Real-Time Kernels & Systems

Academic Year 2022/2023
Master Degree in Computer Science
Department of Mathematics
University of Padua
Tullio Vardanega, tullio.vardanega@unipd.it

Lecture topics

Bibliography
- J. Liu, "Real-Time Systems", Prentice Hall, 2000
- A. Burns and A. Wellings, "Analysable Real-Time Systems" Amazon Books, 2016
- State-of-the-art literature

1. Introduction
2. Scheduling basics
3. Fixed-priority scheduling
   a. Basic workload models
   b. Task interactions and blocking
   c. Exercises and further model extensions
4. Implementation issues
   a. Programming real-time systems (in Ada)
   b. A look under the hood
5. Distributed systems
   a. Worst case with offsets
   b. End-to-end analysis
6. Execution-time analysis
7. Multicore systems
   a. Initial reckoning
   b. Seeking the lost optimality
   c. Global resource sharing
8. Mixed-criticality systems
9. Predictable parallel programming

Protocol

- **Learning outcomes**
  - Realize the existence and the needs of software systems whose response time is critical to their use and consequently to their design
  - Understand the principles, methods, techniques and technology required to develop them so as to guarantee predictability
- **Instructional methods**
  - Web resources in complement to slide desks and presentations
  - Reciprocal feedback
    - Instructor to student, about incremental (self-)assignments
    - Student to instructor, about the progression of learning
  - Interaction
    - During class
    - Via posts in the Moodle's class billboard, for all that must be public
    - Email otherwise

1. Introduction

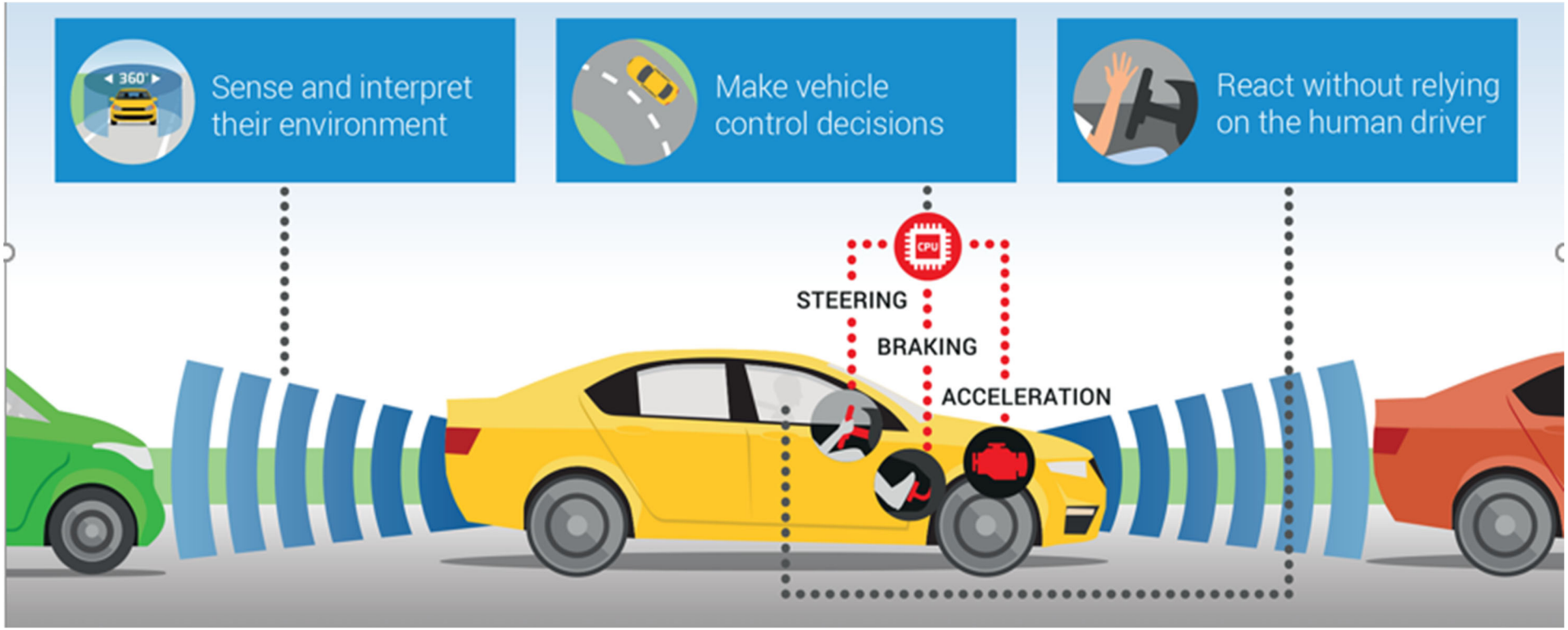
Where we make some initial acquaintance with what real-time systems are and why they came about, and then take a first look at their abstract concept

Initial intuition – 1/2

- **Real-time system**
  - An aggregate of computers, I/O devices and *application-specific software*, characterized by
    - Continuous interaction with the external environment
    - To control it after mission-specific goals
    - Capturing variations of the state of the system in relation to its physical environment, and reacting to them in a timely fashion
  - Comprised of system activities subject to timing constraints
    - Reactivity, accuracy, duration, completion, responsiveness
    - All dimensions of ***timeliness***
  - System activities inherently *concurrent* and increasingly *parallel*
  - The satisfaction of all system constraints must be proved

Illustrative example

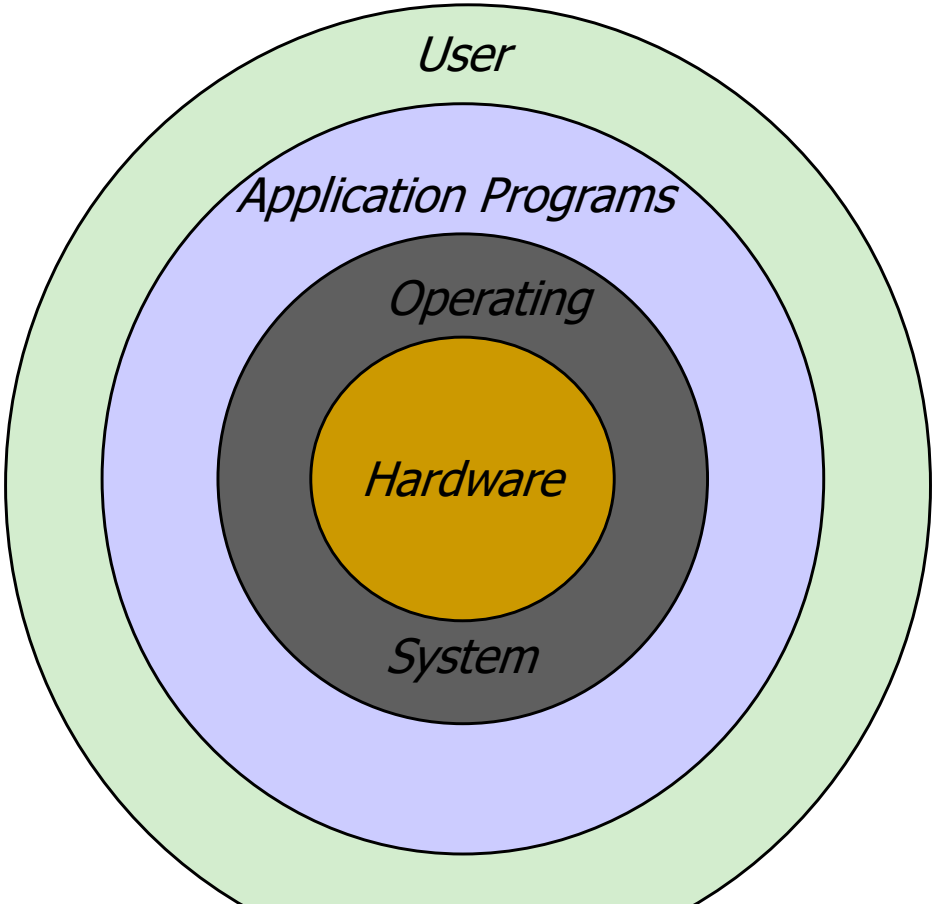
- … capturing variations of the state of the system in relation to its physical environment, and reacting to them in a timely fashion



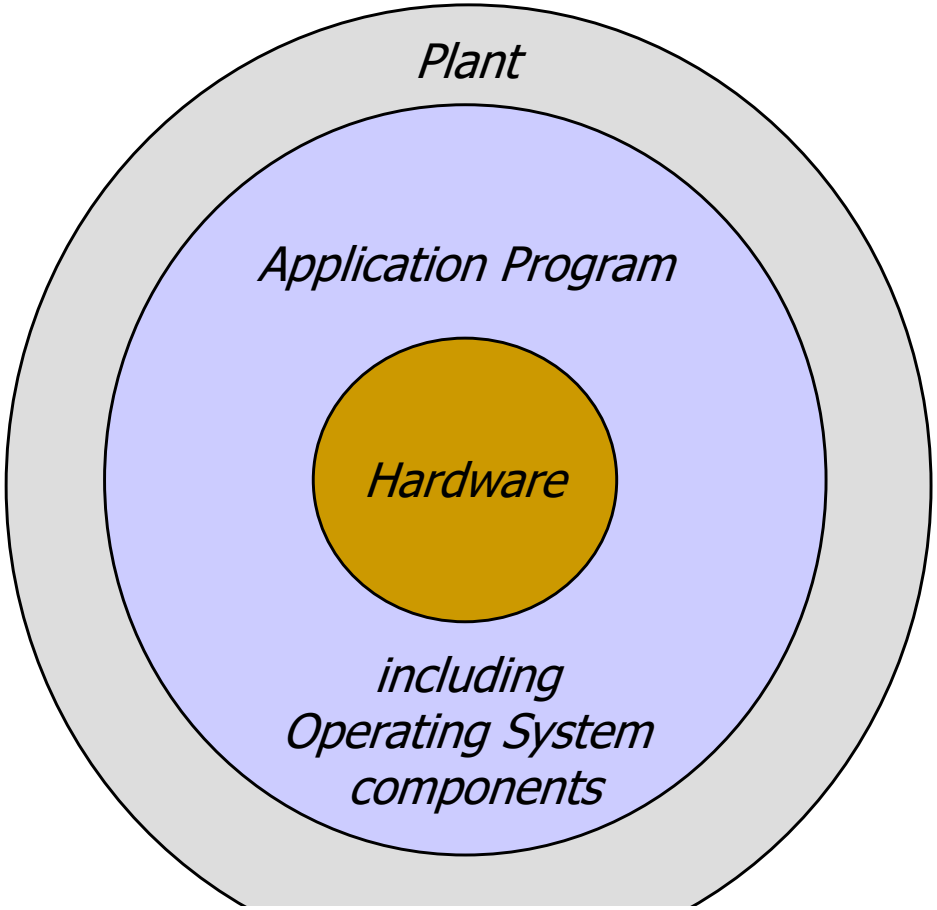
Initial intuition – 2/2

- **Real-time system (cont'd)**
  - Operational correctness does not depend solely on the logical result but also on the time at which the result is produced
    - The computed response has an application-specific utility
    - Correctness is defined in the value domain *and* in the time domain
    - A logically-correct response produced later than due may be bad
- **Embedded system**
  - The computer and its software are fully immersed in an *engineering system* comprised of the external environment subject to its control
    - Defined in terms of physical attributes that can be interacted with

Embedded system – 1/2

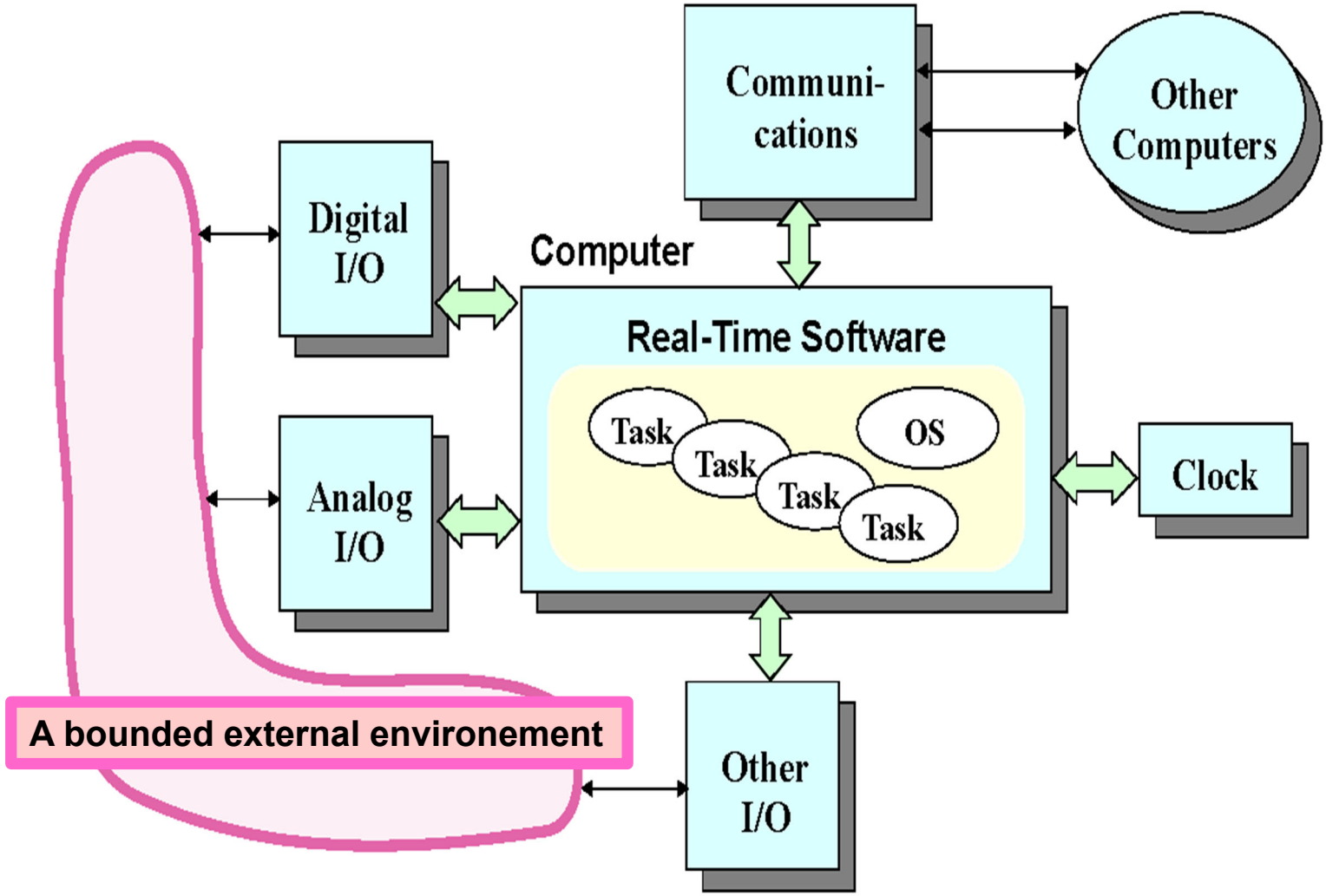


Typical General-Purpose Computing Configuration



Typical Embedded Computing Configuration

Embedded system – 2/2

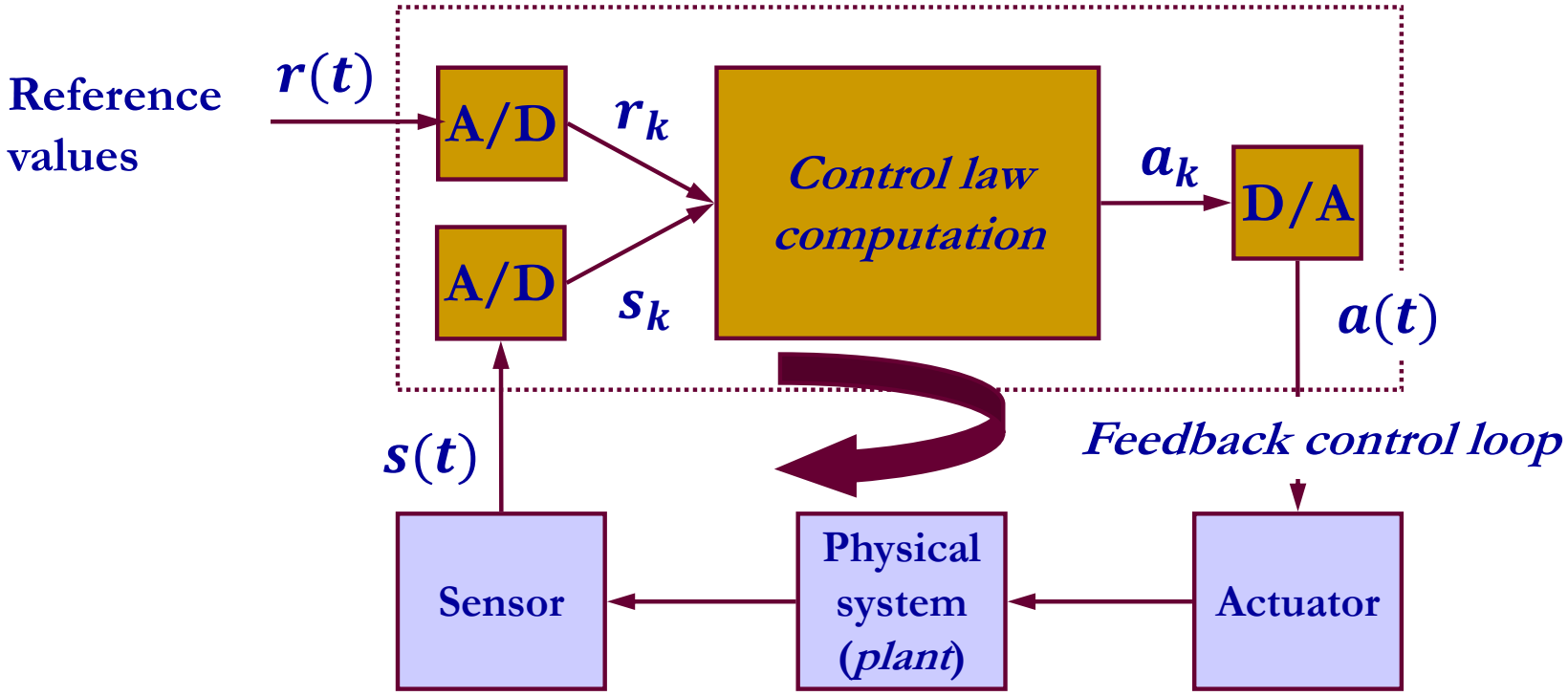


Cybernetics

- Born in 1948 as *the science of control systems*, prerequisite to the **automation of control**
- From Greek's **κυβερνητης** (Latin's "gubernator"), helmsman
  - *Sensing* the external (physical) environment
  - *Computing* the distance from the expected status
  - *Actuating* devices to effect the system or the environment, to stay in course
- Every actuation performed on the external environment causes *feedback* that must be accounted for
- The goal of cybernetics is to calibrate actuation so that the system goal is reached with bounded feedback

