Embedded Information Systems

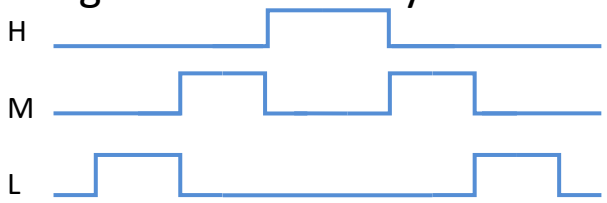
2. Scheduling

October 6, 2020

Reminder:

For simplicity imagine that to every task a different priority level is assigned.

Illustration:



We have one **low priority** (L=low), one **medium priority** (M=medium) and one **high priority** (H=high) task.

This assignment happened in **design-time**. All the tasks start running immediately after the **request**, if their priority is **the highest** among the tasks **ready** to run.

The **response time** of the **lowest priority** task on the figure is: $R_L=C_L+C_M+C_H$

If the medium and/or the high priority task are released **periodically**, then depending on the time relations, it might happen, that these tasks will run **more than one time** during R_L.

In a more general case, for a task at priority level i, the **worst-case response time** can be calculated using the following formula:

$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

where I_i is the so-called **interference.** The **interference time** is the total computation time of those higher priority tasks, which prevent task i to complete its actual run. $\forall k \in hp_i$ refers those tasks, which have **higher priority** than i (*hp*=higher priority). The $\lceil \quad \rceil$ sign is the operator of assigning the **upper integer**. $\lceil 1.02 \rceil = 2$, $\lceil 2.0 \rceil = 2$. Since in the above formula the unknown R_i on the left-hand side is present also in the argument of the **highly nonlinear** function on the right-hand side, it can be evaluated only via an **iterative** procedure:

$$R_i^{n+1} = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i^n}{T_k} \right\rceil C_k$$

The iteration will **stop** at step n_0 where $R_i^{n_0+1} = R_i^{n_0}$

The name of this method in the literature is **Deadline Monotonic Analysis** (DMA).

Schedulability, schedulability tests:

Reminder:

- ***necessary:*** if the necessary condition is not met, then no schedule exists.
- ***sufficient:*** if the sufficient condition is met, then a schedule always exists.
- ***exact****:* gives the necessary and sufficient conditions and shows the existence of the schedule. The complexity of the exact schedulability test is high, these are so called NP-complete problems, which are hard to handle, and therefore they will not be considered.

For **periodic tasks** among the **necessary conditions** the **processor utilization factor** can be mentioned, which is the sum of the processor demands relative to the unit of time:

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

For a single processor system if $\boldsymbol{\mu \leq 1}$ is **not met**, then the tasks are **not schedulable**, i.e. $\mu \leq 1$ is a **necessary condition**.
(Here n stands for the # of tasks.) If we have $\boldsymbol{N}$ processors, then $\boldsymbol{\mu \leq N}$.

Scheduling strategies:

Rate-monotonic (RM) (1973): For periodic and independent tasks if $D_i = T_i$ and C_i are known and constant. The highest priority is assigned to the task with the shortest period. The procedure is pre-emptive. We assume that the time of context switching between tasks is negligible. Is it **OK**?

For the **RM** algorithm sufficient test is available. ***n*** denotes the number of the tasks to be scheduled. It might happen that the actual set of tasks is schedulable with the **RM** strategy even at higher processor utilisation; however there is **no guarantee** for it.

$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \xrightarrow[n\to\infty]{} ln\, 2 \sim 0.7$$

Simulations with randomly selected $\boldsymbol{T_i}$ and $\boldsymbol{C_i}$ values were reported successful up to $\boldsymbol{\mu = 0.88}$.

To achieve **100%** utilization when using fixed priorities, assign periods so that all tasks are **harmonic.** This means that for each task, its period is an **exact multiple** of every other task that has a shorter period.

Example: Under what period and computation time conditions reaches the RM strategy the limits of schedulability for $\boldsymbol{n = 2}$?

Reminder:

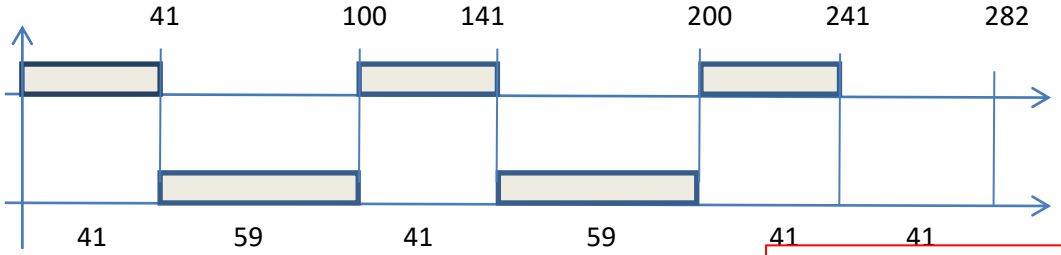
If $\frac{T_2}{T_1} = \sqrt{2}, C_1 = T_2 - T_1, \frac{C_1}{T_1} = \frac{T_2 - T_1}{T_1} = \frac{C_2}{T_2}$:

$\mu = 2(\sqrt{2} - 1)$, and for arbitrary i :

If $\frac{T_{i+1}}{T_i} = 2^{\frac{1}{n}}, C_i = T_{i+1} - T_i, \frac{C_i}{T_i} = \frac{T_{i+1} - T_i}{T_i} = \frac{C_{i+1}}{T_{i+1}}$ then $\mu = n\left(\frac{T_{i+1}}{T_i} - 1\right) = n\left(2^{\frac{1}{n}} - 1\right)$

Example: We have two tasks $T_1 = 100\ ms, C_1 = 41\ ms, T_2 = 141\ ms, C_2 = 59\ ms$.

$\mu = \frac{41}{100} + \frac{59}{141} = 0.41 + 0.4184 = 0.8284 \sim 2(\sqrt{2} - 1)$. If the requests are simultaneous :



If there is slight increase in the computation time, then the **RM** strategy will fail!

Comments:

Between 241 and 282 there is **no schedulable task**!

