8.e Global resource sharing

Contention and blocking

- The premises on which single-runner solutions were based fall apart
  - Suspending is no longer conducive to earlier release of shared resource ← parallelism gets in the way
  - Priority boosting the lock holder does not help either ← per-CPU priorities may not have global meaning
  - Having local *and* global resources causes suspending to become dangerous ← local priority inversions may occur
  - Spinning protects against that hazard but wastes CPU cycles

Multiprocessor PCP /1

- Partitioned FPS with resources bound to processors [Sha, Rajkumar, Lehoczky, 1988]
  - The processor that hosts a resource is the *synchronization processor* (SP) for that resource
    - It knows all the use requirements of all of its resources
  - The critical sections of a resource execute on the processor that hosts that resource
    - Jobs that use *remote* resources employ "*distributed transactions*"
  - The processor to which a task is assigned is the *local processor* (LP) for all of the jobs of that task

Multiprocessor PCP /2

- A task may need local and global resources
  - Local resources reside on the local processor of that task
  - Resources are global when their SP differs from the task's LP
- Resource access control protocols need *actual locks* to protect against parallel contention
  - Hence *lock-free algorithms* become attractive again
- SPs use M-PCP to control access to their global resources

Multiprocessor PCP /3

- The task that holds a global lock should not be preempted locally
  - All global critical sections are executed at higher ceiling priorities than local tasks on the SP and any other tasks in the system (this breaks independence!)
- A task τ_h that is denied access to a global shared resource ρ_g <u>suspends</u> and waits in a priority-based queue for that resource
  - Any task τ_l with lower-priority than τ_h on its LP may thus acquire global resources that have higher ceiling

Blocking under M-PCP

- With M-PCP, task τ_i is *blocked* by lower-priority tasks in 5 ways
  - *Local blocking* (<u>once per release</u>): when finding a local resource held by a local lower-priority task that got running as a consequence of τ_i's suspension on access to a global resource
  - *Remote blocking* (<u>once per request</u>): when finding a global resource held by a lower-priority task running on the global resource's SP
  - *Local preemption*: when global critical sections are executed on τ_i's processor by remote tasks of any priority (<u>multiple times</u>) and by local tasks of lower priority (<u>once</u>)
  - *Remote preemption* (<u>once per request</u>): when higher-ceiling global critical sections execute on the SP where τ_i's global resource resides
  - *Deferred interference* as local higher-priority tasks suspend on access to global resources because of blocking effects

Multiprocessor PCP /4

- If the global resource $\rho_{g'}$ being acquired by τ_l resides on the same SP as ρ_g then τ_h suffers an anomalous form of priority inversion
  - The execution in $\rho_{g'}$ delays the release of ρ_g
- As contention for a global resource involves suspension, M-PCP suffers the risk of deadlock
  - With global resources hosted on > 1 SPs, nesting global resources may lead to deadlock and must be disallowed
- This is why other protocols prefer τ_h to spin

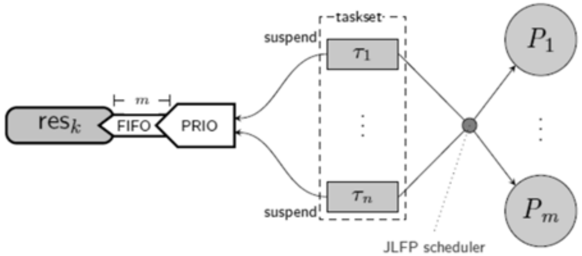
Multiprocessor SRP

- Partitioned EDF with resources bound to processors [Gai, Lipari, Di Natale, 2001]
  - SRP is used for controlling access to local resources
  - Tasks that lock a global resource cannot be preempted
    - They become preemptable again when releasing the resource
  - Tasks that request a global resource that is already locked are held in a FIFO queue on the SP and *spin* on their LP
    - When released by the lock holder, the global resource is assigned to the request at the head of the wait queue

