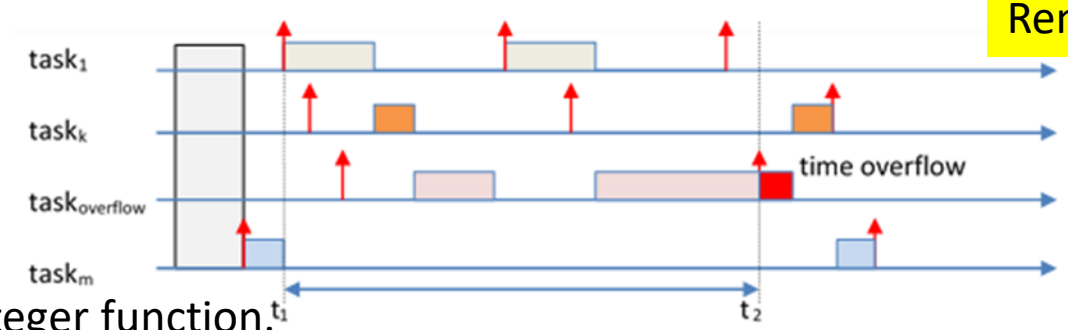
Embedded Information Systems

2. Scheduling (cont.)
3. Memory management

October 13, 2020

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:

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

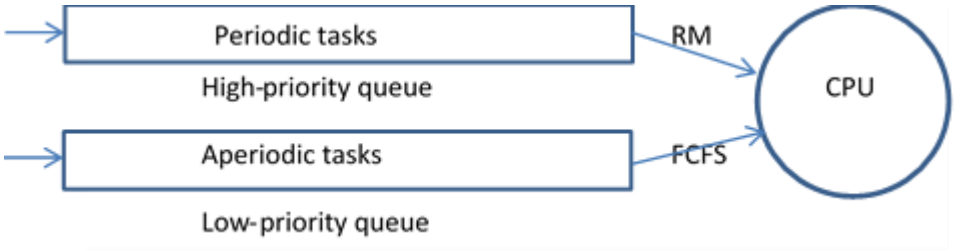
We concentrate on hard RT systems, and soft aperiodic systems, but soft RT systems can also be considered.

Reminder:

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.

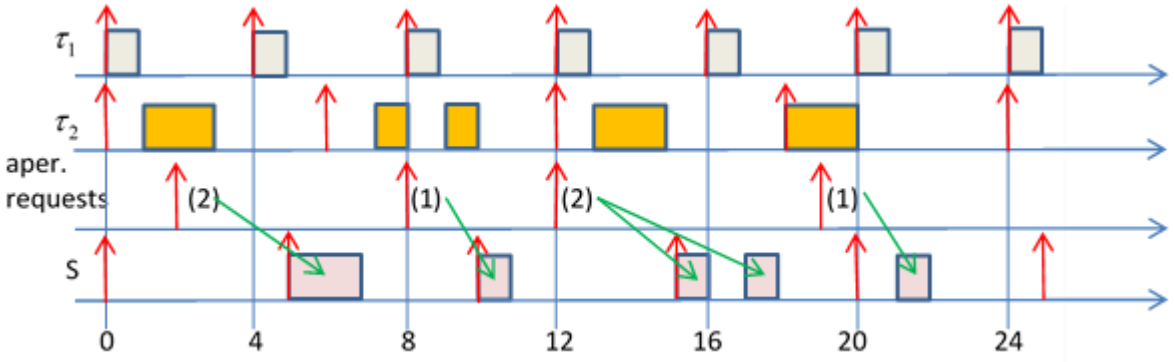
	C	T
τ_1	1	4
τ_2	2	6

Assuming simultaneous start, the schedule will be the following:

Reminder:

Let us have T_S=5, C_S=2

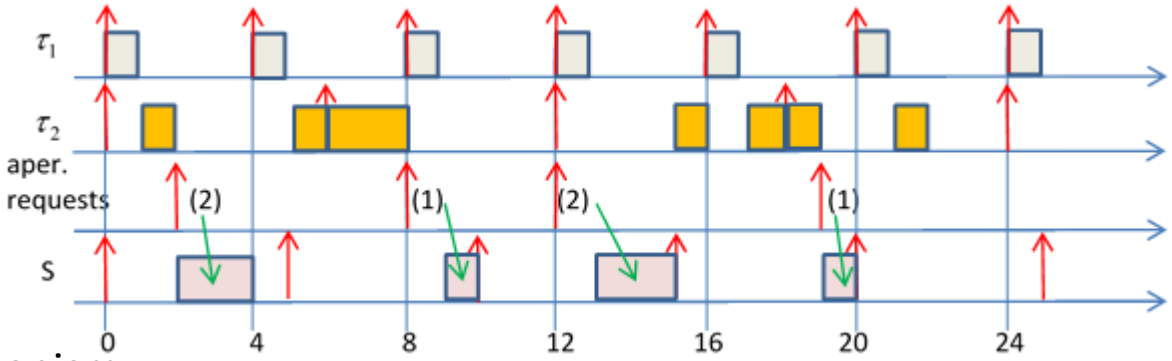
	C	T
τ_1	1	4
τ_2	2	6



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

τ_2 T_S=5, C_S=1.

	C	T
τ_1	4	10
τ_2	8	20

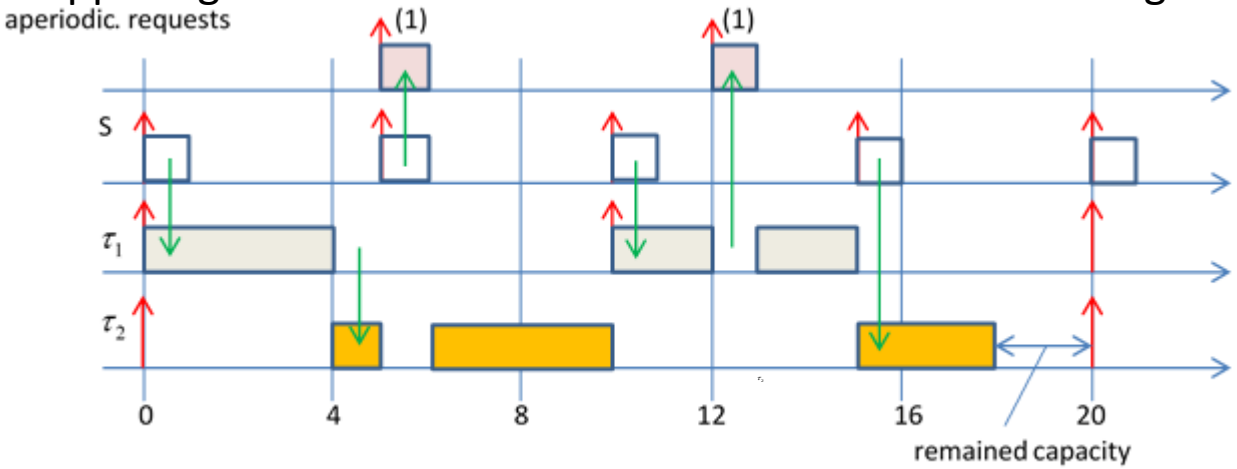
The server task has the **highest** priority (**RM** strategy).

