned}$$

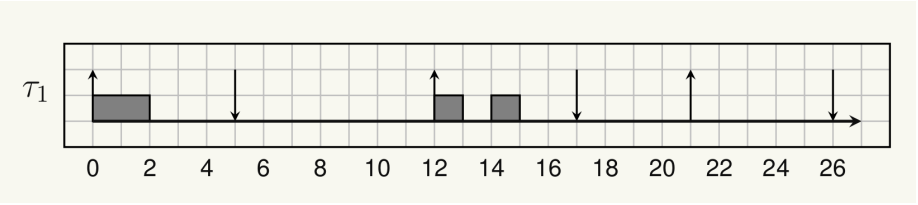


Figure 1.3: Sporadic task graphical representation

1.4 Task Criticality

A deadline is said to be hard if 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

hard real-time task

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

Example:

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 catastrophuc event, whereas a task is said to be a soft real-time task if it has soft deadlines.

soft

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

Exmplaes 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

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

- ...

A common example of implementation of a soft real-time task are the Audio and Video players. Particularly, 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.

skipped frames

In fact missing a lot of frames by 5 ms can be better than missing only a few frames by 40 ms.

Another common example, can be found in some robotic systems where 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
- ...

Chapter 2

Real-Time scheduling and analysis

In the previous chapter the reader was introduced to some fundamental notions and concepts that will be extensively used throughout the course.

Since, in this chapter, we will focus on the scheduling of real-time tasks and the analysis of each scheduling policy, it is worth mentioning some basic definitions:

- Algorithm logical procedure used to solve a problem
- Program formal description of an algorithm, using a programming language
- Process Instance of a program (program in execution)
  - Program: static entity — static entity
  - Process: dynamic entity — dynamic entity
- the term task is used to indicate a schedulable entity (either a process or a thread), in particular;
  - A thread reresents 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, etc...

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 — OS kernel
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 — Concurrency
CPU. Tasks are alternated on a real CPU and the task scheduler decides which task executes at — multiplexing
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 resources at that time. — task scheduler

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

2.1 Real-Time Scheduling

One of the key components of an OS and a OS kernel is the scheduler. A scheduler generates a schedule from a set of tasks. From a mathematical perspective it is a function:

first consider UP systems (given that it has a simpler definition). A schedule $\sigma(t)$ is a function mapping (every point in) time t into an executing task

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

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

taskset

idle task

At the end of the day what the processor/OS kernel does is to implement a function like the scheduler. Clearly, if you have more that one processor, the function under consideration is multivalued, because at every point in time it decide the allocation of the different processors to different tasks.
For an SMP system (i.e. m CPUs), $\sigma(t)$ can be extended to map t in vectors $\tau \in (\mathcal{T} \cup \tau_{idle})^m$.

Therefore, formally, the scheduler implements $\sigma(t)$ and as a consequence the scheduler is responsible for selecting the task to execute at time t. However, a function is a denotational way of describing an algorithm, i.e. we specify mathematically what the function is. The operational way of describing an algorithm on the other hand, is a squence of steps to be taken in order to implement the function.

denotational

operational

Hence, from an operational/algorithmic point of view:

- Scheduling algorithm is an 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 $\sigma_{\mathcal{A}}(t)$

Scheduling algorithm

In the frame of reference of real-time systems we are interested in finding conditions on the scheduling choice that allow us to meet all the deadlines. Hence, a task set is schedulable by an algorithm $\mathcal{A}$, if by applying that scheduling algorithm, σ_A does not contain missed deadlines.

On this note, the Schedulability test is a way to decide if, given a task set $\mathcal{T}$ and an algorithm $\mathcal{A}$, that task set is going to be schedulable.

Schedulability test

The key point here is that given the fact that by making a proper choice of scheduling algorithm, one can obtain a schedulable task set, how can I find a way to schedule the activities so that all the deadlines are met? And this is the problem of real time scheduling.

2.2 Cyclic Executive Scheduling

One way to find a solution of the schedulability problem is to consider a task set of fixed tasks (i.e. you will always have those tasks activated in the very same way). This is the case of legacy applications and where reliability is fundamental, e.g. military and avionics systems (Air traffic control, Space Shuttle, Boeing 777).

In this scenario, one can obtain a paper and pencil solution, i.e. trying to decide how to schedule things in such a way that the task set is schedulable and stick to that solution.

Of course, there is a more systematic way of solving such problem and such process is called Cyclic Executive Scheduling.

