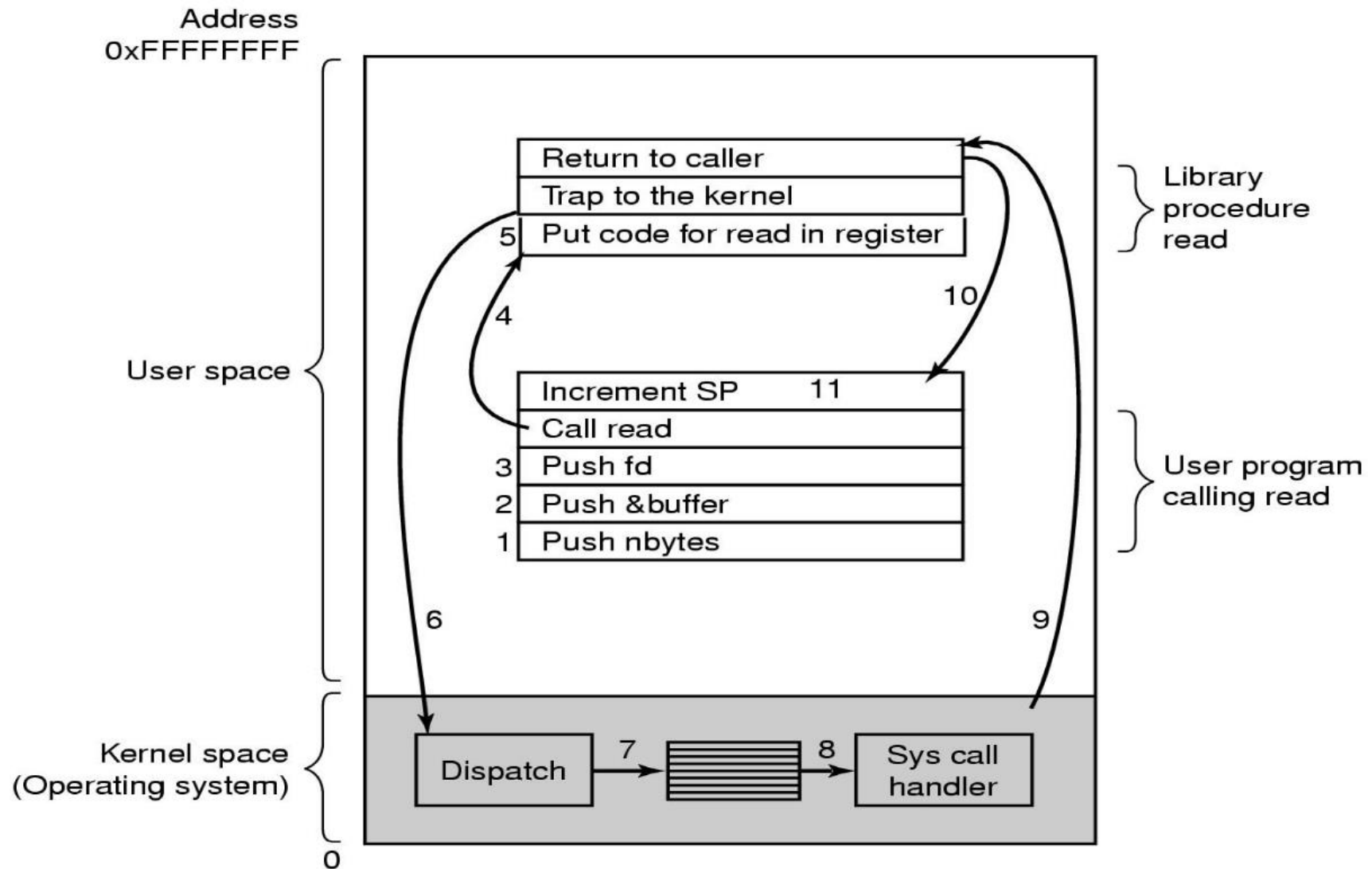

4.b A look under the hood

Where we understand how real-time programming abstractions become “real”, how much that can cost, and how RTA equations can capture their overhead

System calls /1

- Most of the RTOS services execute in response to direct or *indirect* invocations made by application tasks
 - In general-purpose systems, such invocations are termed *system calls*
- For safety reasons, the system call APIs of GPOSs are *not* directly exposed to the application
 - System calls are “hidden” in procedures exported to the programming language via compiler libraries (OS bindings)
 - Those *library procedures* do all of the preparatory work for correct invocation of the designated system call on behalf of the application
- Thanks to that “hiding”, the GPOS does *not* share memory with the application

System calls /2



System calls /3

- In embedded systems, the RTOS and the application often *share* memory
 - ❑ Address-space separation would be too costly for them
 - ❑ The RTOS is compiled *with* the application in a *single* binary
 - ❑ Hence, real-time embedded programs must be much more trustworthy than general-purpose applications
- The RTOS must protect its own data structures from the risk of race condition arising from concurrent tasks
 - ❑ RTOS services must therefore disable preemption selectively