1. If the **RM** scheduling strategy is applied, the most **disadvantageous** is the case when **all the tasks** start with **zero phase**, i.e. the first requests are **simultaneous.**

2. Non-zero phase start is **advantageous** from scheduling point of view!

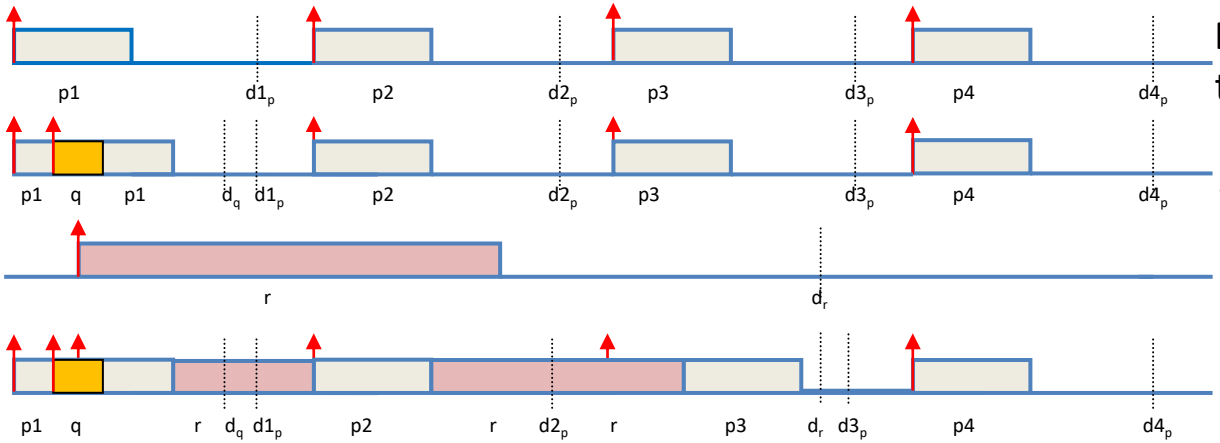
3. If the RM scheduling strategy is applied, and the **necessary condition is met**, but the **sufficient not**, then the schedulability analysis should be performed for **smallest common multiple of the periods** that can be extremely large.

Reminder:

Earliest Deadline First (EDF) strategy: The tasks are **periodic**, **independent** of each other, $D_i \leq T_i$ and C_i are **known** and are **constant**.Priority assignment is in **run-time**, and the processor is given to the task having the **earliest deadline**.The operation is **pre-emptive.**

Here we also assume that the time of context switching is **negligible**. **Is this correct?**

Sufficient schedulability test can be given: tasks meeting the above conditions are schedulable up to $\boldsymbol{\mu \leq 1}$, i.e. **100%** processor utilisation is possible.



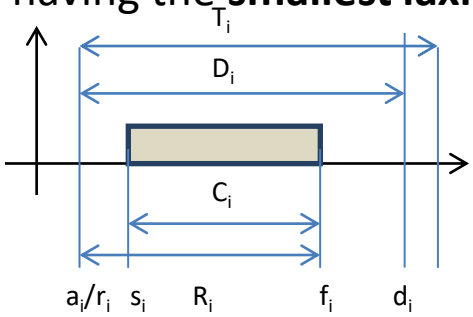
Row #1: requests and deadlines of task **p**

Row #2: request of task **q** during the run of **p1**.

Row #3: request and deadline of task **r**.

Row #4: the run of the three tasks.

Least Laxity First (LLF) strategy: Similar to **EDF**. The conditions of its application are the same, but instead of the task having the **earliest deadline**, the processor is assigned to the task having the **smallest laxity**. This is the **difference** of the **deadline** and the **remaining computation times** at the time instant of investigation.



Tasks meeting the above conditions are **schedulable** up to $\boldsymbol{\mu \leq 1}$, i.e. **100%** processor utilisation is possible.

The EDF and LLF strategies are applicable also for **aperiodic** tasks, but since the processor utilisation factor can only be interpreted in a different way, the sufficient condition above cannot be used.

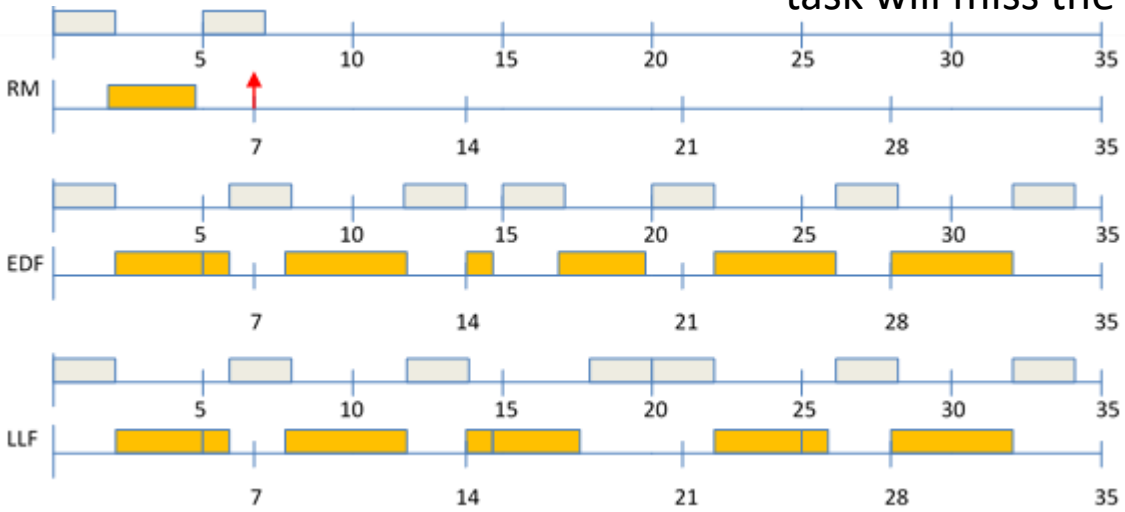
Reminder:

Example: Comparison of **RM** and **EDF** algorithms. We have two tasks.
The period and the deadline is the same. $T_1 = 5$ ms, $C_1 = 2$ ms, $T_2 = 7$ ms, $C_2 = 4$ ms. .