Automation of control – 1/4

- A digital system comprised of sensors and actuators



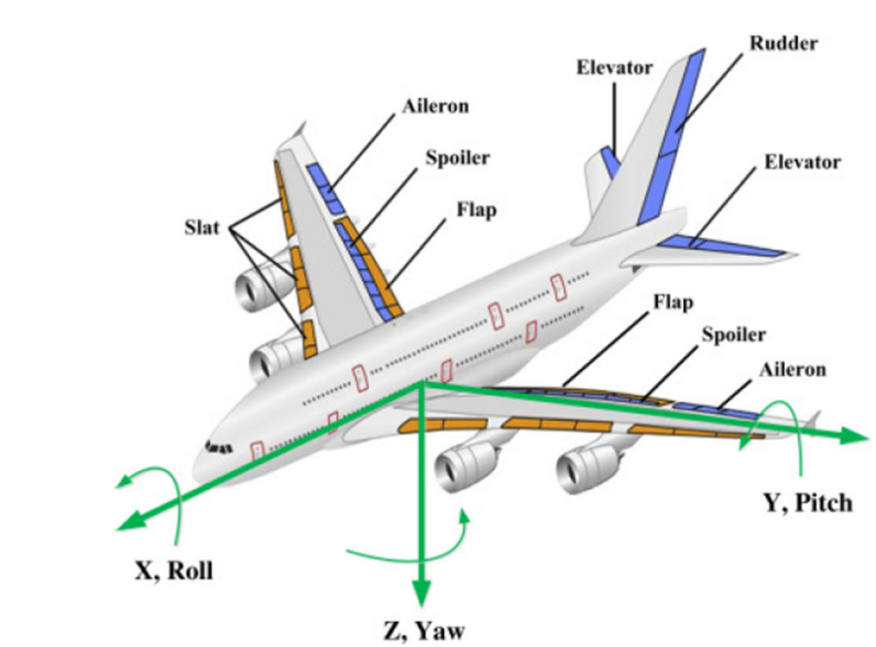
Gradient over time

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

Automation of control – 2/4

- Factors of influence
  - Quality of response (***responsiveness***)
    - Sensor sampling is typically periodic with period T
      - For the convenience of control theory
    - 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 T
    - A rule-of-thumb R/T ratio normally ranges [10 .. 20]

Automation of control – 3/4



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

- Complex systems must support multiple distinct periods T_i
  - A **harmonic** relation among all T_i does help
    - 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

Automation of control – 4/4

(Artificially) harmonic multi-rate system

- 180 Hz cycle
  - Check all sensor data and select sources to sample
  - Reconfigure system in case of read error
- 90 Hz cycle
  - Perform control law for pitch, roll, yaw (internal loop)
  - Command actuators
  - Perform sanity check
- 30 Hz cycle
  - Perform control law for pitch, roll, yaw (external loop) and integration
- 30 Hz cycle
  - Capture operator keyboard input and choice of operation model
  - Normalize sensor data and transform coordinates; update reference data
- *Can you figure how those activities can progress together?*

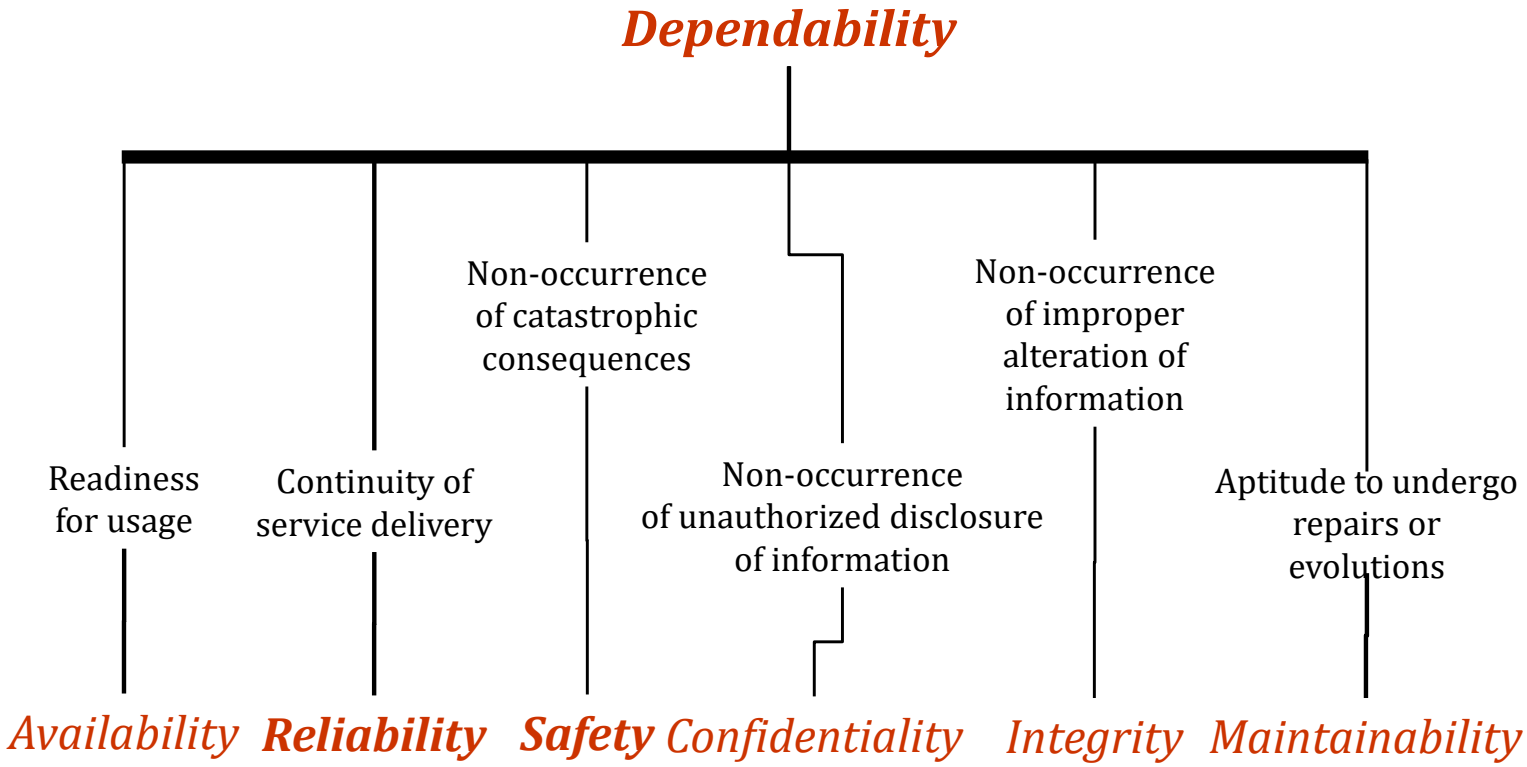
Application requirements

- A control system consists of (distributed) logical and physical resources governed by a *real-time operating system*, RTOS
- The system design overall must meet stringent ***reliability*** requirements
  - Measured in terms of *maximum acceptable probability of failure*
  - For example: 10^{-9}/hour of flight for the Airbus A-3X0 control system
    - One failure allowed in 10^9 hours of flight ($> 114k$ years!)

Traits of real-time systems – 1/2

- **Complex**
  - In algorithms, because of the need to apply discrete control over analog and continuous physical phenomena
  - In development, owing to highly taxing verification and validation processes
- **Heterogeneous** for components and processing activities
  - Multi-disciplinary engineering (spanning control, SW, and system)
- Extremely **variable** in size and scope
  - From tiny and pervasive (nanodevices) to very large (aircraft, plant)
  - In all cases, finite in computational resources
- Provably ***dependable*** [glossary]

Dependability attributes



- See the **self-study material** posted on the lecture schedule page

Traits of real-time systems – 2/2

- Respond to events triggered by the external environment as well as by the passing of time
  - Their schedule is both *event-driven* and *time-driven*
- Ensure continuity of operation
  - Operating without (constant) human supervision
- Inherently *concurrent* and increasingly *parallel* [glossary]
- Temporally ***predictable*** [glossary]
  - Need for static (off-line) verification of correct temporal behaviour
  - How does that relate to ***determinism***? [glossary]

Concurrency vs. parallelism

- ***Concurrent*** programming allows using multiple logical threads of control to reflect *cohesively* the collaborative structure of the solution
  - I do "my" specialized bit, you do "yours"; our mutual waiting is not wasteful
  - Threads form the architecture: they are *long-lived*
- ***Parallel*** programming promotes a divide-and-conquer logic to solve a problem, with multiple threads that work *independently* on the problem space
  - I work as fast as I can and know nothing about you
  - Threads are mindful of throughput: they are *short-lived*

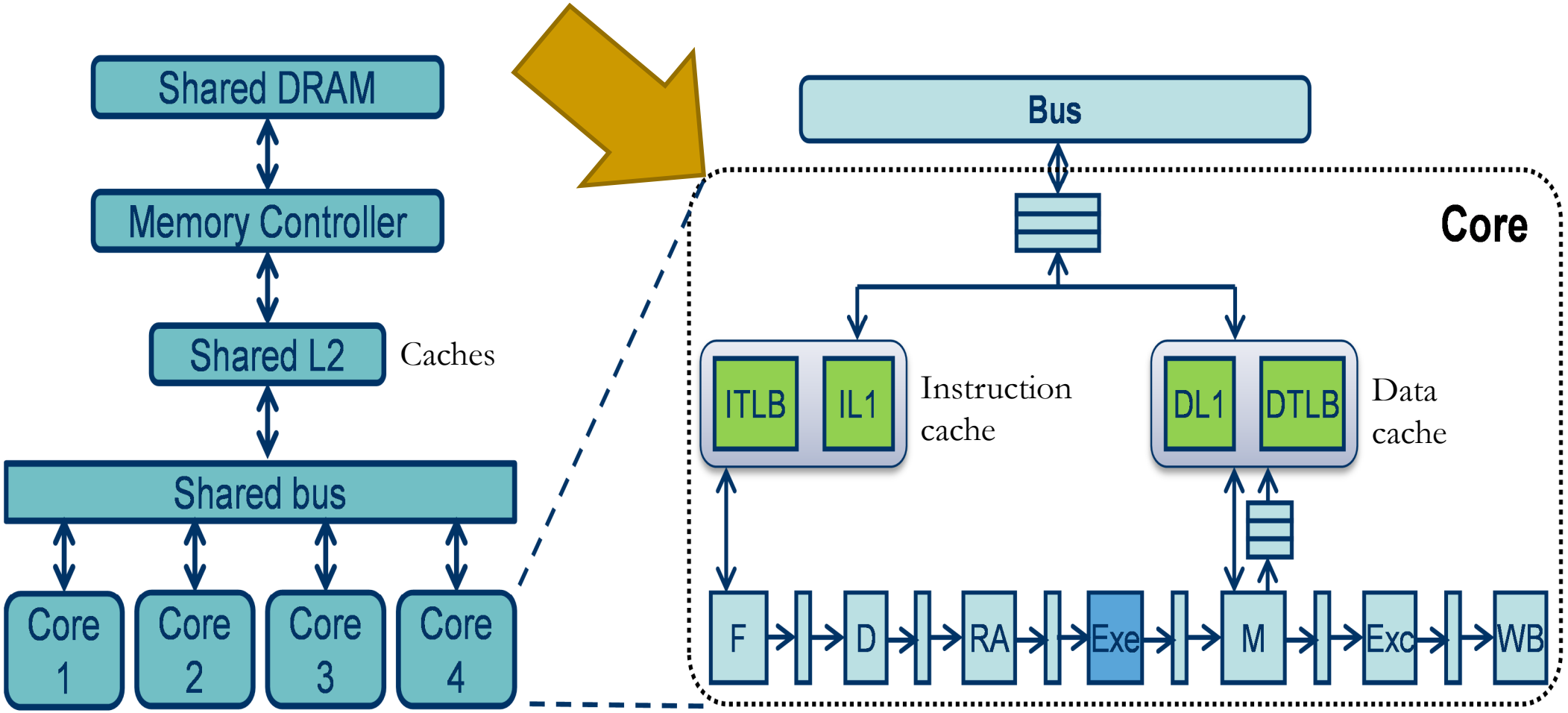
Predictability vs. determinism

- *Predictability* is what can be soundly established a priori
- It may be regarded as a continuum
- Its highest end is full a-priori knowledge
  - Which allows for deterministic reasoning, and yields absolute certainty
- Its lowest end is total absence of a-priori knowledge
  - In the style of "See what happens …"
- Seeking predictability implies reasoning about kinds and degrees of uncertainty, pursuing an acceptable balance
- We must reason conservatively
- Considering *worst-case* conditions, which may be difficult to capture, as very rarely we have full a-priori knowledge of the factors of influence
- At the same time, we must avoid exceedingly pessimistic reasoning

Meeting real-time requirements

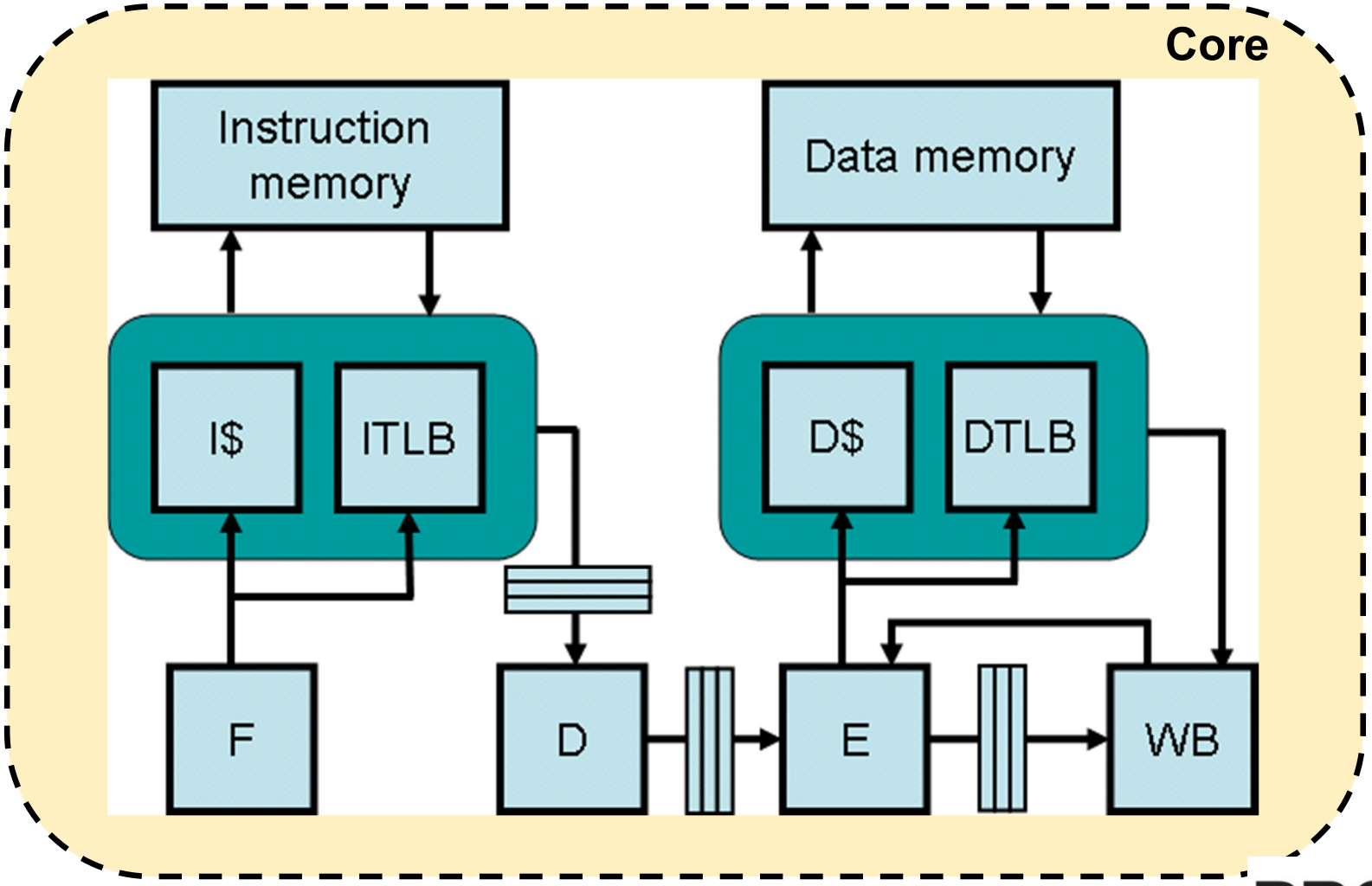
- Minimizing the application tasks' average response time matters to general-purpose computing, but it does **not** to RTS!
- *Real-time* computing is **not** equivalent to *fast* computing
- Given real-time requirements and a concrete implementation, how can one show that those real-time requirements are met?
  - Testing and simulation are **not** sufficient
  - Telling example: maiden flight of space shuttle, April 12, 1981: there was a theoretical $\frac{1}{67}$ probability of a *transient overload* occurring at system initialization
  - It never showed up in testing; it did at launch
- The answer to that lies in seeking system-level ***predictability***
  - Which requires understanding the *worst case*
  - Which further requires understanding how execution works …

Understanding the processor – 1/2



Courtesy of PROXIMA

Understanding the processor – 2/2



Courtesy of PROXIMA

Definitions – 1/6

- ***Task***
  - Concurrent unit of functional architecture: they *never* end
  - Issues *one job at a time*, until completion, to perform actual work
  - Real-time systems' tasks are typically *recurrent* :
  - The body of a task may be seen as an *endless loop*
- ***Job***
  - Unit of work called into execution by a task, following a given law of activation
  - Competes for CPU time under control by a system-level scheduler
  - Needs physical and logical *resources* to execute
  - The job may be seen as a top-level procedure, which carries out the actual functional work required of the corresponding task and then terminates

