3.c Exercises on task interactions, and further model extensions

Credits to A. Burns and A. Wellings

Where we use a running example to recap the effects of resource access control protocols on task blocking, and make further extensions to the workload model

Task interactions and blocking

- Causing a job J_h to wait for a lower-priority job to complete some computation, undermines the principle of priority
- If that happens, job J_h suffers *priority inversion* and it is said to be *blocked*
  - The blocked state is other than *preempted* or *suspended*
- We would like RTA to contemplate blocking $\boldsymbol{B}$, so that we can continue to use it for FPS
  - But then we must determine a conservative bound to it

Incorporating blocking in RTA

- The cost of blocking B adds to response time R, *outside* of the interference factor I
$$\boldsymbol{R_i = C_i + B_i + I_i}$$
- The magnitude of the effects of blocking on response time is an indicator of the effectiveness of the resource access control protocol in use
- We shall now use a running example to expose the principal differences in their performance

Running example

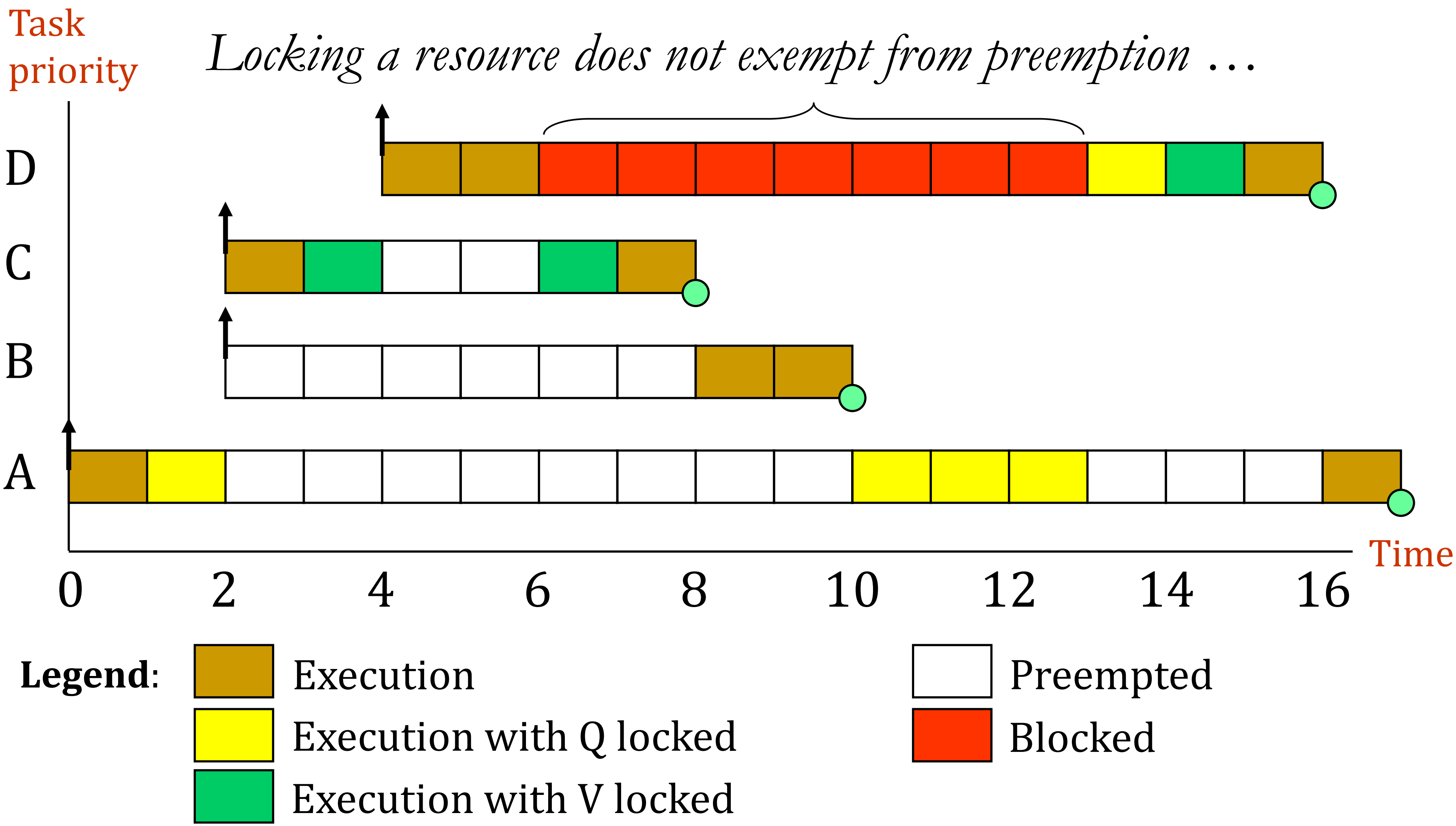
- Consider the example system below: let us see how the principal resource access control protocols treat it

