Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

Proxy Execution

Reducing Complexity and Finding a Path to Upstream

John Stultz <jstultz@google.com>

Quick Background

Previous Talks/Papers

- Watkins, Straub, Niehaus (RTLWS11)
- Peter Zijlstra (RTSumit17)
- Juri Lelli (2018 patchset, OSPM19)
- Valentin Schneider (LPC20 slides)
- Me (w/ special thanks to Connor O'Brien) (OSPM23)

Why do we care?

- Enforce priority between Foreground/Background tasks
- Classic solutions: RealTime Priority -> Priority Inversion -> Priority Inheritance
- Android apps can't generally use RT priorities safely
- Instead mix of cgroups and nice values used to prioritize Foreground apps
- Hit lots of priority inversion issues! - not unbounded, but longer then we like
- Priority Inheritance doesn't work for SCHED_OTHER
- As a result, we cannot usefully limit background activity without introducing inconsistent behavior

Proxy Execution

Simple Idea:
- Track blocked_on relationship of mutex waiters to owners
- Keep mutex blocked tasks on runqueue!
- Treat the scheduler like a black box: It selects the most important task to run.
- If we select a mutex blocked task to run, follow the blocked_on chain and run the unblocked owner

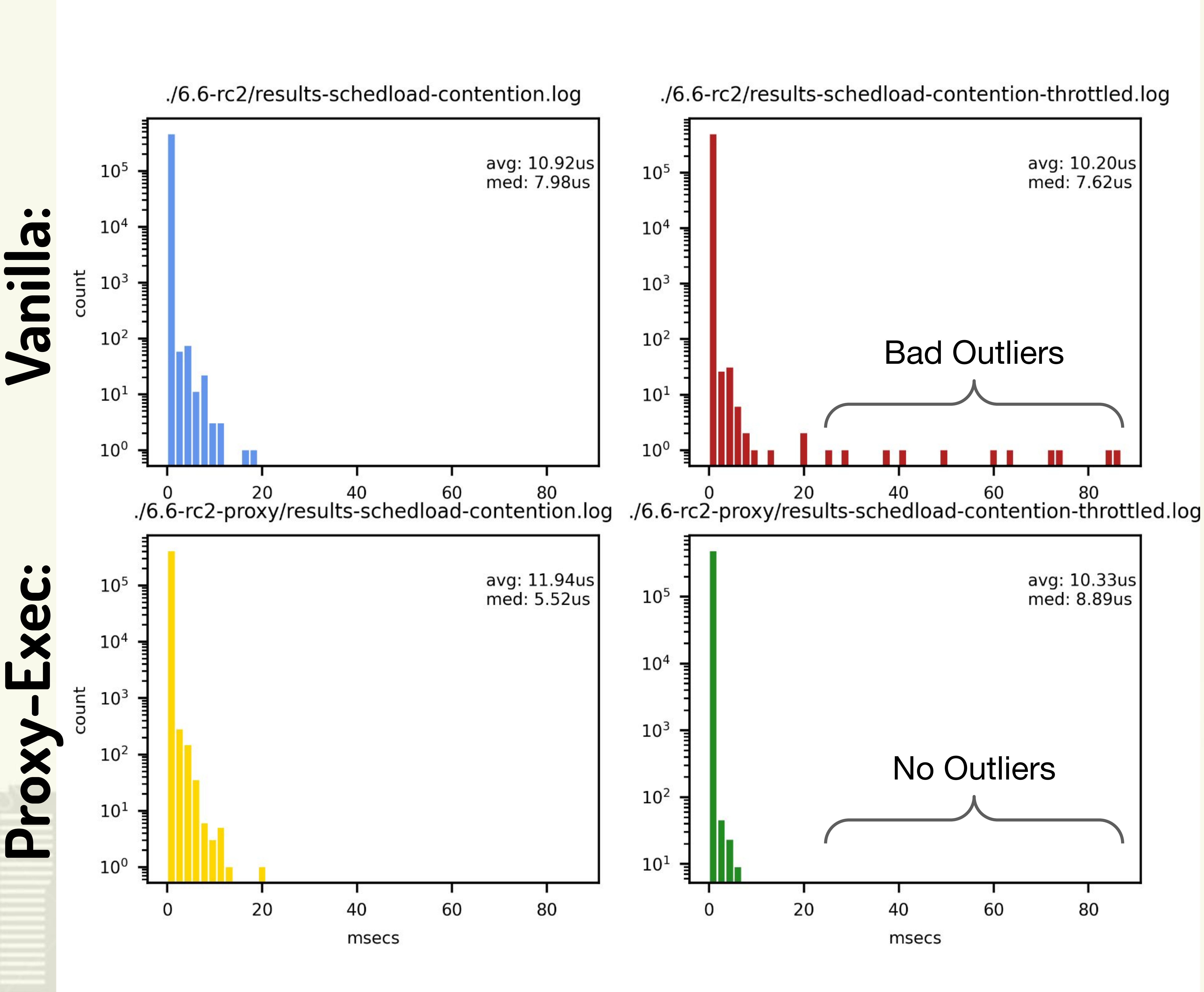
But it gets complex:
- blocked_on chains can cross CPUs run-queues
  - -> Migrate blocked task to the runnable owner's CPU
- Chains might resolve to sleeping owners that can't run.
  - -> Enqueue blocked task on sleeping owner task, to wake with owner
- ... and more!