Definitions – 2/6

- ***Release time***
  - When it occurs, one job becomes 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 recognizes the job as ready
  - It may be set at some *offset* from system start time
    - For example, to avoid congestion on access to the CPU
    - The offset of the *first* job of task τ to the system start time is named ***phase***, φ, and it is one of the attributes of τ

Definitions – 3/6

- ***Jitter***
  - Variability in the arrival of the release time
  - Variability in the arrival of input (data freshness)
  - Variability in the delivery of output (stability of control)
- ***Inter-arrival time***
  - Separation between release time of any two successive jobs
  - Job is *periodic* if inter-arrival time is constant
  - Job is *sporadic* if a guaranteed minimum such value exists
  - Job is *aperiodic* otherwise
- ***Execution time*, C**
  - For any job J_i, C_i may vary between a *best-case* (BCET) C_i^b and a *worst-case* (WCET) C_i^w

Definitions – 4/6

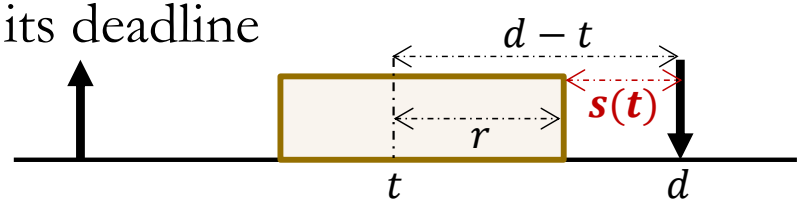
- ***Deadline***
  - The latest time by which a job must complete its execution
  - May be < (*constrained*), = (*implicit*), > (*arbitrary*) than the next job's release time
- ***Response time***, R
  - The time span between the job's release and its actual completion
  - It may be longer than the job's execution time ($R \geq C$)
- The longest admissible response time for a job j_i is termed the job's *relative deadline*, D_i
  - $R_i \leq D_i$ should always hold
- The algebraic summation of release time and relative deadline is termed *absolute deadline*, d_i, which is a specific point in time

Definitions – 5/6

- Deadlines are said to be **hard**
  - If the consequences of a job completing past its deadline are serious and possibly intolerable
  - Satisfaction of all deadlines must be proven offline
- Deadlines are said to be **soft**
  - If the consequences of a job *occasionally* completing past the assigned deadline are tolerable
  - The quantitative interpretation of "occasional" may be established in probabilistic terms or in terms of ***utility***
- Deadlines are said to be **firm**
  - When they are soft but have utility ≤ 0 past the deadline point, and therefore may cause damage if missed

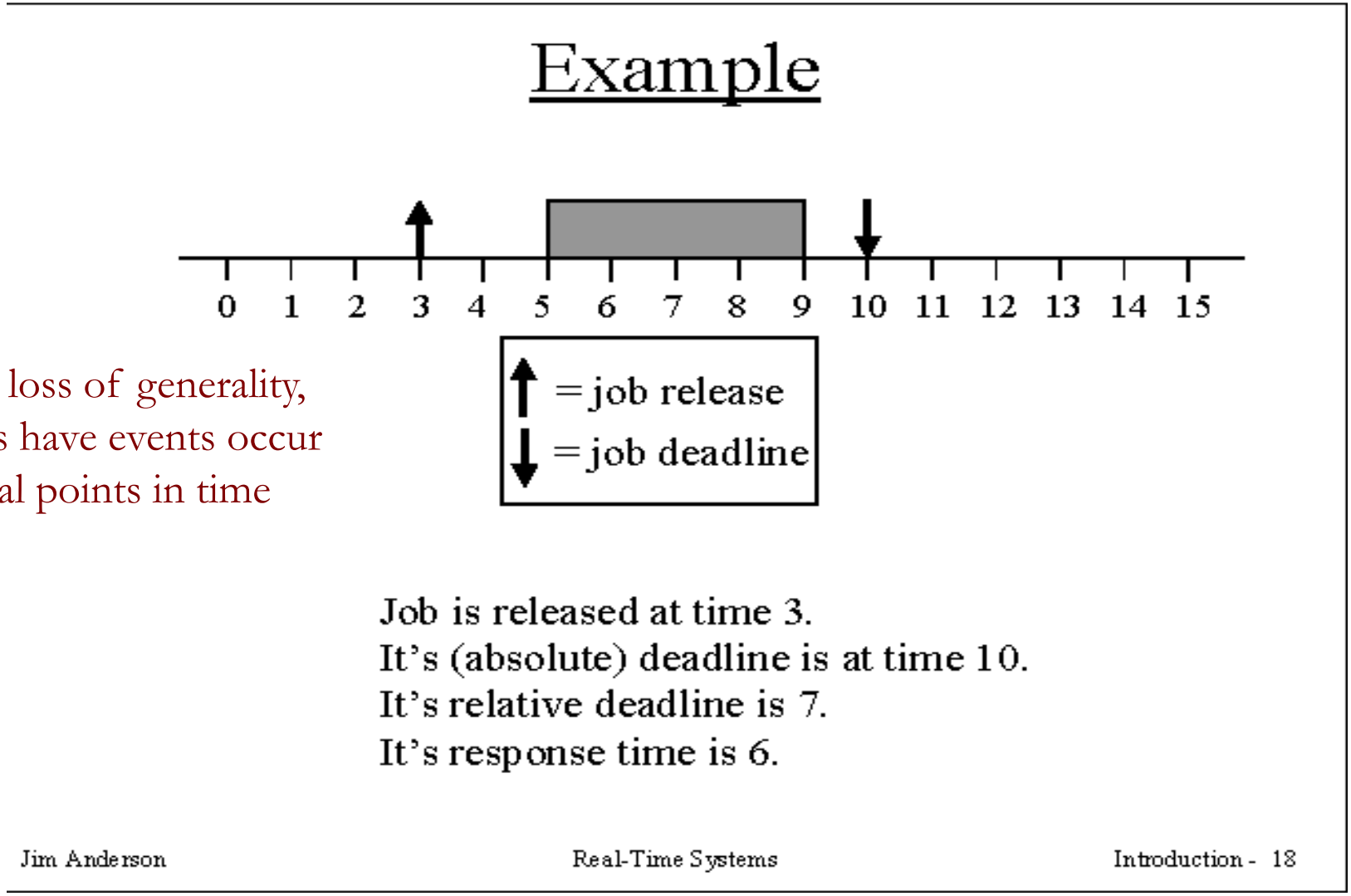
Definitions – 6/6

- ***Laxity***, aka ***slack*** at time t
  - $s(t) = (d - t) - r$ is the *slack* at time t of job J with absolute deadline d and remaining time of execution r
    - A job with non-negative laxity meets its deadline



- ***Tardiness***
  - The span between a job's response time and its deadline
    - A job with negative laxity has tardiness
- ***Usefulness***
  - Value of (residual) utility of the job's computational product as a function of its tardiness

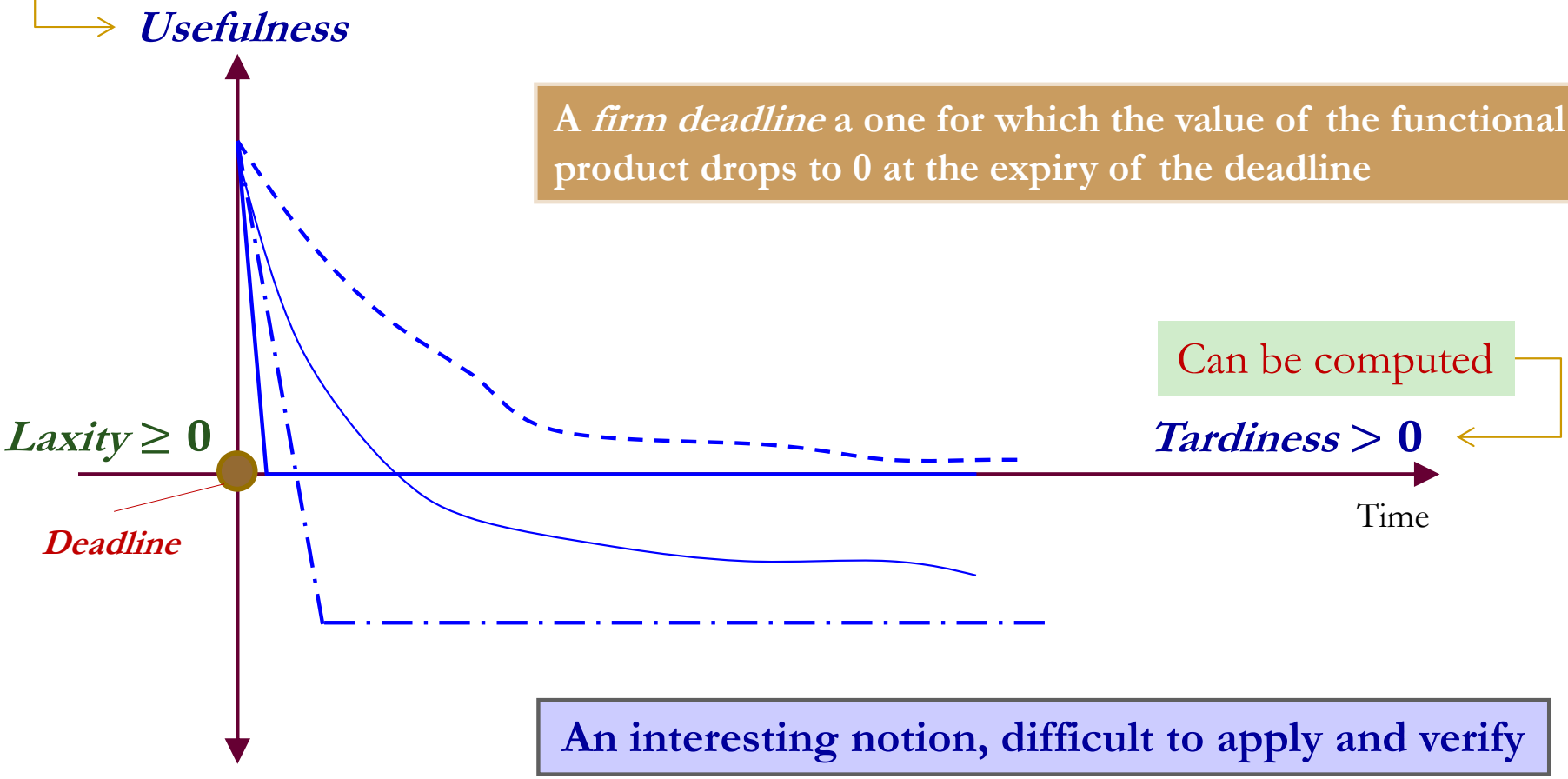
An example timeline



Without loss of generality, timelines have events occur at integral points in time

Utility function

Difficult to quantify



A *firm deadline* a one for which the value of the functional product drops to 0 at the expiry of the deadline

Can be computed

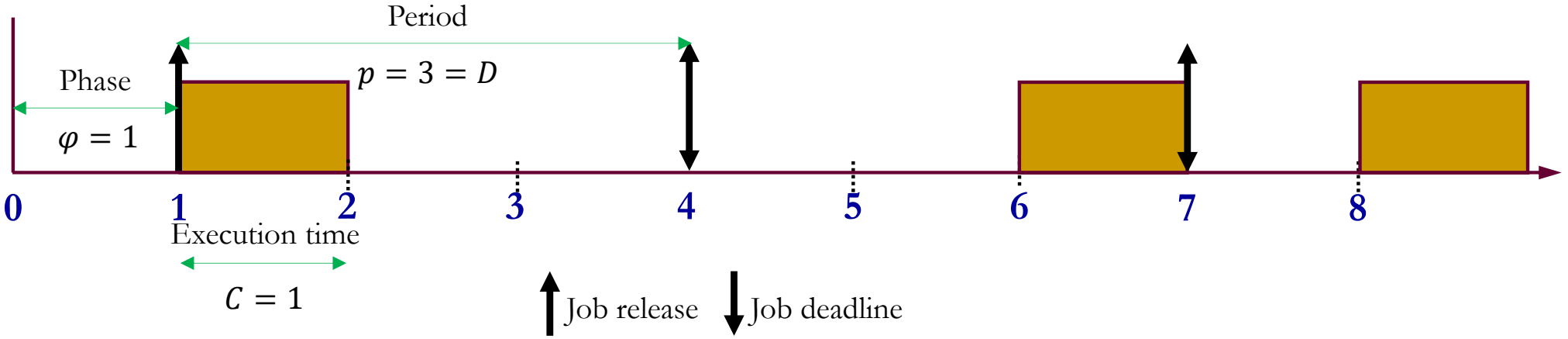
An interesting notion, difficult to apply and verify

An initial taxonomy

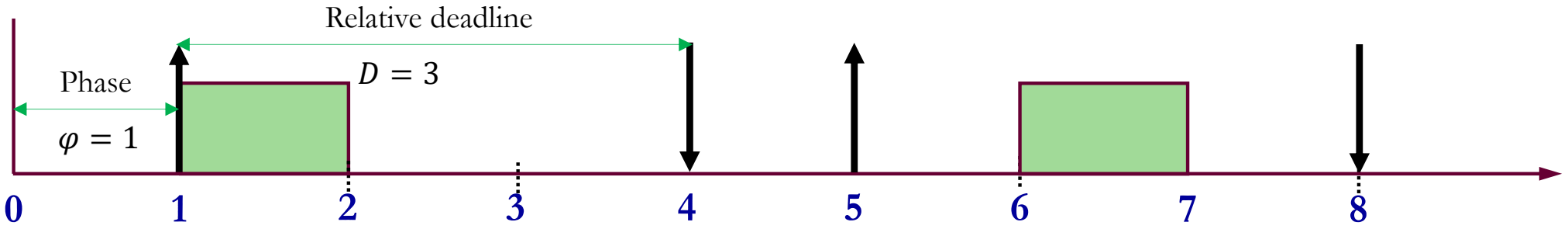
- ***Periodic*** tasks
  - Their jobs become ready at regular intervals of time, *T*
  - Their arrival is synchronous to some time reference
- ***Aperiodic*** tasks
  - Recurrent but irregular
  - Their arrival cannot be anticipated (asynchronous)
- ***Sporadic*** tasks
  - Their jobs become ready at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

Example

- This is a *periodic* task



- This is a *sporadic* task



Abstract modelling – 1/8

- ***Active resources*** (processor, server)
  - They "do" what they have to do
    - Execute machine instructions, move data, process queries, etc.
  - Jobs must acquire them to progress toward completion
    - Contention occurs on access to them
- Active resources have a type
  - Those of the same type can be used interchangeably
  - Those of different types cannot
    - For example, processors may have different speed, which affects the rate of progress for the jobs that run on them

Abstract modelling – 2/8

- ***Passive resources*** (memory, shared data, semaphores, …)
  - A passive resource doesn't do anything per se, but jobs *may need* it to make progress
  - They may be reused if use does not exhaust them
    - If always available in sufficient quantity to satisfy all needs, they are said to be *plentiful* and can be ignored
- Passive resources that matter to real-time systems are those that may cause *bottlenecks*
  - Access to memory may matter more (owing to *arbitration*) than memory itself (which may be considered plentiful)

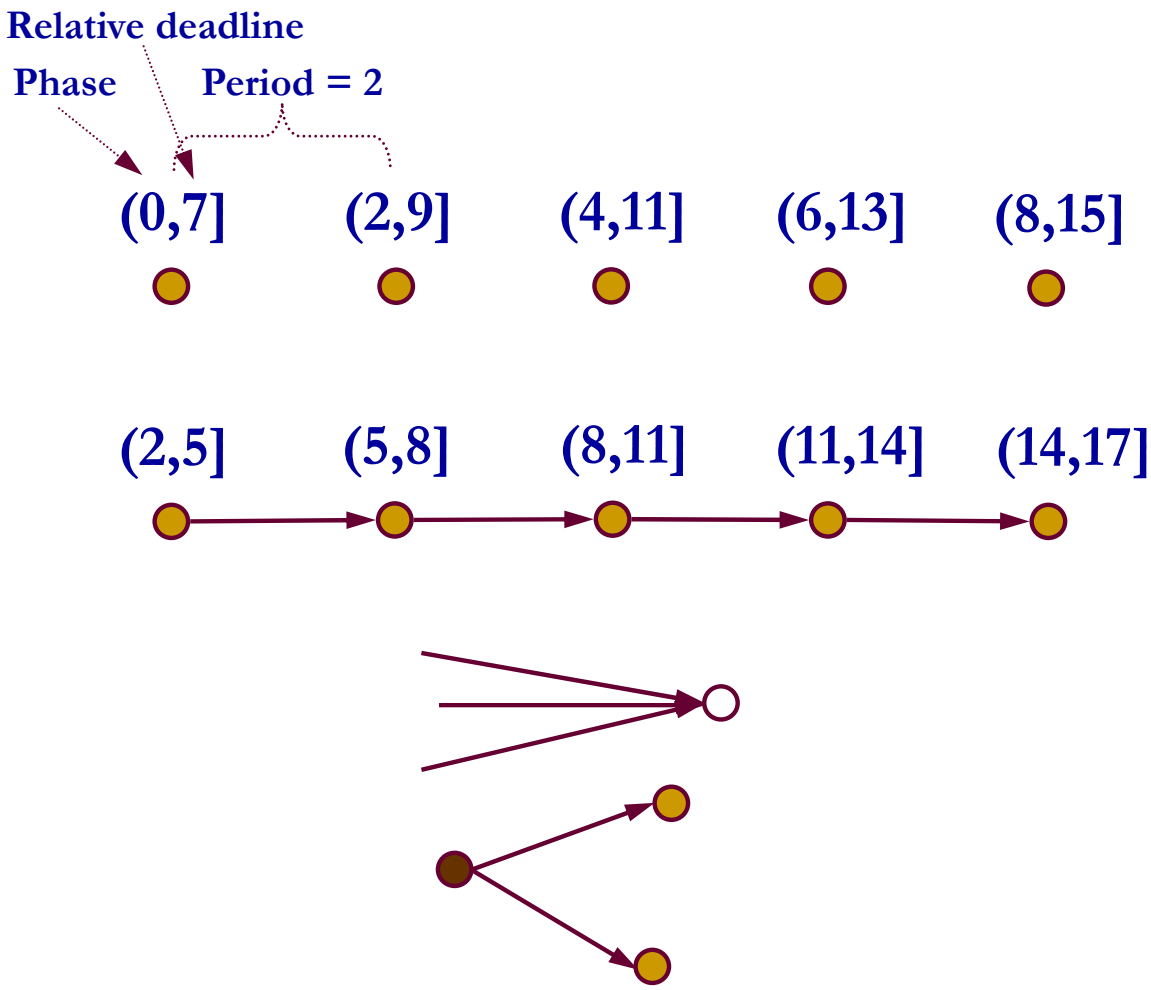
Abstract modelling – 3/8

- Permissibility of job **preemption**
  - May depend on the capabilities of the execution environment (e.g., ***non-reentrancy***) but also on the programming style
  - Preemption causes time and space overhead
- Job ***criticality***
  - Akin to a criterion of execution eligibility
  - It indicates which activities must be guaranteed no matter what
- Permissibility of resource preemption
  - Some resources are intrinsically preemptable
  - Others do not permit it
  - Which ones?