Task	Priority	Execution sequence	Offset
A	1 (low)	e**QQQQ**e	0
B	2	ee	2
C	3	e**VV**e	2
D	4 (high)	ee**QV**e	4

Legend:
- e: one unit of execution;
- Q (or V): one unit of use of resource R_q (or R_v) under mutual exclusion

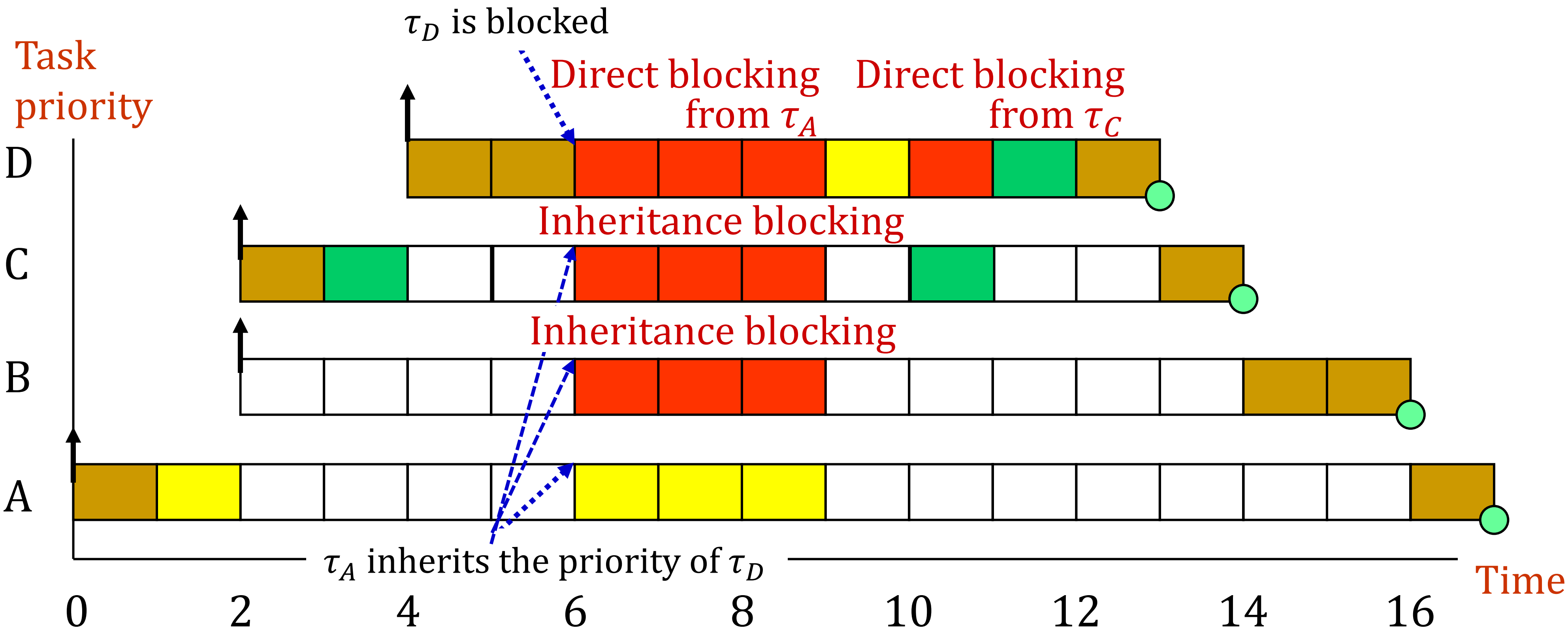
  - Simple locking
  - Basic Priority Inheritance
  - Basic Priority Ceiling (with *system ceiling*)
  - Ceiling Priority

With simple locking



With Basic Priority Inheritance (BPIP)

If task τ_p is blocking task τ_q, then τ_p runs with τ_q's priority …



Bounding *direct* blocking under BPIP

- If the system has $\{r_{j=1,..,K}\}$ critical sections that can lead to a task τ_i being blocked under BPIP, then K is the maximum number of times that τ_i can be blocked
- The upper bound on the blocking time $B_i(rc)$ for τ_i that contends for K critical sections thus is

$$\boldsymbol{B_i(rc) = \sum_{j=1}^{K} use(r_j, i) \times C_{max}(r_j)}$$

Where $use(r_j, i) = 1$ if r_j is used by at least one task $\tau_l\!: \pi_l < \pi_i$ and one task $\tau_h\!: \pi_h \geq \pi_i$ | 0 otherwise, and $C_{max}(r_j)$ denotes the worst-case duration of use of r_j by *any* such task τ_l

- The worst case for task τ_i with BPIP is to block for the longest duration of contending use on access to *all* the resources it needs
- Note that the running example includes *inheritance blocking* too!

