2. Scheduling basics

Where we commence our familiarization with real-time scheduling, that is, the algorithms that decide how the CPU is assigned to the jobs that contend for it

Common approaches /1

- ***Clock-driven (time-driven) scheduling***
  - Scheduling decisions are made beforehand (at system design) and actuated at fixed time instants during execution
    - Such time instants occur at intervals signaled by clock via interrupts
  - The scheduler dispatches to execution 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
    - Consequently, this 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

- ***Weighted round-robin scheduling***
  - With basic round-robin (which requires preemption)
    - All ready jobs are placed in a FIFO queue
    - CPU time is quantized, i.e., assigned in slices
    - The job at head of queue is dispatched to execution for one quantum
      - If not complete by end of quantum, it goes to tail of queue
      - All jobs in queue are given one quantum per round
    - Not good for jobs with precedence relations, but fine for producer-consumer pipelines that proceed in continual 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 (full traversal of the queue)
      - One full round corresponds to $\sum_i \boldsymbol{\omega_i}$ progress for the ready jobs

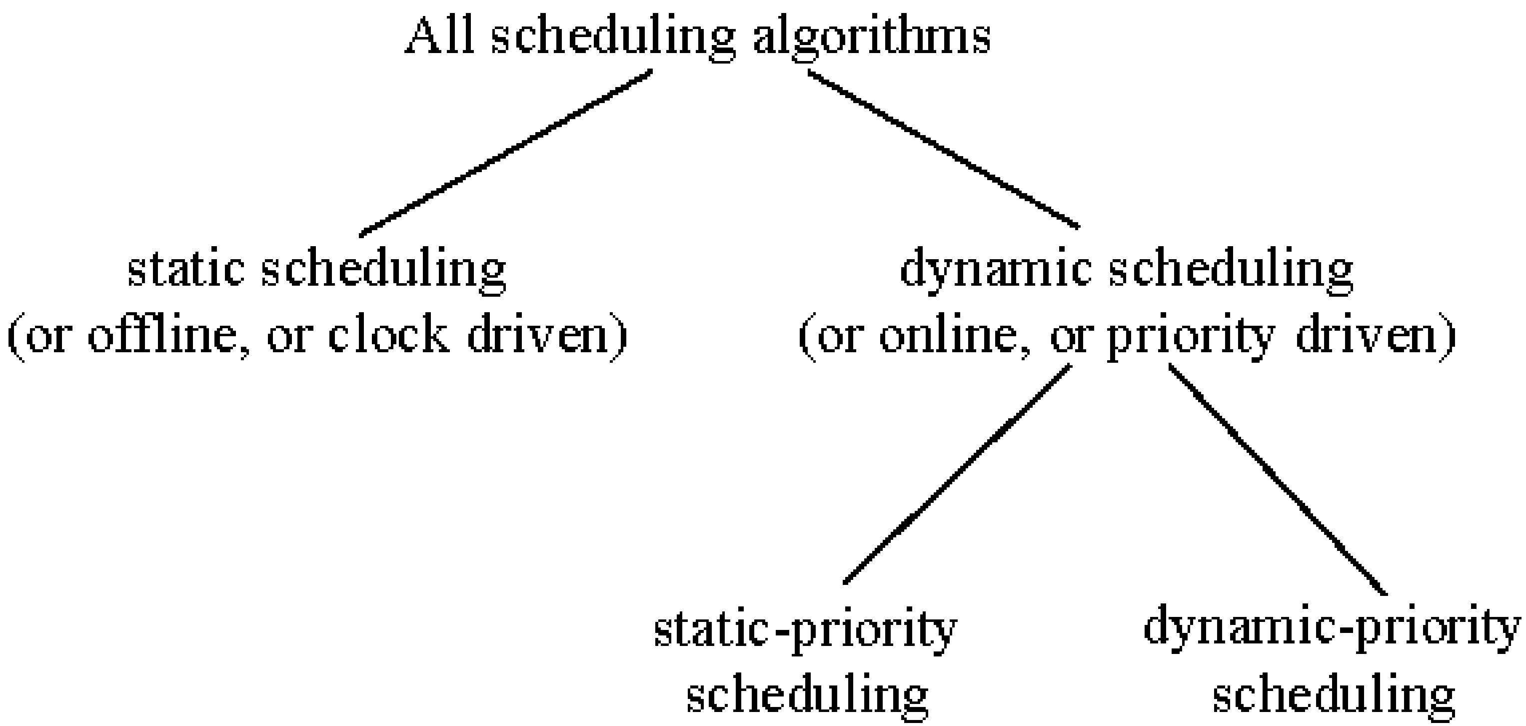
Common approaches /3

- ***Priority-driven (event-driven) scheduling***
  - This class of algorithms is *greedy*
    - Never leave available processing resources unused if they are wanted
    - An available resource may stay unused only if no job ready to use it
      - *Clairvoyant* schedulers may prefer deferring assignment of CPU to improve response time
    - 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 events are called ***dispatching points***
    - Scheduling decisions are made online, based on present knowledge
    - Dispatching employs ***preemption***

Preemption vs. non preemption

- Can we compare the performance of preemptive scheduling against non-preemptive scheduling?
  - There is no single response that be valid in general
  - When all jobs have same release time, and preemption overhead is negligible (**!?**), then preemptive scheduling is *provably better*
- 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 …
  - We do know that, for 2 CPUs, the minimum makespan for non-preemptive scheduling is *never worse* than 4/3 of that for preemptive