In general …

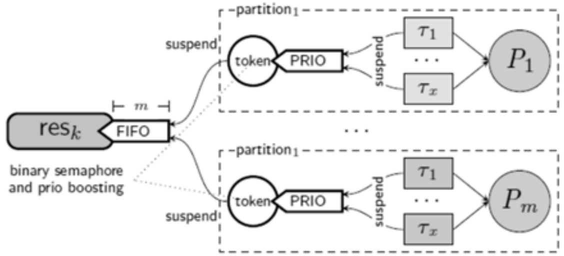
- With lock-based resource control protocols, locks can use either *suspension* or *spinning*
- With suspension, the calling task that cannot acquire the lock is placed in a priority-ordered queue
  - To bound blocking time, priority-inversion avoidance algorithms have to be used
- With spinning, the task busy-waits
  - To bound blocking time, the spinning task becomes non-preemptable and its request is placed in FIFO queue
- The lock owner may also run non-preemptively

$O(m)$ locking protocols : G-sched



- blocking suffered only by tasks using resources
- per-request blocking is $b_k = 2(m-1)\omega_k$, ω_k length of max critical section for res_k
- all resources are global resources

$O(m)$ locking protocols : P-sched

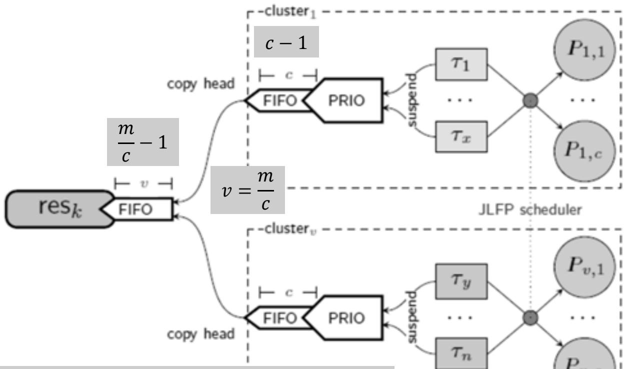


- limiting access to global resources: per-partition *contention token*. Must be acquired before requesting *any* global resource (token + PRIO queue shared for all global resources)
- releasing resources as soon as possible: *priority boosting* for tasks queued in global resources (at most 1 per partition)

Three sources of blocking for P-sched

- *Priority boosting* for earlier release of resource
  - All pay for it as contending tasks may be on any CPU
  - $\beta_i^{boost} = max_k(\omega_k)$
- *FIFO queuing* for the contending tasks
  - $\beta_{i,k} = (m-1)\omega_k$
- *Contention token*
  - Round-robin across CPUs
  - $\beta_i^{token} = (m-1)max_k(\omega_k)$

$O(m)$ independence preservation /1

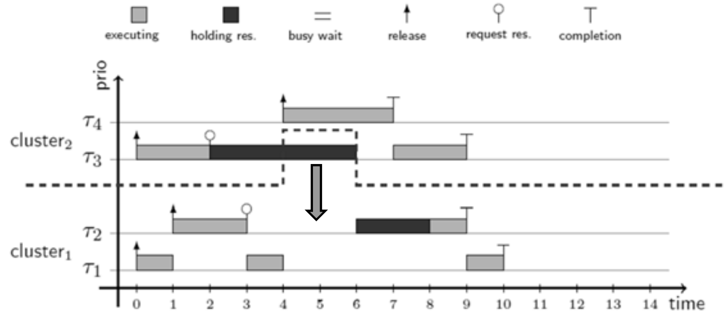


$\beta_{i,k} = \left[\left(\frac{m}{c} - 1\right) + \frac{m}{c} \times (c-1)\right]\omega_k = (m-1)\omega_k$

$O(m)$ independence preservation /2

- Clusters of size $1 \leq c \leq m$
- *Suspension-based*
  - Head of per-cluster FIFO participates in global FIFO
  - The per-cluster queue is FIFO+PRIO
- Independence preserved by inter-cluster migration
  - Head of global FIFO (if pre-empted) can migrate to any CPU along the global FIFO and inherit the priority of a waiting task
- Blocking is *per request*: $\beta_{i,k} = (m-1)\omega_k$

$O(m)$ independence preservation /3



- $t = 3$: task τ_2 suspends and task τ_1 resumes execution
- $t = 4$: task τ_3 migrates to cluster$_1$ and preempts task τ_1

[Brandenburg, 2013]

