# 5. System issues

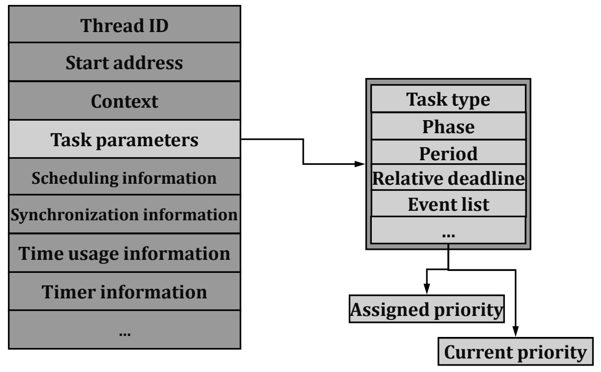
## Context switch

- Preemption causes time and space overheads, which should be duly accounted for in accurate schedulability tests
- Under preemption, every single job incurs at least two context switches
  - One at activation to install its execution context
  - One at completion to clean up
- The resulting costs should be charged to the job
  - Knowing the timing behavior of the run-time system we could incorporate overhead costs in schedulability tests

## Real-time operating systems /1

- The RTOS knows all tasks: they are the unit of CPU assignment
  - Tasks issue jobs: scheduling and dispatching applies to them
  - The *scheduler* decides which task (job) gets the CPU
  - The *dispatcher* gets jobs to run and operates context switches
- One ***Task Control Block*** per task is stored in RAM
  - The insertion of a task in a state queue (e.g., ready) is made by placing a pointer to the corresponding TCB
  - The disposal of a task at end of life (if any) requires removing its TCB and releasing all of its memory (e.g., its stack)

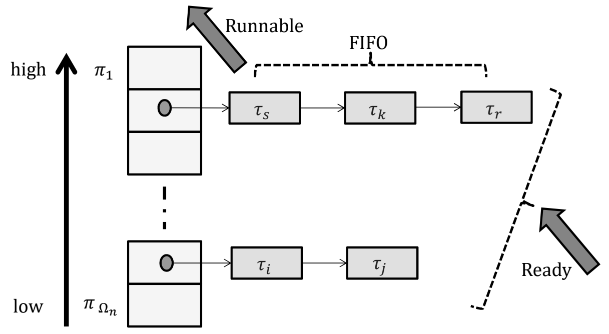
## Task control block (example)



## Priority levels /1

- The scheduling techniques that we have studied assume jobs to have *distinct* priorities
  - Concrete systems may however not always be able to meet this requirement
  - Jobs may therefore have to share priority levels
  - For jobs at the same level of priority, dispatching may be FIFO or round-robin
- If priority levels are shared then we have a worst-case situation to contemplate in the analysis
  - That job $J$ be released immediately *after* all other jobs at its level of priority

## Priority levels /2



## Priority levels /3

- Let $S(i)$ denote the set of jobs $\{J_j\}$ with $\pi_j = \pi_i$, excluding $J_i$ itself
- Then the time demand equation for $J_i$ to study in the interval $0 < t \leq \min(D_i, p_i)$ becomes

$$\boxed{\omega_{i_1}(t) = e_i + B_i + \sum_{S(i)} e_i + \sum_{k=1,..,i-1} \left\lceil \frac{\omega_{i_1}(t)}{p_k} \right\rceil e_k}$$

- This obviously worsens $J_i$'s response time
  - But the impact in terms of ***schedulability loss*** at system level may not be as bad (wait and see …)