The processor utilisation factor:

$$\mu = \frac{2}{5} + \frac{4}{7} = 0.4 + 0.57 = 0.97$$

Here the necessary condition of the schedulability is met, but the sufficient condition only for the **EDF/LLF** strategies. If RM strategy is applied, then the second task will miss the deadline at 7 ms, but using EDF or LLF the tasks are schedulable.



Both for **EDF** and **LLF** it is obvious that if the deadlines are equal, the applied schedule should result in *less context switching*, because context switching takes time.

All the task-specific information of the pre-empted task must be saved: typically, the content of the processor's registers must be copied into the task-specific **Task Control Block** (TCB), while TCB of the task decided to run should be loaded into the registers of the processor.

These copying are supported by fast mechanisms, but still they need time.

Proof of the EDF schedulability:

Reminder:

The proof is given for periodic tasks with $D_i = T_i$. The statement is the following:
A set of periodic tasks is schedulable with EDF if and only if

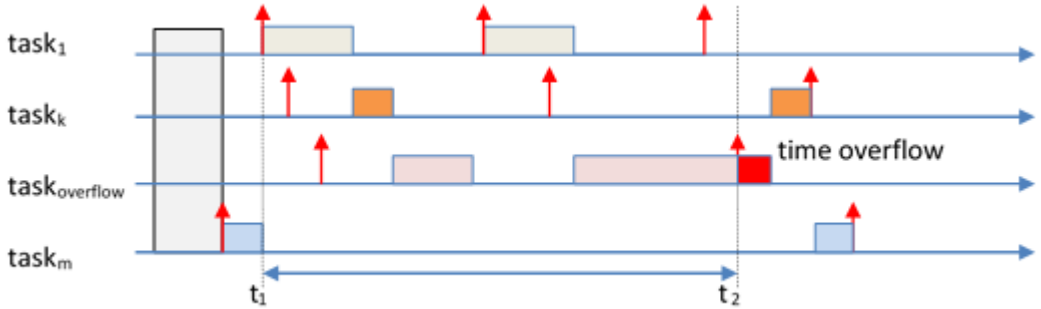
$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

Proof: **1. *Only if***: We show that a task set cannot be scheduled if $\mu > 1$.
By defining $T = T_1 T_2 \dots T_n$, the total demand of computation time requested by all tasks in T can be calculated as:

$$\sum_{i=1}^{n} \frac{T}{T_i} C_i = \mu T.$$

If $\mu > 1$, that is, if the total demand μT exceeds the available processor time T, there is no feasible schedule for the task set.

2. *If part*: We show the sufficiency by contradiction.
Assume that the condition $\boldsymbol{\mu < 1}$ is satisfied **and** yet the taskset is **not schedulable**.
The next figure helps to understand the proof. Here we can see the schedule of periodic tasks according to EDF.

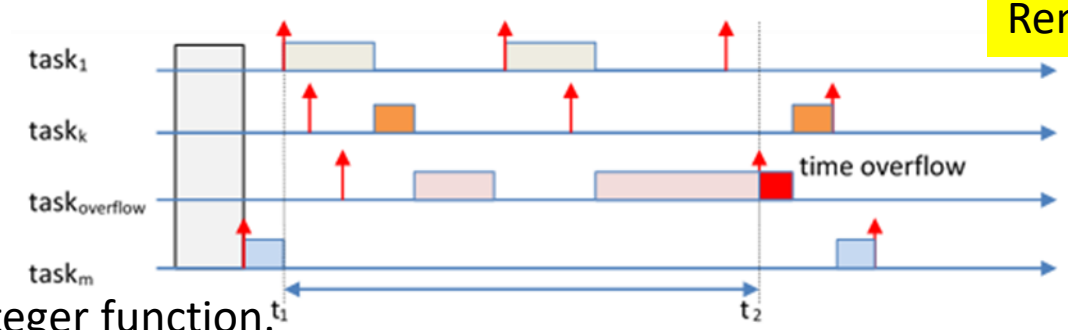


If our assumption is that the task-set is not schedulable, then there must be such a task, which **misses** its deadline.

Let t_2 be the instant at which the time/overflow occurs and let $[t_1, t_2]$ be the longest interval of continuous utilisation before the overflow, such that only instances with deadline less than or equal to t_2 are executed in $[t_1, t_2]$. Note that t_1 must be the release time of some periodic instance. Let $C_P(t_1, t_2)$ be the total computation time demanded by periodic tasks in $[t_1, t_2]$, which can be computed as:

Reminder:

$$C_P(t_1,t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i$$



where $\lfloor \ldots \rfloor$ denotes the lower-integer function.

(Note that for $task_1$ the response to the third request is not considered, therefore the assignment of the lower-integer is correct.)

Now, observe that:

$$C_P(t_1,t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} C_k = \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)\mu$$

Since a deadline is missed at t_2, $C_P(t_1,t_2)$ must be greater than the available processor time i.e. $(t_2 - t_1)$. Thus $(t_2 - t_1) < C_P(t_1,t_2) \leq (t_2 - t_1)\mu$ that is $\mu > 1$,

which is a **contradiction**, i.e. the original statement is **false**!

Combined Scheduling of hard RT and soft RT tasks:

The new material starts here!

Two rules are applied:

Rule#1: Every task should be schedulable with **average** execution and **average** arrival times.

Rule#2: Every hard RT task should be schedulable with **worst-case** execution and **worst-case** arrival time.

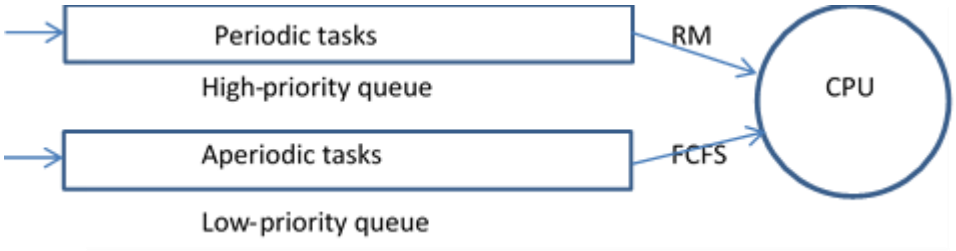
Combined Scheduling of periodic and aperiodic tasks: Fixed Priority Servers

We concentrate on hard RT systems, and soft aperiodic systems, but soft RT systems can also be considered.

