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

Credits to A. Burns and A. Wellings

RTS York

TV1

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

Task interactions and blocking

- That a job J_h should wait for a lower-priority job to complete some computation, before being able to proceed, undermines the principle of priority
  - If that happens, job J_h is said to suffer *priority inversion*
- In that situation, J_h is said to be *blocked*
  - The blocked state is other than *preempted* or *suspended*
- We would like RTA to contemplate blocking, so that we can continue to use it for FPS
  - But then we must determine a conservative bound B to it

Incorporating blocking in RTA

- $R_i = C_i + B_i + I_i$
  - Where $I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$, and $hp(i)$ is the set of tasks with priority higher than τ_i
  - And $\omega_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^n}{T_j} \right\rceil C_j$ is the recurrence relation that we need to solve
- Let us now look at some priority-inversion situations and the effect of various access control protocols on B_i for any task τ_i, under FPS

Running example

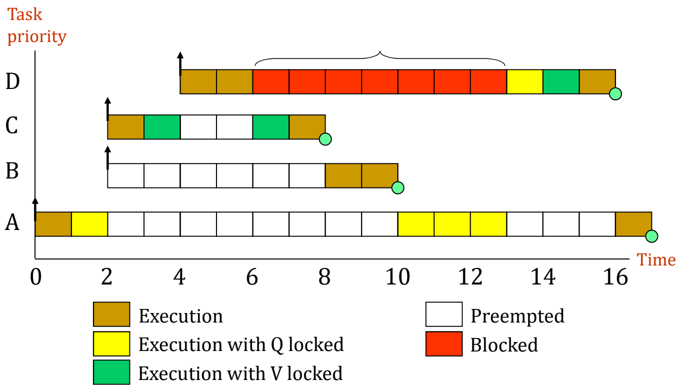
- Consider the task set shown below

Task	Priority	Execution sequence	Release time
A	1 (low)	eQQQQe	0
B	2	ee	2
C	3	eVVe	2
D	4 (high)	eeQVe	4

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

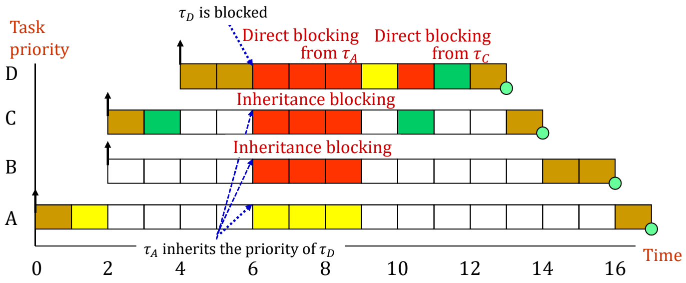
- Let us see how some key access control protocols treat it …

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

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

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 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!

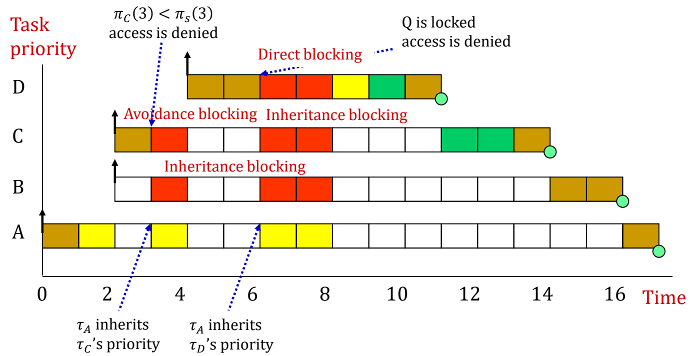
What with Ceiling Priority protocols?

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

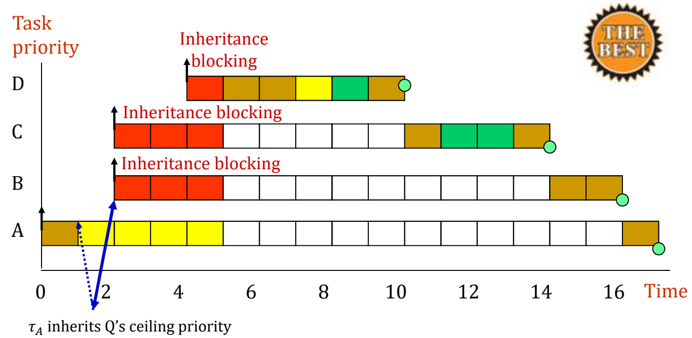
With Basic Priority Ceiling (BPCP)



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 it needs must be free
  - If they were not, then some task would have priority ≥ 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

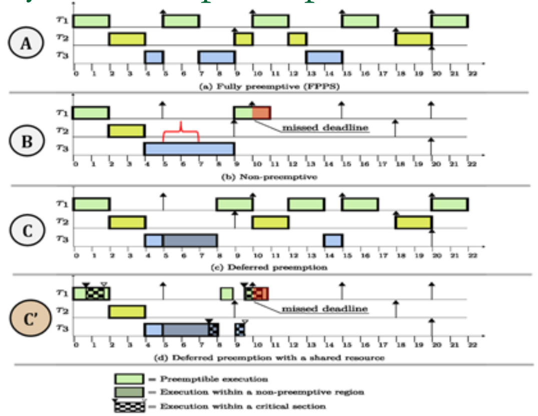
Extending the workload model further

- Our workload model so far contemplates
  - Constrained and implicit deadlines ($D \leq T$), periodic and sporadic tasks, aperiodic tasks under some server scheme, task interactions with blocking factored in the response-time equations
- 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 splits tasks into (*fixed* or *floating*) slots
  - The running task **yield**s the CPU at the end of each such slot
  - If no ***hp*** task is ready, then the running task continues
  - The time duration of any such slot is bounded by B_{max}
  - Mutual exclusion must use non-preemption (else it breaks)
- Deferred preemption has two interesting properties
  - It *dominates* both preemptive and non-preemptive scheduling
  - Each last slot of execution is (obviously) from from interference

