Time-composable software for compositional schedulability analysis

Tullio Vardanega (19-30 May 2014)
University of Padova, Italy

30 June - 1 July 2014
TACLe Summer School

1. Introduction

Definitions

- ***Time Composability***
  - A property that applies to individual components (HW and SW): the bound on the timing behaviour of a component, in response to an execution request, can be determined independently of that component's composition in a system
- ***Time Compositionality***
  - A property that applies to items of a collection that individually hold certain properties: some property of the collection of those items can be determined as a function of the properties of each item considered in isolation and from their composition

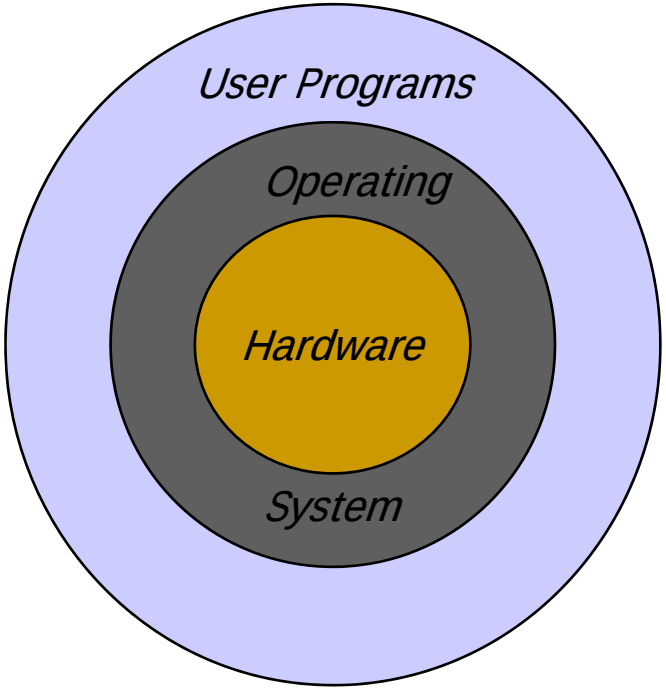
Initial intuition /1

- **Real-time system – I**
  - An aggregate of computers, I/O devices and application-specific software, all characterized by
    - Intensive interaction with external environment
    - Time-dependent variations in the state of the external environment
    - Need to keep control over all individual parts of the external environment and to react to changes
  - System activities subject to timing constraints
    - Reactivity, accuracy, duration, completion, responsiveness: all dimensions of ***timeliness***
  - System activities are inherently concurrent
  - The satisfaction of such constraints must be proved

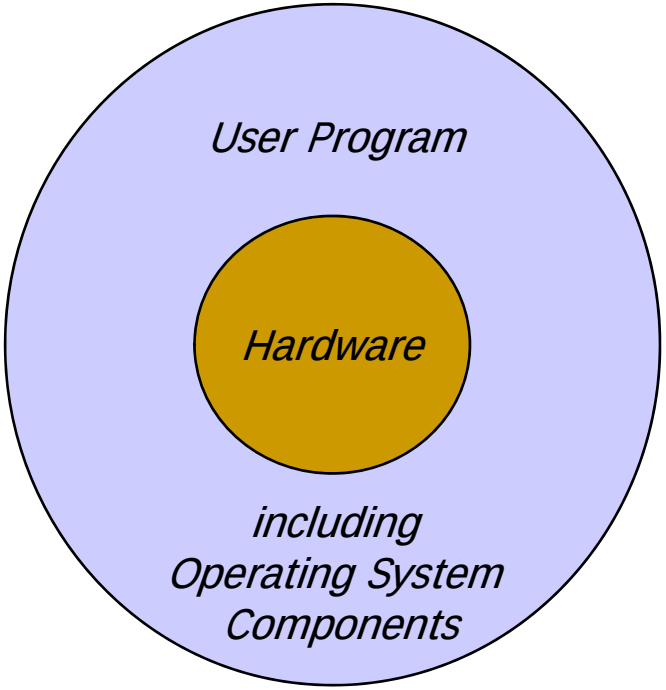
Initial intuition /2

- **Real-time system – II**
  - Operational correctness does not solely depend on the logical result but also on the time at which the result is produced
    - The computed response has an application-specific utility function
    - Correctness is defined in the value domain and in the time domain
    - A logically-correct response produced later than due may be as bad as a wrong response
- **Embedded system**
  - The computer and its software are fully immersed in an engineering system comprised of the external environment subject to its control

Embedded system



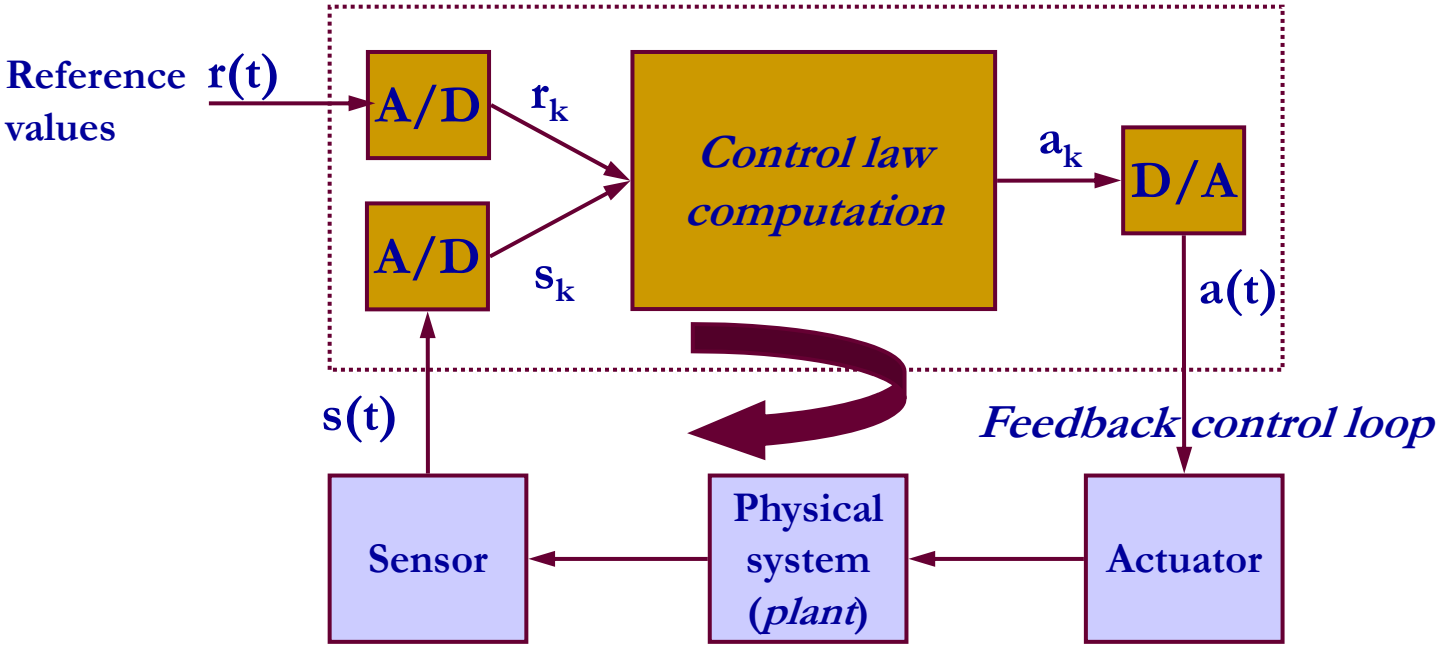
Typical General-Purpose Computing Configuration



Typical Embedded Computing Configuration

Example /1

- A digital system of sensors and actuators

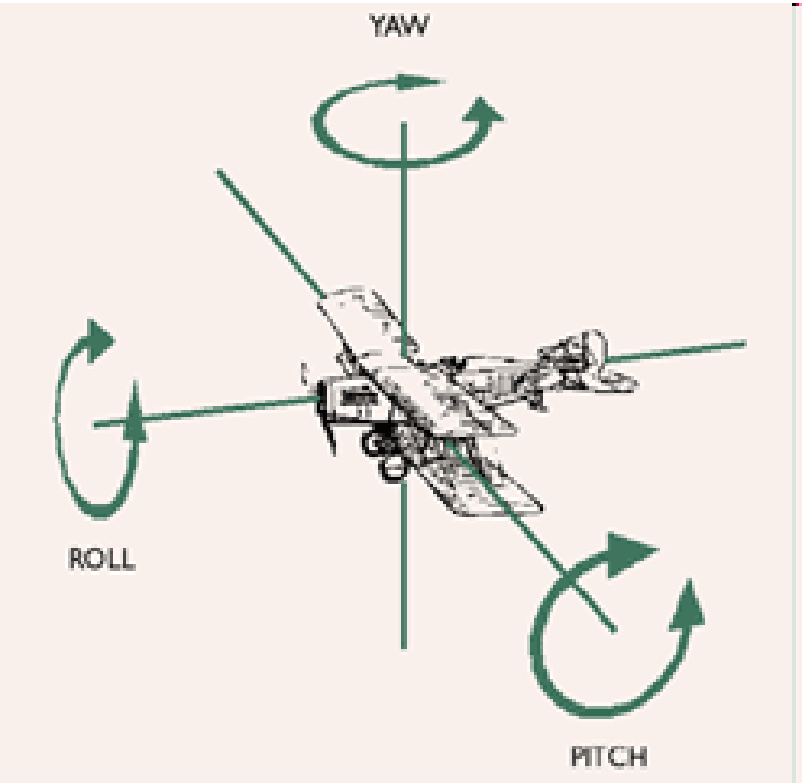


$$a_k = a_{k-2} + \alpha(r_k - s_k) + \beta(r_{k-1} - s_{k-1}) + \gamma(r_{k-2} - s_{k-2})$$

Example /2

- Factors of influence
  - Quality of response (**responsiveness**)
    - Sensor sampling is typically periodic (for convenience)
    - Actuator commanding is produced at the time of the next sampling
      - As part of feedback control mathematics
    - System stability degrades with the width of the sampling period
  - Plant **capacity**
    - Good-quality control reduces oscillations
    - A system that needs to react rapidly to environmental changes and is capable of it within rise time R requires higher frequency of actuation and thus faster sampling hence shorter period T
    - A "good" R/T ratio ranges [10 .. 20]

Example /3



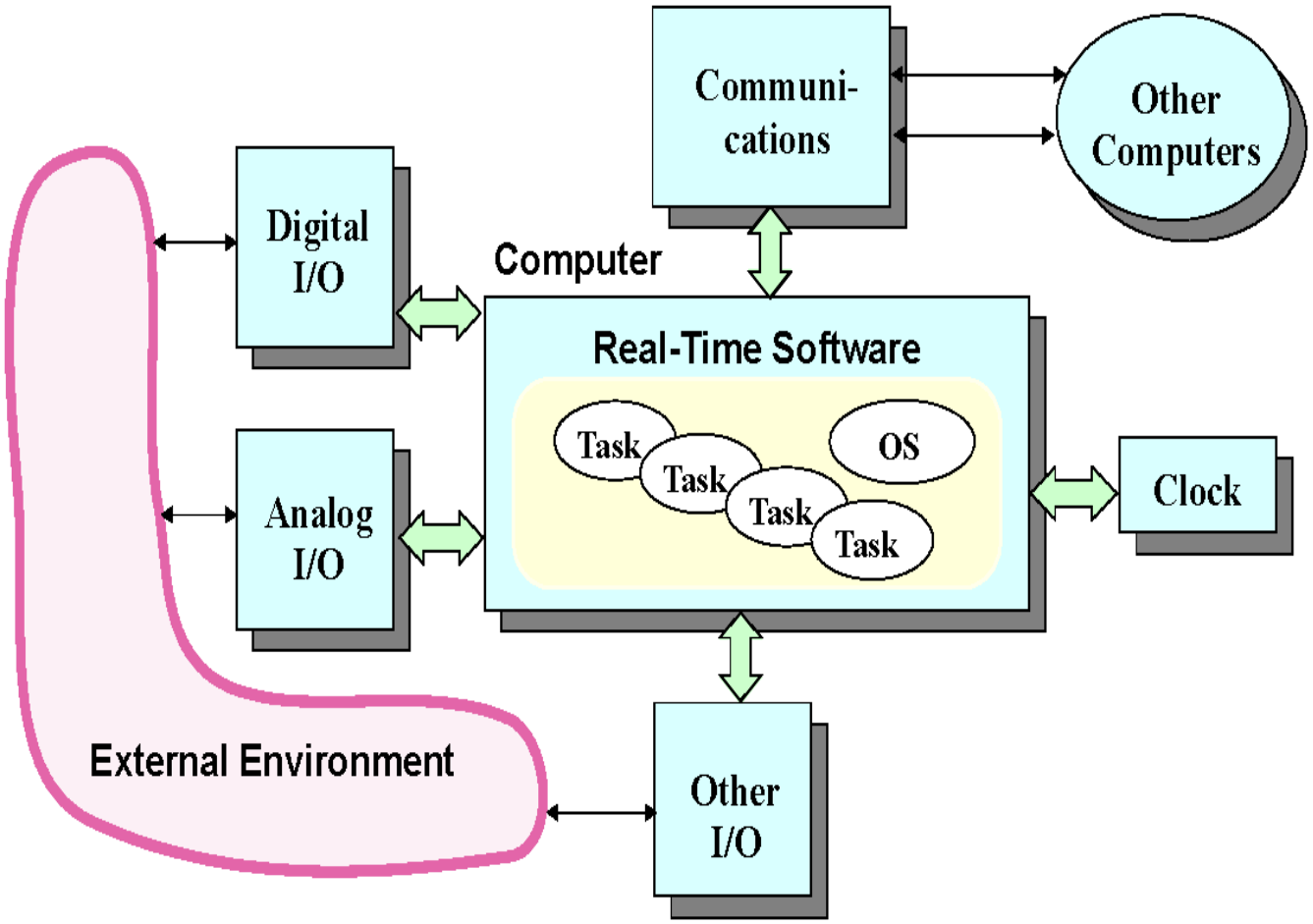
Any three-dimensional rotation can be described as a sequence of roll (x), pitch (y) and yaw (z) rotations

- Complex systems must support multiple distinct periods T_i
  - It is convenient to set a **harmonic** relation between all T_i
    - This removes the need for concurrency of execution in the relevant computations
    - But it causes coupling between possibly unrelated control actions which is a poor architectural choice
  - There may be diverse components of speed
    - *Forward, side slip, altitude*
  - As well as diverse components of rotation
    - *Roll, pitch, yaw*
  - Each of them requires separate control activities each performed at a specific rate

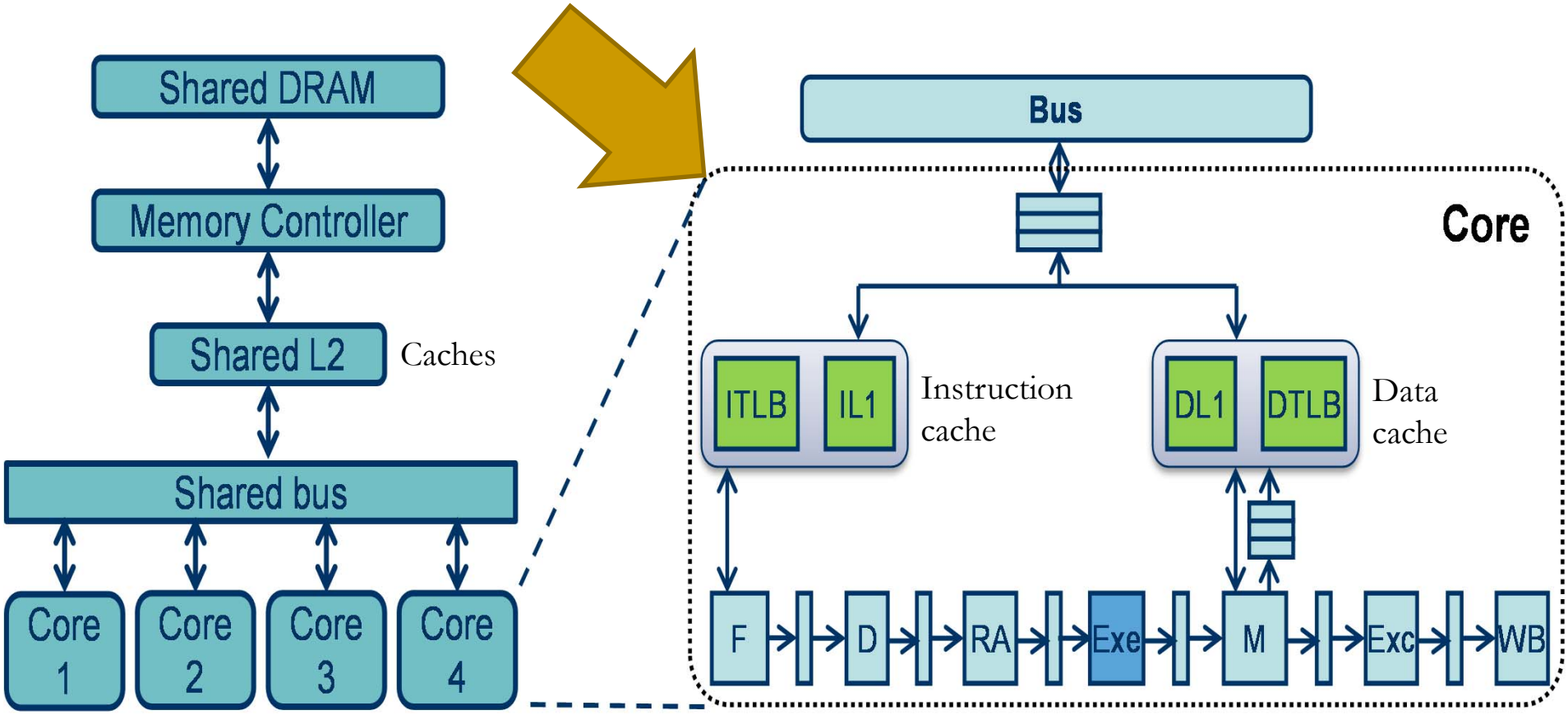
Example /4

- 180 Hz cycle (*harmonic multi-rate* functions)
  - Check all sensor data and select sources to sample
  - Reconfigure system in case of read error
- 90 Hz cycle (at every 2nd activation)
  - Perform control law for pitch, roll, yaw (internal loop)
  - Command actuators
  - Perform sanity check
- 30 Hz cycle (at every 6th activation)
  - Perform control law for pitch, roll, yaw (external loop) and integration
- 30 Hz cycle (at every 6th activation)
  - Capture operator keyboard input and choice of operation model
  - Normalize sensor data and transform coordinates; update reference data

