4.c Task interactions and blocking (recap, exercises and extensions)

Credits to A. Burns and A. Wellings

Task interactions and blocking

- If a task is suspended waiting for a lower-priority task to complete some required computation then the priority model is, in some sense, being undermined
- It is said to suffer *priority inversion*
- If a task is waiting for a lower-priority task, it is said to be *blocked*
  - The blocked state is other than *preempted* or *suspended*

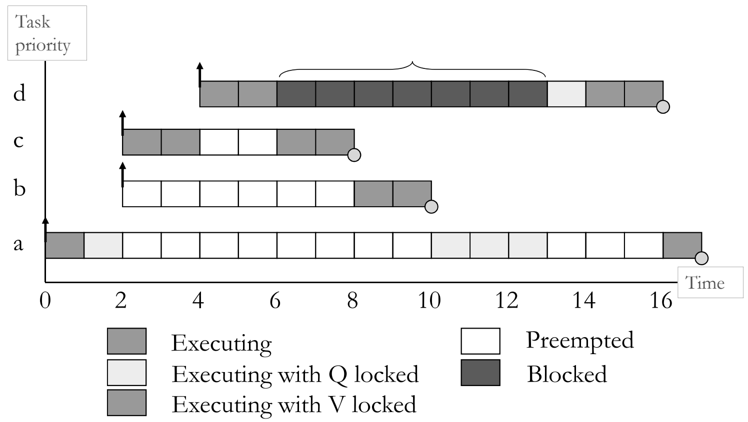
Simple locking and priority inversion /1

- To illustrate an initial example of priority inversion, consider the execution of the task set shown below, under *simple locking* (i.e., by use of binary semaphores)

Task	Priority	Execution sequence	Release time
a	1 (low)	eQQQQe	0
b	2	ee	2
c	3	eVVe	2
d	4 (high)	eeQVe	4

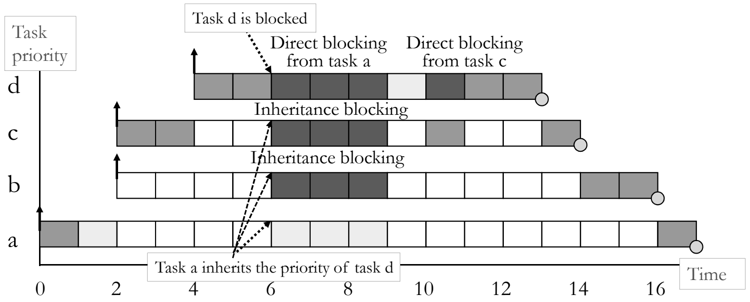
Legend: e: one unit of execution; Q (or V): one unit of use of resource R_q (or R_v)

Simple locking and priority inversion /2



With basic priority inheritance protocol

- 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 the maximum number of times that τ_i can be blocked is K
- The upper bound on the blocking time $B_i(rc)$ for τ_i with K critical sections in the system is given by $B_i(rc) = \sum_{j=1}^{K} use(r_j, i) \times C_{max}(r_j)$
  - $use(r_j, i) = 1$ if r_j is used by at least one task τ_l: $\pi_l < \pi_i$ and one task τ_h: $\pi_h \geq \pi_i$ | 0 otherwise
  - $C_{max}(r_j)$ being 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 <u>all</u> the resources it needs

Incorporating blocking in response time

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

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

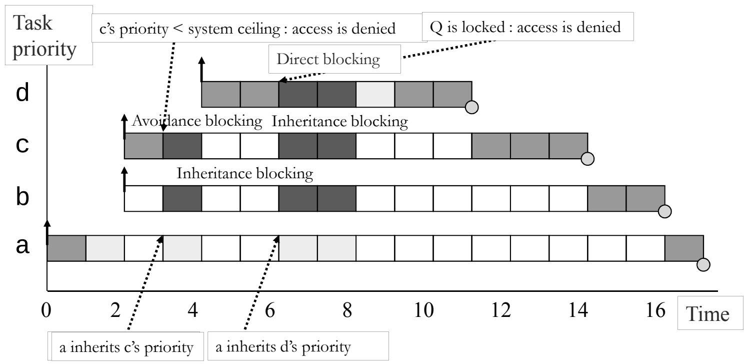
Ceiling priority protocols

- Two variants
  - *Original* CPP (a.k.a. BPCP)
  - *Immediate* CPP (a.k.a. CPP base version)
- When using 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
  - Transitive blocking is prevented by construction
  - Mutual exclusive access to resources is ensured by the protocol itself so that locks are not needed (**!**)