On the left: We test how long it takes to do many file renames in a directory. We do this in two parallel tasks to create contention on fs locks. We also run NRCPU busyloop tasks.

On the right: We re-test with CPU share limiting so one of the file rename tasks is very limited, and set the busy loop tasks to moderate limits. Leaving one of the rename tests unlimited.

With **Vanilla kernels** the average improves slightly with share limiting. But we see bad outliers as a result of priority inversion on fs locks.

With **Proxy-Exec**, we see much more deterministic output as we avoid priority inversion.



https://github.com/johnstultz-work/priority-inversion-demo

Recent Work (Since OSPM - April)

- v4: Attempt to resolve ww_mutex circular blocked_on references
  - However, still ran into rq confusion crashes (more on this)
  - Minimal feedback
- v5: Tearing the patch apart into fine grained bisectable steps
  - Lots of rework and fixes!
  - Return-migration rework - lock ordering trouble
  - Missing 2 parts from v4: chain migration, and sleeping owner enqueuing
  - Introduced performance regression :(
  - Minimal feedback
- v6: Stabilizing sleeping owner enqueueing
  - Focus on trying to fix sleeping owner enqueueing
  - Conditionalized logic on a boot flag
  - A few fixes for problems I introduced in v5's rework
  - Reduced performance regression vs v4
  - Cleanups and fixes from feedback

Current Issues (Summary)

- Sleeping owner enqueuing is difficult to get right
  - List/chains of tasks on a task (are we recreating runqueues?)
  - Mid-chain wakeups (from ww_mutexes)
- Return migration approach from __schedule()
  - Slow but correct
  - Need thoughts on how to avoid locking mess
- Sorting out perf regression since v4
- Limitations with cross-runqueue chains
  - How to allow for better optimizations?
- Scheduler is already terribly subtle, adding more complexity is a concern

Discussion

- Practical questions:
  - How fine grained do folks want patches?
  - Do we need to ship it first?
    - Want to avoid more Android divergence.

- Design questions:
  - Ways to minimizing lock juggling:
    - Keep having the right types of locks, but for the wrong objects
  - Thoughts for avoiding "swimming upstream" of the locking-order?

- **A request**: Reviews for Design & Correctness
  - https://sage.thesharps.us/2014/09/01/the-gentle-art-of-patch-review/

Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

Thank You!