Why deferred preemption is clever



Cooperative scheduling /2

- Let F_i be the execution time of the *final slot* of τ_i's execution, naturally exempt from interference

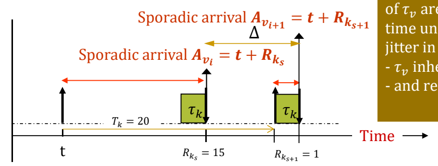
$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- When the response time equation converges (and $w_i^n = w_i^{n+1}$), τ_i's response time is

$$R_i = w_i^n + F_i$$

Release jitter /1

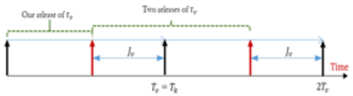
- Most critical for precedence-constrained tasks
- **Example**: a periodic task τ_k with period $T_k = 20$, releases a sporadic task τ_v *at some point* of *some* runs of its (τ_k's) jobs
  - The release is conditional and does not occur at constant time: a perfect example of sporadic activation
- What can we say about the minimum time interval between any two subsequent jobs of τ_v's?



These two subsequent releases of τ_v are spaced by $\Delta = 21 - 15 = 6$ time units instead of $T_k = 20$, owing to jitter in τ_k's response time:
- τ_v inherits τ_k's period T_k
- and release jitter $J_v = R_{k_{max}} - R_{k_{min}}$
$\max(J_v) = R_k - C_k$

Release jitter /2

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

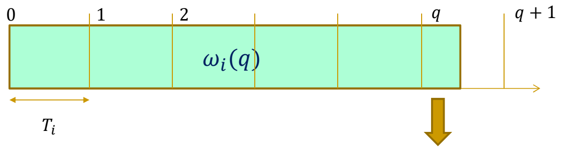


- Higher-priority tasks with release jitter inflict *more* interference
  - The response-time equation must capture that increase potential
    $R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$
- Periodic tasks can only suffer release jitter if the clock is jittery
  - In that case, 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

- To cater for situations where $D > T$, in which, *multiple jobs of the same task compete for execution*, the RTA equation must be modified
  - $\omega_i^{n+1}(q) = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^n(q)}{T_j} \right\rceil C_j$
  - $R_i(q) = \omega_i^n(q) - qT_i$
- The number q of additional releases to consider is bounded by the lowest value of $q : R_i(q) \leq T_i$
  - $\omega_i(q)$ represents the level-i busy period, which extends as long as qT_i falls within it
- The worst-case response time is then $R_i = max_q R_i(q)$

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

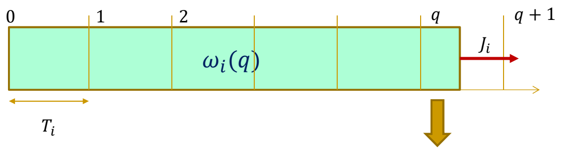
- When the formulation of the RTA equation is combined with the effect of release jitter, two alterations must be made
- First, the interference factor must be increased accordingly

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

- 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 (aka, 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?

Task	T	D	C	O	R
τ_a	8	5	4	0	4
τ_b	20	9	4	0	8
τ_c	20	10	4	10	8

Arbitrary offsets are not tractable with critical-instant analysis, hence we cannot use the RTA equation for them!

In tempo assoluto, τ_c completa a $t = 8 + O_c = 18$

Non-optimal analysis for offsets /2

- Task periods are not entirely arbitrary in reality: they are likely to have some relation to one another
  - In the previous example, two tasks have a common period
  - Then we might give one of them an offset O (tentatively set to $\frac{T}{2}$, as long as $O + D \leq T$) and analyze the resulting system with a transformation that *removes* the offset so that critical-instant analysis continues to apply
- Doing so with the example, tasks τ_b, τ_c (τ_c with $O_c = \frac{T_c}{2}$) are replaced by a single *notional* task with $T_n = T_c - O_c$, $C_n = \max(C_b, C_c) = 4$, $D_n = T_n$ and no offset
  - This technique aids in the determination of a "good" offset
  - The base RTA equation allows offsets, but determining the worst case *with* them is an *intractable problem*.
    - That is why we upper-bound it with the critical instant!

Non-optimal analysis for offsets /3

- This notional task τ_n has two important properties
  - If it is feasible (when 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 lower priority tasks are feasible when suffering interference from τ_n, then they will stay schedulable 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

Notional task parameters

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

Tasks τ_a and τ_b have the same period else we would use $Min(T_a, T_b)$ for 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 more than two 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
- Any such relaxation should *preserve* schedulability
  - Much like what predictability does but for less types of variation
- A sustainable scheduling algorithm does not suffer scheduling anomalies under any such relaxations

Summary

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

Selected readings

- A. Baldovin, E. Mezzetti, T. Vardanega
  Limited preemptive scheduling of non-independent task sets
  DOI: 10.1109/EMSOFT.**2013**.6658596