REAL-TIME OPERATING SYSTEMS

CAPITOLO 1 INTRODUCTION

For real-time operating systems (RTOS) the correctness depends not only on the validity of result, but also on temporal factors. Robots typically have sensors, sonars, used to detected obstacles. Obviously, you need to know with accuracy the frequency recorded, to avoid obstacles successfully. You need to record data periodically, also to make prediction.

The architectures to control systems are mostly composed by multiple computer units. In this course we'll consider just a computer unit that does tasks and periodically asks for data.

How was handled this problem in the past? With timers that helped going through the problem related to the handling of temporal events. Also, the optimization of the code was wanted. Maybe you make the system efficient, but there's a problem: at the end you get a code so optimized that it can be difficult to modify it the future.

How can you know in advanced that the system will behave as expected? It's not just to make it work. ROS is a popular framework for programming robots, it's not s real operating system but provides functionalities to interact with the robot itself. ROS1 was not real time, didn't implemented such functionalities, while ROS2 is. This is what we're going to do: showing how an OS might be designed to be real-time. The **key point** is that the main characteristic of RTOS isn't speed, but **PREDICTABILITY**. Concerning real-time execution of tasks we'll proposing scheduling algorithms for RT, but also if we are able to guarantee that all time constrains are satisfied or not.

DEF REAL-TIME TASK: it is a task that must terminate with a **deadline**. Sometimes we represent tasks with a timeline which

INSERIRE FOTO

The task J1 must finish before the deadline d1.

RT processes are classified as:

- **Hard**: if you finish the execution after the deadline, it may cause catastrophic effects (Sensor acquisition, sensors for motor planning, driving actuators).
- **Soft**: missing the deadline doesn't compromise the system (Streaming video, collecting input from a keyboard, visualization of messages on a monitor).

Hard means you should never miss the deadline. The situation is that we want to have RTOS, we want that the system itself is not only able to accomplish tasks, but also what we saw previously. Many systems not designed to be RT have been upgraded to be, e.g., Linux with RT PREEMPT (it is not the official distro but some part of it have become).

Summarizing: what are the main functionalities of a OS?

- **Schedulability analysis**: the possibility to know a priori, before the execution starts, if the task can be scheduled respecting every deadline.
- **Optimal scheduling** of periodic tasks.
- **Handle sporadic and aperiodic tasks**: tasks executed when something particular happens.
- **Interprocess communication**: task need to communicate with each other, throughout semaphores as instance. What happens if a task blocks a task with deadline? Schedulability analysis must underline this problem to solve it.
- **Worst case execution time analysis**: task have a time of execution, but it doesn't automatically correspond to the time in th worst case scenario, that must be known.

What we know about standard OS also works on RTOS.

The most important parameter is the **jitter**: the deviation from true periodicity of a presumably periodic signal.

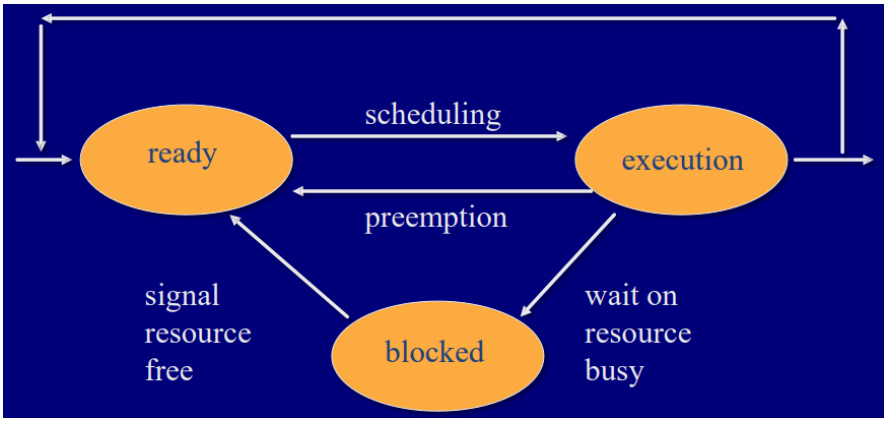
There are four mechanism that increase performance but decrease predictability:

- Interrupts: e.g. I need to get data from a keyboard, the OS can poll the port, check periodically tha port, but this is a waste of time. Interrupts were introduced to avoid polling the port: when there are new data, a signal is sent and a handler (function) is asynchronously executed. This mechanism isn't good for RTOS because typically interrupts have high priority that stops other tasks that were executed (maybe with a strict deadline)
- Memory management: mechanism to ensure that every task in the system think memory is fully available and accessible. With RTOS this may cause problems because somethimg needed may not be on the cache, so the task will take longer. This may be a worst-case scenario that must be known in advance and handled.

These mechanisms are good for standard OS, that need to improve behaviour on average. On RTOS we're reasoning in terms of **PREDICTABILITY**.

CAPITOLO 2 SCHEDULING

A task Ji in a sequence of instruction, in absence of activities, is executed continuously till the end. A task that can be executed is active, and active tasks are waiting for the processor to be ready. A task using the processor is in execution. A task can be blocked if it's waiting for resources.



Pre-emption means that a task is sent back to the execution queue, because it exceeded its temporal quantum.

NON-REAL-TIME SCHEDULING

ROUND ROBIN

ROUND ROBIN algorithm chooses a task at the head of the queue and enables a timer that last a certain quantum. After that the task in enqueued again. This give the impression that tasks are executed in parallel when they're executed concurrently. This algorithm is non-real-time.

The set of rules that determine the choice of the task to be executed among the ready process is called scheduling algorithms. A **pre-emptive** system allows a task in execution to be interrupted. Of course, system can also be non-pre-emptive.

FIRST COME FIRST SERVED

This static algorithm can be used when, as instance, two task managed with Rate Monotonic have the same priority. Having to choose, I can implement either this scheduling or round robin to chose between them.