The algorithms presented here rely on the following assumptions:

1. Periodic tasks are scheduled based on a fixed-priority assignment; here the RM algorithm;
2. All periodic tasks start simultaneously at time t=0 and $D_i = T_i$.
3. Arrival times of aperiodic requests are unknown;
4. When not explicitly specified, the minimum interarrival time of a sporadic task is assumed to be equal to its deadline.

Background Scheduling:



The major **advantage** of background scheduling is its **simplicity**.
Its **drawback** is that the response time of the aperiodic tasks can be **very large**. (**FCFS**=First-Come-First-Served.)

If the response time of the aperiodic tasks is critical, the so-called **server methods** give better result.
The **server method** provides processor time for the aperiodic tasks in a **separate way**.
The tool of this solution is the **server task**, which is scheduled together with the **periodic tasks**.

1. Polling Server (PS): The aperiodic requests are scheduled by the so-called **server task** (**S**), using the **server capacity** (T_S,C_S), and a **separated** scheduling mechanism.
Is there is no aperiodic request while the server task could run, the server task suspends itself, and its capacity will not be preserved!
The server task (according to RM) will have medium priority.

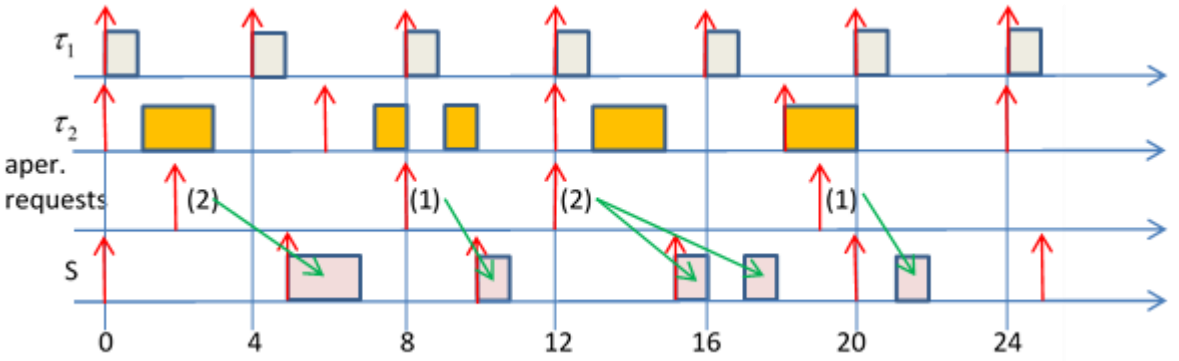
Example:
Let us have T_S=5, C_S=2.

	C	T
τ_1	1	4
τ_2	2	6

Assuming simultaneous start, the schedule will be the following:

Let us have T_S=5, C_S=2.

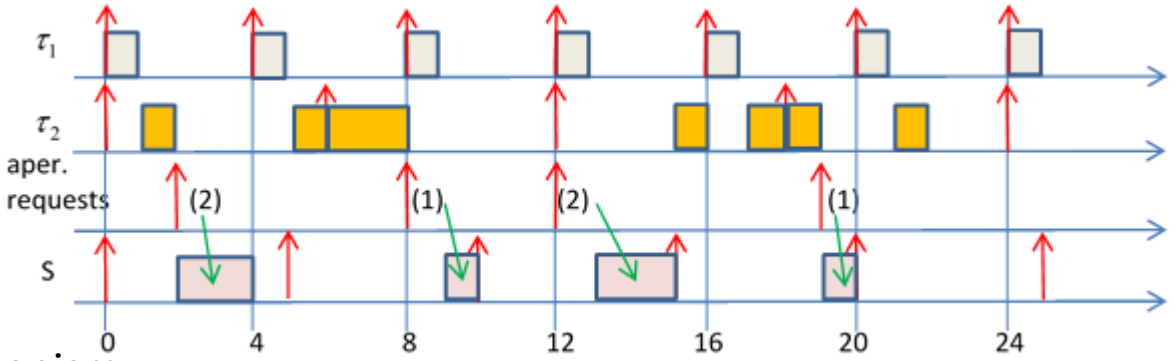
	C	T
τ_1	1	4
τ_2	2	6



In worst case situations, the fulfilment of the aperiodic request will occur only after an almost complete server period.

2. Deferrable Server (DS):

The aperiodic requests are scheduled with the help of the so-called server task (**S**) using the server capacity (T_S, C_S), and a separated scheduling mechanism.



If there is no aperiodic request while the server task could run, the run of the DS will be postponed, its capacity is preserved till the end of the period. **Example:** Previous one…
With this method, **much better response times** to aperiodic requests can be achieved.
(**Scheduling** of the server task **is the same** as previously using **RM** strategy.)

3. Priority Exchange Server (PE):

Like **DS**, the **PE** algorithm uses a periodic server (usually at a high priority) for servicing aperiodic requests. However, it differs from **DS** in the manner in which the capacity is preserved.

Example: The **PE** server has T_S=5, C_S=1.

The data of the normal tasks:

	C	T
τ_1	4	10
τ_2	8	20

T_S=5,
C_S=1.

