3.a Fixed-Priority Scheduling

Credits to A. Burns and A. Wellings

RTS York

Where we look at the schedulability tests for FPS, their strength and weaknesses, we accommodate aperiodic tasks, and we review the priority assignment algorithms

Notation in this section

C:	Worst-case computation time (WCET) of the task ($= e$)
D:	Relative 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:	task period ($= p$) or minimum inter-arrival time between releases
U:	Processor utilization
a-Z:	The name of a task

The simplest workload model

- The application consists of n tasks, for constant n
- All tasks are *periodic* with known periods
  - Whence the name "*periodic workload model*"
- All tasks are assumed *independent*
  - No sharing of logical resources, no precedence constraints
- All tasks have implicit deadline $(D = T)$
  - All jobs must complete before the release of their successor
- All tasks have a single, fixed and known WCET attribute
  - Which can be trusted as *a safe and tight upper-bound*
- All runtime overheads are collated in the tasks' WCET
  - Context-switch times, handing of clock interrupts, etc.

Fixed-priority scheduling (FPS)

- Still the most widely used approach in industry
- Each task has a fixed (static) priority determined off-line
- The "priority" of a real-time task reflects its temporal attributes
  - This is *orthogonal* to the task's contribution to the ***integrity*** of system operation: the latter is called ***criticality***
  - In Section 8, we shall discuss **mixed-criticality systems**, which employ scheduling solutions that also contemplate *criticality* attributes
- The ready jobs are dispatched to execution in the order determined by the static priority of their corresponding task
  - FPS at run time if fully determined by the priority assignment algorithm used at design time!

Preemption and non-preemption /1

- With priority-based scheduling, when a high-priority (HP) task releases a job while a lower-priority (LP) one is running, the HP job is placed *at the top* of the ready queue
  - In a *preemptive* scheme, such an event will cause immediate switch of execution to the HP job
  - With *non-preemption*, the LP job will be allowed to complete before the job at the top of the ready queue will be dispatched to execution
- Preemptive schemes (e.g., FPS and EDF) enable HP tasks to be more reactive, which make them preferable
  - Non-preemptive schemes protect "delicate" fractions of execution

Preemption and non-preemption /2

- Non-preemptive strategies allow LP jobs to continue executing *for a bounded time* before being preempted
  - *Deferred preemption* ("give me a little bit more")
  - *Cooperative dispatching* ("I will tell you when")
- When overload situations are liable to occur, a utility function computed at run time would help mitigate the consequent hazards
  - *Value-based scheduling* (VBS) would use that function to control preemption at high levels of utilization

Rate-monotonic scheduling (RMS)

- Each task is assigned a priority that reflects its period
  - The shorter the period, the higher the priority
  - Such priorities have to be unique: no ties allowed
- For any two tasks $\tau_i, \tau_j : T_i < T_j \rightarrow P_i > P_j$
  - ***Rate monotonic*** assignment is **optimal** under preemptive priority-based scheduling and implicit deadlines
- **Notice**
  - Priority **1** as numerical value is the lowest (least) priority
  - Task indices are sorted highest-priority to lowest-priority

Utilization-based tests /1

- [Liu & Layland, 1973] A simple *sufficient but not necessary* test exists for RMS for constrained-deadline task sets
  - It upper-bounds the *schedulable utilization* of RMS (FPS)

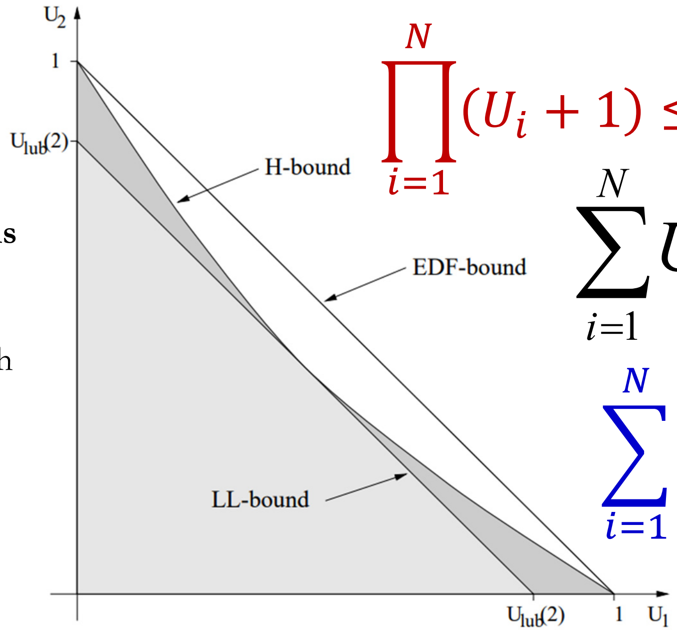
$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