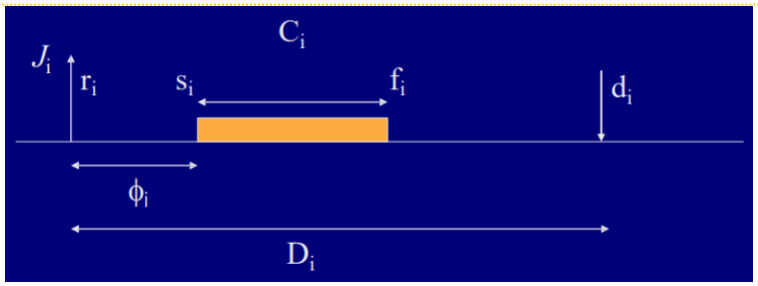
SHORTEST ELAPSED TIME FIRST

Scrivere qualcosaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

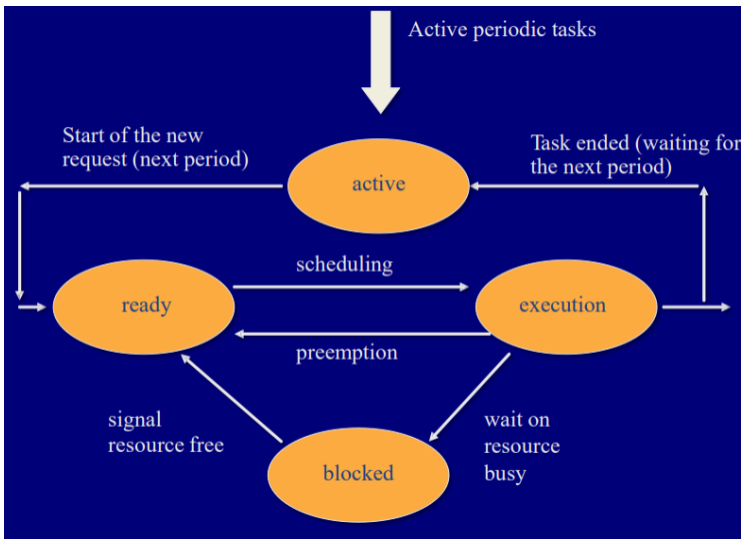
If I have two tasks executing concurrently, I need to use the unique program counter register of the processor for both tasks. In these terms, we talk about **context switching** as the need of switching between the tasks in execution.

TIME CONSTRAINTS

In RT systems we also have temporal constrains. Typically, we describe RT task as:



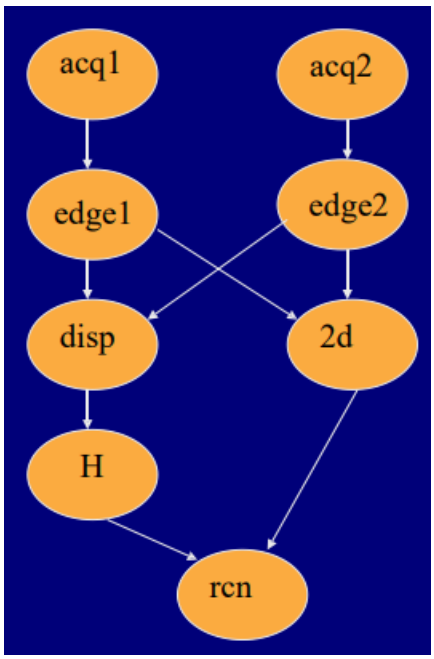
- *ri* stands for the **ready time**, the time at which a task is ready to be executed. It doesn't necessary corresponds to the execution time.
- *fi* it stand for the **finish time**. We're interested in it to calculate Ci.
- *Ci* is the **computational time**. Depends on the tasks and systems characteristics.
- *di* deadline
- *Di* **relative deadline**: deadline with respect to the arrival time *ri*.



Task can be defined as periodic or aperiodic. **Periodic task** should be interpreted as an infinite sequence of identical activities, each called an instance of the task. After a period T you have a new request for this task.

Thanks to these concepts, the scheme can be modified as follow:

OTHER CONSTRAINTS



There are other constrains in addition to time. Task can have **precedence** constraints. Precedence relationships are expressed through a Directed Acyclic Graph.

We say that A must precede B (A<B) if a path on the graph from A to B exists. We say that A is an immediate predecessor of B if it exists an edge directed from A to B.

A GENERAL PROBLEM

What is the real time scheduling problem?

❖ Given a set of n tasks $J^S = \{J_1,..,J_n\}$ of which the DAG and shared resources are known, the real-time scheduling problem consists in finding a feasible scheduling such that:

- All task instances ends within their deadlines;
- All precedence constraints are met;
- All mutually exclusive resources are correctly accessed.

In its general formulation, the problem is NP-hard: doesn't exist an algorithm with polynomial complexity able to find a feasible scheduling.

PERIODIC TASK SCHEDULING

To find easier scheduling's algorithms we focus on periodic tasks, assuming:

- The **period is constant** in time, for each task.
- **Execution time is constant** in time.
- The **relative deadline corresponds to the end of the period**. Before we had a deadline that could be before the end of the period. We want to be sure that every instance is executed before the period lasts.
- All **tasks are independent**, there aren't precedence constraints.

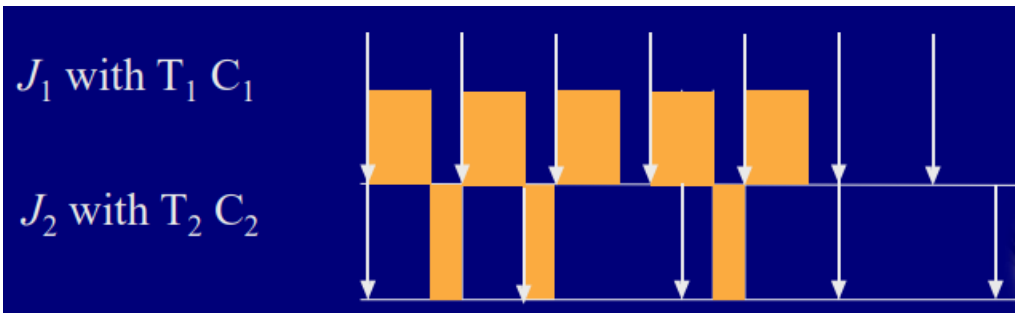
RATE MONOTONIC ALGORITHM

Rate Monotonic is a **static** algorithm, based on fixed parameters, which is to say based of **static priority constraints**. Rate stand for one over period and the priority is monotonic with respect to the period.

	T	C
J1	3	2
J2	5	1

At any time, the algorithm chooses the task with higher priority based on **hyper period**. Hyperperiod is the least common multiple between periods. **Priority** is inversely proportional to the period of tasks; T2 > T1 but P2 < P1.

By now to know if they're schedulable or not I just have to try. Later on we'll see how to check their schedulability without actually schedule them.

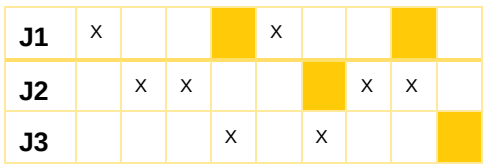


It is very important to consider the **utilization factor**, calculated adding Ci/Ti for each existing task. Later it will be defined rigorously. Whit this algorithm the processor is **fully utilized**, meaning that the utilization can't be improved.

	T	C
J1	4	1
J2	6	2
J3	9	3

P1 > P2 > P3

This isn't schedulable with this algorithm because J3 can't finish before its period lasts: J1 has higher priority and minor time of execution than J3, this means that J3 will never be scheduled due to J1's higher priority. So when J3's period last, it hasn't finished. This happens because rate monotonic is **pre-emptive**.