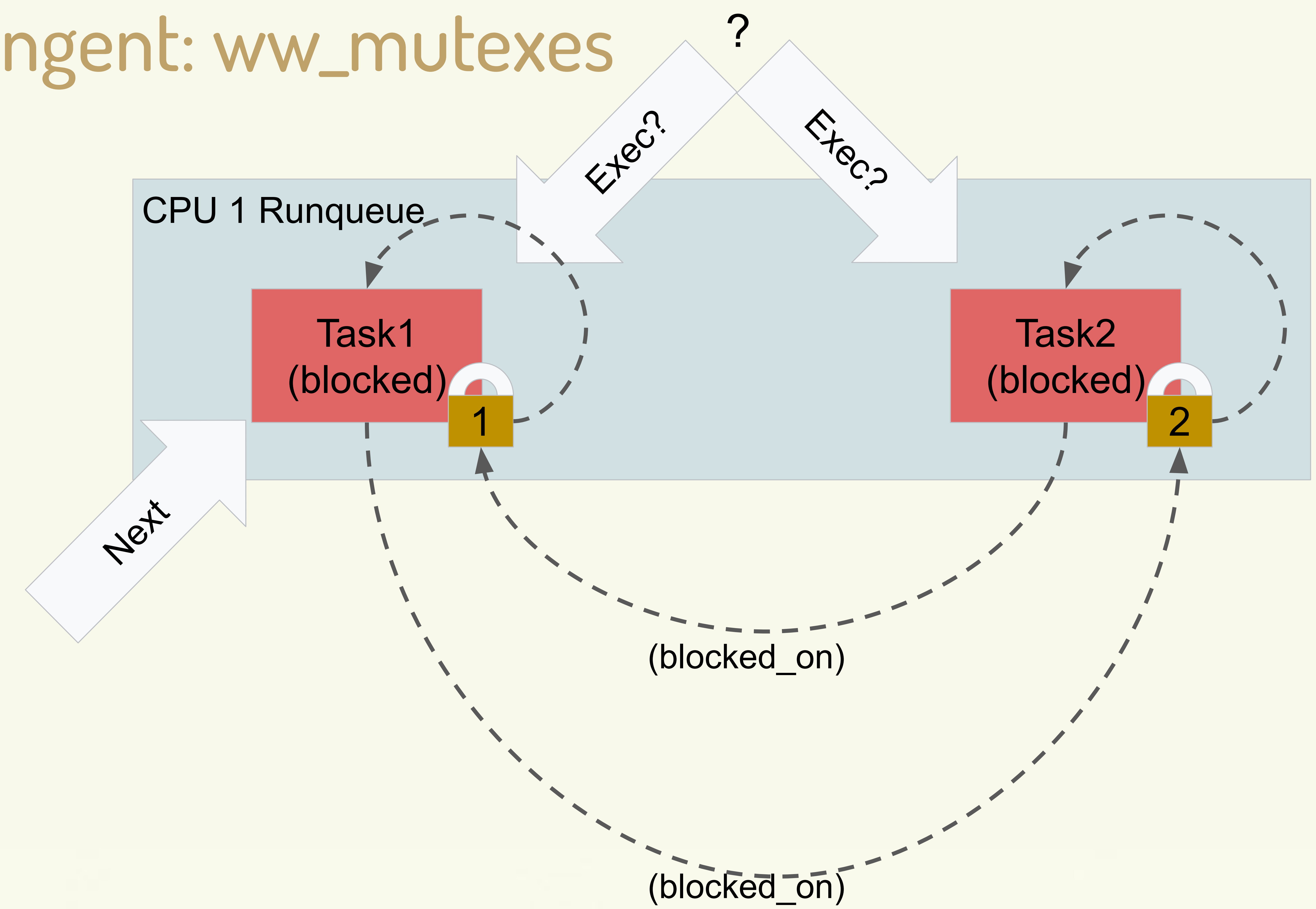
John Stultz <jstultz@google.com>