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

- This shows that the schedulable utilization of FPS (RMS) is *less* than that of EDF
  - But this is a very pessimistic bound
- Utilization-based tests are simple to compute, but highly inaccurate: they often *don't know* …

Utilization-based tests /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



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

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

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

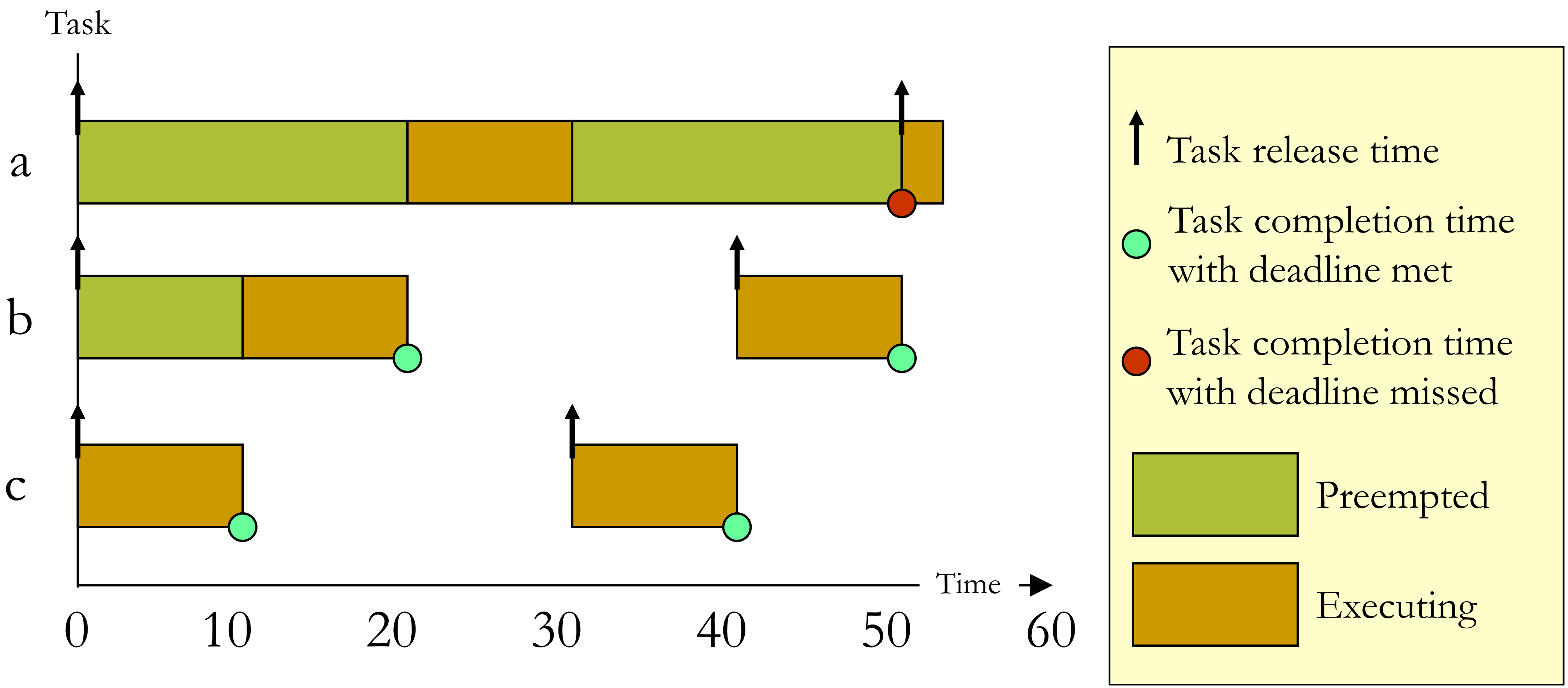
Examples of feasibility regions
Plot in an $n = 2$ U-space, where each point $U = \{U_1, U_2, \dots, U_n\}$ represents a periodic task set with utilization U_i