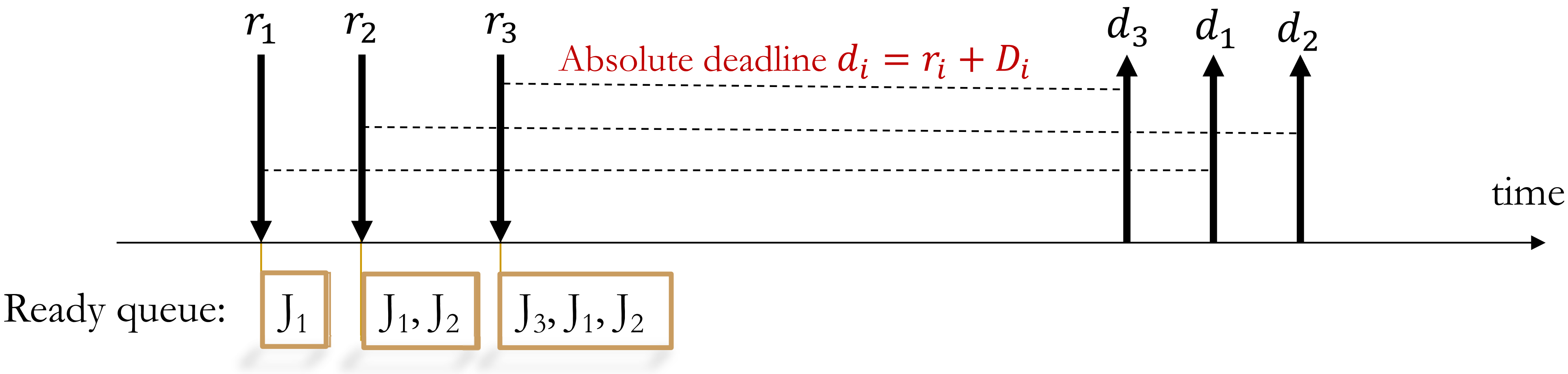
Classification of Scheduling Algorithms



Jim Anderson | Real-Time Systems | Introduction - 30

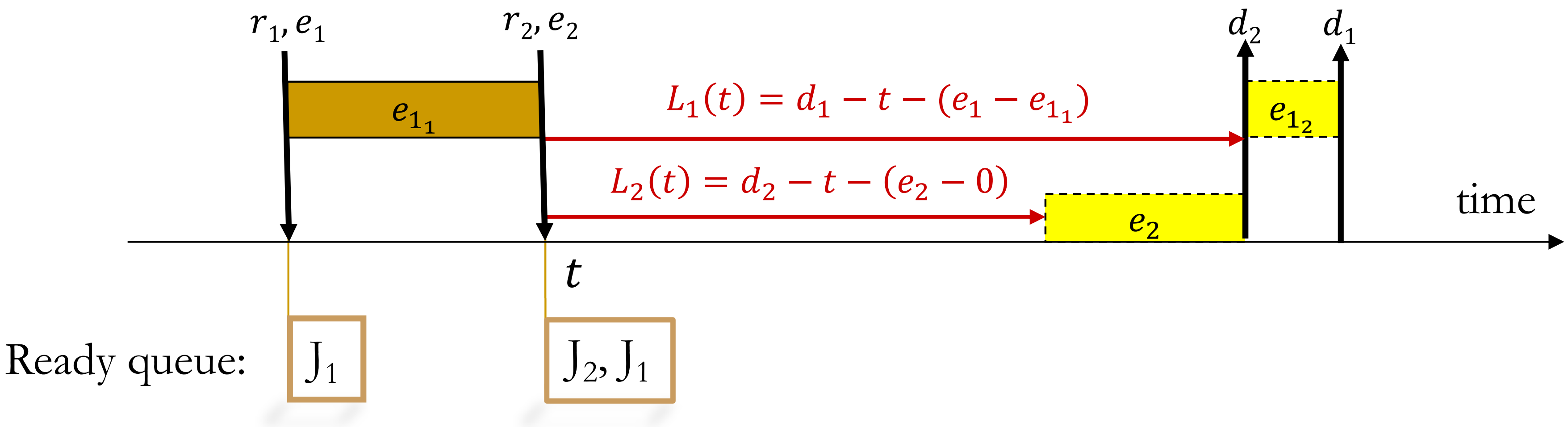
Ways to optimality /1

- Priorities assigned *dynamically* to reflect *absolute deadlines*
  - Ready queue reordering occurs on job release
- [Liu & Layland: 1973] ***Earliest Deadline First*** (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)



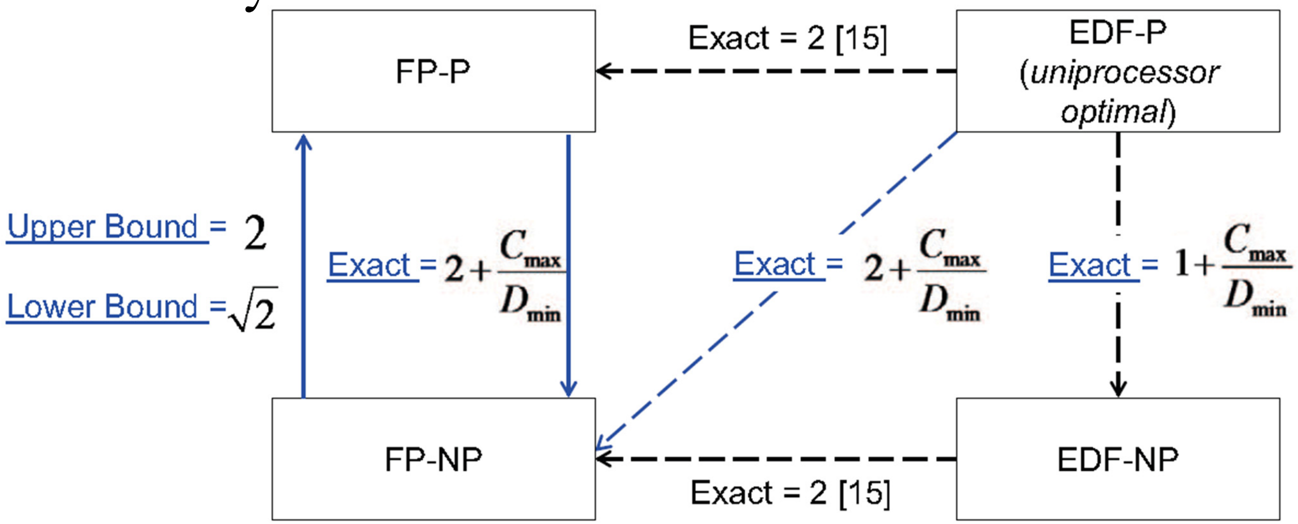
Ways to optimality /2

- Priorities assigned dynamically according to *laxity* $L(t)$
  - $L_i(t) = d_i - t - Y_i(t)$, 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
  - Ready queue reordering occurs on job release and job completion
  - Jobs' priority, $L(t)$, varies with t: more dynamic and costly than EDF
- [Liu & Layland: 1973] ***Least Laxity First*** (LLF) scheduling is ***optimal*** under the same hypotheses as for EDF optimality



Optimality and sub-optimality

- The *processor speed-up factor* determines the increase in processor speed that a scheduling algorithm would require to equalize an *optimal* algorithm of the same class for any task set



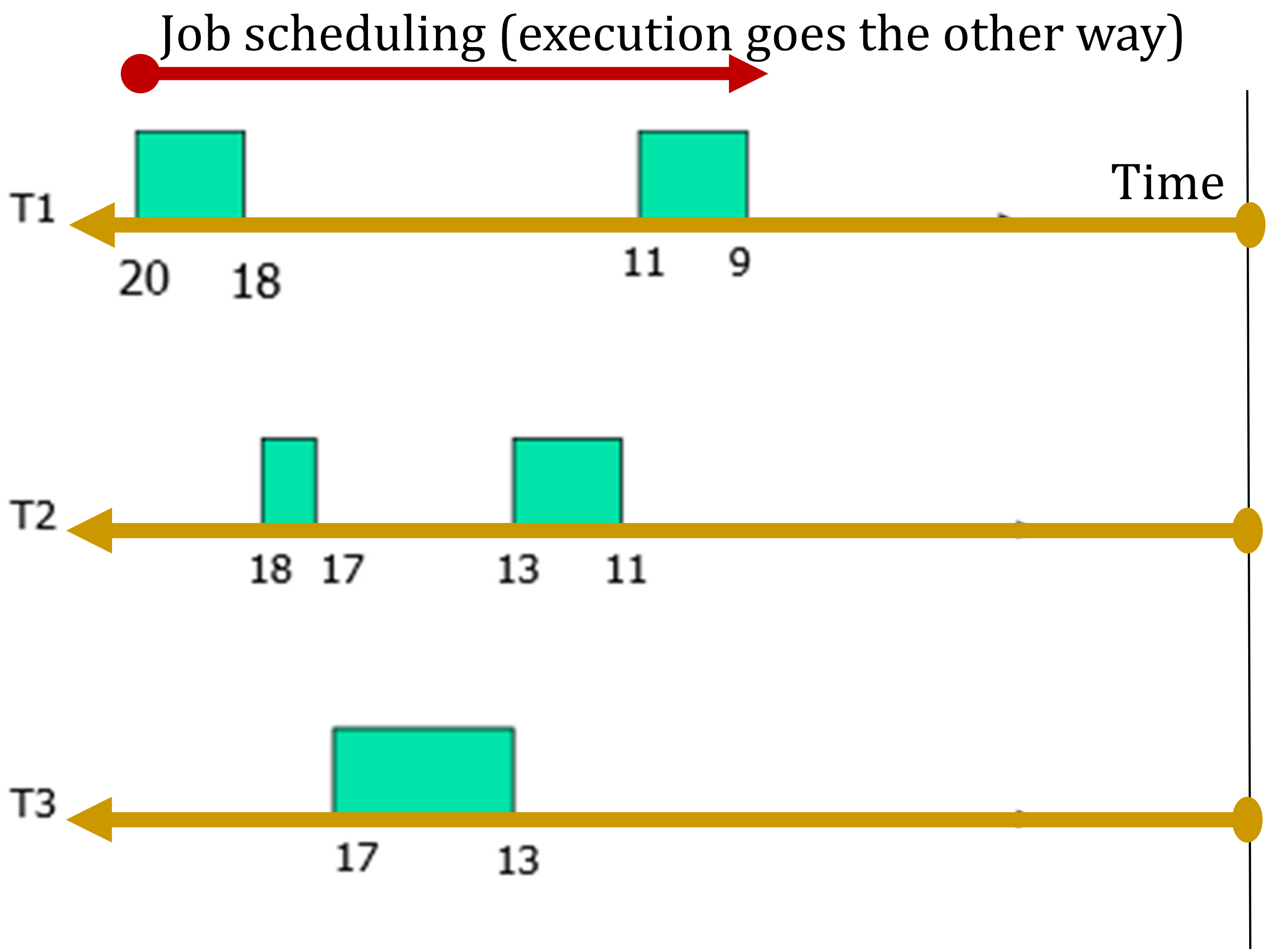
Davis et al., "Quantifying the Exact Sub-Optimality of Non-Preemptive Scheduling", RTSS 2015

Ways to optimality /3

- If one's goal were solely that jobs meet their deadlines, there would be no value in having jobs complete any earlier
  - The ***Latest Release Time*** (LRT) algorithm – the converse of EDF – follows this logic, scheduling jobs *backward* from the latest deadline, treating deadlines as release times and release times as deadlines
    - LRT is *not* greedy: it may leave the CPU unused with ready tasks
- The wisdom of this algorithm is the knowledge that greedy scheduling algorithms may cause jobs to suffer larger interference

