Embedded Information Systems

2. Scheduling

September 29, 2020

Reminder:

Quantities and variables in real-time systems

A periodically updated RT image is called ***parametric***, or **phase-insensitive**, if

$$d_{accuracy} > \left(d_{update} + WCET_{message\ forwarding}\right).$$

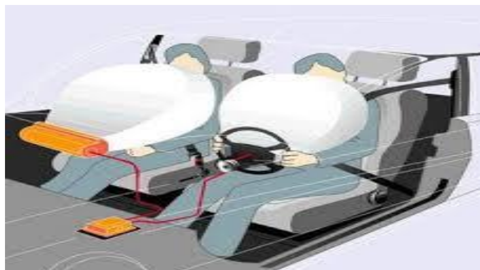
The parametric RT image at the receiver node can be utilised without further investigations, because the updated value arrives within the accuracy interval.

A periodically updated RT image is called ***phase-sensitive,*** if

$$WCET_{message\ forwarding} < d_{accuracy} < \left(d_{updating} + WCET_{message\ forwarding}\right).$$

In this case it is not sure that the update arrives within the accuracy interval: we must check the time conditions and possibly wait for the update.

Characterisation of HRT and SRT systems:



characteristic	hard real-time	soft real-time
response time	hard required	soft desired
peak-load performance	predictable	degraded
control of pace	environment	computer
safety	often critical	non-critical
size of data files	small/medium	large
redundancy type	active	checkpoint-recovery
data integrity	short-term	long term
error detection	autonomous	user assisted

Reminder:

Event triggered and time triggered systems

The **event triggered** systems execute the program associated with the event immediately after the arrival of the request. With this approach, we can get good response times, but if the number of (almost) simultaneous events increase, the throughput/capacity of the system might be insufficient, therefore, to meet the deadlines will be impossible.

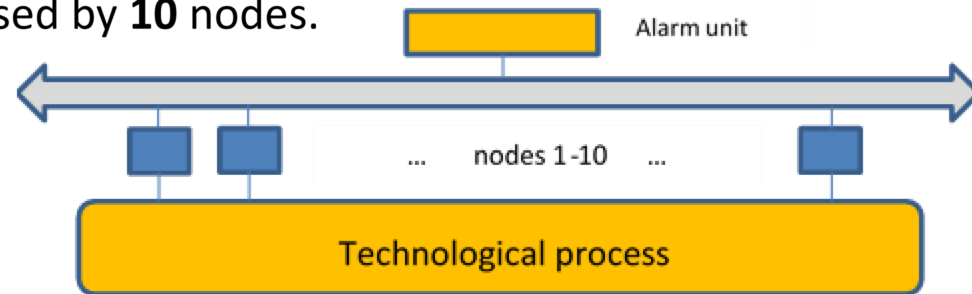
Within the **time-triggered** systems a separate timeslot is assigned to every task in design time, Thus, if the response times are a priori known, the program execution can be guaranteed.

Example: A technological process is supervised by **10** nodes.

Each node monitors **40** binary signals (alarm signals, e.g. limit crossings, etc.).

The communication is solved via a **bus**.

A system-level **alarm unit** is also connected.



The speed of the bus is **100 kbit/s**.

The monitoring nodes should send an alarm message to the alarm unit within **100 ms.**

Event triggered operation: ET/CAN protocol is applied. The shortest message is **one byte.**

According to the protocol the message will contain: **44 bits** overhead, **1-byte** data, **4-bit** length inter-message gap. The total size **is 56 bits.**

100 kbit/s means, that within **100 ms 10 000 bits** can get through.If the messages are of **56 bits**, then 10 000/56 ~ **180** messages can arrive to the alarm unit **within the specified time.**

Since **180 < 400**, therefore it is not possible to send all the possible changes, the communication channel for **~180** simultaneous messages will **completely saturate.**