The processor utilization factor:

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

Reminder:

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 further tasks to be scheduled:

	C	T
τ_1	2	10
τ_2	12	20

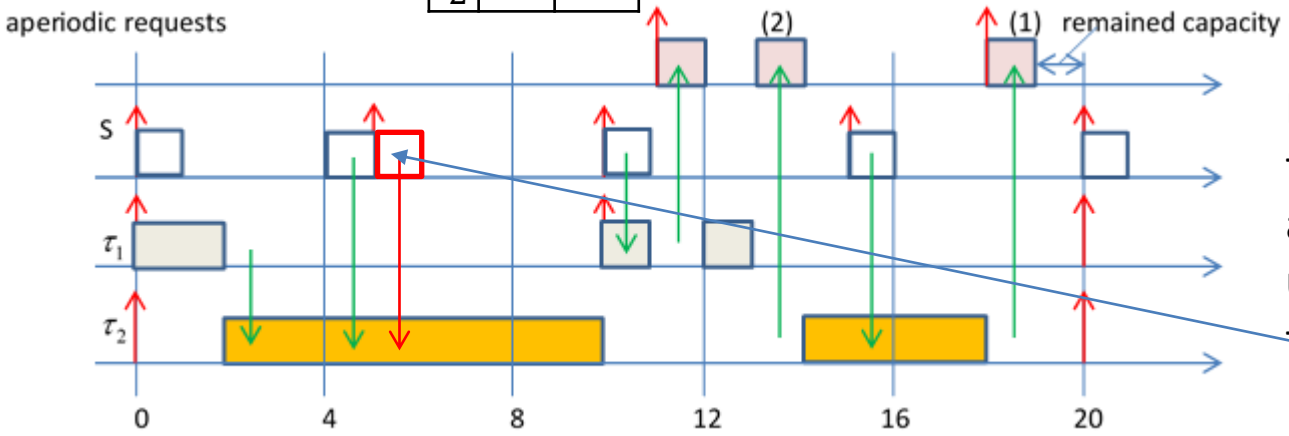
The processor utilization factor:

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

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

Reminder:

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 τ_1, while the second at τ_2. Therefore at **12** the execution of τ_1 is continued, and the aperiodic task should wait.

At 18 the aperiodic task will get processor time from τ_2.

Between [**19**-**20**], at the priority of τ_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:

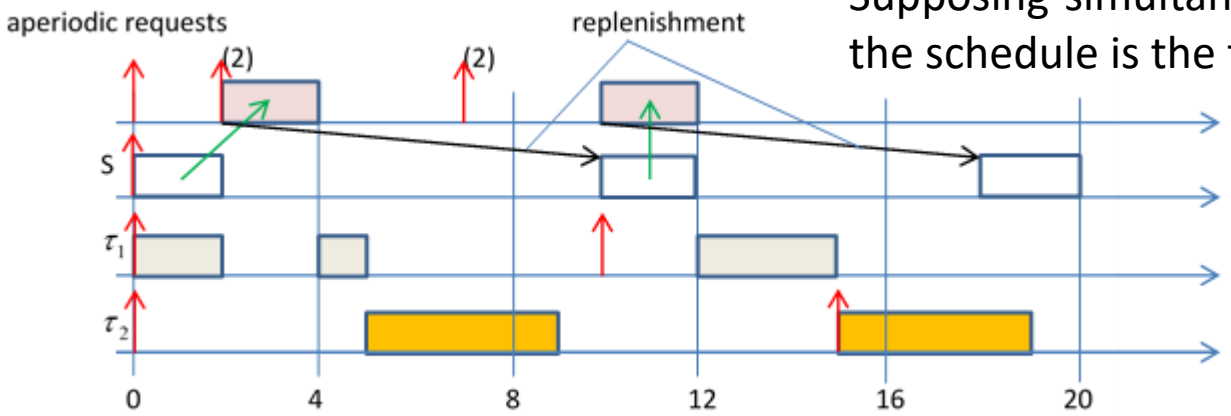
	C	T
τ_1	3	10
τ_2	4	15

Reminder:

T_S=8, C_S=2.

The server task has the highest priority.

	C	T
τ_1	3	10
τ_2	4	15



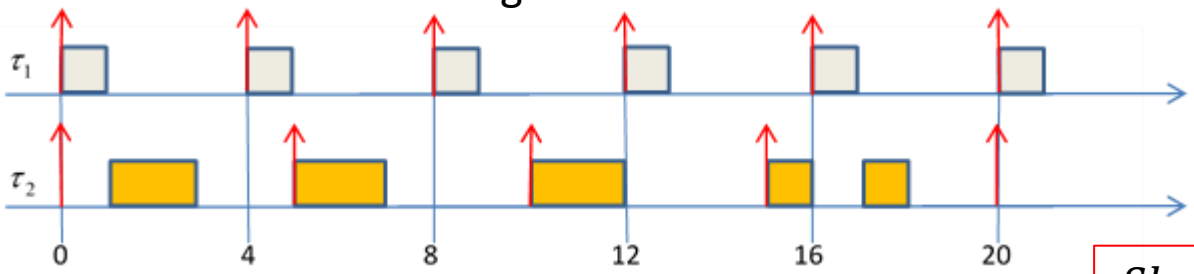
Supposing simultaneous start the schedule is the following:

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

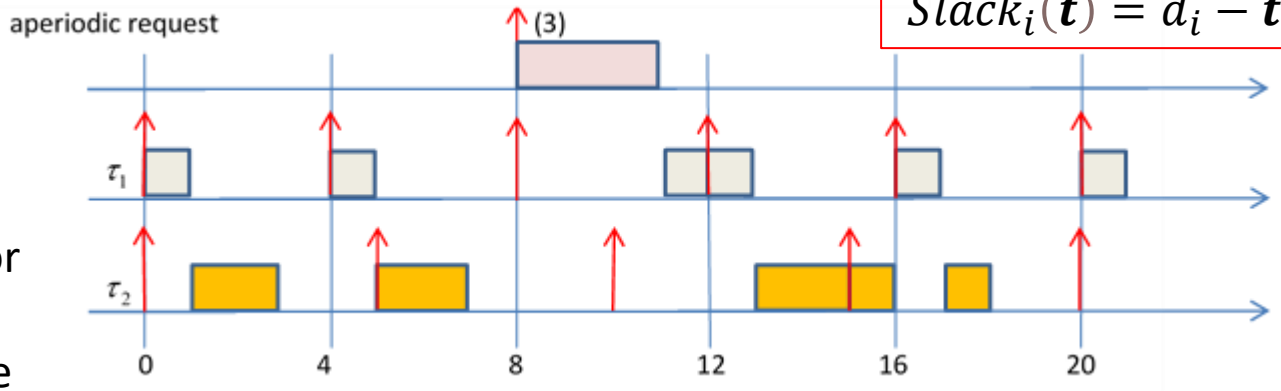
Normal **RM** scheduling:



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

Reminder:

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:

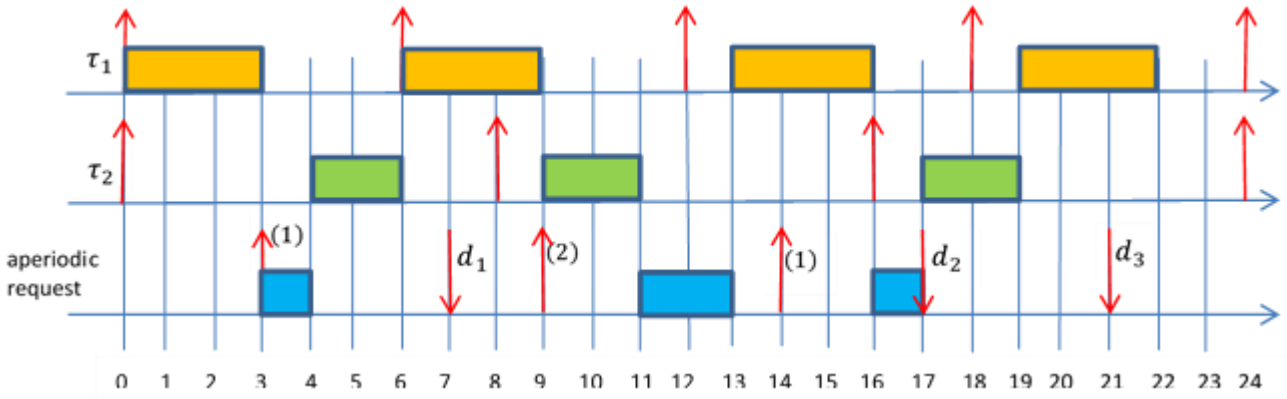
$$d_k = max(r_k, d_{k-1}) + \frac{C_{ak}}{\mu_S},$$

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.

Reminder:

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

Reminder:

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

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 $D_i = T_i$. 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 $C_i + I_i \leq D_i$.

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:

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

Reminder:

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 $L > 0$:

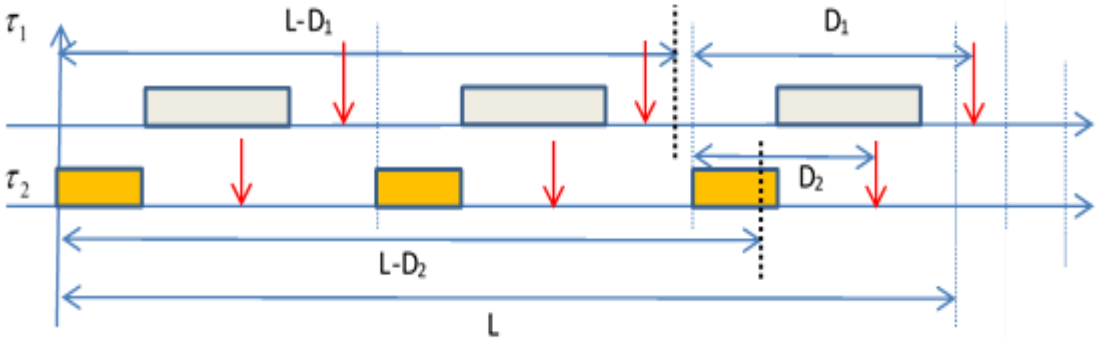
$$L \geq \sum_{k=1}^{n} \left\lfloor \frac{L}{T_k} \right\rfloor C_k \quad (*)$$

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:

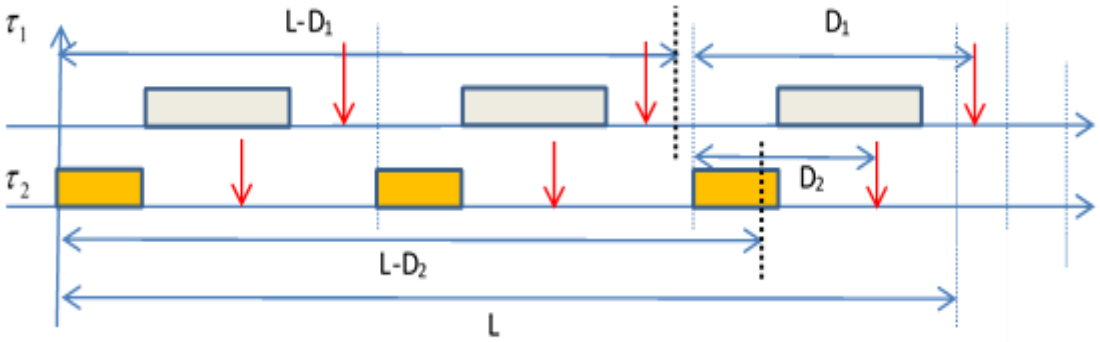
Since deadline of the third period is out of the range of the interval of length L, the processor demand of τ_1:

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

while for τ_2 this can be given by

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

?



Reminder:

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

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

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

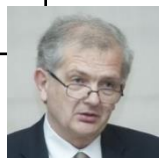
$$C_i(0,L) = \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$$

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:

Reminder:

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:

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

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

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

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.**

Reminder:

If the **scheduler decides** a task to run, **then first the registers of the processor should be saved, after this the context** of the new task should be loaded into the registers, and then comes the **execution** of the task.
The response time should be increased by time of this „**context switch**”.