Having two tasks with:

- **The same period**, which is to say the **same priority.**
- **Different computational time** (that is the same as having a task with that period and a computational time equal to the sum of the computational times).

The scheduler is adopting a first come first served policy. Tasks can only be interrupted while computing by tasks with higher priority. If a task can't be performed, the scheduling is not feasible.

SCHEDULABILITY ANALYSIS

We want to know if it's possible to have a schedulability analysis. We can either simulate the scheduling or use the concept of utilization.

The **utilization factor** is defined as follows:

Given a set of periodic tasks $J^S = \{J_1, \ldots, J_N\}$, we call utilization factor U the ratio of time used by the CPU to schedule tasks.

- C_i/T_i is the ratio of time used by the CPU for the i-th task.
- $U = \Sigma\, C_i/T_i$ is the utilization factor, the ratio of time used to execute the whole task set.
- U can be increased by making the computation time C_i longer, or by making periods T_i shorter.

Having U, it is possible to apply a theorem necessary to schedulability, that helps understanding whether a scheduling is feasible or not.

Non-schedulability theorem: for a set of periodic tasks J S= {J1, ..., Jn}, it does not exist any algorithm A such that J is schedulable if U > 1.

This condition is **necessary**, but NOT sufficient. If $U \leq 1$ we still don't know whether the scheduling is feasible or not; if U >1 we know for sure it isn't.

A concept already mentioned is the one of **fully utilized processor**. A processor is fully utilized by a set of tasks if an increment of the computational time of tasks or a decrement of its period (higher priority) makes the scheduling unfeasible, U can't be increase anymore representing an upper bound. It is also possible to compute the lower bound as the minimum among all the possible utilization factors (computed among different set of tasks). If U is lower than the lower bound, it is a **sufficient** condition to schedulability.

We can compute the **lower upper bound** for rate monotonic as $U\ lub\ (RM) = n(2^{\frac{1}{n}} - 1).$

Sufficient conditions for RM: $\mathrm{U} < \mathrm{U\,lub\ (RM)}.$

When we talk about the "lower upper bound," it generally refers to a **range** that is bounded both from below and above. In the context of schedulability theory, this could be a range of values for parameters like task execution times or task inter-arrival times that must be satisfied to guarantee schedulability.

Now we have a fast check to know if the scheduling is potentially feasible. This equation is an important result and represents the minimum asset computing by the processor. If you consider only task sets with **harmonic relationships**, meaning that for every task:

$$\text{for every task Ji, Jj either Ti} = \text{nTj or Tj} = \text{nTi.}$$

Which means that the period of a task is a multiple of the period of every other task. In the case of harmonic relationships, the necessary and the sufficient conditions correspond, meaning that RM is the algorithm that performs best.

EXAMPLE

I want to check if they're schedulable, checking the necessary and sufficient conditions.

	T	C
J1	3	1
J2	6	2
J3	9	3

$n = 3, U\ lub = n\left(2^{\frac{1}{2}} - 1\right) = 2 * \left(2^{\frac{1}{2}} - 1\right) = 0.78$, $U = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$

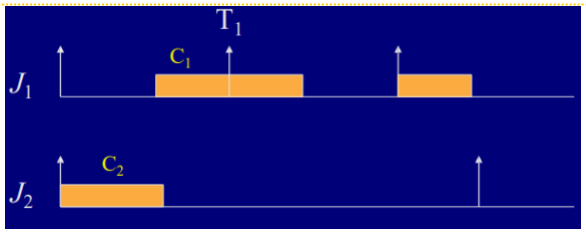
Since U = 1 we must check lower upper bound, and it's supposed to be lower than U lub to satisfy the sufficient condition. Actually U > U lub, so the test is not feasible. It seems to be a harmonic relationship, but its condition isn't true for every period. ($T2 \neq nT3$).

RM is proved to be optimal among static algorithms.

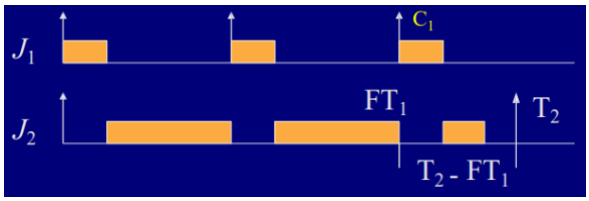
What does optimal mean? If a task is not schedulable with RM it isn't with other algorithms.

Generally, a scheduling is feasible if I can schedule these two scenarios:

- **Acond:** Condition for the NON RM.
  $C1 + C2 \leq T1.$



- **RMcond**: Condition for the RM.
  $(F+1)\ C1 + C2 \leq T2$.



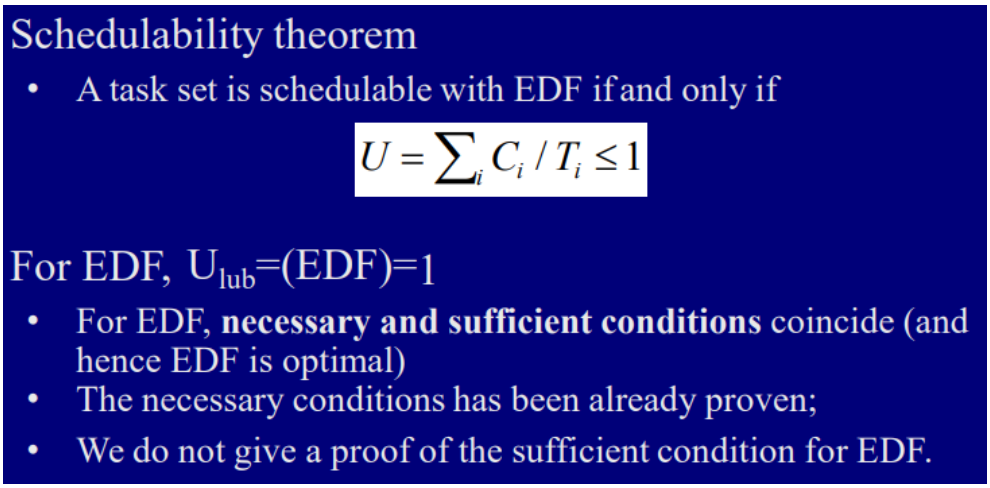
DYNAMIC ALGORITHMS

Is it possible to do better with RM? Just dynamic algorithms do better. Dynamics algorithms change priority during execution time.

EARLIEST DEADLINE FIRST

It assigns priority to the task with the closest deadline; at each task is assigned a priority that is inversely proportional to the difference between the deadline and the current time. From a pragmatic point of view, it acts worse than RM, not being used on every CPU.