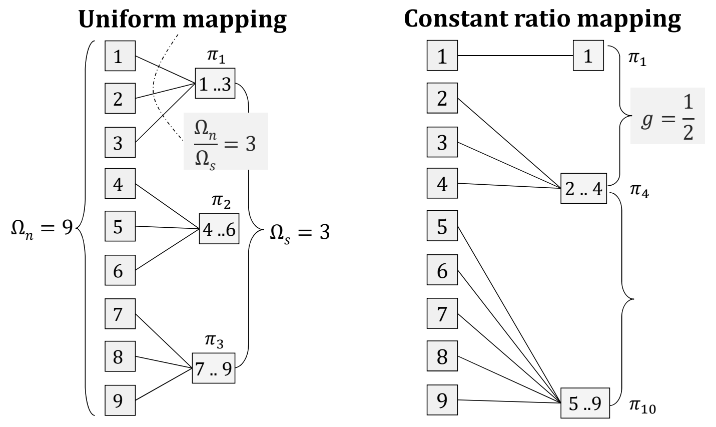
## Priority levels /4

- When the number $[1, .., \Omega_n]$ of *assigned priorities* is $>$ than the number $[\pi_1, .., \pi_{\Omega_s}]$ of *available priorities* (aka **priority grid**), then we need some $\Omega_n : \Omega_s$ mapping
  - All assigned priorities $\geq \pi_1$ will take value $\pi_1$
  - For $1 < k \leq \Omega_s$, the assigned priorities in the range $(\pi_{k-1}, \pi_k]$ will take value $\pi_k$
- Two main techniques
  - **Uniform mapping**
  - **Constant ratio mapping** [Lehoczky & Sha, 1986]

## Priority levels /5

- **Uniform mapping**
  - $Q = \left\lfloor \frac{\Omega_n}{\Omega_s} \right\rfloor$ ➔ $\pi_k \leftarrow [k, \dots, kQ], \pi_{k+1} \leftarrow [kQ+1, \dots, (k+1)Q]$
  - **Example**: $\Omega_n = 9, \Omega_s = 3$, $(\pi_1 = 1, \pi_2 = 2, \pi_3 = 3)$
    $Q = \frac{9}{3} = 3$ ➔ $\pi_1 \leftarrow [1..3]$, $\pi_2 \leftarrow [4..6], \pi_3 \leftarrow [7..9]$
- **Constant ratio mapping**
  - Spaces the $\pi_i$ values keeping the ratio $g = \frac{(\pi_{i-1}+1)}{\pi_i}$ constant for $i = 2, .., \Omega_s$ for the better good of higher-priority jobs
  - **Example (as above)**: choosing $g = \frac{1}{2}$ and $\pi_1 = 1$ (top) then $\pi_2 = 4,\ \ \pi_3 = 10$ ➔ $\pi_1 \leftarrow [1], \pi_2 \leftarrow [2..4], \pi_3 \leftarrow [5..9]$

## Priority levels /6



## Priority levels /7

- Lehoczky & Sha showed that constant ratio mapping degrades the schedulable utilization of RMS *gracefully*
  - For large $n$, with $D_i = p_i\ \forall i$, and $g = min_{2 \le j \le \Omega_s} \frac{(\pi_{j-1}+1)}{\pi_j}$

    CRM's schedulable utilization $f(g) = \begin{cases} ln(2g) + 1 - g, & g > \frac{1}{2} \\ g, & g \le \frac{1}{2} \end{cases}$
- The $\frac{f(g)}{ln(2)}$ ratio consequently represents the CRM's ***relative schedulability*** in relation to the RMS' utilization bound
  - **Example**: for $\Omega_s = 256, \Omega_n = 100{,}000$, and the corresponding $g$, the relative schedulability of CRM reaches up to $0.9986$
  - 256 priority levels should then suffice for RMS

## Real-time operating systems /2

- Must be small, modular, extensible
  - **Small footprint**: they are often stored in ROM (which used to be little) and most embedded systems have little RAM
  - **Modular**: this eases the qualification of its design and implementation, including of temporal predictability
  - **Extensible**: some but not all specific systems may need functionalities above and beyond the core ones
- Adhering to the principle of *microkernel architecture*
  - Minimal kernel services to include scheduling, inter-task communication and synchronization, interrupt handling

