7.c Global resource sharing

Contention and blocking

- The single-runner premise on which previous solutions were based falls apart
  - Suspending on wait no longer favours earlier release of shared resources ← parallelism gets in the way
  - Boosting the priority of the lock holder does not help either ← per-CPU priorities have no global meaning under partitioned scheduling
  - With local *and* global resources, suspensive wait becomes dangerous ← local priority inversions (PI) may occur
  - Spinning protects against PI, but wastes CPU cycles

Multiprocessor PCP /1

- P-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 statically knows all the use requirements of all of its resources
  - The critical sections of a resource execute on its SP
    - 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 use local *and* global resources
  - Local resources reside on the LP of that task
  - Resources are global when their SP differs from the client tasks' LP
- Resource access control protocols need *actual locks* to protect against parallel contention
  - Which causes *lock-free algorithms* to become attractive
- 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 must execute at higher ceiling priorities than all local tasks on their SP
  - This breaks independence!

- A task τ_h that is denied access to a global shared resource ρ_g <u>suspends</u> on its LP and waits in a priority-based queue for that resource
  - Any task τ_l with lower-priority than τ_h on the same LP may thus acquire global resources on ρ_g's SP, with higher ceiling than ρ_g, thus delaying the progress of τ_h

Multiprocessor PCP /4

- If the global resource ρ_k being acquired by τ_l, resides on the same SP as ρ_g, then τ_h suffers an anomalous form of PI
  - The execution in ρ_k 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

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 locked 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 that it seeks
  - *Local preemption*: when global critical sections are executed on τ_i's LP by remote tasks of any priority (<u>multiple times</u>) and by local tasks of lower priority (<u>once per release</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 SRP

- P-EDF with resources bound to processors [Gai, Lipari, Di Natale, 2001]
  - Normal SRP is used for controlling access to local resources
- Tasks that lock a global resource execute the critical section at the highest local priority
  - As the lock-holder cannot be pre-empted, the wait time is shorter
  - But this provision breaks independence
- Tasks that request a global resource ρ_G already locked, are held in a FIFO queue on ρ_G's SP and *spin* on their LP
  - This policy upper-bounds the requesting task's *spin time* to $m - 1$ executions of the longest critical section of ρ_G
  - This duration adds to the task's WCET

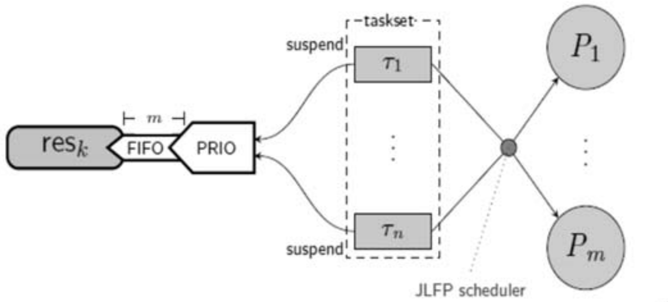
In general …

- With lock-based resource access 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, PI avoidance algorithms should be used
- With spinning, the task busy-waits
  - To bound blocking, the spinning task becomes non-preemptable and its request is placed in a FIFO queue
- The lock holder may also run non-preemptively
  - But this breaks independence

$O(m)$ locking protocols : G-EDF /1

- All resources are global
  - To request a resource, a task must first acquire a general priority-queue, PQ, lock (one of m)
  - If the resource is busy, the requestor waits, *suspending*, on a resource-specific FIFO queue, FQ (of m positions)
  - The lock-holder inherits the highest priority of tasks waiting in the chain of queues (FQ and PQ)
- Per-request blocking is $2m - 1$ executions of the longest critical section for the resource
  - When FQ is full with m lp-jobs and m hp-tasks run (*including the job of interest*) that all want to acccess the same resource
- The other tasks suffer inheritance blocking