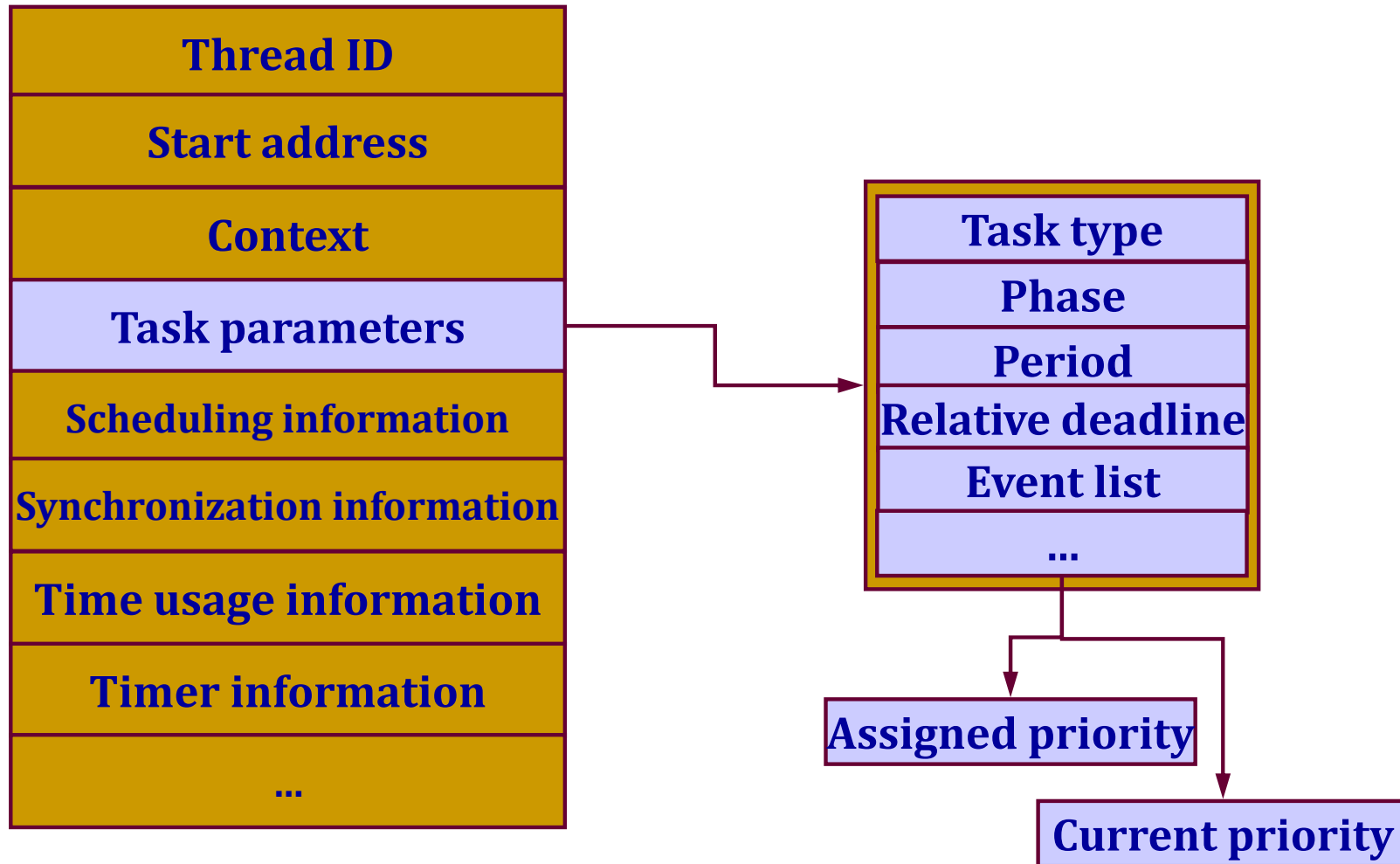
Runtime support (aka the RTOS) /1

- If the programming language provides the abstraction of application-level “task”, then the *runtime support* of that language realizes its implementation
 - As in Ada, Java, and more recently C11
- For programming languages that do *not* provide such abstraction, tasks exist *only* in the RTOS to which the implementation is bound
 - As in old-style C
- For a real-time embedded systems, the two (runtime and RTOS), by and large, are *functionally equivalent*
 - Where I am saying RTOS in the sequel, I also mean runtime

Runtime support (aka the RTOS) /2

- Application-level tasks issue jobs: now we know how
 - The jobs are the unit of CPU assignment
 - The *scheduler* decides which job gets the CPU
 - The *dispatcher* gets jobs to run and operates context switches
- The RTOS knows all tasks, and manages their life cycle
 - The task abstraction exists thanks to a descriptor: the ***Task Control Block*** (TCB)
 - One such TCB exists per task, stored in RAM
 - The insertion of a task in a state queue (e.g., ready) happens by placing a pointer from a queue place to the corresponding TCB
- End-of-life task disposal requires removing its TCB and releasing all of its memory
 - Its stack and its global data placed in the heap
- This is onerous: real-time embedded systems prefer *infinite* tasks

