Embedded Information Systems

4. Measuring time, clocks, clock synchronization

October 20, 2020

Reminder:

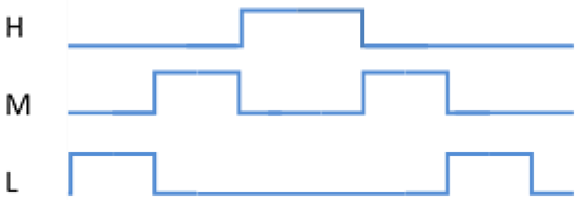
If the **scheduler decides** a task to run, **then first the registers of the processor should be saved, after this the context** of the new task should be loaded into the registers, and then comes the **execution** of the task.
The response time should be increased by time of this „**context switch**”.

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

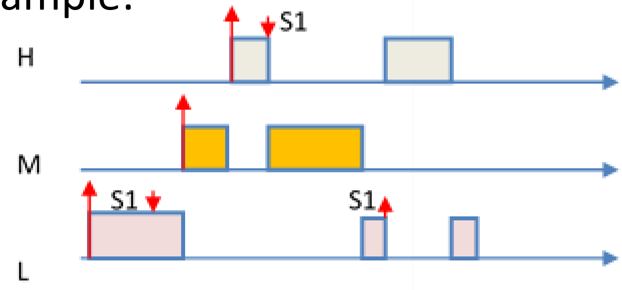
Except for the **time-sharing systems**, where the processor’s capacity is shared among **independent users**, for most of the applications the runs of the different tasks **are not completely independent.** Tasks are **communicating** with each other, **exchange data**, they are **waiting for results** 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!



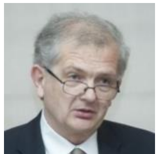
If here task L would use such a resource, which is later also used by task H, then it might happen that task H should wait until the resource will be released.

Example:



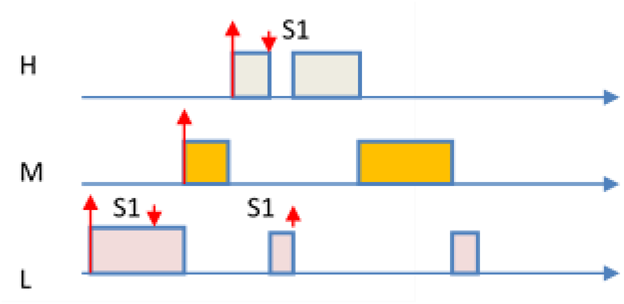
This type of waiting is called **blocking**, because **lower priority** task **forces higher priority** task to **wait**.

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



Reminder:

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.



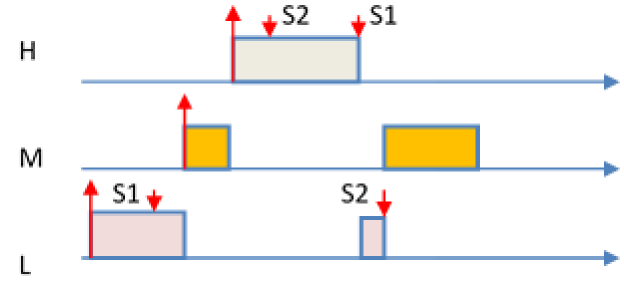
The **response time** of task **H** will be **much shorter**, and the worst-case blocking time equals the duration of the critical section of task **L**.

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:

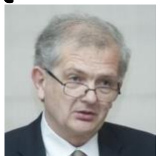
The Priority Inheritance Protocol should be extended/modified if more common resources are to be handled. This is illustrated by the following figure:



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.



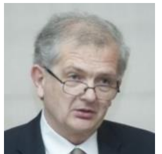
Reminder:

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.

