3.b Task interactions and blocking

Inhibiting preemption /1

- In certain implementations, some processing actions should not be preempted
  - The execution of *non-reentrant* code shared by multiple jobs, directly (by direct call) or indirectly (e.g., as part of a system call), cannot tolerate preemption
- For reasons of integrity or efficiency, some system-level activities should not be preempted
  - Operations on external devices may not allow preemption
- At its simplest, preemption is inhibited by just disabling dispatching
  - How does that happen?

Inhibiting preemption /2

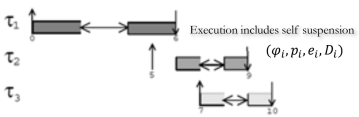
- A higher-priority job J_h that, at its release time, finds a lower-priority job J_l executing with disabled preemption, gets ***blocked*** for a time duration that depends on J_l
  - Under FPS, this is a flagrant case of ***priority inversion***
- The feasibility of J_h now depends on J_l too!
  - Under FPS, this form of blocking for J_i is determined as $B_i(np) = \max_{k=i+1,..,n}(\theta_k)$ where $\theta_k \leq e_k$ is the longest non-preemptible execution of job J_k
  - This cost is paid by of J_i only *once* per release

Self suspension /1

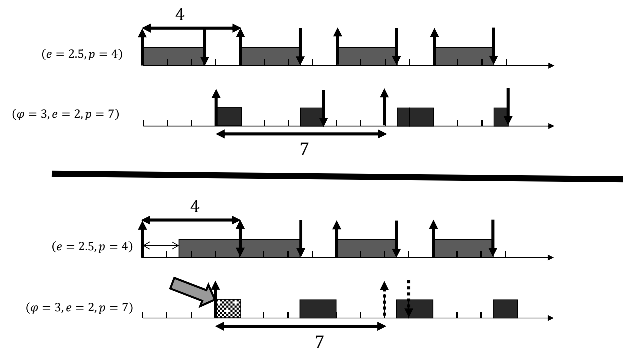
- A job J_i that invokes suspending operations or self suspends, suffers a time penalty that worsens its response time
- J_i incurs a degenerate form of blocking that can be bounded as $B_i(ss) = \max(\delta_i) + \sum_{k=1,..,i-1} \min(e_k, \max(\delta_k))$
  - $\max(\delta_i)$ is the longest duration of self suspension by job J_i
  - The $\sum$ term is the cumulative interference caused by self-suspending higher-priority jobs that become ready during the busy period of J_i
  - Every J_k might in fact resume from self-suspension exactly when J_i does, and therefore interfere each up to $max(\delta_k)$ but never more than e_k
- In general, a job J_i that self suspends K times during execution incurs total blocking $B_i = B_i(ss) + (K+1)B_i(np)$
  - As $B_i(np)$ is potentially incurred at at *every* resumption

Self suspension /2

- Self suspension with independent tasks on single-core processors causes *scheduling anomalies*
  - Deadlines can be missed when task utilization or suspension delays are <u>decreased</u>
- Example: consider a feasible task set under EDF
  - $\tau_1 = \{0,10,(2,2,2),6\}$
  - $\tau_2 = \{5,10,(1,1,1),4\}$
  - $\tau_3 = \{7,10,(1,1,1),3\}$
  - τ_3 would miss its deadline if τ_1's execution or suspension lasted 1 time unit less



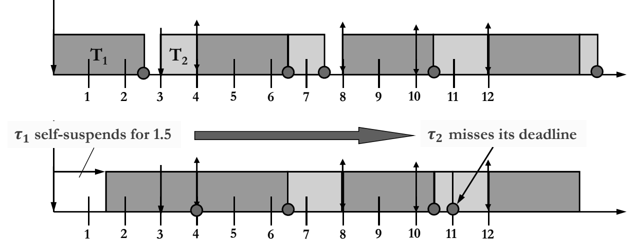
Effects of self suspension /1



Effects of self suspension /2

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

$\tau_1 = \{0, 4, 2.5, 4\}, \tau_2 = \{3, 10, 2, 10\}$ $U = 0.875$



$B_2(ss) = 0 + min(2.5, 1.5) = 1.5$ is a pessimistic upperbound!
With $\varphi_2 = 3$, the actual blocking for τ_2 in [3,10) reduces to 1
But still $B_2(ss) = 1 > \sigma_{2,1}(0) = 0.5$