Example: taskset A

Task	Period	Computation Time	Priority	Utilization
	T	**C**	**P**	**U**
a	50	12	1 (low)	0.24
b	40	10	2	0.25
c	30	10	3 (high)	0.33

- The combined utilization of this task set is $U_A = 0.82$
- Above the threshold for three tasks: $U_A > U(3) = 0.78$
  - Task set A fails the utilization-based test
- Hence, we have *no* a-priori answer on its actual feasibility from this test

Timeline for taskset A



Example: taskset B

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	80	32	1 (low)	0.40
b	40	5	2	0.125
c	16	4	3 (high)	0.25

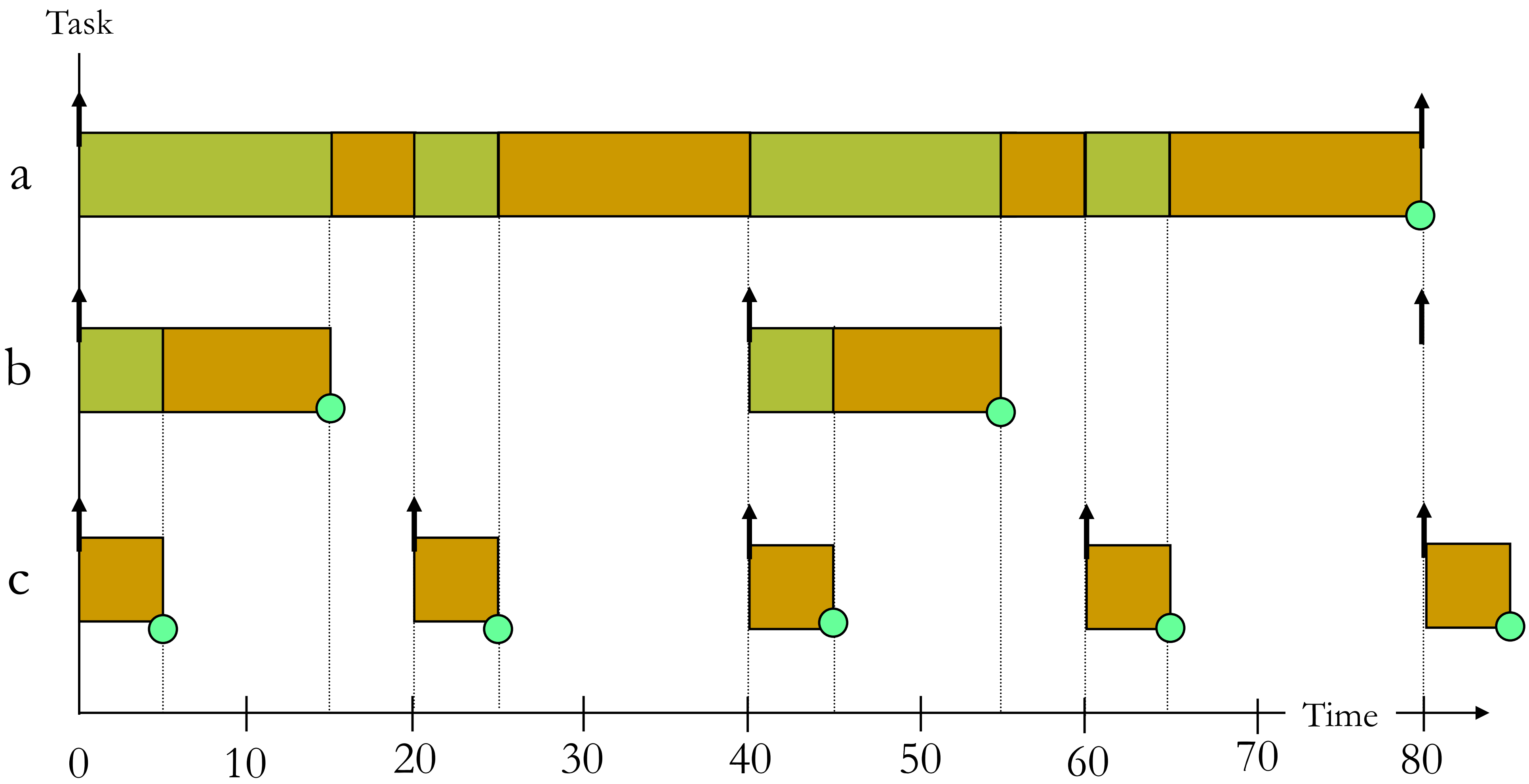
- Its combined utilization is $U_B = 0.775 < U(3) = 0.78$
  - It passes the utilization-based test
- Hence, this task set is guaranteed to meet all its deadlines

