7.c Global resource sharing

Where we continue to lift unrealistic workload-model restrictions, allowing tasks to share protected resources across cores under partitioned or global scheduling

Contention and blocking

- Parallelism breaks the *single-runner* premise on which single-core access control solutions rested
  - Suspending on wait does *not* favour earlier release of shared resources because parallelism gets in the way
  - Suspending does *not* stop other tasks, including lower-priority local ones, from making access requests that may cause future priority-inversion (PI) damage
- Spinning helps prevent PI, at the cost of wasting CPU cycles

Multiprocessor PCP /1

- **P-FPS** with strict resource-to-CPU binding [Sha, Rajkumar, Lehoczky, 1988]
  - The processor that hosts a *global* shared resource is called the *synchronization processor* (SP) for that resource
    - All use requirements for shared resources are known statically
  - The protected methods of a resource execute on its SP
  - The processor to which a task is assigned is the *local processor* (LP) for all of the jobs of that task
    - Jobs that call protected methods that are remote, employ "*distributed transactions*" to execute them on the resource's SP

Multiprocessor PCP /2

- A task is permitted to use local *and* global resources
  - Local resources reside on the task's LP, governed by single-processor PCP
  - Resources are global when their SP differs from any client task's LP
- To protect against parallel contention, resource access control protocols need *actual locks*
  - The consequent overhead makes *lock-free algorithms* attractive
- SPs use **M-PCP** to control access to global resources that they host

Multiprocessor PCP /3

- The execution that holds a global lock should *not* be preempted locally
  - All global protected methods that run on an SP must execute at ceiling priorities higher than all tasks local to it
  - This privilege breaks independence! 
- Task τ_h that is denied access to global shared resource ρ_g *suspends* on its LP, and waits in a *priority-based* queue for ρ_g
  - Suspension causes the nesting of global resources that reside on distinct SPs to be liable to circular-wait deadlock
  - This is why other protocols prefer τ_h to spin

Multiprocessor PCP /4

- When τ_h suspends on access to ρ_g, any lower-priority task τ_l local to τ_h's LP, may acquire global resources on ρ_g's SP
  - If those resources had higher ceiling priority than ρ_g, their execution would delay the progress of τ_h further
- This situation causes τ_h to suffer an anomalous form of PI 
  - The action of a lower-priority task (τ_l) delays the release of ρ_g, which in turn delays τ_h longer

Blocking under M-PCP

- With M-PCP, lp tasks may damage task τ_i in many ways!
  1. ***Local blocking*** (*once per release*): when τ_i finds a local resource held by a local lp task that got running as a consequence of τ_i's suspension on access to a locked global resource
  2. ***Remote blocking*** (*once per request*): when τ_i finds a global resource held by a lp task running on the global resource's SP that it seeks
  3. ***Local preemption*** (*multiple times*): when remote tasks of any priority execute global critical sections on τ_i's LP
  4. ***Local preemption*** (*once per release*): when local lp tasks execute global critical sections on τ_i's LP
  5. ***Remote preemption*** (*once per request*): when hp global critical sections execute on the SP where τ_i's global resource resides
  6. ***Deferred interference***: as local hp tasks resume after suspending on access to global resources because of blocking effects

Multiprocessor SRP

- **P-EDF** with shared resources bound to SPs [Gai, Lipari, Di Natale, 2001]
  - Normal stack-based ceiling to control access to *local* resources
- Tasks that lock a *global* resource execute its critical sections at the highest *local* priority
  - The wait time is shorter if the lock-holder cannot be preempted, but this privilege 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 wait time to $\boldsymbol{m-1}$ executions of the longest critical section of ρ_G
  - The spinning time adds to the task's WCET