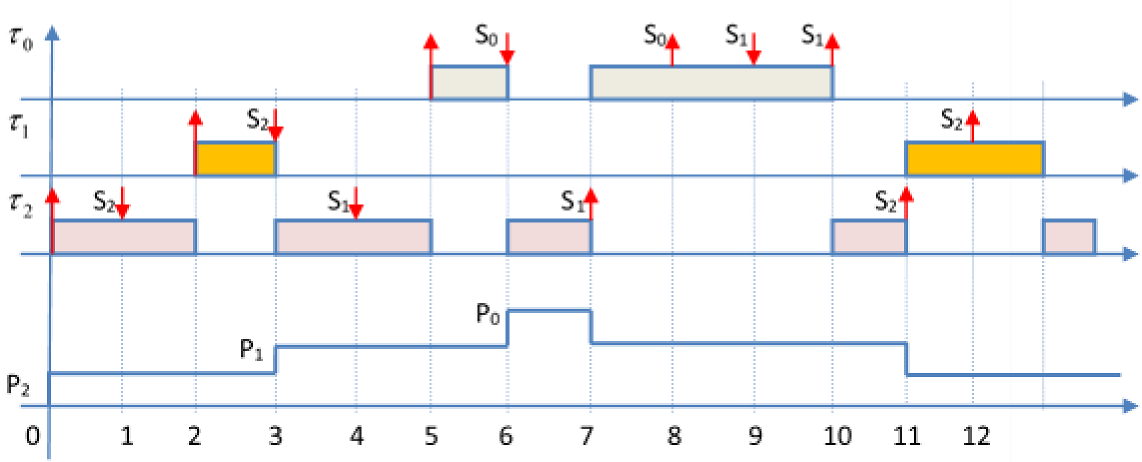
The PCP protocol:

- 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 $\boldsymbol{\tau_k}$.



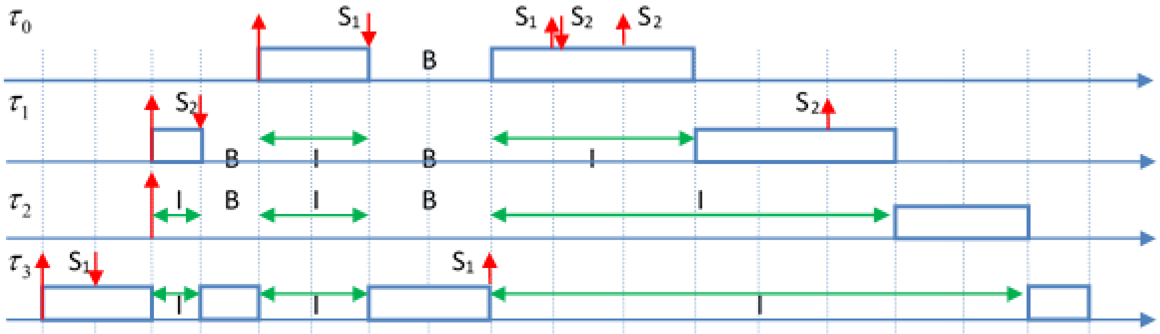
Reminder:

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: $C(S_0) = P_0$, $C(S_1) = P_0$, $C(S_2) = P_1$.

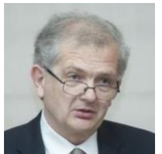


Note that task τ_0 will be blocked even though the requested resource is not blocked.
The reason of this blocking is that task τ_2 is within a critical section guarded by semaphore S_1 the priority of which is equal of that of τ_0.

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: $C(S_1) = P_0$, $C(S_2) = P_0$.



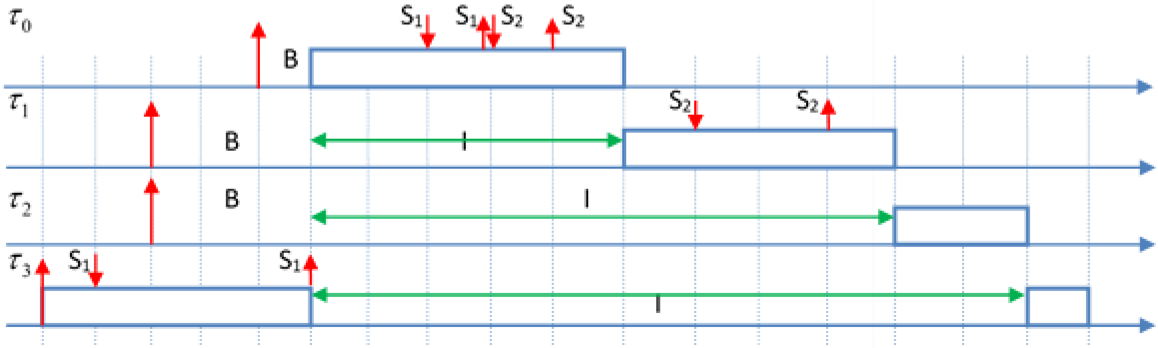
On the figure it is easy to follow the operation of the PCP protocol. „**I**" denotes the interference intervals, while „**B**" stands for the blocking intervals.
The sum of these latter gives the effective blocking time, the worst-case value of which equals length of the critical section of task τ_3.