Task control block (example)



Runtime support (aka the RTOS) /3

- The *Model of Computation* of the application tasks defines what tasks can do and their life cycle
- The MoC should be fully determined by the RTOS
 - Outside or inside of the programming language, contingent on the binding of it with the RTOS
 - For Ada, we know it is *inside* of it
- At one extreme, the MoC may also be defined by the user, making “creative” use of the RTOS API
 - Very risky: the user determines whether the execution semantics of the program eventually conforms with the assumptions made in feasibility analysis

Runtime support (aka the RTOS) /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 suspension call until the next release

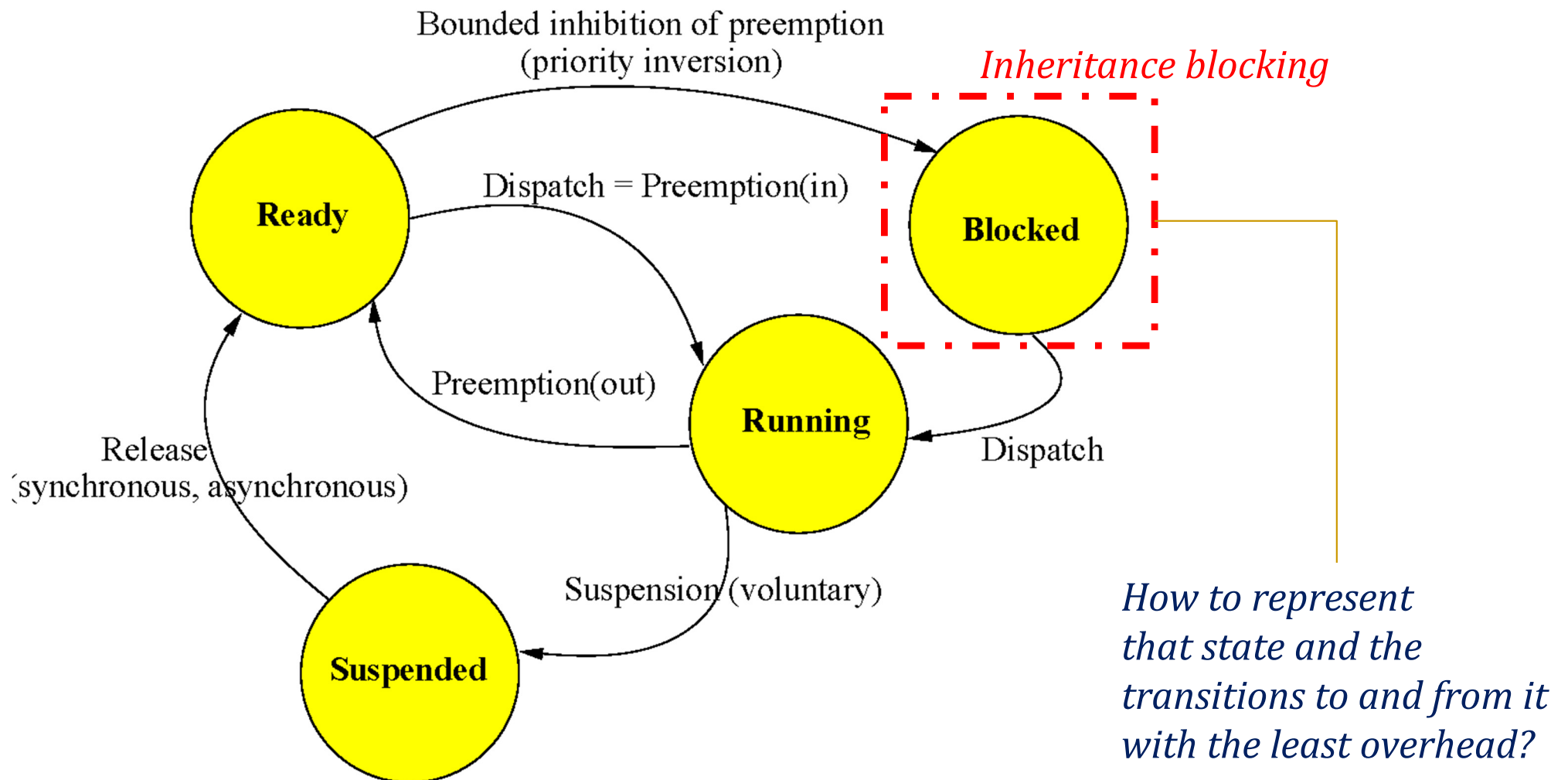
■ Sporadic task

- An RTOS thread whose suspension point is not released periodically but with *guaranteed* minimum distance
 - After release, it executes the job and then calls a wait-for-signal service

■ 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



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 suspension
 - An *event-based queue* for sporadic suspension
 - How to assure minimum separation between subsequent releases?
 - The *inversion of control* pattern that we have seen in the model discussed earlier allows doing that transparently