Understanding the system

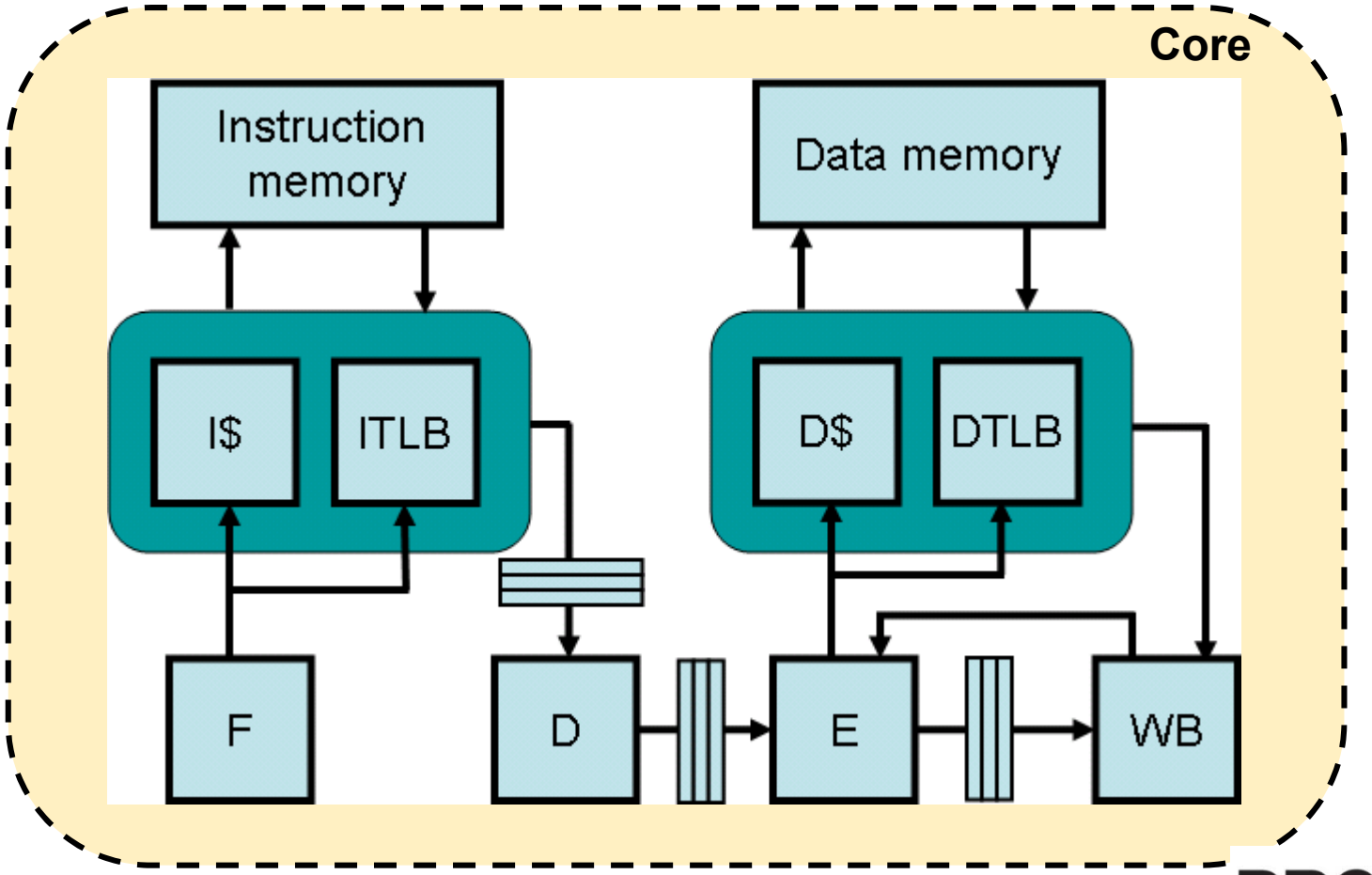


Understanding the hardware /1



Courtesy of PROXIMA

Understanding the hardware /2



Courtesy of PROXIMA

An initial taxonomy /1

- The prevailing classification stems from the traditional standpoint of control algorithms
  - **Strictly periodic** systems
    - Harmonic multi-rate (artificially harmonized)
    - Polling for not-periodic events
  - **Predominantly (but not exclusively) periodic** systems
    - Lower coupling
    - Better responsiveness to not-periodic events
  - **Predominantly not-periodic systems** but still **predictable**
    - Events arrive at variable times but within bounded intervals
  - **Not-periodic and unpredictable** systems
    - Another ballgame!

Definitions /2

- ***Task***
  - Unit of functional and architectural composition
  - Issues jobs (one at a time) to perform actual work
    - One such task is said to be *recurrent*
- ***Job***
  - Unit of work selected for execution by the scheduler
  - Needs physical and logical *resources* to execute
  - Each job has an entry point where it awaits activation

An initial taxonomy /2

- ***Periodic*** tasks
  - Their jobs become ready at regular interval of time
  - Their arrival is synchronous to some time reference
- ***Aperiodic*** tasks
  - Recurrent but irregular
  - Their arrival cannot be anticipated (asynchronous)
- ***Sporadic*** tasks
  - Their jobs become ready at variable times but at bounded minimum distance from one another

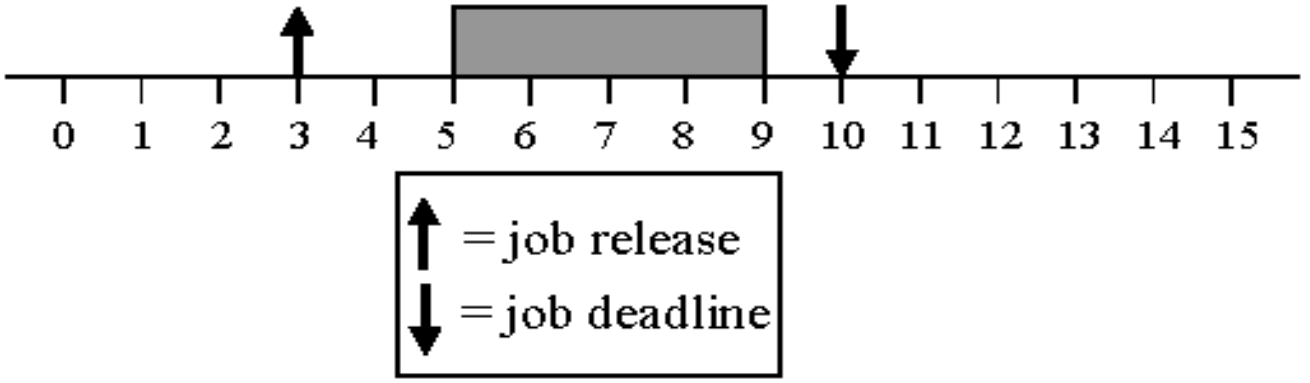
Definitions /3

- ***Release time***
  - When a job should become eligible for execution
    - The corresponding trigger is called ***release event***
    - There may be some temporal delay between the arrival of the release event and when the scheduler actually recognizes the job as ready
  - May be set at some offset from the system start time
    - The offset of the first job of task τ is named ***phase*** and it is an attribute of τ

Definitions /4

- ***Deadline***
  - The time by which a job must complete its execution
    - For example, by the next release time
  - May be < (*constrained*), = (*implicit*), > (*arbitrary*) than the job's next release time
- ***Response time***
  - The span of time between the job's release time and its actual completion
  - The longest admissible response time for a job is termed the job's *relative deadline*
- The algebraic summation of release time and relative deadline is termed *absolute deadline*

Example



Job is released at time 3.
It's (absolute) deadline is at time 10.
It's relative deadline is 7.
It's response time is 6.

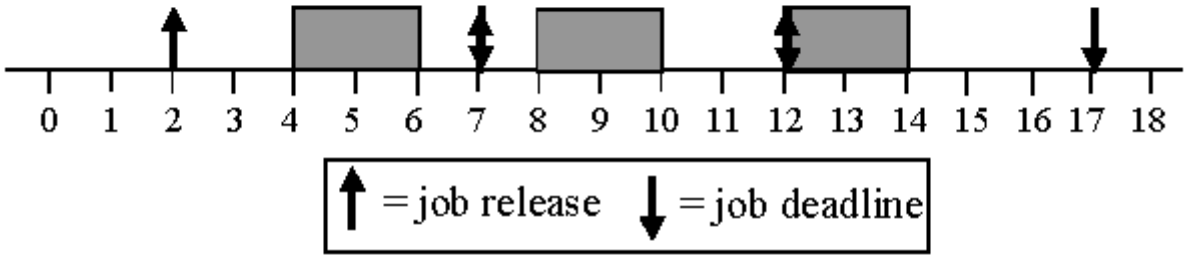
Temporal parameters

- ***Jitter***
  - Variability in the release time or in the time of input (data freshness) or output (stability of control)
- ***Inter-arrival time***
  - Separation between the release time of successive jobs which are not strictly periodic
    - Job is ***sporadic*** if a guaranteed minimum value exists
    - Job is ***aperiodic*** otherwise
- ***Execution time***
  - Vary between *best-case* (BCET) and *worst-case* (WCET)

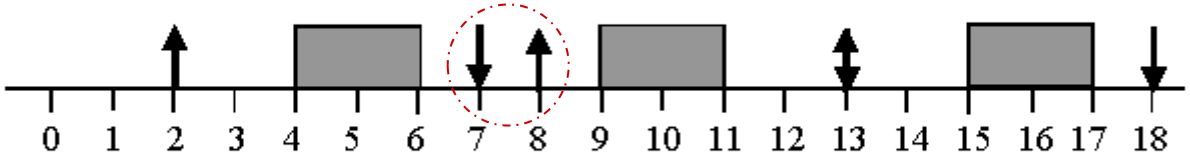
Periodic task and sporadic task

Examples

A periodic task T_i with $r_i = 2$, $p_i = 5$, $e_i = 2$, $D_i = 5$ executes like this according to the rest of the world:



According to Liu, it could execute like this:



To the rest of the world, this is a sporadic task.

Jim Anderson Real-Time Systems Introduction - 26

Abstract models

- ***Periodic model***
  - Comprises periodic and sporadic jobs
  - Accuracy of representation decreases with increasing jitter and variability of execution time
  - ***Hyperperiod*** H_S of task set $S = \{\tau_i\}, i = 1, \ldots, N$
    - LCM (least common multiple) of periods $\{T_i\}$
  - ***Utilization***
    - For every task τ_i : ratio between execution time and period : $U_i = \frac{C_i}{T_i}$
    - For the system (*total utilization*) : $U = \sum_i U_i$

Abstract models /2

- Selecting jobs for execution
  - The scheduler assigns a job to the processor resource
    - Notice we are talking single core here
  - The resulting assignment is termed ***schedule***
  - A schedule is ***valid*** if
    - Each processor is assigned to at most 1 job at a time
    - Each job is assigned to at most 1 processor at a time
    - No job is scheduled before its release time
    - The scheduling algorithm ensures that the amount of processor time assigned to a job is no less than its BCET and no more than its WCET
    - All precedence constraints in place among tasks as well as among resources are satisfied

Abstract models /3

- A valid schedule is said to be ***feasible*** if the temporal constraints of every job are all satisfied
- A job set is said to be ***schedulable*** by a scheduling algorithm if that algorithm always produces a valid schedule for that problem
- A scheduling algorithm is ***optimal*** if it always produces a feasible schedule when one exists
- Actual systems may include multiple schedulers that operate in some hierarchical fashion
  - E.g., some scheduler governs access to logical resources; some other schedulers govern access to physical resources

Abstract models /4

- Two algorithms are of prime interests for real-time systems
  - The *scheduling algorithm* that we should like to be optimal
    - Comparatively easy problem
  - The *analysis algorithm* that tests the *feasibility* of applying a scheduling algorithm to a given job set
    - Much harder problem
- The scientific community, but not always in full consistency, divides the analysis algorithms in
  - ***Feasibility tests***, which are exact
    - Necessary and sufficient
  - ***Schedulability tests***, which are only sufficient

2. Scheduling issues

Common approaches /1

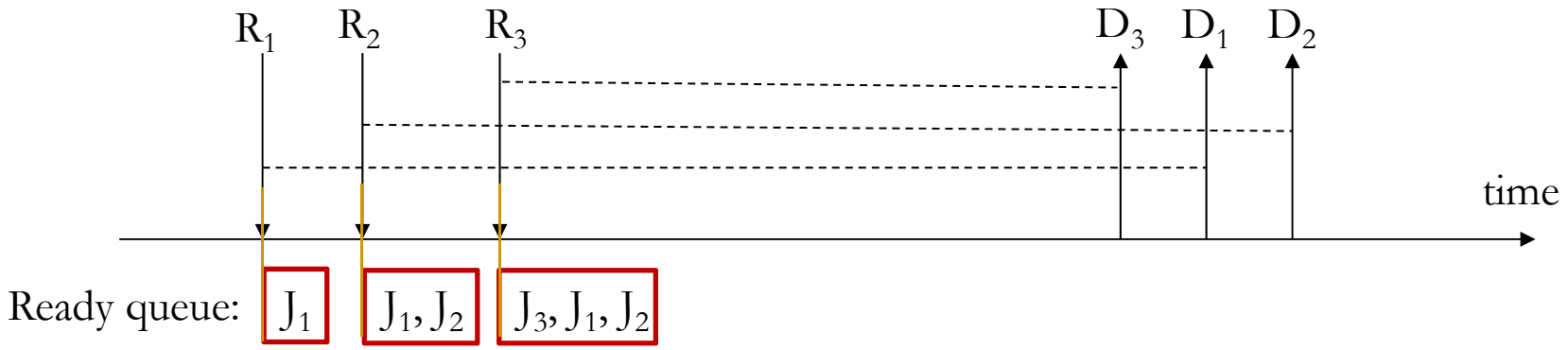
- ***Clock-driven (time-driven) scheduling***
  - Scheduling decisions are made beforehand (off line) and carried out at predefined time instants
    - The time instants normally occur at regular intervals signaled by a clock interrupt
    - The scheduler first dispatches jobs to execution as due in the current time period and then suspends itself until then next schedule time
    - The scheduler uses an off-line schedule to dispatch
  - All parameters that matter must be known in advance
  - The schedule is static and cannot be changed at run time
  - The run-time overhead incurred in executing the schedule is minimal

Common approaches /2

- ***Priority-driven (event-driven) scheduling***
  - This class of algorithms is *greedy*
    - They never leave available processing resources unutilized
      - Seeking local optimization
    - An available resource may stay unused iff there is no job ready to use it
    - A *clairvoyant* alternative may instead defer access to the CPU to incur less contention and thus reduce job response time
    - Anomalies may occur when job parameters change dynamically
  - Scheduling decisions are made at run time when changes occur to the "ready queue", hence on local knowledge
    - The event causing a scheduling decision is called "*dispatching point*"
  - It includes algorithms also used in non real-time systems
    - FIFO, LIFO, SETF (shortest e.t. first), LETF (longest e.t. first)
      - Normally applied at every round of RR scheduling