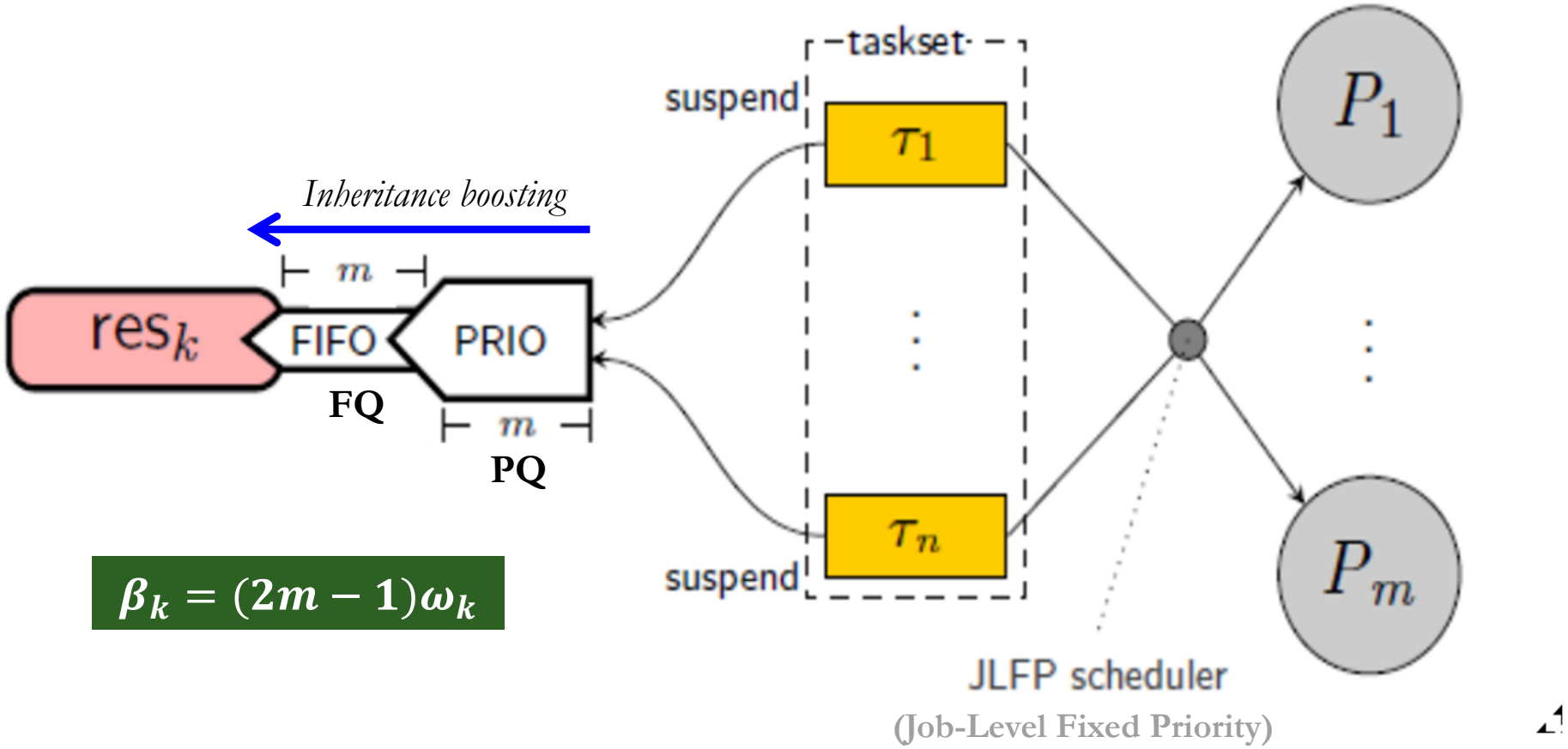
Summing up

- Lock-based resource access control protocols under **partitioned scheduling** may use either *suspension* or *spinning*
- With suspension, the task that cannot acquire the lock, is placed in a priority-ordered queue (FIFO would be better)
  - The use of inheritance boosting may reduce the wait time
- With spinning, the task busy-waits and its request is placed in a FIFO queue attached to the resource
  - Inhibiting preemption of the spinning task reduces wait time, at the cost of breaking independence
- Also the lock holder may enjoy non-preemption
  - Widening the breakage of independence

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

- Global scheduling: shared resources are obviously global
  - Before requesting a resource, a task must acquire one of m general priority-queue, PQ, locks (hence, up to m simultaneous requestors)
  - If the resource is busy, the requestor *suspends* on a per-resource FIFO queue, FQ (of m positions)
  - On preemption, the lock-holder inherits the highest priority of the tasks waiting in the chain of queues (FQ and PQ)
- Worst-case *per-request* blocking is $\boldsymbol{2m-1}$ 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 access the same resource
- The other tasks suffer inheritance blocking