Time triggered operation: TT/CAN protocol is applied. The nodes periodically send all the signalling bits to the alarm unit. Reminder:

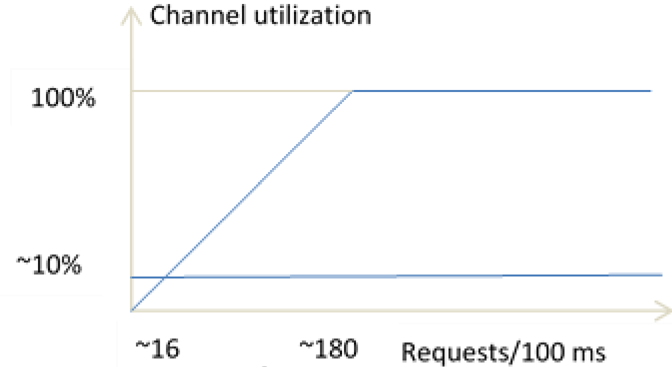
This can be performed for every **40** binary signals using a single message.

According to the protocol the message will contain: **44 bits** overhead, **5-byte** data, followed by a **4-bit** length inter-message gap. The total size is **88 bits**.

100 kbit/s means, that within **100 ms 10 000 bits** can get through. If the messages are of **88 bits**, then 10 000/88 ~ **110** messages can arrive to the alarm unit **within the specified time**.

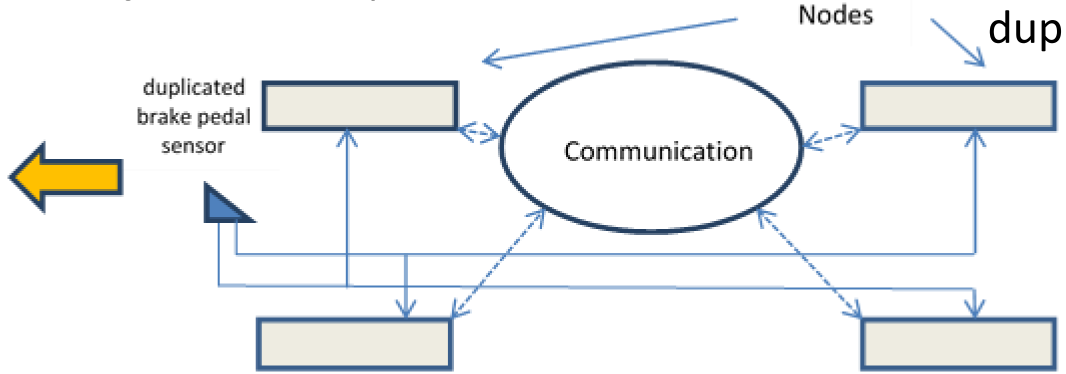
Since **110 > 10**,

thus all the signalling bits will arrive to the alarm unit, and what is more, at a load level of **~ 10%.**



The importance of agreement protocols

Example: brake-by-wire:



In this example, for safety reasons, duplicated brake pedal sensors are applied.

The brakes of each wheel have separate control nodes.

The nodes inform each other about their knowledge of the actual sensor value, and calculate the braking force.

If a node is violated and fails, the corresponding wheel will run free automatically, and braking force will not be provided.

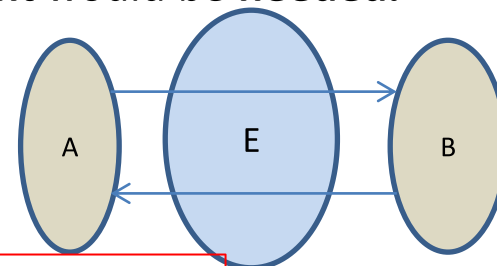
Reminder:

The other three nodes, after observing this situation, recalculate the braking forces, and will brake safely.