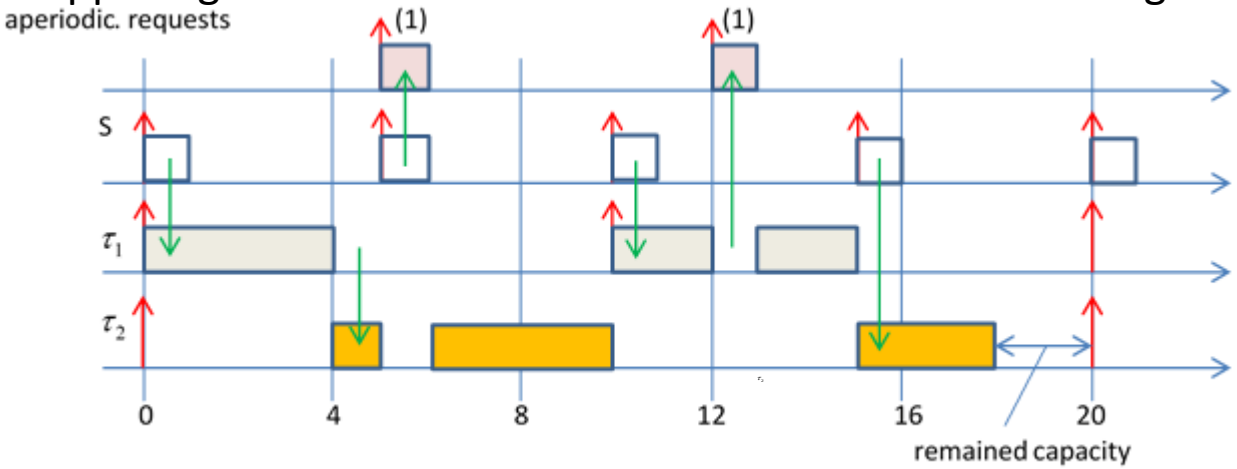
	C	T
τ_1	4	10
τ_2	8	20

The server task has the **highest** priority (**RM** strategy).

The processor utilization factor:

$$\mu = \frac{1}{5} + \frac{4}{10} + \frac{8}{20} = 1$$

Supposing **simultaneous** start the schedule is the following:



Since there is no aperiodic request to process, **the server capacity** is used by task τ_1.

As a consequence, task τ_2 can run earlier, i.e. the **server capacity** will be used at this level.

The server capacity of the **second period** is used **immediately.**

The server capacity of the **third period** is used by τ_1, but it is given back to fulfil the second aperiodic request.

The server capacity of the **fourth period** is used by τ_2.

Between [18-20], at the priority of τ_2, remaining server capacity is available, which could be used to serve **further aperiodic requests**.

Example: The PE server has T_S=5, C_S=1. The **server task** has the highest priority (RM strategy).

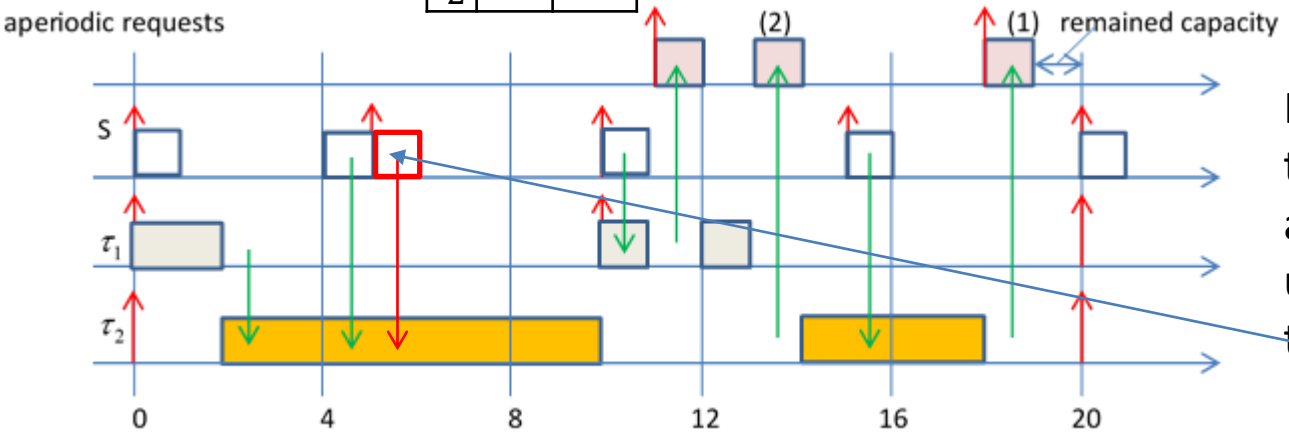
The further tasks to be scheduled:

	C	T
τ_1	2	10
τ_2	12	20

The processor utilization factor:

$$\mu = \frac{1}{5} + \frac{2}{10} + \frac{12}{20} = 1.$$

Supposing simultaneous start the schedule is the following:



In the figure we can see, that if we pass capacity to another task, then it can be utilized **at the priority** of the receiving task.

Correction:

At time instant **11** the first unit of the requested two can be found at $\boldsymbol{\tau_1}$, while the second at $\boldsymbol{\tau_2}$.
Therefore at **12** the execution of $\boldsymbol{\tau_1}$ is continued, and the aperiodic task should wait.

At 18 the aperiodic task will get processor time from $\boldsymbol{\tau_2}$.

Between [**19**-**20**], at the priority of $\boldsymbol{\tau_2}$, remaining server capacity is available, which could be used to serve further aperiodic requests.

4. Sporadic Server (SS): Like **DS**, differs from **DS** in the way it replenishes its capacity.
SS replenishes its capacity only after it has been **consumed** by aperiodic task execution.
The server capacity is replenished **one server period later** as the utilization has started.