Current/Recent Issues

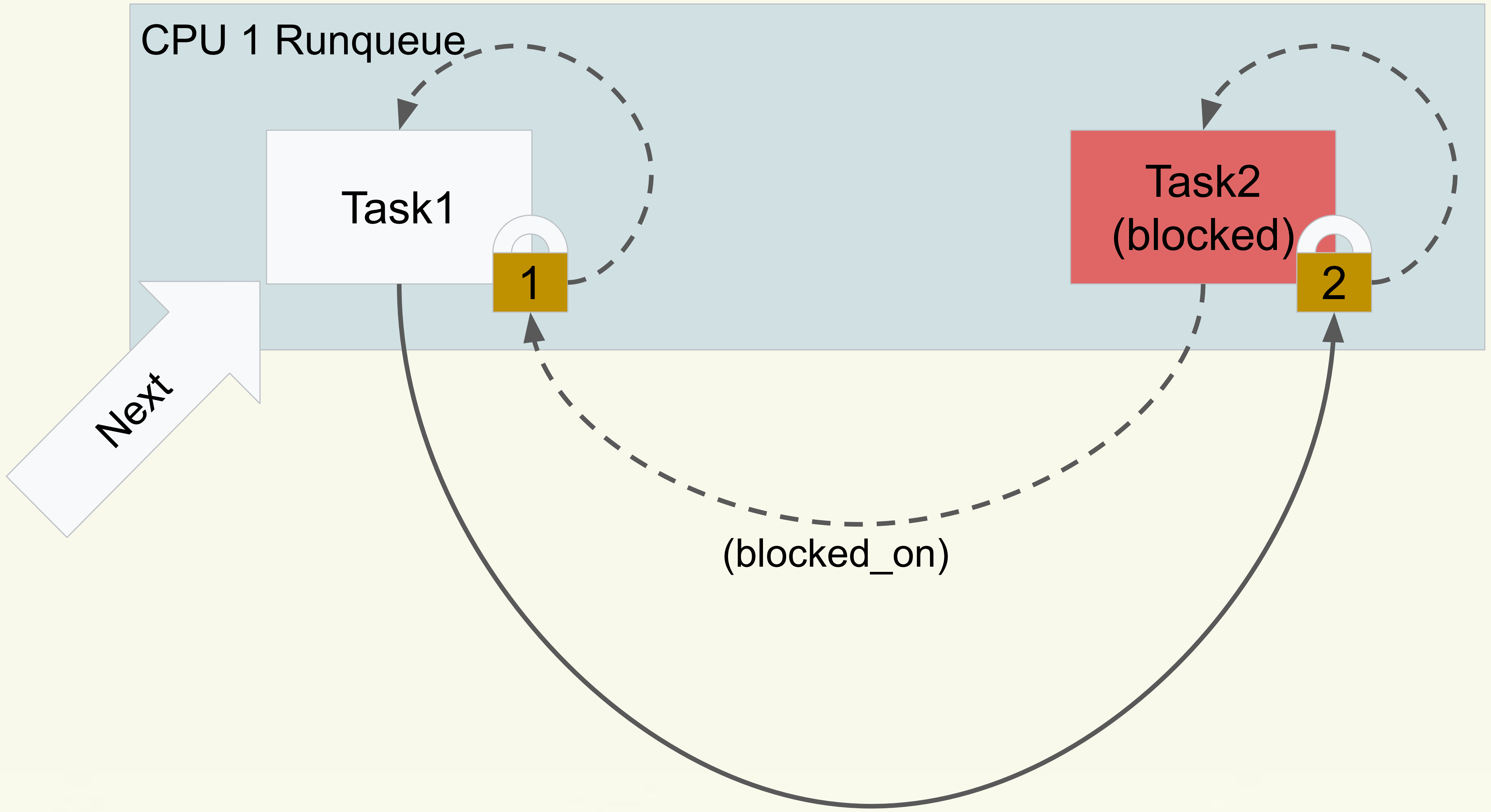
(backup slides)