The necessary and sufficient conditions for EF correspond, making it generally optimal: its upper bound is 1.



	T	C
J1	6	2
J2	15	5
J3	10	3

J1	X	X					X	X					X	X					X	X								X	X	
J2						X			X	X	X	X						X			X	X	X	X						
J3			X	X	X										X	X	X								X	X	X			

At the end we choose J3 and not J1 because J3 has been in the queue for longer, actually applying FIFO policy.

We must take decisions when a task ends or when a new period begins. Otherwise, the scheduler is not involved.

NEW SUFFICIENT CONDITION FOR RATE MONOTONIC

We don't have general and sufficient conditions for rate monotonic in general. Being simple, RM is better in general. Bini, Buttazzo and Buttazzo proposed a different sufficient condition for rate monotonic, being sufficient you can have more than one.

Theorem 1.1 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i. Then, Γ is schedulable with the RM algorithm if*

$$\prod_{i=1}^{n}(U_i + 1) \leq 2. \qquad (1.1)$$

Here we're computing a product. Let's check if the last example can be scheduled with this new condition for RM.

$$\left(\frac{1}{3}+1\right) * \left(\frac{1}{3}+1\right) * \left(\frac{3}{10}+1\right) \leq 2 \rightarrow \frac{104}{45} \leq 2 \rightarrow NO$$

This task can't be scheduled with RM at all.

This condition is verified when using the **parabolic bound**. Why? Let's graphically visualize its behaviour with two tasks.

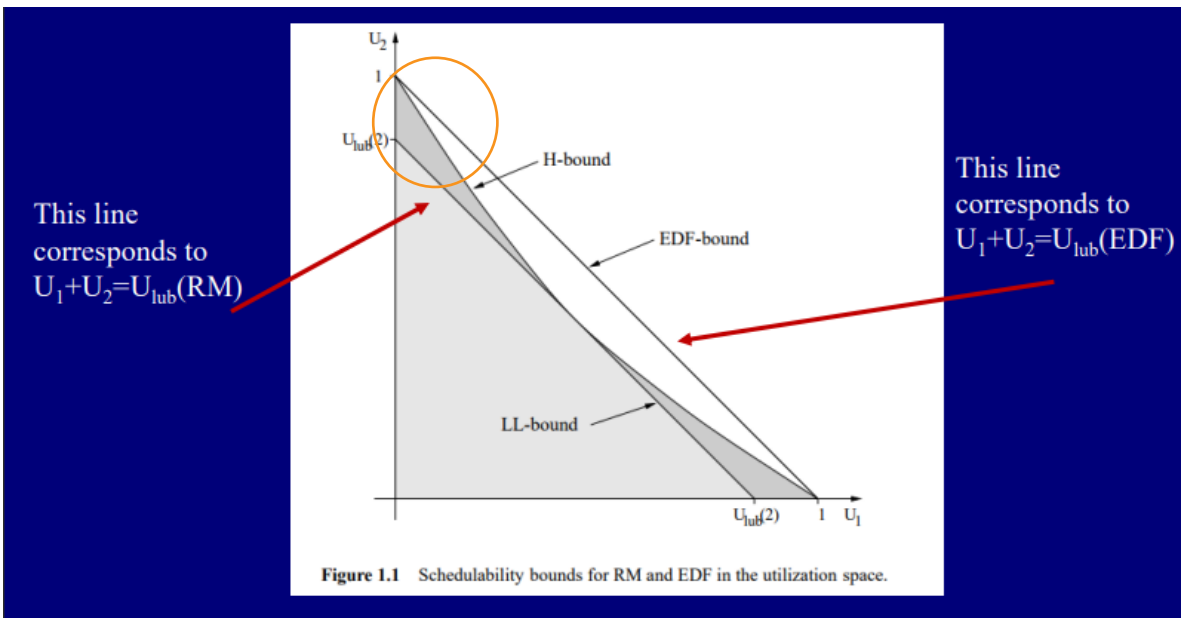


Figure 1.1 Schedulability bounds for RM and EDF in the utilization space.

Tasks in the orange circle satisfy the parabolic bound.

DEADLINE MONOTONIC

It's another **static algorithm** for which the relative deadline corresponds to the period. The idea of DM described as:

- The period is constant in time.
- Execution time is constant in time.
- The relative deadline corresponds to
- All tasks are independent.

Priority is inversely proportional to the relative deadline, while RM uses the period.

	T	C	D
J1	4	1	3
J2	10	2	9
J3	12	3	8

RM: P1 > P2 > P3; DM: P1 > P3 > P2

It's interesting the **schedulability test**, which is the same as RM by substituting T with D:

$$\sum_i C_i / D_i \leq n\left(2^{1/n} - 1\right)$$

However, this is pessimistic. Because of that, has been developed a new test based on the concept of **critical instant**: we're introducing an algorithm that checks every instant. We're also introducing the interference I on a task (due to all higher priority tasks), with whom the schedulability test can be written as follows.

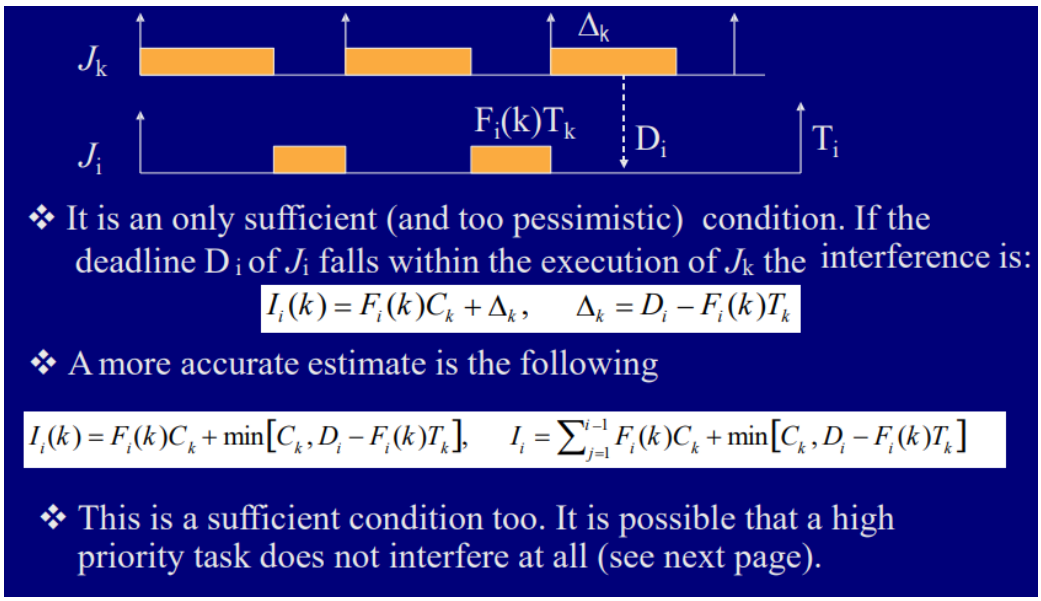
$$I_i = \sum_{j=1}^{i-1} \lceil F_i(j) + 1 \rceil C_j \qquad \forall i : 1 \leq i \leq n \qquad \frac{C_i + I_i}{D_i} \leq 1$$

N.B. the highest priority task's interference is zero.

$$\forall i \; from \; 1 \; to \; n: \sum_{j=1}^{n} \frac{Cj}{Tj} \leq i(2^{\frac{1}{i}} - 1)$$

We're using the rate monotonic test to check every pair of tasks. This is useful because even if the whole asset can't be scheduled, a subset of it can.

