4.b Task interactions and blocking

Inhibiting preemption /1

- In many real-life situations some (fractions of) jobs should not be preempted
  - This is the case e.g. with the execution of *non-reentrant* code shared by multiple jobs whether directly (by direct call) or indirectly (e.g., within a system call primitive)
- Considerations of data integrity or efficiency require that some system-level activities should not be preempted
  - Preemption is inhibited by simply disabling dispatching

Inhibiting preemption /2

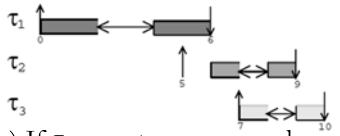
- A higher-priority job J_h that at its release time finds a lower-priority job J_l executing with disabled preemption gets ***blocked*** for a time duration that depends on J_l
  - Under FPS this is a flagrant case of ***priority inversion***
- The feasibility of J_h now depends on J_l too!
  - Under 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 activation

Self suspension /1

- A job J_i that invokes suspending operations or that 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 other term accounts for the <u>cumulative</u> interference from self-suspending higher-priority jobs that may become ready during the busy period of J_i which, for every J_k, can never be $> \max(\delta_k)$ and $> e_k$
- 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

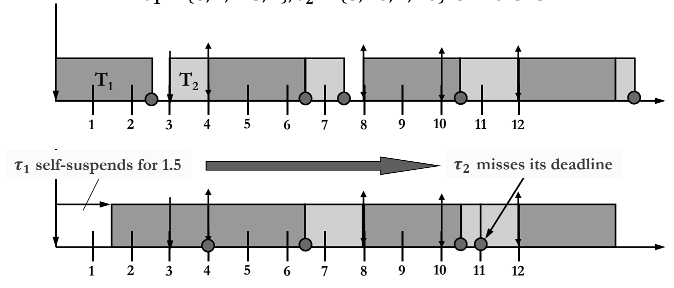
- Self suspension with independent tasks on single-core processors causes *scheduling anomalies*
    - Deadlines can be missed when task utilization or suspension delays are decreased
- Example: a feasible task set with EDF
    - $\tau_1 = \{0, 10, (2,2,2), 6\}$
    - $\tau_2 = \{5, 10, (1,1,1), 4\}$
    - $\tau_3 = \{7, 10, (1,1,1), 3\}$
    - (In red the self suspension) If τ_1 executes or suspends 1 time unit less, then τ_3 misses its deadline



Blocking effects with RMS

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

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



$B_2(ss) = 0 + min(2.5, 1.5) = 1.5$ 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 must be controlled by specialized protocols
- A ***resource access control protocol*** specifies
    - When and under what condition a resource access request may be granted
    - The order in which requests must be serviced
- Access contention situations may cause priority inversion to arise