Example: taskset C

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	80	40	1 (low)	0.50
b	40	10	2	0.25
c	20	5	3 (high)	0.25

- Its combined utilization is $U_C = 1.0 > U(3) = 0.78$
  - It fails the utilization-based test
  - But, interestingly, the task periods are fully harmonic
- The timeline shows that the task set meets all its deadlines
  - FPS (RMS) performs very well with harmonic-rate tasks

Timeline for taskset C



Response time analysis (RTA) /1

- RTA is a *feasibility test* : it is 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 some tasks will miss their deadline at run time
    - Unless the WCET values turn out to be pessimistic
- FPS determines *exactly* which tasks will miss their deadline in that case

Response time analysis /2

- Task τ_i's response time R_i is defined as $R_i = C_i + I_i$
  - $I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$, where $hp(i)$ is the set of tasks with higher priority than τ_i's, upper-bounds the ***interference*** suffered by task τ_i within its *busy period*
  - The ceiling function $\lceil f \rceil$ gives the smallest integer greater than f: a job of τ_i will be preempted for a *full* execution of a job of $\tau_{j<i}$ released *exactly* at τ_i's end
- The RTA fixed-point equation is solved by forming a recurrence relation

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

  - where the set of values $w_i^0, w_i^1, w_i^2, \ldots, w_i^n$ is monotonically non-decreasing
- The solution of the equation is when $w_i^n = w_i^{n+1}$, when the time supply meets the time demand
- If $R_i \leq D_i$, then task τ_i is feasible

Response time algorithm

```
for i in 1..N loop
   n := 0
   w_i^n = C_i
   loop
      calculate w_i^{n+1}
      if w_i^{n+1} = w_i^n then
         exit value found
      else if w_i^{n+1} > d_i then
          exit deadline missed
         end if
      end if
      n := n + 1
   end loop
end loop
```

If the recurrence does not converge before d_i we may set a termination condition to attempt to determine how long past T_i, job i completes

Example: taskset D

Task	Period T	Computation Time C	Priority P	Utilization U
a	7	3	3 (high)	0.4285…
b	12	3	2	0.25
c	20	5	1 (low)	0.25

$$R_a = 3$$

$$\begin{cases} w_b^0 = 3 \\ w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6 \\ w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6 \\ R_b = 6 \end{cases}$$

Example (cont'd)

$$\begin{cases} w_c^0 = 5 \\ w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11 \\ w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14 \\ w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17 \\ w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20 \\ w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20 \\ \boxed{R_c = 20} \end{cases}$$

Revisiting taskset C

Task	Period T	Computation Time C	Priority P	Response Time R
a	80	40	1 (low)	80
b	40	10	2	15
c	20	5	3 (high)	5

- Its combined utilization is $U_C = 1.0 > U(3) = 0.78$
- The utilization-based test fails, but RTA shows that the task set will meet all its deadlines

Sporadic tasks and other extensions

- Sporadic tasks have a ***minimum inter-arrival time***
  - This should be preserved at run time if schedulability is to be ensured, but how can it **?**
- The RTA for FPS works perfectly well for $D \leq T$ as long as the stopping criterion becomes $W_i^{n+1} > D_i$
- Interestingly, RTA also works perfectly well with *any* priority ordering, as long as the task indices reflect it