Sleeping Owner Enqueuing Troubles

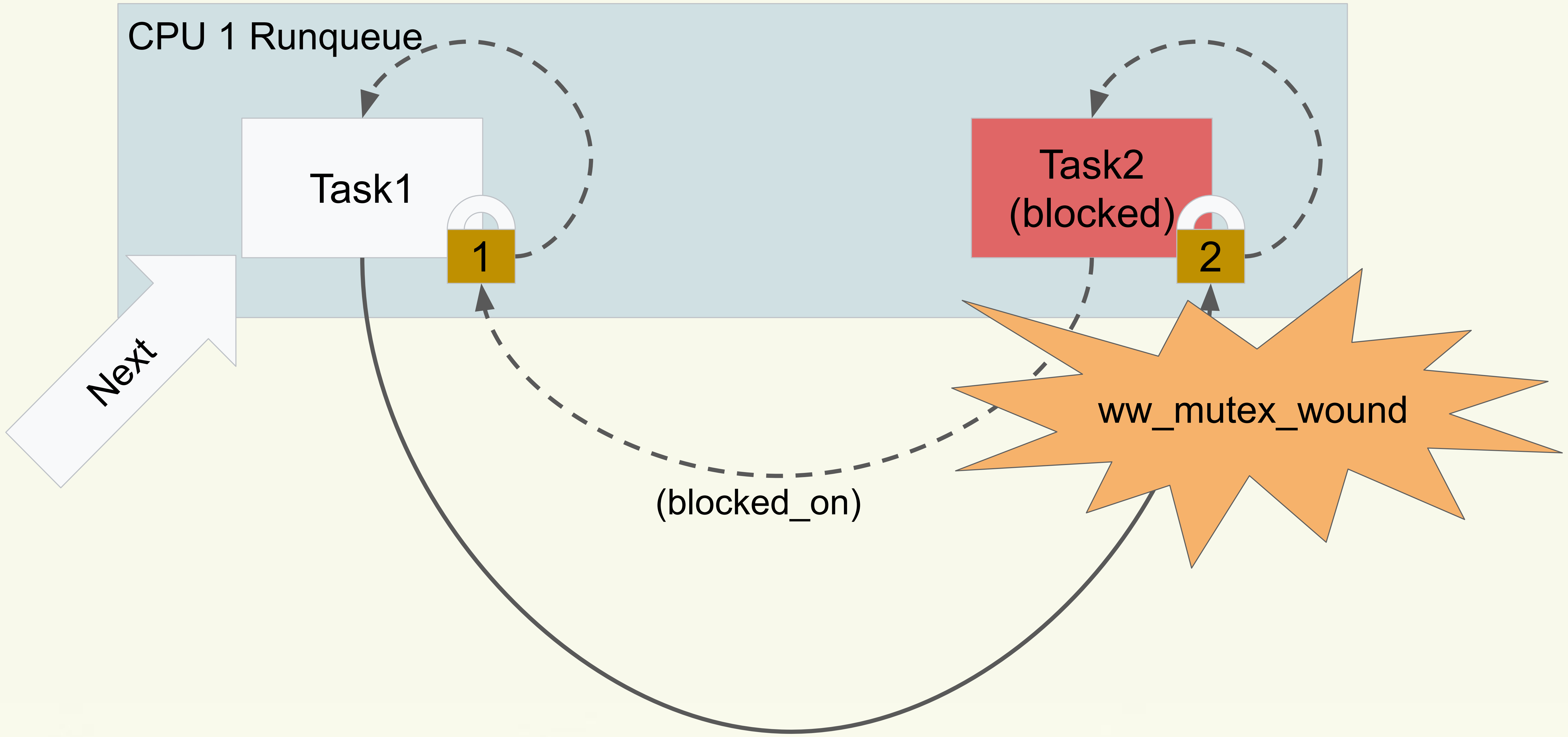
Tangent: ww_mutexes



Tangent: ww_mutexes