Cyclic Executive Scheduling

The Cyclic Executive Scheduling (aka timelice scheduling or cyclic scheduling) is a scheduling algorithm with very low overhead (in the sense that scheduling decisions are taken-offline) and with a very simple and well tested idea, that was originally used in avionic systems to schedule periodic tasks.

timelice scheduling

cyclic scheduling

2.2.1 The Idea

Cyclic Executive Scheduling is a

- static scheduling algorithm (i.e. it is decided offline and it is always applied repeatedly in the same way).
  As a consequence, all scheduling decision are taken upfront; it is not something that defers decision to the runtime.
- Jobs are assumed not to be preemptable, i.e. a scheduled job executes until termination, it retains control of the CPU for the entire computation time.

static scheduling algorithm

The procedure to follow in order to apply this scheduling algorithm is:

1. Split the time axis into slots, each of which is statically allocated to the tasks (scheduling table).

scheduling table

2. Since we are dealing with periodic tasks, the same sequence of tasks will repeat periodically, hence once the scheduling table is completed, it starts from the beginning and repeats the same sequence over and over again.

A periodic timer activates execution (allocation of a slot).
In summary three components are needed to uniquely define this scheduling algorithm:

1. Major Cycle: least common multiple (lcm) of all the tasks' periods (aka hyperperiod)
2. Minor Cycle: greatest common divison (gcd) of all the tasks' periods (i.e. granularity at which the scheduling decisions will repeat)
3. A timer: that fires every Minor Cycle (Δ)

allocation of a slot
Major Cycle
hyperperiod
Minor Cycle
A timer

2.2.2 Implementation

As previously mentioned from an execution stand point this scheduling algorithm is extremely simple because every minor cycle:

1. A periodic timer is initialized
2. Every time the timer fires the scheduler read the scheduling table
3. The scheduler calls the approriate function(s) associated to the corresponding task(s)
4. The scheduler returns to sleep until the next minor cycle (i.e. the timer fires)

2.2.3 Advantages

From it's radical simplicity, allows to create a total check and a certification of the code that can be shipped to a space shuttle or any other avionic application and execute theoretically forever.

An OS is much more difficult to certify, because asynchronious events can be generated and it is impossible to consider upfront every possible course of events that can happen. On the contrary, once you solved the cyclic executing scheduling problem, you have a global solution that is guaranteed to be replicated.

Furthermore, with its simple implementation there is no need for a real-time operating system, no real task exist (just function calls), and only one single stack is necessary for all the *tasks* (since each task runs until completion, there will never be a situation in which the execution of the function are active at the same time).

Since it involves a non-preemptable scheduling there is no critical races (i.e. there will never be two tasks that shared a variable) and therefore there is no need to protect data (i.e. no need for semaphores, pipes, mutexes, mailboxes, etc...).

non-preemptable scheduling

Lastly, since there is no need for an OS, all the CPU/execution time is taken by the tasks, hence It has low run-time overhead, and the jitter (delay in which the result is completed, i.e. difference between when a task start and it finishes) can be explicitly controlled.

jitter

2.2.4 Drawbacks

However, this scheduling algorithm has some important drawbacks:

- it is not robust w.r.t. overloads. This type of solution assumes that all the tasks do not give back control until they are finished (until its allocated slots terminates). However, if the task execute for much more of its WCET or the allocated time slot, what happens is that the schedule gets disrupted with a subsequent domino effect.
- It is difficult to expand the schedule (static schedule): introducing a new task requires that the whole system/schedule must be redesigned.
- Not easy to handle aperiodic/sporadic tasks
- All task periods must be a multiple of the minor cycle time
- Difficult to incorporate processes with long periods (big tables)
- Variable computation time imply that it might be necessary to split tasks into a fixed number of fixed size procedures. This requires to actively change the code into smaller parts, and, in the case of third party library/code, this is not always allowed.

2.3 Fixed Priority Scheduling

In general, we want to have more flexibility than the cyclic executive scheduling (in fact it is applied on very niche and particular situations).

What can be done therefore is utilize preemptive scheduling algorithms, i.e. algorithm that are allowed to suspend the execution of a task and resume it later.

preemptive scheduling algorithm

Therefore a task has, two states: ready and executing, and the OS is allowed to enforce a state change.

ready

executing

In the class of preemptive algorithm we will look at different algorithms, but the simplest is the Fixed Priority Scheduling. In this scheduling algorithm:

Fixed Priority Scheduling

- Every task τ_i, when it is created, is assigned a integer number that is encoding a fixed priority p_i
- The active task with the highest priority is scheduled (POSIX convention)

Priorities are integer numbers: the higher the number, the higher the priority (In the research literature, sometimes authors use the opposite convention).