Coexistence of hard and soft tasks /1

- In many real-world situations, the tasks' given WCET values are considerably higher than the average case
  - WCET are far off the center of the execution-time Gaussian
- Occasionally, interrupts may arrive in bursts, or abnormal sensor readings may require significant extra computation to restore a baseline truth
  - This may cause the worst-case conditions to be extremely pessimistic
- Analyzing feasibility with WCET may thus lead to *very low* processor utilization at run time
  - The goal of worst-case analysis is to *preserve the rights of hard tasks*: once they are guaranteed, the possible waste is not their problem
  - But it is the problem of soft tasks, which only get the "remainder", which excessive pessimism may reduce dramatically
- Some common-sense rules help contain such pessimism

Coexistence of hard and soft tasks /2

- **Rule 1 :** All tasks (hard and soft) should be schedulable using *average* execution times and *average* sporadic arrival rates
  - Hence, when some tasks exceed their average demand, it may *not* be possible to meet all deadlines
  - This condition is known as a *transient overload*, when the current workload exceeds the utilization deemed schedulable
    - Transient in that not all tasks simultaneously are in worst-case mode
- **Rule 2 :** All hard real-time tasks should be schedulable using WCET and worst-case arrival rates of all tasks (including soft)
  - Hard tasks receive the high partition of the available priorities
  - Soft tasks receive the other (lower) priorities
    - Both partitions use RM or DM priority assignment algorithms
- With Rule 2 no hard real-time task will miss its deadline
  - If Rule 2 causes unacceptably low feasibility for soft tasks, then WCET values or arrival rates should be “tuned down”

Handing aperiodic tasks /1

- They do *not* have minimum inter-arrival times: consequently *cannot* claim deadlines
  - We may be interested in the system being responsive to them
  - In cyclic scheduling we would use *slack stealing* for them
- We might run aperiodic tasks one priority level below all hard tasks, just above soft tasks
  - In that manner, under preemption, aperiodic tasks won't be able to steal resources from hard tasks
  - Yet, this solution would penalize soft tasks, which might miss their deadlines too often
- We need another kind of solution …

Handing aperiodic tasks /2

- Besides preserving hard tasks and giving fair opportunities to soft tasks, an aperiodic-geared solution must *choose* which objective to optimize
  - The response time of the job *at the head* of the aperiodic queue (one, as soon as possible)
  - The average response time of *as many jobs as possible* for a given aperiodic queuing discipline
- Possible choices
  - Execute the aperiodic jobs by interrupting the periodic jobs
  - Execute the aperiodic jobs in the background
  - Use slack stealing