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

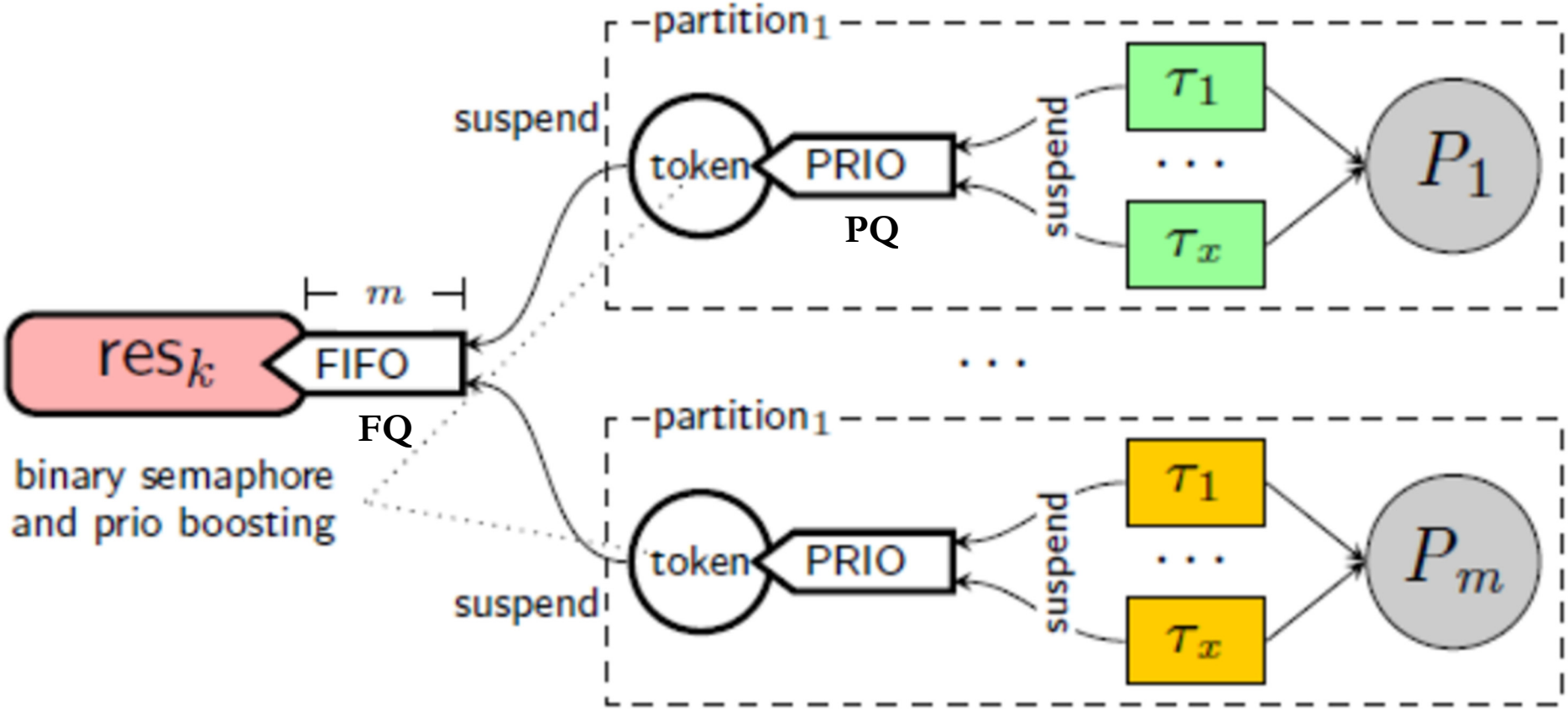


$$\beta_k = (2m - 1)\omega_k$$

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

- Partitioned scheduling: shared resources are local or global
  - One priority queue (PQ) *per processor*: the task at head of PQ acquires a *token* that allows contending for global resources
    - Hence, up to m simultaneous requestors
  - Pending requests for G-resources are held in a *per-resource* FQ
    - The waiting tasks *suspend*
  - On preemption, lock-holders' priority is *inheritance-boosted* from FQ
- Worst-case blocking has three components
  - *Local*, when lock-holder is a local lp-task (*per release*)
  - *Remote direct*, when requestor is last in FQ (*per request*)
  - *Remote transitive*, when a local lp-task has acquired PQ token and is last in FQ (*per release*)