Access contention

- Access to shared resources causes potential for contention that needs specialized access protocols
- A ***resource access control protocol*** specifies
  - When and on what condition, a resource access request may be granted
  - The order of servicing of such requests
- Access contention situations may cause *priority inversion* to arise (see following examples)

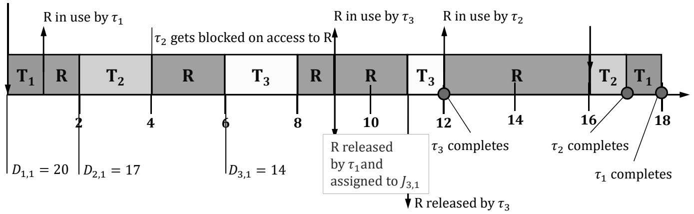
Example /1

$(\varphi_i, p_i, e_i, D_i)$ — Max use of shared resource per execution

τ_1 = {-, -, 2, 20, R(4)}, τ_2 = {2, -, 3, 17, R(4)} , τ_3 = {6, -, 3, 14, R(2)}

under EDF

τ_1 :: e; R(4); e. τ_2 :: e; e; R(4); e. τ_3 :: e; e; R(2); e.



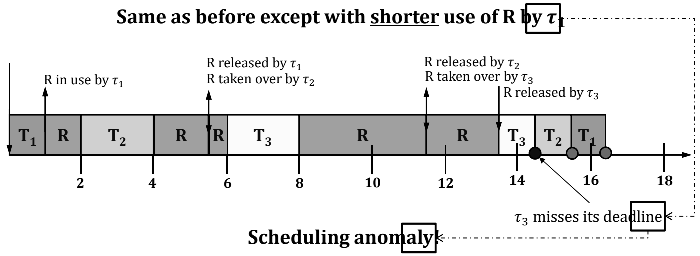
Example /2

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

τ_1 = {-, -, 2, 20, R(2.5)}, τ_2 = {2, -, 3, 17, R(4)} , τ_3 = {6, -, 3, 14, R(2)}

under EDF

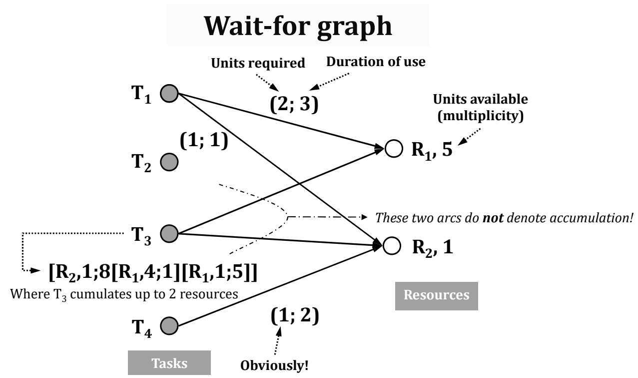
Same as before except with shorter use of R by τ_1



Assumptions and notations

- In order that interference can be minimized, it is preferable for real-time design to prescribe that
  - All jobs do not self suspend (directly or indirectly)
  - All jobs can be preempted
- We say that job J_h is ***directly blocked*** by a lower-priority job J_l when
  - J_l is granted exclusive access to a shared resource R
  - J_h has requested R and its request has not been granted
- To study the problem we may want to use a ***wait-for graph***

Example



Resource access control [option a]

- ***Inhibiting preemption*** in critical sections
  - A job that requires access to a resource is always granted it
  - A job that has been assigned a resource runs at a priority higher than any other job
    - These two clauses imply each other (why?)
    - They jointly prevent deadlock situations from occurring (why?)
- They cause ***bounded*** priority inversion
  - At most once per job
    - We already understood why
  - For a maximum duration $B_i(rc) = max_{k=i+1,..,n} C_k$
    - For job indices in monotonically non-increasing order and C_k denoting worst-case duration of critical-section activity by job J_k

Critique of [a]