And so, the work of the scheduler at any point in time is to look at the ready queue and pick up the task with the highest priority.

Observations:

- The response time of the task with the highest priority depends only on its computation time, and therefore it is always less than or equal to its WCET.
- The response time of tasks with lower priority depends on its own computation time and the interference of higher priority tasks.

interference

- Clearly, the scheduling priorities can change how the things behave: are we sure that choosing a different priority assignment policy we would obtain a schedulable task set? Yes.
  So, we have two issues: the scheduling analysis (given a task set and the priorities, is everything schedulable) and syntesis problem (given the task set and computation time, find the assignment of priorities that guarantees schedulability).

scheduling analysis

syntesis problem

2.3.1 Priority Assignment

Given a task set how to assign priorities?
Depends on the different objectives that you would like to achieve:

- 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)

2.3.2 Optimal Priority Assignment

A priority assignment is an optimal priority assignment (Opt):

- If the task set is schedulable with another priority assignment, then it is schedulable with priority assignment Opt (i.e. the optimal priority assignment cannot do any worse that any other feasible assignment).
- If the task set is not schedulable with Opt, then it is not schedulable by any other assignment. (If the optimal choice fails then there is no other possible solutions/other alternatives will fail as well).

Formally, given a periodic task set $\mathcal{T}$ (all tasks are activated periodically) with all tasks having relative deadline D_i equal to the period T_i ($\forall i, D_i = T_i$), and with all offsets equal to 0 ($\forall i, r_{i,0} = 0$):

- The best assignment, in this case, is the Rate Monotonic (RM) assignment

Rate Monotonic (RM)

- Shorter period imply higher priority (hence the name rate monotonic: the priority is monotonic w.r.t. the rate. The higher the frequency/rate, the shorter the period, the higher the priority).

Given a periodic task set with deadline different from periods and with all offsets equal to 0 ($\forall i, r_{i,0} = 0$)

- The best assignment is the Deadline Monotonic (DM) assignment
- Shorter relative deadline imply higher priority (the shorter the relative deadline, the higher the priority).

Deadline Monotonic (DM)

Note that the Deadline Monotonic assignment refers to relative deadline (interval in time, fixed value) not absolute deadline (instant in time, change over time).

For sporadic tasks, the same rules are valid as for periodic tasks with offsets equal to and by considering maximum activation rate.

2.4 Analysis

The best possible assignment for fixed prority is the Rate Monotonic and Deadline Monotonic assignement.
But how can I be sure that even using the optimal priority assignment, the schedule will meet all the deadlines?

This is the problem of Real-Time analysis: an activity that given a task set and a scheduling policy, it responds to the query on whether or not the task set is schedulable.

Real-Time analysis

What we have done so far is something very naive, i.e. simulating the schedule and verify that there is no deadline miss, but does not scale well in the case of long observation time of the system, until the schedule repeats itself.
Luckly there is a point/tme horizon, such that if no deadline miss happens withing this time horizon, you can certify none will occur ever after. How long is this time horizon depends:

- If there is no offset, we will find ourselves in the critical case, because if we have simultaneous activation of all the tasks in the task set, the system have an immediate request of computation time that needs to be satisfied.
  In this case, it is sufficient to simulate the schedule until the hyperperiod, i.e. the least common multiple of the periods

  hyperperiod

  $$H = lcm\{T_i\}$$

  This is because after the least common multiple of the period the task activation will be repeated.
- If there is an offset, the schedule diagram can be shifted to meet the condition of the previous point.
  Anyhow, given an offset

  $$\phi_i = r_{i,}$$

  it is necessary to simulate the schedule for at least

  $$2\,H + \phi_{max}$$

  before being sure that the schedule will repeat itself.
- If tasks periods are prime numbers the hyperperiod can be very large

Fortunately, there are better tests, that we can find, called Utilisation Bound Analysis.

Utilisation Bound Analysis

2.4.1 Utilisation-Based Analysis

The Utilisation is the fraction of the processor that a task needs for its execution.

Hence, if a task have an activation time of 5 and a computation time of 1, then its utilization is around $^1/_5$=20%.

So it would be nice to have a test that tells if a scheduling algorithm produces as feasible schedule simply based, on the utilisation bound.

A sufficient test is based on the Utilisation bound:

Utilisation

The utilisation least upper bound for 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} ($U \leq U_{lub}$), then the task set is schedulable by algorithm $\mathcal{A}$

utilisation least upper bound

Formally:

- Each task uses the processor for a fraction of time, called worst case utilisation

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

- The total processor utilisation is

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

  and it is a measure of the processor's load

One thing that we can immediately say is that if the total utiilisation exceed 100 % there is no way that the task set is schedulable, i.e.

$$U > 1 \qquad \text{taskset is not schedulable}$$

If the sum of all the utilisation is greater than 1/100% we encounter a situation called stochastic instability, meaning that in the average you are asking the system more than it can offer.