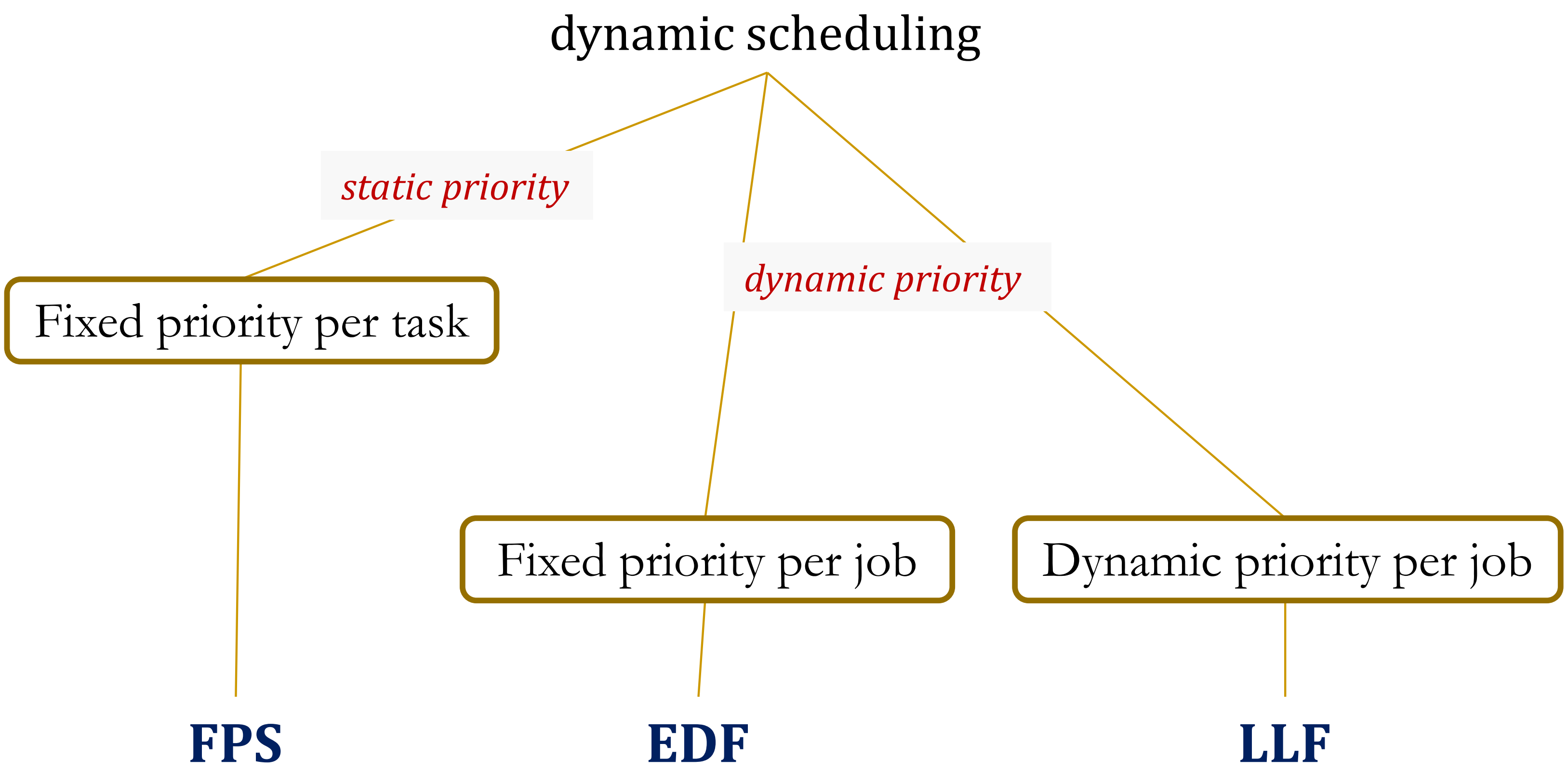
Latest Release Time scheduling

	T_1	T_2	T_3
φ_i	0	11	12
C_i	4	3	4
d_i (absolute)	20	18	17



LRT needs preemption and off line decisions

Taxonomy of dynamic scheduling



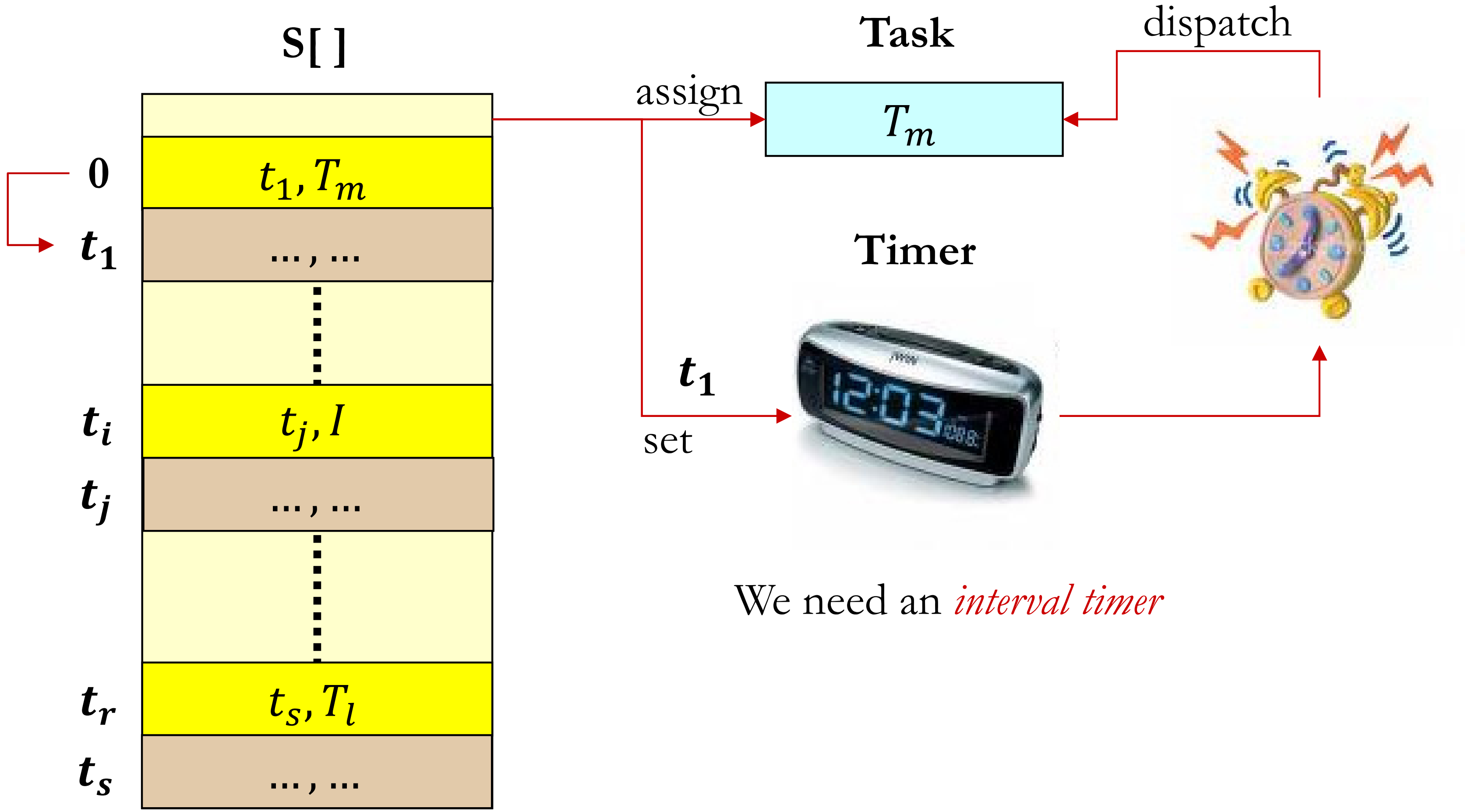
Clock-driven scheduling /1

- ***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

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

Clock-driven scheduling /3



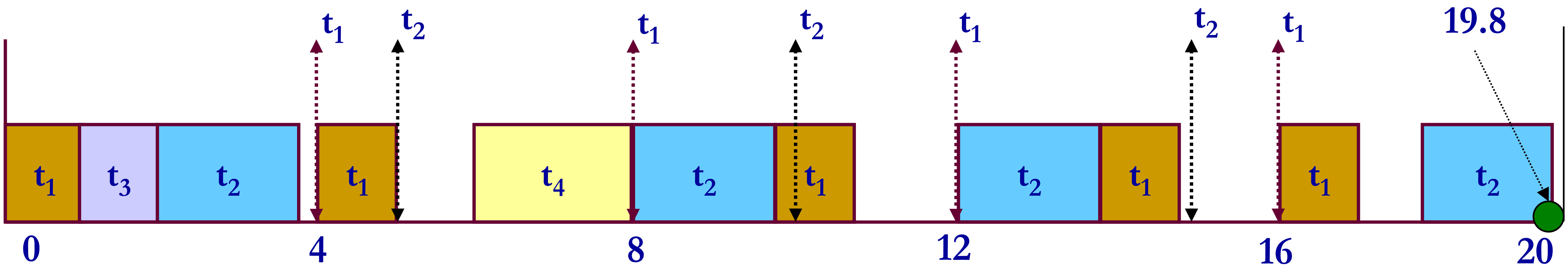
We need an *interval timer*

Where the t_j values need *not* be equally spaced

Example

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

$$J = \{t_1 = (0,4,1,4), t_2 = (0,5,1.8,5), t_3 = (0,20,1,20), t_4 = (0,20,2,20)\}$$
$$U = \sum_i \frac{e_i}{p_i} = 0.76, H = 20$$



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

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

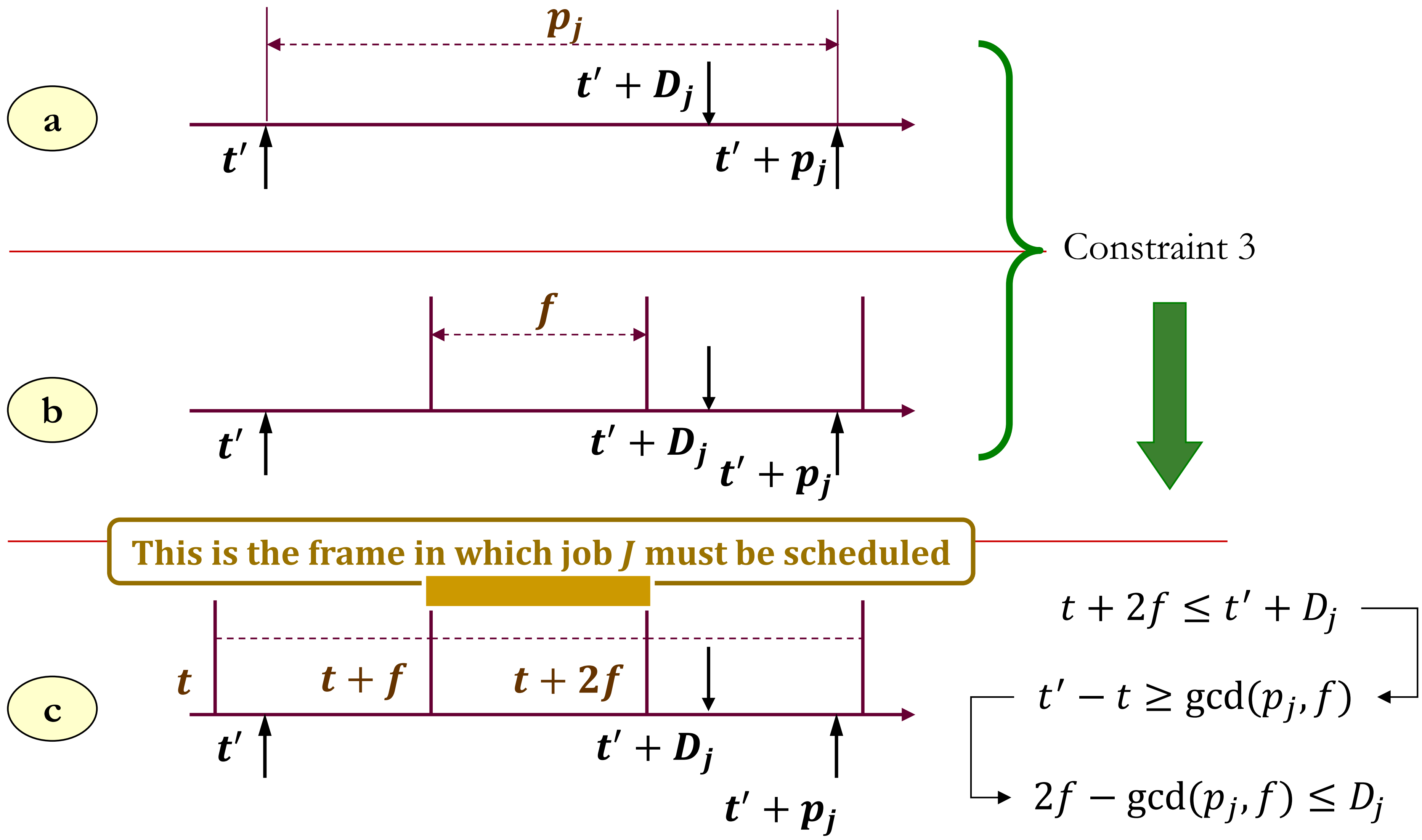
Clock-driven scheduling /4

- Reasons of complexity control suggest *minimizing* the size 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 (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