The computation time of the **higher priority tasks**, which pre-empt the execution of an actual task, **should be increased** by the time needed to perform **context switching**, as well.

Scheduling if the tasks are not independent: Resource Access Protocols

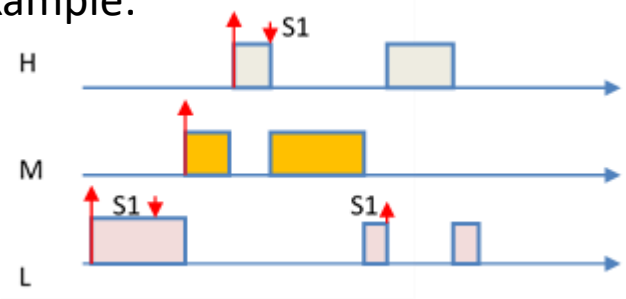
Except for the **time-sharing systems**, where the processor's capacity is shared among **independent users**, for most of the applications the runs of the different tasks **are not completely independent.** Tasks are **communicating** with each other, **exchange data**, they are **waiting for results** from other tasks, they use **common resources**, and it can happen, that **higher priority** tasks **are blocked by** runs of **lower priority tasks**.

Let us recall the illustration of the priority-based scheduling!

If here task L would use such a resource, which is later also used by task H, then it might happen that task H should wait until the resource will be released.

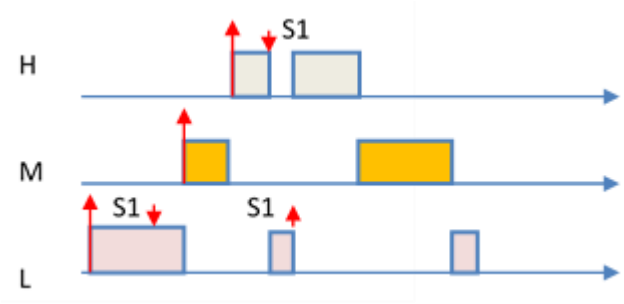
This type of waiting is called **blocking**, because **lower priority** task **forces higher priority** task to **wait**.

Example:



This situation is called **priority inversion** because seemingly the priorities of task M and H are inverted.

Priority Inheritance Protocol (PIP): To avoid priority inversion, task **L** should dynamically inherit the priority of task **H** upon its request to enter the critical section. Thus, task **L** can complete the critical section much earlier and unlock semaphore **S1**. The inherited priority is called **dynamic priority**. After unlocking semaphore **S1** the static priority will be restored.



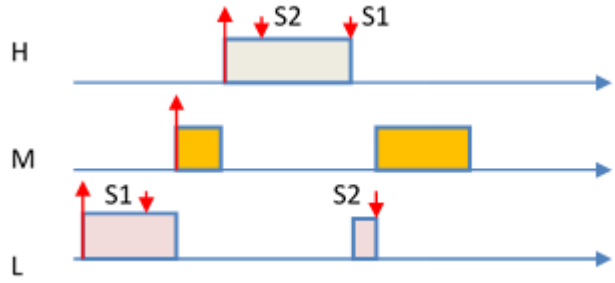
The **response time** of task **H** will be **much shorter**, and the worst-case blocking time equals the duration of the critical section of task **L**.

The **worst-case response time** will increase with the worst-case blocking time ($\boldsymbol{B_i}$):

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

Deadlock avoidance:

The Priority Inheritance Protocol should be extended/modified if more common resources are to be handled. This is illustrated by the following figure:



Task **L** by locking semaphore **S1** enters a critical **section**. Within this critical section semaphore **S2** will be also locked by task **L**. These two resources – with the given timing – are used by task **H**, as well.

As task **H** would like to lock semaphore **S1**, it will be blocked. Task **L** inherits priority **H**, but trying to lock semaphore **S2** it will also block. Both task **H** and **L** will wait for the other. This situation is called: **deadlock**. To avoid it **priority ceiling protocols** are used.

Priority Ceiling Protocol (PCP): The basic idea of this method is to extend the PIP with a rule for granting a lock request on a free semaphore. To avoid multiple blocking, this rule does not allow a task to enter a critical section if there are locked semaphores that could block it.

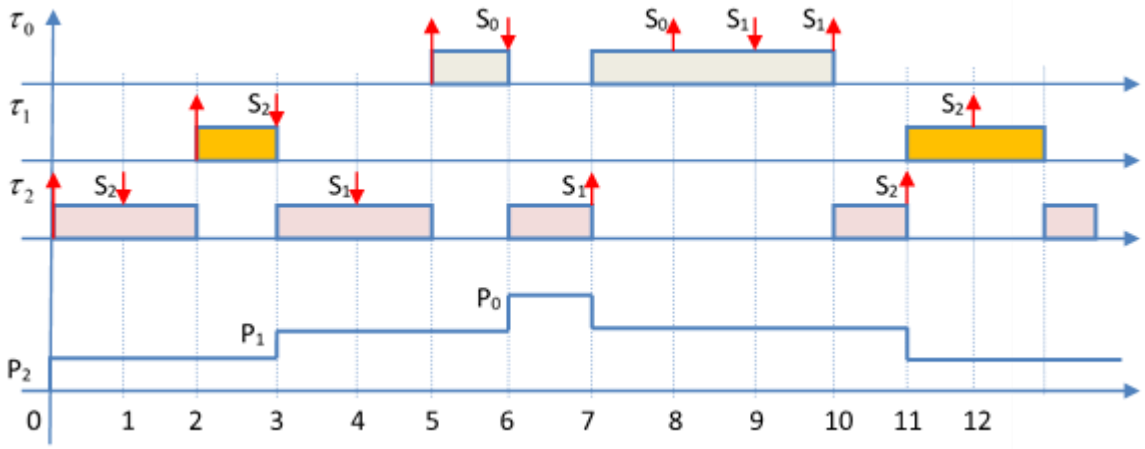
This means that, once a task enters its first critical session, it can never be blocked by lower-priority tasks until its completion.

To realize this idea, each semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. Then, a task i can enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks other than i.