In this case we have something similar since we have the worst case utilisation. In case the system requires more that it is available, there is no way the taskset is schedulable. Hence, this analysis yield a worst case guarantee.

Moreover, if $U < U_{lub}$ the taskset is schedulable. However:

- there is a "gray area" between U_{lub} and 1, it depends on the specific performance of the algorithm under consideration.
  So the ability of an algorithm to achieve a good utilisation bound is a measure of how efficient and effective the algorithm is.
- We would like to have U_{lub} close to 1.
- $U_{lub} = 1$ would be optimal (hard upper bound, after 1 we are sure that the task set is not schedulable), it is physical impossible to outperform 100% utilisation.

Utilisation Bound for RM

The notion of least upper bound can be described considering 2 tasks: one can make many choices of periods and computation times of the two tasks, taking the minimum value of U below which all the task sets that I can choose are schedulable.
It can happen that some task that exceed the U_{lub} is schedulable, but following the notion of least upper bound, as far as the task set is below this value the task set is guaranteed to be schedulable.

Formally: consider n periodic (or sporadic) tasks with relative deadline equal to periods. The least upper bound for Rate Monotonic assignment can be obtained via the following expression:

$$U_{lub} = n(2^{1/n} - 1)$$

Note:

- U_{lub} is a decreasing function of n
- For large n: $U_{lub} \approx 0.69$

Therefore the schedulability test consists in:

1. Computing the utilisation

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

n	U_{lub}	n	U_{lub}
2	0.828	7	0.728
3	0.779	8	0.724
4	0.756	9	0.720
5	0.743	10	0.717
6	0.734	11	...

2. If $U \leq U_{lub}$, the task set is schedulable
3. If $U > 1$ the task set is not schedulable
4. If $U_{lub} < U \leq 1$, the task set may or may not be schedulable

Utilisation Bound for DM

In case of deadlines different from periods, one can always consider a conservative approximation in which the period is equal to the deadline.

If relative deadlines are less than or equal to periods, instead of considering

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

we can consider:

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

Then the test is the same as the one for RM, except that we must use U', instead of U.

And consequently:

$$\tau = (C, D, T) \rightarrow \tau' = (C, D, D)$$

If I can guarantee that this conservative approximation is schedulable than I can guarantee that the original task set is schedulable as well.

In summary:

- τ' is a *worst case* for τ
- If τ' can be guaranteed, τ can be guaranteed too

Pessimism

The previously described condition is an extremely conservative approximation and most often than not the test would yield a non-schedulable result (above the least upper bound, even though the task set is schedulable).
The bound is very pessimistic: most of the times, a task set with $U > U_{lub}$ is schedulable by RM.

A particular case is when tasks have periods that are harmonic: 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. In this case, and only in this case, the least upper bound of RM grows to1 (i.e. For a harmonic task set, the utilization bound is $U_{lub} = 1$). harmonic

In other words, Rate Monotonic is an optimal algorithm for harmonic task sets.

2.4.2 Response Time Analysis

So far with the utilisation bound we have seen that there is a sufficient schedulability condition: if the total utilisation is below the least upper bound than the system is schedulable. But this test works only if one makes very specific choices, i.e. to have the priority set using the RM assignment, otherwise the least upper bound does not apply.
Moreover, in the case in which the total utilisation is above the least upper bound, no conclusion can be drawn/no guarantees.

Therefore, we would like to utilize a necessary and sufficient test, which won't be analytical and necessary and sufficient

nice as we have seen from an algorithmic point of view, but it will be a little more reliable since it requires to make much less assumptions.

Before introducing this test, let us recall the concept of response time: difference between the activation and the finishing/completion time. If one is able to compute the worst-case response time and if one can verify that such quantity remains below the relative deadline, then the system is schedulable.

worst-case response time

relative deadline

The best thing about this test is that there is no assumption on the priority assignment, hence it can be applied to all algorithms. However, the periodic or sporadic tasks that we will be consider, are assumed to have no offsets (even though it is possible to have a variant of this analysis that applies).