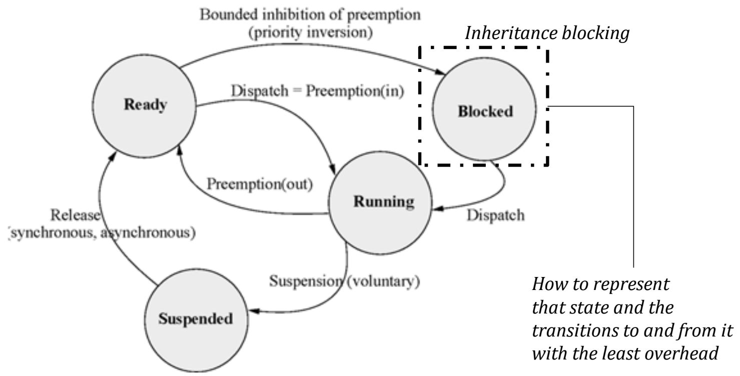
## Real-time operating systems /3

- Tasks may be realized as specialized primitive entities that live within the RTOS
  - Then the *model of computation* is determined by the RTOS
- Or at application level with generic support from API exposed by the RTOS (e.g., pthread_*)
  - Then it is the user responsibility to ensure that the execution semantics conforms with the analysis model

## Real-time operating systems /4

- **Periodic task**
  - An RTOS thread that hangs on a periodic *suspension point*
    - After release it executes the application-code of the job and then makes a suspensive call
- **Sporadic task**
  - An RTOS thread whose suspension point is not released periodically but with *guaranteed* minimum distance
    - After release it executes the job and then makes a suspensive call
- **Aperiodic task**
  - Indistinguishable from the rest other than its being placed in a server's *backlog queue* and not in the ready queue

## Task states /1


Bounded inhibition of preemption
(priority inversion)
Inheritance blocking
Ready
Dispatch = Preemption(in)
Blocked
Preemption(out)
Running
Dispatch
Release
(synchronous, asynchronous)
Suspension (voluntary)
Suspended
How to represent that state and the transitions to and from it with the least overhead

## Task states /2

- Tasks enter the *suspended* state only voluntarily
  - By making a primitive invocation that causes them to hang on a periodic / sporadic suspension point
- The RTOS needs specialized structures to handle the distinct forms of suspension
  - A time-based queue for periodic suspensions
  - An event-based queue for sporadic suspensions
    - But someone shall still take care of warranting minimum separation between subsequent releases (!)

## The scheduler /1

- This is a distinct part of the RTOS that does **not** execute in response to explicit application invocations
  - Other than when using cooperative scheduling (which we are not considering here)
- It acts every time the ready queue changes
  - The corresponding time events are termed *dispatching points*
- Scheduler "activation" is often periodic in response to *clock interrupts*

## The scheduler /2

- At every clock interrupt the scheduler must
  - Manage the queue of time-based events pending
  - Increment the execution time budget counter of the running job to support time-based scheduling policy (e.g., LLF)
  - Manage the ready queue
- The $\geq 10ms$ period (aka ***tick size***) typical of general-purpose operating systems is **not** fit for RTOS
  - But a higher frequency incurs larger overhead
- The scheduler needs to make provisions for event-driven execution as well

## Tick scheduling /1

- So far we have tacitly assumed that the scheduler operates on an *event-driven* basis
  - The scheduler always immediately executes on the occurrence of a *scheduling event* (i.e., dispatching point)
  - If it was so then we could assume that a job is placed in the ready queue at its release time
- Schedulers may also operate in a *time-driven* manner
  - But then the scheduling decisions are made and executed on the arrival of periodic clock interrupts
  - This mode of operation is termed *tick scheduling*

## Tick scheduling /2

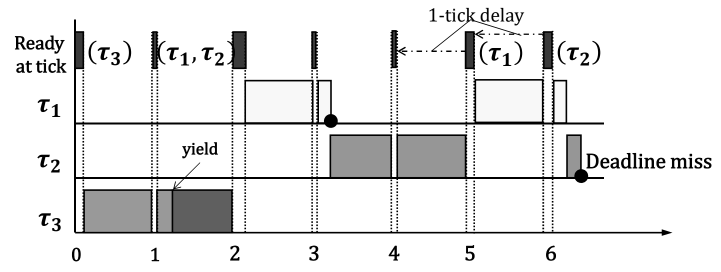
- The tick scheduler may acknowledge a job's release time one tick later than it arrived
  - This delay has negative impact on the job's response time
  - We must assume a logical place where jobs in the "*release time arrived but not yet acknowledged*" state are held
  - The time and space overhead of transferring jobs from that logical place to the ready queue is not null and must be accounted for in the schedulability test together with the time and space overhead of handling clock interrupts