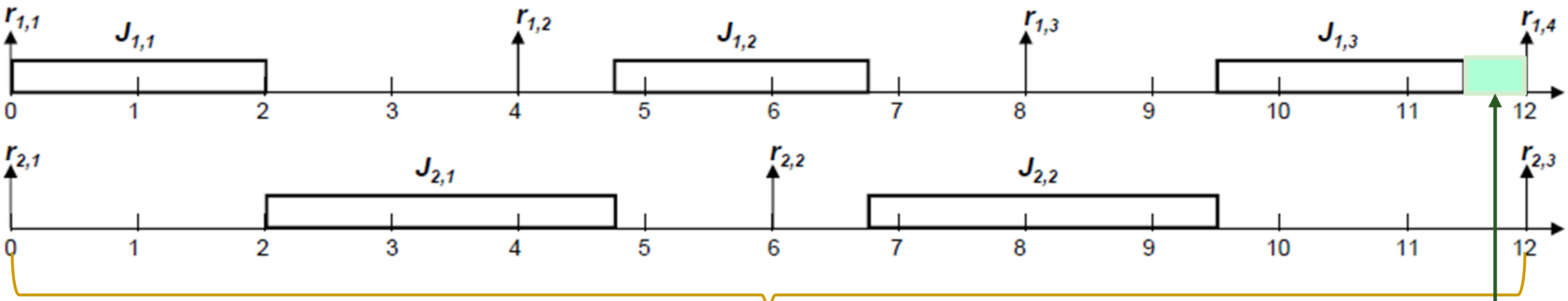
  - Use dedicated servers

Handing aperiodic tasks /3

- ***Slack stealing***
  - Difficult to implement for preemptive systems
    - For them, the slack $\sigma(t)$ is a *not* a constant but a function of the time t at which it is computed
  - The slack stealer is ready when the aperiodic queue is not empty; it is suspended otherwise
  - When ready and $\sigma(t) > 0$, the slack stealer is assigned the highest priority; the lowest when $\sigma(t) = 0$
  - Static computation of $\sigma(t)$ for some t is useful but only when the release jitter in the system is very low
    - Under EDF, $\boldsymbol{\sigma(t = 0) = min_i\{\sigma_i(0)\}}$, where $\sigma_i(0) = D_i - \sum_{k=1,\dots,i} e_k$ for *all* jobs released in the hyperperiod starting at $t = 0$

Computing the slack under EDF

$T_1 = (4, 2)$, $T_2 = (6, 2.75)$ - EDF scheduling: (ϕ_i, p_i, e_i, D_i)



$H = 12$

$min_{i,j}\left(\sigma_{i,j}(0)\right)$

$$\sigma_{1,1}(0) = D_1 - C_1 = 4 - 2 = \mathbf{2}$$
$$\sigma_{2,1}(0) = D_2 - C_1 - C_2 = 6 - 2 - 2.75 = \mathbf{1.25}$$
$$\sigma_{1,2}(0) = D_{1_2} - 2 \times C_1 - C_2 = 8 - 2 \times 2 - 2.75 = \mathbf{1.25}$$
$$\sigma_{2,2}(0) = D_{2_2} - 2 \times C_1 - 2 \times C_2 = 12 - 2 \times 2 - 2 \times 2.75 = \mathbf{2.5}$$
$$\sigma_{1,3}(0) = D_{1_3} - 3 \times C_1 - 2 \times C_2 = 12 - 3 \times 2 - 2 \times 2.75 = \mathbf{0.5}$$

Computing the slack under FPS /1

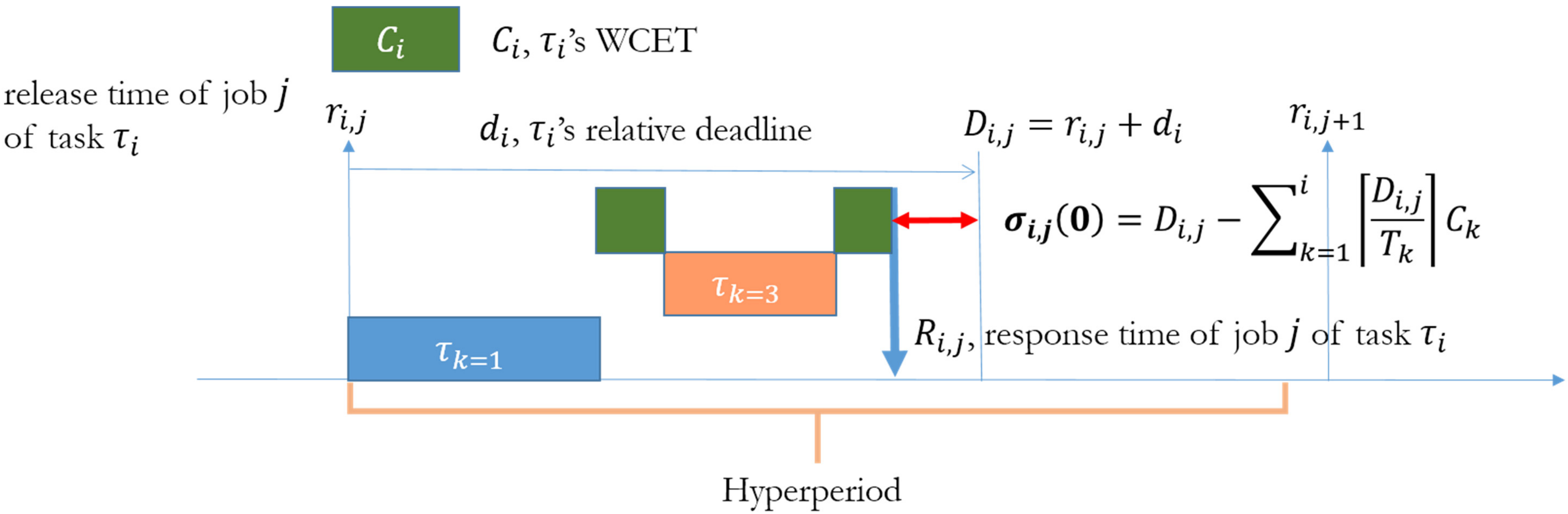
- The slack of periodic jobs of τ_i should be computed based on their *effective deadline* D_i^e
  - The effective deadline for a precedence-constrained task is the successor's deadline minus the successor's WCET
  - The smallest duration that the successor task needs to be able to complete in time
- The *initial* slack of periodic job J_{ij} (the j^{th} job of task τ_i) in hyperperiod H is determined as