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



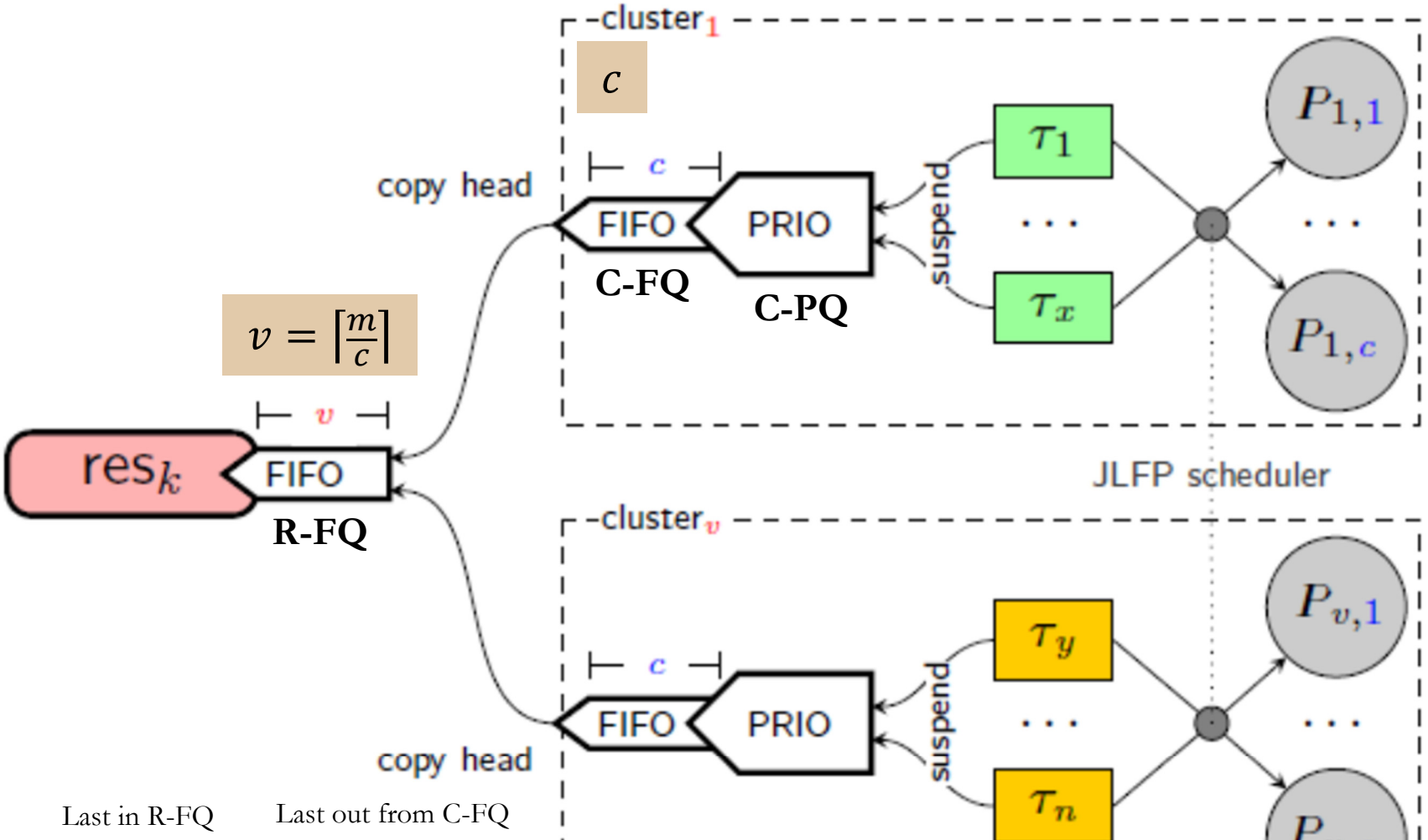
Blocking 1 (*per release*): lock holder is local lower-priority
Blocking 2 (*per request*): requestor is last in FQ
Blocking 3 (*per release*): token holder is local lower-priority and last in FQ