Abstract modelling – 4/8

- Fixing execution parameters
  - The time that elapses between when a periodic job becomes ready and the next period p is certainly $\leq p$
  - Setting phase $\varphi > 0$ and relative deadline $\mathrm{D} < p$ for a job may help limit its output jitter (**why?**)
  - The jobs of a system may be independent of one another
    - They may execute in any relative order
  - Or they may be subject to *precedence constraints*
    - As is the case in collaborative architectural styles
    - E.g., producer – consumer

Task precedence graphs



Independent jobs

Dependent jobs

Job of type AND (join)

Job of type OR (branch) typically followed by a join job

Types of precedence constraints

- The release time of a job with successors cannot follow the release of any of its successors
- ***Effective release time (ERT)***
  - For a job J_i with predecessors $\{J_{k=1,\dots,i-1}\}$, ERT_i is the *latest* value between its own release time and the maximum ERT of its predecessors, ERT_k, plus C_k
  - J_i cannot be released before $max_k(R_k)$, when all its predecessors have completed
- The absolute deadline of a job with predecessors cannot precede the absolute deadline of any of its predecessors
- ***Effective deadline (ED)***
  - For a job J_i with successors $\{J_{k=i+1,\dots,n}\}$, ED_i is the *earliest* value between D_i and the minimum ED of its successors, ED_k, less C_k
  - $D_i \leq min_k(ED_k - C_k)$: no successor may run before the latest end of its predecessors
- For single processors with preemptive scheduling, ERT and ED are the only precedence constraints that matter

Abstract modelling – 5/8

- ***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 $S = \{\tau_i\}, i = 1, \ldots, N$
    - Defined as LCM (least common multiple) of task periods $\{p_i\}$
    - It is the time span within which all tasks have issued at least one job
  - ***Utilization***
    - For every task τ_i : defined as the ratio between execution time and period : $U_i = \frac{C_i}{p_i} \leq 1$
    - For the system (*total utilization*) : $U = \sum_i U_i \leq m$, where m is the number of CPUs ($m = 1$, for now)

Abstract modelling – 6/8

BCET: best-case execution time
WCET: worst-case execution

- Selecting jobs for execution
  - The scheduler assigns a job to the processor's CPU
  - 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 ≥ than its BCET and ≤ than its WCET
  - All precedence constraints in place among tasks as well as among resources are satisfied

Abstract modelling – 7/8

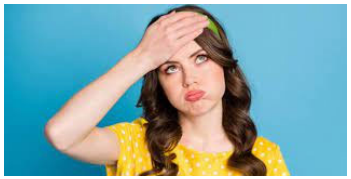
- A *valid schedule* is said to be ***feasible*** if it satisfies the temporal constraints of every job
- A *job set* is said to be ***schedulable*** by a scheduling algorithm if that algorithm always produces a *valid* schedule for that problem
- A *scheduling algorithm* is ***optimal*** if it always 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 modelling – 8/8

- Two algorithms are of prime interests for real-time systems
  - The *scheduling algorithm*, which 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 consistently, divides the analysis algorithms in
  - ***Feasibility tests***, which are exact (necessary and sufficient)
  - ***Schedulability tests***, which are only sufficient

Cute teasers

- Björn Brandenburg's post on SIGBED Blog …
  - https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/
- … shows a cute way to reconcile the basic theory seen so far and simple-but-sound conveniency programming
  - Check it out
- Linux is certainly not what you would expect to find in a real-world embedded real-time system
  - Yet, lots of users like "convenience-development"
  - https://www.zdnet.com/article/to-infinity-and-beyond-linux-and-open-source-goes-to-mars/



Further characterization – 1/2

	Time-Share Systems	Real-Time Systems
Capacity	High throughput	Ability to meet timing requirements: Schedulability
Responsiveness	Fast average response	Ensured worst-case latency
Overload	Fairness	Stability of critical part

Further characterization – 2/2

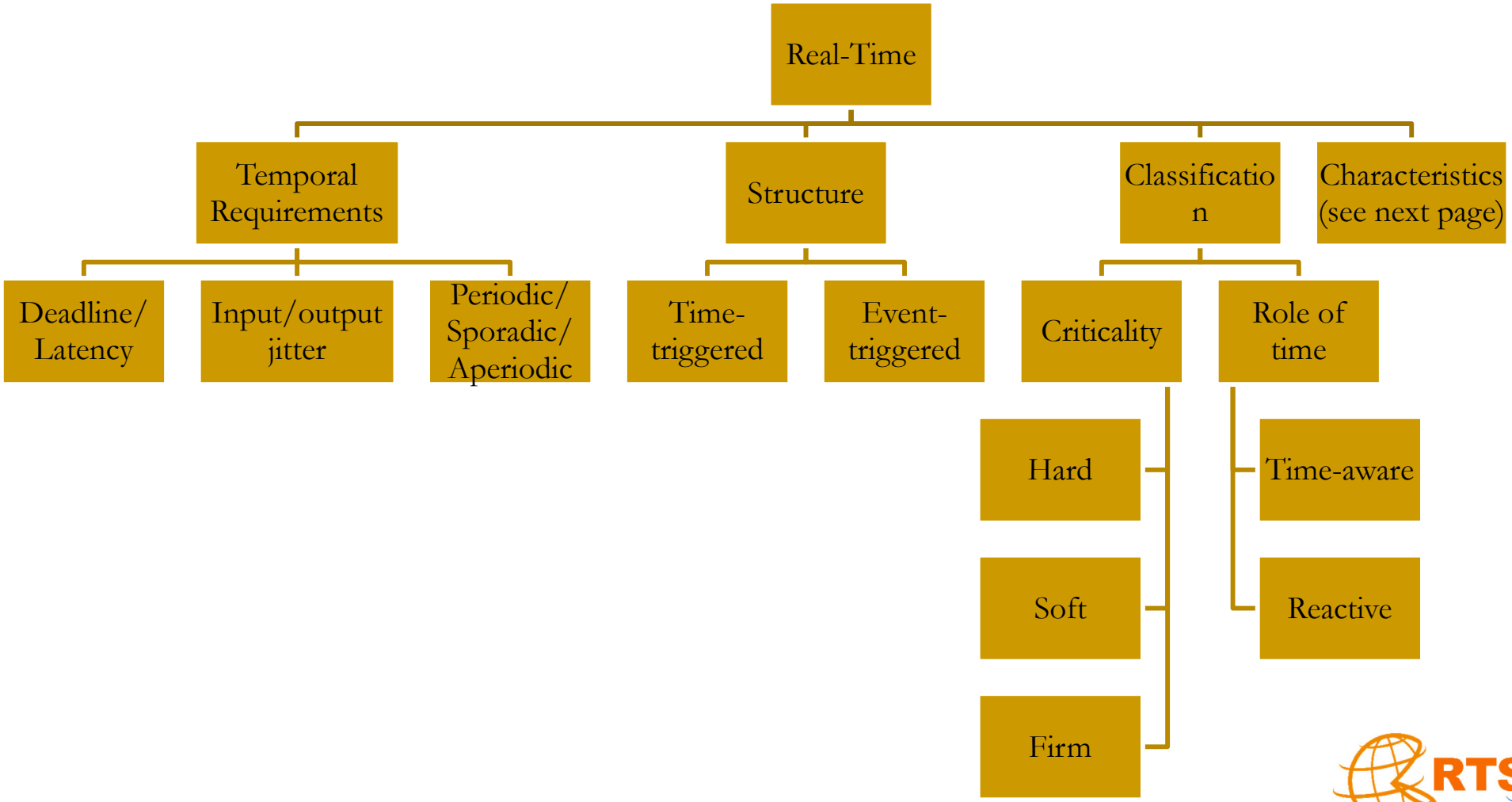
- The design and implementation of an RTS mind the worst case before considering the average case (if at all)
  - Improving the average case is of *no use* and it may even be counterproductive
  - The processor cache is an *average-case* device
  - It operates *adversarially* to the needs of any RTS

- Stability of control prevails over fairness
  - The former concern is selective, the other is general
- When feasibility is proven, starvation is of no consequence
  - The non-critical part of the system may experience starvation

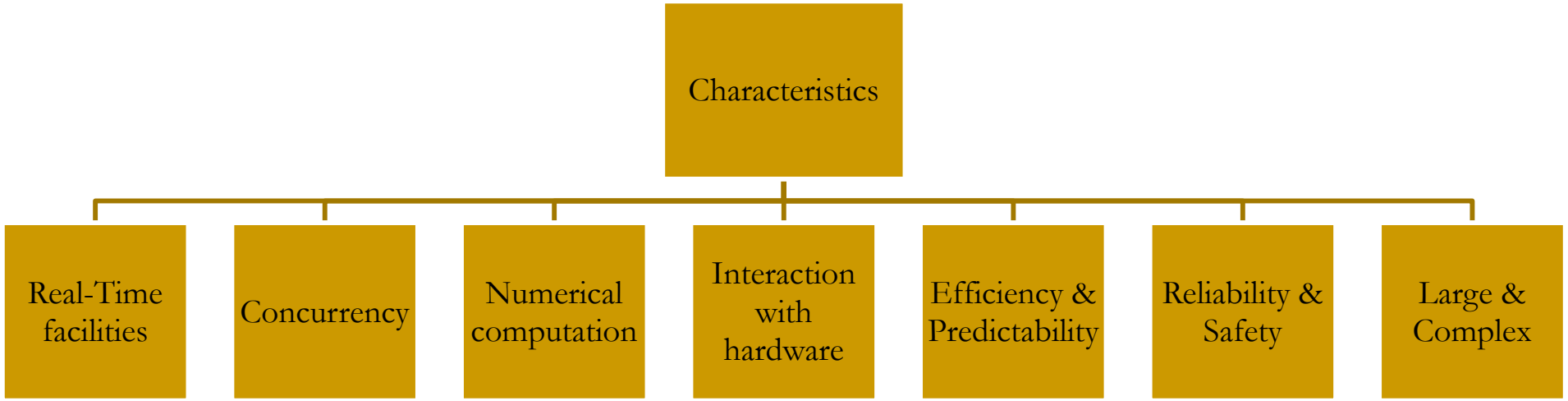
Summary – 1/3

- From an initial intuition to a more solid definition of real-time embedded systems
- Bird's-eye survey of application requirements and key characteristics
- Taxonomy of tasks
- Abstract models to help reason in general about real-time systems

Summary – 2/3


Real-Time
Temporal Requirements
Structure
Classification
Characteristics (see next page)
Deadline/ Latency
Input/output jitter
Periodic/ Sporadic/ Aperiodic
Time-triggered
Event-triggered
Criticality
Role of time
Hard
Soft
Firm
Time-aware
Reactive

Summary – 3/3


Characteristics
Real-Time facilities
Concurrency
Numerical computation
Interaction with hardware
Efficiency & Predictability
Reliability & Safety
Large & Complex

2. Scheduling basics

Where we begin to familiarize with the scheduling algorithms that decide how the CPU is assigned to tasks which have a job ready for execution

Common approaches – 1/3

- ***Clock-driven (time-driven) scheduling***
  - Scheduling decisions are made at design time and actuated at fixed time instants during execution by a *scheduler*
    - Such time instants occur at intervals signaled by *clock* via *interrupts*
  - The scheduler dispatches to execution (calls) the job due in the current time interval and then suspends itself until the next schedule time
    - The scheduler *is* the prime actor: the jobs are mere called procedures
  - Jobs must complete within the assigned time intervals
    - This manner of scheduling does not require *preemption*
    - All scheduling parameters must be known in advance
    - The schedule, computed offline, is fixed forever
    - The scheduling overhead incurred at run time is very small

Common approaches – 2/3

- ***Weighted round-robin scheduling***
  - With basic round-robin (which requires preemption)
    - All the ready jobs are placed in a FIFO queue
    - CPU time is quantized, i.e., assigned in slices
    - Job at head of queue is dispatched to execution for one time slice
      - If not complete by end of slice, it goes to tail of queue
      - All jobs in queue are given one single slice per round (full traversal of queue)
    - Not fit for jobs with precedence relations (τ_i must complete before τ_j may start) whose execution needs multiple slices
    - Fit for producer-consumer pipelines that progress in one-slice increments
  - With weighted correction to it (used in network scheduling)
    - Jobs are assigned CPU time according a (fractional) 'weight' attribute
    - Job J_i gets $\boldsymbol{\omega_i}$ time slices per round
    - One full round corresponds to $\sum_i \boldsymbol{\omega_i}$ progress for the ready jobs

Common approaches – 3/3

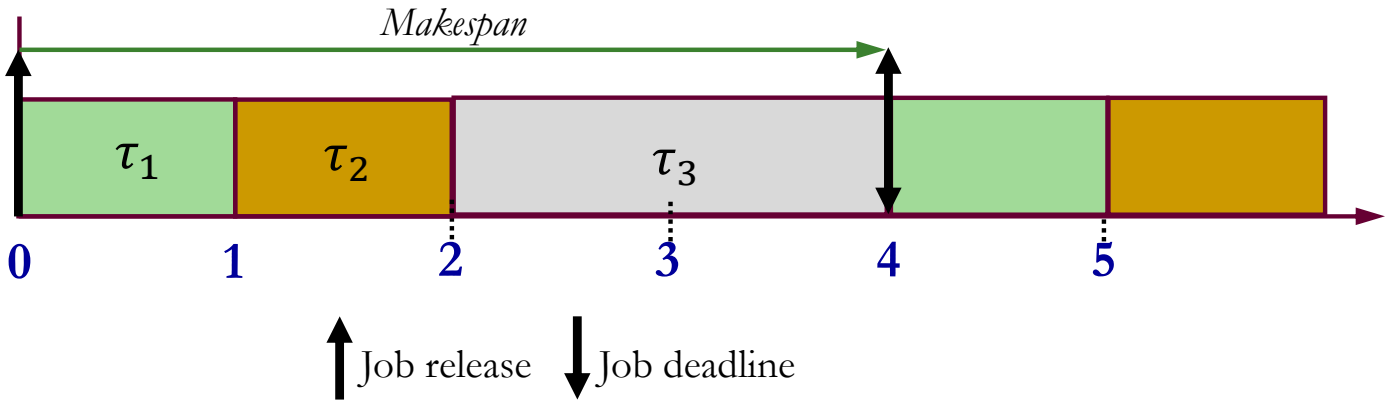
- ***Priority-driven (event-driven) scheduling***
  - A *greedy* class of algorithms
    - Never leave available processing resources unused if they are wanted
    - An available resource may stay unused only if no job is ready to use it
    - May not always be good: *clairvoyancy* may show that deferring an assignment of CPU may improve response time (Why?)
    - Anomalies may occur when job parameters change dynamically
  - The jobs that contend for execution are kept in a *ready queue*
  - Scheduling takes place when the ready queue changes
    - Such scheduling events are called ***dispatching points***
    - Scheduling decisions are made online
    - Dispatching employs *preemption*

Preemption vs. non preemption

- Can we compare preemptive to non-preemptive scheduling in terms of performance?
  - No single response for all cases
  - Preemptive scheduling *provably better* than non-preemptive when all jobs have same release time and preemption overhead is negligible
    - But preemption costs hardly are negligible
- Does the improvement in the last finishing time (*minimum makespan*) under preemptive scheduling pay off the time overhead of preemption?
  - We do *not* know in general …
  - For 2 CPUs, the minimum makespan for non-preemptive scheduling is *never worse* than 4/3 of that for preemptive

Makespan

$$\tau_1 = \{p_1 = 4, e_1 = 1, D_1 = p_1\}$$
$$\tau_2 = \{p_2 = 4, e_2 = 1, D_2 = p_2\}$$
$$\tau_3 = \{p_3 = 4, e_3 = 2, D_3 = p_3\}$$