$$\boldsymbol{\sigma_{i,j}(0) = max\left(0, D_{ij}^e - \sum_{k=1}^{i} \left\lceil \frac{D_{ij}^e}{T_k} \right\rceil C_k\right)}$$

- The slack *cannot* be negative
  - If it was, the task would *not* have enough time to execute

Computing the slack under FPS /2

- For independent tasks (with no precedence constraints), the effective deadline is *just* the normal deadline, which reduces the computation to



Computing the slack under FPS /3

- The amount of slack that a system has in a given time interval may depend on *when* the slack is used
  - The decision of when to schedule an aperiodic job J_a to minimise its response time, must consider its execution time
  - It may be opportune to schedule it later, even if slack is currently available: greed is no good
- For any periodic task set, under FPS, and any aperiodic queuing policy, *no* valid algorithm exists that minimizes the response time of *every* aperiodic jobs
- Similarly, no valid algorithm exists that minimizes the *average* response time of the aperiodic jobs

T.-S. Tia, J. W.-S. Liu, and M. Shankar, "Algorithms and Optimality of Scheduling Aperiodic Requests in Fixed-Priority Preemptive Systems," Journal of Real-Time Systems, 10(1), pp. 23-43, 1996.

Handing aperiodic tasks /4

- ***Periodic server*** (PS)
  - The PS is a notional (T_{ps}, C_{ps}) periodic task scheduled at the highest priority solely to execute aperiodic jobs
    - The PS has a ***budget*** C_{ps} time units and a ***replenishment period*** of length T_{ps}
    - When the PS is scheduled and executes aperiodic jobs, it consumes its budget at the rate of 1 unit per unit of time
    - Budget is exhausted when $C_{ps} = 0$ and replenished periodically
  - The PS is *backlogged* when the aperiodic job queue is nonempty, and it is idle otherwise
  - The PS is eligible for execution only when ready, backlogged, and with a non-exhausted budget, $C_{ps} > 0$
- Specific instances of this model legislate over consumption and replenishment

Handing aperiodic tasks /5

- ***Polling server***, a simple (naïve) kind of PS
  - It is given a fixed budget that is *replenished at every period*
  - The budget is *immediately consumed* if the server is scheduled while having no backlog
  - It is *not* ***bandwidth preserving***, hence it is inefficient
    - An aperiodic job that arrives just after the server has been scheduled while idle, must wait until the next replenishment time
  - Bandwidth-preserving servers need additional rules for consumption and replenishment of their budget

Handing aperiodic tasks /6

- ***Deferrable Server*** (DS), a *bandwidth-preserving* PS
  - On empty backlog, it retains its budget while staying ready
    - If an aperiodic job requires execution during the DS period, it can be served immediately
  - The budget is replenished at the start of the new period (**!**)
    - If an aperiodic job arrives ε time units before the end of T_{ds}, the request begins to be served and blocks lower-priority periodic tasks
    - When the budget is replenished, new aperiodic jobs may then be served for the full budget
  - If that happens, in $\omega(t)$, DS contributes a solid interference of $C_{ds} + \left\lceil \frac{t - C_{ds}}{T_{ds}} \right\rceil C_{ds}$, *longer* than $1 \times C_{ds}$ per busy period

Handing aperiodic tasks /7

- ***Sporadic Server*** (SS), fixes the bug in DS
  - The budget is replenished only when exhausted and at a minimum guaranteed distance from previous execution
    - Hence no longer at a fixed rate
  - This places a tighter bound on its interference and makes schedulability analysis simpler and less pessimistic
- This is the default server policy in POSIX