With Ceiling Priority protocols

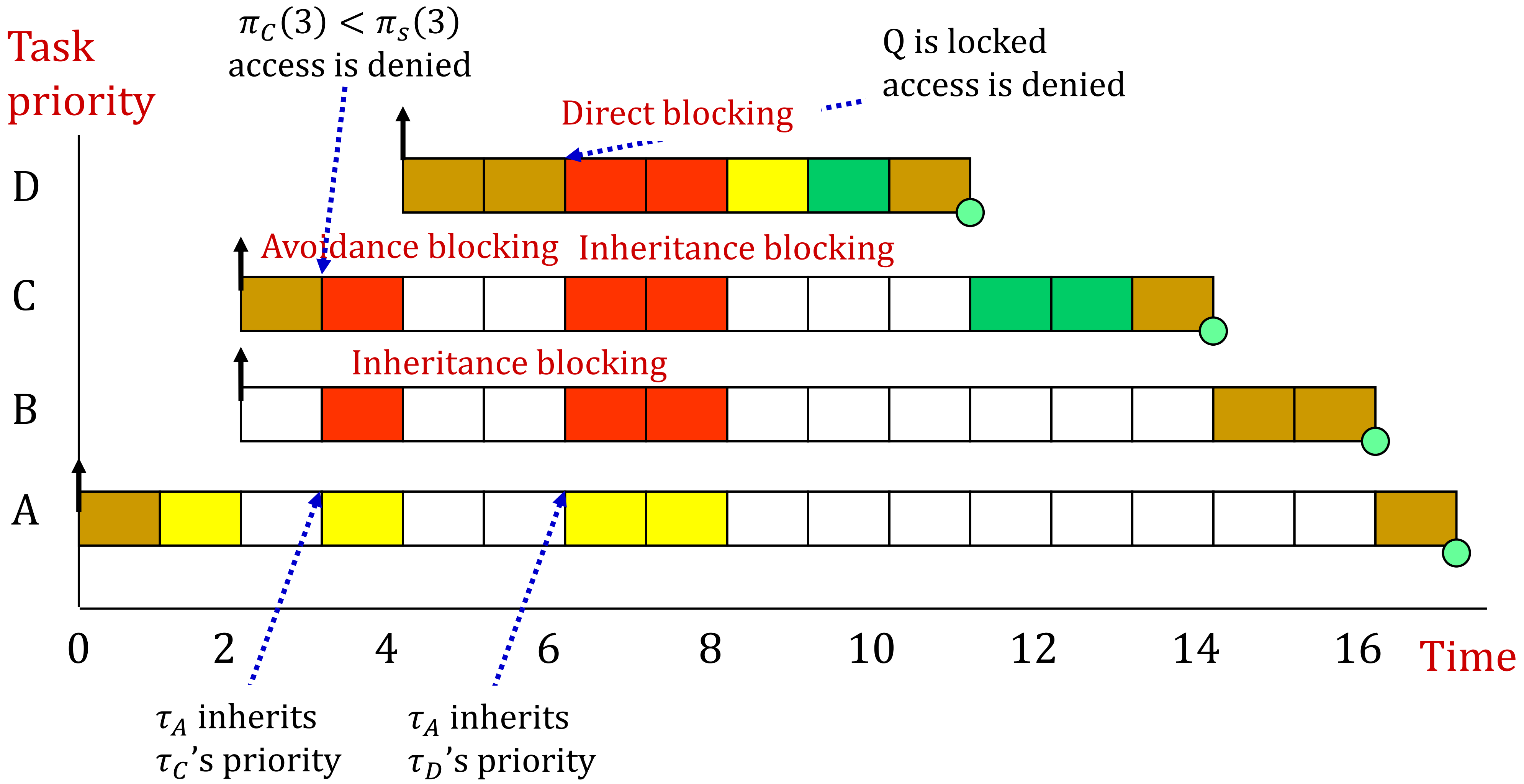
- We shall consider two main variants of them
  - *Basic Priority Ceiling Protocol* (aka "Original CPP")
    - Which uses the system ceiling $\pi_s(t)$
  - *Ceiling Priority Protocol* (aka "Immediate CPP")
    - Which does *not* use the system ceiling
- When using either of them on a single processor
  - A high-priority task can only be blocked by lower-priority tasks *at most once* per job
  - Deadlocks are prevented by construction because transitive blocking is also prevented by construction
  - Mutual exclusive access to resources is ensured by the protocol itself, hence locks are *not* needed

Recalling the BPC protocol (BPCP)