- **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 an integer divisor of the 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 set to be scheduled in that frame
  - This can be expressed as: $\boldsymbol{2f - \gcd(p_i, f) \leq D_i}$ 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 ≥ 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 ≤ 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 task τ_i's jobs by ***slicing*** their (WCET) e_i^w into fragments small enough to artificially yield a "good" *f*

Clock-driven scheduling /6

- 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 /7

```
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 slices in currentBlock;
    while aperiodic job queue not empty do
      execute aperiodic job at top of queue;
    end do;
  end do;
end SCHEDULER
```

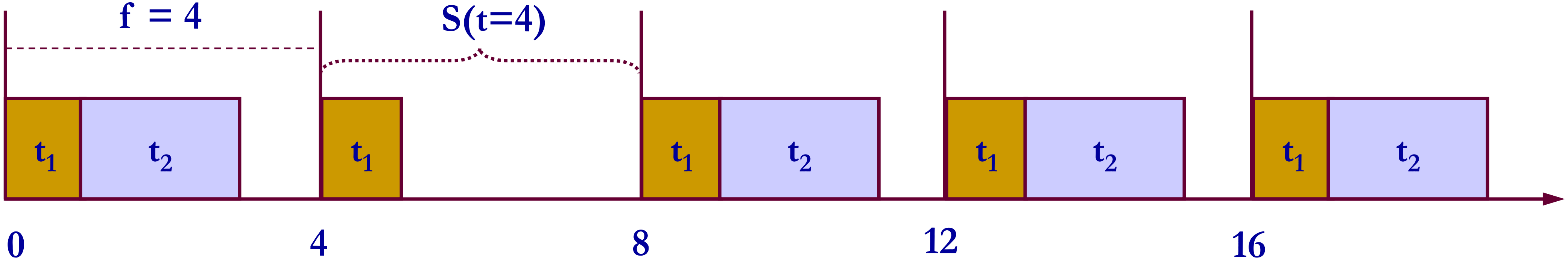
Example (slicing) – 1/2

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

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

$\boldsymbol{\tau_3}$ causes disruption since we need $\boldsymbol{e_3 \le f \le 4}$ to satisfy c3

We must therefore slice $\boldsymbol{e_3}$: how many slices do we need?