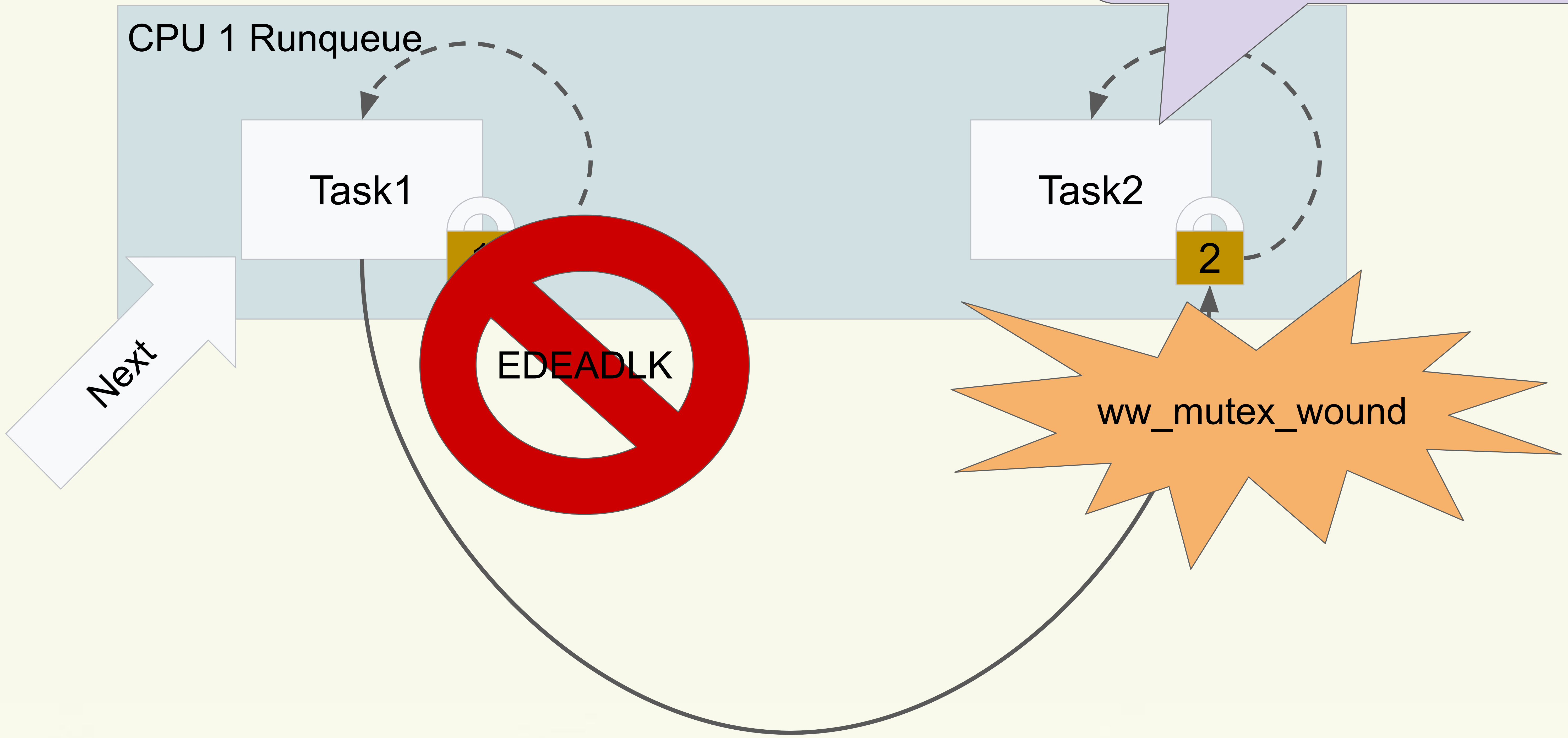


Tangent: ww_mutexes

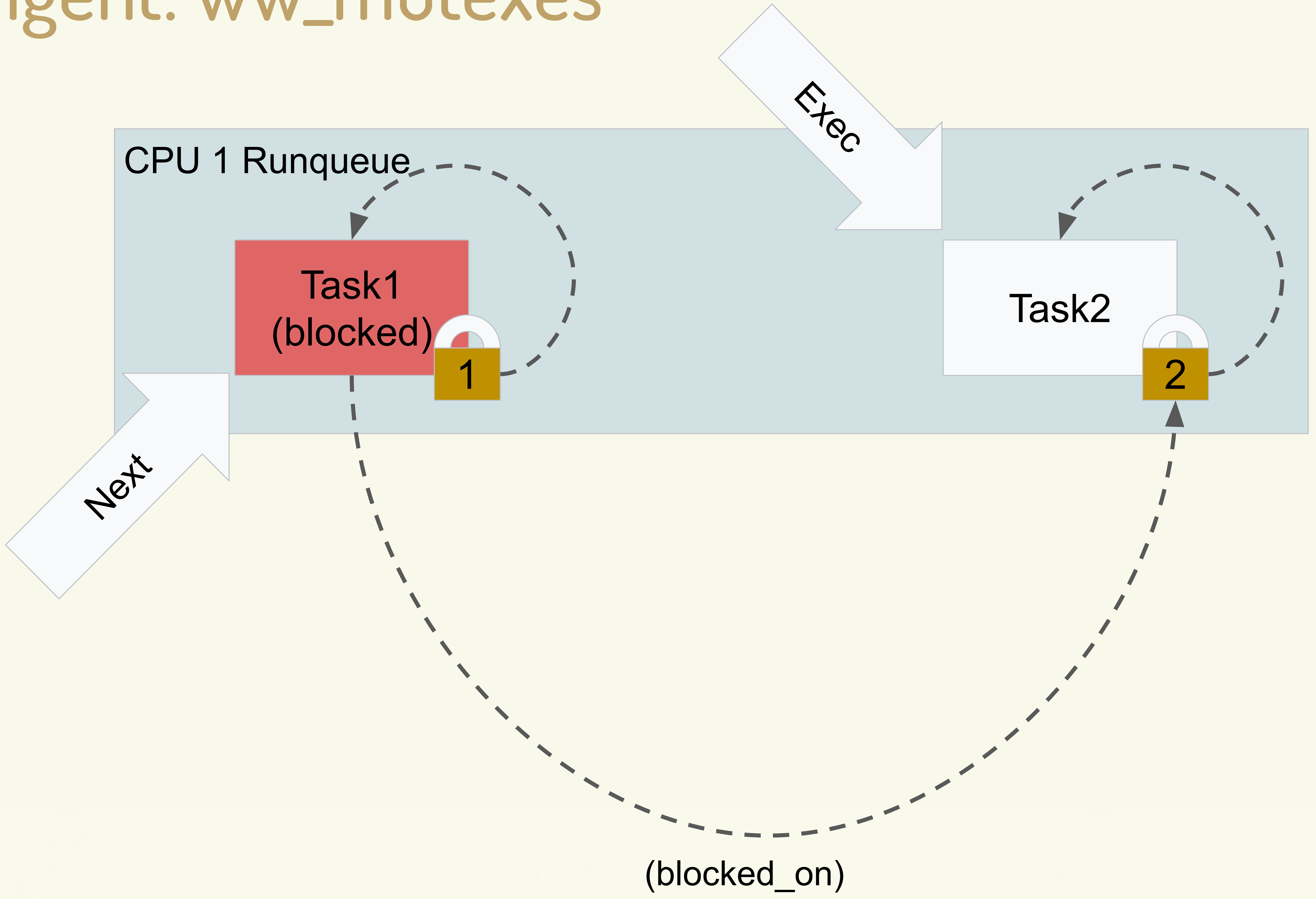


Tangent: ww_mutexes

Prior to v4, the mistake here was not clearing the blocked_on state, causing the task to not be runnable, thus unable to receive the EDEADLK and release locks