- Each task τ_i has an assigned *static* priority
  - Perhaps determined by deadline monotonic assignment
- Each resource r_k has a *static* ceiling attribute defined as the maximum priority of the tasks that may use it
- τ_i has a *dynamic* current priority $\pi_i(t)$ at time t, set to the maximum of its assigned priority and any priorities it has inherited at t from blocking higher-priority tasks
- τ_i can lock a resource r_k at time t *if and only if* $\pi_i(t) > \pi_s(t)$
  - Where $\pi_s(t) = max_j(\pi_{r_j})$ for all r_j currently locked at t, *excluding those that* τ_i *locks itself*
- The blocking B_i suffered by τ_i is bounded by the longest critical section with ceiling $\pi_{r_k} > \pi_i$ used by lower-priority tasks

$$\boldsymbol{B_i = max_{k=1}^{K}(use(r_k, i) \times C_{max}(r_k))}$$

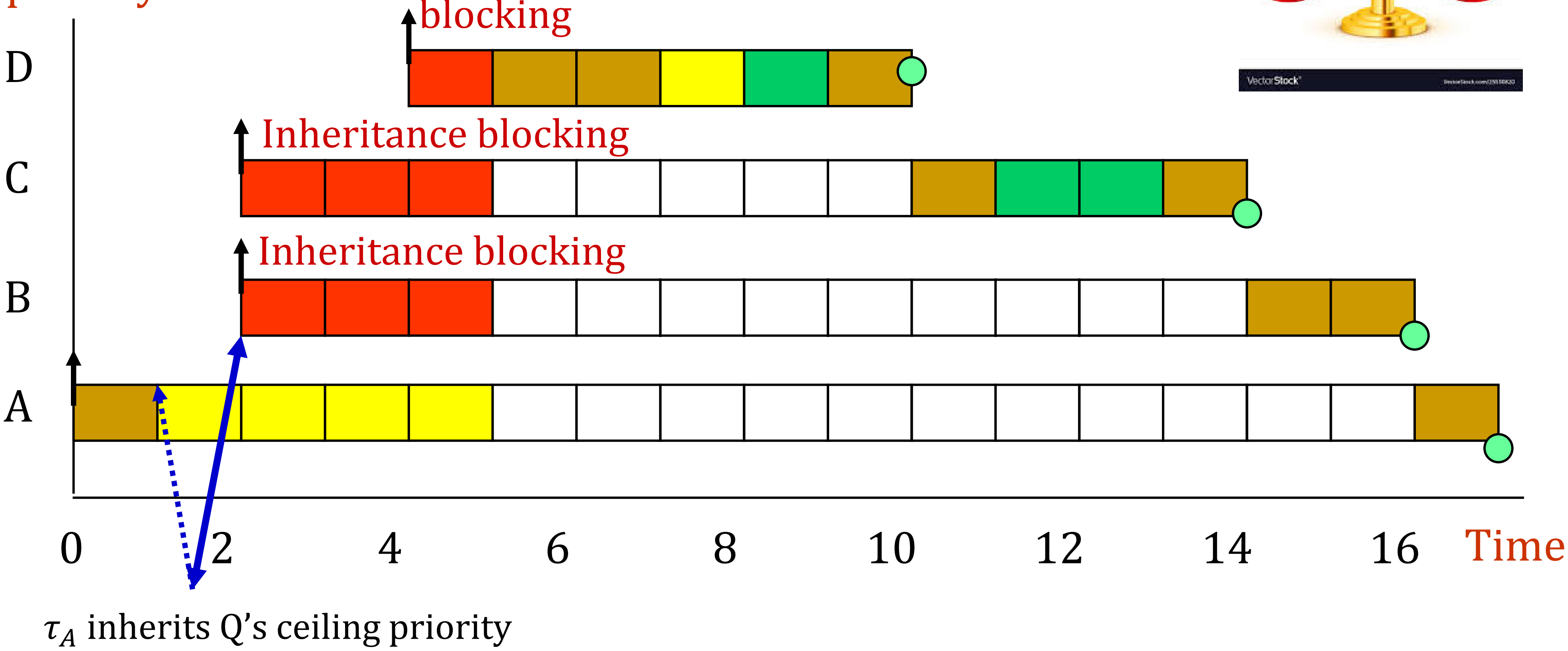
With Basic Priority Ceiling (BPCP)



Recalling the CP Protocol (CPP)

- Each task τ_i has an assigned *static* priority
  - Perhaps determined by deadline monotonic assignment
- Each resource r_k has a *static* ceiling attribute defined as the maximum priority of the tasks that may use it
- τ_i has a *dynamic* current priority $\pi_i(t)$ at time t, that is set to the maximum of its own static priority and the ceiling values of any resources it is currently using
- Any job of that task will suffer blocking *only once*, at release
  - Once the job starts executing, all the resources that it may use are free
  - If they were not, then some task would have priority $\geq$ than the job's, hence its execution would be postponed
- Blocking computed exactly as for BPCP

With Ceiling Priority (CPP)



BPCP vs. CPP

- Although the worst-case behavior of the two ceiling priority schemes is identical from a scheduling viewpoint, there are some points of difference between them
  - CPP is easier to implement than BPCP as blocking relationships *need not* be monitored
  - CPP leads to *less* context switches as blocking occurs *prior* to job activation
  - CPP requires *more* priority movements as they happen with *all* resource usages: BPCP changes priority only if an actual block has occurred
- CPP is called *Priority Protect Protocol* in POSIX and *Priority Ceiling Emulation* in Ada and Real-Time Java