Original CPP (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 that is 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 iff $\pi_i(t) > max_j(\pi_{r_j})$ for all r_j currently locked (excluding those τ_i locks itself) at time t
  - The blocking suffered by τ_i is bounded by the longest critical section with ceiling $\pi_{r_k} > \pi_i$, that is to say:
  - $B_i = max_{k=1,..K}(use(r_k, i) \times C_{max}(r_k))$

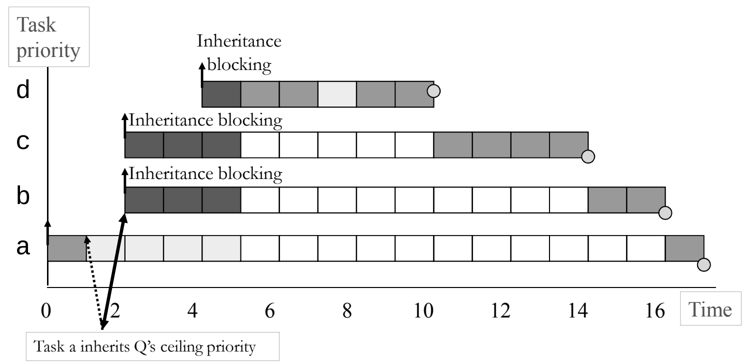
Inheritance with O-CPP



Immediate CPP

- Each task has an assigned *static* priority
  - Perhaps determined by deadline monotonic assignment
- Each resource has a static ceiling attribute defined as the maximum priority of the tasks that may use it
- A task has a *dynamic* current priority at time t that is the maximum of its own static priority and the ceiling values of any resources it is currently using
- Any job of that task will only suffer a block 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 as for O-CPP

Inheritance with I-CPP



O-CPP versus I-CPP

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

An extendible task model

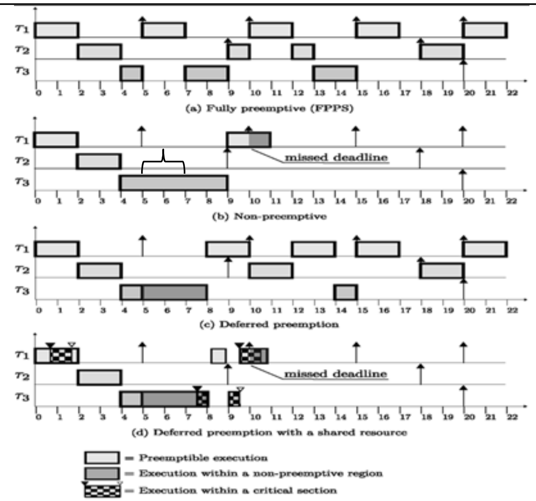
- Our workload model so far allows
  - Constrained and implicit deadlines ($D \leq T$)
  - Periodic and sporadic tasks
    - As well as aperiodic tasks under some server scheme
  - Task interactions with the resulting blocking being (compositionally) factored in the response time equations

Model extensions

- Cooperative scheduling
- Release jitter
- Arbitrary deadlines
- Fault tolerance
- Offsets
- Optimal priority assignment

Cooperative scheduling /1

- Fully preemptive behavior may not be always acceptable for safety-critical systems
- ***Cooperative*** or ***deferred-preemption*** scheduling splits tasks into (fixed or floating) slots
  - The running task calls the scheduler (**yield**) at the end of each slot
  - If no higher-priority task is ready then the task continues into the next slot
  - The time duration of each such slot is bounded by B_{max}
  - Mutual exclusion is realized by non-preemption (else it gets broken)
- The use of deferred preemption has two important benefits
  - It increases system feasibility as it can lead to lower response time values
    - It dominates both preemptive and non-preemptive scheduling (**!**)
  - No interference can occur (by definition) during each last slot of execution