- This strategy causes ***distributed overhead***
  - All jobs – including those that do not compete for resource access – incur some time penalty
  - Very unfair hence not desirable
- Better if time overhead is solely incurred by the jobs that do compete for resource access
  - The priority of the job that is granted the resource must only be higher than that of its competitor jobs
    - This is the principle of the *ceiling priority*: we shall return to it
  - The resource requirements must be statically known

Resource access control [option b]

- ***Basic priority inheritance protocol*** (BPIP)
  - The priority of a job varies over time from that initially assigned
  - The variation follows inheritance principles
- **Protocol rules**
  - Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they take on their *assigned priority*
  - Allocation: when job J requires access to resource R at time t
    - If R is free, R is assigned to J until release
    - If R is busy, the request is denied and J becomes *blocked*
  - Priority inheritance: when job J becomes blocked, job J_l that blocks it takes on J's *current priority* as its *inherited priority* and retains it until R is released; at that point J_l reverts to its previous priority

Critique of [b]

- BPIP suffers two forms of blocking
  - ***Direct blocking*** owing to resource contention
  - ***Inheritance blocking*** owing to priority raising
- Priority inheritance is transitive
  - Direct blocking is transitive as jobs may need to acquire multiple resources
- BPIP does not prevent deadlock as cyclic blocking proceeds from transitive direct blocking
- BPIP incurs reducible distributed overhead
  - Under BPIP, a job may become blocked multiple times when competing for more than one shared resource
- BPIP needs no prior knowledge on which resources are shared
  - It is inherently dynamic, hence usable for open systems

Resource access control [option c]

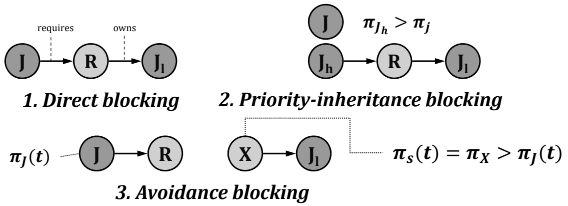
- ***Basic priority ceiling protocol*** (BPCP)
  - As BPIP, but with the additional constraint that all resource requirements must be statically known
  - Every resource R is assigned a *priority ceiling attribute* set to the highest priority of the jobs that require R
    - At time t, the system has a ceiling $\pi_s(t)$ attribute set to the highest priority ceiling of all resources currently in use
    - If no resource is currently in use at t, $\pi_s(t)$ defaults to Ω < the lowest priority of all jobs

BPCP protocol rules

- Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they take on their assigned priority
- Allocation: when job J requests access to resource R at time t
  - If R is assigned to another job, request is denied and J becomes blocked
  - If R is free and J's priority $\pi_J(t) > \pi_s(t)$, the request is granted
  - If J owns the resource with priority ceiling $\pi_s(t)$, the request is granted
  - Otherwise the request is denied and J becomes blocked — Avoidance blocking
- Priority inheritance: when job J becomes blocked by job J_l, J_l takes J's current priority $\pi_J(t)$ until J_l releases all resources with priority ceiling $> \pi_J(t)$; at that point J_l's priority reverts to the level that preceded access to those resources

Critique of [c] /1

- BPCP is *not* greedy (BPIP is!)
  - Under BPCP a request for a free resource may be denied
- Hence BPCP causes each job J to incur **three** distinct forms of blocking caused by lower-priority job J_l



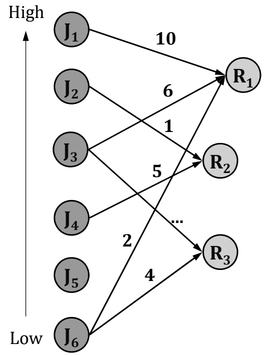
Critique of [c] /2

- ***Avoidance blocking*** is what makes BPCP not greedy and prevents deadlock from occurring
  - If job J at time t has $\pi_J(t) > \pi_s(t)$ then it must be so that
    - J will never use any of the resources in use at time t
    - So won't all jobs with higher priority than J
  - The system ceiling $\pi_s(t)$ determines which jobs can be assigned a resource free at time t without risking deadlock
    - All jobs with priority higher than the system ceiling $\pi_s(t)$
- **Caveat**
  - To stop job J from blocking itself in the attempt of nesting resources, BPCP must grant its request if $\pi_J(t) \leq \pi_s(t)$ but J holds the resources $\{X\}$ with ceiling $= \pi_s(t)$