The PCP protocol:

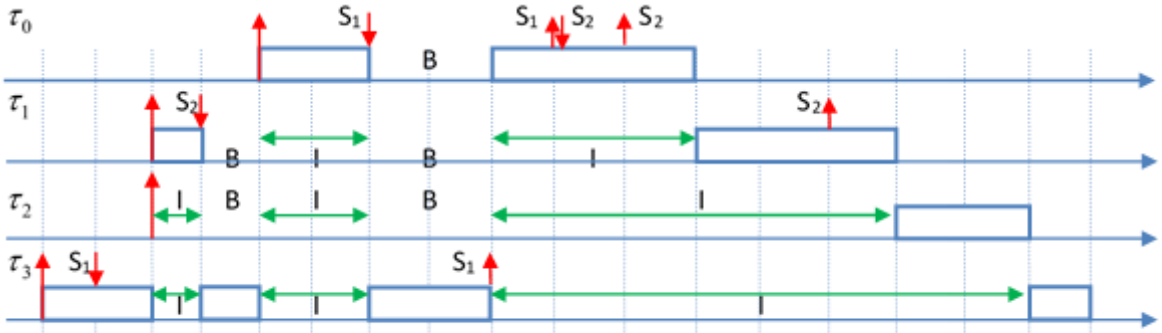
- Each semaphore $\boldsymbol{S_k}$ is assigned a priority ceiling $\boldsymbol{C(S_k)}$ equal to the priority of the highest-priority task that can lock it. Note that $\boldsymbol{C(S_k)}$ is a static value that can be computed offline.
- Let $\boldsymbol{\tau_i}$ be the task with the highest-priority among all tasks ready to run; thus, $\boldsymbol{\tau_i}$ is assigned to the processor.
- Let $\boldsymbol{S^*}$ be the semaphore with the highest-priority ceiling among all the semaphores currently locked by tasks other than τ_i, and let $\boldsymbol{C(S^*)}$ be its ceiling.
- To enter a critical section guarded by a semaphore $\boldsymbol{S_k}$, $\boldsymbol{\tau_i}$ must have a priority ($\boldsymbol{P_i}$) higher than $\boldsymbol{C(S^*)}$. If $\boldsymbol{P_i \leq C(S^*)}$, the lock request is denied and $\boldsymbol{\tau_i}$ is said to be blocked on semaphore $\boldsymbol{S^*}$ by the task that holds the lock on $\boldsymbol{S^*}$.
- When a task $\boldsymbol{\tau_i}$ is blocked on a semaphore, it transmits its priority to the task, say $\boldsymbol{\tau_k}$, that holds that semaphore. Hence, $\boldsymbol{\tau_k}$ resumes and executes the rest of its critical section with the priority of $\boldsymbol{\tau_i}$. In general, a task inherits the highest priority of the task blocked by it.
- When $\boldsymbol{\tau_k}$ exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. Moreover, the active priority of $\boldsymbol{\tau_k}$ is updated as follows: if no other jobs are blocked by $\boldsymbol{\tau_k}$, its priority is set to the nominal (static) priority; otherwise it is set to the highest-priority of the tasks blocked by $\boldsymbol{\tau_k}$.

Example: The tasks to be scheduled with descending priority are: $\boldsymbol{\tau_0, \tau_1, \tau_2}$. Their priorities: P_0, P_1 and P_2. The resources are guarded by semaphores S_0, S_1 and S_2. Their priority ceilings: $C(S_0) = P_0$, $C(S_1) = P_0$, $C(S_2) = P_1$.



Note that task τ_0 will be blocked even though the requested resource is not blocked.
The reason of this blocking is that task τ_2 is within a critical section guarded by semaphore S_1 the priority of which is equal of that of τ_0.

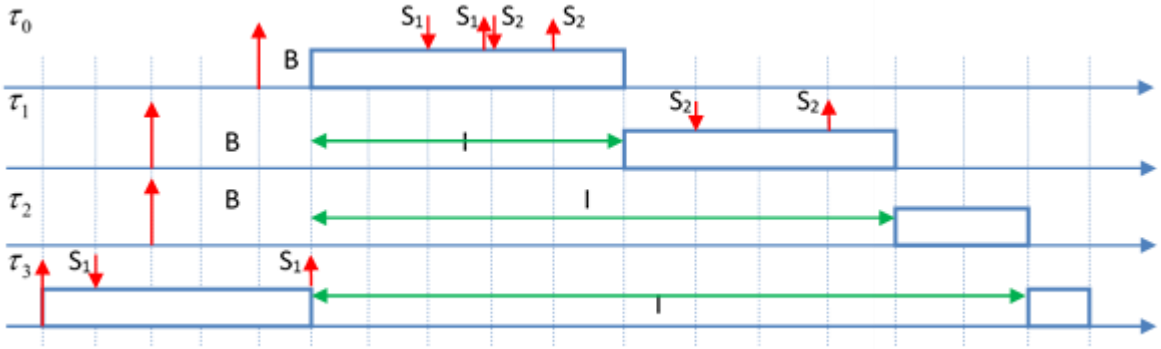
Example: The tasks to be scheduled with descending priority are: $\boldsymbol{\tau_0, \tau_1, \tau_2, \tau_3}$. Their priorities: P_0, P_1, P_2 and P_3. The resources are guarded by semaphores S_1 and S_2. Their priority ceilings: $C(S_1) = P_0$, $C(S_2) = P_0$.



On the figure it is easy to follow the operation of the PCP protocol.
„**I**" denotes the interference intervals, while „**B**" stands for the blocking intervals.
The sum of these latter gives the effective blocking time,
the worst-case value of which equals length of the critical section of task τ_3.

Immediate Priority Ceiling Protocol (IPCP):

The essence of the protocol is that the tasks entering a critical section **immediately** inherit the **ceiling priority** of the semaphore which guards the **critical section**!



Thus, on the figure below, task $\boldsymbol{\tau_3}$ at entering the critical section receives as dynamic priority $\boldsymbol{P_0}$, and will operate at this priority level till the end of the **critical section.**

The implementation of IPCP is **easier** than that of the PCP, and there are **less** task-switching, and consequently context switching.

It is interesting to note that **the semaphores** do not need implementation because after leaving the first critical section they **are and remain unlocked**!

It is also interesting to realize that using IPCP the response time of the highest priority task became shorter. The name IPCP in POSIX **is Priority Protect Protocol**, and in Real-Time Java: **Priority Ceiling Emulation**.

Embedded Information Systems: 1st Mid-Term Exam 2019

1. Characterize the hard and soft real-time systems concerning (a) response time; (b) peak-load performance; (c) control of pace; (d) safety; (e) size of data files; (f) redundancy type; (g) data integrity; (h) error detection (max. 4 points)!

Hard RT Systems versus Soft RT Systems:

Hard real-time system (HRT): which must produce the result at the correct instant, because if we do not meet the time limitation, it might result in catastrophic consequences. (See e.g. the electronic control of vehicles).

