5.a Task interactions and blocking

Inhibiting preemption – 1

- In many real-life situations some (parts of) jobs should not be preempted
  - This is typically the case 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 and/or efficiency require that some system level activities must not be preempted
  - Preemption is inhibited by simply disabling dispatching

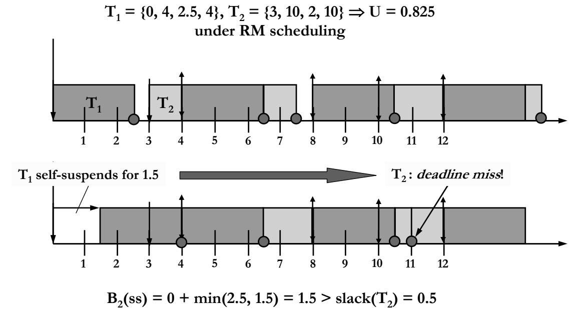
Inhibiting preemption – 2

- 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 $B_i(np)$ time duration
  - Under FPS this is a flagrant case of ***priority inversion***
- The feasibility of J_h now depends on $B_i(np)$ too
  - Under FPS, $B_i(np) = \max_{(i+1,\dots,n)} \theta_k$ where ($\theta_k \leq e_k$) is the longest non-preemptable execution of job J_k
  - This cost is paid by of J_h only *once* per activation

Self suspension

- 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 $B_i(ss) = \max(\delta_i) + \Sigma_{(k=1,.., i-1)} \min(e_j, \delta_k)$
  - Where $\max(\delta_j)$ is he longest duration of self suspension of job J_j and the other term accounts for the cumulative *additional* interference caused by higher-priority jobs that may become ready during the suspension of J_i possibly deferred by their own self-suspension
- For a job J_i that may self suspend K times during execution
  - $B_i = B_i(ss) + (K+1)\, B_i(np)$
  - At every resumption J_i may incur $B_i(np)$ again

Example



$B_2(ss) = 0 + \min(2.5, 1.5) = 1.5 > \text{slack}(T_2) = 0.5$

Access contention

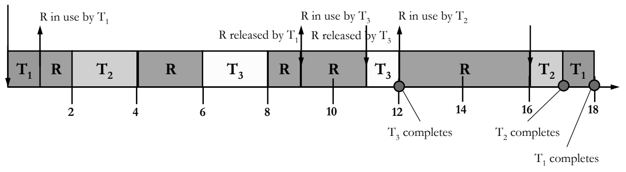
- Access to shared resources causes potential for contention that must be controlled by specialized protocols
- A *resource access control protocol* specifies
  - When and under what condition a resource access request may be granted
  - The order in which requests must be serviced
- Access contention situations may cause priority inversion to arise

Example – 1

Max use of shared resource per execution

T_1 = {-, -, 2, 20, R(4)}, T_2 = {2, -, 3, 17, R(4)} , T_3 = {6, -, 3, 14, R(2)}
under EDF

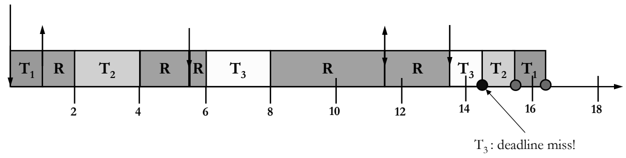
T_1 :: e; R(4); e. T_2 :: e; e; R(4); e. T_3 :: e; e; R(2); e.



Example – 2

T_1 = {-, -, 2, 20, R(<u>2.5</u>)}, T_2 = {2, -, 3, 17, R(4)} , T_3 = {6, -, 3, 14, R(2)}
under EDF

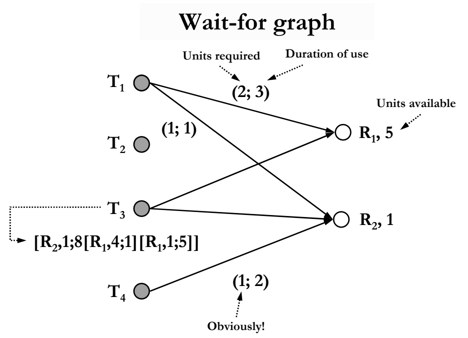
Same as before except with shorter use of R by T_1