$O(m)$ independence preservation /1

- *Suspension-based*, clusters of size $1 \leq c \leq m$
  - Global scheduling per cluster, partitioned cluster assignment
  - Per cluster: one C-FQ + C-PQ, for $O(c)$ local blocking
  - Per resource: one global R-FQ
    - Head of C-FQ is *copied* in R-FQ and removed only *after* service
    - R-FQ contains ≤ 1 request per cluster, for $O\left(\frac{m}{c}\right)$ global blocking
- Independence is preserved by *inter-cluster migration*
  - Preempted lock-holder (head of R-FQ) can migrate with inheritance boosting to another cluster and CPU along the R-FQ
- Worst-case *per-request* blocking occurs when requestor is last in R-FQ and last out from C-FQ

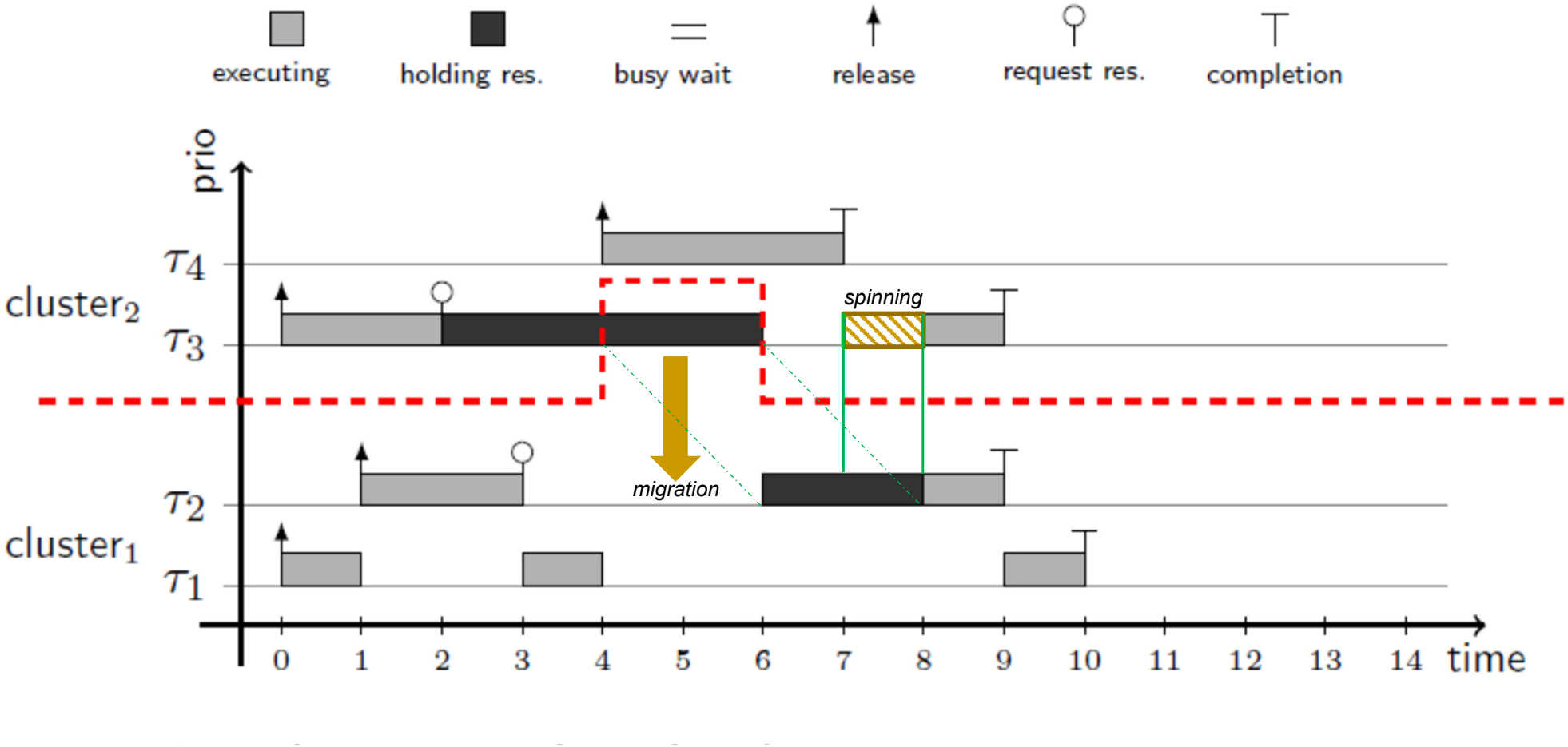
$$\boldsymbol{\beta_{i,k} = \left[\left(\frac{m}{c} - 1\right) + \frac{m}{c} \times (c-1) + \right] \omega_k = (m-1)\omega_k}$$