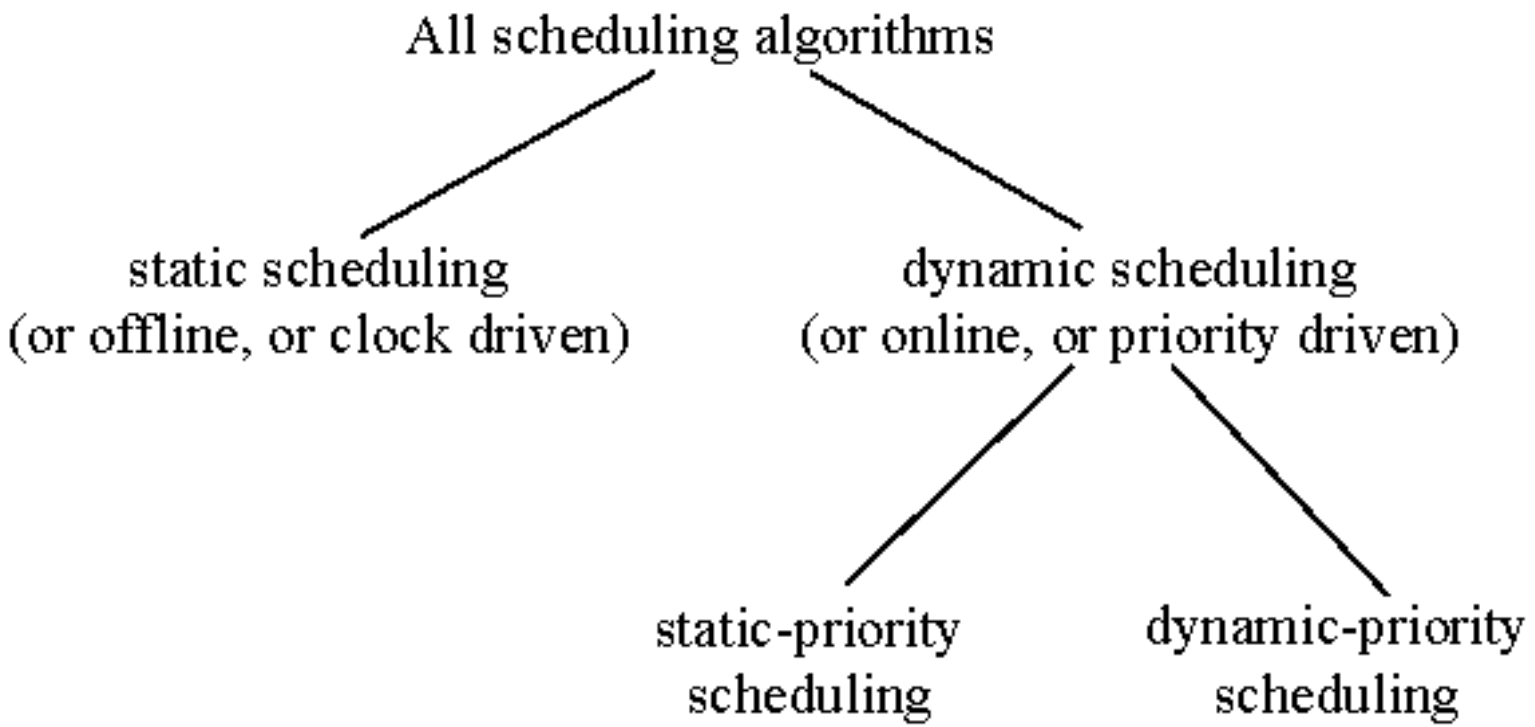


From the release of the first job of the first task to the completion of the first job of the last task

Interpreting predictability of execution

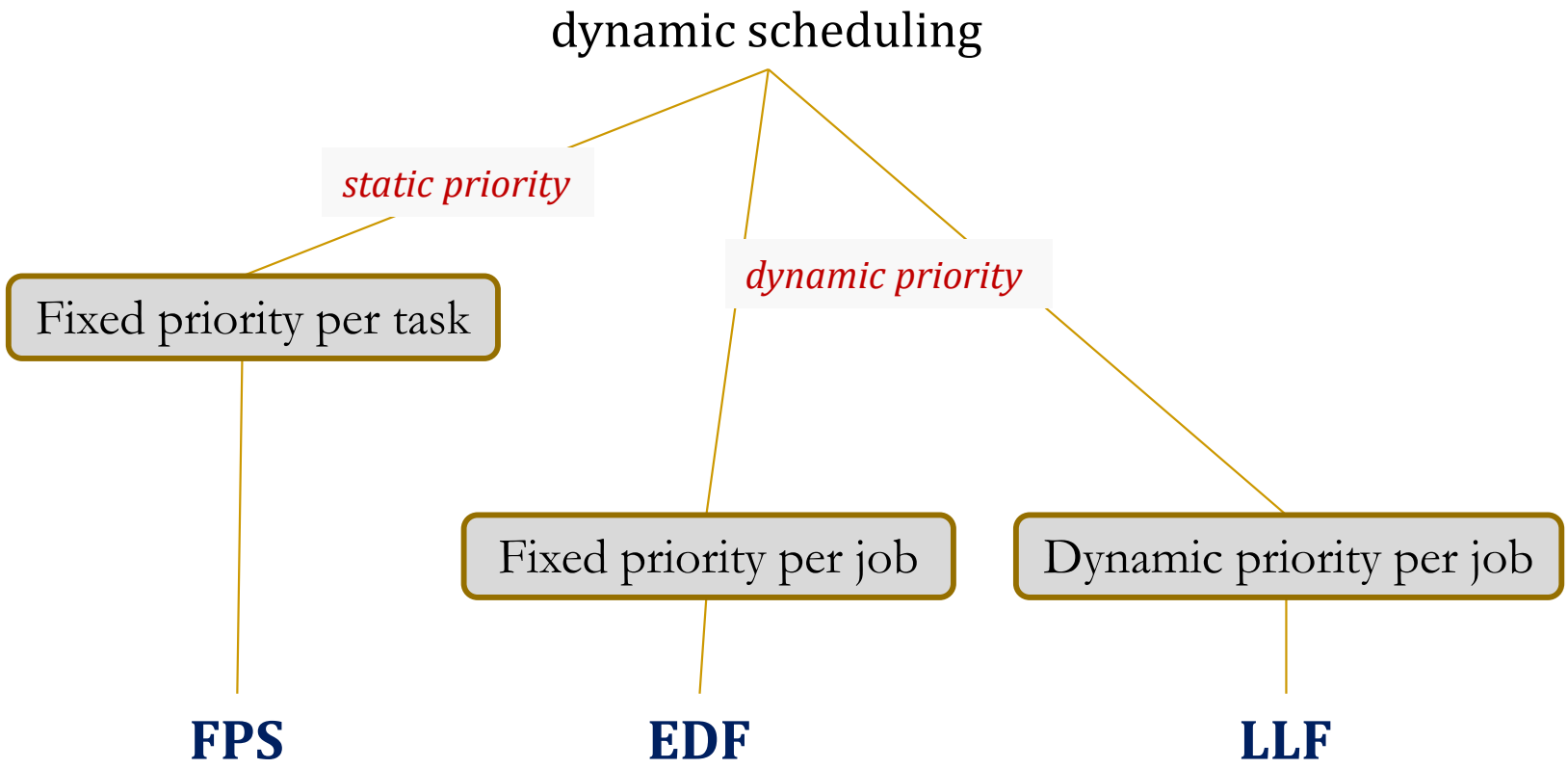
- The execution of system *S* under a given scheduling algorithm *A* is *predictable* if the start time and response time of every job in *S* strictly vary between a *minimal* and a *maximal schedule*
  - *Maximal schedule* is created by *A* under worst-case conditions for contention and execution demands
  - *Minimal schedule*, analogously for the best case
- Predictable scheduling algorithms can be reasoned about in terms of worst-case behavior!
- **Theorem**: the execution of *independent* jobs with given release times under preemptive priority-driven scheduling on a single processor **is** predictable

Classification of Scheduling Algorithms



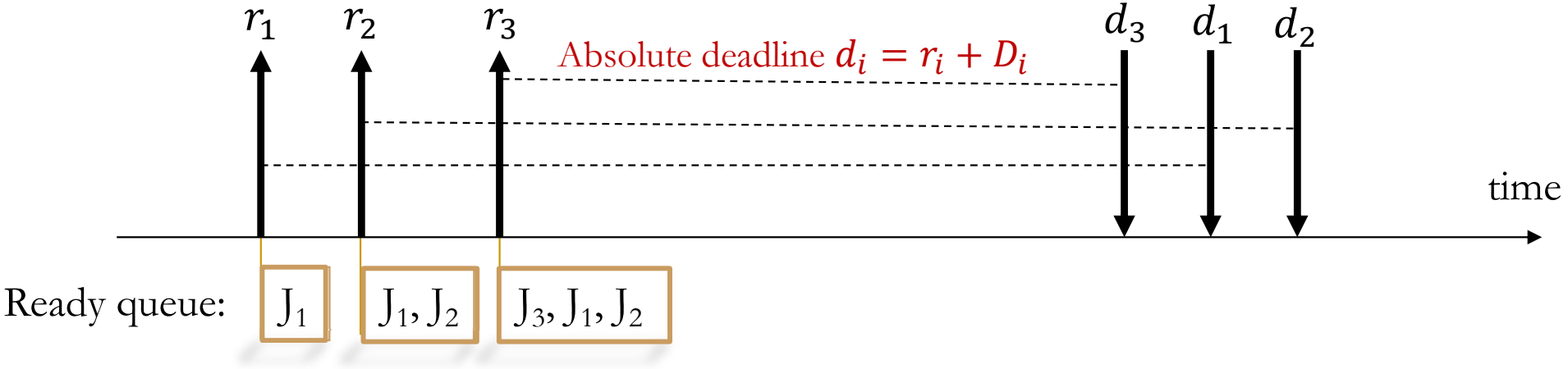
Jim Anderson Real-Time Systems Introduction - 30

Taxonomy of dynamic scheduling



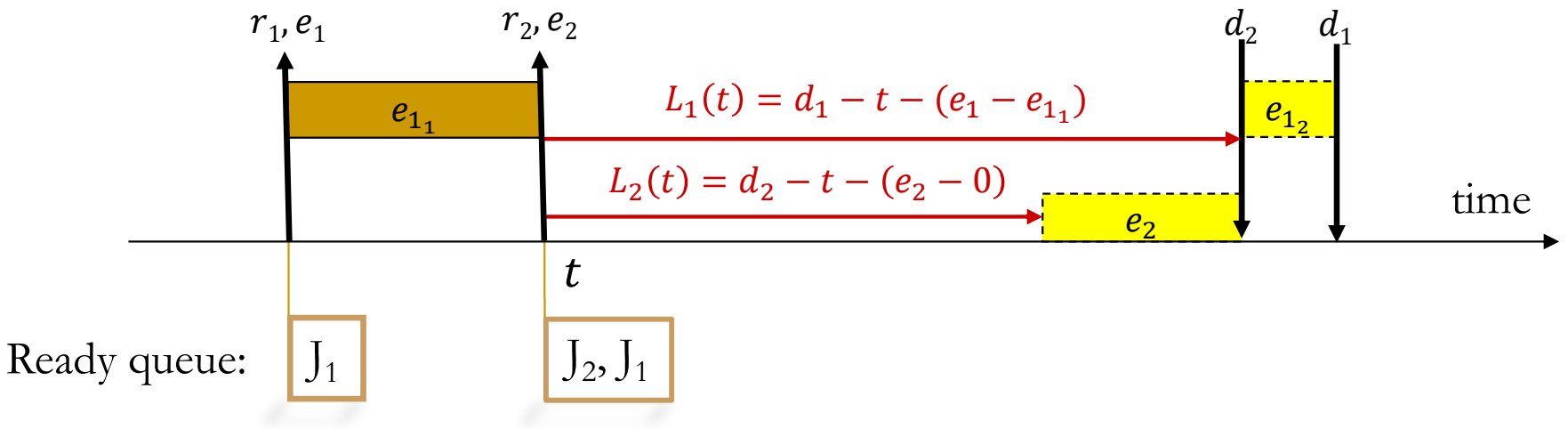
Ways to optimality – 1/2

- ***Earliest Deadline First*** (EDF) scheduling assigns priorities in inverse relation to *absolute deadlines*
  - This makes it dynamic per task but static per job
  - Ready queue changes on job release and on job completion
- [Liu & Layland: 1973]: EDF scheduling is ***optimal*** for single-CPU systems with independent jobs and preemption
  - For any job set, EDF produces a feasible schedule if one exists
  - The optimality of EDF breaks otherwise (e.g., no preemption, parallelism)



Ways to optimality – 2/2

- ***Least Laxity First*** (LLF) scheduling assigns priorities in inverse relation to *laxity* $L(t)$
  - Jobs' priority, $L(t)$, varies with t: dynamic per job and more costly than EDF
  - $L_i(t) = d_i - t - Y_i(t)$, where $Y_i(t)$ is the residual execution time needed for τ_i at time t, with release time r_i and relative deadline D_i $(d_i = r_i + D_i)$
  - Ready queue reordering occurs on job release and job completion
- [Liu & Layland: 1973]: LLF is as ***optimal*** as EDF



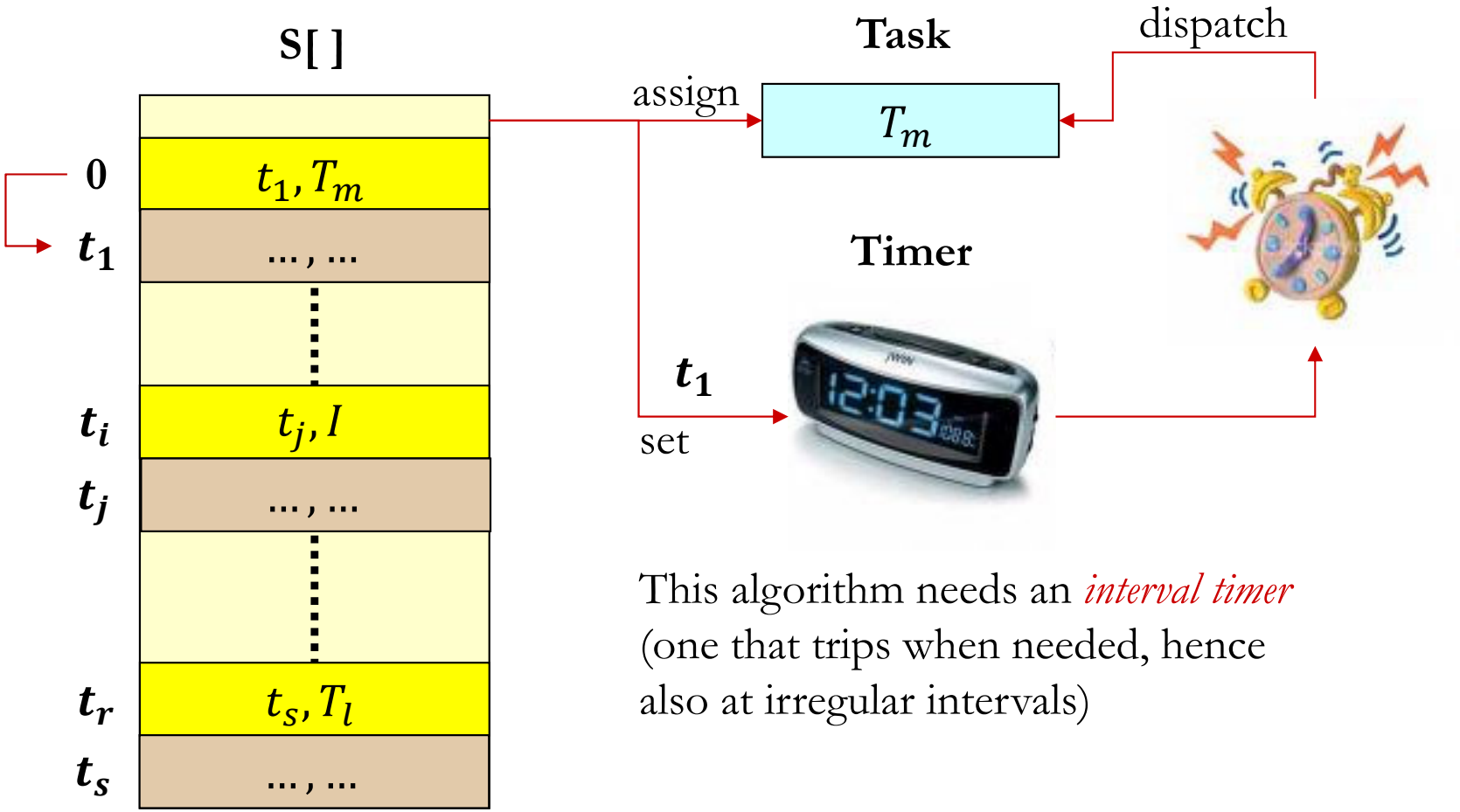
Clock-driven scheduling – 1/8

- ***Workload model***
  - N periodic tasks, for N constant and statically defined
  - 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 at design to a table **S** of decision times t_k, where:
  - $S[t_k] = \tau_i$ if a job of task τ_i must be dispatched at time t_k
  - $S[t_k] = I$ (*idle*) if no job is due at time t_k
  - Schedule computation can be as sophisticated as we like since we pay for it only at design time
  - Jobs *cannot overrun* otherwise the system is in error

Clock-driven scheduling – 2/8

Input: stored schedule $S[t_k]$, $k = \{0, .., N-1\}$; H (hyperperiod)
SCHEDULER ::
  $i := 0$; // slot number
  $k := 0$; // N-modulo correspondent if index i
  set timer to expire at t_k ;
  do forever :
    sleep until timer interrupt;
    if aperiodic job executing **then** preempt; **end if**;
    T $:= S[t_k]$; // current task
    $i := i + 1$;
    $k := i \ \boldsymbol{mod}\ N$;
    set timer to expire at $t_k + \lfloor i/N \rfloor \times H$;
    if $T = I$ **then** execute job at head of aperiodic queue;
      else execute job of task T;
    end if;
  end do;
end SCHEDULER

Clock-driven scheduling – 3/8



This algorithm needs an *interval timer* (one that trips when needed, hence also at irregular intervals)

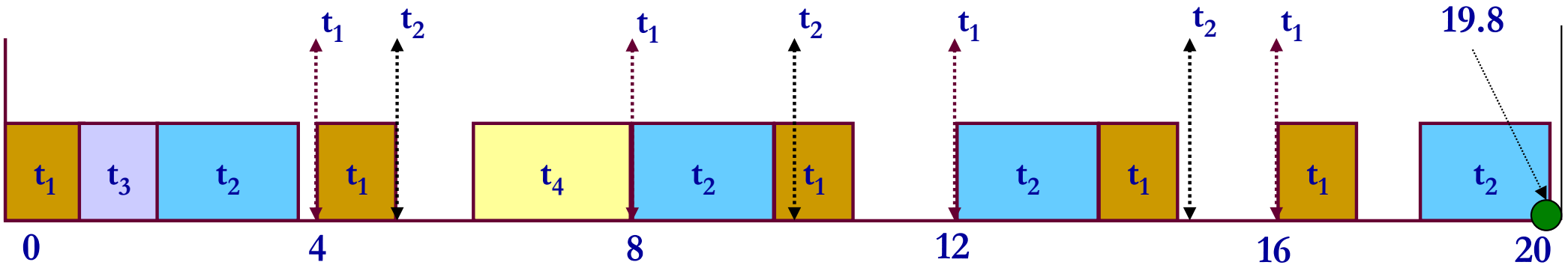
Where the t_j values need *not* be equally spaced