Soft real-time system (SRT), online system: the result has a value also if we do not meet the time limitation, only the quality of the service will degrade (See e.g. transaction processing 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

2. What does it mean that a periodically updated real-time image is phase sensitive (max. 1 point)? What is its consequence (max. 1 point)?

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

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

Consequence: In this case it is not sure that the update will arrive within the interval of the temporal accuracy, therefore the time of update and use should be monitored.

3. What does state-observation, and what does event-observation mean (max. 2 points)? What does it mean that a real-time variable is out the sphere of control (SOC) of a system (max. 2 points)?

***State observations*:** Every observation is self-contained because it carries an absolute value. In many cases equidistant sampling is applied, i.e. periodic time-triggered readings.

Event observations: an event is a state-change at a point of time. Since an observation is also an event, therefore it is not possible to observe an event in the controlled object directly.

Every *R*T variable is in ***the sphere of control*** (SOC) of a subsystem that has the authority to set the value of the *RT* variable. Outside its SOC the *RT* variable can only be observed, but not modified.

4. Within a communication system the maximum value of the message forwarding time is $5.5\,ms$, while the jitter is $5\,ms$. Determine the value of the action delay (1) if the global time is available (max. 1 point); (2) if the global time is not available (max. 1 point)! What can we do if the temporal accuracy of the transmitted value is $8\,ms$ (max. 2 points)?

$$jitter = d_{max} - d_{min} = 5\,ms, d_{min} = 0.5\,ms$$

(1) If the global time is available: **action delay**: $d_{max} = \mathbf{5.5\,ms}$ because the time instant of the message forwarding is known.

(2) If the global time is not available: **action delay**: $2d_{max} - d_{min} = \mathbf{10.5\,ms}$

If the temporal accuracy of the transmitted value is $8\,ms < 10.5ms$, them **state estimation** is to be applied.

The increase of the updating rate does not help since action delay is to applied also for the updated value.

5. What is the difference between **aperiodic** and **sporadic** tasks (max. 1 point)? Why is this difference **important** (max. 1 point)?

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.

The difference is important, because in the case of aperiodic tasks **the DMA method** cannot be applied.

6. Using the Deadline Monotonic Analysis (DMA) method, calculate the worst-case response time of Task4, if the time difference between two occurrence of a single failure during task execution is minimum $T_F = 50ms$, and the error handling requires at maximum $C_F = 2ms$. (max. 4 points):

Task	T[ms]	C[ms]	D[ms]
1	100	5	10
2	10	2	10
3	100	25	50
4	100	30	100

$$R_4' = C_4 + I_4 + \left\lceil \frac{R_4}{T_F} \right\rceil C_F$$

Step	R_4	$I_4 + \left\lceil \frac{R_4}{T_F} \right\rceil C_F$	R_4'
1	0	0	30
2	30	5+6+25+2	68
3	68	5+14+25+4	78
4	78	5+16+25+4	80
5	80	5+16+25+4	80

The worst-case response time: $80ms < 100ms$.

Imagine the application of Dual Priority Scheduling! Determine the promotion time of task 3 (max. 2 points)!

Step	R_3	$I_3 + \left\lceil \frac{R_3}{T_F} \right\rceil C_F$	R_3'
1	0	0	25
2	25	5+6+2	38
3	38	5+8+2	40
4	40	5+8+2	40

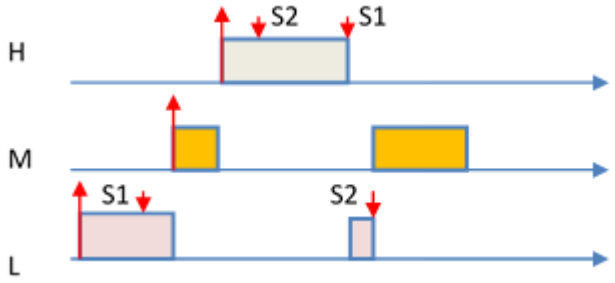
The worst-case response: $40ms < 50ms$.

The promotion time after the request arrival is $10ms$, since only with this timing can Task3 be completed before deadline.

7. What is the key idea of the priority inheritance algorithm, when it is used, and what can it cause in case of more critical sessions (max. 2 points)?

Priority Inheritance Protocol (PIP)**:**

To avoid priority inversion, task **L** should dynamically inherit the priority of task **H** upon its request to enter the critical section. Thus, task L can complete the critical section much earlier and unlock semaphore **S1**. The inherited priority is called dynamic priority valid only for the critical section. After unlocking semaphore **S1** the static priority will be restored.



In case of more critical sessions **deadlock** may occur.

What is the key idea of the Immediate Priority Ceiling Protocol and why is it relatively easy to implement (max. 2 points)?

Immediate Priority Ceiling Protocol (IPCP):

The essence of the protocol is that the tasks entering a critical section **immediately** inherit the **ceiling priority** of the semaphore which guards the critical section.

The first task at entering the critical section receives **as dynamic priority** the ceiling priority, and will operate at this priority level till the end of the critical section.

The **implementation of IPCP is easier** than that of the PCP, and there are **less** task-switching, and consequently context switching.

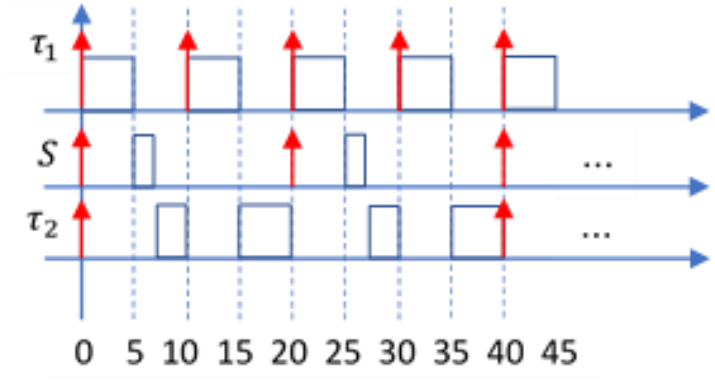
It is interesting to note that the semaphores **do not need implementation** because after leaving the first critical section they are and remain unlocked.

8. We must schedule two hard real-time tasks. The requests are simultaneous: at the beginning both tasks are ready to run.

	C[ms]	T[ms]
τ_1	5	10
τ_2	16	40

In addition to the two tasks a server task is also scheduled to provide processor capacity for aperiodic requests. The server period is $T_S = 20ms$, and the server capacity is $C_S = 2ms$.