Extending the workload model further

- Our workload model so far contemplates
  - Constrained and implicit deadlines ($D \le T$)
  - Periodic and sporadic tasks
  - Aperiodic tasks under some server scheme
  - Task interactions with blocking factored in RTA
- There are further extensions that we may need
  - Allowing *cooperative scheduling*
  - Incorporating *release jitter*
  - Allowing *arbitrary deadlines*
  - Allowing *offsets* (phases)

Cooperative scheduling /1

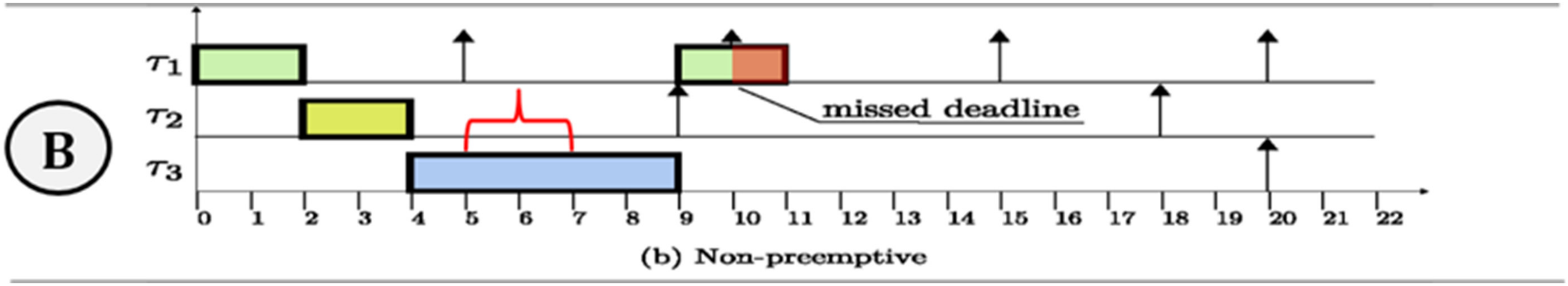
- Full preemption may not always suit critical systems
- ***Cooperative*** or ***deferred-preemption scheduling*** addresses this problem by chopping tasks into distinct slots of execution
  - Slots are said to be *floating* if their start is commanded at task level or *fixed* if it is programmed into the runtime schedule
  - The **yield** command marks the end of each such slot (not the last one)
    - If no *hp* task is ready at that point, the running task continues
  - The time duration of any such slot across all tasks is bounded by B_{max}
  - *Mutual exclusion must use non-preemption (else it breaks)*
- Deferring preemption has two interesting properties
  - It *dominates* both preemptive and non-preemptive scheduling
  - Each last slot of execution is free from interference

Cooperative scheduling /2

- Let F_i be the execution time of the *final slot* of τ_i's job, and B_{max} the worst-case blocking from deferring preemption
- The RTA recurrence relation must be adapted accordingly and becomes
$$\boldsymbol{w_i^{n+1} = C_i + B_{max} + I_i(w_i^n) - F_i}$$
  - Because the last slot is exempt from preemption
- When the fixed-point equation converges ($w_i^{n+1} = w_i^n$), τ_i's response time is computed as
$$\boldsymbol{R_i = w_i^n + F_i}$$

Deferred (limited) preemption /1

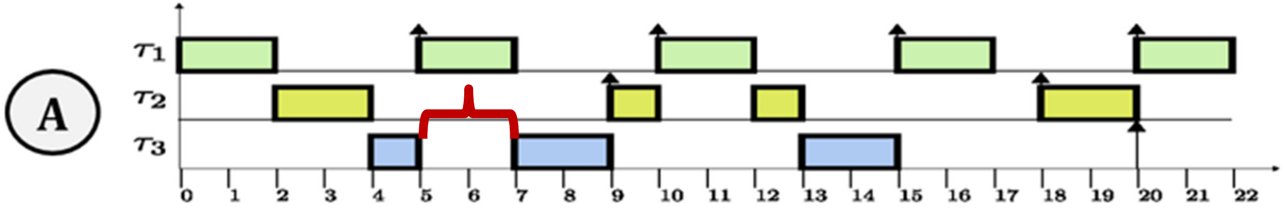
- Let us consider an implicit-deadline system in which τ_3 (lowest-priority task) has a *slot* [1,3] that should run free from preemption



- Allowing τ_3 to disable preemption for *all* of its execution (**case B**) is simple to implement, but unacceptably bad for τ_1