In distributed systems there are several situations where run-time agreements are needed: *time synchronisation, consistency of distributed states, distributed mutual exclusion, distributed transactions, distributed completion, distributed election, etc.*

A further problem is that even **in case of errors**, **an agreement** would be **needed.**

This is not always possible.



Example: Two armies' problem:

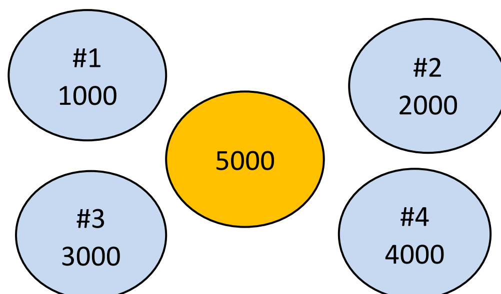
The allied armies, **A** and **B** together have more soldiers than the enemy (**E**), but separately each has less.

What we can do is to increase the probability of successful agreement.

Impossibility Result: It can be proven by **formal methods**, that to reach to an **agreement** of two or more **distributed** units in **limited time**, and through an **asynchronous medium**, which is **lossy**, cannot be **guaranteed**.

Agreement in case of Byzanthine errors: **Example**: Synchronisation of clocks: A, B, C and D.

To filter out a node with Byzanthine error is possible only if at least **3*k*+1** nodes participate in the synchronization, where ***k*** is the number of nodes having Byzanthine error.



The generals of the "blue" armies **try to agree** the total number of soldiers available for **a joint action**.

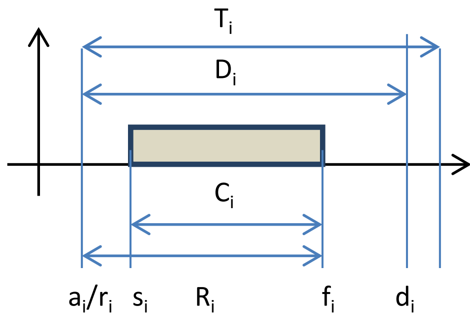
In the meantime, it turns out that **one of the generals** sends **wrong information.**

The generals of the allied armies inform each other about the number of the available soldiers.

Reminder:

2. Scheduling

Problem: The processors should execute **various tasks** with **different timing** requirements. The timing conditions can be interpreted using the following figure:



Here $\mathbf{a_i}$ or $\mathbf{r_i}$ is the **arrival/release/request time**, $\mathbf{s_i}$ is the **start time** of execution, its **finishing time** is $\mathbf{f_i}$, $\mathbf{d_i}$ stands for **deadline**, T_i is the **period time**, $\mathbf{D_i=d_i-a_i}$ is the **deadline** relative to the **request time**, $\mathbf{C_i}$ stands for **computation time,** and $\mathbf{R_i=f_i-a_i}$ is the **response time**.

1. Periodic scheduling: this is the simplest method: in design time fix time slots are assigned for the completion of the periodic requests, and this is repeated periodically.

The assignment is typically **clock-driven**; therefore these types of scheduling are called **time-triggered.** They have **different** versions, but what is **common**: the decisions concerning schedules are made **in design-time**, thus their run-time **overhead is low**.

A further feature is that **the parameters** of the **HRT** tasks **are known** and **fixed** in advance.

Example: To each task there is assigned a frame of **10 ms**. **4 functions** are implemented:

The first function operates with a periodicity of **50 Hz**, i.e. it receives **10 ms** in every **20 ms**.

The second function operates with a periodicity of **25 Hz**, i.e. it receives **10 ms** in every **40 ms**.

The third function at a rate of **12.5 Hz**, i.e. in every **80 ms** receives **10 ms.**

The fourth function at a rate of **6.25 Hz**, i.e. in every **160 ms** receives **10 ms.**

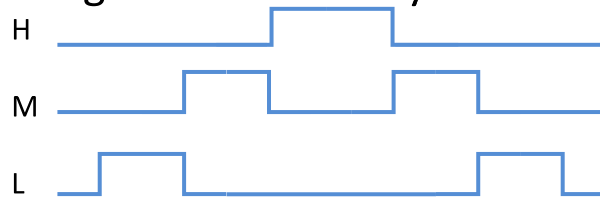
Obviously, the assignment of the frames can be different; it can be done only in design time.