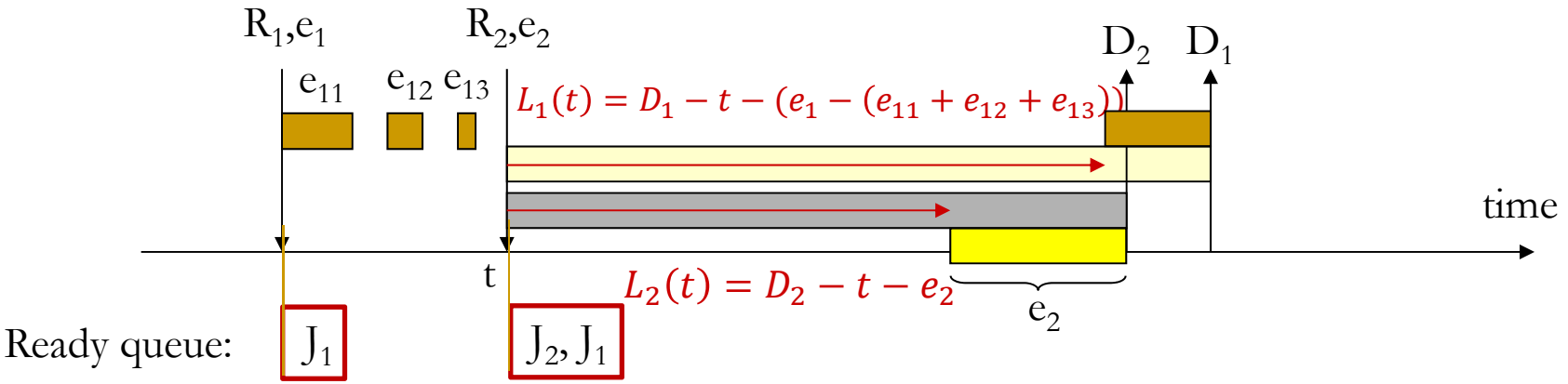
Optimality /1

- Priorities assigned in accord to (effective) deadlines
  - ***Earliest Deadline First*** scheduling is *optimal* for single processor systems with independent jobs and preemption
    - For any given job set, EDF produces a feasible schedule if one exists
    - The optimality of EDF falls short under other hypotheses (e.g., no preemption, multicore processing)

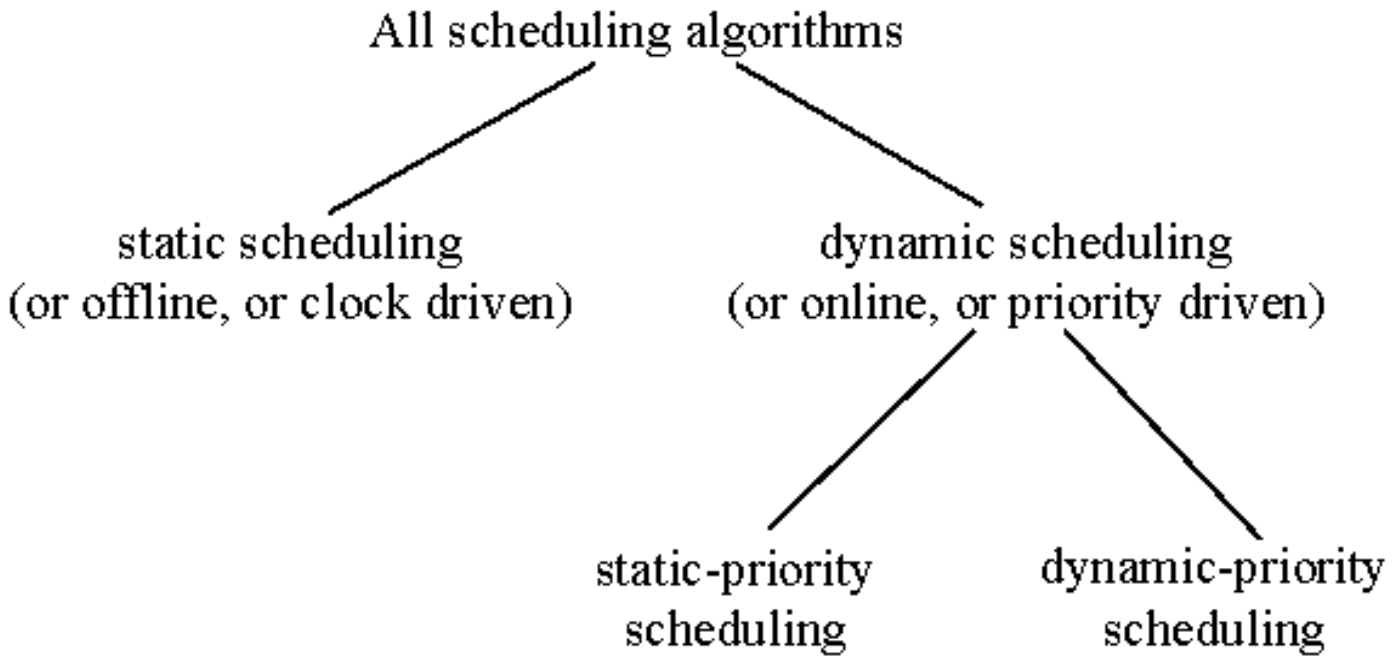


Optimality /2

- Priorities assigned in accord to *slack* (i.e., *laxity*)
  - ***Least Laxity First*** scheduling is optimal under the same hypotheses as for EDF optimality
    - LLF is far more onerous than EDF to implement as it has to keep tab of execution time!

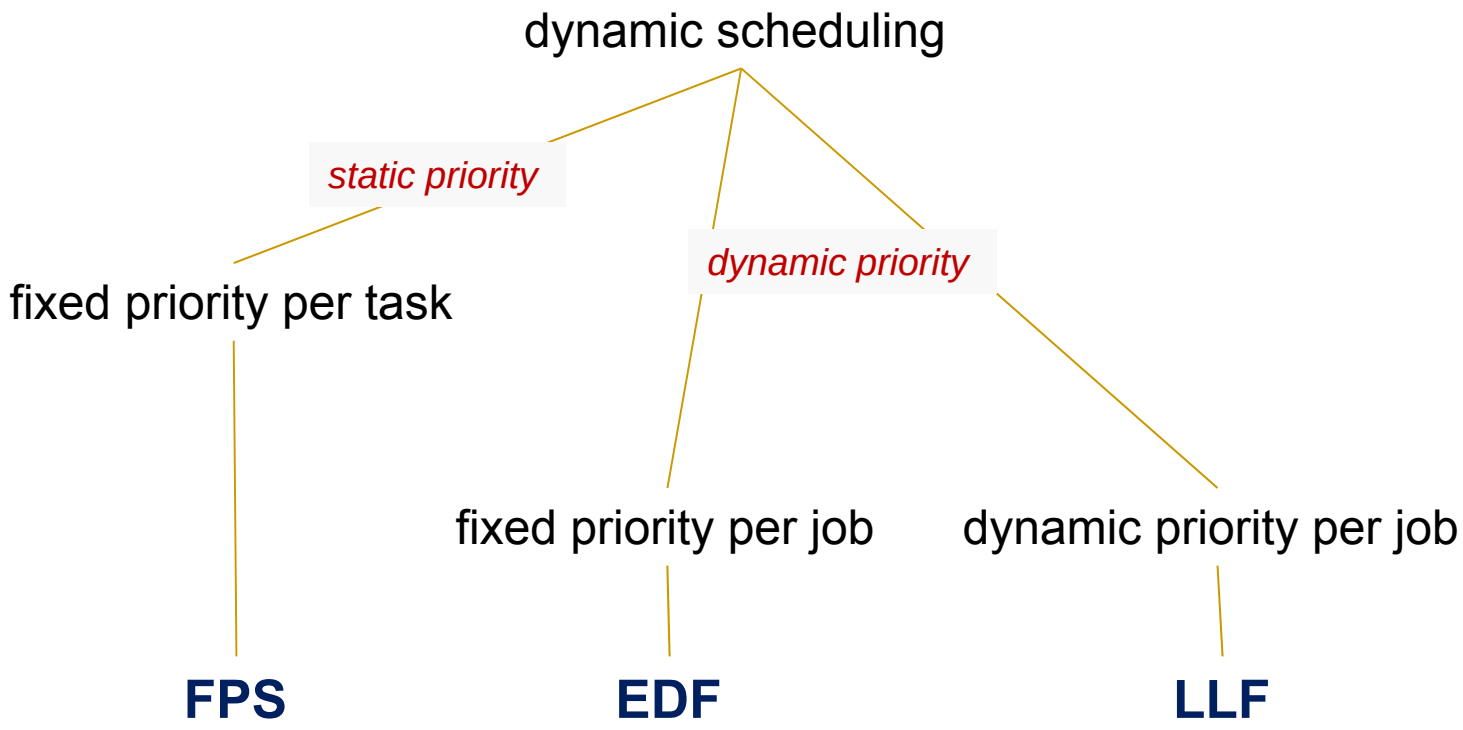


Classification of Scheduling Algorithms



Jim Anderson Real-Time Systems Introduction - 30

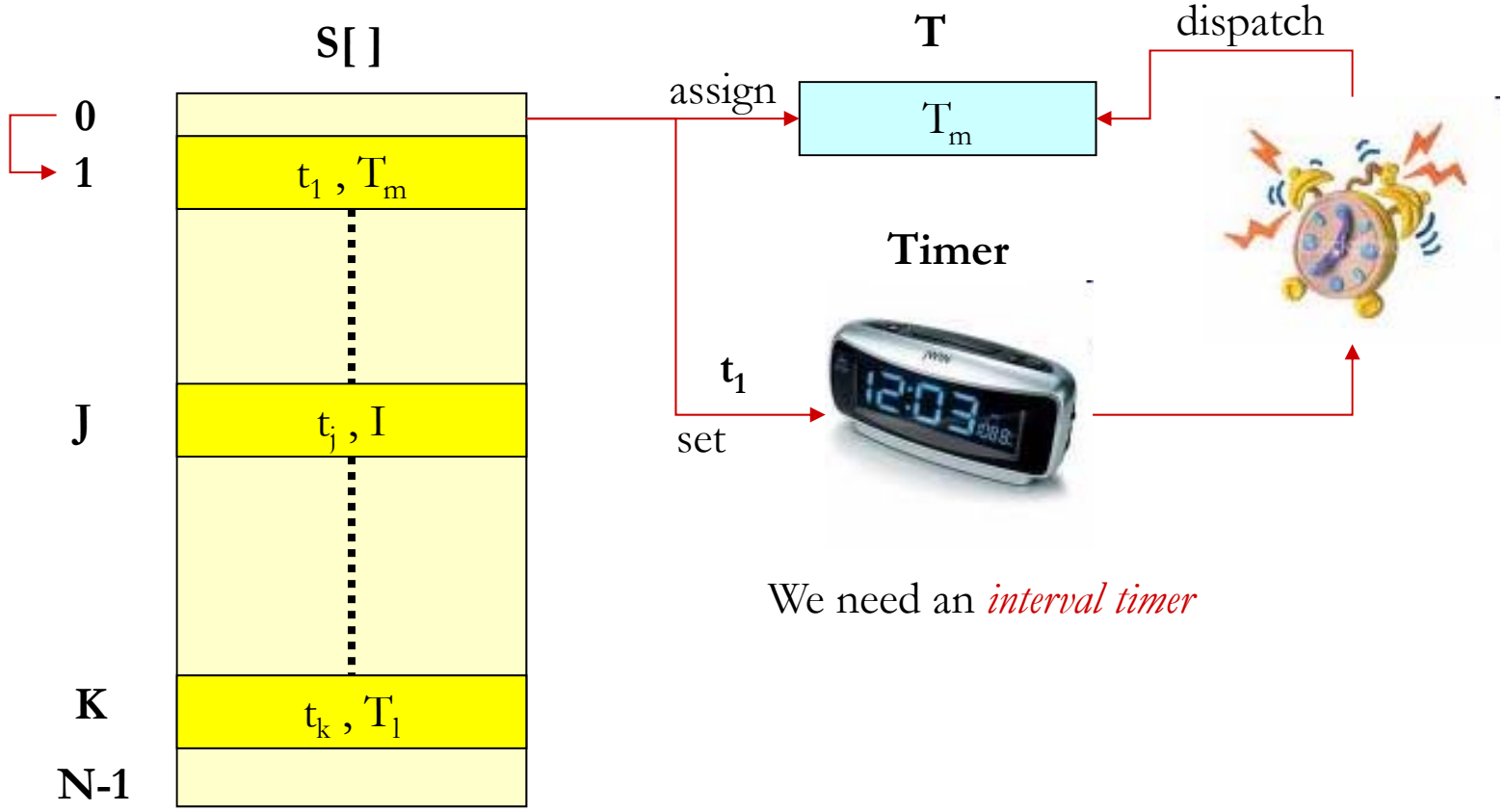
Ramifications for dynamic scheduling



Clock-driven scheduling /1

- ***Workload model***
  - N periodic tasks with N constant and statically defined
    - In Jim Anderson's definition of periodic (not Jane Liu's)
  - The $(\varphi_i, p_i, e_i, D_i)$ parameters of every task τ_i are constant and statically known
- The schedule is static and committed off line before system start to a table **S** of decision times t_k
  - $S[t_k] = \tau_i$ if a job of task τ_i must be dispatched at time t_k
  - $S[t_k] = I$ (*idle*) otherwise
  - Schedule computation can be as sophisticated as we like since we pay for it only once and before execution
  - Jobs cannot overrun otherwise the system is in error

Clock-driven scheduling /2

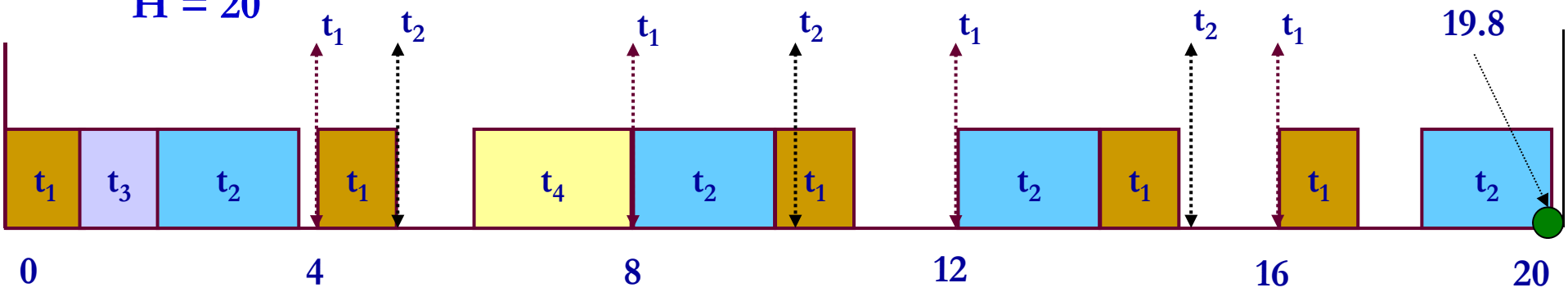


We need an *interval timer*

Example

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

J = {t_1 = (0, 4, 1, 4), t_2 = (0, 5, 1.8, 5), t_3 = (0, 20, 1, 20), t_4 = (0, 20, 2, 20}
U = 0.76
H = 20



- Static schedule table S for J would need 17 entries
  - That's too many and too fragmented!
- **Why 17?**

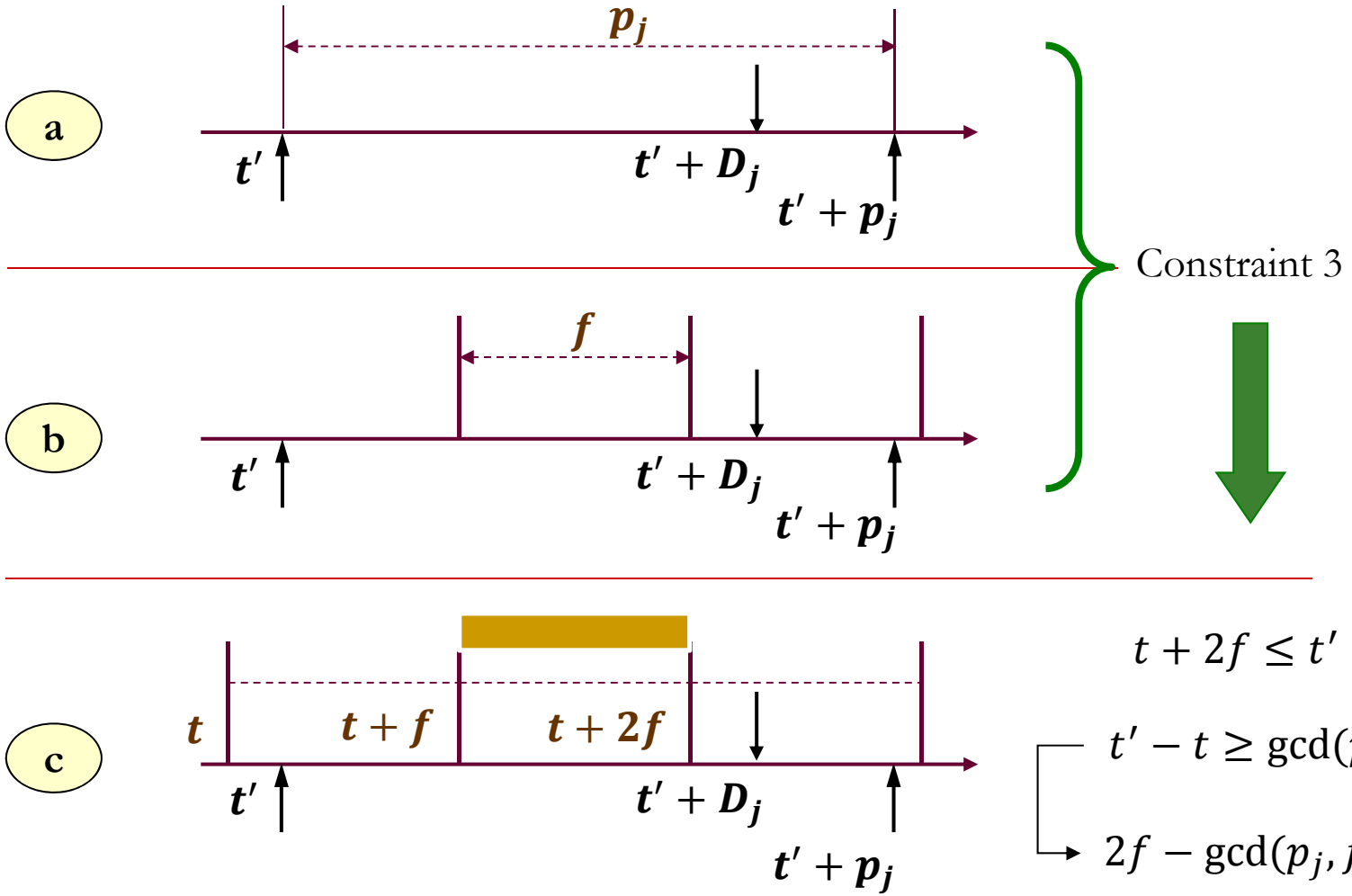
Clock-driven scheduling /3

- Obvious reasons suggest we should minimize the size and complexity of the cyclic schedule (table S)
  - The scheduling point t_k should occur at regular intervals
    - Each such interval is termed ***minor cycle*** (*frame*) and has duration f
    - We need a *periodic timer*
    - Within minor cycles there is no preemption but a single minor cycle may contain the execution of multiple (run-to-completion) jobs
  - φ_i for every task τ_i must be a non-negative integer multiple of f
    - The first job of every task has release time (forcedly) set at the beginning of a minor cycle
- We must therefore enforce some artificial constraints