- **Theorem**
  - Under non-global scheduling (for cluster size $c < m$) it is *impossible* for a resource access control protocol to simultaneously:
    - Prevent unbounded priority-inversion (PI) blocking
    - Be independence-preserving
      - Tasks don't suffer PI blocking from resources that they don't use
    - Avoid inter-cluster job migration
- *Seeking independence preservation and bounded PI-blocking requires inter-cluster job migration* (**!**)

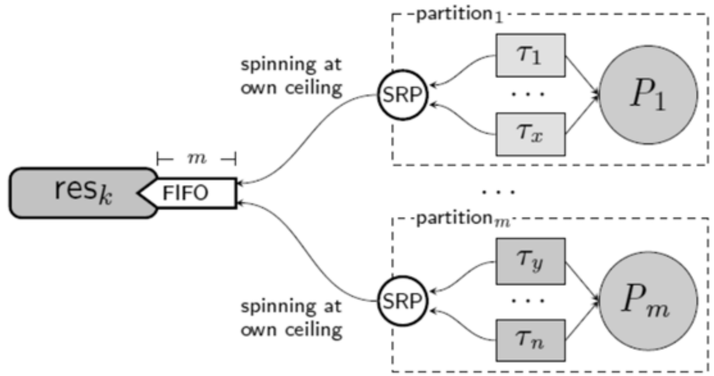
MrsP [Burns, Wellings, 2013] /1

- Want RTA for a partitioned multiprocessor to be *identical* to the single-processor case
  - The cost of accessing global resources should be *increased* to reflect the need to serialize parallel contention
- The property that once a task starts executing, its resources *are* available, is intrinsic to RTA
  - It should therefore be supported by global resource control protocols
  - Cannot live with suspension-based solutions!

MrsP [Burns, Wellings, 2013] /2

- Spinning non-preemptively may decrease feasibility
  - More urgent tasks would suffer longer blocking
- Spinning at the *local* ceiling priority is better
  - With all processors using PCP/SRP, at most one task per processor may contend globally
  - Access requests are served in FIFO order
- To bound blocking from preemption of the lock-holder task, spinning tasks should "donate" their cycles to it
  - Lock-holder job migrates to the processor of a spinning task and runs in its stead until it either completes or migrates again

MrsP [Burns, Wellings, 2013] /3



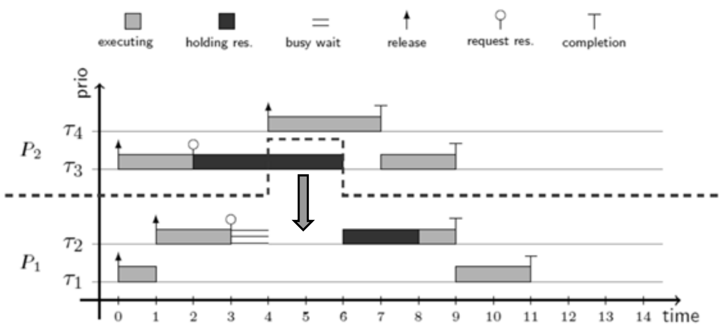
MrsP [Burns, Wellings, 2013] /4

- For partitioned scheduling ($c = 1$)
- *Spinning-based*
  - Local wait spinning at local ceiling
- Allows using uniprocessor-style RTA
- Blocking is *per resource*, increased by parallelism
  - $\beta_i = max_k(\omega_k^{MrsP}) = max_k((m-1)\omega_k) = (m-1) \times max_k(\omega_k)$
- Earlier release obtained by migrating lock holder (if preempted) to the CPU where the first contender in the global FIFO is currently spinning

MrsP [Burns, Wellings, 2013] /5

- Resource nesting can be supported with either *group locking* or *static ordering* of resources
  - With static ordering, resource access is allowed only with order number greater than any currently held resources
  - The implementation should provide an «out of order» exception to prevent run-time errors
- The ordering solution is better than banning nesting and has less penalty than group locking

MrsP [Burns, Wellings, 2013] /6



- $t = 3$: task τ_2 start spinning at ceiling priority
- $t = 4$: task τ_3 migrates to P_1 and executes in place of τ_2

Summary

- Issues and state of the art
- Dhall's effect: examples
- Scheduling anomalies: examples
- P-fair scheduling
- Sufficient tests for simple workload model
- Recent extensions: DP-Fair and RUN
- Incorporating global resource sharing