2. Time-shared/round-robin scheduling: … **3. Priority based scheduling: …**

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:

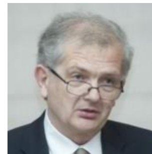
$$R_i = C_i + I_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

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).



It supposes priority assignment according to the **deadlines**: tasks with larger D_i will have lower priority. Only such cases are considered, where $D_i \leq T_i$. The method is suitable both for **periodic** and **sporadic** tasks.

Periodic task: is characterised by known and fixed period T_i.

Sporadic task: the requests are **not periodic**, but there is a known and fixed T_i value that is the minimum time between two subsequent requests.

Aperiodic task: the requests are **not periodic**, and there is no specified T_i between two requests, i.e. a request can be followed **immediately** by a second request.

Obviously in the case of **aperiodic tasks** the DMA method **cannot be applied**.

It might be important to emphasize, that by the application of the DMA method not the response time but the **worst-case response time** will be derived.

Example: A system with four tasks can be characterized with the following time values (the time is measured in **milliseconds**):

Task	T	C	D
1	250	5	10
2	10	2	10
3	330	25	50
4	1000	29	1000

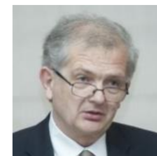
The priority order corresponds to the order of the tasks in the list.
If the deadline values are equal **secondary conditions** are used to decide **priority order**.

In the example, the first task has higher computation time, i.e. its laxity is smaller, therefore the higher priority might be a better choice. Let us calculate the **worst-case response time** of task **3.**

The **iterative procedure**:

Step	R^n	I	R^{n+1}
1	0	0	25
2	25	5+3*2	36
3	36	5+4*2	38
4	38	5+4*2	38

Comments:
38<50, thus task 3 will meet the deadline also under worst case conditions.



Note that the data of task 4 were not utilized at all. They are not required!

Note that the tasks up till now were considered **independent** from each other!

However, if they are not independent, then they are **communicating**. In this case might happen, that higher priority tasks should wait for data provided by lower priority.

This **additional waiting** time will increase both the response time and its worst-case version!

Response time calculation for periodic and sporadic tasks: Example:

An embedded system, devoted to executing requests of four periodic/sporadic tasks and one periodic/sporadic interrupt, has the following parameters (time values are in ms):

Task	T	C	D
i_1	10	0.5	3
τ_1	3	0.5	3
τ_2	6	0.75	6
τ_3	14	1.25	14
τ_4	50	5	50

Let us calculate the **worst-case** response time of task $\boldsymbol{\tau_4}$ using the **iterative** procedure!

The **interrupt** will be served on the **highest priority**, otherwise the priority level of the tasks follows the **deadline monotonic** assignment.

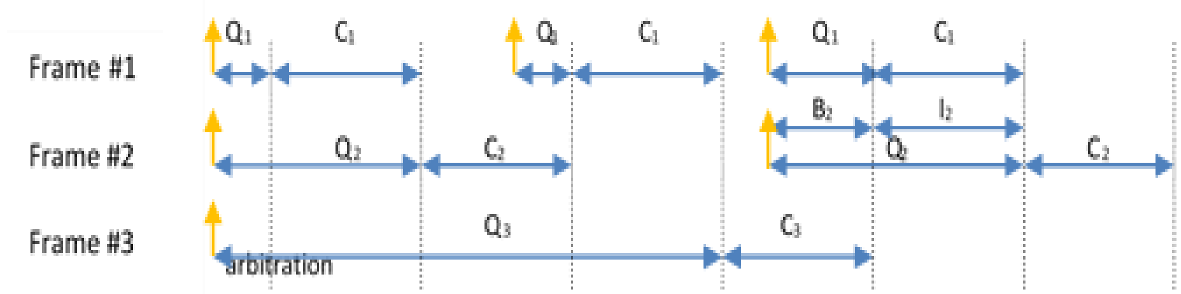
Step	R^n	I	R^{n+1}
1	0	0	5
2	5	0.5+1.0+0.75+1.25	8.5
3	8.5	0.5+1.5+1.50+1.25	9.75
4	9.75	0.5+2.0+1.50+1.25	10.25
5	10.25	1.0+2.0+1.50+1.25	10.75
6	10.75	1.0+2.0+1.50+1.25	10.75