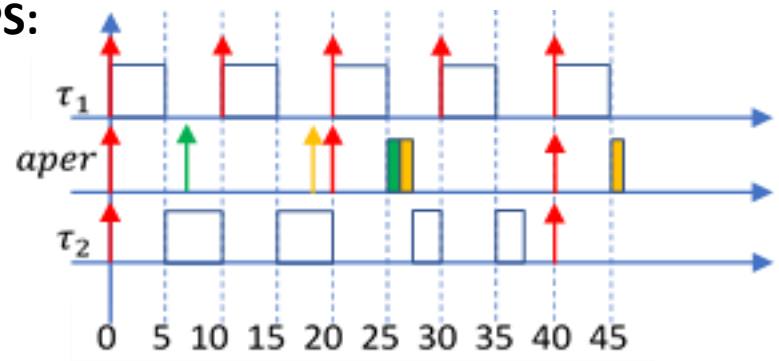
Show how the three tasks are scheduled with the RM algorithm (max. 2 points)!



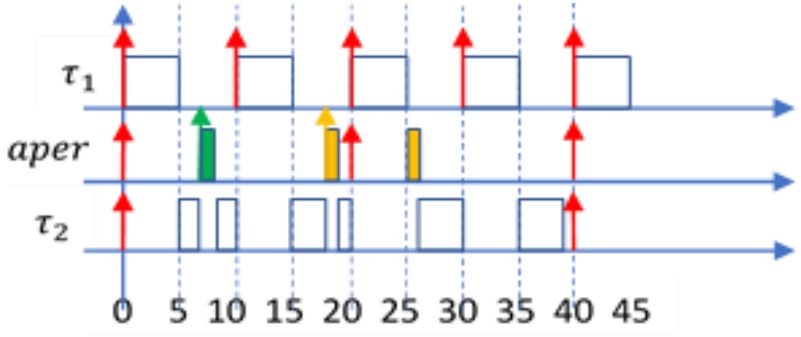
Firstly, apply the **Polling Server** (PS) algorithm, and after it use the **Deferrable Server** (DS)!

Show how an aperiodic request at $\boldsymbol{7ms}$ asking for processor time of $\boldsymbol{1ms}$, and another aperiodic request at $\boldsymbol{18ms}$ asking for processor time of $\boldsymbol{2ms}$ is scheduled using **PS** and **DS** (max. 4 points)?

PS:



DS:

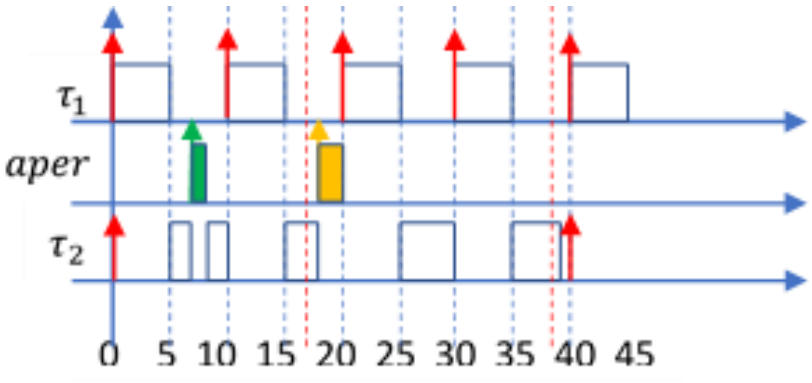


Determine the response time of the aperiodic requests (max. 1 point)!

$R_{PS1} = 19ms, R_{PS2} = 28ms$

$R_{DS1} = 1ms, R_{DS2} = 8ms$

How will the response time change if **EDF** together with a **Total Bandwidth Server** is applied (max. 3 points)?



$$d_1 = 7 + \frac{1}{0.1} = 17ms \qquad d_2 = 18 + \frac{2}{0.1} = 38ms$$

$$R_{TBS1} = 1ms,\ R_{TBS2} = 2ms$$

9. Under what condition will be schedulable a periodic hard real-time task-set if $D_i < T_i$? Consider both static and dynamic priority assignment strategies (max. 2 points)!

	$D_i<T_i$
static priority	**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 demand approach $\forall L > 0 \quad L \geq \sum_{k=1}^{n} \left(\left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k$

10. Characterize the operation of the event triggered and that of the time triggered systems! In case of safety critical systems which one is preferable (max. 2 point)?

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, therefore in the case of a priori known response times, the program execution can be guaranteed.

3. Memory management

Scheduling not independent tasks we faced some problems of **handling resources**. Here we discuss the problems of **memory management** from the viewpoint of embedded systems.

In the case of embedded systems, it is typically not possible to eliminate the side-effects of **not completely correct resource handling** time-to-time by resetting the device.

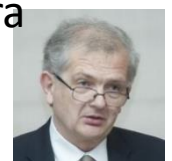
We must design such systems, where the performance of the resources remains **stable**, **degradation is not possible**.

- **Static memory allocation:** If the memory is allocated statically, then it can be established at compile time exactly how each byte of RAM will be used during the running of the program. This has the advantage, for embedded systems, that the whole issue of bugs due to leaks and failures due to fragmentation simply does not exist. The global and static data is allocated in a fixed location since it must remain valid for the life of the program.

> This approach **prohibits** the use of **recursion**, function pointers, or any other mechanisms that require **re-entrant code**.

For example, and interrupt routine cannot call a function that may also be called by the main flow of execution.

- **Stack based management:** The next step up in complexity is to add a stack. Now a block of memory is required for every call of a function, and not just a single block for each function in existence. The stack grows and shrinks as the program executes, and for many programs it is **not possible to predict**, at compile time, what **the worst-case stack size** will be. In multitasking system, there will be one stack per task to manage, plus possibly an extra one for interrupts.



Some **judgement** must be exercised to make sure that each stack is **big enough** for all its activities. It is awful shame to **suffer from** an untimely **stack overflow**, when one of the other stacks has reserve of space that it never uses.

Unfortunately, most embedded system does not support any kind of virtual memory management that would allow the tasks to draw from a common pool as the need arises.

One rule of thumb is to make each stack **50% bigger** than the worst case seen during testing. One simple technique is to paint the stack space with **simple pattern**. As the stack grows and shrinks, it will **overwrite** the area with its data. This technique is called **watermarking.** Many RTOS's support this mechanism.