Clock-driven scheduling /4

- **Constraint 1**: Every job J must complete within f
  - $f \geq max_{i=\{1,..n\}}(e_i)$ so that *overruns* can be detected
- **Constraint 2**: f must be an integer divisor of hyper-period H : $H = Nf$ where N is an integer
  - Satisfied if f is an integer divisor of at least one task period p_i
  - The hyper-period beginning at minor cycle kf for $k = 0,..N-1$ is termed ***major cycle***
- **Constraint 3**: There must be one *full* frame f between J's release time t' and its deadline: $t' + D_j \geq t + 2f$ so that J can be scheduled in that frame
  - This can be expressed as: $2f - \gcd(p_i, f) \leq D_i$ for every task τ_i

Understanding constraint 3



Example

- $T = \{(0, 4, 1, 4), (0, 5, 2, 5), (0, 20, 2, 20)\}$
- $H = 20$
- [c1] : $f \geq \max(e_i)$: $f \geq 2$
- [c2] : $\lfloor p_i/f \rfloor - p_i/f = 0$: $f = \{2, 4, 5, 10, 20\}$
- [c3] : $2f - \gcd(p_i, f) \leq D_i$: $f \leq 2$

$f = 2 : 4 - \gcd(4,2) \leq 4$ **OK**
$4 - \gcd(5,2) \leq 5$ **OK**
$4 - \gcd(20,2) \leq 20$ **OK**

$f = 4 : 8 - \gcd(4,4) \leq 4$ **OK**
$8 - \gcd(5,4) \leq 5$ **KO**

$f = 5 : 10 - \gcd(4,2) \leq 4$ **KO**

$f = 10 : 20 - \gcd(4,2) \leq 4$ **KO**

$f = 20 : 40 - \gcd(4,2) \leq 4$ **KO**

Clock-driven scheduling /5

- It is very likely that the original parameters of some task set T may prove unable to satisfy all three constraints for any given *f* simultaneously
- In that case we must decompose T's jobs by ***slicing*** their larger e_{max} into fragments small enough to artificially yield a "good" *f*

Clock-driven scheduling /6

- To construct a cyclic schedule we must therefore make three design decisions
  - Fix an f
  - Slice (the large) jobs
  - Assign (jobs and) slices to minor cycles
- There is a very unfortunate inter-play among these decisions
  - Cyclic scheduling thus is very fragile to any change in system parameters

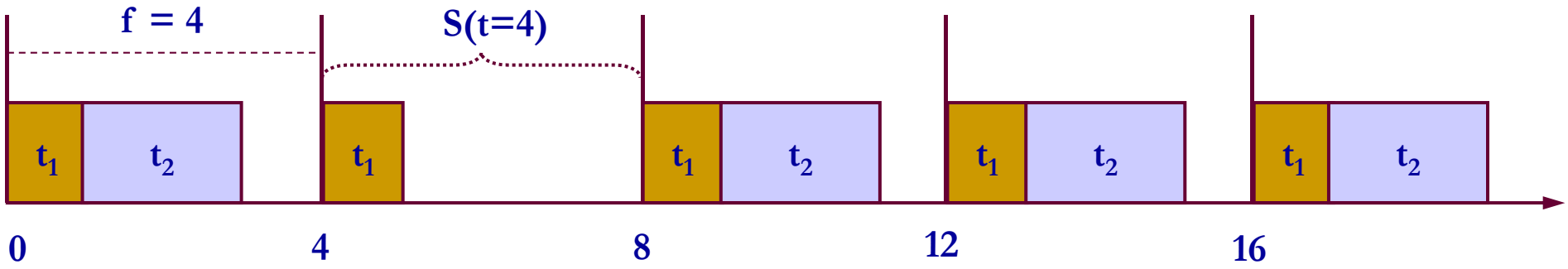
Example (slicing) – 1/2

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

$$J = \{\tau_1 = (0, 4, 1, 4), \tau_2 = (0, 5, 2, 7), \tau_3 = (0, 20, 5, 20)\}, H = 20$$

τ_3 causes disruption since we need $e_3 \leq f \leq 4$ to satisfy c3

We must therefore slice e_3 : how many slices do we need?

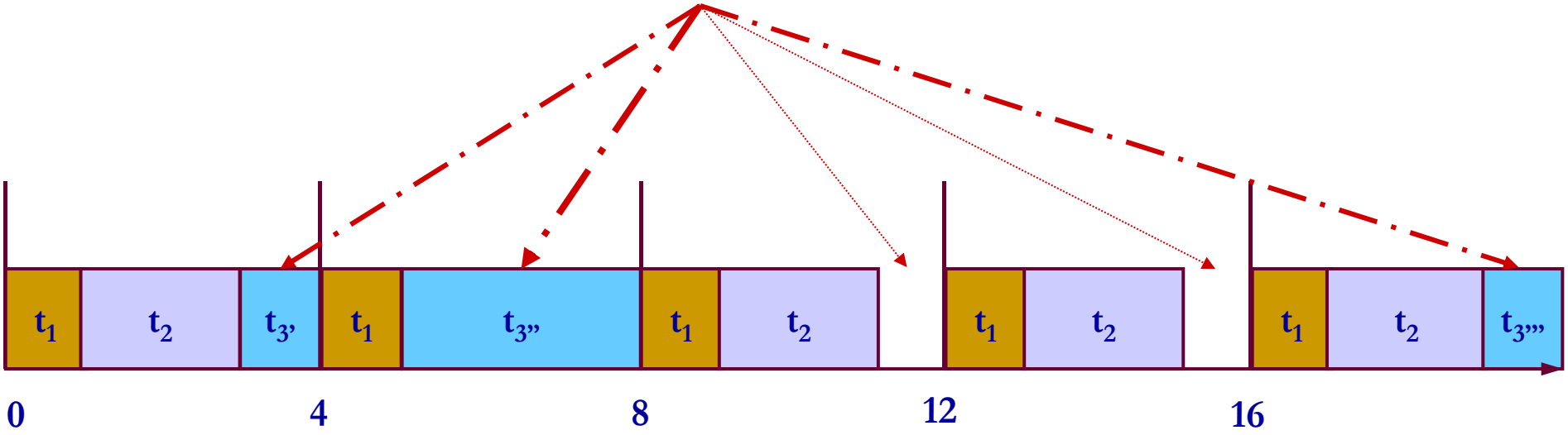


We first look at the schedule with $f = 4$ and $F = \left(\frac{H}{f}\right) = 5$

without τ_3, to see what least-disruptive opportunities we have …

Example (slicing) – 2/2

… then we observe that $e_3 = \{1, 3, 1\}$ is a good choice



$$\tau_3 = \{\tau_3' = (0, 20, 1, x), \tau_3'' = (0, 20, 3, y), \tau_3''' = (0, 20, 1, 20)\}$$

where $x < y \leq 20$ represent the precedence constraints that must hold between the slices (could have used phases instead)

Priority-driven scheduling

- Base principle
  - Every job is assigned a priority
  - The job with the highest priority is selected for execution
- ***Dynamic-priority scheduling***
  - Distinct jobs of the same task may have distinct priorities
- ***Static-priority scheduling***
  - All jobs of the same task have one and same priority

Dynamic-priority scheduling

- Two main algorithms
  - ***Earliest Deadline First*** (EDF)
  - ***Least Laxity First*** (LLF)
- **Theorem** [Liu, Layland: 1973] EDF is optimal for independent jobs with preemption
  - Also true with sporadic tasks
  - The relative deadline for periodic tasks may be arbitrary with the respect to period (<, =, >)
- Result trivially applicable to LLF
- EDF is not optimal for jobs that do not allow preemption

Static (fixed)-priority scheduling (FPS)

- Two main variants with respect to the strategy for priority assignment
  - ***Rate monotonic***
    - A task with lower period (faster rate) gets higher priority
  - ***Deadline monotonic***
    - A task with higher urgency (shorter deadline) gets higher priority
  - What about "*execution*-monotonic"?
- Before looking at those strategies in more detail we need to fix some basic notions

Critical instant /1

- Feasibility and schedulability tests must consider the ***worst case*** for all tasks
  - The worst case for task τ_i occurs when the worst possible relation holds between its release time and that of all higher-priority tasks
  - The actual case may differ depending on the admissible relation between D_i and p_i
- The notion of ***critical instant*** – if one exists – captures the worst case
  - The response time R_i for a job of task τ_i with release time on the critical instant is the longest possible value for τ_i

Critical instant /2

- **Theorem**: under FPS with $D_i \leq p_i \; \forall i$, the critical instant for task τ_i occurs when the release time of *any* of its jobs is in phase with a job of every higher-priority task in the task set
- We seek $\max(\omega_{i,j})$ for all jobs $\{j\}$ of task τ_i for

$$\omega_{i,j} = e_i + \sum_{(k=1,..,i-1)} \left\lceil \frac{(\omega_{i,j} + \varphi_i - \varphi_k)}{p_k} \right\rceil e_k - \varphi_i$$

For task indices assigned in decreasing order of priority

  - The summation term captures the *interference* that any job j of task τ_i incurs from jobs of all higher-priority tasks $\{\tau_k\}$ between the release time of the first job of task τ_k (with phase φ_k) to the response time of job j of task τ_i , which occurs at $\varphi_i + \omega_{i,j}$

Time-demand analysis /1

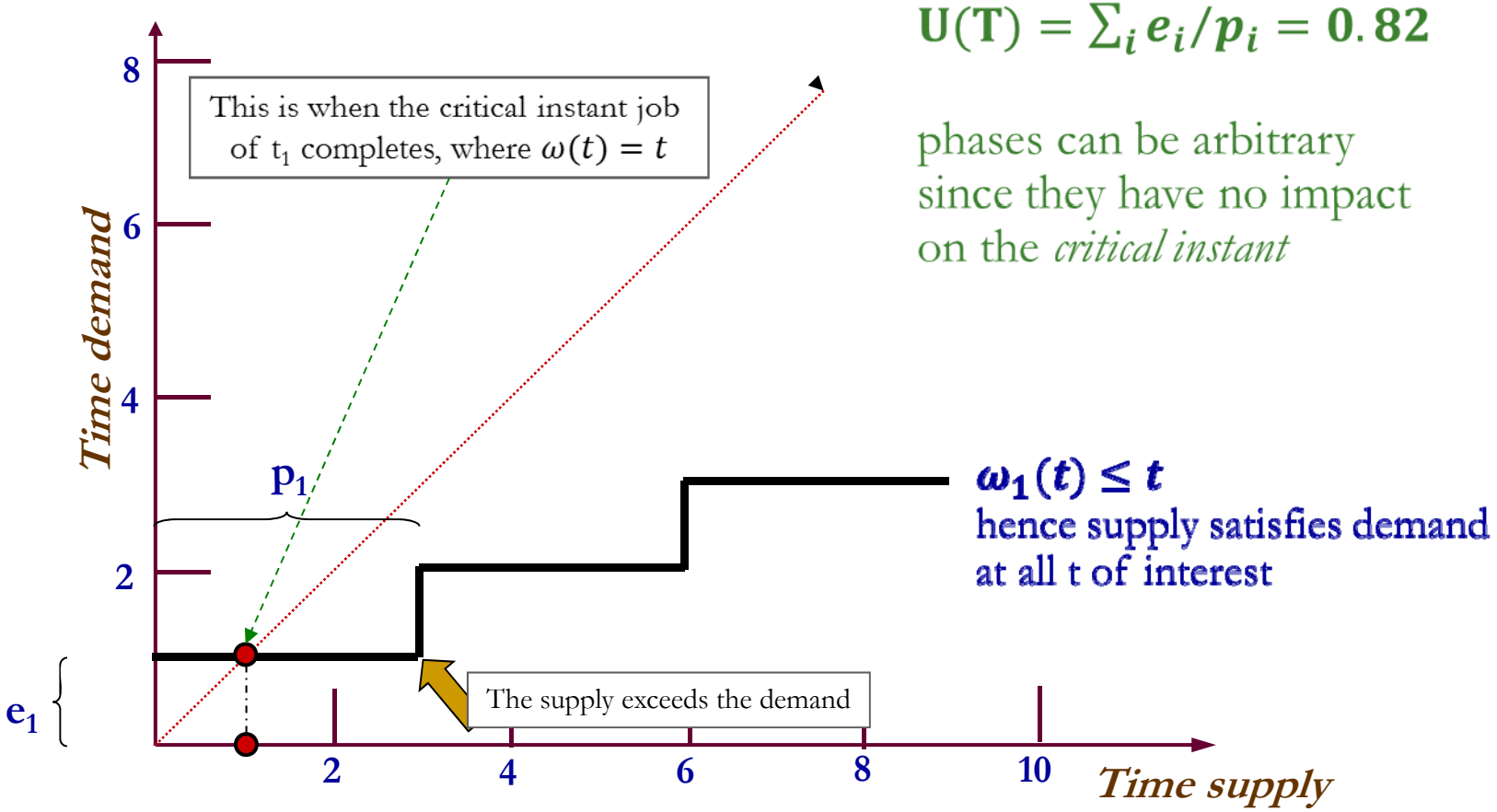
- When φ is 0 for all jobs considered then this equation captures the absolute worst case for task τ_i
- This equation stands at the basis of ***Time Demand Analysis*** which investigates how ω varies as a function of time
  - So long as $\omega(t) \leq t$ *for some t* within the time interval of interest the supply satisfies the demand, hence the job can complete in time
- **Theorem** [Lehoczky, Sha, Ding: 1989] condition $\omega(t) \leq t$ is an *exact feasibility test* (necessary and sufficient)
  - The obvious question is for which 't' to check
  - The method proposes to check at all periods of all higher-priority tasks until the deadline of the task under study

Time demand analysis /2

$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}$ $(\varphi_i, p_i, e_i, D_i)$

$U(T) = \sum_i e_i/p_i = 0.82$

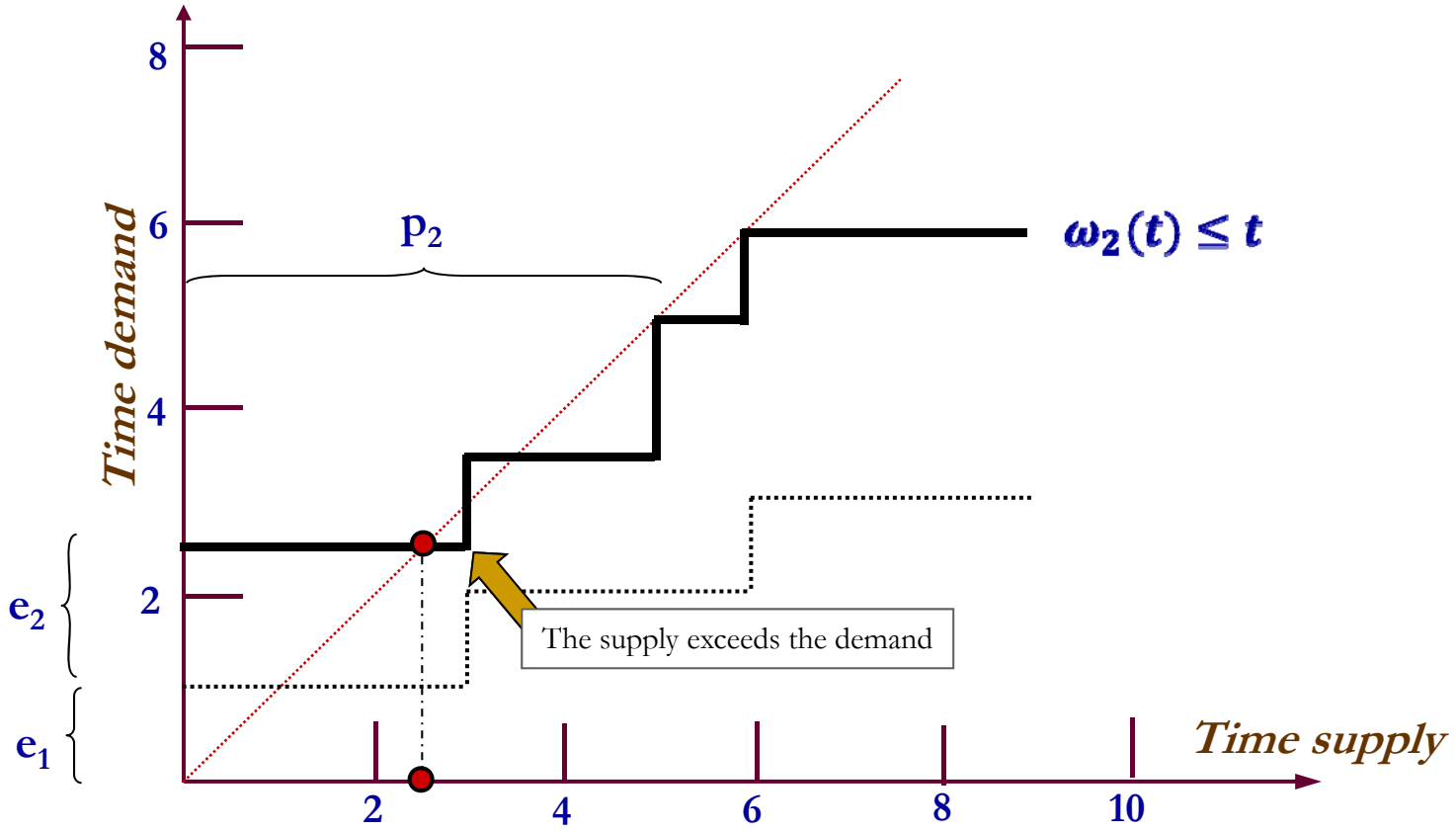
phases can be arbitrary since they have no impact on the *critical instant*