How do we compute the interference? It will lead us to the final algorithm that is necessary sufficient.



We need the interference of all the higher priority tasks on Ji. When considering all the interferences, the resulting equation is the previous one.

	T	C	D
J1	4	1	3
J2	6	1	5
J3	8	2	7

The order of priorities is: $P1 > P2 > P3$

Let's now calculate the condition for each task to check their schedulability:

- $J1 : C1 + I1 \leq 3\ ?\ Yes \rightarrow I1 = 0 \rightarrow C1 = 1 < 3$
- $J2 : C2 + I2 \leq 5\ ?\ Yes \rightarrow I2 = (F2(1) + 1) * C1 \rightarrow F2(1) = floor\left(\frac{5}{4}\right) = 1$
- $J3 : C3 + I3 \leq 7\ ?\ Yes \rightarrow F3(1) = floor\left(\frac{7}{4}\right) = 1,\ F3 = floor\left(\frac{7}{6}\right) = 1 \rightarrow I3 = (F3(1) + 1) * C1 + (F3(2) + 1) * C2$

They are, but we must remember that this condition it's **just sufficient**, since the computation of I is pessimistic: it's possible that a high priority task doesn't interfere at all.

EXAMPLE

	T	C	D
J1	4	2	3
J2	14	4	9

The priority order is P1 > P2. Also, we can compute the (canonical) schedulability test consists of computing:

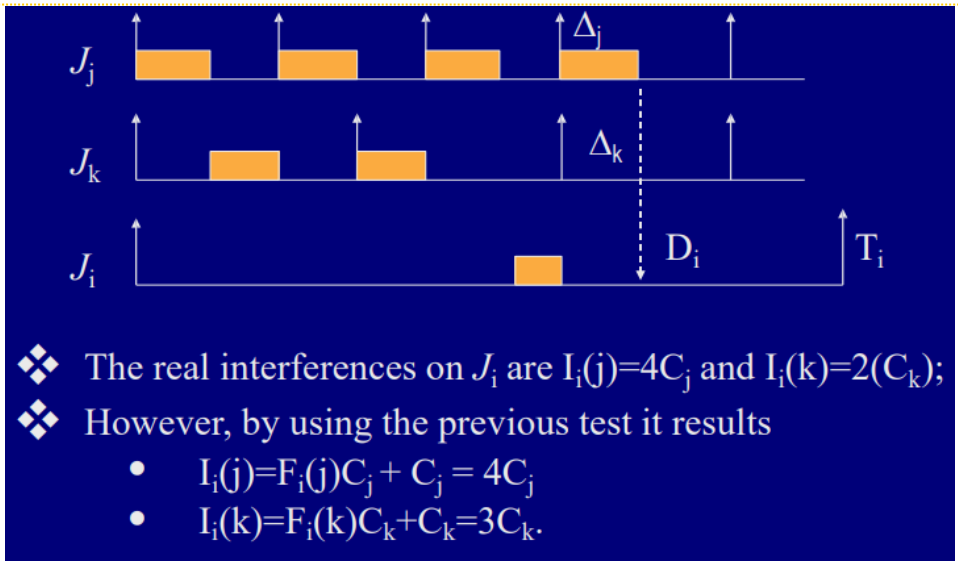
$$\frac{C1}{D1} + \frac{C2}{D2} \leq 2 * \left(2^{\frac{1}{2}} - 1\right) \sim 0.83, \qquad \frac{2}{3} + \frac{4}{9} = \frac{10}{9}\ \ not\ \leq 0.83$$

We can also do another test, considering the **critical instant** for each task. This test considers the interference of tasks with higher priority.

- $J1: \frac{2+0}{3} \leq 1,$ there's no need to compute I1 since J1 is the Task with highest priority.
- $J2: \frac{4+6}{9} \leq 1\ NO! \rightarrow\ \ F2(1) = floor\left(\frac{9}{4}\right) = 2 \rightarrow I2(1) = (2 + 1) * 2 = 6.$

But if we apply $F2(1) = 2 * 2 + \min(2,1) = 5,$ with $\Delta 1 = 9 - 2 * 4 = 1, that\ stands\ for\ D2 - F2(1) * T1$ it is schedulable.

This is a sufficient condition again because we're not considering the real interference.



AUDSLEY THEOREM

This problem was solved by Audley implementing an algorithm which generates necessary and sufficient conditions for RM and DM. There are two concepts that need to be clarified.

- t_k is NOT a task, but an **attempt**. It is the first instant t0 in which the task could finished if there weren't interference.
- I_i^k is the interference on Ji due to higher priority task in the interval [0, t_k]. N.B. My notation: I(i,k).

I'm iterating this computation until I reach a time instant in which I'm sure I can finish without any interference.

A subset of tasks is schedulable with Deadline Monotonic iif the following conditions are true:

$$\forall i, 1 \le i \le n, \quad \exists k : \left(C_i + I_i^k \le t_k\right) and \left(t_k \le D_k\right)$$

$$dove \quad I_i^0 = \sum_{j=1}^{i-1} C_j \qquad t_k = I_i^{k-1} + C_i \qquad I_i^k = \sum_{j=1}^{i-1} ceiling\left(\frac{t_k}{T_j}\right) C_j$$

EXAMPLE

	T	C	D
J1	4	1	3
J2	5	1	4
J3	6	2	5
J4	11	1	10

N.B. Don't forget that t2 represent the instant in which J2 may finish (or maybe not!)

N.B. Floor = parte intera inferiore, ceiling = parte intera superiore.