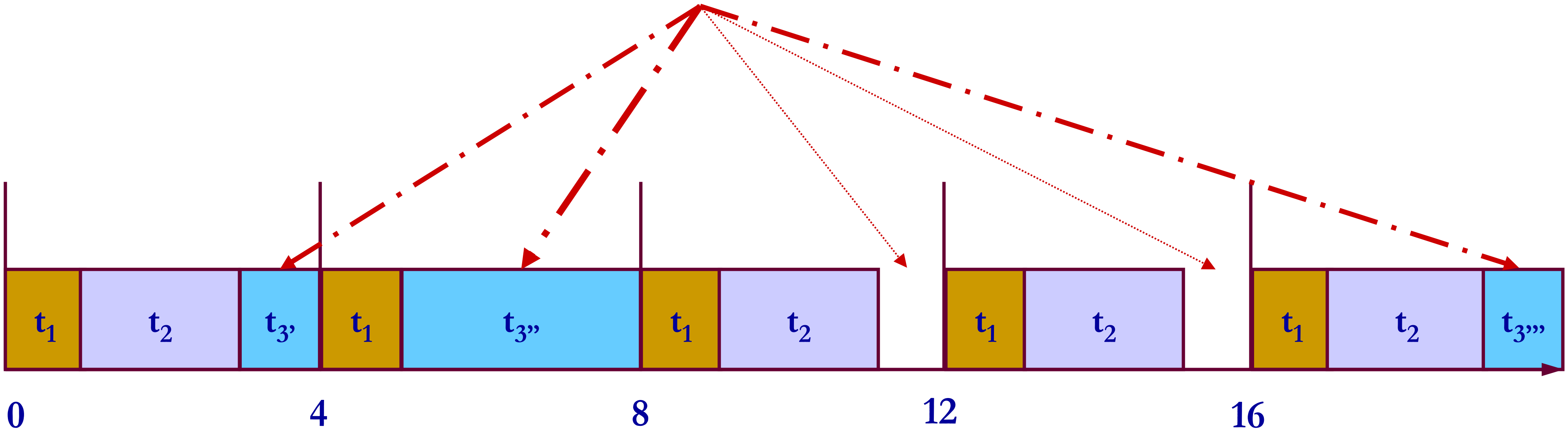


We first look at the schedule with $\boldsymbol{f = 4}$ and $\boldsymbol{F = \left(\frac{H}{f}\right) = 5}$

without $\boldsymbol{\tau_3}$, 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

- Completing a job much ahead of its deadline is of no use
- Any spare time in time slices should be given to *aperiodic jobs*, thus allowing the system to produce more value added
- The principle of ***slack stealing*** allows aperiodic jobs to execute *in preference* to periodic jobs when possible
  - Each minor cycle may include some amount of slack time not used for scheduling periodic jobs
    - The slack is a *static* attribute of each minor cycle
- 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)
  - This allows the system to become more reactivy
  - But it also requires a fine-grained interval timer (again!) to signal the end of the slack time for each minor cycle

Design issues /2

- 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

- 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

- **Pro**
  - Comparatively simple design
  - Simple and robust implementation
  - Complete and cost-effective verification
- **Con**
  - 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

Static/fixed priority scheduling (FPS)

- Two main strategies exist for priority assignment, which is all we need to determine FPS
- ***Rate monotonic***
  - A task with *faster rate* (hence lower period) takes precedence
  - Optimal assignment under preemptive *task-level* priority-based scheduling and implicit deadlines
  - The consequent scheduling is called **RMS**
- ***Deadline monotonic***
  - A task with *higher urgency* (shorter relative deadline) goes first
  - Equivalently optimal for constrained deadlines

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

Appraising scheduling /1

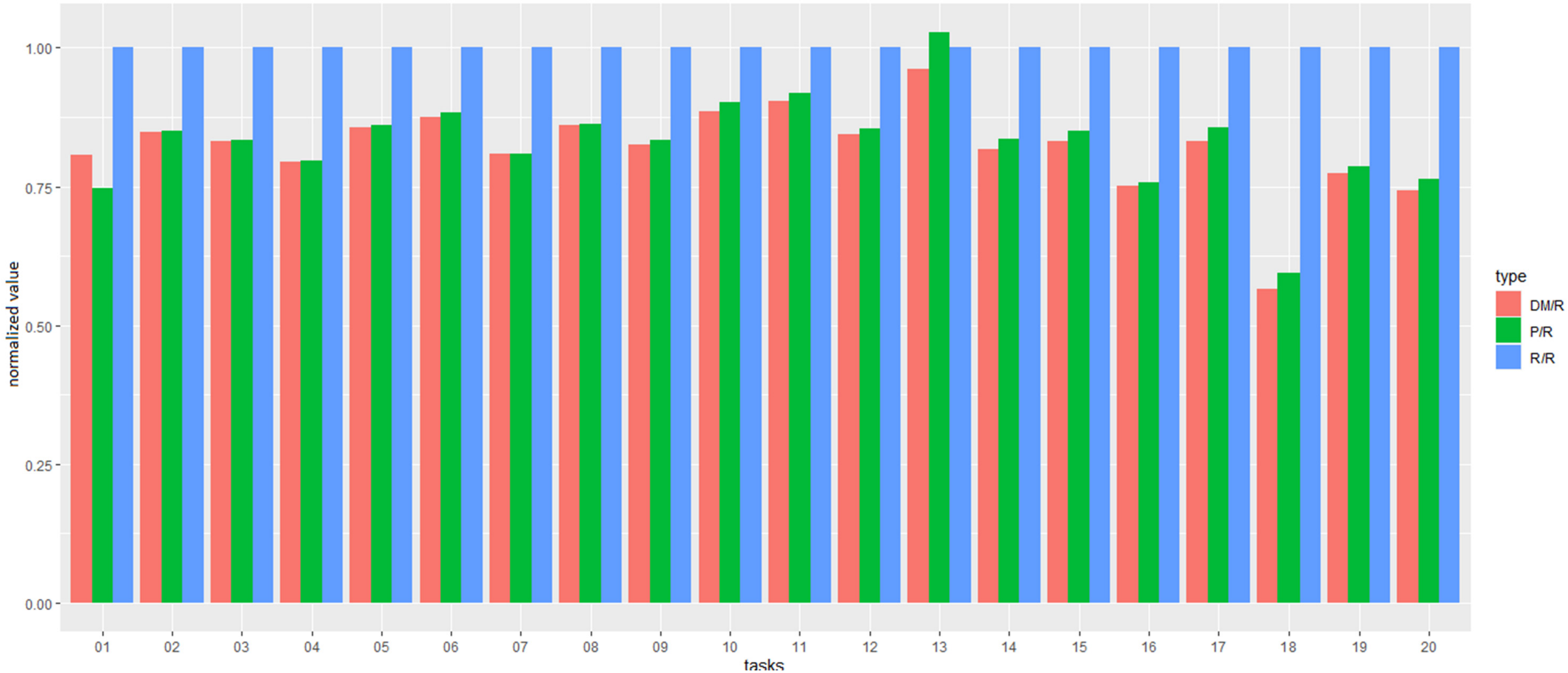
- **Theorem** [Liu & Layland: 1973]
  For single processors and implicit or constrained deadlines, EDF's *schedulable utilization is* 1
  - A *necessary and sufficient* (i.e., exact) test for implicit deadlines
- Checking for $\Delta = \sum_{i=1}^{n} \frac{e_i}{\min(d_i, p_i)} \leq 1$, aka ***density***, is a *sufficient* schedulability test for EDF for constrained deadlines, $U \leq 1 \leq \Delta$

Appraising scheduling /2

- 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 becomes instable
  - 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

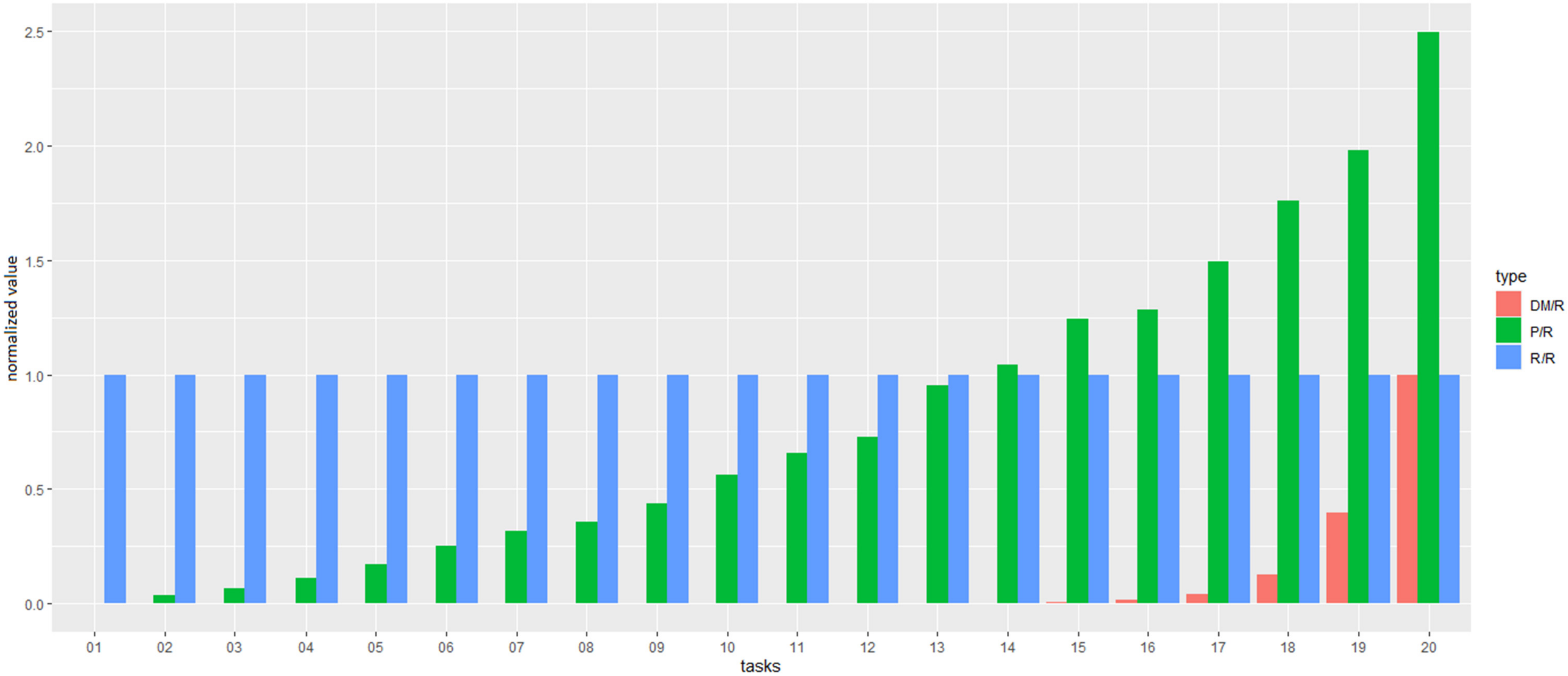
Deadline miss and preemption count ratio over normalized run count (EDF, $U > 1$)