$\omega_1(t) \leq t$
hence supply satisfies demand at all t of interest

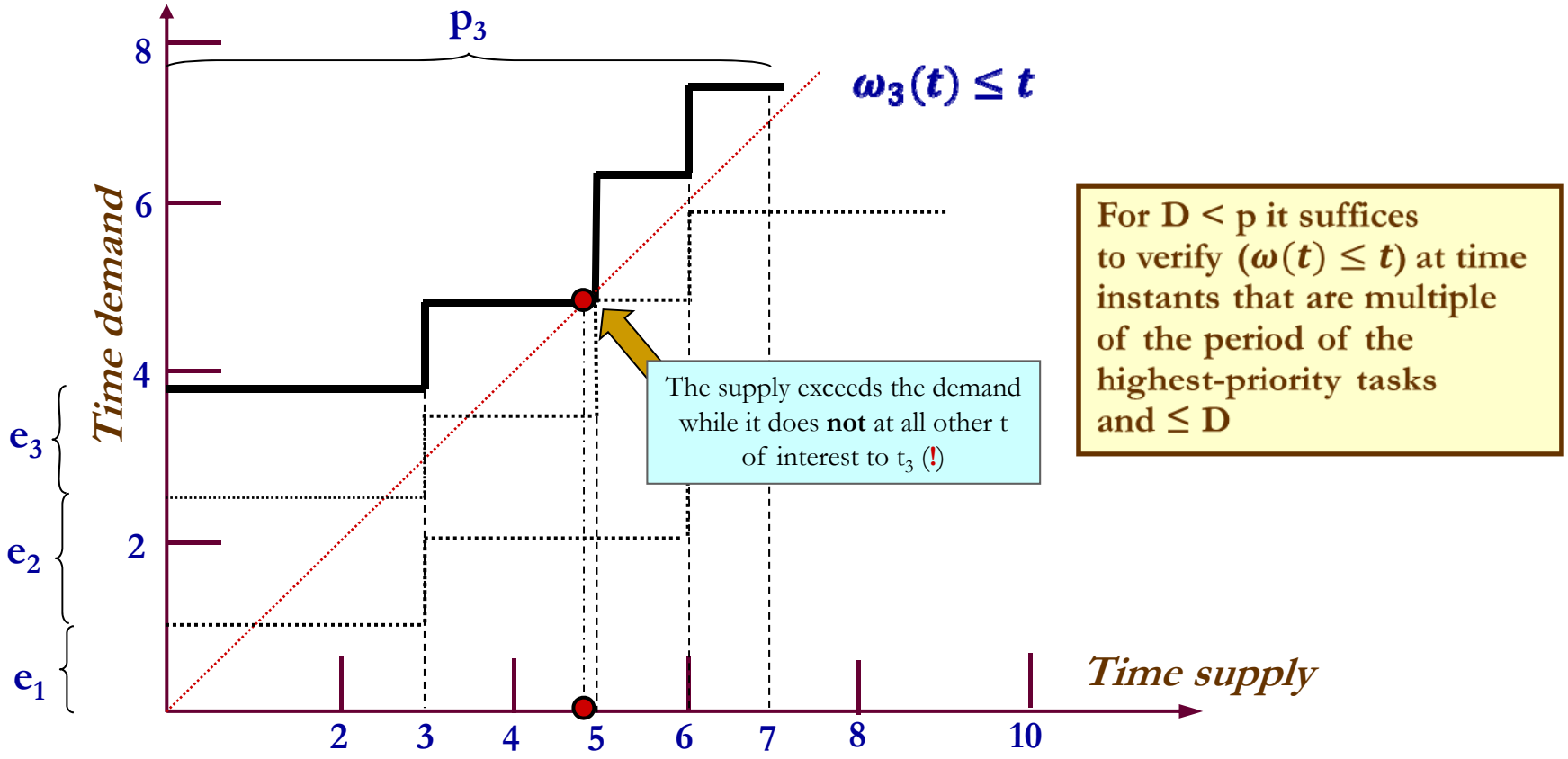
Time demand analysis /3

$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}$



Time demand analysis /4

$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}$



Time demand analysis /5

- It is straightforward to extend TDA to determine the *response time* of tasks
- The smallest value t that satisfies the fixed-point equation $t = e_i + \sum_{(k=1,..i-1)} \left\lceil \frac{t}{p_k} \right\rceil e_k$ is the *worst-case response time* of task τ_i
- Solutions methods to calculate this value were independently proposed by
  - [Joseph, Pandia: 1986]
  - [Audsley, Burns, Richardson, Tindell, Wellings: 1993]

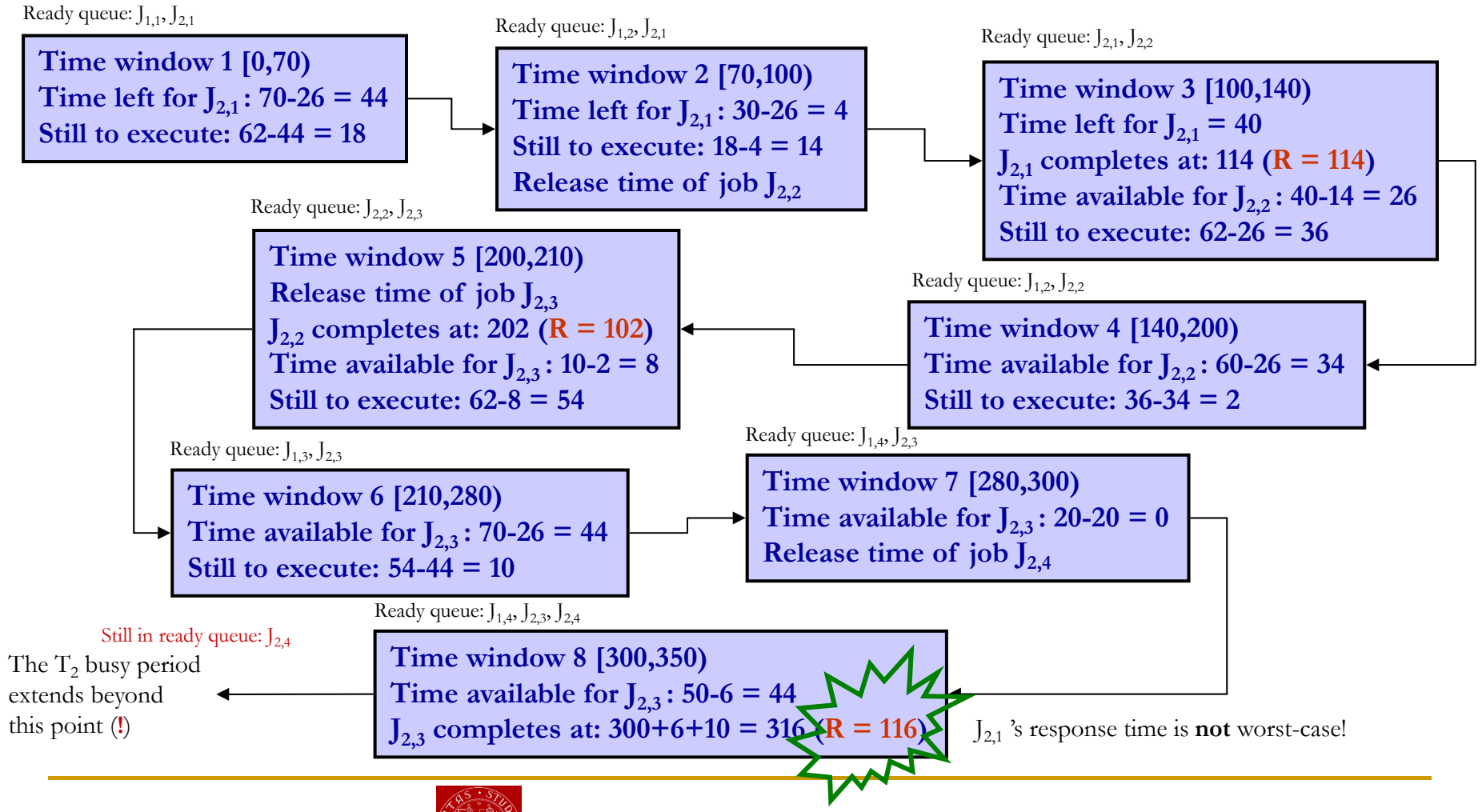
Time demand analysis /6

- What changes in the definition of critical instant when $D>p$?
- **Theorem** [Lehoczky, Sha, Strosnider, Tokuda: 1991] The first job of task τ_i may *not* be the one that incurs the worst-case response time
- Hence we must consider *all* jobs of task τ_i within the so-called ***level-i busy period***
  - The (t_0, t) time interval within which the processor is busy executing jobs with priority $\geq i$, release time in (t_0, t), response time falling within t
  - The release time in (t_0, t) captures the full backlog of interfering jobs
  - The response time of all those jobs falling within t ensures that the busy period includes their completion

Example

$T_1 = \{-, 70, 26, 70\}$, $T_2 = \{-, 100, 62, 120\}$ $(\varphi_i, p_i, e_i, D_i)$

Let's look at the level-2 busy period

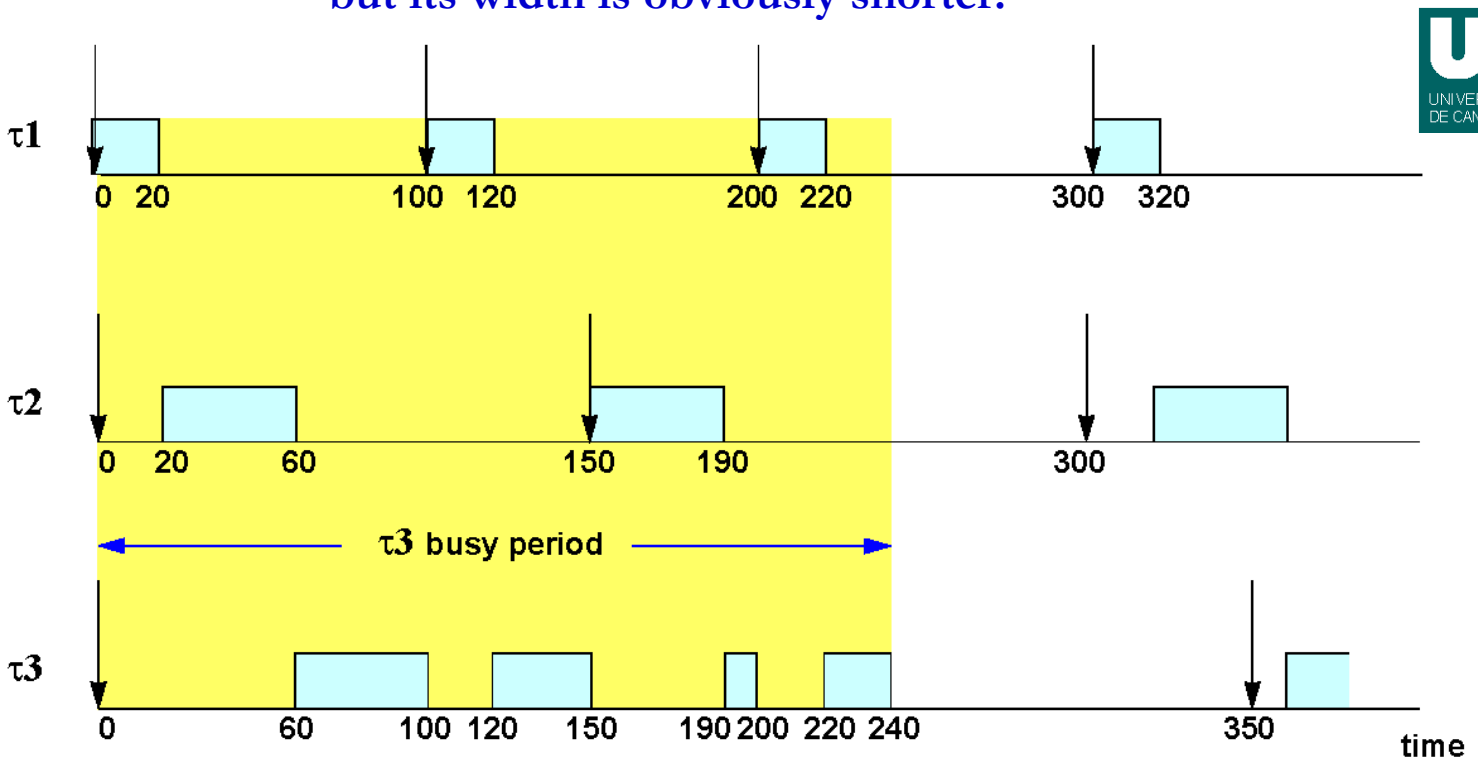


Level-i busy period

$T_1 = \{-, 100, 20, 100\}$, $T_2 = \{-, 150, 40, 150\}$, $T_3 = \{-, 350, 100, 350\} \Rightarrow U = 0.75$

The same definition of level-i busy period holds also for $D \leq p$

but its width is obviously shorter!



3 Fixed-Priority Scheduling

Credits to A. Burns and A. Wellings

(Locally) standard notation

B:	Worst-case blocking time for the task (if applicable)
C:	Worst-case computation time (WCET) of the task
D:	Deadline of the task
I:	The interference time of the task
J:	Release jitter of the task
N:	Number of tasks in the system
P:	Priority assigned to the task (if applicable)
R:	Worst-case response time of the task
T:	Minimum time between task releases (or task period)
U:	The utilization of each task (equal to C/T)
a-Z:	The name of a task

Fixed-priority scheduling (FPS)

- At present the most widely used approach
  - The distinct focus of this segment
- Each task has a fixed (static) priority computed off-line
- The ready tasks are dispatched to execution in the order determined by their priority
- In real-time systems the "priority" of a task is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity

Preemption and non-preemption /1

- With priority-based scheduling, a high-priority task may be released during the execution of a lower priority one
- In a *preemptive* scheme, there will be an immediate switch to the higher-priority task
- With *non-preemption*, the lower-priority task will be allowed to complete before the other may execute
- Preemptive schemes enable higher-priority tasks to be more reactive, hence they are preferred

Response time analysis /1

- The worst-case response time R_i of task τ_i is first calculated and then checked (trivially) with its deadline

$$R_i \leq D_i$$

$$R_i = C_i + I_i$$

  - Where I is the interference from higher-priority tasks

Calculating R

- Within R_i, each higher priority task τ_j will execute a $\left\lceil \frac{R_i}{T_j} \right\rceil$ times
  - The ceiling function $\lceil f \rceil$ gives the smallest integer greater than the fractional number f on which it acts
    - E.g., the ceiling of $1/3$ is 1, of $6/5$ is 2, and of $6/3$ is 2
- The total interference suffered by τ_i from τ_j in R_i is given by $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Response time equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Where $hp(i)$ is the set of tasks with priority higher than task τ_i
- Solved by forming a recurrence relationship

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- The set of values $w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots$ is monotonically non-decreasing
- When $w_i^n = w_i^{n+1}$ the solution to the equation has been found
- w_i^0 must not be greater than C_i (e.g. 0 or C_i)

Response time analysis /2

- RTA is a *feasibility test*
  - Thus exact, hence necessary and sufficient
- If the task set passes the test then all its tasks will meet all their deadlines
- If it fails the test then, at run time, some tasks will miss their deadline and FPS tells us exactly which
  - Unless the computation time estimations (the WCET) themselves turn out to be pessimistic