I'll initialize the interference to the sum of all higher priority tasks:

- $J1$: $I(1,0) = \sum_{j=1}^{0} 0 = 0 \rightarrow t1 = I(1,0) + C1 = 1$. Now we need to check if $C1 + I1 \le t1 \quad AND \quad t1 \le D1$
- $J2$: $I(2,0) = \sum_{j=1}^{1} C1 = 1 \rightarrow t2 = I(2,0) + C2 = 2$. It's important to notice that this computation stands just for the computation of the interference w.r.t t0.

$$J2:\ I(2,1) = ceiling\left(\frac{t1}{T1}\right) * C1 = ceiling\left(\frac{2}{4}\right) * C1 \rightarrow\ C2 + I2 \leq t1\ AND\ t1 \leq D2\ ARE\ TRUE$$

N.B. Sometimes you can just initialize t2 at the sum of the duration of the task J1 and J2 (which seems legit)

- $J3:\ I(3,0) = C1 + C2 = 2 \rightarrow t1 = 4.\quad I(3,1) = ceiling\left(\frac{t1}{T1}\right) * C1 + ceiling\left(\frac{t1}{T2}\right) * C2 = 2.$
  $\rightarrow C3 + I3 \leq t1\ AND\ t1 \leq D3\ ARE\ TRUE$
- $J4:\ I(4,0) = C1 + C2 + C3 = 4 \rightarrow t1 = I(4,0) + C4 = 5.\quad I(4,1) = ceiling\left(\frac{t1}{T1}\right) * C1 + ceiling\left(\frac{t1}{T2}\right) * C2 + ceiling\left(\frac{t1}{T3}\right) * C3 = celing\left(\frac{5}{4}\right) * 1 + ceiling\left(\frac{5}{5}\right) * 1 + ceiling\left(\frac{1}{6}\right) * 2 = 5.$
  $\rightarrow C4 + I4 \leq t1\ AND\ t1 \leq D4\ AREN'T\ TRUE$
- So now we must continue with another t2 for J4 since the schedulability of J1, J2, J3 actually works, but J4's one doesn't. $t2 = 5 + 1 = 6 = I(4,1) + C4 \rightarrow I(4,2) = ceiling\left(\frac{6}{4}\right) * 1 + ceiling\left(\frac{6}{5}\right) * 1 + ceiling\left(\frac{6}{6}\right) * 2 = 6 \rightarrow\ C4 + I4 \leq t2\ AND\ t2 \leq D4\ AREN'T\ TRUE$
- $t3 = 7 \rightarrow\ I(4,3) = ceiling\left(\frac{7}{4}\right) * 1 + ceiling\left(\frac{7}{5}\right) * 1 + ceiling\left(\frac{7}{6}\right) * 2 = 8 \rightarrow\ C4 + I4 \leq t3\ AND\ t \leq D4\ AREN'T\ TRUE$
- $t4 = 9 \rightarrow I(4,4) = 9 \rightarrow\ C4 + I4 \leq t4\ AND\ t4 \leq D4\ AREN'T\ TRUE$
- LAST ATTEMPT (this is the last time that satisfies the second condition): $t5 = 10 \rightarrow I(4,4) = 9 \rightarrow\ C4 + I4 \leq t5\ AND\ t5 \leq D4\ ARE\ TRUE$

This theorem provides necessary and sufficient condition for DM but also for RM. When we iterate the algorithm, we're not doing it for each instant of time, but for the useful ones (tk = I(I,k-1)+Ci). It's important to that schedulability test is quick, since I'm consuming resources and I may have to try it on different tasks set until I find a suitable set with a feasible scheduling.

CAPITOLO 3 NON-PERIODIC TASKS

Non periodic tasks are quite different from periodic ones: while it's crucial that periodic tasks end before deadline lasts, we assume non periodic tasks either hard or soft. Soft non periodic tasks don't have necessarily to end before the deadline lasts, the earlier they do, the better, but it won't have dangerous consequences.

	C	D
Ja	5	10

In general, it doesn't make sense to say that we want always to meet the deadline. Needing constraints, we define the **minimum interarrival time**: we have a constraint on the time that must last between the request of the aperiodic tasks and the following one.

EXAM EXAMPLE

- We have a camera acquiring pictures of a manufacturer tools at 25Hz. This is a periodic task.
- Let's suppose we have another task that elaborates J1's pictures and detects flags. This a periodic task since it samples every picture.
- There's also J3 that reports flagged pictures. This is an aperiodic task, executed when flags are detected. This is a particular kind of aperiodic task called **sporadic** task: we can set a minimum interarrival time since J3 can't be executed faster than J2.

We can consider **sporadic** task as a periodic one using the minimum interarrival time as it was its period. With sporadic tasks, the schedulability test becomes:

$$U_{periodic} + U_{sporadic} \leq U_{lub}(A).$$

We compute U for sporadic tasks considering MIT as their period (the minimum interarrival time). For truly aperiodic task, we obviously can't define the MIT.

If we want to guarantee the schedulability we can both do **online test** (test is performed when the aperiodic task arrives), namely we check if given the current situation is possible or not to schedule the new aperiodic task with the periodic ones that are in execution. Or **offline test** that checks schedulability in the worst case, assuming all aperiodic tasks making request at the same time, unknown a priori.

N.B. being it a test, it is possible to reject the task if the scheduling is not feasible.

BACKGROUND SCHEDULING

We can use **background scheduling** to schedule aperiodic tasks. The idea is that periodic tasks are always considered to have the highest priority, while aperiodic ones are left in the background. This is because non periodic tasks don't have to meet a deadline.

Background scheduling can be adopted either when we use RM or EDF for periodic tasks. There are two different queues, to handle high and low priority tasks:

- Aperiodic tasks are scheduled either with (non-real time) FCFS or with EDF.
- Low priority tasks can be scheduled with FCFS policy.

Is there test for aperiodic tasks schedulability? Of course, there are. One of the most popular tests is an offline one that provides sufficient conditions.

First, we need to compute how much residual time you have in a hyperperiod Φ; how? To compute it we need to compute both the utilization factor and the hyperperiod; then: $\Phi = (1 - U)H$. We don't need the absolute deadline in the time axis, since we're doing an offline test; so we compute the relative deadline to check schedulability (the computation time divided by the residual time times the hyperperiod must be less or at most equal to the relative deadline). If it doesn't work, we have more conditions to check.

$$ceiling(\frac{C_i}{\Phi})H \leq D_i$$

To compute the available CPU time it is possible to compute the hyper- period H of periodic tasks (least common square of periods) and $\phi = (1- U)H$ (the idle time for each hyper-period)

An aperiodic instance with computation time C_i and deadline D_i can be guaranteed if

$$ceiling\,(C_i / \Phi)H \leq D_i$$

which simply computes the number of required hyper-periods $ceiling\,(C_i / \Phi)$ times the length of an hyper-period H.

Notice that, with this test, a task J_i whose relative deadline D_i<H will never result as schedulable.

The same offline check can be obtained just by summing all the computation times, which also must be less or equal to the relative deadline.