Example: T_S=8, C_S=2. The further tasks to be scheduled are:

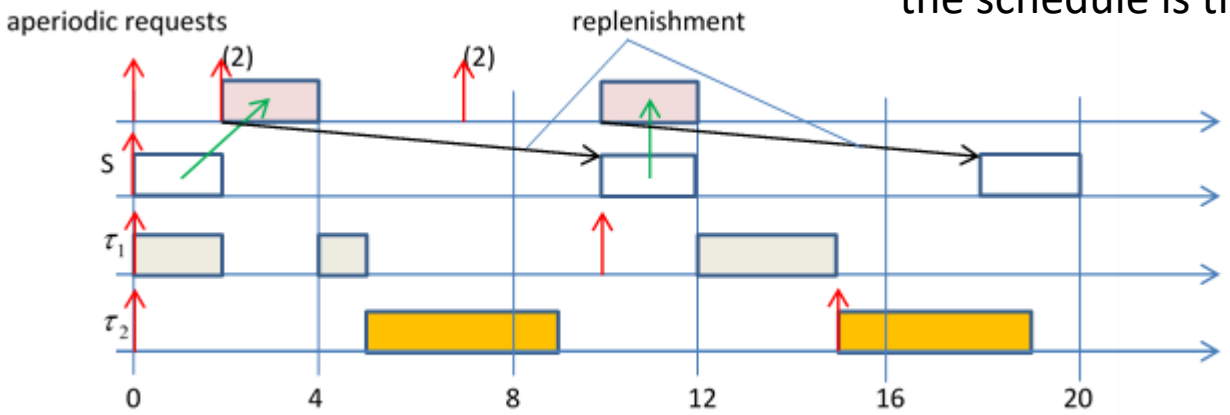
	C	T
τ_1	3	10
τ_2	4	15

T_S=8, C_S=2.

The server task has the highest priority.

Supposing simultaneous start the schedule is the following:

	C	T
τ_1	3	10
τ_2	4	15

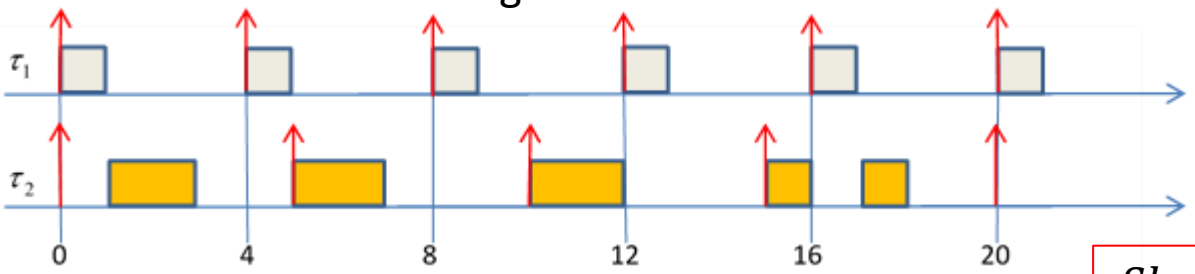


5. Slack stealing: This algorithm does not create a periodic server for the aperiodic service. Offers substantial improvements in response time over the previous methods.

Example:

	C	T
τ_1	1	4
τ_2	2	5

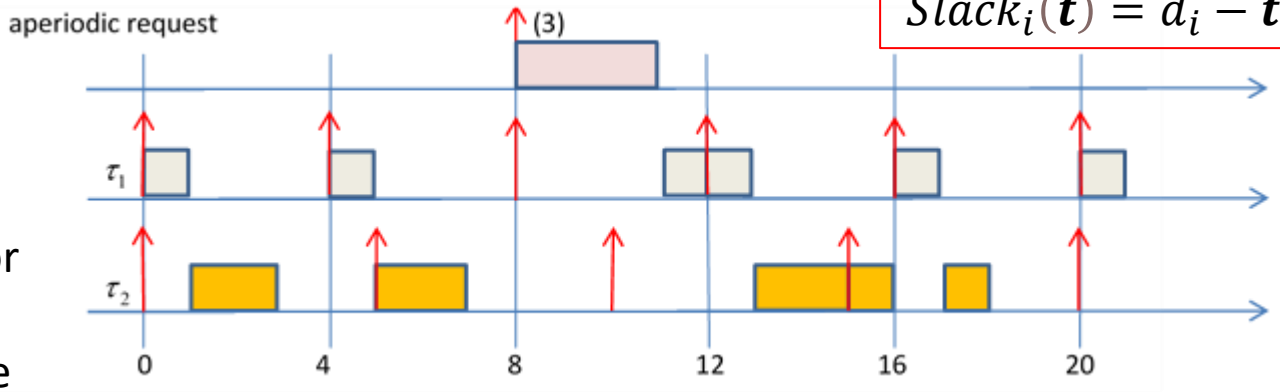
Normal **RM** scheduling:



If $\boldsymbol{C_i(t)}$ is the remaining computation time at time $\boldsymbol{t}$, then the slack of a task $\boldsymbol{\tau_i}$ is

$$Slack_i(\boldsymbol{t}) = d_i - \boldsymbol{t} - \boldsymbol{C_i(t)}$$

Upon arrival of aperiodic request, the slack is calculated, and this amount of processor time is given to the aperiodic task at the highest priority:



The **price**: larger implementation **complexity.**

6. Dual Priority Scheduling: Idea: there is **no benefit** in **early completion** of hard tasks.

Use three ready queues: **High, Middle** and **Low**. The **hard RT** tasks start running at **Low priority**. The **soft RT** and the **aperiodic** tasks run at **Middle priority.**

The hard RT tasks at approaching the so-called **promotion time** (X_i) before their **deadline** (D_i) are promoted and put in the **High** queue just to able to meet their **deadline**.

The **promotion time** can be calculated as follows: $X_i = D_i - R_i$ $(R_i = B_i + C_i + I_i)$

Obviously the three priority queues can be subdivided into further priority levels.

Comments: The server tasks introduced above were scheduled using the RM strategy. Similar solutions can be derived in the case of the EDF. These are **dynamic priority servers**.

Total Bandwidth Server (TBS):

This approach assigns a possible **earlier deadline** to each **aperiodic request**. This is done in such a way that the total utilization of the aperiodic load never exceeds a specified maximum value $\boldsymbol{\mu_S}$.

The name of the server comes from the fact that, when an aperiodic request enters the system; the **total bandwidth** of the server is immediately assigned to it, **whenever possible**.