## Example

$(\varphi_i, p_i, e_i, D_i)$

$T = \{\tau_1 = \{0.1, 4, 1, 4\}, \tau_2 = \{0.1, 5, 1.8, 5\}, \tau_3 = \{0, 20, 5, 20\}\}$
$\tau_3$ with a first not preemptable section of duration $1.1$

With RTA and event-driven scheduling $R_1 = 2.1, R_2 = 3.9, R_3 = 14.4$ (OK)
What with tick scheduling, clock period 1 and time overhead $0.05 + (0.06 \times n)$?



## Tick scheduling /3

- The effect of tick scheduling is captured in the RTA for job $J_i$
  - By introducing a notional task $\tau_0 = (p_0, e_0)$ at the highest priority to account for the $e_0$ cost of handling periodic clock interrupts
  - For all jobs $J_k : \pi_k \geq \pi_i$, by adding to $e_k$ the time overhead $m_0$ due to moving each of them to the ready queue
    - $(K_k + 1)$ times for the $K_k$ times that job $J_k$ may self suspend
  - For every individual jobs $J_l : \pi_l < \pi_i$, by introducing a distinct notional task $\tau_\gamma = (p_l, m_0)$ to account for the time overhead of moving them to the ready queue
  - Computing $B_i(np)$ as function of $p_0$: $J_i$ may suffer up to $p_0$ units of delay after becoming ready even without non-preemptable execution
    - $B_i(np) = (\left\lceil max_k(\frac{\theta_k}{p_0}) \right\rceil + 1)p_0$ before including non-preemption
    - Where $\theta_k$ is the maximum time of non-preemptable execution by any job $J_k$

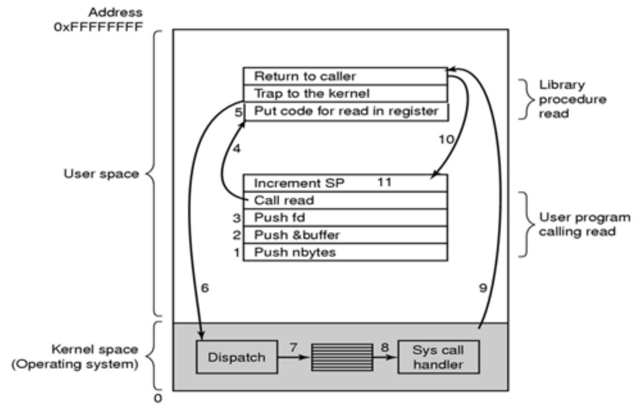
## System calls /1

- The most part of RTOS services are executed in response to direct or indirect invocations by tasks
  - These invocations are termed *system calls*
- For safety reasons, system calls may not be directly visible to the application
  - Then they would be hidden in procedure calls exported by compiler libraries
  - The library procedure does all of the preparatory work needed to make the correct invocation of the actual system call on behalf of the application

## System calls /2

- In embedded systems, the RTOS and the application often share memory
  - But they never do in general-purpose operating systems
  - Real-time embedded applications are more trustworthy and we do not want to pay the space and time overhead arising from address space separation
  - The RTOS must then protect its own data structures from the risk of race condition
- RTOS services must therefore be non-preemptable

## System calls /3



## I/O issues

- The I/O subsystem of a real-time system may require its own scheduler
- Simple methods to access an I/O resource
  - Use a run-to-completion non-preemptive FIFO policy
  - Use some kind of TDMA scheme
    - Non-preemptive quantized
- Priority-driven scheduling techniques as those in use for processor scheduling
  - RM, EDF, LLF can be used to schedule I/O requests