Context switch /1

- At time t of execution, the processor has a *context* defined by the contents of its *registers* at that time
- As tasks share the processor transparently to one another, their progress of execution at time t is captured by their own *context*
- The task context (which would be the thread context in a GPOS) is comprised of
 - The processor context
 - The task execution memory (stack, heap, task control block)
- Upon preemption, the context of the outgoing task is saved in RTOS memory, and replaced by the context of the incoming task

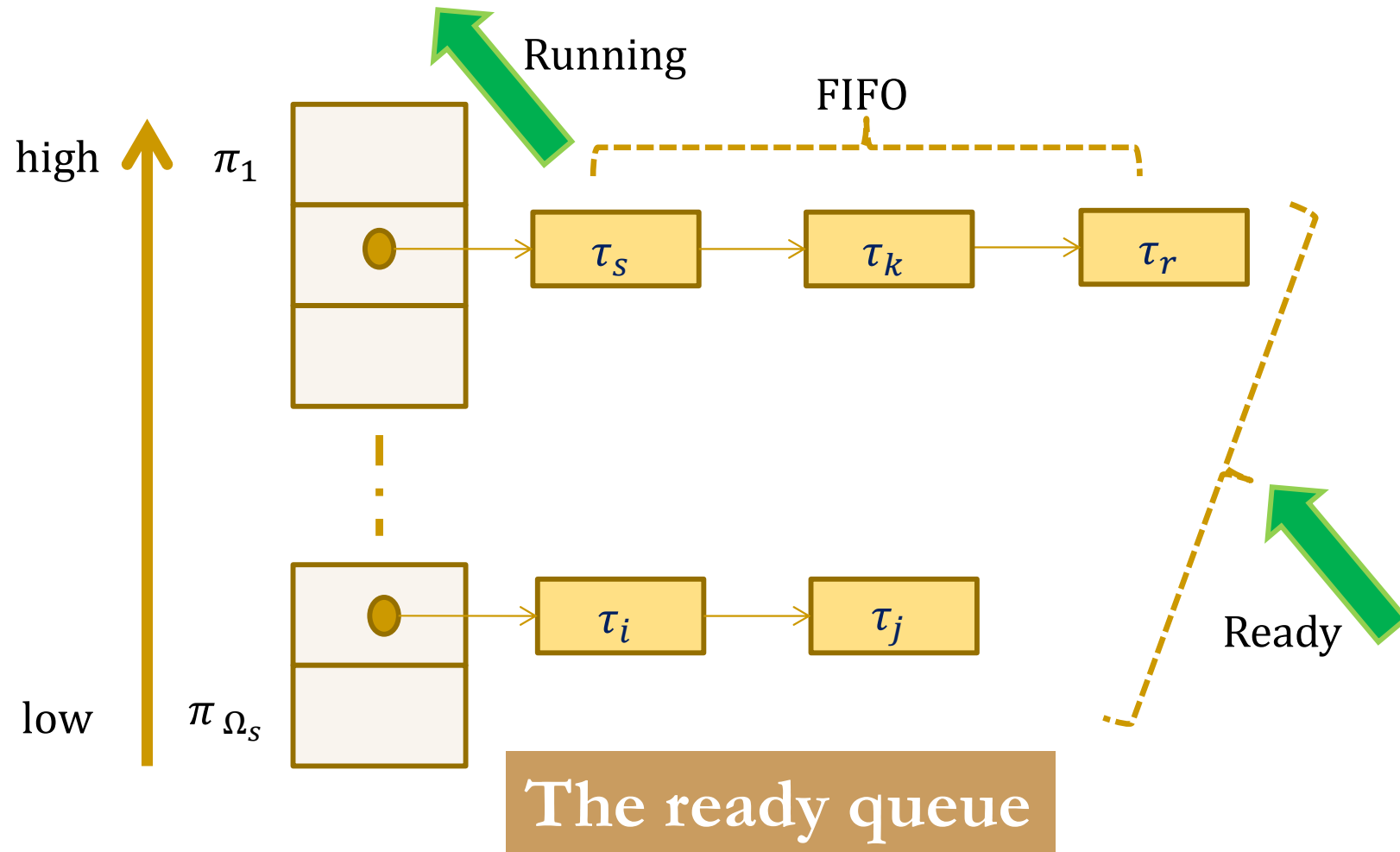
Context switch /2

- The time and space overhead incurred at preemption may be considerable
 - ❑ Should be accounted for in schedulability analysis
- Under preemptive scheduling, every task run incurs no less than *two* context switches
 - ❑ At activation, to install its execution context
 - ❑ At resumption after preemption (if any), to restore it
 - ❑ At completion, to clean it up
- The corresponding time cost should be charged to the WCET of the task's jobs
 - ❑ This requires knowing the internals of RTOS