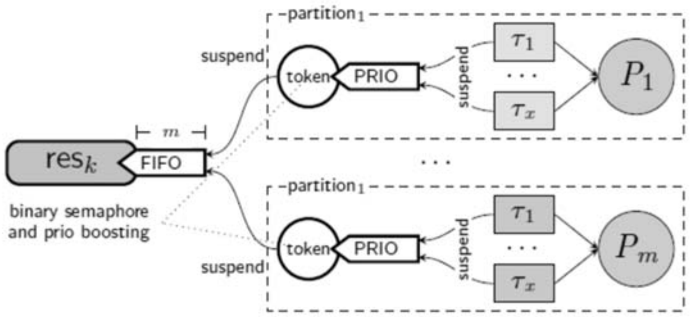
$O(m)$ locking protocols : G-EDF /2



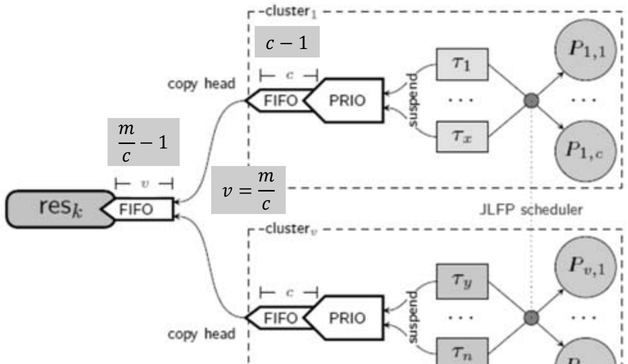
$O(m)$ locking protocols : P-EDF /1

- Shared resources may be local or global
  - One *priority queue* (PQ) per processor: the task at the head of it acquires a *token* to use to contend for global resources
  - Requests for G-resources wait in a per-resource FQ
    - The waiting tasks suspend
  - Lock-holders' priority is inheritance-boosted from their PQ
- Blocking for all tasks has three components
  - *Local*, when the lock-holder is a local lp-task (per release)
  - *Remote direct*, when the requestor is last in the FQ (per request)
  - *Remote transitive*, when a local lp-task has acquired the PQ token and is last of the FQ (per release)

$O(m)$ locking protocols : P-EDF /2



$O(m)$ independence preservation /1

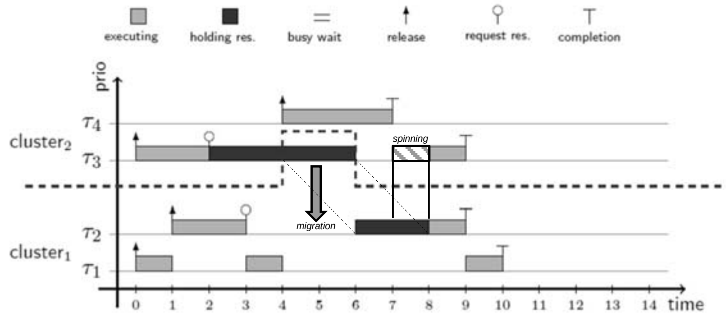


$\beta_{i,k}=\left[\left(\frac{m}{c}-1\right)+\frac{m}{c}\times(c-1)\right]\omega_k=(m-1)\omega_k$

$O(m)$ independence preservation /2

- Clusters of size $1 \leq c \leq m$
  - Global scheduling per cluster, partitioned cluster assignment
- *Suspension-based*
  - One FIFO+PRIO queue per cluster, for $O(m)$ blocking
  - One per-resource global FIFO queue
    - Head of cluster FQ copied in G-FQ and removed only after service
- Independence preserved by *inter-cluster migration*
  - Head of G-FQ (if pre-empted) can migrate to any CPU along the queue (hence across clusters), with priority boosted by inheritance from 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 (with cluster size $c < m$), *no* resource access control protocol can simultaneously
    - Prevent unbounded PI blocking
    - Preserve independence (you don't suffer if you don't contend)
    - Avoid migration
- *Seeking independence preservation and bounded PI-blocking requires inter-cluster job migration* (**!**)