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

Overload situations /2

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

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
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

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

Overload situations /4

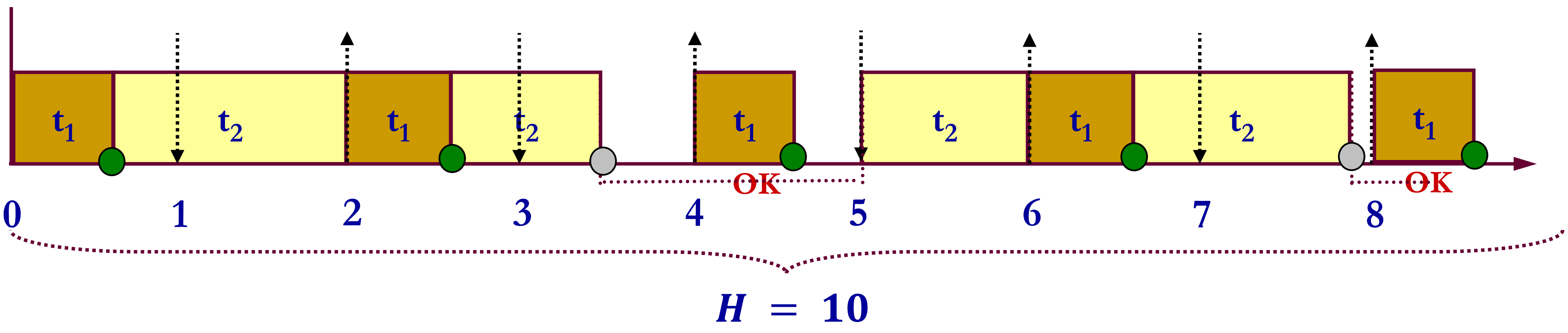
$(\varphi_i, p_i, e_i, D_i)$

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

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

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

What happens to *T* under EDF?



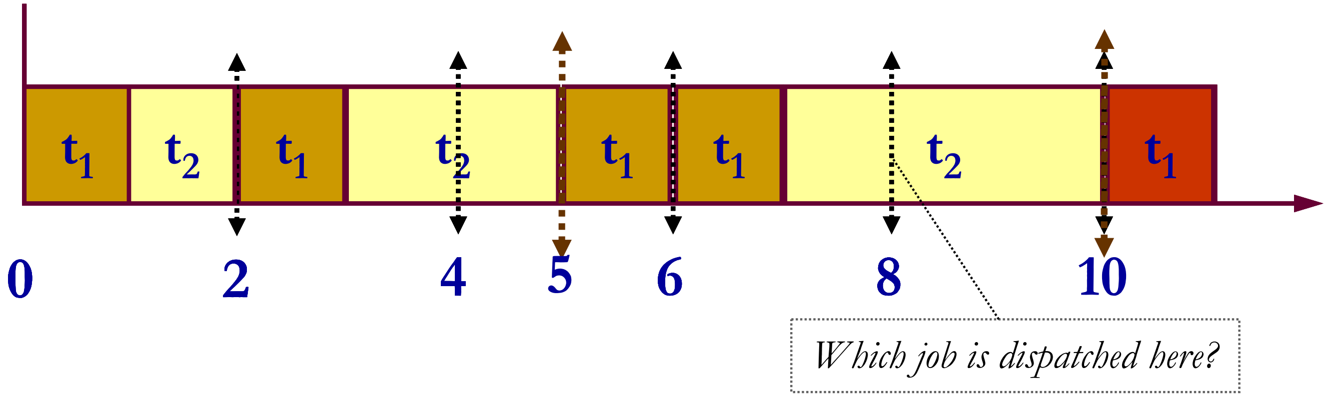
The exact utilization-based test tells us that T is feasible under EDF
(We don't need to draw its timeline to tell that!)

Overload situations /5

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

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?

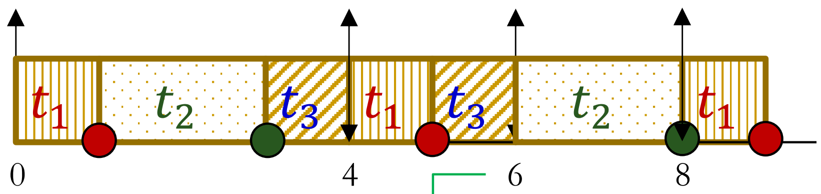
What about

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

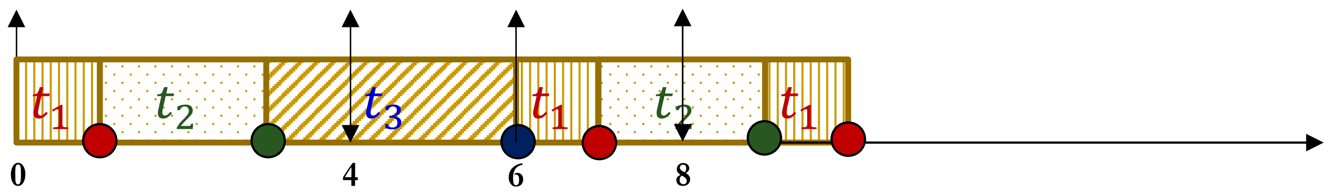
$$T = \{t_1 = (0,4,1,4), t_2 = (0,6,2,6), t_3 = (0,8,3,8)\}, U = \frac{23}{24}, H = 24$$

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, <u>which misses its deadline</u>