Reminder:

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**!

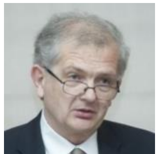


Thus, on the figure below, task τ_3 at entering the critical section receives as dynamic priority P_0, and will operate at this priority level till the end of the **critical section.**

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**!

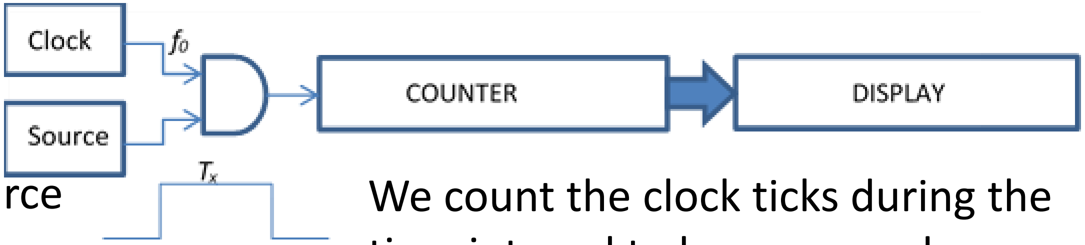
It is also interesting to realize that using IPCP the response time of the highest priority task became shorter. The name IPCP in POSIX **is Priority Protect Protocol**, and in Real-Time Java: **Priority Ceiling Emulation**.



4. Measuring time, clocks, clock synchronization

Tools and methods:

(1) Measuring time using an electronic counter:



The gate time T_x generated by the source is the time duration to be measured.

We count the clock ticks during the time interval to be measured:

$$\boxed{T_x \cong \frac{N}{f_0}}$$

, where N is the content of the counter, and f_0 stands for the clock frequency. The approximate equality refers that N is always integer, while $T_x f_0$ is not necessarily. The (worst-case) relative error of the time measurement:

$$\boxed{\left|\frac{\Delta T_x}{T_x}\right| \cong \left|\frac{1}{N}\right| + \left|\frac{\Delta f_0}{f_0}\right|}$$

This equation can be derived from the complete differential

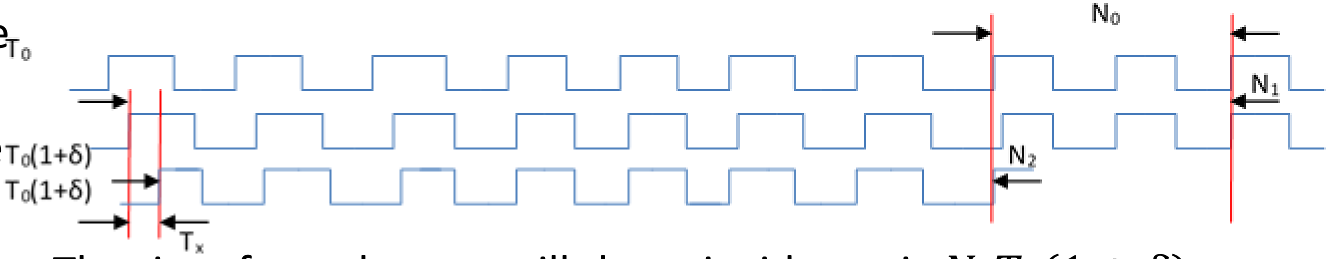
$$dT_x = \frac{\partial T_x}{\partial N} dN + \frac{\partial T_x}{\partial f_0} df_0 = \frac{1}{f_0} dN - \frac{N}{f_0^2} df_0,$$

which divided by $T_x = \frac{N}{f_0}$ gives $\frac{dT_x}{T_x} = \frac{dN}{N} - \frac{df_0}{f_0}$.

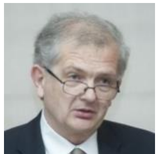
Since the sign of the changes is not known, therefore we use the absolute value of the changes and express the worst case relative error.