## Interrupt handling /1

- HW interrupts are the most efficient manner for the processor to notify the application about the occurrence of external events
  - E.g., completion of asynchronous I/O operations like DMA (direct memory access)
- Frequency and computational load of the interrupt handling activities vary with the interrupt source

## Interrupt handling /2

- For better efficiency the interrupt handling service is subdivided in an *immediate* part and a *deferred* part
  - The immediate part executes at the level of interrupt priorities, above all SW priorities
  - The deferred part executes as a normal SW activity
- The RTOS must allow the application to tell which code to associate to either part
  - Interrupt service can also have a *device-independent* part and a *device-specific* part

## Interrupt handling /3

- When the HW interface asserts an interrupt the processor saves state registers (e.g., PC, PSW) in the interrupt stack and jumps to the address of the needed *interrupt service routine* (ISR)
  - At this time interrupts are disabled to prevent race conditions on the arrival of further interrupts
  - Interrupts arriving at that time may be lost or kept pending (depending on the HW)
- Interrupts operate at an assigned level of priority so that interrupt service is subject to scheduling

## Interrupt handling /4

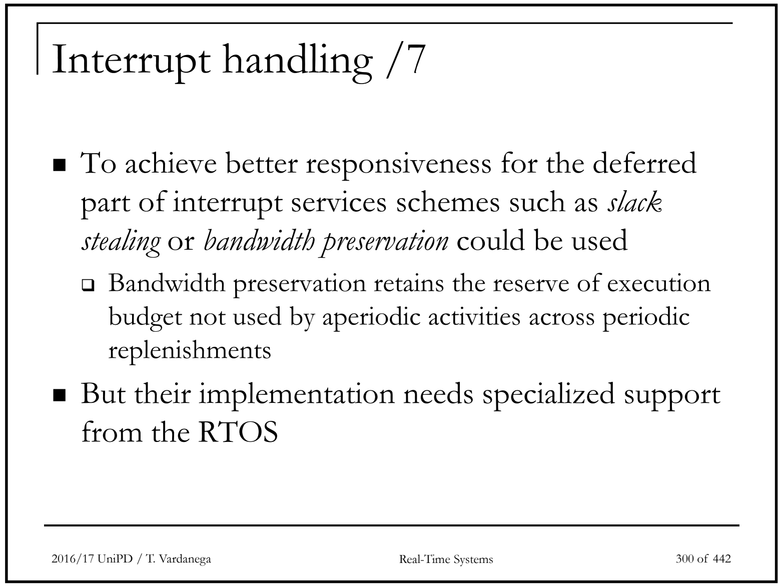
- Depending on the HW the interrupt source is determined by *polling* or via an *interrupt vector*
  - Polling is HW independent hence more generally applicable but it increases latency of interrupt service
  - Vectoring needs specialized HW but it incurs less latency
- After the interrupt source has been determined registers are restored and interrupts are enabled again

## Interrupt handling /5

- The worst-case latency incurred on interrupt handling is determined by the time needed to
  - Complete the current instruction, save registers, clear the pipeline, acquire the interrupt vector, activate the trap
  - Disable interrupts so that the ISR can be executed at the highest priority
    - This duration corresponds to *interference across interrupts*
  - Save the context of the interrupted task, identify the interrupt source and jump to the corresponding ISR
  - Begin execution of the selected ISR