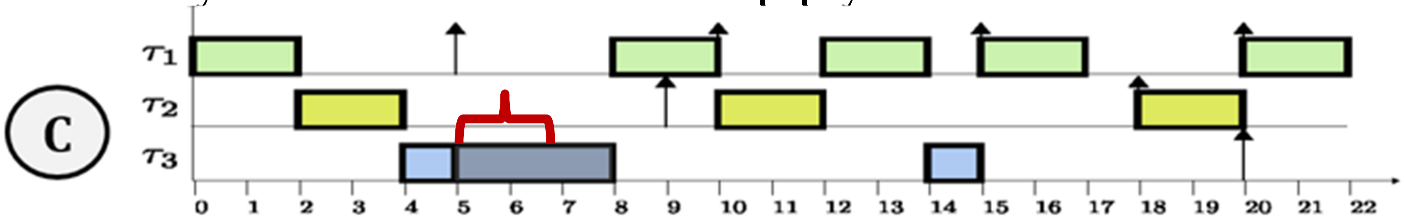
Deferred (limited) preemption /2

- If we were to run with full preemption (**case A**), then it would be τ_3 to be dissatisfied



(a) Fully preemptive (FPPS)

- If we gave τ_3 a *slot* of deferred preemption (**case C**) then everyone would be happy

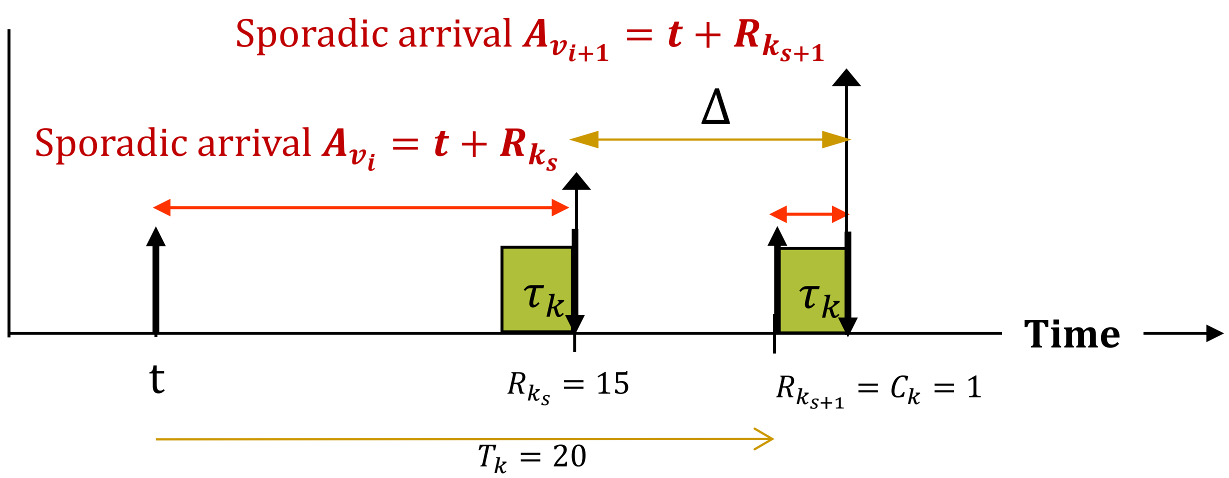


(c) Deferred preemption

  - Such slot would start at $t = 1$ into τ_3's execution and would last for the longest feasible duration ($c = 3$, in this case)

Release jitter /1

- Especially critical for *precedence-constrained* tasks
- **Example**: a periodic task τ_k with period $T_k = 20$, releases a *sporadic task* τ_v *at some point* of *some* runs of its (τ_k's) jobs
  - The release command ("signal") is conditional: it does not occur at constant time
  - This is a typical source of sporadic activation
- What is the minimum inter-arrival time of any two subsequent jobs of τ_v's?
  - To contain the variability we require the signal to be the last command of τ_k's job

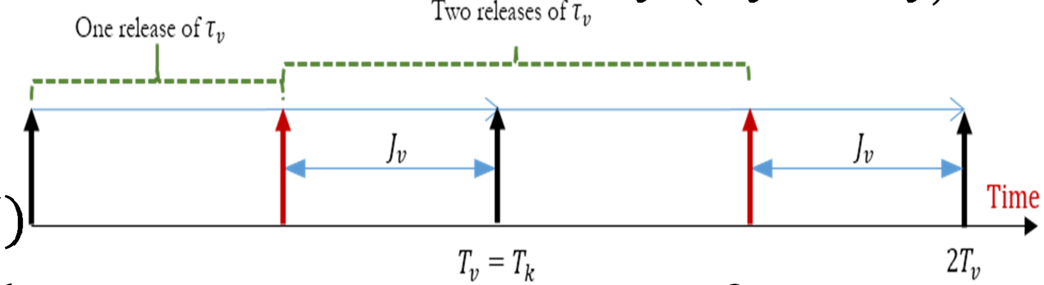


Release jitter /2

- The two successive releases of τ_v shown in the picture are spaced by $\Delta = 21 - 15 = 6$ time units from t
  - A much smaller interval than $T_k = 20$ (the predecessor's period)
- This phenomenon reflects τ_k's *response time jitter*, whose largest span is $R_{k_{max}} - R_{k_{min}}$
  - Which corresponds to τ_v's *release time jitter*
- To model this behaviour, we stipulate that
  - τ_v inherits τ_k's period T_k and suffers release jitter $J_v = R_k - C_k$
  - In the example, $J_v = 15 - 1 = 14$
- Hence, τ_v's *minimum interarrival time* is $T_k - J_v$
  - In the example, $20 - 14 = 6$