Example

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

$$\boldsymbol{J = \{t_1 = (0,4,1,4), t_2 = (0,5,1.8,5), t_3 = (0,20,1,20), t_4 = (0,20,2,20)\}}$$

$$\boldsymbol{U = \sum_i \frac{e_i}{p_i} = 0.76, H = 20}$$



Time	Schedule
0	t_1
1	t_3
2	t_2
3.8	I
4	t_1
…	…
19.8	I
20	Goto $t \bmod (H)$

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

Clock-driven scheduling – 4/8

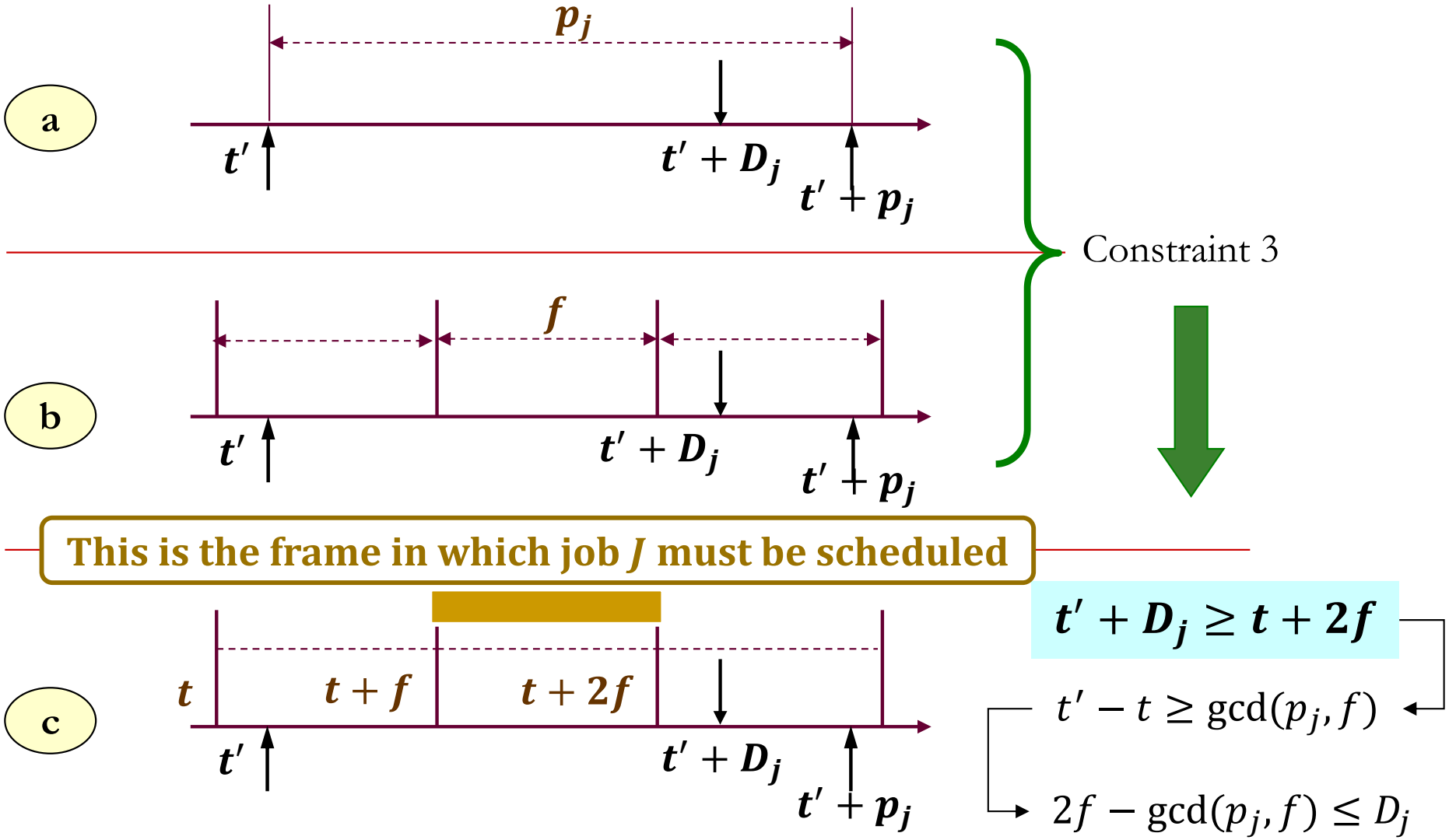
- To keep design complexity low, we should *minimize* the size of the cyclic schedule (table S): how can that be done?
- The scheduling point t_k should occur at regular intervals
  - Each such interval is termed ***minor cycle*** (*frame*) and has duration f
  - We need a (cheaper, more standard) *periodic timer* instead of a (more costly) interval timer
  - Within minor cycles there is no preemption, but a single frame may allow the execution of multiple (run-to-completion) jobs
- For every task τ_i, φ_i must be a non-negative integer multiple of f
  - Forcedly, the first job of every task has its release time set at the start edge of a minor cycle
- To build such a schedule, we must enforce some constraints

Clock-driven scheduling – 5/8

- **Constraint 1**: Every job J must complete within f
  - $\boldsymbol{f \geq max_{i=\{1,..n\}}(e_i)}$ so that *overruns* can be detected
- **Constraint 2**: f must be integer divisor of hyperperiod
  - $\boldsymbol{H : H = Nf}$ where $\boldsymbol{N \epsilon \mathbb{N}}$
  - It suffices that f be an integer divisor of at least one task period p_i
  - The hyperperiod beginning at minor cycle kf for $k = 0,\ \ N-1, 2N-1$ 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 and complete in that one frame
  - This can be expressed as: $\boldsymbol{2f - \gcd(p_i, f) \leq D_i}$ for every task τ_i

Understanding constraint 3

Finding the worst *allowable* offset between t' (the job's release time) and t (the edge of the frame)



Example

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

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

$f = 2 : 4 - \gcd(4,2) \leq 4$ **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 – 6/8

- 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 the jobs of for some task(s) τ_i by ***slicing*** their (WCET) e_i^w into fragments artificially sized to yield a "good" f

Clock-driven scheduling – 7/8

- To construct a cyclic schedule, we must make three design decisions
  - Fix an f
  - Slice (the large) jobs
  - Assign (jobs and) slices to minor cycles
- Sadly, these decisions are very tightly coupled
  - This defect makes cyclic scheduling *very* fragile to any change in system parameters

Clock-driven scheduling – 8/8

```
Input: stored schedule S[k], k in 0 .. F − 1
CYCLIC_EXECUTIVE ::
  t := 0; k := 0;
  do forever
    sleep until clock interrupt at time t × f;
    currentBlock := S[k];
    t := t + 1; k := t mod F;
    if last job not completed then take action;
    end if;
    execute all job slices in currentBlock;
    while aperiodic job queue not empty do
      execute aperiodic job at top of queue;
    end do; // the clock interrupt may occur!
  end do;
end SCHEDULER
```

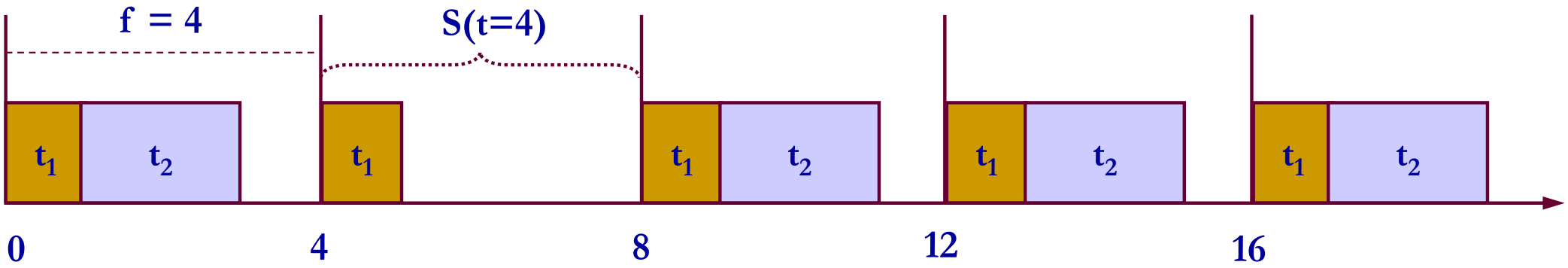
Example (slicing) – 1/2

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

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

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?

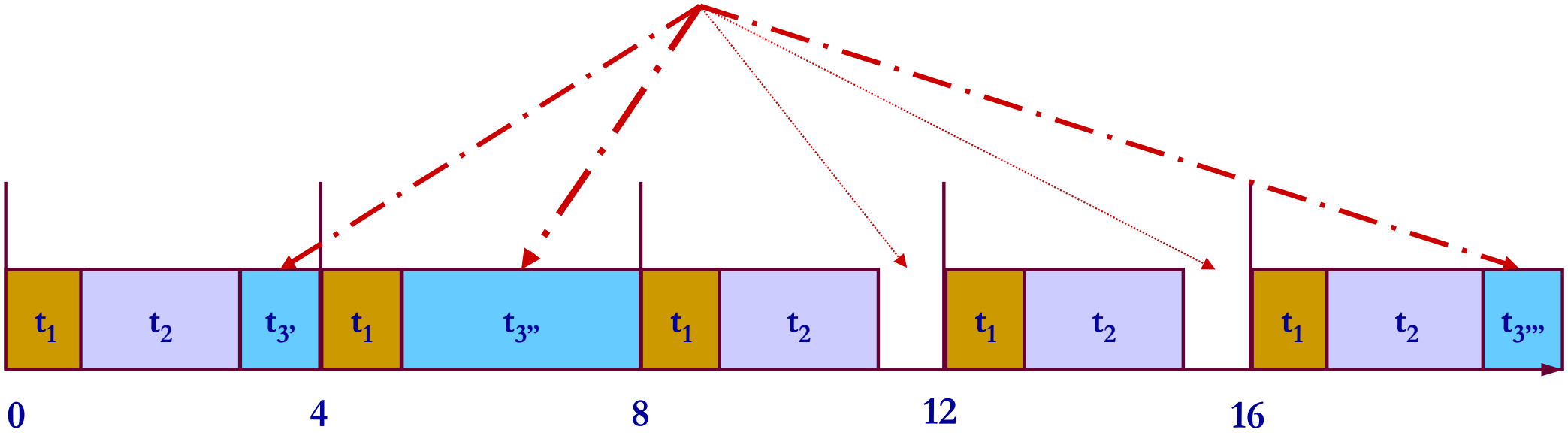


We first look at the schedule with $f = 4$ and $F = \left(\frac{H}{f}\right) = 5$

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)

Design issues – 1/3

- Completing a job much ahead of its deadline may be of little or no use!
- Spare time in time slices should be given to *aperiodic jobs*, to allow the system to produce more (piece-rate) value added
- This can be achieved with ***slack stealing***, which allows aperiodic jobs to execute *in preference* to periodic jobs when possible
  - Minor cycles may have *slack* time, not used for scheduling periodic jobs, that may be devolved to aperiodic jobs
  - The slack of minor cycles is a *statically* known at design time
- A cyclic scheduler does slack stealing if it assigns the available slack time *at the beginning* of every minor cycle (instead of at the end)
  - The system becomes *more reactive* (which is good)
  - A fine-grained interval timer signals end of slack time for each minor cycle

Design issues – 2/3

- What can we do to handle ***overruns*** ?
  - Halt the job found running at the start of the new minor cycle
    - But that job may not be the one that overrun!
    - Even if it was, stopping it would only serve a useful purpose if producing a late result had no residual *utility*
  - Defer halting until the job has completed all its "critical actions"
    - To avoid the risk that a premature halt may leave the system in an inconsistent state
  - Allow the job some extra time by delaying the start of the next minor cycle
    - Plausible if producing a late result still had *utility*

Design issues – 3/3

- What can we do to handle ***mode changes***?
  - A mode change is when the system incurs some reconfiguration of its function and workload parameters
- Two main axes of design decisions
  - With or without deadline during the transition
  - With or without overlap between outgoing and incoming operation modes

Overall evaluation

- **Pros**
  - Comparatively simple design
  - Simple and robust implementation
  - Complete and cost-effective verification
- **Cons**
  - Very fragile design
    - Construction of the schedule table is a NP-hard problem
    - High extent of undesirable architectural coupling
  - All parameters must be fixed a priori at the start of design
    - Choices may be made arbitrarily to satisfy the constraints on *f*
    - Totally inapt for sporadic jobs

Priority-driven scheduling

- Base principle
  - Every job is assigned a priority
  - The job with the highest priority is dispatched to execution
- Two implementation decisions
  - When jobs' priority should change
  - When dispatching should occur
- ***Dynamic-priority scheduling***
  - Distinct jobs of the same task may have *distinct* priorities
    - EDF: the job priority is *fixed* at release, but changes across releases
    - LLF: the job priority may change at every dispatching point
- ***Static-priority scheduling***
  - All jobs of the same task have *one and the same* priority

Fixed-priority scheduling (FPS)

- Two main strategies for priority assignment
  - This is all we need to determine FPS
- ***Rate monotonic*** (when deadlines are implicit)
  - A task with *faster rate* (lower period) takes precedence
  - Optimal *static-per-task* priority assignment under preemptive scheduling and implicit deadlines
  - The consequent scheduling is called **RMS**
- ***Deadline monotonic*** (when deadlines are constrained)
  - A task with *higher urgency* (shorter relative deadline) goes first
  - Optimal (static-per-task) for constrained deadlines
- In effect, both algorithms are deadline-based ☺

Preliminary observations

- Priority-driven scheduling algorithms that disregard job urgency (deadline) perform *poorly*
- The WCET is *not* a factor of consequence for priority assignment
  - Weighed round-robin scheduling is "utilization-monotonic", but is unfit for real-time systems

- ***Schedulable utilization*** is a good metric to compare the performance of scheduling algorithms
  - A scheduling algorithm S can produce a feasible schedule for a task set T on a single processor if and only if $U(T)$ does not exceed the schedulable utilization of S

Utilization-based tests – 1/2

- **Theorem** [Liu & Layland: 1973]
  For single processors and implicit deadlines, EDF's *schedulable utilization* $U(n) = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$ is *always* feasible
  - A *necessary and sufficient* (exact) test condition for EDF
- For RMS, the equivalent test is

$$U(n) = \sum_{i=1}^{n} \frac{e_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

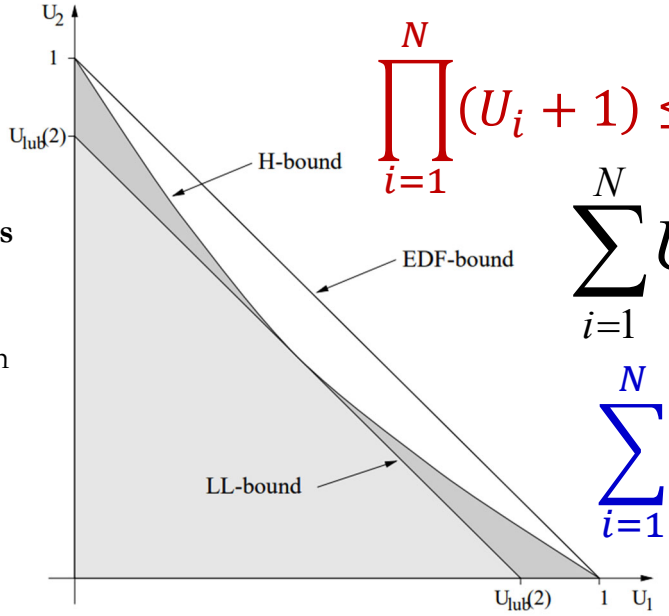
$$\lim_{n \to \infty} n\left(2^{\frac{1}{n}} - 1\right) = \ln 2 \sim 0.69$$

  - A *sufficient but not necessary* (pessimistic) test for RMS
- The schedulable utilization of RMS is *less* than for EDF

Utilization-based tests – 2/2

- The *hyperbolic bound* [Bini & Buttazzo, 2001] improves the Liu & Layland utilization test for RMS
  - It helps prove that RMS achieves 100% utilization when *all pairs* of periods in the task set are in harmonic relation

Examples of feasibility regions
Plot in an $n = 2$ U-space, where each point $U = \{U_1, U_2, \ldots, U_n\}$ represents a periodic task set with utilization U_i



$$\prod_{i=1}^{N}(U_i + 1) \leq 2$$

$$\sum_{i=1}^{N} U_i \leq 1$$

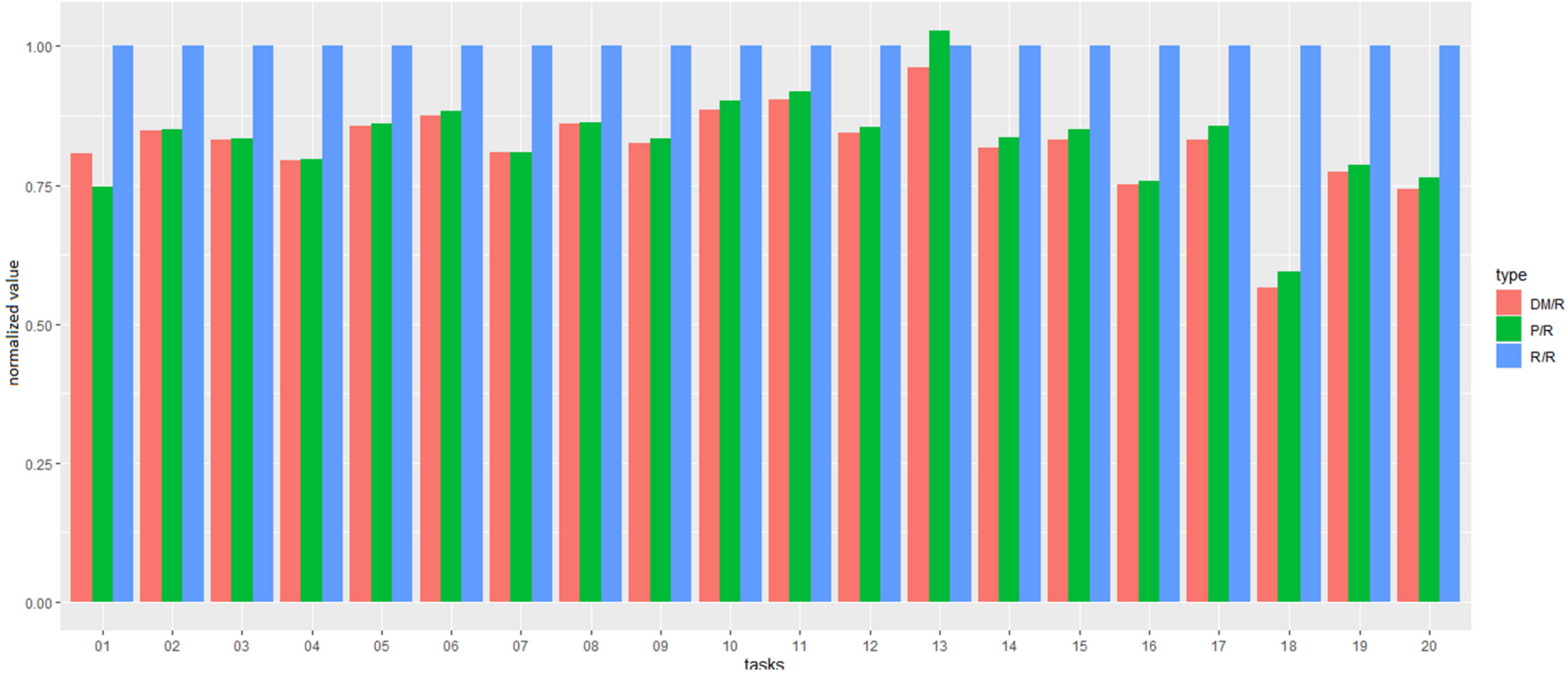
$$\sum_{i=1}^{N} U_i \leq N\left(2^{\frac{1}{N}} - 1\right) \rightarrow \leq ln(2)$$

Appraising feasibility

- Schedulable utilization alone is *not* a sufficient criterion: we must also consider *predictability*
  - Recall its intuition, given in Section 1
- On ***transient overload***, the behavior of static-priority scheduling can be determined a-priori and is reasonable
  - The overrun of any job of a given task τ does not harm the tasks with higher priority than τ
- Under transient overload, EDF acts "strange" (locally)
  - A job that missed its deadline is *more urgent* than a job with a deadline in the future: one lateness may cause many more!