Priority levels /1

- The feasibility analysis techniques that we have studied assume tasks (and jobs) to have *distinct* priorities
 - Each index in the RTA equations denotes a *single* task
- Concrete systems may *not* have sufficient priorities
 - In that case, jobs may have to *share* priority levels
- For jobs at the same level of priority, we might use FIFO or round-robin
 - FIFO is *better* in the RT domain: predictability wins over fairness!
- If priority levels were shared, we would have a worsening of worst-case situation to contemplate in the analysis
 - Job J might be released last after all other jobs at its level of priority

Example: FIFO within priorities



Priority levels /2

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

$$\omega_i(t) = e_i + B_i + \sum_{S(i)} e_{j \in S(i)} + \sum_{k=1, \dots, i-1} \left\lceil \frac{\omega_i(t)}{p_k} \right\rceil e_k$$

- This obviously worsens J_i 's response time
 - The notion of ***schedulability loss*** helps quantify the penalization that results from that additional overhead

Priority levels /3

- When the *assigned priorities* $\mathcal{R} = [1, \dots, \Omega_n]$ exceed the *available priorities* $\mathcal{H} = [\pi_1, \dots, \pi_{\Omega_s}]$, ($\|\mathcal{R}\| > \|\mathcal{H}\|$), we need a $\Omega_n: \Omega_s$ mapping function to collapse \mathcal{R} into \mathcal{H}
- The resulting function is known as the *priority grid*
 - All assigned priorities in range $(0, \pi_1]$ will take value π_1
 - For $1 < k \leq \Omega_s$, the assigned priorities in range $(\pi_{k-1}, \pi_k]$ will take value π_k
- Two main techniques are used to produce such grids
 - **Uniform mapping**
 - **Constant ratio mapping** [Lehoczky & Sha, 1986]

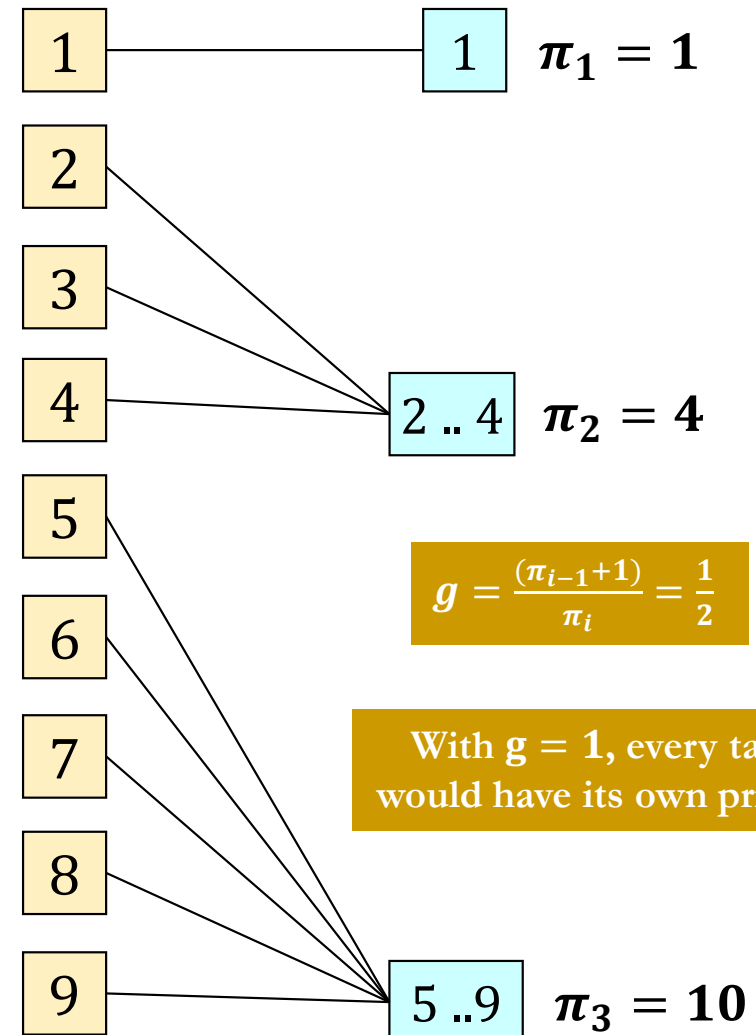
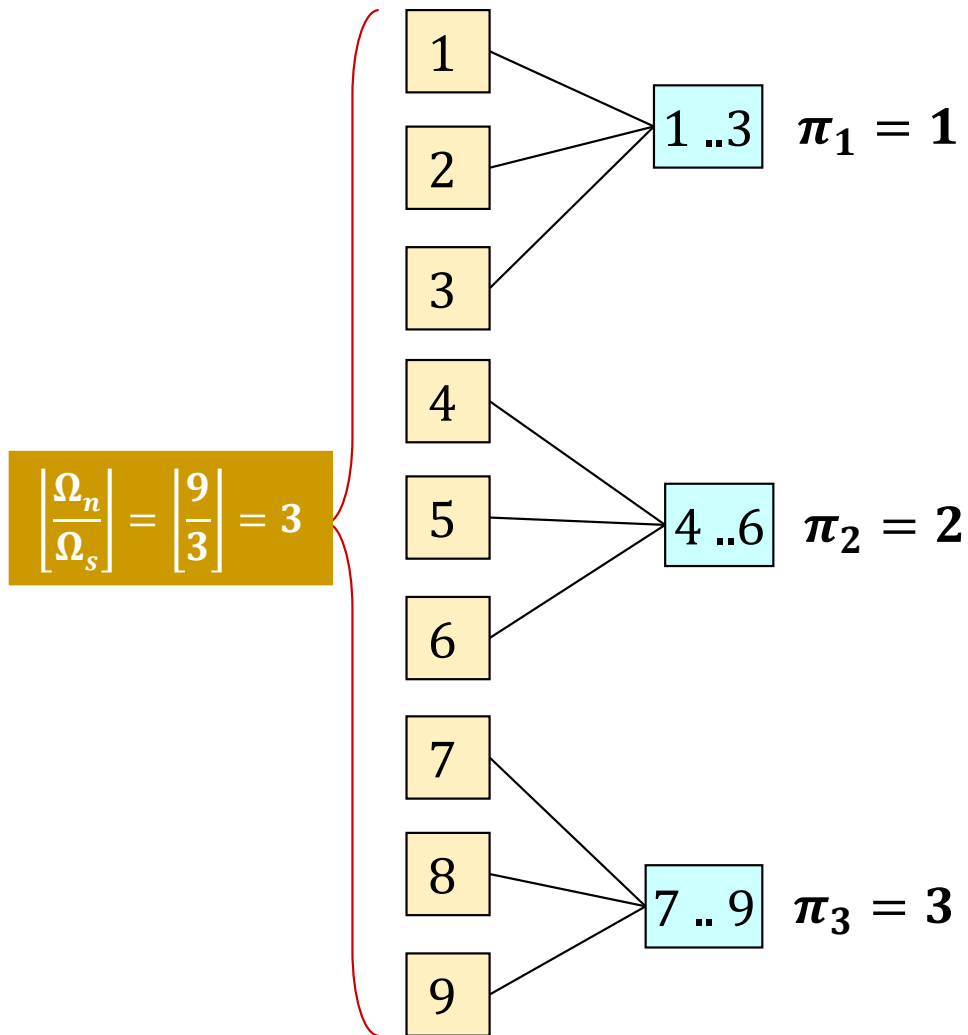
Priority levels /4