With EDF



EDF may incur *less* preemptions than FPS

Preemption count /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

- 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

- **<u>Theorem</u>**: 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

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

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

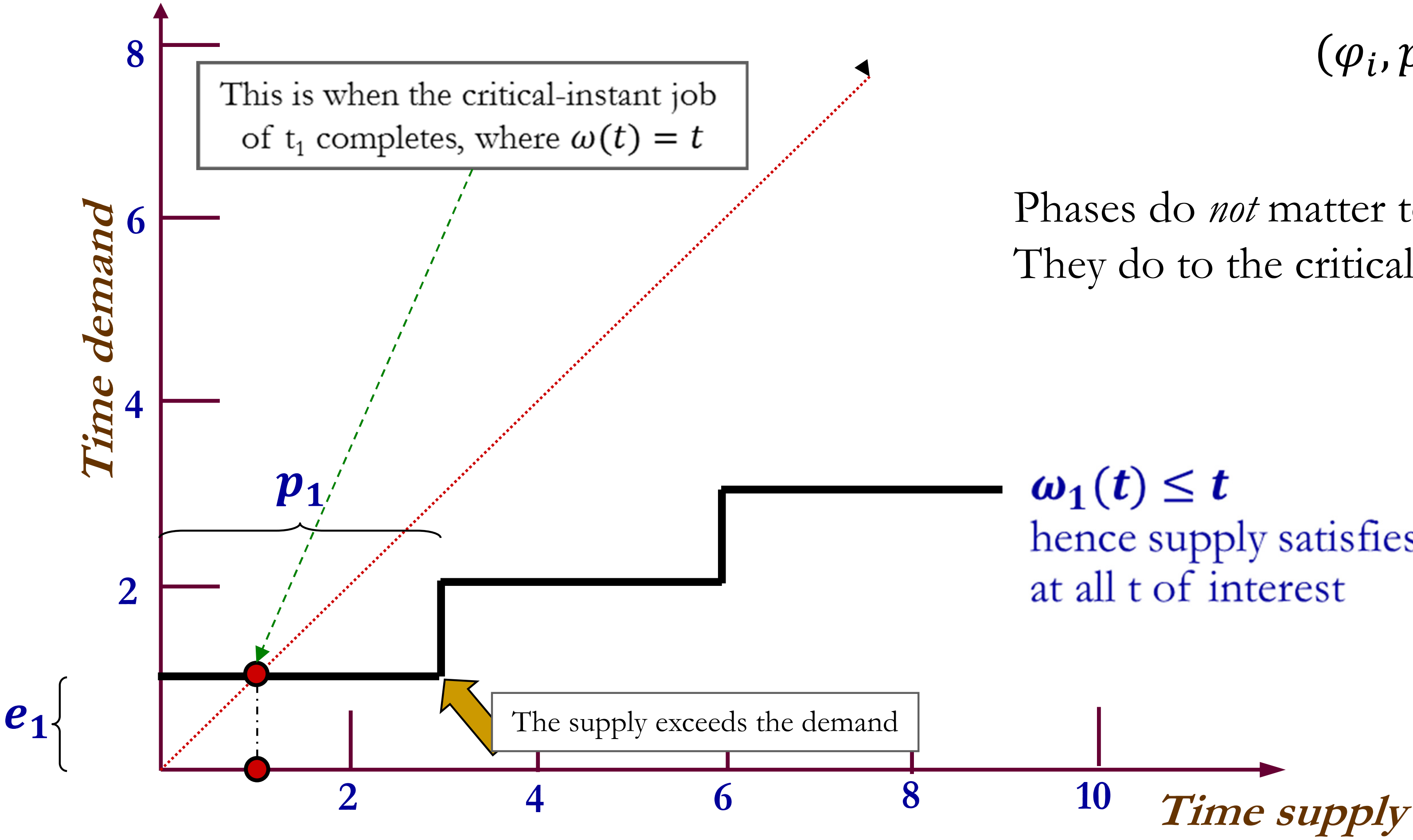
- ***Time Demand Analysis***, TDA, studies ω as a function of time, $\omega(t)$
  - As long as $\omega(t) \leq t$ *for some (selected)* t for the job of interest, the supply satisfies the demand, hence the job can complete in time

- **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

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

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

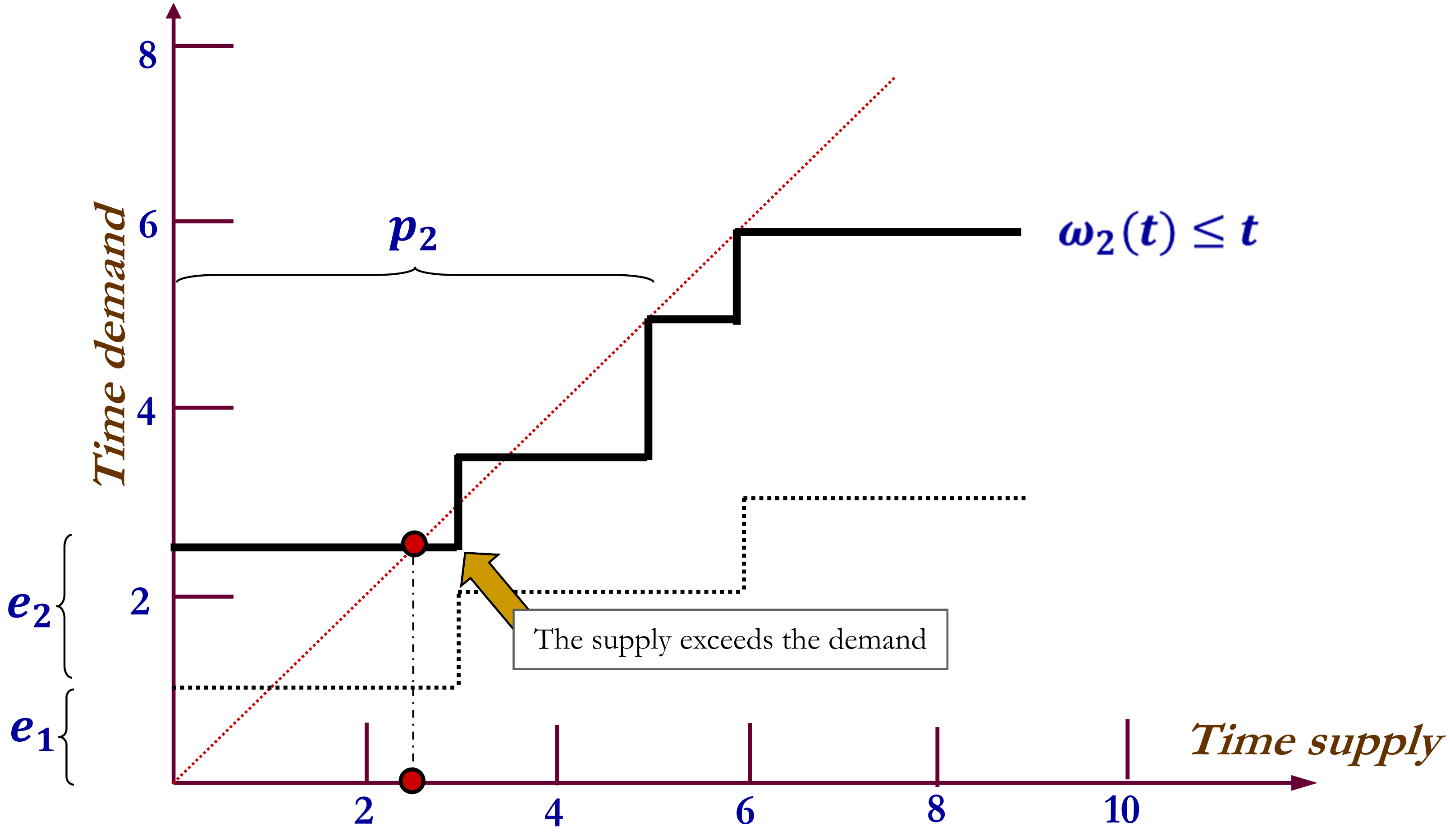


Phases do *not* matter to TDA
They do to the critical instant!