Since **10.75<50**, the deadline is met. **Comment:**

The different versions of the **DMA analysis** are widely used for worst-case response time analysis **to optimize products** concerning the necessary **clock frequencies**/bandwidths to increase **noise immunity** and **reduce costs**.

(Volvo Corporation has introduced this technique already in 1995, first regarding S80.)

Example: A modified version of DMA method can be used also if the operation is not pre-emptive, i.e. when the running task will not be pre-empted. ***Response time analysis of the priority-based CAN bus.*** The key features of the communication through the **CAN** (Control Area Network, ISO 11898, Bosch) bus:



Calculation of the response time:

$$R_i = C_i + Q_i$$

where

$$Q_i = B_i + \sum_{\forall k \in hp_i} \left\lceil \frac{Q_i}{T_k} \right\rceil C_k$$

Message	T [ms]	C[ms]
1	3	1.35
2	6	1.35
3	10	1.35
4	30	1.35
5	40	1.35
6	40	1.35
7	100	1.35

The messages are **periodic**, and their priority is **decreasing** from the top. The requests are **asynchronous.** The **7th** message is related to braking, it should be received in $\mathbf{100}\ \boldsymbol{ms}$. The iteration for the waiting time:

Step	Q^n	I							B	Q^{n+1}
		1	2	3	4	5	6	Sum		
1	0	-	-	-	-	-	-	0	1.35	1.35
2	1.35	1	1	1	1	1	1	8.1	1.35	9.45
3	9.45	4	2	1	1	1	1	13.5	1.35	14.85
4	14.85	5	3	2	1	1	1	17.55	1.35	18.9
5	18.9	7	4	2	1	1	1	21.6	1.35	22.95
6	22.95	8	4	3	1	1	1	24.3	1.35	25.65
7	25.65	9	5	3	1	1	1	27	1.35	28.35
8	28.35	10	5	3	1	1	1	28.35	1.35	29.7
9	29.7	10	5	3	1	1	1	28.35	1.35	29.7

The worst-case waiting time: $29.7\ ms$.

The worst-case response time: $29.7ms + 1.35ms = 31.05\ ms$.

This value is smaller than the specified $\mathbf{100}\ \boldsymbol{ms}$: the deadline is met.

Schedulability, schedulability tests:

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

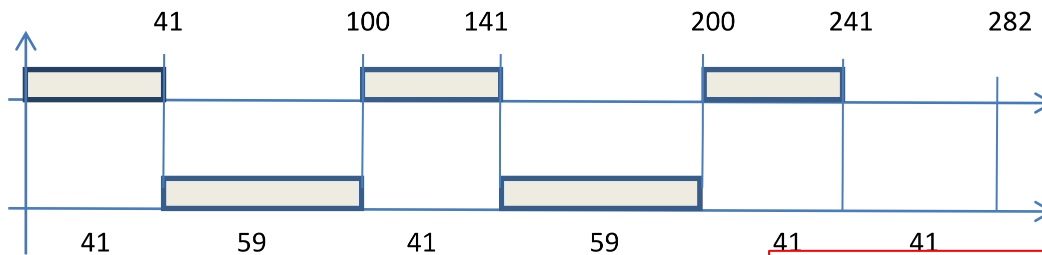
$\mu = 2\left(\sqrt{2} - 1\right)$, and for arbitrary i :

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 $T_1 = 100\ ms$, $C_1 = 41\ ms$, $T_2 = 141\ ms$, $C_2 = 59\ ms$.