Assumptions and notations

- It is safer for real-time design to require 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 – 1

- ***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
    - Reason is obvious
  - 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 worst-case duration of critical-section activity by job J_k

Critique – 1

- 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
    - The principle of the *ceiling priority*: we shall return to it
  - The resource requirements must be statically known

Resource access control – 2

- ***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 – 2

- BPIP incurs 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 (i.e., that can be dispensed with)
  - Under BPIP a job may become blocked multiple times when competing for more than one shared resource
- BPIP does not need to have a-priori knowledge of the shared resources
  - It is inherently dynamic

Resource access control – 3

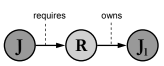
- ***Basic priority ceiling protocol*** (BPCP)
  - As BPIP but with the additional constraint that all resource requirements must be statically known
  - Every resource R is assigned a *priority ceiling* attribute set to the highest priority of the jobs that require R
    - At time t the system has a ceiling $\Pi(t)$ attribute set to the highest priority ceiling of all resources currently in use
    - Otherwise it defaults to $\Omega <$ the lowest priority of all jobs

Resource access control – 4

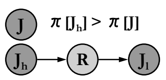
- **Protocol rules**
  - Scheduling: jobs are dispatched by preemptive priority-driven scheduling; at release time they take on their *assigned priority*
  - Allocation: when job J requests access to resource R at time t
    - If R is assigned to another job, request is denied and J becomes blocked
    - If R is free and J's priority $\pi(t)$ is $> \Pi(t)$, the request is granted
    - If J owns the resource with priority ceiling $= \Pi(t)$, the request is granted
    - Otherwise 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 $\pi(t)$ until it releases all resources with priority ceiling $\geq \Pi(t)$; then J_l's priority reverts to the level that preceded resource access

Critique – 3

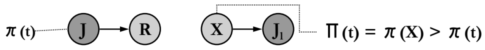
- BPCP is not greedy (whereas BPIP is)
  - Under BPCP a request for a free resource may be denied
- Hence under BPCP each job J incurs three distinct forms of blocking caused by lower-priority job J_l



1. Direct blocking



2. Priority-inheritance blocking



3. Avoidance blocking

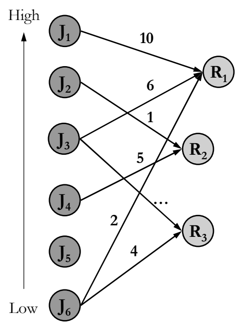
Critique – 4

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

Critique – 5

- 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 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 – 4

- ***(Stack-based) ceiling priority protocol***
  - Improves over BPCP in terms of
    - Saving memory resources especially precious to embedded systems by sharing stack space across jobs
      - Stack-based CPP prevents a job's stack space from fragmenting because it ensures that no job request for resources may be denied *during execution*
        - Which BPCP instead allows
        - Stack fragmentation follows from blocking and not from preemption (!)
      - Of course we must also require that jobs do not self suspend
    - This protocol has lower algorithmic complexity
      - To reduce the run-time overhead in space and time (e.g., from the dynamic computation of the system ceiling)

Ceiling priority protocol – 1

- ***Stack-based* version** [Baker, 1991]
  - Computation of and updates to ceiling $\Pi(t)$: when all resources are free, $\Pi(t)$ evaluates to Ω; the ceiling value is updated any time a resource is assigned or released
  - Scheduling: on its release time a job stays blocked until its assigned priority $\pi(t) > \Pi(t)$
    - 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

Comments

- Under SB-CPP a job can only begin execution when the resources it needs are free
  - Otherwise $\pi(t) > \Pi(t)$ could not hold
- Under SB-CPP a job that may get preempted does not become blocked
  - The preempting job does certainly not share any resources with the preempted job
- SB-CPP prevents deadlock from occurring
- Under SB-CPP $B_i(rc)$ is computed in the same way as with BPCP

Ceiling priority protocol – 2

- ***Base version***
  - CPP does not use the system ceiling $\Pi(t)$ although the resources continue to have a ceiling priority attribute
  - <u>Scheduling</u>:
    - Each job that does not hold any resource executes at the level of its assigned priority
    - Jobs with the same priority are scheduled in a FIFO ordering (`FIFO_within_priorities`)
    - The current priority of a job that holds any resources takes on 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