SS rules under FPS

- ***Consumption rules***
  - After replenishment, a backlogged SS consumes budget only if executing, hence when no HP task is ready
  - Replenishment is limited to the quantity of actual consumption
- ***Replenishment rules***
  - t_r records the time that SS' budget was last replenished
  - t_e records the time when SS first begins to execute since t_r
    - $t_e > t_r$ is the latest time at which LP tasks execute
  - The next replenishment time is set to $t_e + T_{ss}$
- ***Exception***
  - If only HP tasks had been busy since t_r, then $t_e + T_{ss} > t_r + T_{SS}$ and SS is late: hence, budget is fully replenished as soon as exhausted

SS rules unveiled

- Let t_a be the time at which SS has full budget *and* becomes backlogged, and $t_f \geq t_a$ the time at which SS becomes idle
- In the $[t_a, t_f]$ interval, when SS is continuously active, three cases are possible

1. SS has consumed no capacity: $t_{r_{next}} = t_f + T_{SS}$ → no replenishment, and no interference in that interval
2. SS has consumed all capacity: $t_{r_{next}} = t_a + T_{SS}$ → full replenishment, and bounded interference in that interval
3. SS has consumed fractional capacity: $t_{r_{next}} = t_f + T_{SS}$ → fractional replenishment, and interference lower than allowed in that interval

Task sets with D < T

- For $D = T$, Rate Monotonic priority assignment (aka RMS) is optimal
- For $D < T$, ***Deadline Monotonic*** priority ordering (DMPO), where $D_i < D_j \rightarrow P_i > P_j$, is optimal
  - Any task set Q that is schedulable by priority-driven scheme W, it is also schedulable by DMPO
- The proof of optimality of DMPO involves transforming the priorities of Q as assigned by W until the ordering becomes as assigned by DMPO
  - Each step of the transformation preserves schedulability

DMPO is optimal /1

- Let τ_i, τ_j be two tasks with adjacent priorities in Q such that under W we have $P_i > P_j \wedge D_i > D_j$
- Define scheme W' to be identical to W except that tasks τ_i, τ_j are swapped
- Now consider the schedulability of Q under W'
- All tasks $\{\tau_k\}$ with priority $P_k > P_j$ will be unaffected
- All tasks $\{\tau_s\}$ with priority $P_s < P_i$ will be unaffected as they will experience the same interference from τ_j and τ_i
- Task τ_j which was schedulable under W, now has a higher priority, suffers less interference, and hence must be schedulable under W'

DMPO is optimal /2

- All that is left to show is that task τ_i, which has had its priority lowered, is still schedulable
- Under W we have $R_j \leq D_j, D_j < D_i$ and $R_i \leq T_i$
- Task τ_j only interferes once during the execution of task τ_i hence $R_i' = R_j \leq D_j < D_i$
  - Under W' task τ_i completes at the time task τ_j did under W
  - Hence task τ_i is still schedulable after the swap
- Priority scheme W' can now be transformed to W'' by choosing two more tasks that are in the wrong order for DMPO and swapping them

Generalized priority assignment (aka simulated annealing)

Theorem: If task *p* is assigned the lowest priority and it is feasible, then, if a feasible priority ordering exists for the complete task set, one such ordering exists where task *p* is assigned the lowest priority

```
procedure Assign_Pri (Set : in out Task_Set;
                      N   : Natural; -- number of tasks
                      OK  : out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, OK); -- is task K feasible now?
      exit when OK;
    end loop;
    exit when not OK; -- failed to find a schedulable task
  end loop;
end Assign_Pri;
```

Summary

- A simple (periodic) workload model
- Delving into fixed-priority scheduling
- A (rapid) survey of schedulability tests for FPS
- Some extensions to the workload model
- Priority assignment techniques

Selected readings

- N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, A.J. Wellings (**1995**)
  Fixed priority pre-emptive scheduling: an historical perspective
  DOI: 10.1007/BF01094342
- D. Faggioli, M. Bertogna, F. Checconi (**2010**)
  Sporadic Server revisited
  DOI: 10.1145/1774088.1774160