Cooperative scheduling /2

- Let the execution time of the final slot be F_i

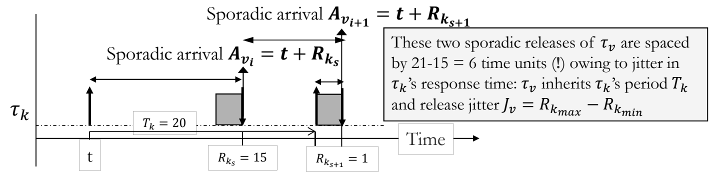
$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- When the response time equation converges, that is, when $w_i^n = w_i^{n+1}$, the response time is given by

$$R_i = w_i^n + F_i$$

Release jitter /1

- A serious problem for precedence-constrained tasks
  - Especially under parallelism (hence in distributed systems and multi-cores)
- **Example**: a periodic task τ_k with period $T_k = 20$ releases a sporadic task τ_v at the end of every run of τ_k's jobs
- What is the interval time between any two subsequent releases of jobs of τ_v?



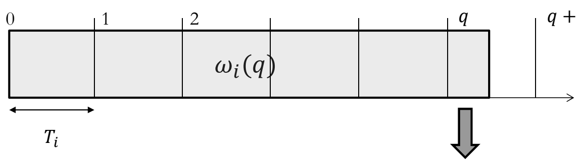
Release jitter /2

- Sporadic task τ_s released at $0, T-J, 2T-J, 3T-J$
- Examination of the derivation of the RTA equation implies that task τ_i will suffer
  - One interference from τ_s if $R_i \in [0, T-J)$
  - Two interferences if $R_i \in [T-J, 2T-J)$
  - Three interferences if $R_i \in [2T-J, 3T-J)$
- Release jitter in higher-priority tasks extends their interference potential: the response time equation captures that as
  $R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$
- 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

- The RTA equation must be modified to cater for situations where D > T in which multiple jobs of the same task compete for execution
  - $\omega_i^{n+1}(q) = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i^n(q)}{T_j} \right\rceil C_j$
  - $R_i(q) = \omega_i^n(q) - qT_i$
- The number q of additional releases to consider is bounded by the lowest value of $q : R_i(q) \le T_i$
  - $\omega_i(q)$ 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

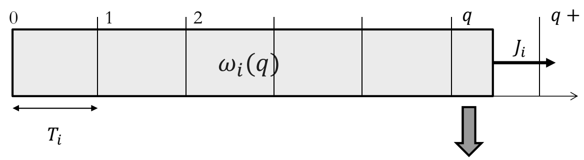
- When the formulation of the RTA equation is combined with the effect of release jitter, two alterations must be made
  - First, the interference factor must be increased if any higher priority tasks suffers release jitter

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

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

$$R_i(q) = w_i^n(q) - qT_i + J_i$$

Arbitrary deadlines /4



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

Offsets

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

Task	T	D	C	R	U=0.9
a	8	5	4	4	
b	20	9	4	8	Deadline miss!
c	20	10	4	16	

- What if we allowed offsets?

Task	T	D	C	O	R
a	8	5	4	0	4
b	20	9	4	0	8
c	20	10	4	10	**8**

Note that arbitrary offsets are not tractable with critical-instant analysis hence we cannot use the RTA equation for it!

Non-optimal analysis /1

- Task periods are not arbitrary in reality: they are likely to have some relation to one another
  - In the previous example two tasks have a common period
  - In this case we might give one of such tasks an offset O (e.g., tentatively set to $\frac{T}{2}$, so long that $O + D \leq T$) and then 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 $O_c = 10$) are replaced by a single *notional* task with $T_n = T_b - O_b = \frac{T_b}{2}$, $C_n = C_b = 4$, $D_n = T_n$ and no offset
  - This technique aids in the determination of a "good" offset
  - The RTA equation on slide 150 shows how to consider offsets , but determining the worst case with them is an intractable problem

Non-optimal analysis /2

- 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 half-period 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 it subsumes

Task	T	D	C	O	R	U=0.9
τ_a	8	5	4	0	4	
τ_n	10	10	4	0	**8**	

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

Priority assignment (simulated annealing)

- **Theorem**: If task p is assigned the lowest priority and is feasible then, if a feasible priority ordering exists for the complete task set, an ordering exists with task p 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;
```


Sustainability [Baruah & Burns, 2006]

- Extends the notion of predictability for singlecore 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 for increase
- A sustainable scheduling algorithm does not suffer scheduling anomalies

Summary

- Completing the survey and critique of resource access control protocols using some examples
- Relevant extensions to the simple workload model
- A simulated-annealing heuristic for the assignment of priorities