EXAMPLE

	T	C	D
J1	5	2	5

J2	8	3	8
Ja	…	3	10

- The utilization factor is $U = \frac{2}{5} + \frac{3}{8} = \frac{16}{40} + \frac{15}{40} = \frac{31}{40}$
- Now we compute the hyperperiod $H = 40$ (mcd)
- Then we need to compute how much free time I have in an hyperperiod $\varphi = \left(1 - \frac{31}{40}\right) * H = \frac{40-31}{40} * 40 = 9$
- How many hyperperiods do we need to schedule the aperiodic task? $celing\left(\frac{Ca}{9}\right) = 1$

Since we don't know when the task arrives, we must reason in terms of hyperperiod (at least one), but it's a very pessimistic view.

EXAMPLE

	C	D
J3	3	10 → 12
J4	2	12

$$\Phi = \left(1 - \frac{7}{12}\right) * 12 = 5 \rightarrow ceiling\frac{3+2}{5} * 12 \leq 12$$

Having J3 and J4 the same relative deadline, the tests coincide. For both aperiodic tasks the worst case is when the other task arrives.

In **online test** you may have a task already running and a task that arrives. I need to performe the online test every time that there's a new task. We need to introduce the idea of **residual computational time** of a task, namely the remaining time to accomplish it.

Referring to the last example:

We suppose tasks are scheduled with decreasing residual computational time. In details, the schedulability analysis is performed considering the new task and all the tasks with highest priority with regards to EDF.

$$\forall i = 1, \ldots, n \quad \text{it must hold} \quad t + ceiling\left(\frac{S_i}{\Phi}\right) H \leq d_i$$

$$where \quad \Phi = (1 - U)H \qquad S_i = \sum_{j=1}^{i} c_j(t)$$

$c_j(t)$ residual computational time of task J_j at time t;

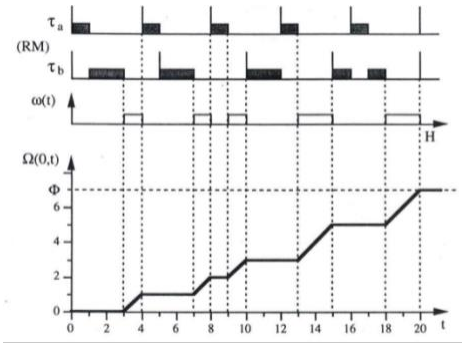
- Firstly, we reorder the tasks according to their priority (we look at the deadlines).
- We perform the analysis starting from the highest priority task. The following ones will also consider task with higher priority.

Referring to the last example (we're considering time 0 already lasted):

- $i = 1 \rightarrow J_4: S_1 = C_4(1) = 2 \rightarrow 2 + ceil\frac{2}{5} * 2 \leq 7(rel\ deadline)$. It doesn't work, the condition is very strict.

- $i = 2 \rightarrow J_3: S_2 = C_3(2) + C_4(2) = 2 + 1 = 3 \rightarrow 1 + \text{ceil}\frac{3}{5} * 12 \leq 8.$

This test is very conservative, but alternatives are not so good. One of them is to explicitly compute **idle times**, namely intervals in which the processor is not working.



Suppose having two periodic tasks (black ones) scheduled with RM. $\omega(t)$ is a function that is 1 just when the processor is idle. $\Omega(0,t)$ is the integral of $\omega(t)$.

How's the schedulability of a single task Ji? It is guaranteed if the computational time is less or equal to the integral between arrival time and deadline: $C_i \leq \Omega(r_i, d_i)$.

So, if you have a set of aperiodic tasks for which all conditions holds, also must hold that

$$\forall i = 1,\ldots, n \quad \text{it must hold} \quad \sum_{j=1}^{i} c_j(t) \leq \Omega(t, d_i)$$

If I want to store the values of the integral into a vector, how long must be this vector? It must consider just what happen in an hyperperiod (this question comes from the fact that deadline and hyperperiod aren't necessarily close to each other).

Also $\Omega(0, t)$ can be stored in a vector of H entries by storing the start and end of each interval where the processor is idle, since we can compute:

$$\Omega(a,b) = \Omega(0,b_h) - \Omega(0,a_h) + \Phi[\,floor\,(b/H) - floor\,(a/H)]$$

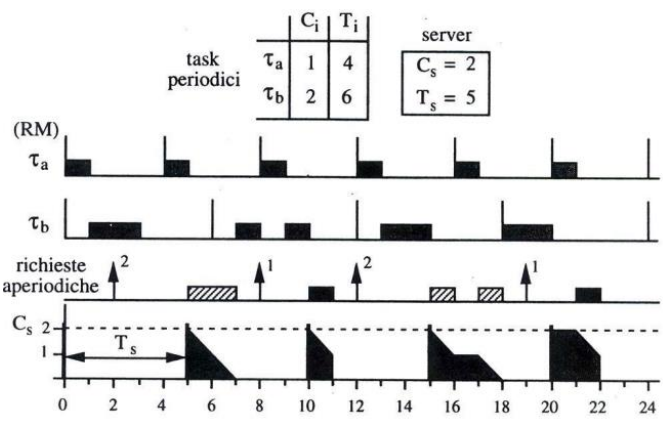
where a_h and b_h are the remainders of the divisions a/H and b/H, respectively.

COPIARE DISEGNETTO

One of the problems with background scheduling is that depending on how many aperiodic tasks we have and how many resources they use, it can be difficult to execute background tasks. We introduce the concept of **server**, a periodic task that is devoted to serve the request of aperiodic tasks, namely executes aperiodic tasks if present.

- The server is activated at regular intervals.
- In absence of pending requests, the residual capacity is lost until the beginning of the next period. An aperiodic task that makes a request while the Server is active will be executed in the next period.
- Usually, the Polling Server is scheduled as a periodic task with RM and maximum priority (lowest period) → not mandatory but possible.

POLLING SERVER EXAMPLE



All tasks are ready at the initial time, but since at the beginning of the time there are no periodic requested, the server's capacity is immediately lost. At 3 you may expect that the server schedules the task, but it lost capacity if there aren't request at the beginning.

Why do we need to impose this? Because we will schedule aperiodic tasks with RM (or EDF) together with the periodic tasks, but a task with high priority is scheduled as soon as it's ready: in this case the priority of the server is higher than the priority of tau b (Ts < Tb), if we didn't lose server capacity, we would violate this assumption (since the server, with higher priority, will be executed after tau b). (Server is periodically executed).

What happen if a periodic task has higher capacity than the server? It won't be continuously scheduled.