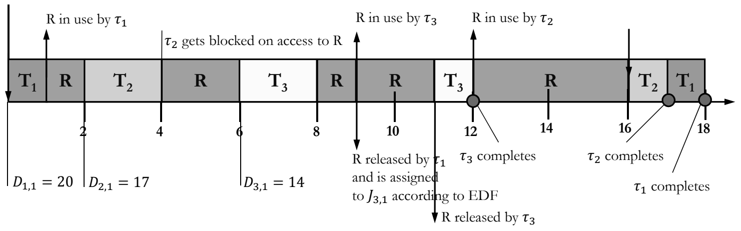
Example /1

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



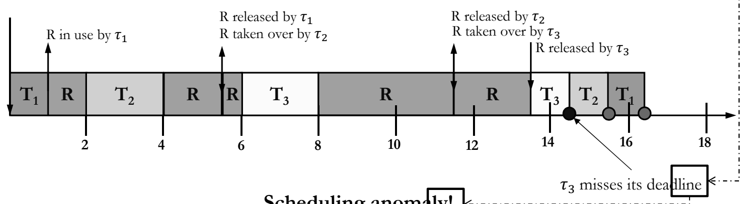
Example /2

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

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 $\boxed{\tau_1}$

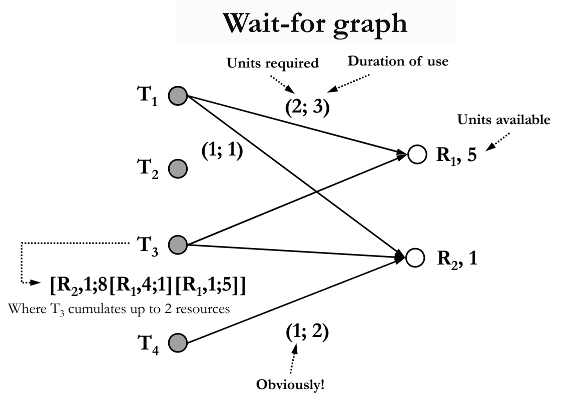


Scheduling anomaly!

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 [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
    - They jointly prevent deadlock situations from occurring
- 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 actually 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 [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 is a devious form of 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

Resource access control [c]

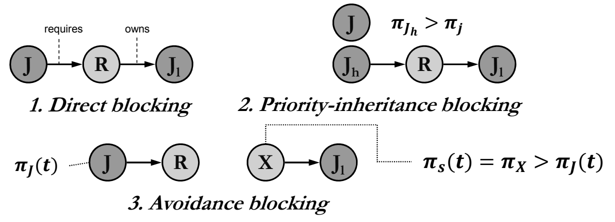
- ***Basic priority ceiling protocol*** (BPCP)
  - As BPIP but with the additional constraint that all resource requirements must be statically known
  - Every resource R is assigned a priority ceiling attribute set to the highest priority of the jobs that require R
    - At time t the system has a ceiling $\pi_s(t)$ 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 $\Omega <$ 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
- Priority inheritance: when job J becomes blocked by job J_l – for direct or avoidance blocking – 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 (whereas BPIP is)
  - Under BPCP a request for a free resource may be denied !
- Hence under BPCP each job J incurs **three** distinct forms of blocking caused by lower-priority job J_l



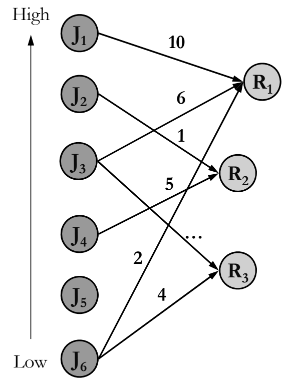
Critique of [c] /2

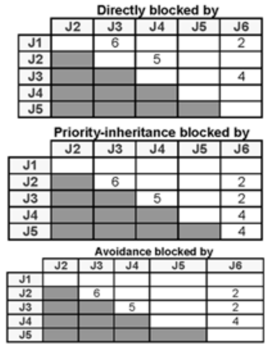
- ***Avoidance blocking*** is what makes BPCP not greedy and prevents deadlock from occurring
  - If job J at time t has $\pi_J(t) > \pi_s(t)$ then it must be so that
    - J will never use any of the resources in use at time t
    - So won't all jobs with higher priority than J
  - The system ceiling $\pi_s(t)$ 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 $\pi_J(t) \le \pi_s(t)$ 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	▨	▨	▨	▨	

$\boldsymbol{B_i(rc)}$ = **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 [d]

- ***Stack-based ceiling priority protocol***
  - SB-CPP beats BPCP in terms of
    - Saving memory resources especially precious to embedded systems by sharing stack space across jobs
      - It prevents a job's stack space from fragmenting because it ensures that none of the job's request for resources may be denied *during execution*
        - What BPCP instead allows
        - Stack fragmentation from blocking and not from preemption (**!**)
      - We must also require that jobs do <u>not</u> self suspend
    - Having lower algorithmic complexity in time and space from needing less checks against $\pi_s(t)$

SB-CPP protocol rules [Baker, 1991]

- <u>Computation of and updates to ceiling</u> $\pi_s(t)$:
  - When all resources are free, $\pi_s(t) = \Omega$
  - $\pi_s(t)$ is updated any time t a resource is assigned or released
- <u>Scheduling</u>: 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
- <u>Allocation</u>: 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 surely does not contend any 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 [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
- <u>Scheduling</u>:
  - 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
- <u>Allocation</u>: 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