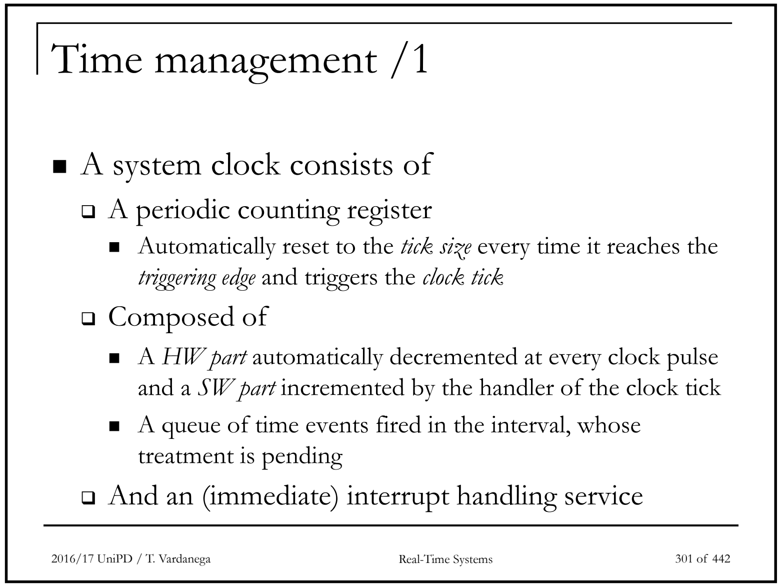
## Interrupt handling /6

- To reduce *distributed overhead*, the deferred part of the interrupt handling service must be preemptable
  - Hence it must execute at software priority
- But it still may directly or indirectly operate on RTOS data structures
  - Which must be protected by appropriate access control protocols
  - If we can do that then we do not need the RTOS to spawn its own tasks for this purpose



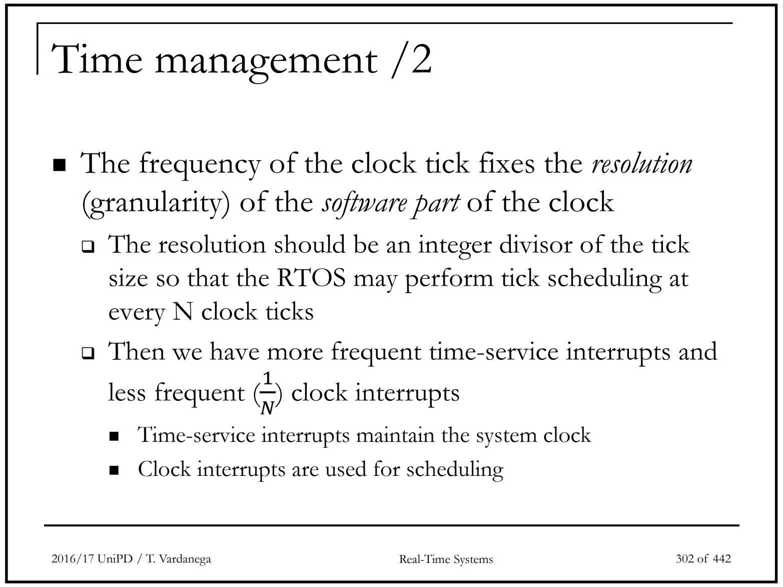
## Interrupt handling /7

- To achieve better responsiveness for the deferred part of interrupt services schemes such as *slack stealing* or *bandwidth preservation* could be used
  - Bandwidth preservation retains the reserve of execution budget not used by aperiodic activities across periodic replenishments
- But their implementation needs specialized support from the RTOS