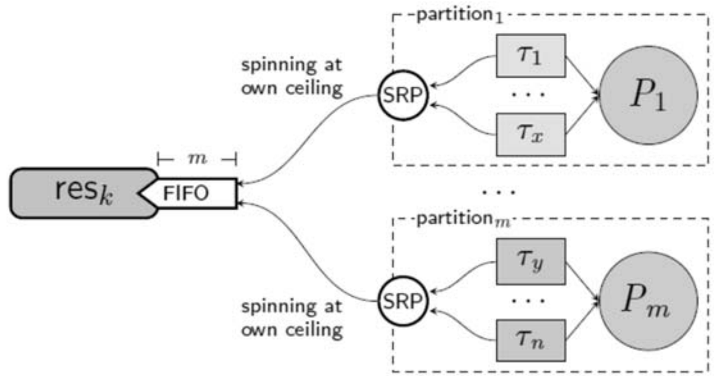
MrsP [Burns, Wellings, 2013] /1

- Rendering RTA for partitioned multiprocessors *identical* to the single-processor case
  - The cost of accessing global resources should be *increased* to reflect the need to serialize parallel contention
- Preserving the property that, once a task starts executing, its resources *are* available
  - It needs global resource control protocols
  - Cannot use suspension-based solutions!

MrsP [Burns, Wellings, 2013] /2

- Spinning non-preemptively may decrease feasibility
  - 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, which assures $O(m)$ blocking
  - Access requests are served in FIFO order
- To bound blocking, spinning tasks "donate" their cycles to the pre-empted lock-holder
  - The lock-holder 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 spins at local ceiling
  - Combined with PCP/SRP, this assures blocking at most once before execution
- Allows using uniprocessor-style RTA
- Wait is *per resource*, increased by parallel contention
  - $\beta_i = max_k(\omega_k^{MrsP}) = max_k((m-1)\omega_k) = (m-1) \times max_k(\omega_k)$
- Earlier release obtained by migrating lock holder (if preempted) to the CPU where the first contender in the global FIFO is currently spinning

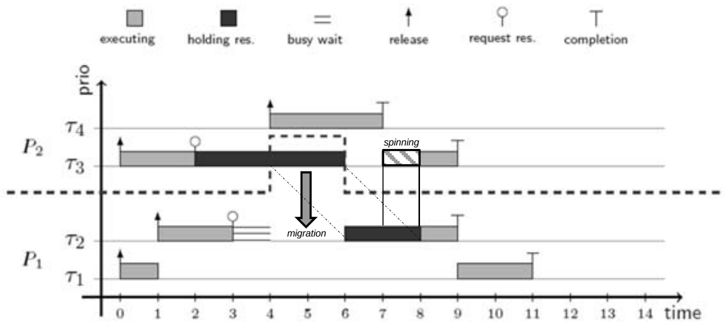
MrsP [Burns, Wellings, 2013] /5

- $R_i = C_i' + B_i + I_i$
- $B_i = max\{\rho_l, b\}$
  - ρ_l is the longest critical section of a resource used by a lower-priority task with ceiling no less than τ_i's priority
  - b is the longest duration of RTOS inhibited preemption
- $I_i = \sum_{j \epsilon hp\mathbf{l}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j'$
- $C_i' = C_i + \sum_j n_i e_j$
  - C_i is task τ_i's WCET outside of critical sections
  - n_i is the number of times task τ_i uses shared resource j
  - $e_j \leq (m-1)\rho_j$, with ρ_j the longest critical section of resource j

MrsP [Burns, Wellings, 2013] /6

- 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
- Recent work has extended MrsP to proper nesting

MrsP [Burns, Wellings, 2013] /7



- $t = 3$: task τ_2 startsspinning 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