4. System issues

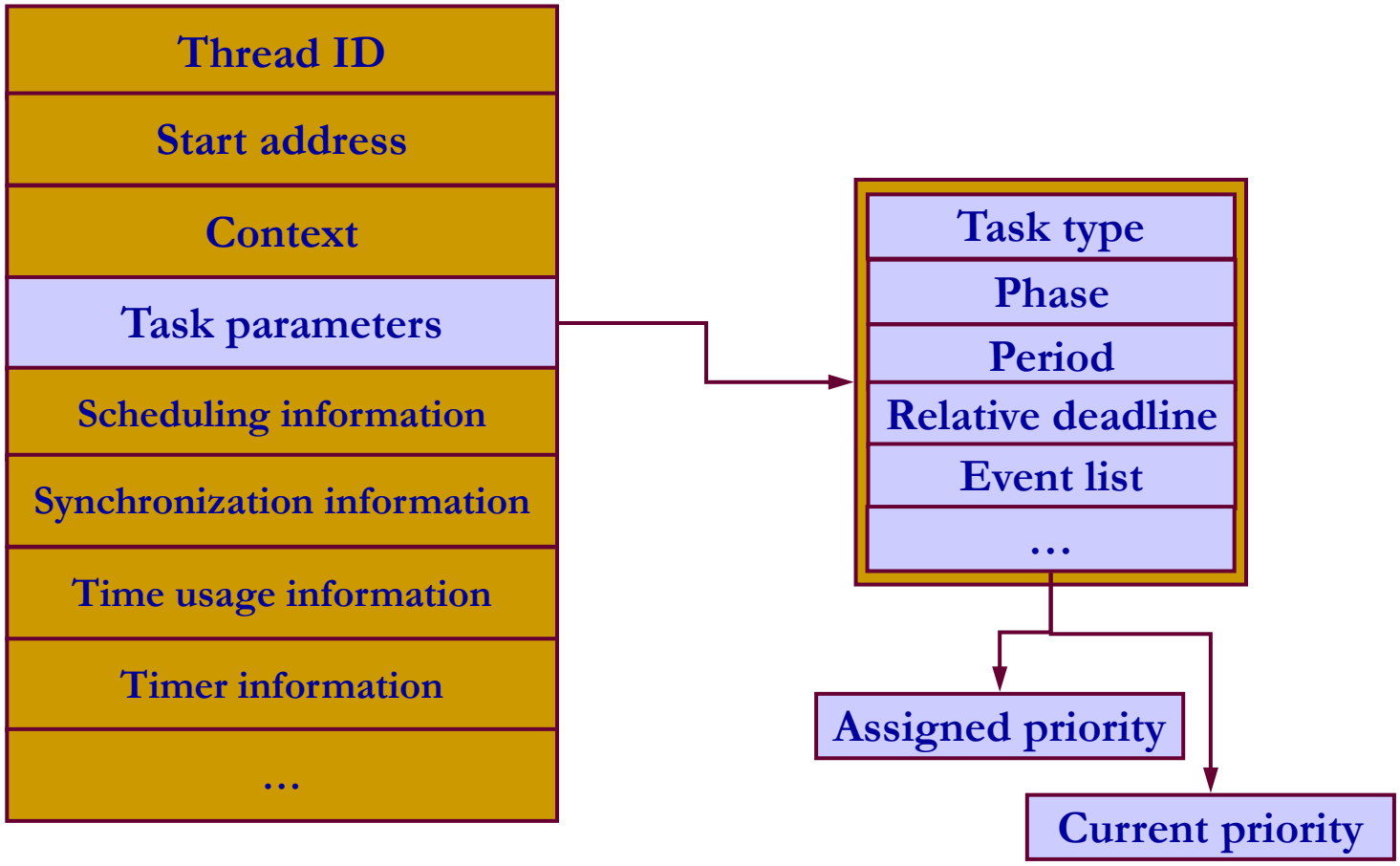
Context switch

- Preemption causes time and space overheads which should be duly accounted for in realistic 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

- Tasks must be known to the RTOS
  - Tasks are the unit of CPU allocation by the scheduler
    - Tasks issue jobs, one at a time, which are subject to scheduling and dispatching
    - The scheduler decides which task gets the CPU
      - Typically by the position assigned to tasks in the ready queue
    - The dispatcher gets tasks to run and operates the context switch
- On task creation, some RAM is assigned to the ***Task Control Block*** for that task
  - The insertion of a task in a state queue (e.g., ready) is made by placing a pointer to the relevant TCB
  - The disposal of a task at end of life requires removal of its TCB and de-allocation of any memory it had in use
    - In typical embedded systems, tasks *never* terminate

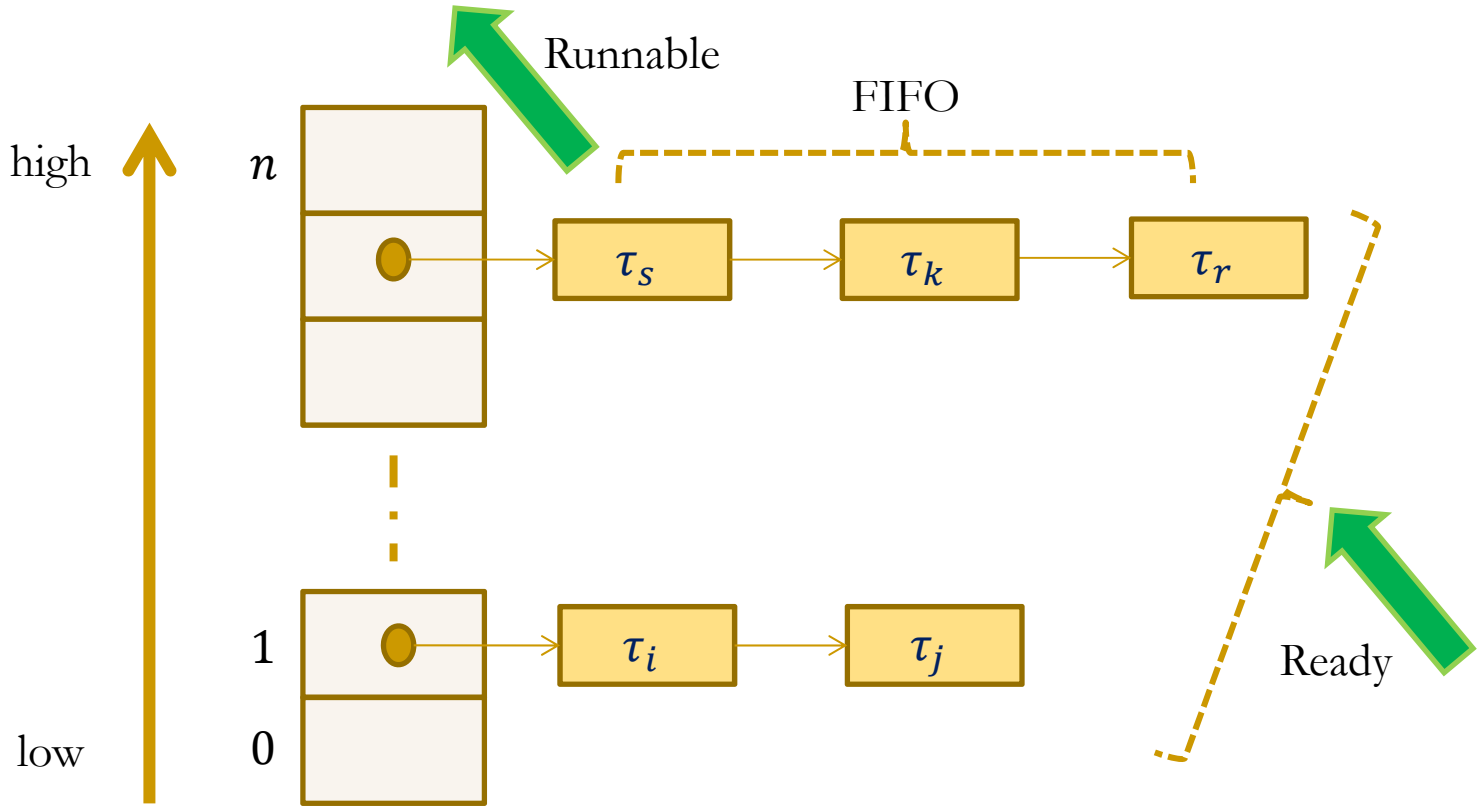
Task control block



Priority levels /1

- The scheduling techniques that we have studied assume jobs to have *distinct* priorities
  - It is not obvious however that concrete systems can always meet this requirement
  - Consequently jobs may have to share priority levels
  - 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 residing 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
- The time demand equation for J_i to study in the interval $0 < t \leq \min(D_i, p_i)$ then becomes
  - $\omega_{i_1}(t) = e_i + B_i + \sum_{S(i)} e_i + \sum_{k=1,..,i-1} \left\lceil \frac{\omega_{i_1}(t)}{p_k} \right\rceil e_k$
- This obviously worsens J_i's response time
  - But the impact in terms of ***schedulability loss*** at system level may not be as bad (see later …)

The scheduler /1

- This is a distinct part of the RTOS that does not execute in response to explicit application invocations
- It acts every time a task changes state (hence the ready queue does)
  - 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 (a.k.a. ***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 too

Tick scheduling /1

- So far we have tacitly assumed that the scheduler operates on an *event-driven* basis
  - The scheduler always immediately executes upon the occurrence of a *scheduling event*
  - If it was so then we could reasonably assume that a job is placed in the ready queue at its release time
- Schedulers may also operate in a *time-driven* manner
  - In that case 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 1 (scheduling) tick later than it arrived
  - This delay has negative impact on the job's response time
  - We also need to assume that a logical place exists 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

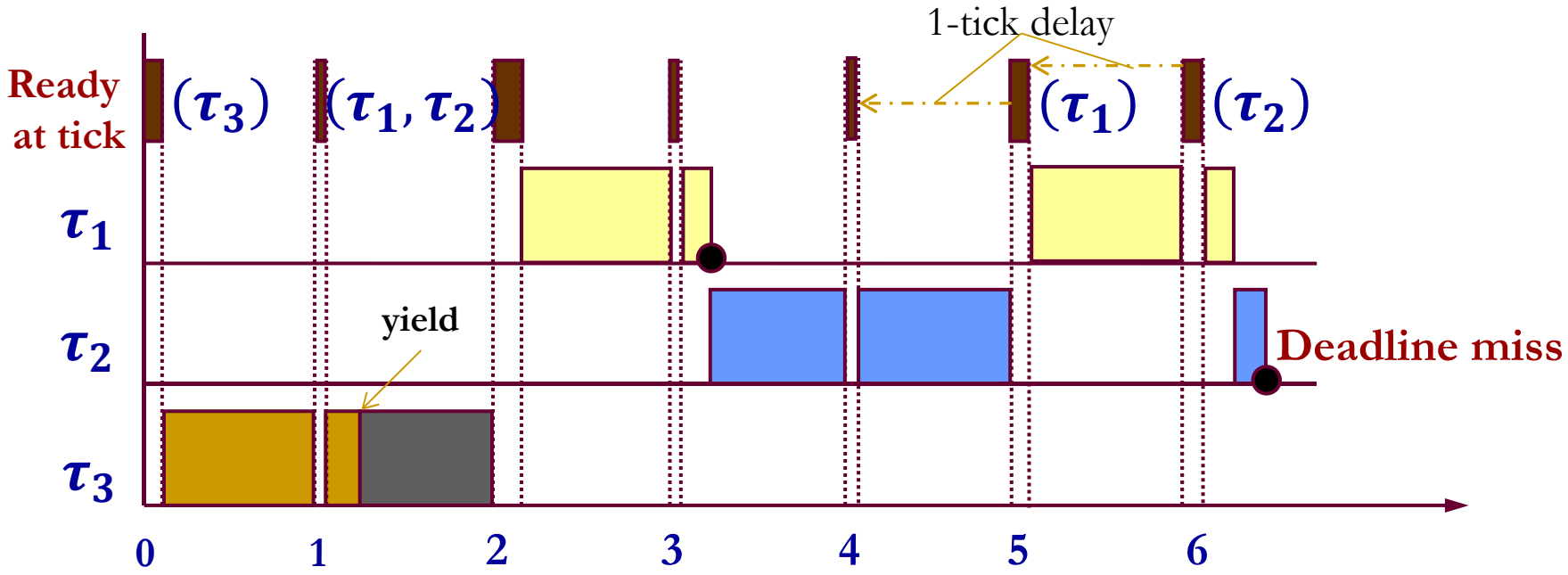
$(\varphi_i, p_i, e_i, D_i)$

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

τ_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 θ_k is the maximum time of non-preemptable execution by any job J_k

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 reasons of efficiency the interrupt handling service is typically 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