$O(m)$ independence preservation /2



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

$O(m)$ independence preservation /3



- $t = 3$: τ_2 suspends and τ_1 resumes execution
- $t = 4$: τ_3 migrates to cluster$_1$ where it impersonates τ_2 and preempts 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 (if you don't contend you are left alone)
    - Avoid migration
- Seeking independence preservation and bounded PI-blocking **requires** inter-cluster job migration (**!**)

MrsP [Burns, Wellings, 2013] /1

- Goal 1: allow global resources with P-FPS, with an RTA *identical* to the single-processor case
  - The cost of accessing global resources should be inflated to reflect serialization of parallel contention
- Goal 2: preserve the single-processor PCP benefit that, when a job starts running, all the resources that it may use, should be available
  - This requires *spinning* because suspending on wait would wreak havoc

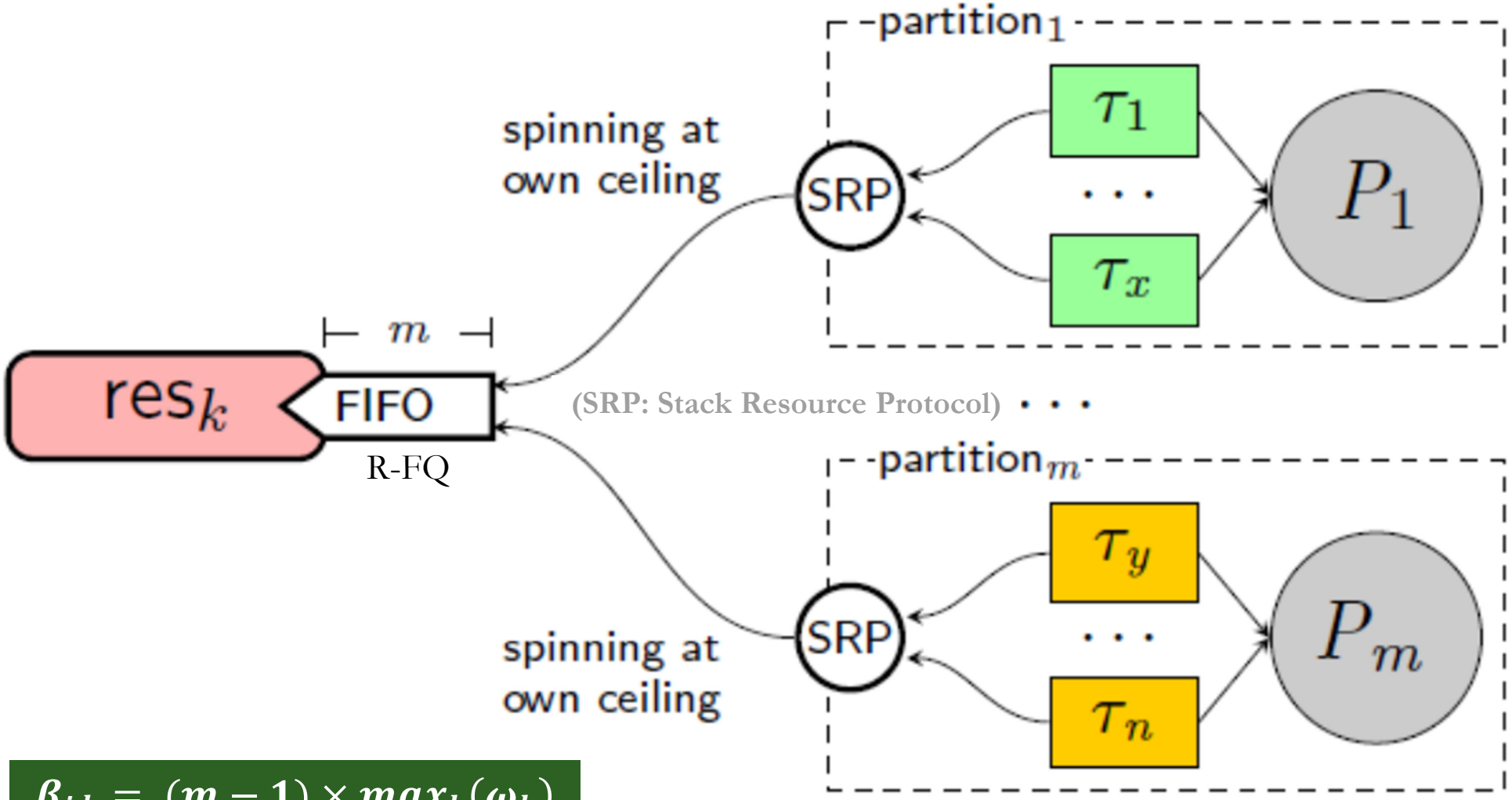
MrsP [Burns, Wellings, 2013] /2

- Spinning at the highest local priority may delay local urgent tasks and thus decrease overall feasibility
- Spinning at the *local* resource ceiling priority is better
  - With all cores using PCP, *at most one task per core* may contend globally, which assures $O(m)$ global blocking
  - Requests are served in global R-FQ
- To bound blocking, spinning tasks "donate" their cycles to preempted lock-holder
  - Lock-holder migrates to the processor of a spinning task and runs in its stead until lock release or another migration