- **Uniform mapping** ($Q = \left\lfloor \frac{\Omega_n}{\Omega_s} \right\rfloor$)
 - $\pi_k \leftarrow [k, \dots, kQ]; \pi_{k+1} \leftarrow [kQ + 1, \dots, (k+1)Q]; k = 1, \dots, \Omega_s - 1$
 - **Example** ($\Omega_n = 9, \Omega_s = 3, Q = \left\lfloor \frac{9}{3} \right\rfloor = 3$)
 $\pi_1 = 1, \pi_2 = 2, \pi_{\Omega_s=3} = 3 \Rightarrow \pi_1 \leftarrow [1..3], \pi_2 \leftarrow [4..6], \pi_3 \leftarrow [7..9]$
- **Constant ratio mapping** ($g = \frac{(\pi_{i-1}+1)}{\pi_i} : i = 2, \dots, \Omega_s$)
 - Collapses subsets of \mathcal{R} into π_i values of \mathcal{H} in a *logarithmic grid* such that the ratio $0 < g \leq 1$ of two adjacent points in \mathcal{H} is constant
 - **Example** ($\Omega_n = 9, \Omega_s = 3, g = \frac{1}{2}$)
 $\pi_1 = 1, \pi_2 = 4, \pi_3 = 10 \Rightarrow \pi_1 \leftarrow [1], \pi_2 \leftarrow [2..4], \pi_3 \leftarrow [5..9]$

Priority levels /5

Uniform mapping

Constant ratio mapping



Priority levels /6

- The constant-ratio mapping degrades the schedulable utilization of RMS *gracefully*
 - For large n , implicit deadlines, and $g = \min_{1 < j \leq \Omega_s} \frac{(\pi_{j-1} + 1)}{\pi_j}$, the schedulable utilization that CRM can achieve approximates
$$f(g) = \begin{cases} \ln(2g) + 1 - g, & \frac{1}{2} < g \leq 1 \\ g, & 0 < g \leq \frac{1}{2} \end{cases}$$
- The $\frac{f(g)}{\ln(2)}$ ratio represents the limit of *relative schedulability* of CRM in relation to RMS' utilization bound

Example

$$\Omega_s = 256, \Omega_n = 100,000 \rightarrow g \cong 0.956, \frac{f(g)}{\Omega_n \left(2^{\frac{1}{\Omega_n}} - 1\right)} = 0.9986$$

256 priority levels (a one-byte value) should suffice for RMS

Time management /1

- The *system clock* abstraction is composed of
 - A HW part decremented by one unit at every clock pulse, as determined by the clock rate
 - A *periodic-counting register*, automatically reset to a default *tick size* when it reaches the *triggering edge* (0), and trips the *clock tick*
 - A SW part incremented by SW at the clock tick
 - The system clock effectively counts clock ticks
 - A queue of time events, fired and not serviced
 - Pending until they are serviced
 - A handler of clock-tick interrupts
 - Which increments the clock-tick counter and, every $N > 0$ occurrences, also services the pending time events