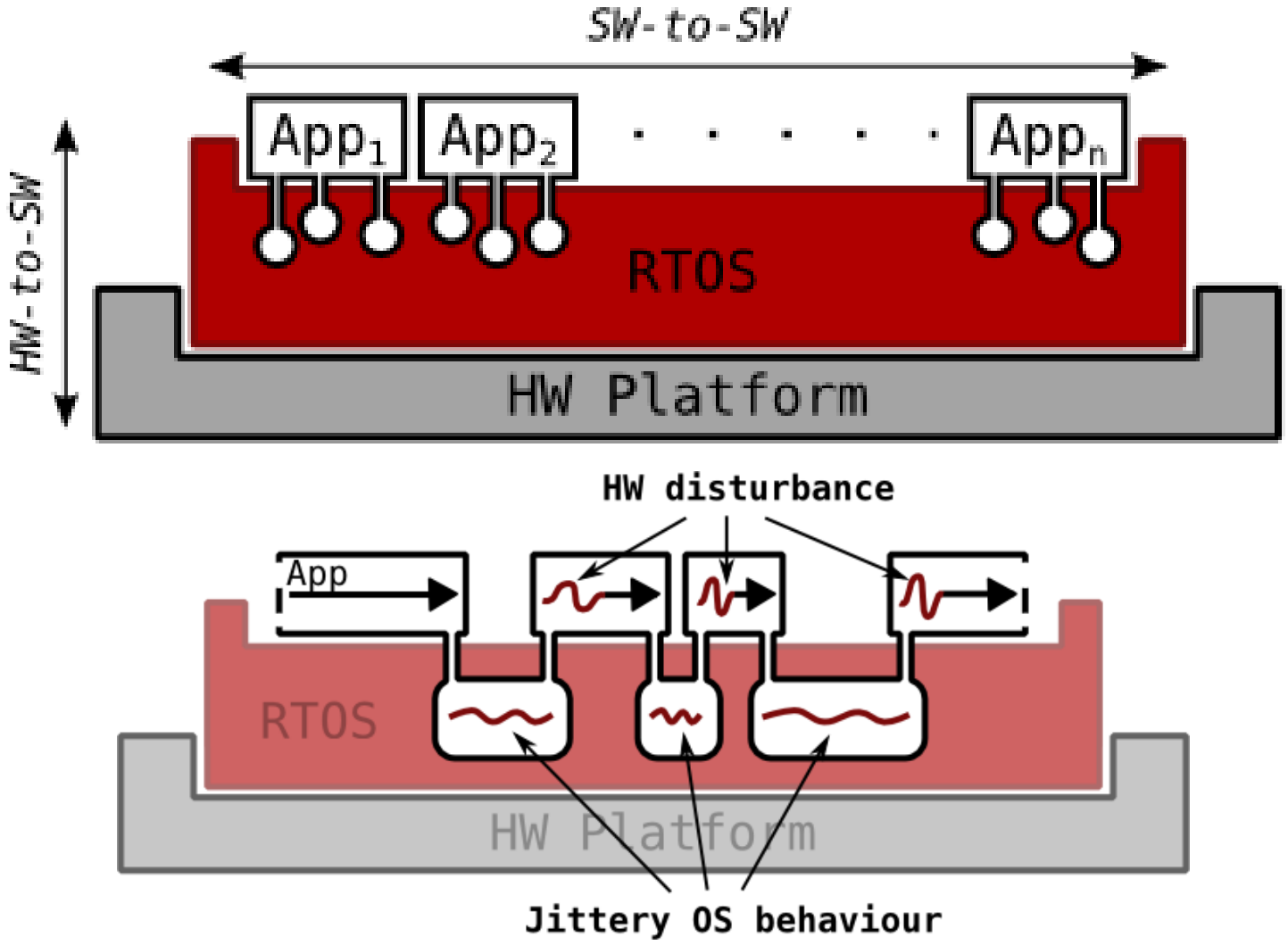
Interrupt handling /5

- 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

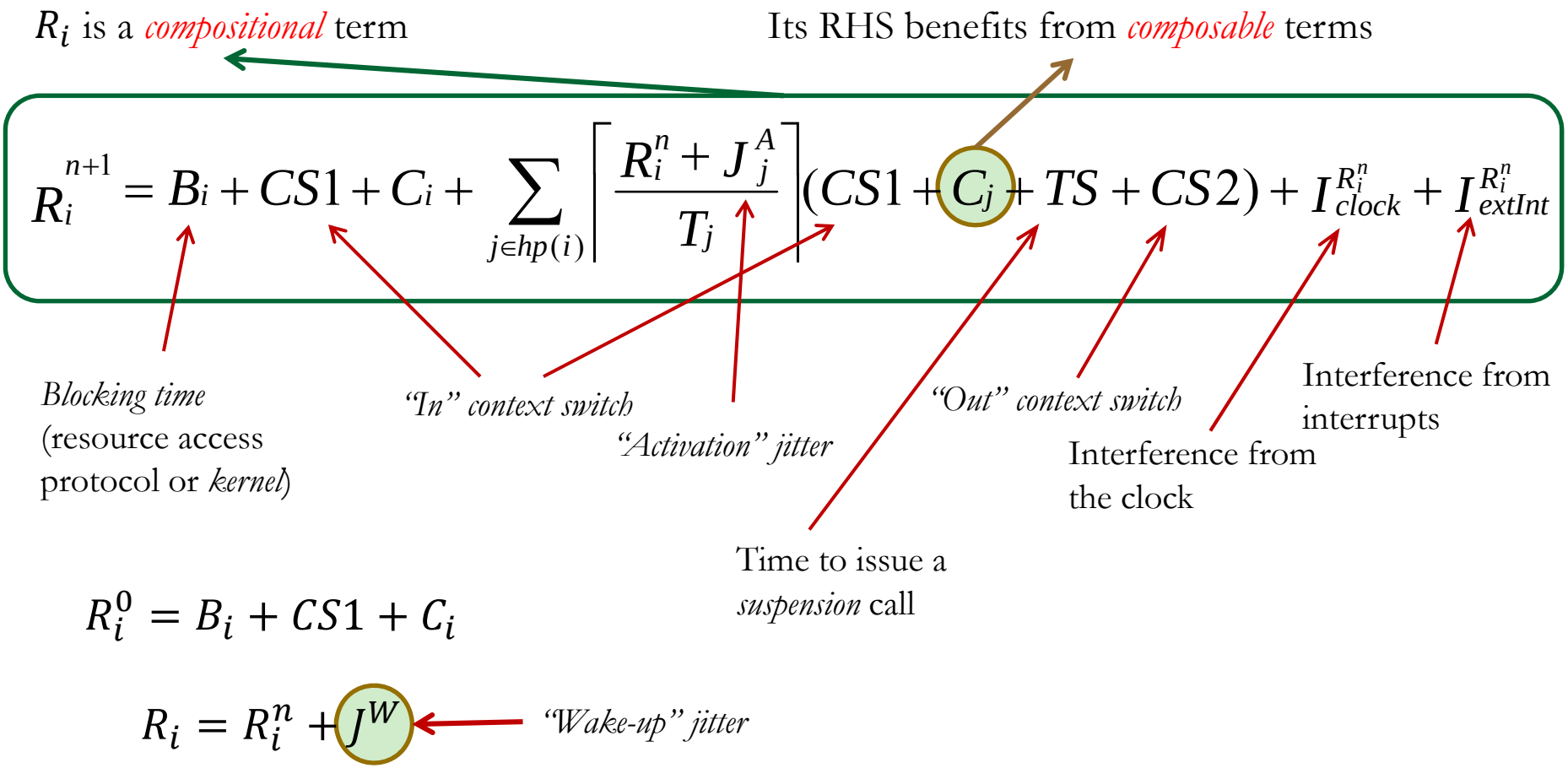
- To reduce *distributed overhead*, the deferred part of the interrupt handling service must be preemptable
  - Hence it must execute at software priority
- But it still may directly or indirectly operate on RTOS data structures
  - Which must be protected by appropriate access control protocols
  -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

Understanding the software /1



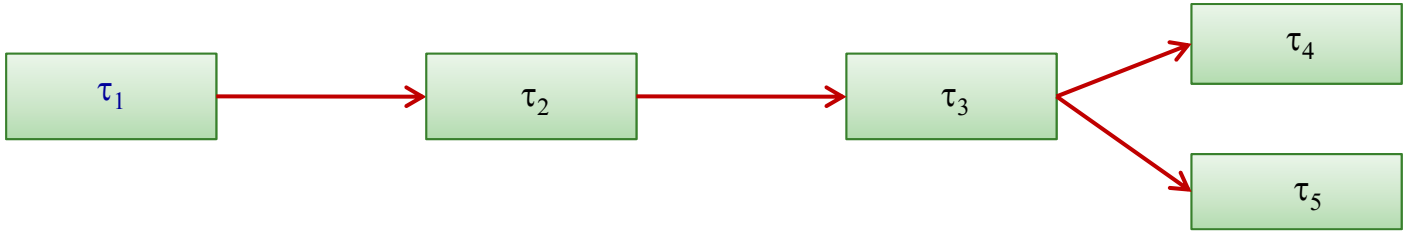
Friends or foes to time composability

Fine-grained response time analysis



Transactions /1

- Causal relations between activities
  - They consider information relevant to analysis that is not captured by classic workload models
    - Dependences in the activation of jobs

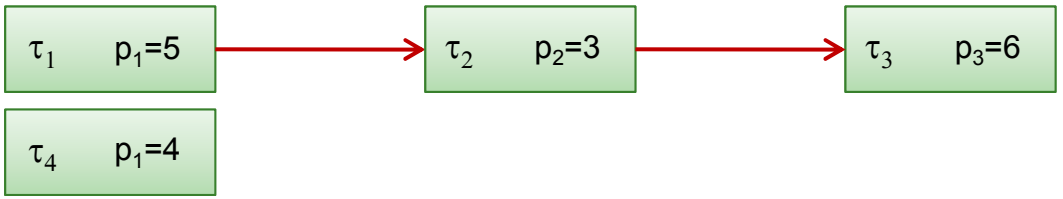


  - Originally introduced for the analysis of distributed systems
    - Also useful for the analysis of "collaboration patterns" employed for single-CPU systems

Transactions /2

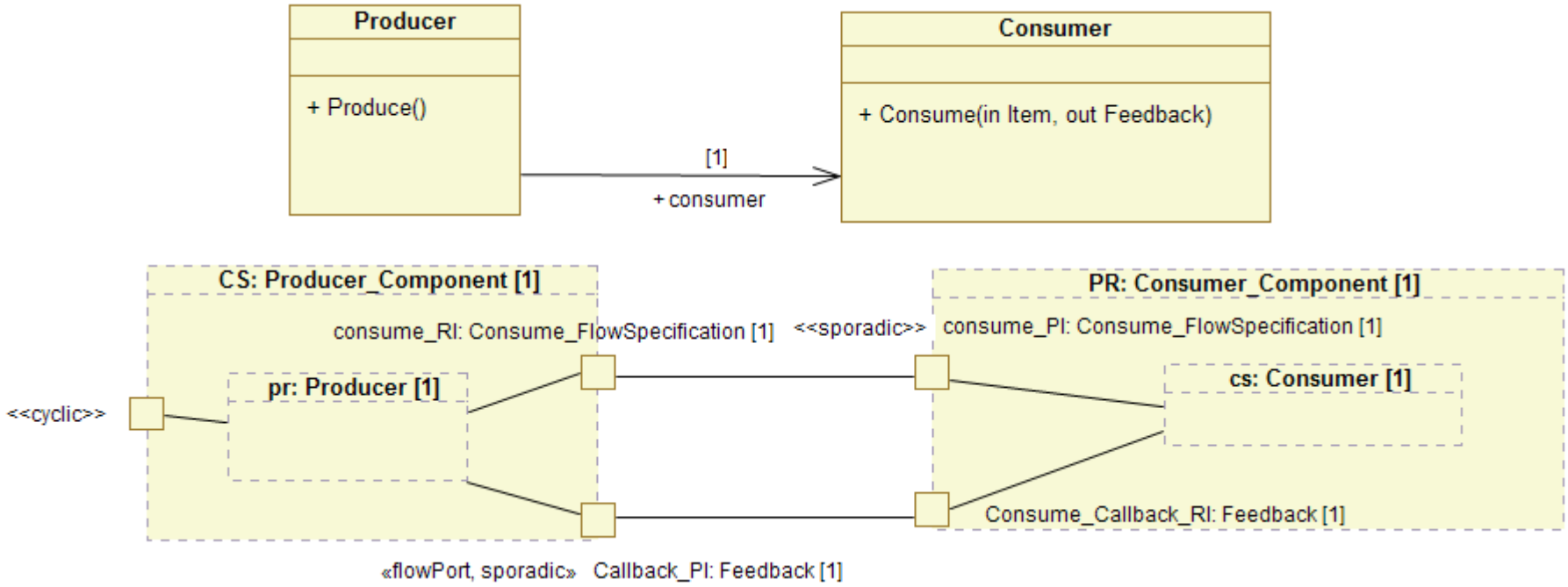
- Two main kinds of dependence
  - *Direct precedence* relation (e.g., producer-consumer)
    - τ_2 cannot proceed until τ_1 completes

  - *Indirect priority* relation
    - τ_4 does not suffer interference from τ_3 (under FPS and synchronous release of τ_1 and τ_4 for priorities increasing with values)

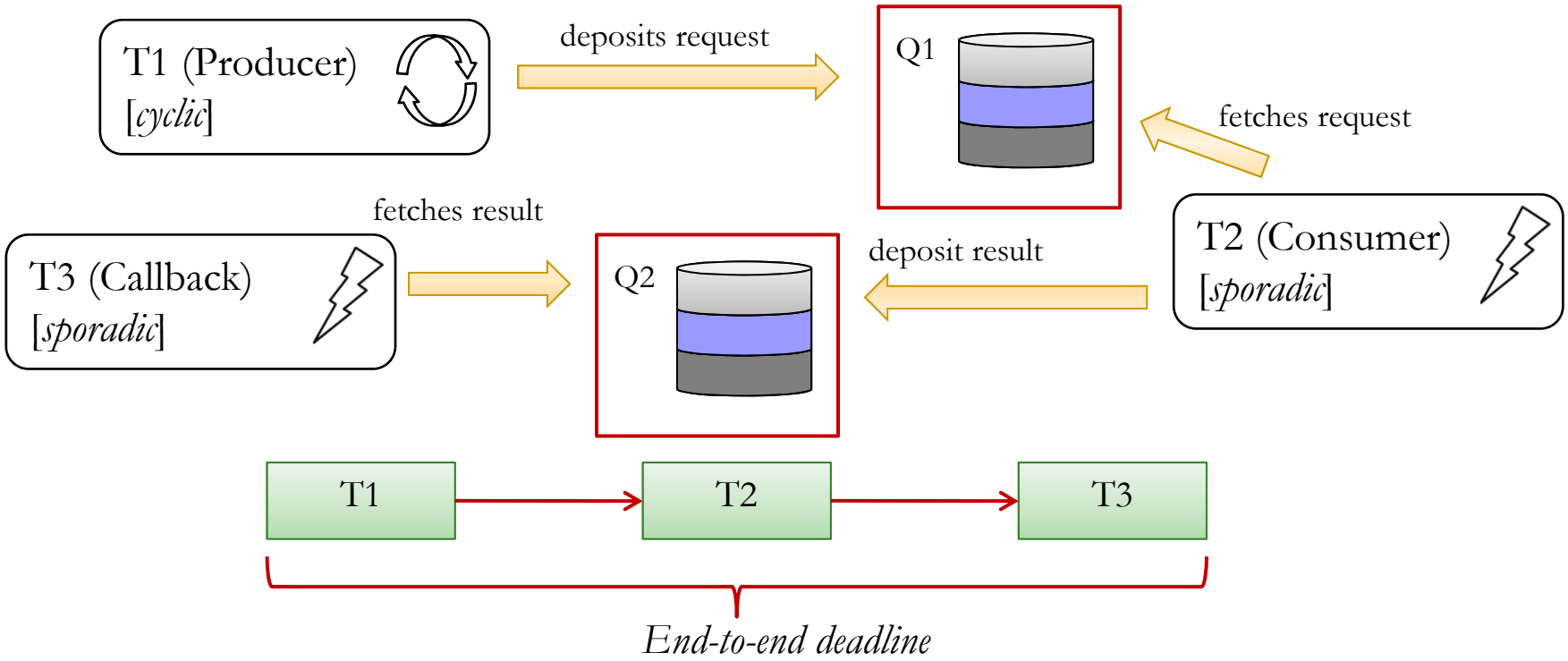


Example /1

- A "*callback pattern*" to permit **in out** interactions between tasks in Ravenscar systems



Example /2



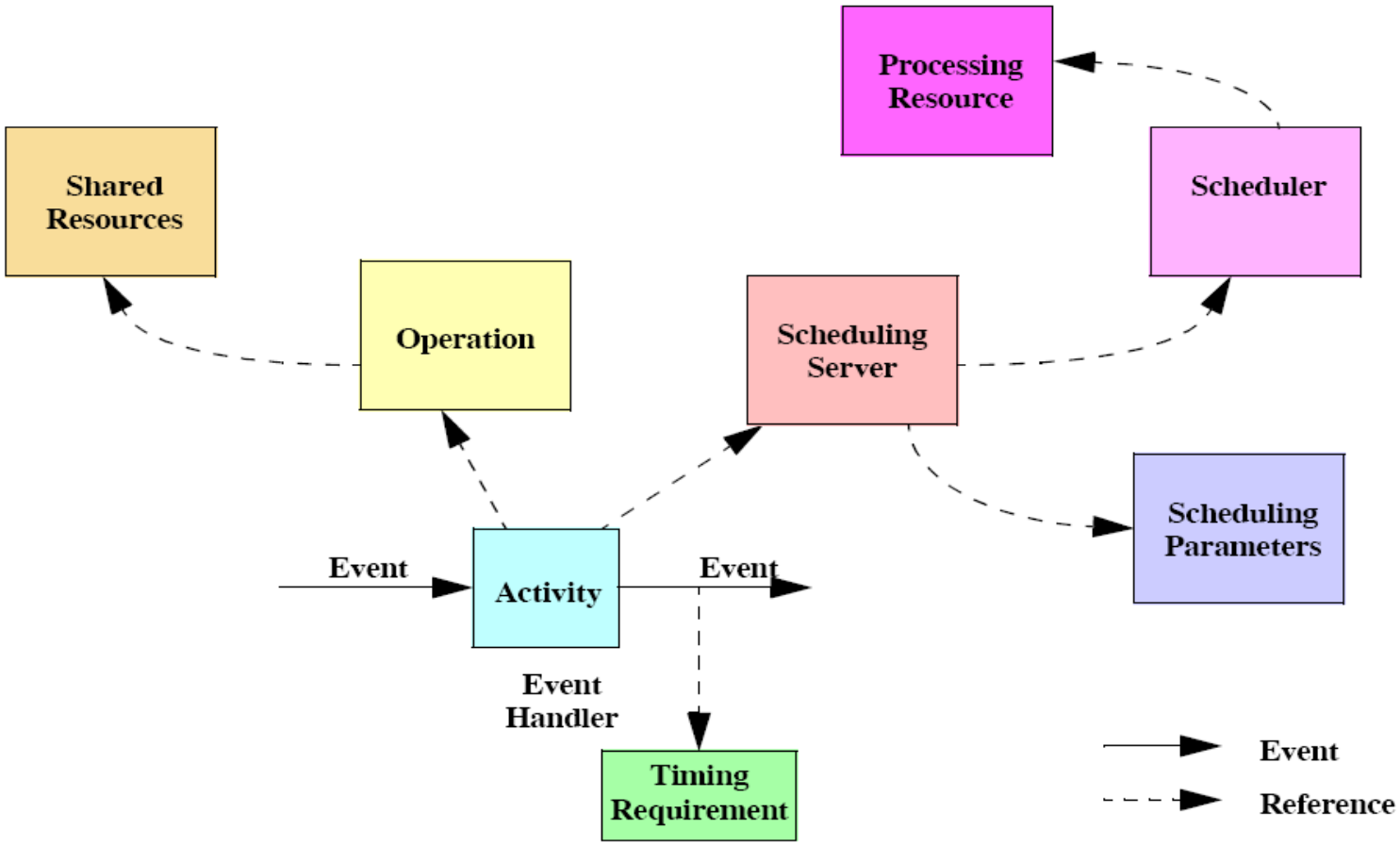
The feasibility of the *end-to-end response time* against this deadline is what matters (!)