Release jitter /3

- Task τ_v in the example is released at $0, T-J, 2T-J, 3T-J$
- RTA says that task τ_i will suffer interference from τ_v $(\pi_i < \pi_v)$
  - Once, if $R_i \in [0, T-J)$
  - Twice, if $R_i \in [T-J, 2T-J)$
  - Thrice, if $R_i \in [2T-J, 3T-J)$



- This shows that tasks with release jitter cause *more* interference
  - RTA must be adjusted to capture it

$$\boldsymbol{R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j} \quad (\textit{less pessimistic} \text{ than } \left\lceil \frac{\boldsymbol{R_i}}{\boldsymbol{T_j - J_j}} \right\rceil)$$

- Periodic tasks can only suffer release jitter if the clock is jittery
  - The response time of a jittery periodic task τ_p measured relative to the *real* release time becomes $R'_p = R_p + J_p$

Arbitrary deadlines /1

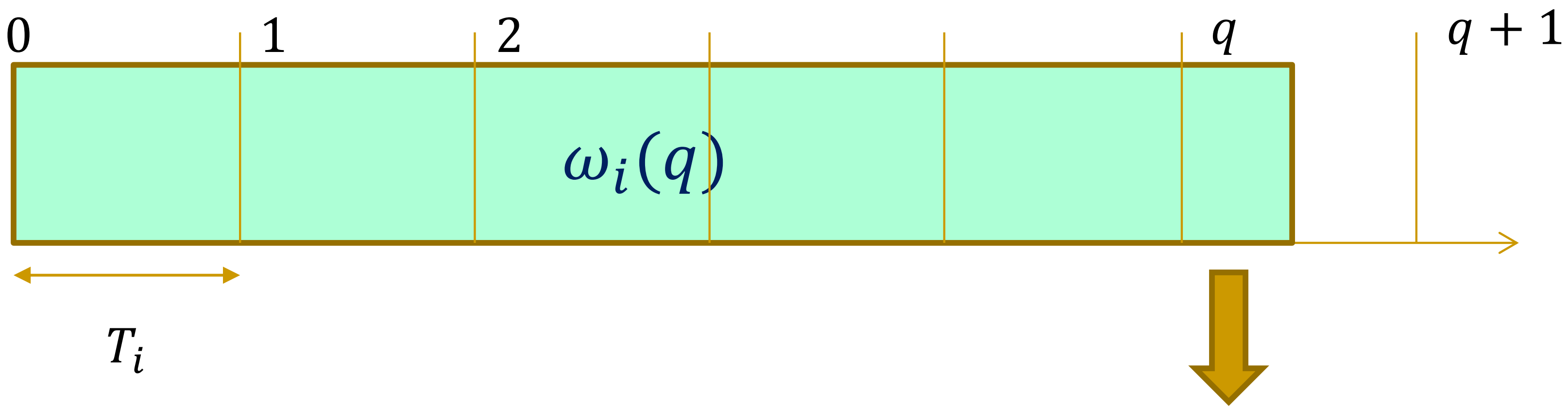
- When $D > T$, then $\boldsymbol{q > 1}$ jobs of the same task may compete for execution (in FIFO-within-priority mode)
- The RTA equation must be adapted to capture that event

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

$$\boldsymbol{R_i(q) = \omega_i^n(q) - qT_i}$$

  - $\boldsymbol{\omega_i(q)}$ extends as long as qT_i falls *within it*
    - Because that means that some jobs of τ_i's are still in the ready queue
  - The number $\boldsymbol{q}$ of releases is bounded by the lowest value for which $q : R_i(q) \leq T_i$
- τ_i's worst-case response time then is $\boldsymbol{R_i = max_q R_i(q)}$

Arbitrary deadlines /2



The $(q+1)^{th}$ job release of task τ_i falls in the level-i busy period, but this q is also the last index to consider as the next job release belongs in a different busy period