- **Heap based management:** In C programs heap management is carried out by the $\boldsymbol{malloc}()$ and $\boldsymbol{free}()$ functions. $\boldsymbol{Malloc}()$ allows the programmer to acquire a pointer to an available block of memory of a specified size. $\boldsymbol{Free}()$ allows the programmer to return a piece of memory to the heap when the application has finished it.

While **stack management** is handled by the **computer**, using **heap management** requires care by the **programmer**, or many devious bugs can creep into the program.

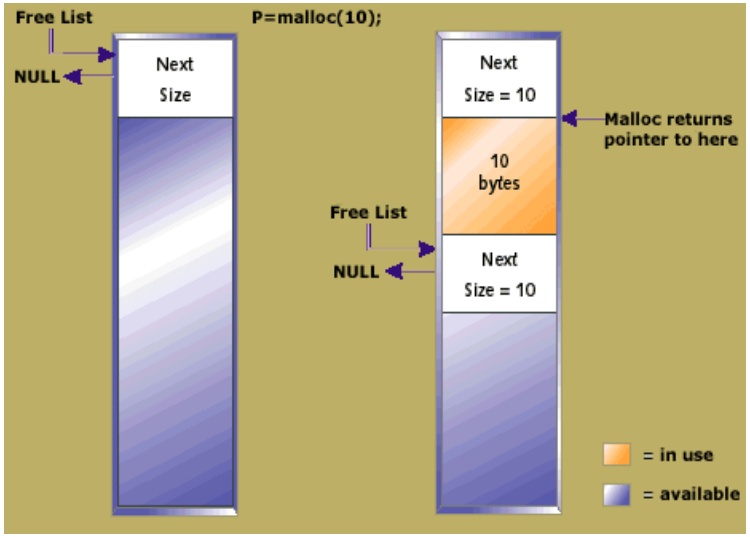
At a certain point in the code you may be **unsure** if a particular block is **no longer needed**. If you $free()$ this piece of memory, but continue to access it (probably via a second pointer to the same memory), then your program may function perfectly, until that particular piece of memory is **reallocated** to another part of the program.Then two different parts of the program will proceed to write **over each other's data**. If you decide not to free the memory, on the grounds that it may still be in use, then **you may not get** another **opportunity** to free it, since all pointers to the block may have gone **out of scope**, or been **reassigned** to point elsewhere.

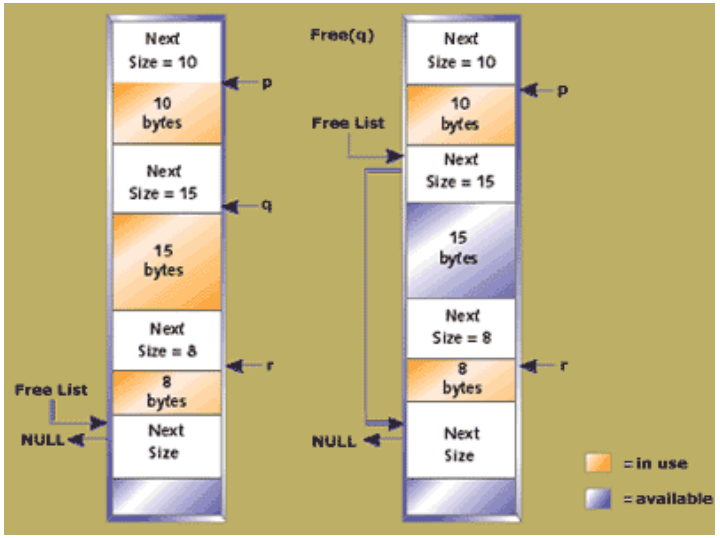
The result is: **memory leakage**. In this case the program logic will not be affected, but if the piece of code that leaks memory is visited on a regular basis then the leak **will tend toward infinity**, and the **execution time** of the program **increases**.

Any leak is a bug, which can be rectified by correcting the logic of the program.

There is another problem called **fragmentation**, which cannot be corrected at the application program level. This is a property **inherent** in most applications of $\boldsymbol{malloc()}$.

It is caused by **blocks of memory** available **being broken down into smaller pieces** as many allocations and frees are performed.





These two figures are from the lecture of Niall Murphy (Panelsoft) entitled Memory Management, presented among others at the Embedded Systems Conference Europe in 2000.

Fragmentation can be reduced by using the appropriate policy when allocating and freeing blocks.

Allocation policies include:

- Allocate (and possibly split) first block found, larges than the request (**First Fit**).
- Allocate the best fit after exhaustive search (**Best Fit**).

Free list policies include:

- Maintaining the list in order of address, to simplify merging of free blocks.
- Maintain the list in most recently used order, to match patterns of use where similar sizes are allocated and freed in bursts.

Seeing the **difficulties**, the conclusion is that **mission critical project** cannot afford these **dynamic memory allocation** mechanisms.

Recommendation: limited heap functionality, static allocation:
(1) malloc() is used only during initialization, and freeing is not applied.
(2) It worth writing a separate function (E.g. $salloc(\,)$ (static allocation)).
(3) Following the initialization $salloc(\,)$ is inhibited.

Recommendation:
Dynamic allocation but fixed block size. (Partitions or pools of fixed size memory blocks.)

Multitasking: While each task must have its **own stack**, it may or may **not have** its **own heap**, regardless of whether the heap is based on the static scheme, pools, or general-purpose allocation scheme. Having **more than one heap** means that you must tune the size of several heaps, which is a **disadvantage.**
However, one heap for many tasks must be re-entrant, which means adding locks that will slow down each allocation and deallocation.

A single heap also allows one task to allocate a piece of memory which may be freed by another task**.** This is **useful** for passing **inter-task messages**.

When memory is passed between tasks in this way, make sure that it is **always well defined who owns** the memory at each point.

Libraries: Problems:

(1) Memory should be allocated by the library.

(2) Freeing memory is the role of the application program.

(3) We can assign static memory to the library; however, this is not suitable for re-entrant code, which is so essential to many embedded systems.

(4) All these problems should be considered by the programmer of the library: possibly by offering library routines for freeing memory. (So-called Pluggable memory management).

Automatic garbage collection: e.g. **Java**, **LISP**, **Smalltalk** offer such a service.

Two basic mechanisms:

(1) The **pointers can be objects**, which have **destructors** that are called when the pointer goes out of scope. In C++ this is possible with **smart pointers**.

(2) To test whether a piece of memory is free a search is performed within the memory in use to find **a pointer to that block**. If no pointers to the block are found, then the **block is free**. This is obviously an expensive way to check for available memory.