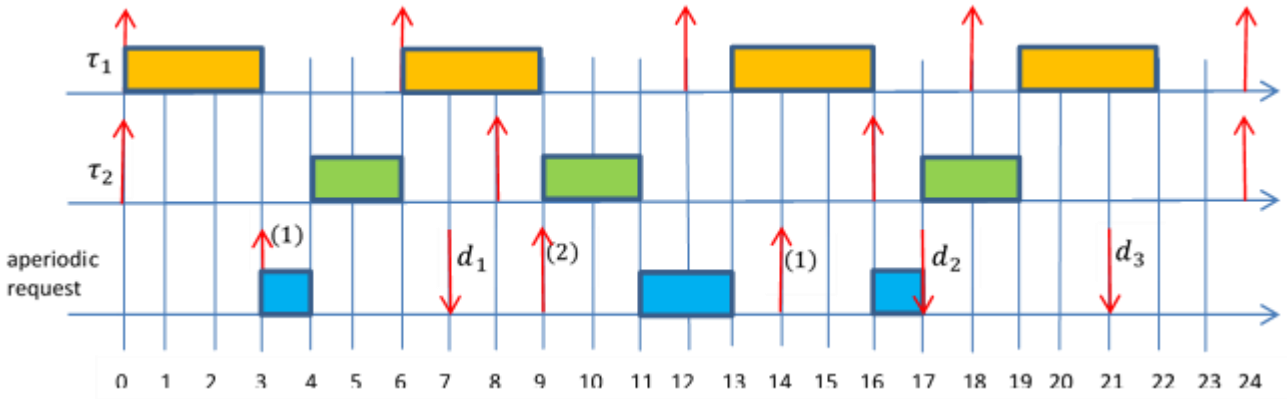
When the $\boldsymbol{k}$**-th** aperiodic request arrives at time $\boldsymbol{t = r_k}$, it receives a deadline:

$$d_k = max(r_k, d_{k-1}) + \frac{C_{ak}}{\mu_S},$$

where $\boldsymbol{C_{ak}}$ is the execution time of the request and $\boldsymbol{\mu_S}$ is the server utilization factor (that is, its **bandwidth**).

By definition $d_0 = 0$. In the deadline assignment rule the bandwidth allocated to previous aperiodic requests is considered through the deadline d_{k-1}. Once a deadline is assigned, the request is inserted into the ready queue of the system and scheduled by EDF as any other periodic instance. Implementation overhead is practically negligible.

The Figure below illustrates this method.



We have two periodic tasks:

$T_1 = 6ms, C_1 = 3ms$, and
$T_2 = 8ms, C_2 = 2ms$.

Consequently $\mu_P = 0.75$ and thus $\mu_S = 0.25$.

The first aperiodic request arrives at time $\boldsymbol{t = 3ms}$, and is serviced with deadline $\boldsymbol{d_1 = r_1 + C_{a1}/\mu_S = (3 + 1/0.25)ms = 7ms}$. Being this value the earliest deadline in the system, the aperiodic request is executed immediately.

The second request, which arrives at time $\boldsymbol{t = 9ms}$, receives a deadline $\boldsymbol{d_2 = r_2 + C_{a2}/\mu_S = (9 + 2/0.25)ms = 17ms}$, however this is not serviced immediately, because at time $\boldsymbol{t = 9ms}$ there is an active periodic task, $\boldsymbol{\tau_2}$ with a shorter deadline: $\boldsymbol{16\ ms}$. Finally, the third aperiodic request arrives at time $\boldsymbol{t = 14ms}$ and gets a deadline $\boldsymbol{d_3 = max(r_3, d_2) + C_{a3}/\mu_S = = (17 + 1/0.25)ms = 21ms}$. It does not receive immediate service, since at time $\boldsymbol{t = 14ms}$ task $\boldsymbol{\tau_1}$ is active and has an earlier deadline: $\boldsymbol{18\ ms}$.

It can be proved that if the processor utilization factor of the periodic tasks is $\boldsymbol{\mu_P}$, and that of the **Total Bandwidth Server** is $\boldsymbol{\mu_S}$, then this task set can be scheduled using EDF if and only if

$$\mu_P + \mu_S \leq 1.$$

Proof: If in every $[t_1, t_2]$ interval $\boldsymbol{C_a}$ is the total computation time of those aperiodic requests, which arrived at $\boldsymbol{t_1}$ or later, and served with deadlines less than or equal to $\boldsymbol{t_2}$, then $\boldsymbol{C_a \leq (t_2 - t_1)\mu_S}$, because

$$\boldsymbol{C_a} = \sum_{k=k_1}^{k_2} C_{ak} = \boldsymbol{\mu_S} \sum_{k=k_1}^{k_2} \left(d_k - max(r_k, d_{k-1})\right) \leq \boldsymbol{\mu_S} \left(d_{k_2} - max\left(r_{k_1}, d_{k_1-1}\right)\right) \leq \boldsymbol{\mu_S}(\boldsymbol{t_2} - \boldsymbol{t_1}).$$

After this, the proof of the schedulability test follows closely that of the periodic case.

Further examples of dynamic priority servers: Dynamic Priority Exchange Server (DPE), Dynamic Sporadic Server (DSS), Earliest Deadline Late Server (EDL) + their improved versions.

Schedulability if $D_i < T_i$:

Almost all the methods, statements and proofs discussed up till now cover cases where $\boldsymbol{D_i = T_i}$. If the deadline is earlier than the period, then the **priority can be assigned** according to the **deadlines**. One such a technique is the ***Deadline Monotonic*** **(DM)** algorithm, where the highest priority is assigned to the task having earliest deadline relative to the request time.

Obviously the condition $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$ can be a **sufficient** schedulability test, however, this is **not necessary**, and sometimes rather **pessimistic.**

Less pessimistic, if assuming simultaneous start (since concerning processor demand this is the worst case) for all the tasks we investigate the fulfilment of the condition $\boldsymbol{C_i + I_i \leq D_i}$.

Here $I_i = \sum_{\forall k \in hp_i} \left\lceil \frac{D_i}{T_k} \right\rceil C_k$. This condition is **sufficient** but **not necessary**.

The **necessary and sufficient** condition is given by the already discussed worst-case response time analysis:

$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k < D_i.$$

If the **EDF** strategy is applied while $D_i < T_i$, then the processor utilisation factor cannot be used. Instead the so-called **processor demand approach** can be suggested. First this will be introduced for the $D_i = T_i$ case.