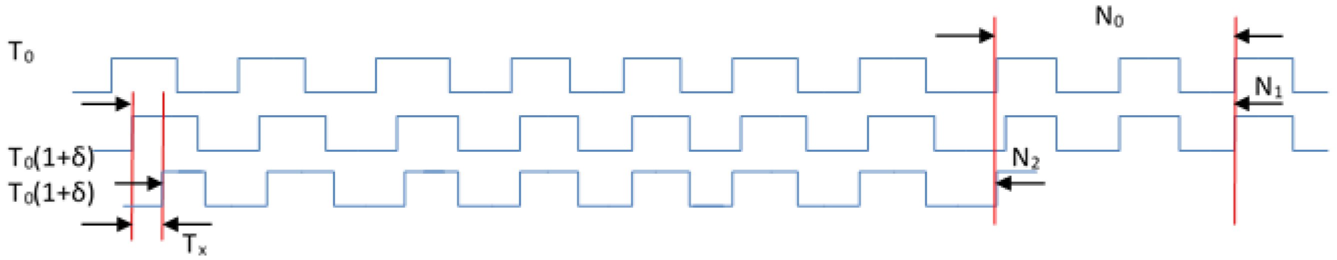
(2) The dual vernier method:

The sliding edge of the time interval to be measured and falling edge of the time interval start the **phase-startable phase-lockable oscillators** having periods of $T_0(1+\delta)$.



The time from the **start** till the coincidence is $N_1 T_0(1+\delta)$, while from the **end** till the coincidence it is $N_2 T_0(1+\delta)$. The **time between** the two coincidences is $N_0 T_0$.



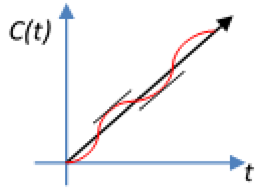


The time from the **start** till the coincidence is $N_1 T_0(1+\delta)$, while from the **end** till the coincidence it is $N_2 T_0(1+\delta)$.

The time between the **two coincidences** is $N_0 T_0$. Thus $\boxed{T_x = T_0[\pm N_0 + (N_1 - N_2)(1+\delta)]}$

where the sign before N_0 is determined by the order of the two coincidences

If T_0**=5 nsec** and **δ=0.004**, then the shortest measurable duration is **20psec**.



Clocks are the sources of the knowledge of time with a given accuracy:

The source of the knowledge of time is called **clock**.

Reference clock or standard clock: if $C_k(t) = t;\ \forall t$.

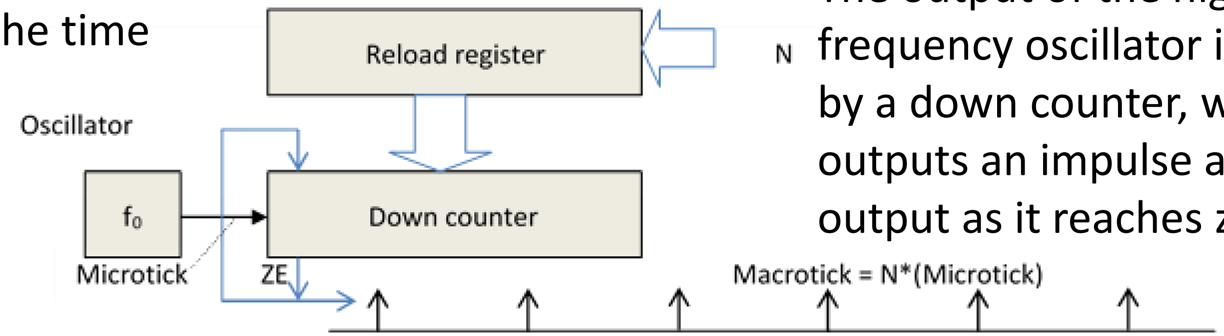
Correct clock: Clock k is correct in t_0, if $C_k(t_0) = t_0$.

Accurate clock: Clock k is accurate in t_0, if $\frac{\partial C_k(t)}{\partial t} = 1;\ t = t_0$.

Clock k is a function $C_k(t)$ of time, which maps real time to the time at clock k.

If a clock is inaccurate at some point of time, we say that the clock *drifts* at that point of time.

Physical clock: Oscillator + counter, its granularity $g = \frac{1}{f}$ is the time between microticks.



The output of the high-frequency oscillator is divided by a down counter, which outputs an impulse at its ZE output as it reaches zero.