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



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

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

Comments:

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.**

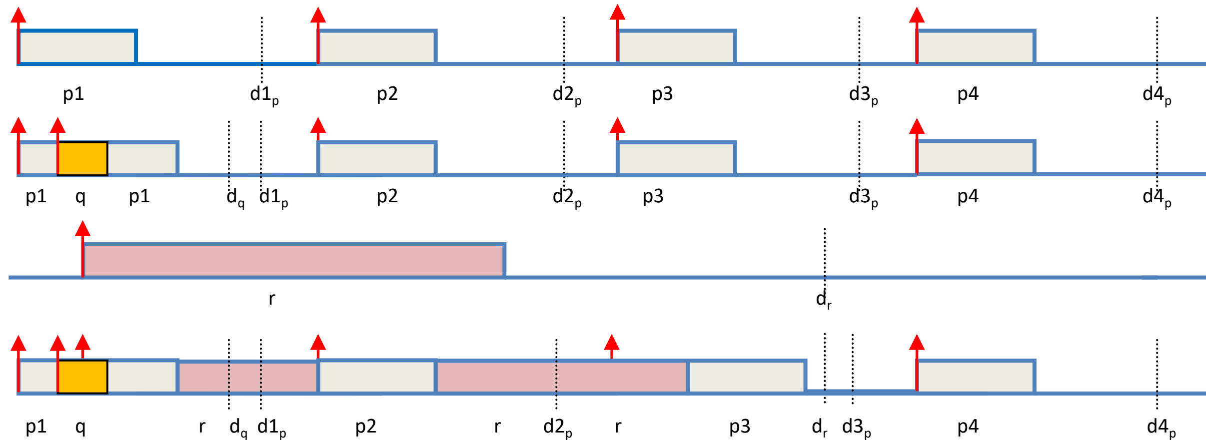
2. Non-zero phase start is **advantageous** from scheduling point of view!

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.

Earliest Deadline First (EDF) strategy: We assume that 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 $\boldsymbol{\mu} \leq \mathbf{1}$, i.e. **100%** processor utilisation is possible.



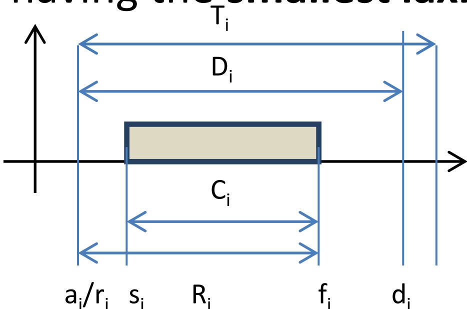
Row #1: requests and deadlines of task **p**

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 $\boldsymbol{\mu} \leq \mathbf{1}$, i.e. **100%** processor utilisation is possible.