MrsP [Burns, Wellings, 2013] /3

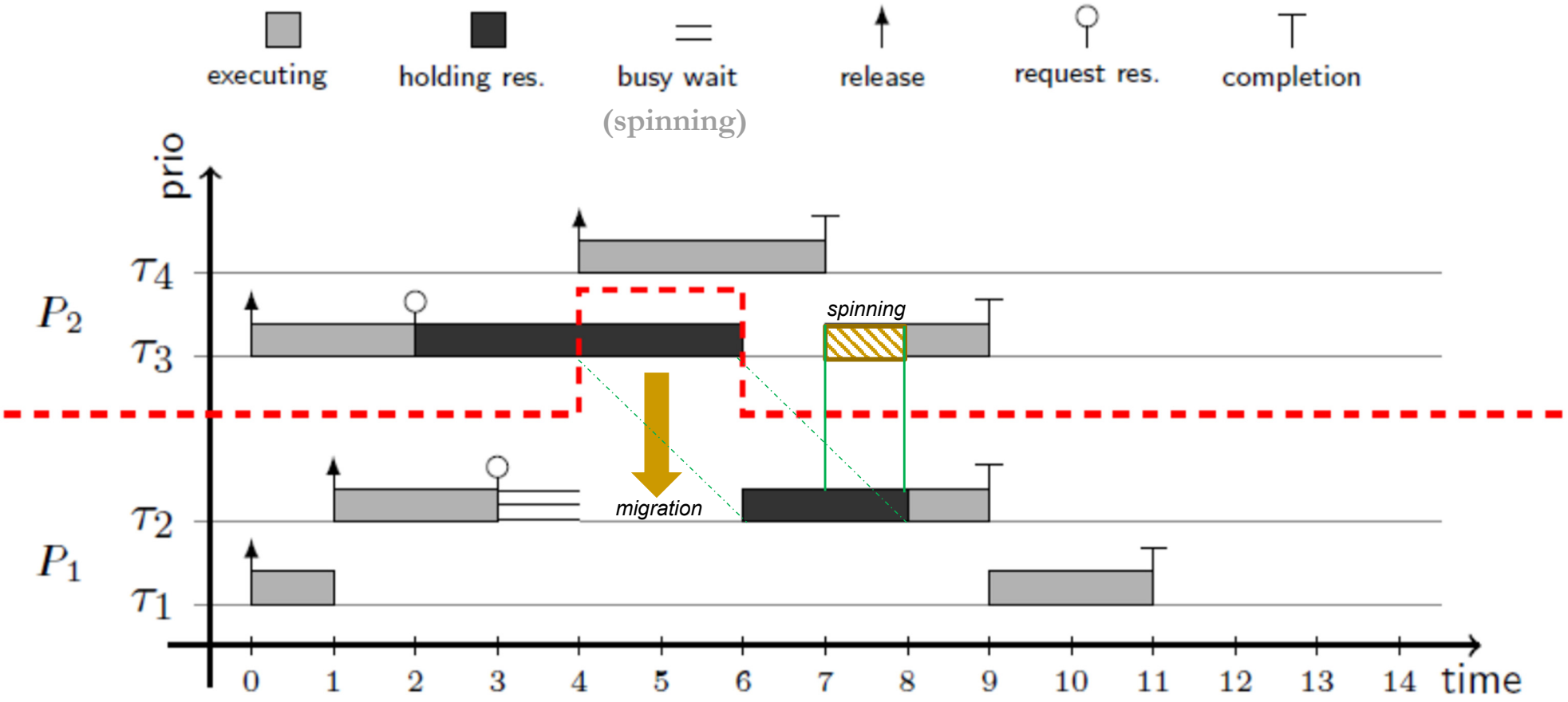
- Partitioned scheduling ($c = 1$), with spinning at local ceiling when waiting for global resource
  - Combined with PCP, this assures local blocking *at most once* before execution, which allows using canonical RTA
- Worst-case wait $\boldsymbol{\beta_{i,k} = (m - 1) \times max_k(\omega_k)}$
  - Computed across resources used by lp tasks with ceiling not inferior to τ_i's priority
- Cost of spinning adds to task's WCET
- Earlier release of resource obtained by migrating pre-empted lock holder to the CPU where the first contender in the global R-FQ is currently spinning

MrsP [Burns, Wellings, 2013] /4



$$\beta_{i,k} = (m-1) \times max_k(\omega_k)$$

MrsP [Burns, Wellings, 2013] /5



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

MrsP [Burns, Wellings, 2013] /6

- $\boldsymbol{R_i = C'_i + B_i + I_i}$
- $\boldsymbol{B_i = max\{\omega_l, b\}}$
  - ω_l is the longest critical section of resource ρ_l used by a lp task with ceiling no less than τ_i's priority
  - b is the longest duration of kernel's run with inhibited preemption
- $\boldsymbol{I_i = \sum_{j \epsilon hpl(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C'_j}$, *local* interference only
- $\boldsymbol{C'_i = C_i + \sum_j n_i e_j}$, must include spinning
  - C_i is τ_i's WCET (*except for spinning*)
  - n_i is the number of times τ_i uses shared resource ρ_j
  - $e_j \leq (m-1)\omega_j$, with ω_j the longest critical section of ρ_j

MrsP [Burns, Wellings, 2013] /7

- 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

Summary

- Various solutions exist to enable the sharing of global resources with either partitioned or global scheduling
  - The associated overhead is often very high
- Parallel contention calls for the use of actual locks and requires either suspension or spinning
  - Neither is very satisfactory
- We have seen that spinning *and* migration can provide the best (least-bad?) solution

Selected readings

- B. Brandenburg (**2013**)
  A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications
  DOI: 10.1109/ECRTS.2013.38
- A. Burns, A. Wellings (**2013**)
  A Schedulability Compatible Multiprocessor Resource Sharing Protocol – MrsP
  DOI: 10.1109/ECRTS.2013.37
- S. Zhao et al. (**2017**)
  New schedulability analysis for MrsP
  DOI: 10.1109/RTCSA.2017.8046311