Time management /2

- The frequency of the clock tick determines the *resolution* (granularity) of the system clock
 - It should be an integer divisor of the tick size so that the RTOS may service time events at *exactly* every $N \in INT$ clock ticks
- Clock-tick interrupts maintain the system clock
 - More frequent, tolerable overhead
- One such interrupt in N handles scheduling events
 - Less frequent, high overhead

Time management /3

- The clock resolution is an important design parameter
 - The finer the resolution the better the clock accuracy, at the cost of a higher interrupt overhead
- There must be a sound 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 to read the clock
 - The ORK runtime for the Leon microprocessor takes 493 clock cycles to read the clock (www.dit.upm.es/~ork/)
 - @ 40 MHz, 500 clock cycles correspond to 12.5 μsec
- The clock resolution cannot be finer-grained than the worst-case latency incurred reading the clock (!)

Time management /4

- Beside periodic clocks, if the processor allows, the RTOS may also support *one-shot* (aka interval) *timers*
 - ❑ They operate in a programmed (non-repetitive) way so that time events suffer *no* latency from resolution problems
 - ❑ The RTOS scans the queue of the programmed time events to set the next interrupt alarm due from the interval timer
- Interval timers are costly
 - ❑ They have to be written by SW and the value to set depends on the time events pending in the queue
 - ❑ Their resolution is limited by the time overhead of its handling by the RTOS: 7,061 clock cycles in ORK for Leon

Time management /5

- The accuracy of a *time event* is the delta between when the time set and when the event triggers
- It depends on three fundamental factors
 - The frequency at which the time-event queues are inspected
 - Without interval timer, it would be at every N clock ticks
 - With interval timer, it would be at every interval expiry
 - The policy used to service the time-event queues
 - Expiry-based, LIFO, FIFO
 - The time overhead cost of handling the event queue
- *The release time of periodic tasks is naturally exposed to jitter (!)*

The scheduler /1

- This is a distinct part of the RTOS that does *not* execute in response to explicit application invocations
 - Except when using cooperative scheduling
- The scheduler acts when the ready queue changes
 - The corresponding time events are termed *dispatching points*
- When the MoC is defined *outside* of the programming language, the scheduler “activation” is periodic in response to clock interrupts
 - The poor RTOS has no other way to know when to schedule!

The scheduler /2

- If execution-time-based scheduling (e.g. LLF) is used
 - The scheduler must increment the execution-time budget counter of the running job at every clock interrupt
 - Possibly service the queue of time-based events pending
 - Possibly attend to the ready queue
- GPOS have a tick size in the region of **10 ms**
 - This is *much too coarse-grained* for RTOS, but too high frequency incurs excessive overhead
- The scheduler should support event-driven execution with *minimum latency*

Tick scheduling / 1

- The tick scheduler may acknowledge a job's release time up to one clock 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
 - It must be accounted for in the schedulability test along with the overhead of handling clock interrupts