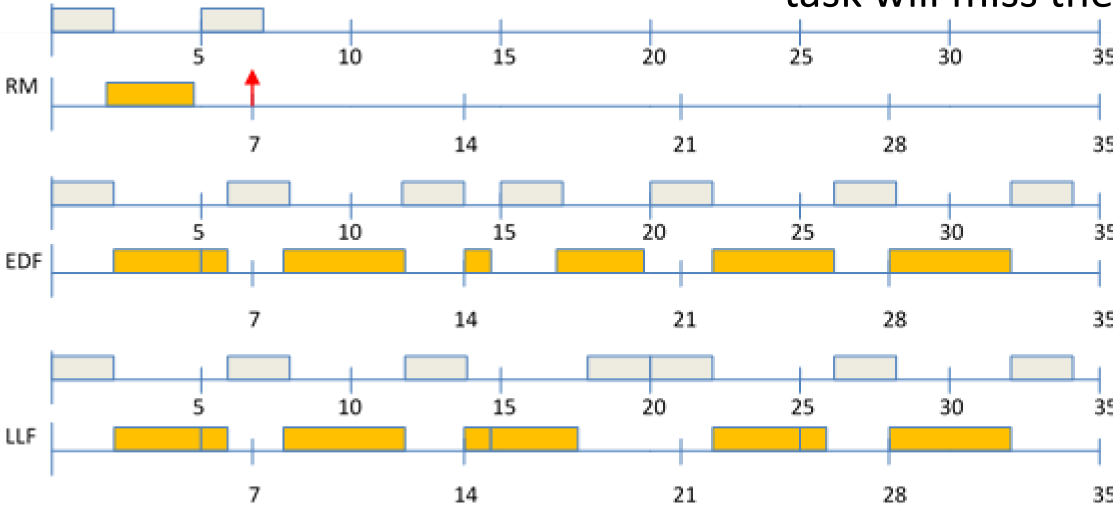
The EDF and LLF strategies are applicable also for **aperiodic** tasks, but since the processor utilisation factor can only be interpreted in a different way, the sufficient condition above cannot be used.

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:

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

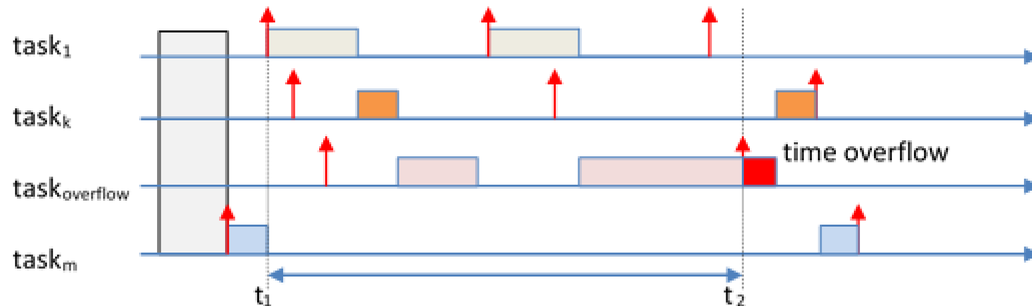
$$\mu = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

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

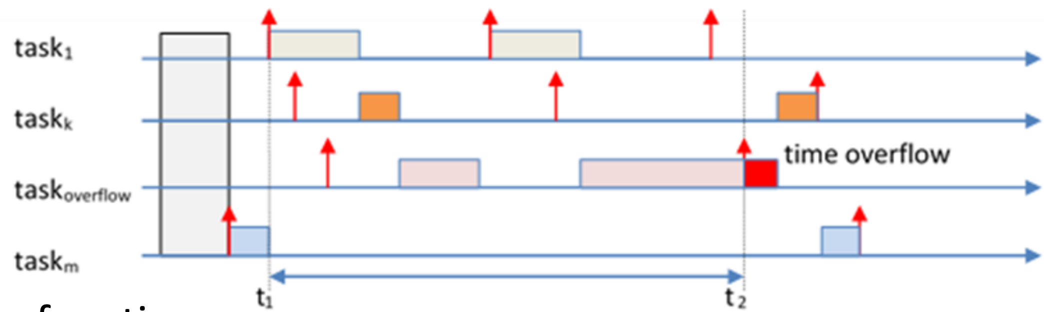
2. *If part*: We show the sufficiency by contradiction.
Assume that the condition $\boldsymbol{\mu < 1}$ is satisfied **and** yet the taskset is **not schedulable**.
The next figure helps to understand the proof. Here we can see the schedule of periodic tasks according to EDF.



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:

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



where $\lfloor \dots \rfloor$ 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**!