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

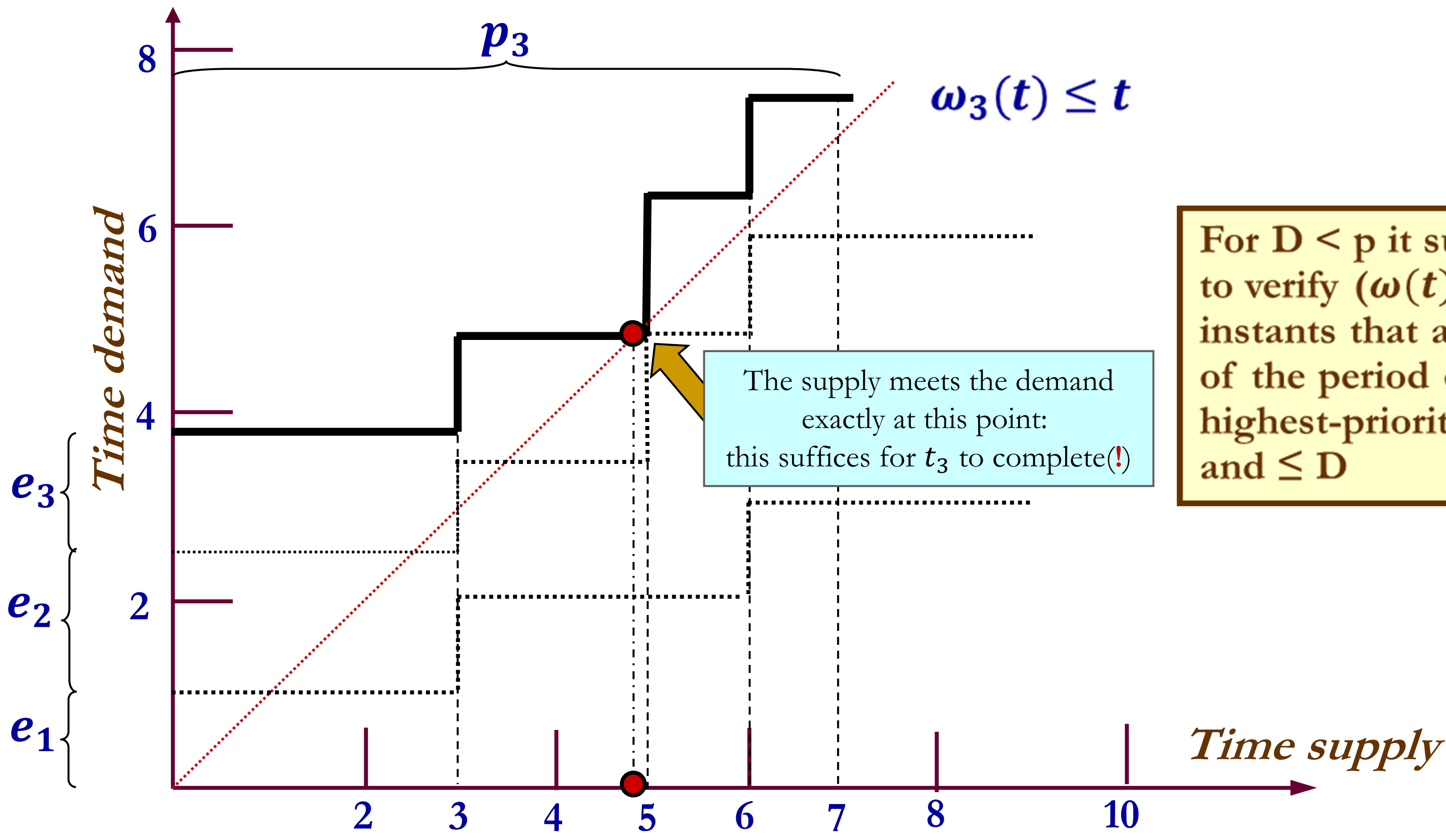
Time demand analysis /3

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



Time demand analysis /4

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



For D < p it suffices to verify $(\omega(t) \leq t)$ at time instants that are multiple of the period of the highest-priority tasks and ≤ D

Time demand analysis /5

- 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

- **<u>Theorem</u>** [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

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

Ready queue: $J_{1,1}$, $J_{2,1}$

Time window 1 [0,70)
Time left for $J_{2,1}$: 70-26 = 44
Still to execute: 62-44 = 18

Ready queue: $J_{1,2}$, $J_{2,1}$

Time window 2 [70,100)
Time left for $J_{2,1}$: 30-26 = 4
Still to execute: 18-4 = 14
Release time of job $J_{2,2}$

Ready queue: $J_{2,1}$, $J_{2,2}$

Time window 3 [100,140)
Time left for $J_{2,1}$ = 40
$J_{2,1}$ completes at: 114 (R = 114)
Time available for $J_{2,2}$: 40-14 = 26
Still to execute: 62-26 = 36

Ready queue: $J_{1,3}$, $J_{2,2}$

Time window 4 [140,200)
Time available for $J_{2,2}$: 60-26 = 34
Still to execute: 36-34 = 2

Ready queue: $J_{2,2}$, $J_{2,3}$

Time window 5 [200,210)
Release time of job $J_{2,3}$
$J_{2,2}$ completes at: 202 (R = 102)
Time available for $J_{2,3}$: 10-2 = 8
Still to execute: 62-8 = 54

Ready queue: $J_{1,4}$, $J_{2,3}$

Time window 6 [210,280)
Time available for $J_{2,3}$: 70-26 = 44
Still to execute: 54-44 = 10

Ready queue: $J_{1,4}$, $J_{2,3}$

Time window 7 [280,300)
Time available for $J_{2,3}$: 20-20 = 0
Release time of job $J_{2,4}$

Ready queue: $J_{1,5}$, $J_{2,3}$, $J_{2,4}$

Time window 8 [300,350)
Time available for $J_{2,3}$: 50-6 = 44
$J_{2,3}$ completes at: 300+6+10 = 316 (R = 116)

$J_{2,1}$'s response time is **not** worst-case!

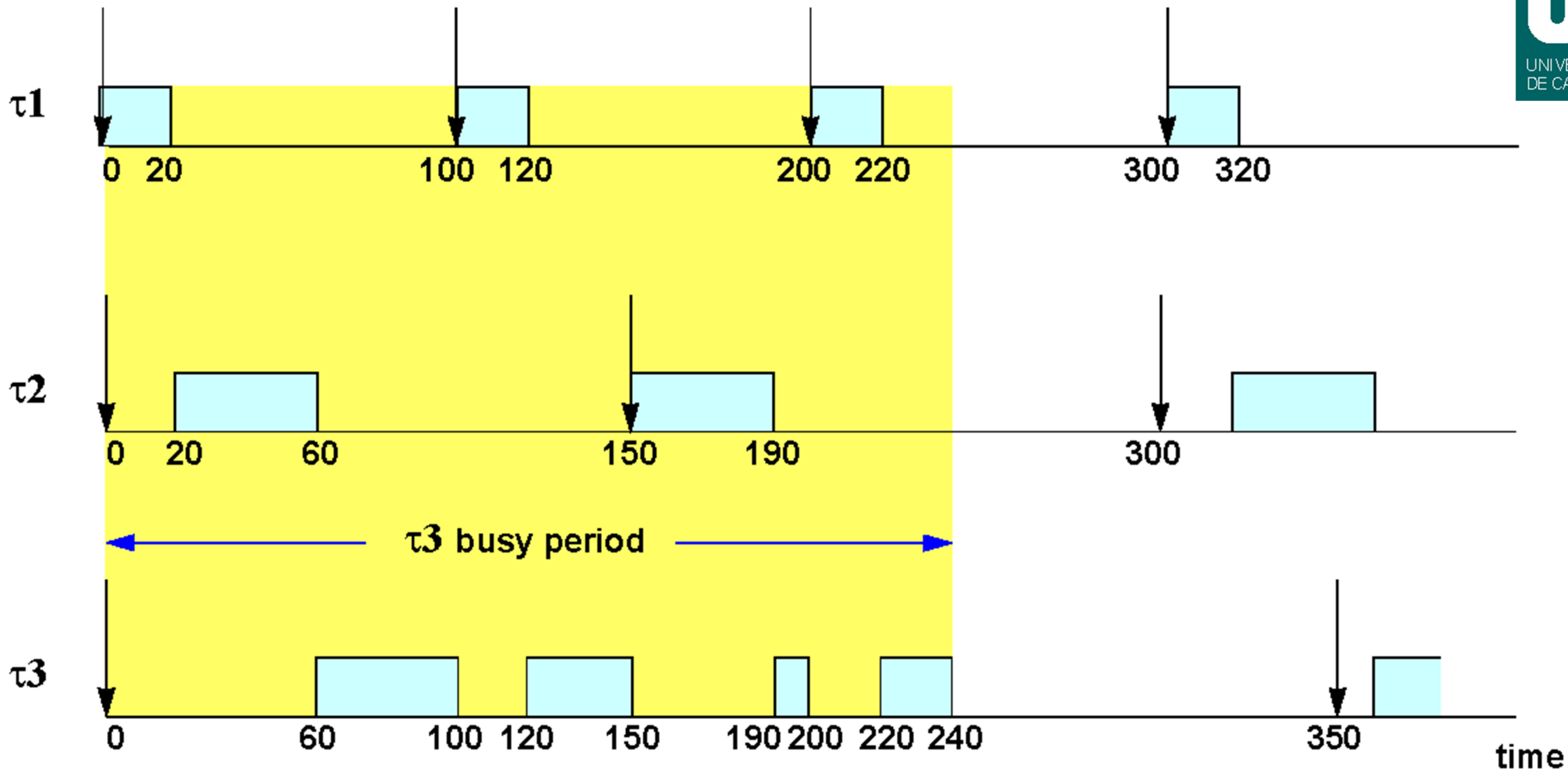
Still in ready queue: $J_{2,4}$

The T_2 busy period extends beyond this point (!)

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 ≤ 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:
$$\forall t_1, t_2 : t_2 > t_i, \boldsymbol{df}(t_1, t_2) \le t_2 - t_1$$
- For periodic tasks with no offsets and $U \le 1$, it holds that:
$$\boldsymbol{df}(t_1, t_2) \le \boldsymbol{df}(0, t_2 - t_1)$$
- The ***demand bound function*** helps generalize the test
$$\boldsymbol{dbf}(L) = \max_t \big(df(t, t + L)\big) = df(0, L), L > 0$$
- **Theorem** [Baruah, Howell, Rosier: 1990] Exact test for EDF:

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

  - $D(T)$ 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**)