5. Practical

MAST

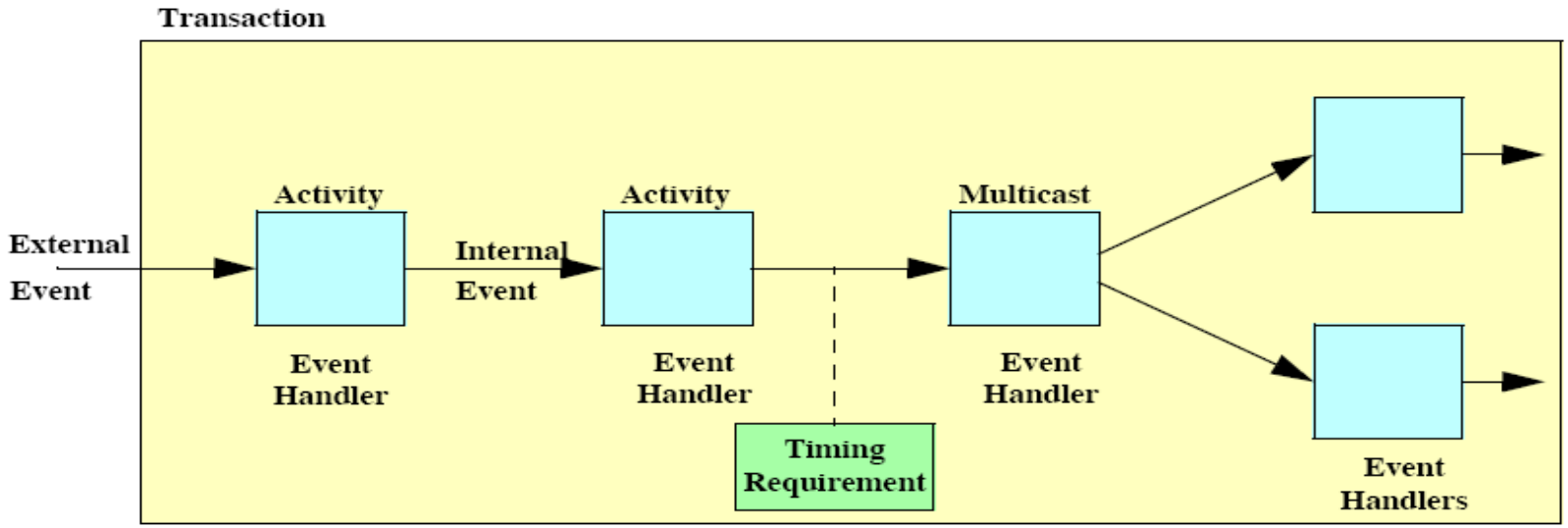
- Modeling and Analysis Suite for Real-Time Systems (MAST, http://mast.unican.es)
  - Developed at University of Cantabria, Spain
  - Open source
  - Implements several analysis techniques
    - For uniprocessor and distributed (no-shared memory) processor systems
    - Under FPS or EDF

MAST: real-time model

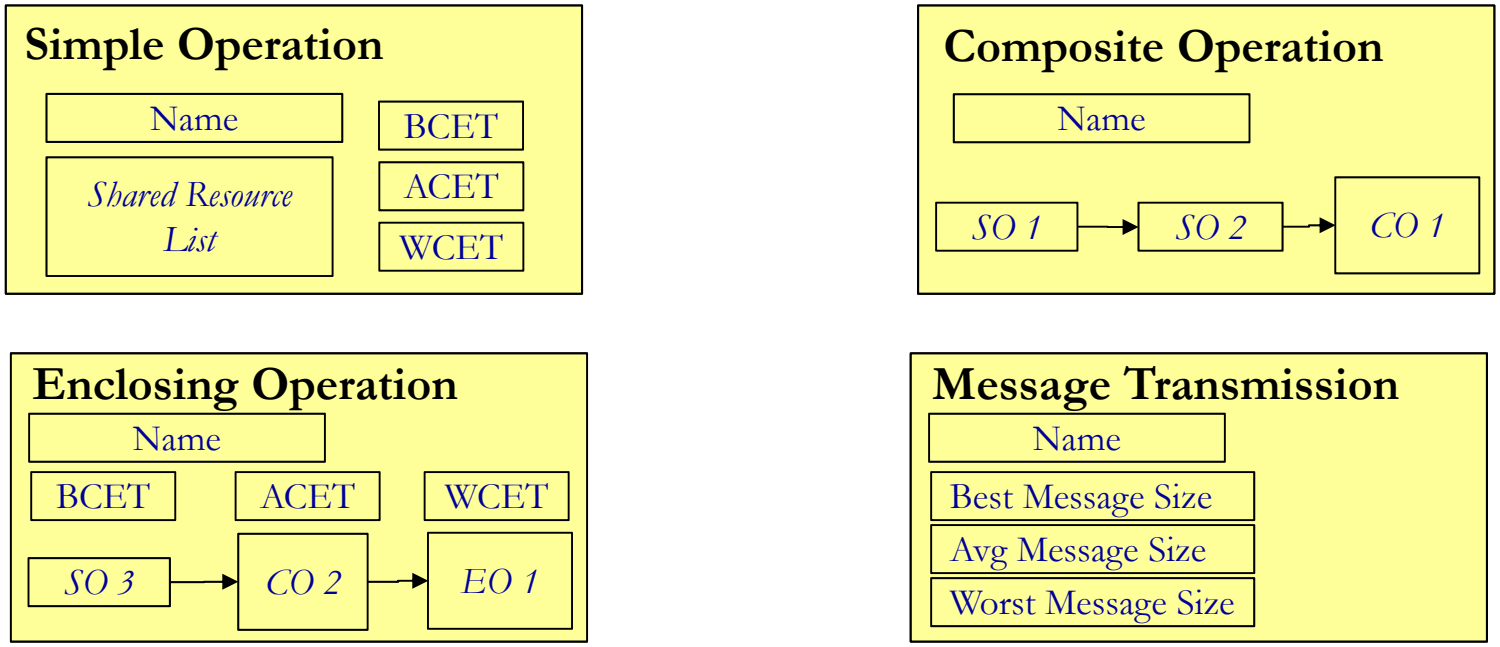


MAST: transaction

- To model causal relations between activities
  - Triggered by external events
    - Periodic, sporadic, aperiodic, etc…

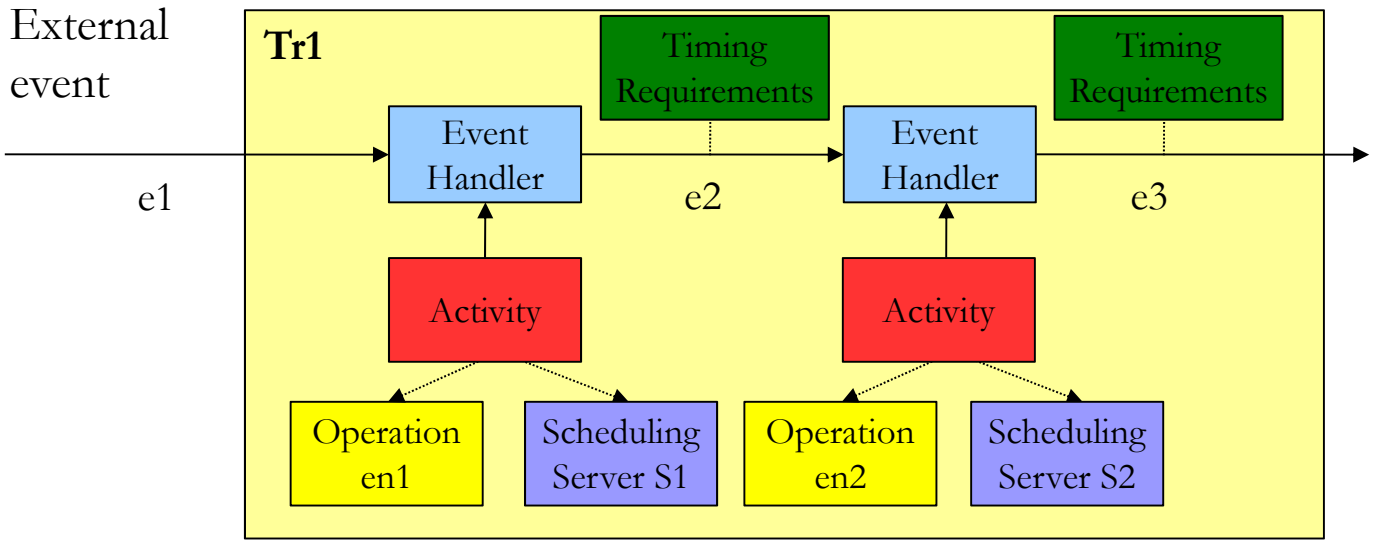


MAST: operations

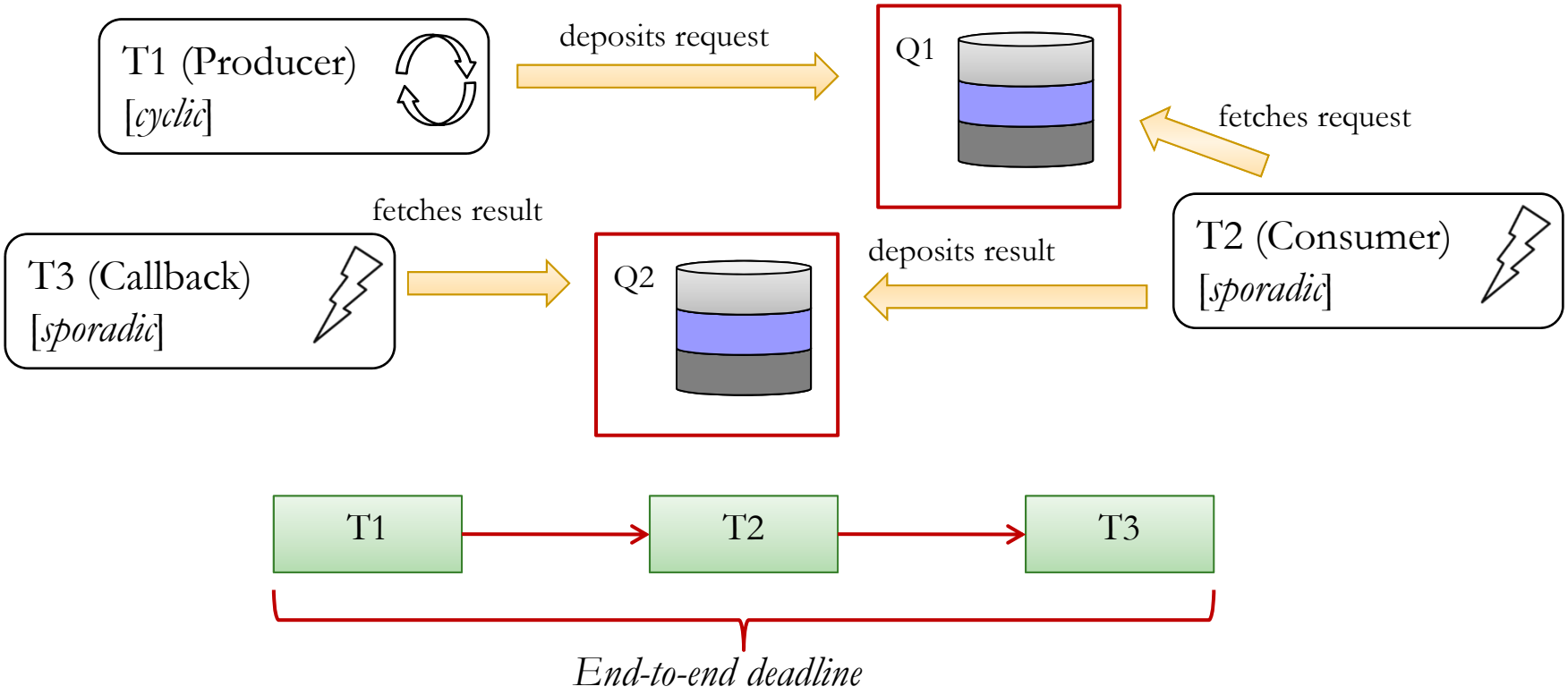


- The real-time model includes the description of all the operations in the system

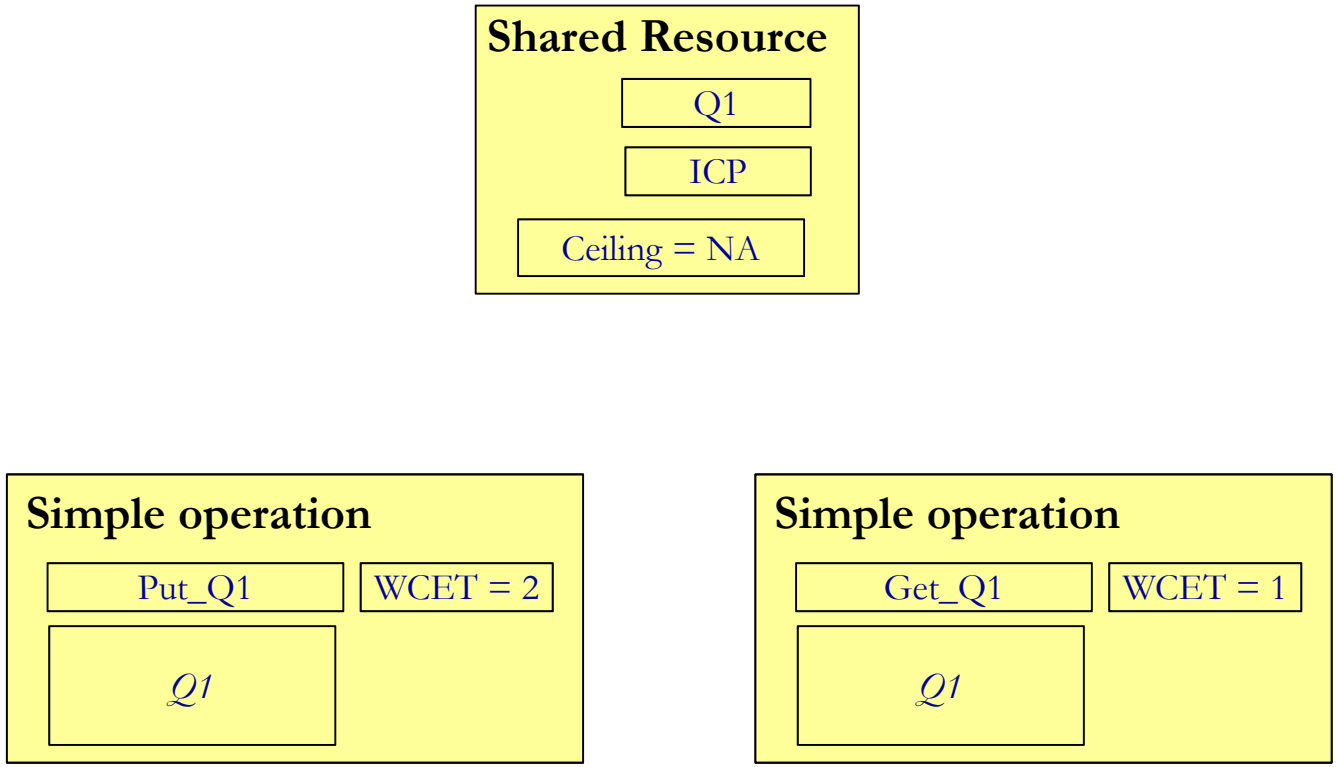
MAST: creation of a transaction



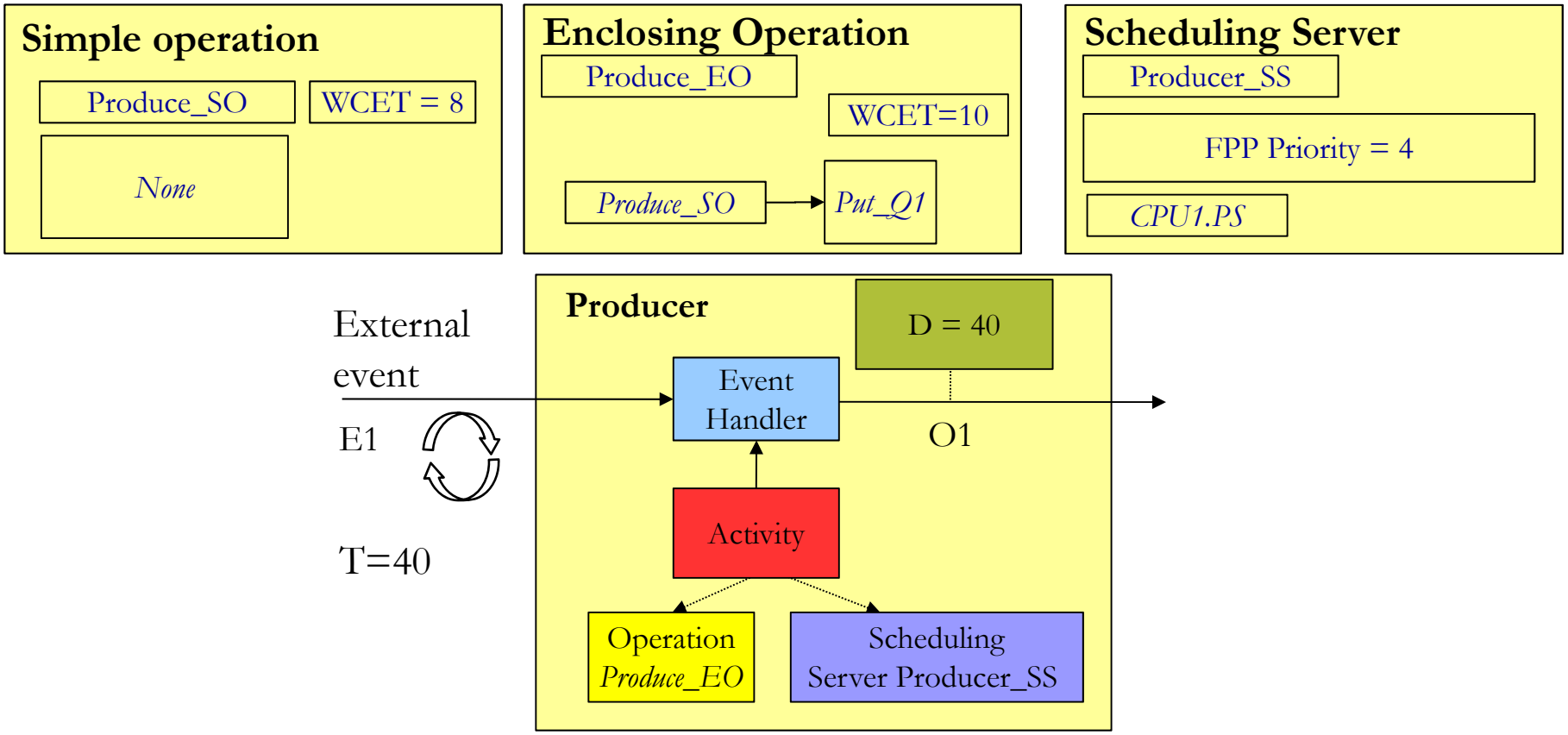
Example: Ravenscar callback



Example: shared resources in MAST



Example: modeling tasks in MAST



Example: introducing transactions