Formally:

- For every task τ_i in the task set
- Compute the worst case response time R_i for τ_i, i.e.

$$R_i = \max_j \{\rho_{i,j}\} \quad \text{with: } \rho_{i,j} = f_{i,j} - r_{i,j}$$

- If $R_i \leq D_i$, then the task is schedulable
- Otherwise, the task is not schedulable

Consider a number of task set ordered by decreasing priority (i.e. $i < j \rightarrow p_i > p_j$) and consider no assumptions about the tasks offset (i.e. tasks can be activated at any time). We can define the Critical Instant as the most critical condition for the computation of the response time (if you allow the offset to change).

Critical Instant

The obvious result is that if you allow the offset to change, the worst case situation in terms of response time (i.e. situation that produces the longest response time) is one in which all tasks start at the same time. Because if all the tasks start at the same time one would observe an utilisation peak.

Consider:

- a number of task set ordered by decreasing priority (i.e. $i < j \rightarrow p_i > p_j$)
- no assumptions about the tasks offset (i.e. tasks can be activated at any time)
- the worst possible offsets combination

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

critical instant

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

Theorem 1. *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.*

Hence if all the offsets are 0, the first job of every task is released at the critical instant.

How one can compute the worst case response time? The worst case response time R_i for task τ_i depends on:

- Its execution time
- The execution time of higher priority tasks (since the higher priority tasks can preempt task τ_i and increase its response time), aka scheduling interference.
  This factor is given by the number of activations of the higher priority tasks, in the window from the critical instant to the response time times the execution time/computation time of the higher priority tasks.

preempt

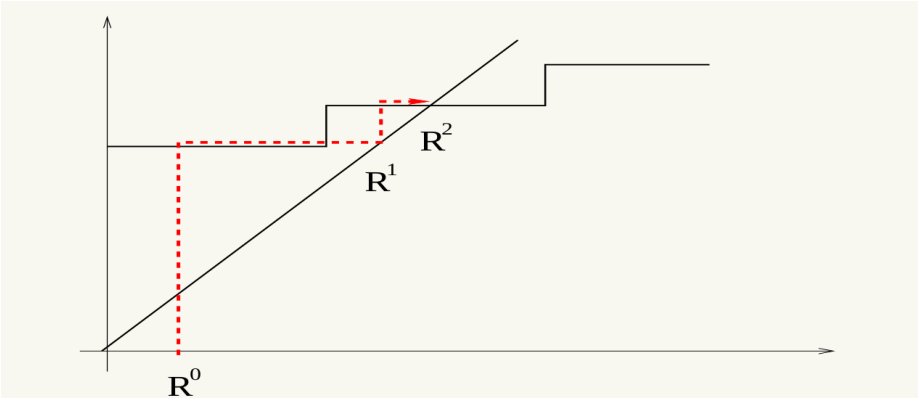
scheduling interference

as a consequence, the worst case response time can be computed as:

$$R_i = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{T_h} \right\rceil C_h$$

Unfortunately, one can notice that the above expression in recursive/implicit equation + nonlinear

implicit equation



given the presence of the ceiling operator (i.e. $R_i = f(R_i)$), as a consequence, there is no closed-form expression for computing the worst case response time R_i.

We need an iterative method to solve the equation.

The point where the bisector intersects the stairway, is the solution to the equation. But how one can find such intersection? bisector

In terms of algorithms it means that, one have a first guess which is

- Iterative solution:
$$R_i = \lim_{k \to \infty} R_i^{(k)}$$
where $R_i^{(k)}$ is the worst case response time for τ_i at step k
- Hence, we can start from a first estimation of the response time:
$$R_i^{(0)} = C_i$$
- Compute the interference in this interval
$$R_i^{(1)} = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(0)}}{T_h} \right\rceil C_h$$
Since all the tasks starts at the same instant (critical instant), I am sure that I would have at least one activation, hence
$$R_i^{(0)} = C_i + \sum_{h=1}^{i-1} C_h$$
- Repeat the iterative step:
$$R_i^{(1)} = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i^{(k-1)}}{T_h} \right\rceil C_h$$
- The iteration stops when:
  - $R_i^{(k+1)} = R_i^{(k)}$
  - $R_i^{(k)} > D_i$ (non schedulable)

This is a standard method to solve non-linear equations in an iterative way. If a solution exists (the system is not overloaded, i.e. total utilisation of the task is below 1), $R_i^{(k)}$ converges to it, otherwise the failing condition avoids infinte iterations.

Considerations

1. The response time analysis is an efficient algorithm
2. In the worst case the number of steps N for the algorithm to converge is exponential

3. depends on the total number of jobs of higher priority tasks in the interval $[0, D_i]$

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

4. If s is the minimum granularity of the time, then in the worst case

$$N = \frac{D_i}{s}$$

5. However, such worst case is very rare: usually, the number of steps is low.

Bibliography