The first check to do regards whether we can build the server or no, namely if it can be scheduled with the other tasks. The schedulability test requires, first to guarantee all periodic tasks, including the Polling Server Js.

$$U_{periodic} + \frac{C_s}{T_s} \leq U_{lub}$$

If there are m sporadic tasks J_j that require to be guaranteed for all possible instances, we must consider the worst case and assume that their periods equal to the minimum interarrival time

$$U_{periodic} + U_{sporadic} + C_s/T_s \leq U_{lub}$$

Next we can guarantee a single instance of an aperiodic task Ja with computational time Ca executed within the Polling Server.

We have both offline and online tests.

OFFLINE. The **worst case** with the polling server is when the request arrives a little after the beginning of the of the period, when the server has already lost its capacity.

Let F_a=floor (C_a/C_s) the number of complete Server periods that are necessary to schedule J_a;

- In the worst case (i.e., if J_a makes request immediately after the beginning of the Server period) the execution of J_a starts after T_s (i.e., at the following period) and terminates after $(F_a+1)T_s$;
- Therefore it must hold, in the worst case

$$r_a + T_s + (F_a + 1)T_s \leq d_a$$

i.e.,

$$(F_a + 2)T_s \leq D_a$$

Why +1? Suppose that we have a server, can we schedule this server together with the periodic tasks?

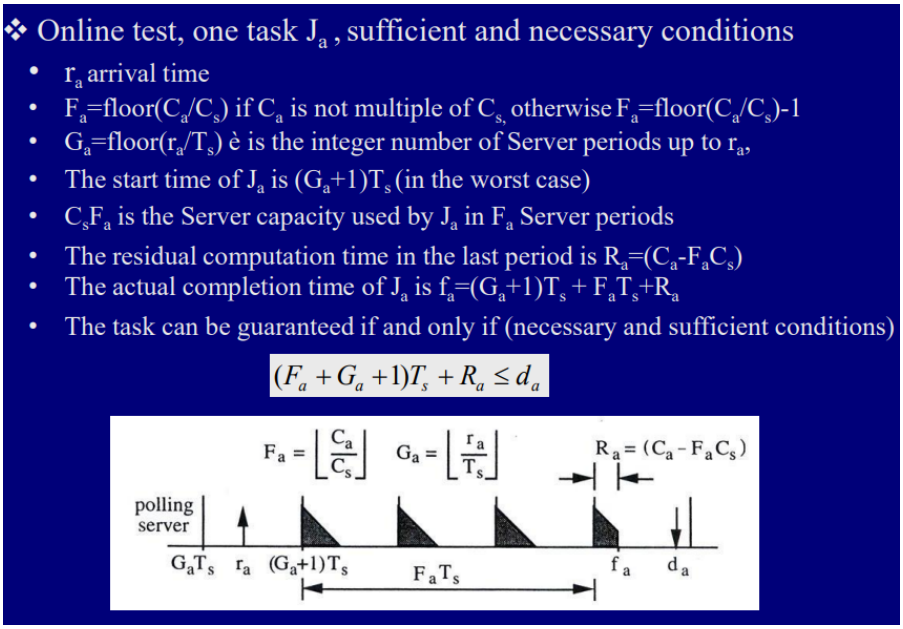
	C	Ts
Js	2	5

Let's do this computing that the utilization factor is less or equal to U to the lower bound. Now suppose that we have an aperiodic task with computation time of 3 and unknown arrival time. For sure we'll lose one Ts, then how many periods do we need? With the generic polling server, we don't know where the task will be executed. We haven't already assumed that the server has the highest priority. So, we compute the full number of periods that we need: $F_a = floor\frac{3}{2} + 1$. So, we need at least two full period.

To have necessary and sufficient conditions we must assume that the server has the highest priority among real time tasks.

Let's see a new example managed both with online and offline test.

ONLINE.



We define all these things.

A task is guaranteed iif the formula in figure holds.

F_a gives the number of full periods.

I want to know if the ending time is less or equal to the deadline.

The formula to get Ra works if they're not multiple of each other.

After having checked that aperiodic and periodic tasks are schedulable together, we can check if aperiodic tasks meet their deadlines just by considering them, since they have (the polling server) the highest priority and we have already guaranteed that periodic tasks end before their deadlines last.

Do we have similar approaches but different servers? Yes, we have.

DEFERRABLE SERVER

The Deferrable Server has a period Ts and a computation time Cs, referred to as Server Capacity.

- At regular intervals, the Server is activated to execute aperiodic tasks.
- In absence of pending request, the residual capacity is preserved (and then restored to its maximum value at the beginning of the next period).
- An aperiodic task which arrives while the server is active will be served immediately (if the server is not busy with other requests and/or there is residual capacity left).
- It is scheduled as a periodic task with RM (usually with maximum priority, i.e., minimum period

It works like a polling server but in absence of capacity it doesn't lose its whole capacity, it just consumes it during time. Isn't it a violation of RM constraints? If the server is ready it should be scheduled! It can be done, but it affects lower upper bound computation, namely it's different.

So, the schedulability test is shown on the right.

$$U_{\text{lub}} = n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n} - 1\right]$$

$$\lim_{n\to\infty} U_{\text{lub}} = \ln 2 - \ln\left(\frac{4U_s+2}{U_s+2}\right)$$

REAL-TIME COMMUNICATION

The problems we want to solve are:

- Ensure the execution times of tasks. When a task is ready it must be executed, if pre-empted it goes back to the ready queue. When an active task concludes, at the beginning of the period it's newly inserted in the ready queue. A task can be blocked by a lower priority task from which is waiting for something (es. Waiting for data).
- Ensure that communication works in real time (data are sent and received within a given deadline).

By now we're considering there's only a CPU doing the scheduling, but if there were more these CPUs must exchange data during a certain period.

How do we do this? With semaphores and mutexes.

Given two periodic tasks J1 or J2 that execute some loops. Supposing that I want to execute the loops in a certain period e.g. T1 = 100ms and T2 = 50ms. Also, I have a sleep function that shuts down the task if it's been executing for too long. The sleep function can't be into the loop with the period as parameter because the task's execution will last the time it needs + the sleep time. In J1 we'll have sleep(T1-(T1-T2).

A semaphore or a mutex is something that can be taken with a primitive p(S1) and freed with the primitive v(S0). Mutex and semaphores stand for **mutual exclusion**, there can only be a task executing. As soon as the semaphore is released, another task can take it and be executed. This is true and useful in general, but how do we handle with this in real time? This a big deal, since tasks can be blocked, so the priority we set before is no longer true. We need new and more schedulability policy that consider this possibility.

This problem is also known as **priority inversion**:

- A high priority task is blocked by a low priority task for an indefinite time.
- If the low priority task enters the mutual exclusion region the first, the high priority task is blocked.
- The blocking time can become longer because of medium priority tasks that pre-empt the low priority task (even if the medium priority tasks do not access the mutual exclusion region)