Arbitrary deadlines /3

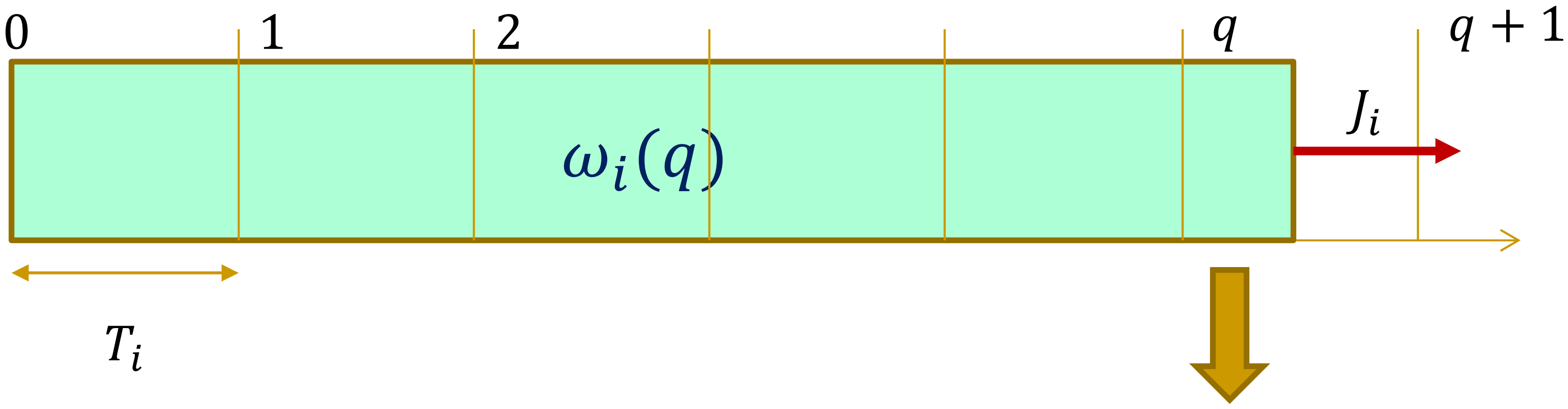
- When the formulation of the RTA equation is combined with the effect of release jitter, two alterations must be made
- First, the interference factor must be increased

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

- Second, if the task under analysis can suffer release jitter, then two consecutive windows could overlap if (response time plus jitter) were greater than the period

$$R_i(q) = \omega_i^n(q) - qT_i + \boldsymbol{J_i}$$

Arbitrary deadlines /4



If task τ_i has release jitter then the level-i busy period may extend until the next release

Non-optimal analysis for offsets /1

- So far, we assumed all tasks share a common release time (the *critical instant*)

Task	T	D	C	R	U
τ_a	8	5	4	4	0.5
τ_b	20	9	4	8	0.2
τ_c	20	**10**	4	**16**	0.2

Deadline miss!

- What if we allowed offsets?
  - Arbitrary offsets are *not* tractable with critical-instant based analysis
  - Hence we cannot use the RTA equation *directly* for them
- The critical instant assumption conservatively upper-bounds all possible combinations of offsets and releases

Non-optimal analysis for offsets /2

- Task periods are not entirely arbitrary in reality: they are likely to have some relation to one another
  - If at least two tasks have a common period, then we give one of them an offset O such that $\boldsymbol{O + D \leq T}$) and apply RTA to a transformation that *removes* the offset
- Doing so here, tasks τ_b, τ_c (tentatively with $O_c = \frac{T_c}{2}$) are replaced by a *single* notional task τ_n with
  - $T_n = T_c - O_c$
  - $C_n = \max(C_b, C_c) = 4$
  - $D_n = T_n$
  - *no* offset
- This technique allows using RTA and helps determine a "good" offset

Non-optimal analysis for offsets /3

- The notional task τ_n has two important properties
  - If it is deemed feasible (sharing a critical instant with all other tasks), then the two real tasks that it represents will meet their deadlines when one is given the stipulated offset
  - If all LP tasks are feasible when suffering interference from τ_n, then they will stay feasible when the notional task is replaced by the two real tasks (one of which with offset)
- These properties follow from the observation that τ_n always has no less CPU utilization than the two real tasks that it subsumes

Task	T	D	C	R	U
τ_a	8	5	4	4	0.5
τ_n	10	10	4	**8**	0.4

  - $R_n = 8 < D_n = 10$ becomes the (pessimistic but feasible) response time for τ_b and τ_c

Non-optimal analysis for offsets /4

- In a more general way, the notional task's parameters are set as follows

$$T_n = \frac{T_a}{2} = \frac{T_b}{2}$$

Where τ_a and τ_b have the same period, else we would use $Min(T_a, T_b)$ at the cost of greater pessimism

$$C_n = Max(C_a, C_b)$$

$$D_n = Min(D_a, D_b)$$

$$P_n = Max(P_a, P_b)$$

Priority relations

- This strategy can be extended to handle $k > 2$ tasks

Sustainability [Baruah & Burns, 2006]

- Extends the notion of predictability for single-core systems to wider range of relaxations of workload parameters
  - Shorter execution times
  - Longer periods
  - Less release jitter
  - Later deadlines
- For a scheduling algorithm to be sustainable, any such relaxation should *preserve* feasibility
  - Much like what predictability does but for less types of variation

Summary

- Completing the survey and critique of resource access control protocols by means of a running example
- Considering further desirable extensions to our workload model
- Contemplating the notion of *sustainability* for scheduling

Selected readings

- A. Baldovin, E. Mezzetti, T. Vardanega (**2013**)
  Limited preemptive scheduling of non-independent task sets
  DOI: 10.1109/EMSOFT.2013.6658596