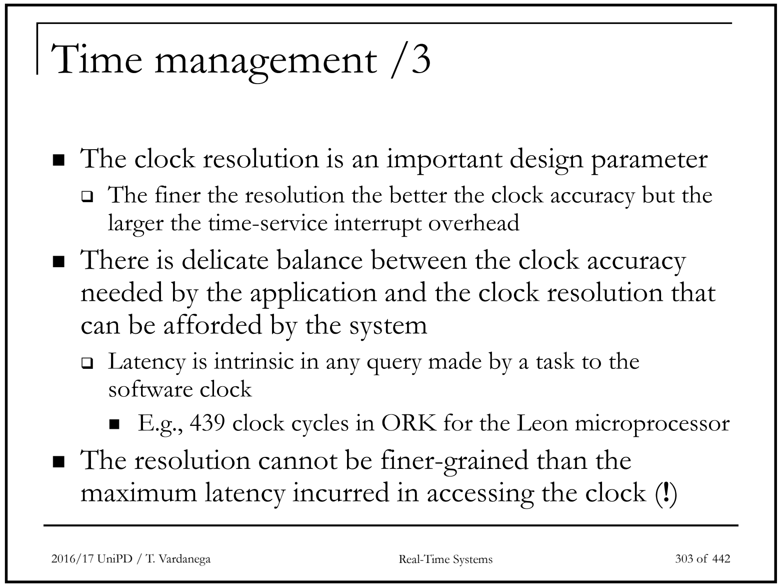
## Time management /1

- A system clock consists of
  - A periodic counting register
    - Automatically reset to the *tick size* every time it reaches the *triggering edge* and triggers the *clock tick*
  - Composed of
    - A *HW part* automatically decremented at every clock pulse and a *SW part* incremented by the handler of the clock tick
    - A queue of time events fired in the interval, whose treatment is pending
  - And an (immediate) interrupt handling service



## Time management /2

- The frequency of the clock tick fixes the *resolution* (granularity) of the *software part* of the clock
  - The resolution should be an integer divisor of the tick size so that the RTOS may perform tick scheduling at every N clock ticks
  - Then we have more frequent time-service interrupts and less frequent $(\frac{1}{N})$ clock interrupts
    - Time-service interrupts maintain the system clock
    - Clock interrupts are used for scheduling



## Time management /3

- The clock resolution is an important design parameter
  - The finer the resolution the better the clock accuracy but the larger the time-service interrupt overhead
- There is delicate balance between the clock accuracy needed by the application and the clock resolution that can be afforded by the system
  - Latency is intrinsic in any query made by a task to the software clock
    - E.g., 439 clock cycles in ORK for the Leon microprocessor
- The resolution cannot be finer-grained than the maximum latency incurred in accessing the clock (**!**)

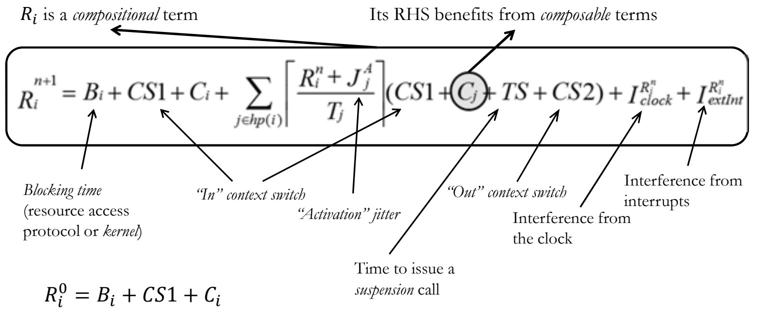
## Time management /4

- Beside periodic clocks RTOS may also support *one-shot timers* aka interval timers
  - They operate in a programmed (non-repetitive) way
- The RTOS scans the queue of the programmed time events to set the time of the next interrupt due from the interval timer
  - The resolution of the interval timer is limited by the time overhead of its handling by the RTOS
    - E.g., 7,061 clock cycles in ORK for Leon
      - www.dit.upm.es/~ork/

## Time management /5

- The accuracy of time events is the difference between the time of event occurrence and the time programmed
- It depends on three fundamental factors
  - The frequency at which the time-event queues are inspected
    - If interval timers were not used, this would correspond to the period of time-service interrupts
  - The policy used to handle the time-event queues
    - LIFO vs. FIFO
  - The time overhead cost of handling time events in the queue
- The release time of periodic tasks is exposed to jitter (**!**)

## Fine-grained response time analysis


$R_i$ is a *compositional* term
Its RHS benefits from *composable* terms
*Blocking time* (resource access protocol or *kernel*)
*"In" context switch*
*"Activation" jitter*
Time to issue a *suspension* call
*"Out" context switch*
Interference from the clock
Interference from interrupts

$$R_i^{n+1} = B_i + CS1 + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n + J_j^A}{T_j} \right\rceil (CS1 + C_j + TS + CS2) + I_{clock}^{R_j^n} + I_{extInt}^{R_i^n}$$

$$R_i^0 = B_i + CS1 + C_i$$

$$R_i = R_i^n + J^W$$

*"Wake-up" jitter*

## Summary

- RTOS design issues
- Context switch
- Priority levels
- Tick scheduling
- Interrupt handling
- Time management