Overload situations – 1/5

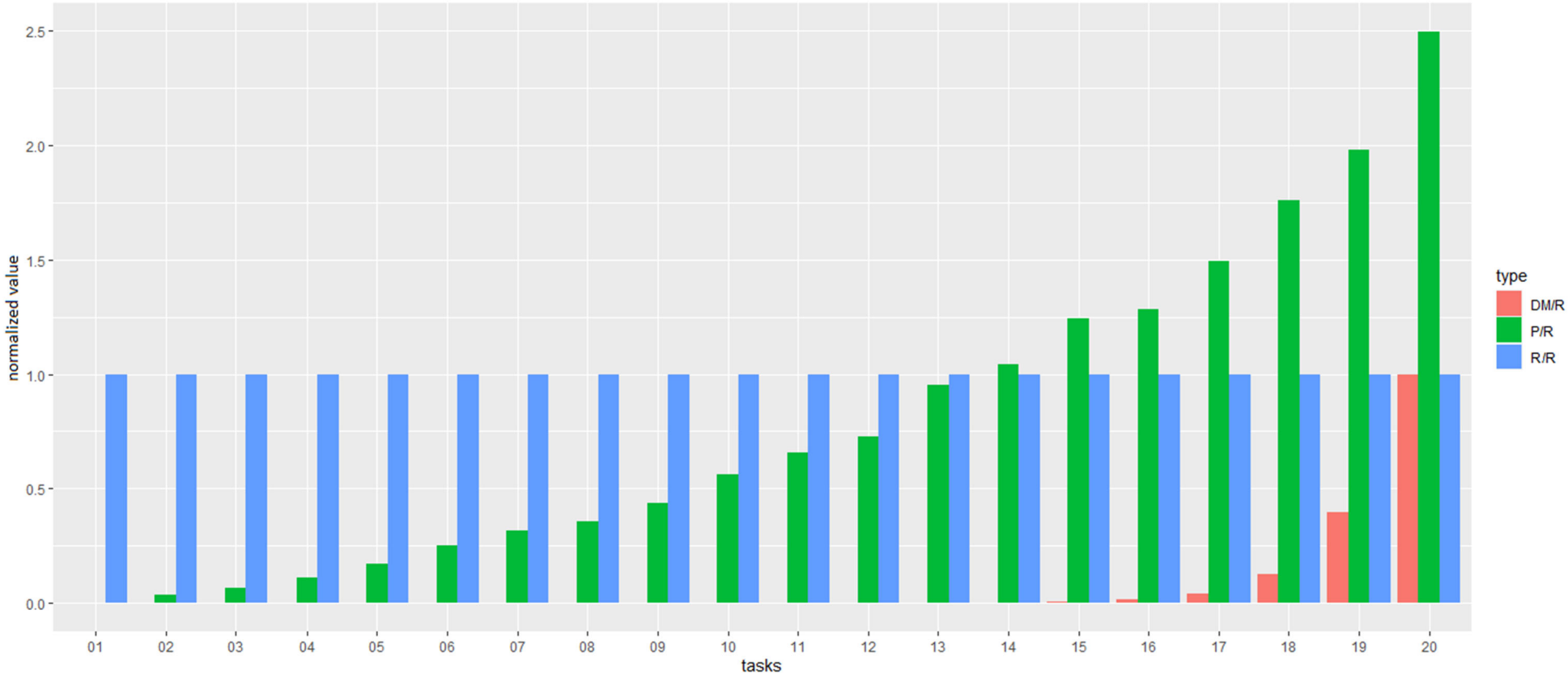
Deadline miss and preemption count ratio over normalized run count (EDF, $U > 1$)



Legend: DM/R (deadline misses over releases); P/R (preemptions over releases); R (release; run)

Overload situations – 2/5

Deadline miss and preemption count ratio over normalized run count (FPS, $U > 1$)



Legend: DM/R (deadline misses over releases); P/R (preemptions over releases); R (release; run)

Overload situations – 3/5

An interesting property of EDF during permanent overloads is that it automatically performs a period rescaling, and tasks start behaving as they were executing at a lower rate. This property has been proved by Cervin et al. (2002) and it is formally stated in the following theorem.

Theorem 1 [Cervin]. *Assume a set of n periodic tasks, where each task is described by a fixed period T_i, a fixed execution time C_i, a relative deadline D_i, and a release offset Φ_i. If $U > 1$ and tasks are scheduled by EDF, then, in stationarity, the average period $\bar{T}_i$ of each task τ_i is given by $\bar{T}_i = T_i U$.*

Real-Time Systems, 29, 5–26, 2005

- EDF's throughput decreases by period rescaling
- FPS's throughput decreases by discarding lower-priority jobs

Overload situations – 4/5

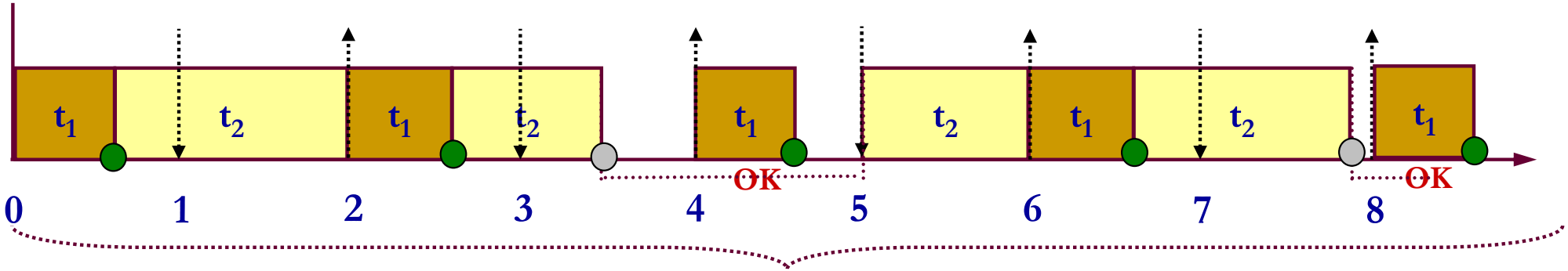
$$(\varphi_i, p_i, e_i, D_i)$$

$$\boldsymbol{T = \{\tau_1 = (0, 2, 0.6, 1), \tau_2 = (0, 5, 2.3, 5)\}}$$

$$\textbf{\textit{Density}}\ \boldsymbol{\Delta(T) = \frac{e_1}{D_1} + \frac{e_2}{D_2} = 1.06 > 1}$$

$$\textbf{\textit{Utilization}}\ \boldsymbol{U(T) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 0.76 < 1}$$

What happens to T under EDF?



$$H = 10$$

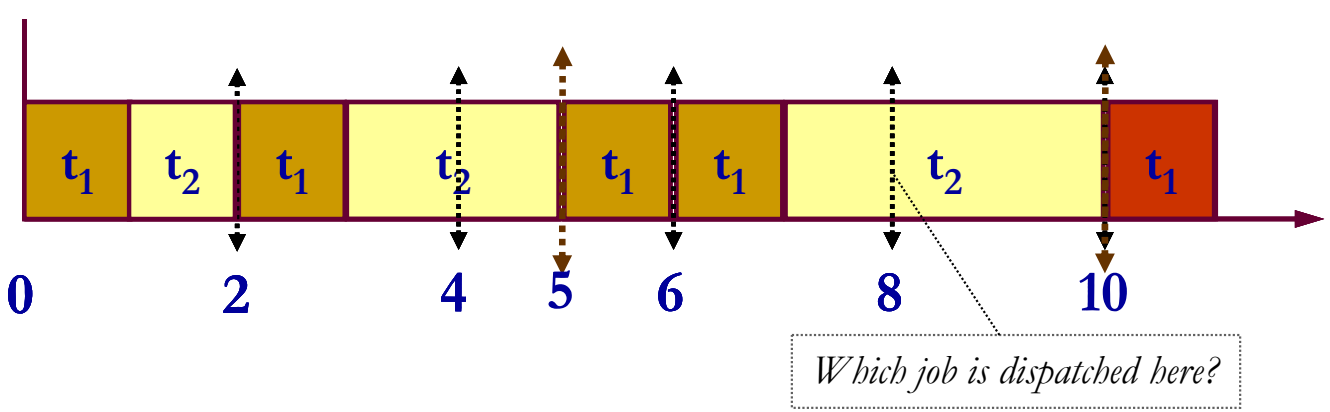
The exact utilization-based test tells us that T is feasible under EDF
(We don't need to draw its timeline to tell that!)

Overload situations – 5/5

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

T = {t_1= (0, 2, 1, 2), t_2= (0, 5, 3, 5)} ⇨ $U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.1$

T has *no* feasible schedule: what job suffers most under EDF?



T = {t_1= (0, 2, 0.8, 2), t_2= (0, 5, 3.5, 5)} ⇨ $U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.1$

T has *no* feasible schedule: what job suffers most under EDF?

What about

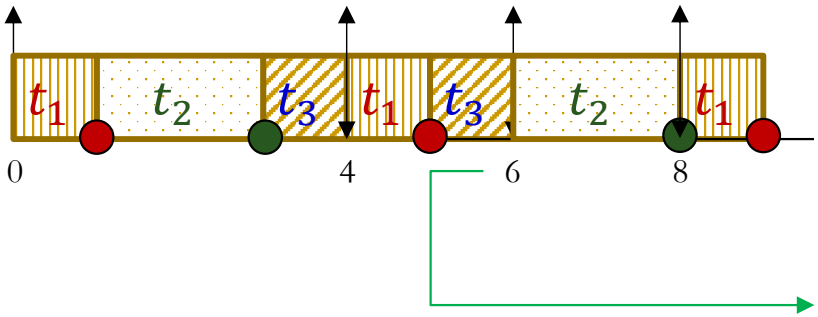
T = {t1 = (0, 2, 0.8, 2), t2 = (0, 5, 4, 5)} with $U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.2$?

Preemption count – 1/2

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

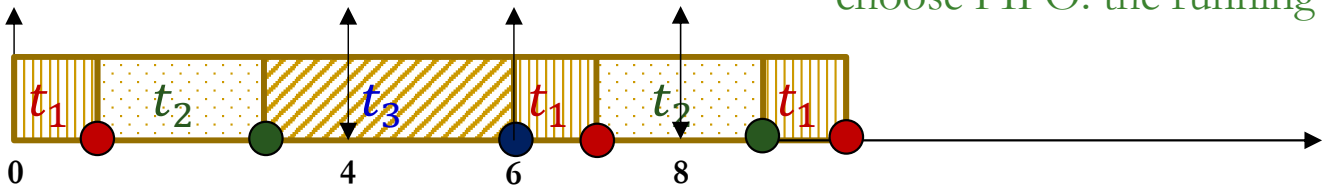
$$T = \{t_1 = (0,4,1,4), t_2 = (0,6,2,6), t_3 = (0,8,3,8)\}, U = \frac{23}{24}, \mathrm{H} = 24$$

With FPS and rate-monotonic priority assignment



With FPS, at time 4, with
t_3's absolute deadline = 8, priority = low
t_1's absolute deadline = 8, priority = high
t_1 preempts t_3
And, at time 6, with
t_2's absolute deadline = 12, priority = medium
t_2 preempts t_3, which misses its deadline
(Remember: in case of equal priorities, EDF and FPS choose FIFO: the running job continues to run)

With EDF



EDF may incur *less* preemptions than FPS

Preemption count – 2/2

Experiment	Run time	Mean preemptions FPS	Mean preemptions EDF	Min $\frac{P_{EDF}-P_{FPS}}{P_{FPS}}$	Max $\frac{P_{EDF}-P_{FPS}}{P_{FPS}}$
Fully-Harmonic	Hyperperiod	32,34	32,19	-0.5714	0.8571
Semi-Harmonic	Hyperperiod	4.265	4.255	-0.0282	0.1788
1.0 < U < 1.0004	Hyperperiod * U	23.385	41.171	-1.3866	-0.3089

Mean across task sets

Back to FPS: critical instant – 1/2

- Feasibility and schedulability tests must consider the ***worst case***, WC, for all tasks
  - The WC for task τ_i occurs when the worst possible relation holds between its own 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 WC
  - 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 – 2/2

- **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 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 $\sum$ component captures the ***interference*** that any job j of task τ_i incurs from jobs of 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, which occurs at $\varphi_i + \omega_{i,j}$
- When φ is 0 for all jobs considered, all tasks are *in phase* and the equation captures the *absolute worst case* for task τ_i

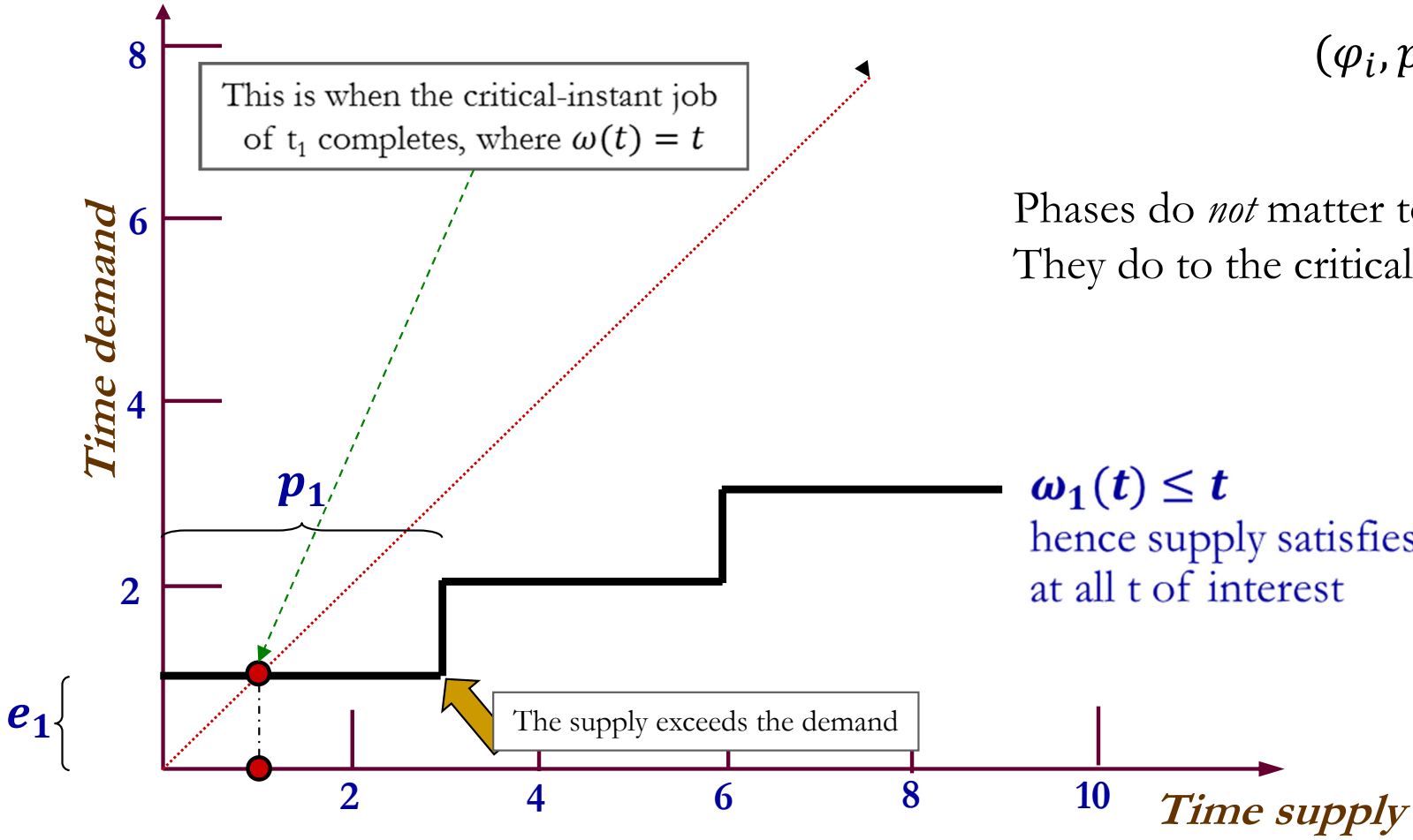
Time-demand analysis – 1/6

- ***Time Demand Analysis***, TDA, studies ω as a function of time, $\omega(t)$
  - As long as $\omega(t) \leq t$ *for some (selected)* t for the job of interest, the supply satisfies the demand, hence the job can complete in time