Example

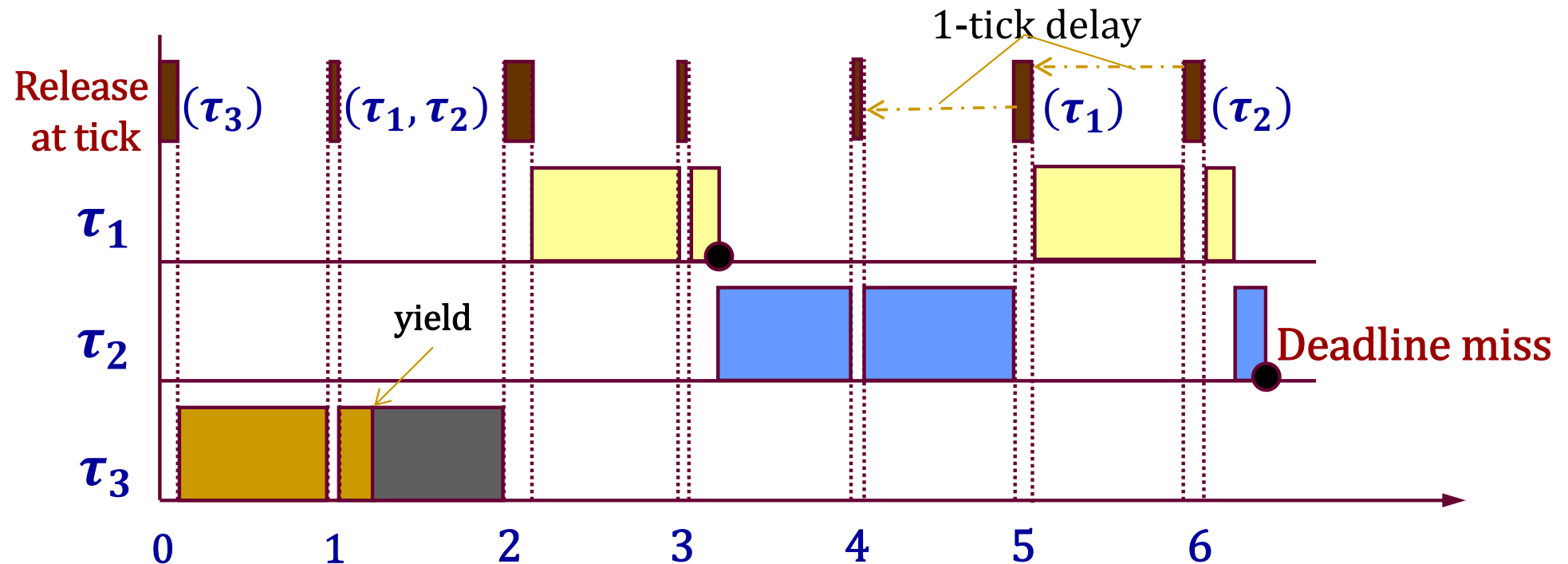
$(\varphi_i, p_i, e_i, D_i)$

$T = \{\tau_1 = \{0.1, 4, 1, 4\}, \tau_2 = \{0.1, 5, 1.8, 5\}, \tau_3 = \{0, 20, 5, 20\}\}$

τ_3 with a first no-preemption section of duration 1.1 time units

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 $0.05 + (0.06 \times n)$ per tick handling and queue movement?



Tick scheduling /2

- The effect of tick scheduling is captured in RTA for job J_i by
 - Introducing a notional task $\tau_0 = (p_0, e_0)$ with highest priority, to account for the e_0 cost of handling clock interrupts with period p_0
 - For every job $J_k : \pi_k \geq \pi_i$, adding to e_k the time overhead m_0 due to moving J_k to the ready queue
 - $(K_k + 1)$ times for the K_k times that job J_k may self suspend
 - For every job $J_l : \pi_l < \pi_i$, introducing a distinct notional task $\tau_l = (p_l, m_0)$ to account for the time cost of moving J_l 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-preemptive execution
 - $B_i(np) = (\lceil \max_k (\frac{\theta_k}{p_0}) \rceil + 1)p_0$ before including non-preemption
 - Where θ_k is the maximum time of non-preemptive execution by any job J_k

I/O subsystems

- When the I/O subsystem is an *active* resource (in the taxonomy seen in §1), it would need its own scheduler
- Methods to serve I/O access requests may employ
 - Run-to-completion non-preemptive FIFO semantics
 - Non-preemptive time-division (quantized) schemes
 - Priority-driven scheduling as seen for CPU scheduling

Interrupt handling /1

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

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 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 incurs scheduling if interrupts nest

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
- Once the interrupt source is determined, registers are restored and interrupts are enabled again

Interrupt handling / 5

- The worst-case latency of interrupt handling is determined by the time needed to perform the following actions
 1. Complete the current instruction
 2. Save the processor registers and the general context of the task being interrupted
 3. Clear the processor pipeline
 4. Acquire the interrupt vector
 5. Activate the trap to kernel mode (for kernels with more privileges)
 6. Disable interrupts, so that the immediate part of the ISR can execute at the highest priority
 7. Identify the interrupt source and jump to the corresponding ISR
 8. Begin execution of the selected ISR

Interrupt handling / 6

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

Interrupt handling /7

- Using the code patterns we saw in §4.a, the deferred part of interrupt handling would map to a *sporadic* task released by the immediate part of the ISR
- For better responsiveness, *aperiodic servers* could be used
 - ❑ So that total interference from interrupts is still bounded, but a given quota of them may receive full service within replenishment intervals
 - ❑ During those intervals, bandwidth preservation retains the unused reserve of execution budget, which can help serve occasional bursts
- These solutions need *specialized* support from the RTOS

Putting it all together

R_i is a *compositional* term

Its RHS benefits from *composable* terms

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

Blocking time
(resource access
protocol or *kernel*)

“In” context switch

“Activation” jitter

“Out” context switch

Interference from
the clock

Interference from
interrupts

Time to issue a
suspension call


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

$$R_i = R_i^n + J^W \text{ “Wake-up” jitter}$$

Summary

- We have seen how an RTOS (or runtime) supports the application-level abstractions that recur in the real-time systems theory
- We have appreciated how complex those abstraction services may be
- We have understood that they may cause latency in the occurrence of scheduling and dispatching events
- We have realized that their impact should be captured in the Response-Time Analysis equation

Selected readings

- T. Vardanega, J. Zamorano, J.A. de la Puente (**2005**), *On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels*
DOI: 10.1023/B:TIME.00000048937.17571.2b
- Again ... 
- P. Carletto, T. Vardanega (**2017**), *Ravenscar-EDF: Comparative Benchmarking of an EDF Variant of a Ravenscar Runtime*
DOI: 10.1007/978-3-319-60588-3_2