In general, within **an arbitrary interval** $[t, t+L]$ the **processor demand** of a task τ_i is the time needed to become completed till the time instant $t+L$ or before.

In the case of such periodic tasks, which start running at $t = 0$, and for which $D_i = T_i$, the total processor time in any $[0, L]$ interval is:

$$C_p(0,L) = \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$$

Statement: A periodic task set can be scheduled by EDF iff for any $L > 0$:

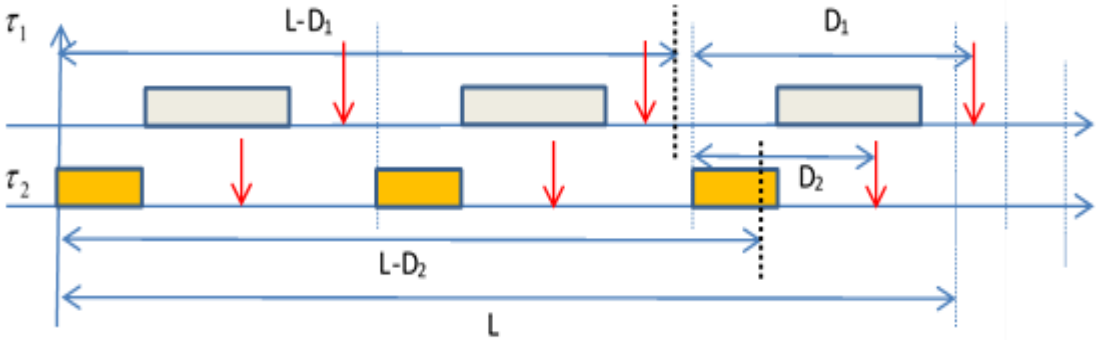
$$L \geq \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k \quad (*)$$

Proof: On one hand, since $\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, therefore

$$L \geq \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k \geq \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$$

On the other, if $\mu > 1$, then there exists such $L > 0$, for which (*) does not hold, since if e.g. $L = lcm(T_1 T_2 \ldots T_n)$, then:

$$L < \mu L = \sum_{k=1}^{n} \left(\frac{L}{T_k} \right) C_k = \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k$$



If $D_i < T_i$, then the calculation of $C_p(0, L)$ is different.

For simplicity let us have the same period but different deadlines:

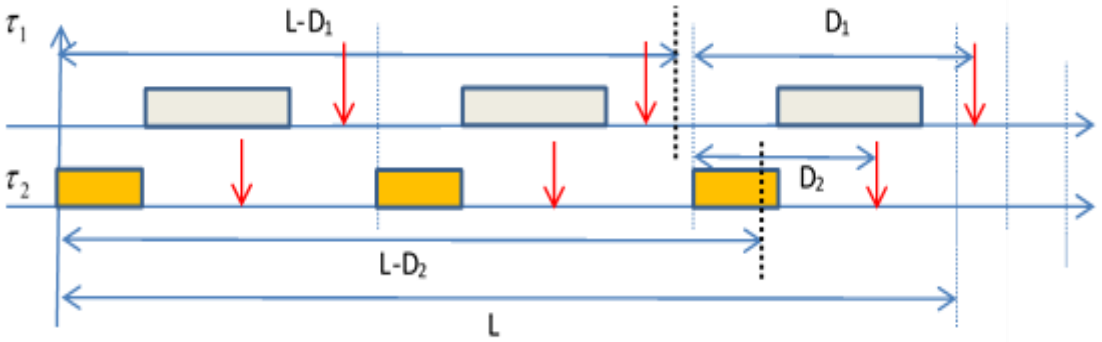
Since deadline of the third period is out of the range of the interval of length L, the processor demand of τ_1:

$$C_1(0,L) = \left\lfloor \frac{L}{T_1} \right\rfloor C_1$$

while for τ_2 this can be given by

$$C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2$$

?



$$C_1(0,L) = \left\lceil \frac{L}{T_1} \right\rceil C_1$$

$$C_2(0,L) = \left(\left\lfloor \frac{L}{T_2} \right\rfloor + 1 \right) C_2$$

Using the figure, it is easy to understand that the two cases can be handled with a single formula of the form:

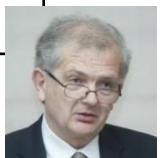
$$C_i(0,L) = \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

Statement: With this formula: A periodic task/set can be scheduled by EDF if and only if for every $L > 0$

$$L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k$$

Summary:

	$D_i=T_i$	$D_i<T_i$
static priority	**RM** processor utilisation approach $\mu \leq n\left(2^{\frac{1}{n}} - 1\right)$	**DM** response time approach for $\forall i\ R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k \leq D_i$
dynamic priority	**EDF** processor utilisation approach $\mu \leq 1$	**EDF** processor demand approach $\forall L > 0 \quad L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k$



Extensions to the response time calculation:

1. Cooperative scheduling: At a given point of the task execution it might be a requirement the completion of the task **as early as possible**.

This can be achieved if the **pre-emption** of the task **is prohibited** till the end it's run.

If this takes time F_i, then the response time can be written in the form of $R_i = R_i' + F_i$, where

$$R_i' = B_i + C_i - F_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i'}{T_k} \right\rceil C_k$$

In this case the last part of the execution if runs, it will run on the highest priority.

2. Fault tolerance : exception handlers, recovery blocks, etc.: + computation time is needed.

C_i^f extra computation time for every task. In case of single fault:

$$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{max}_{k \in hep_i} C_k^f$$

Please note: hep_i !

For $\boldsymbol{F}$ faults:

$$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{max}_{k \in hep_i} \left(F C_k^f \right)$$

$$R_i = B_i + C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k + \underbrace{max}_{k \in hep_i} \left\lceil \frac{R_i}{T_f} \right\rceil C_k^f$$

If T_f denotes the shortest inter arrival time between two faults, then:

3. The additional time demands of the clock handler and that of the context switches:

In many applications the scheduler is triggered by a clock interrupt (**tick scheduling**).

In this case the response time should be increased by the **worst-case time difference** of the arrival and the clock tick. If the time of the arrival is not measurable, then the time between two clock ticks is the **correcting value.**