- **Theorem** [Lehoczky, Sha, Ding: 1989]
  $\omega(t) \leq t$ is an *exact feasibility test* for FPS
  - 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 – 2/6

$$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}, U = 0.82$$



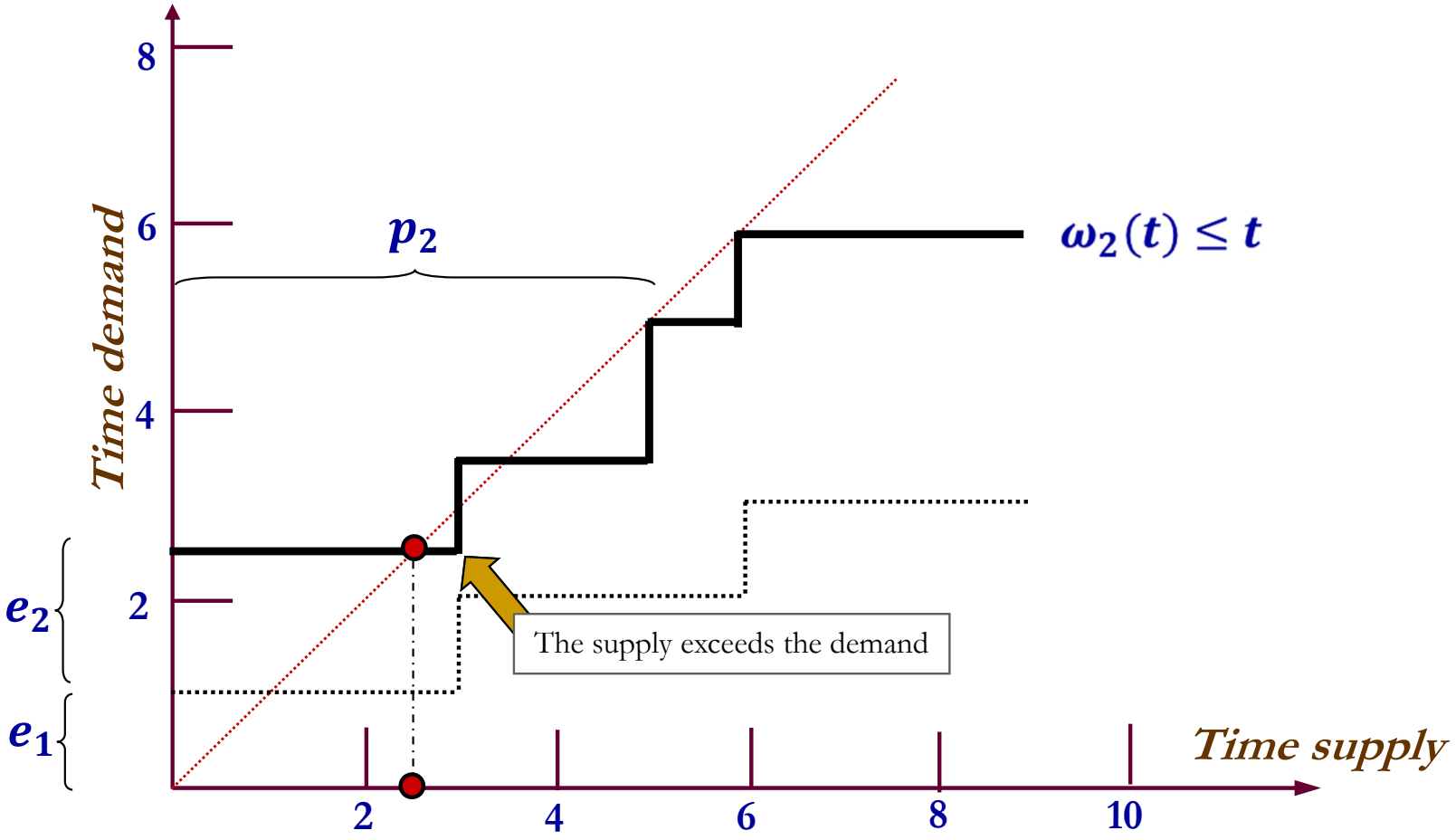
$(\varphi_i, p_i, e_i, D_i)$

Phases do *not* matter to TDA
They do to the critical instant!

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

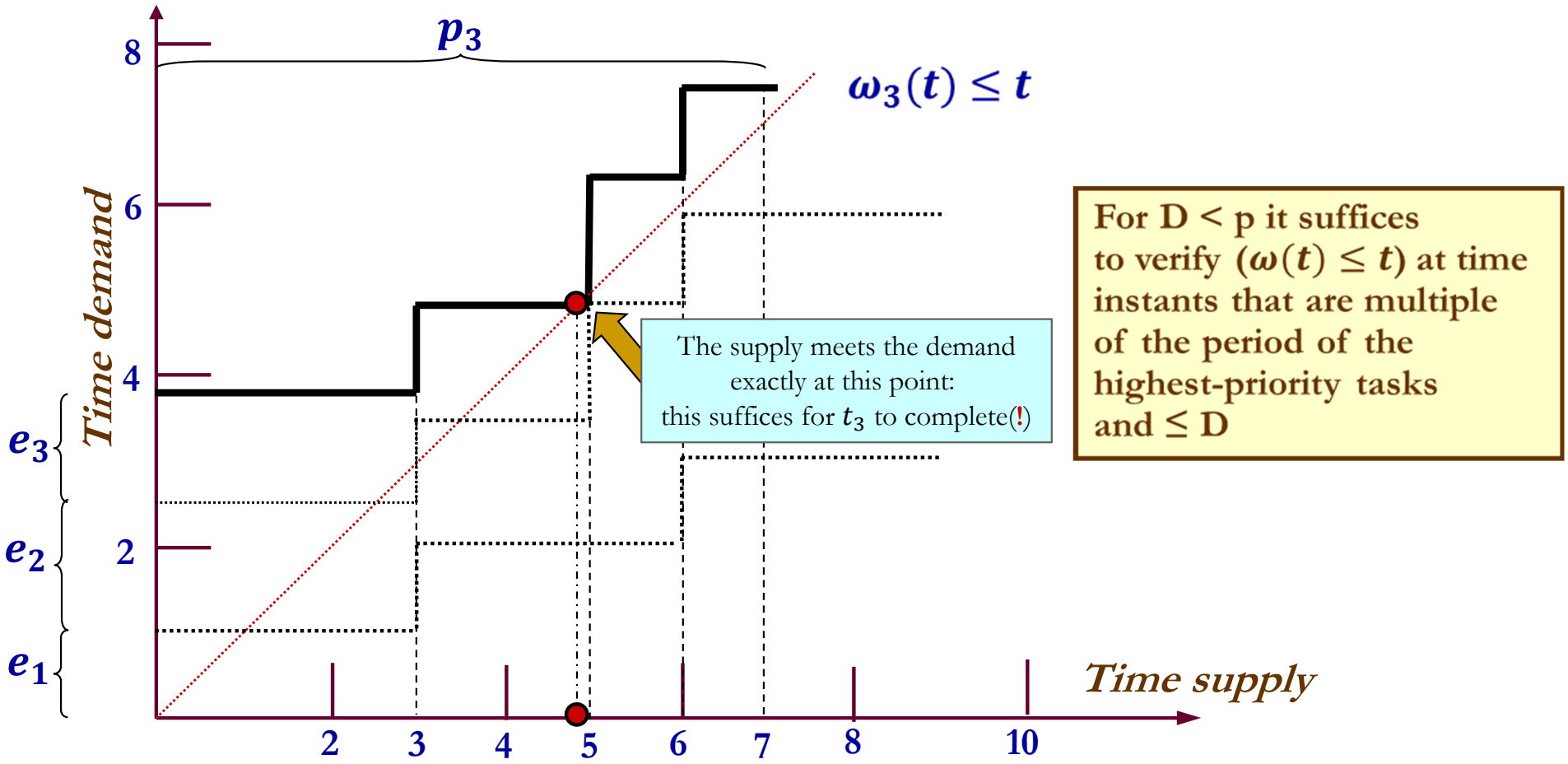
Time demand analysis – 3/6

$$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}, U = 0.82$$



Time demand analysis – 4/6

$T = \{t_1 = (-, 3, 1, 3), t_2 = (-, 5, 1.5, 5), t_3 = (-, 7, 1.25, 7)\}, U = 0.82$



Time demand analysis – 5/6

- We can use TDA to capture the *response time* of tasks and then use the critical instant notion to see that

The smallest value t that satisfies

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

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 – 6/6

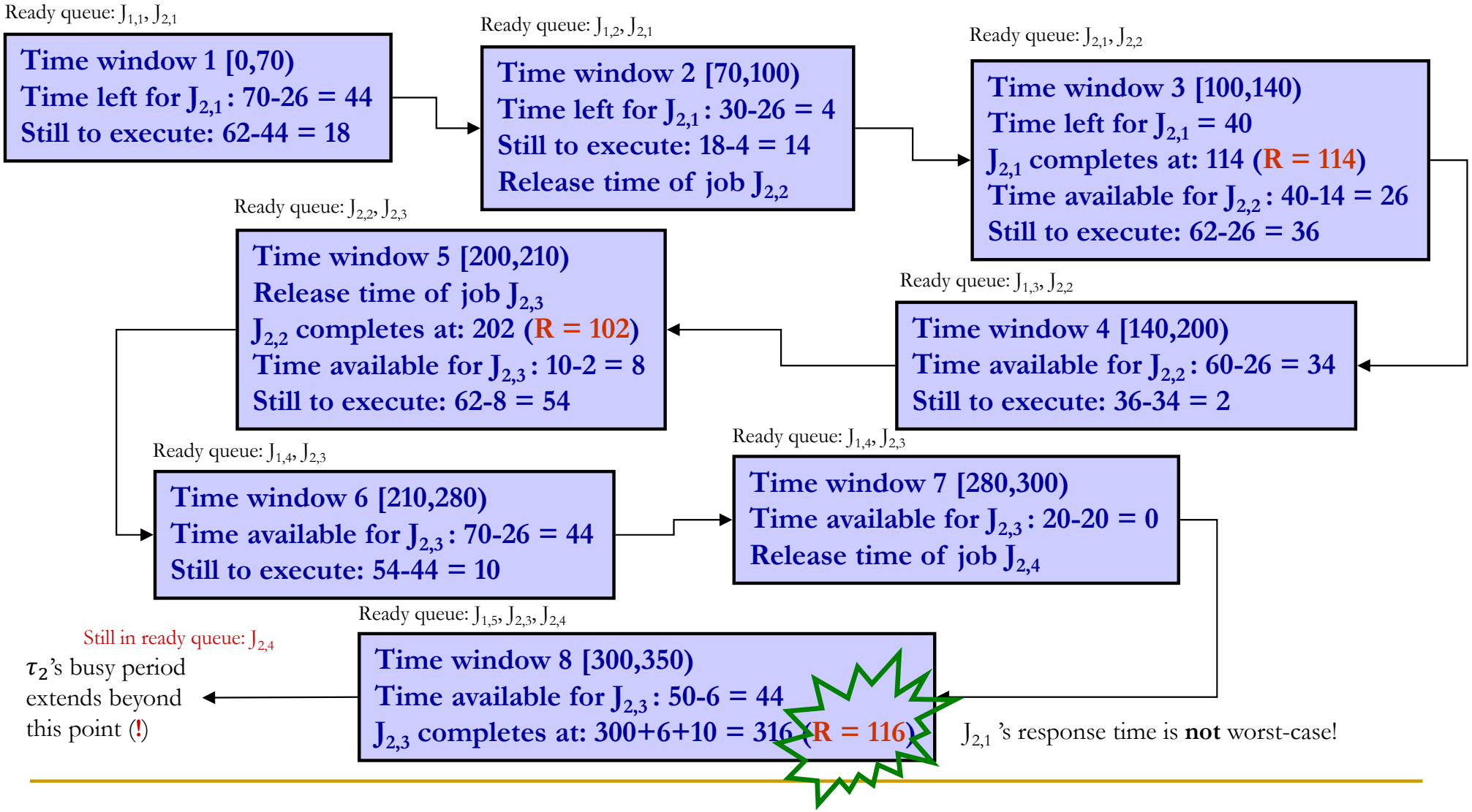
- **Theorem** [Lehoczky, Sha, Strosnider, Tokuda: 1991]
  When $D > p$, the first job of task τ_i may *not* be the one that incurs the worst-case response time

- 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 $\geq i$, with release time in (t_0, t), and response time falling within t
  - The release time in (t_0, t) captures all backlog of interfering jobs
  - The response time of all jobs falling within t ensures that the busy period extends to their completion

Example

τ_1 = {-, 70, 26, 70}, τ_2 = {-, 100, 62, 120} $(\varphi_i, p_i, e_i, D_i)$

Let's look at the level-2 busy period

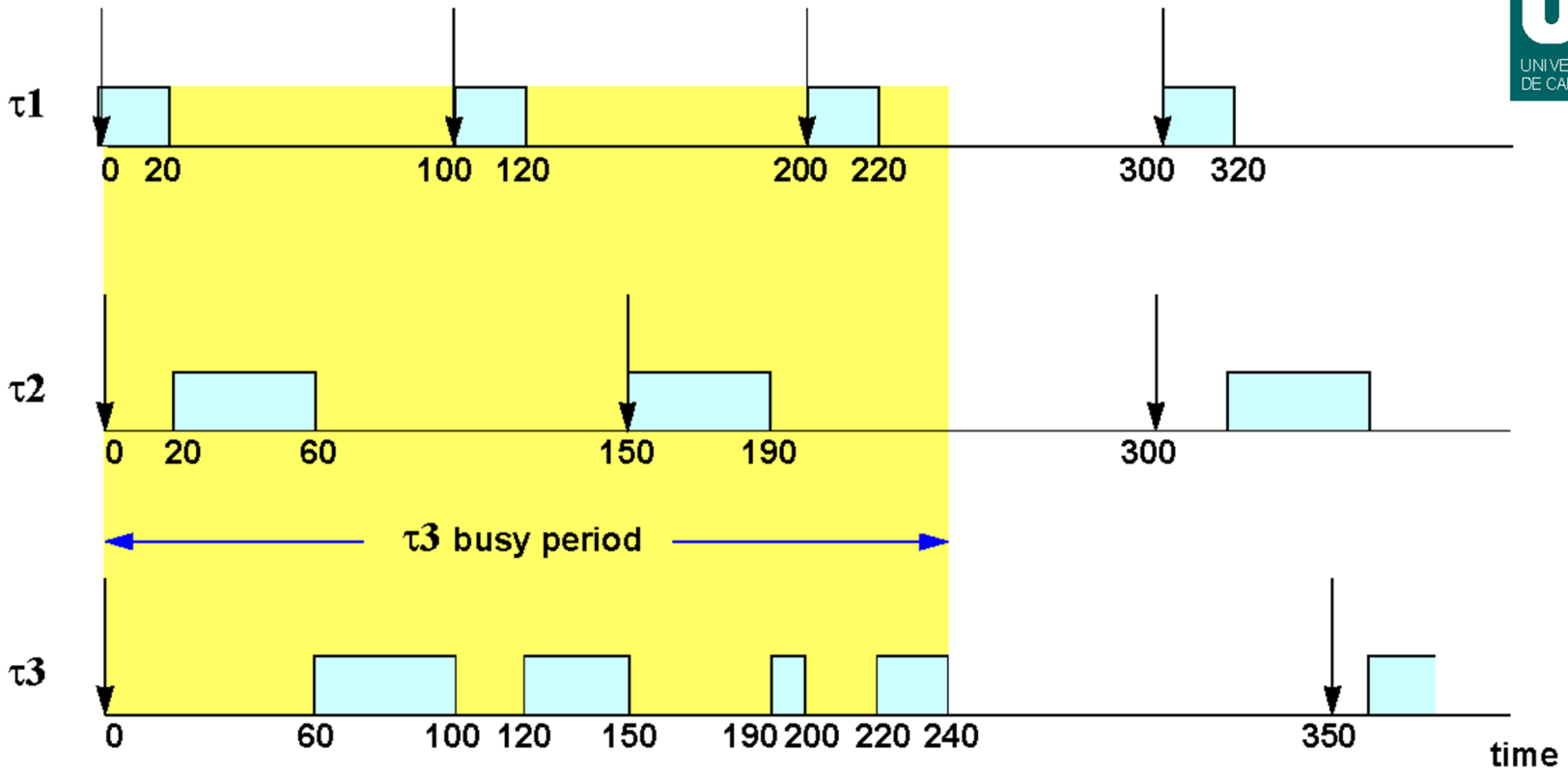


Level-i busy period

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

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

but its width is obviously shorter!



Demand bound analysis (EDF)

- For $\boldsymbol{df}$, the EDF *demand function* and time t_i, an *exact* test for a task set T under EDF is:

$$\forall t_1, t_2 : t_2 > t_i, \boldsymbol{df}(t_1, t_2) \leq t_2 - t_1$$

- For periodic tasks with no offsets and $U \leq 1$, it holds that:

$$\boldsymbol{df}(t_1, t_2) \leq \boldsymbol{df}(0, t_2 - t_1)$$

- The ***demand bound function*** helps generalize the test

$$\boldsymbol{dbf}(L) = \max_t \left(df(t, t + L) \right) = df(0, L), L > 0$$

- **Theorem** [Baruah, Howell, Rosier: 1990] Exact test for EDF:

$$\forall L \in D(T), \boldsymbol{dbf}(L) \leq L, U < 1$$

  - $D(T)$ is the set of deadlines for T in $[0, L_m]$, $L_m = min(L_a, L_b)$, $L_a = max\left\{D_1, \dots, D_n, \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1-U}\right\}$, $L_b =$ first idle time in $T's$ busy period

Summary

- Initial survey of scheduling approaches
- Important definitions and criteria
- Detail discussion and evaluation of main scheduling algorithms
- Initial considerations on feasibility analysis techniques

Selected readings

- T. Baker, A. Shaw
  The cyclic executive model and Ada
  DOI: 10.1109/REAL.**1988**.51108
- C.L. Liu, J.W. Layland
  Scheduling algorithms for multiprogramming in a hard-real-time environment
  DOI: 10.1145/321738.321743 (**1973**)
- D. Perale, T. Vardanega
  Removing bias from the judgment day: A Ravenscar-based toolbox for quantitative comparison of EDF-to-RM uniprocessor scheduling
  DOI: 10.1016/j.sysarc.**2021**.102236