Critique of [c] /3

- BPCP does not incur reducible distributed overhead because it does not permit transitive blocking
- **Theorem** [Sha & Rajkumar & Lehoczky, **1990**]: under BPCP a job may become blocked for at most the duration of one critical section
  - Under BPCP when a job becomes blocked, its blocking can only be caused by a single job
  - The job that causes others to block cannot itself be blocked
    - Hence BPCP does not permit transitive blocking
  - Demonstration: By exercise
- The maximum possible value of that duration for job J_i is termed the *blocking time* $B_i(rc)$ due to resource contention
  - $B_i(rc)$ must be accounted for in the schedulability test for J_i

Computing the BPCP blocking time /1



Directly blocked by

	J2	J3	J4	J5	J6
J1		6			2
J2			5		
J3					4
J4					
J5					

Priority-inheritance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					4

Avoidance blocked by

	J2	J3	J4	J5	J6
J1					
J2		6			2
J3			5		2
J4					4
J5					

$B_i(rc)$ = max value in row *i* across all tables

Computing the BPCP blocking time /2

- Table "*directly blocked by*" is straightforward
- Table "*priority-inheritance blocked by*"
  - The value in cell [i, k] is the maximum value found in (rows 1, …, i-1; column k) in Table "*directly blocked by*"
- Table "*avoidance blocked by*"
  - If (desirably) jobs are assigned distinct priorities, the cells here are as in Table "*priority-inheritance blocked by*" except for the jobs that do not request resources (whose cell value is set to zero)

Resource access control [option d]

- ***Stack-based ceiling priority protocol***
  - SB-CPP beats BPCP, by allowing stack space to be shared across jobs, thus saving precious memory resources
    - It prevents a job's stack space from fragmenting since it ensures that *none* of the job's request for resources may be denied during execution
    - BPCP instead allows that
- Blocking causes stack fragmentation, not preemption (**!**)
  - One more reason to prescribe that jobs do not self suspend
- SB-CPP also has lower algorithmic complexity in time and space, as it needs less checks against $\pi_s(t)$

SB-CPP protocol rules [Baker, 1991]

- Computation of and updates to ceiling $\pi_s(t)$:
  - When all resources are free, $\pi_s(t) = \Omega$
  - $\pi_s(t)$ is updated any time t a resource is assigned or released
- Scheduling: on its release time job J stays blocked until its assigned priority $\pi_J(t) > \pi_s(t)$
  - Jobs that are not blocked are dispatched to execution by preemptive priority-driven scheduling
- Allocation: whenever a job issues a request for a resource, the request is granted

Critique of [d]

- Under SB-CPP, a job J can only begin execution when the resources it may need are free
  - Otherwise $\pi_J(t) > \pi_s(t)$ cannot hold
- Under SB-CPP, a job J that may get preempted does not become blocked on resumption
  - The preempting job cannot contend resources with J
- SB-CPP prevents deadlock from occurring
- Under SB-CPP, $B_i(rc)$ for any job J_i is computed in the same way as with BPCP

Resource access control [option e]

- ***Ceiling priority protocol*** (base version)
  - CPP does *not* use the system ceiling $\pi_s(t)$ although the resources continue to have a ceiling priority attribute
- Scheduling:
  - A job that does not hold any resource executes at the level of its assigned priority
  - Jobs are scheduled under FPS with FIFO_within_priorities
  - A job that holds any resources has its current priority set to the highest value among the ceiling priority of those resources
- Allocation: Whenever a job issues a request for a resource, the request is granted

Summary

- Issues arising from task interactions under preemptive priority-based scheduling
- Survey of resource access control protocols
- Critique of the surveyed protocols

Selected readings

- L. Sha, R. Rajkumar, J.P. Lehoczky (**1990**)
  Priority inheritance protocols: an approach to real-time synchronization
  DOI: 10.1109/12.57058
- T. Baker (1990)
  A Stack-Based Resource Allocation Policy for Real-time Processes
  DOI: 10.1109/REAL.**1990**.128747