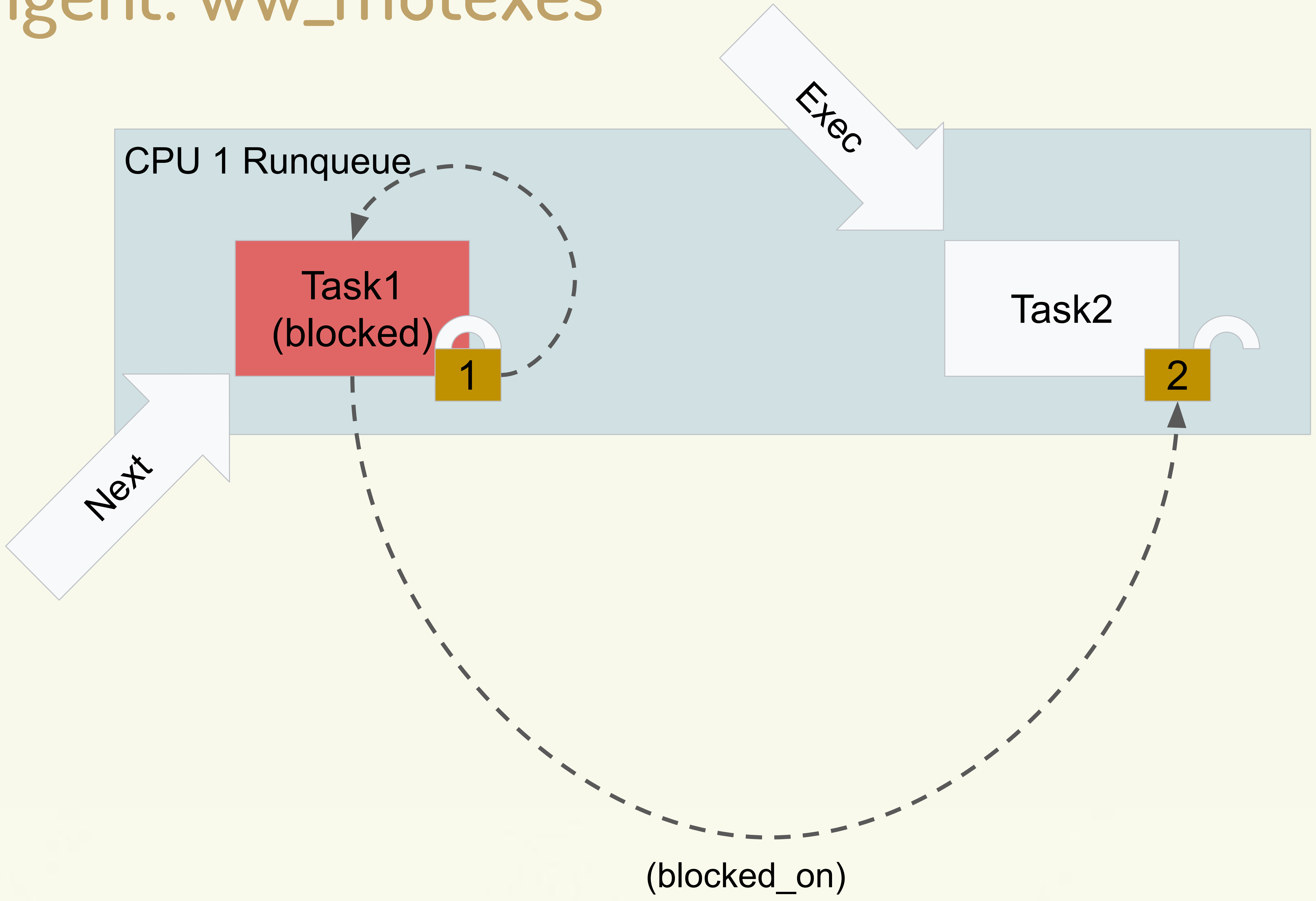


Tangent: ww_mutexes

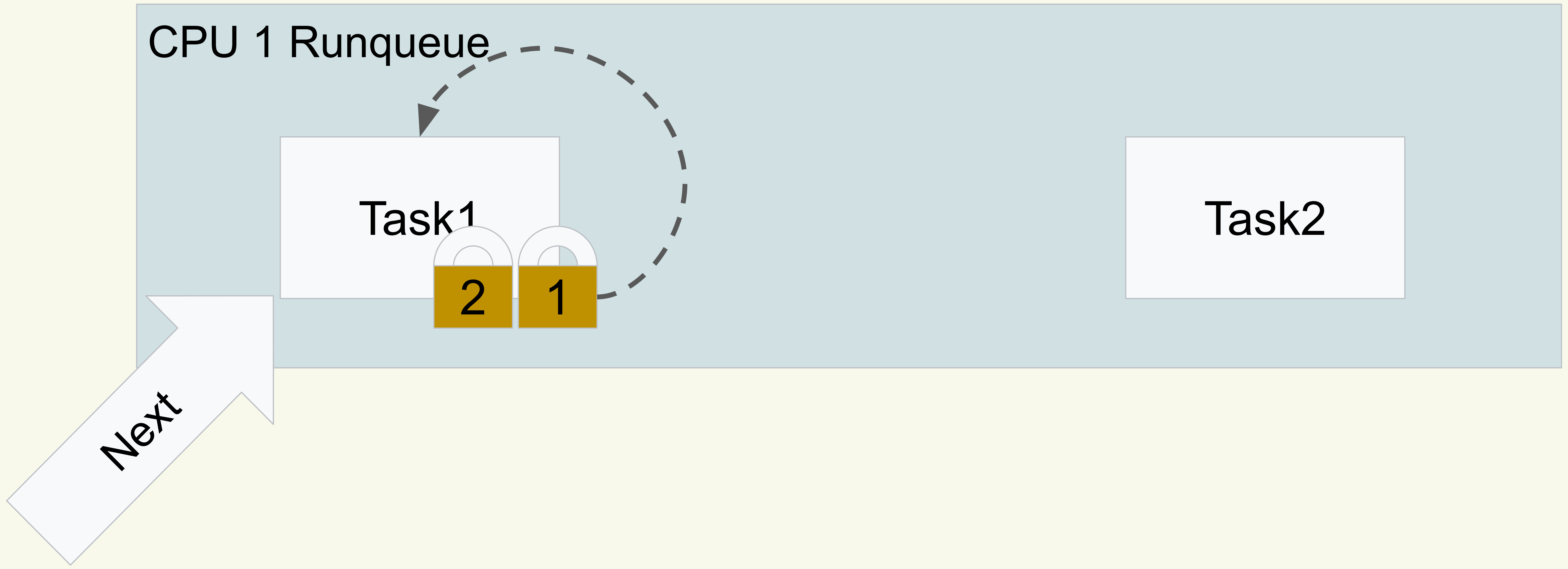


Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

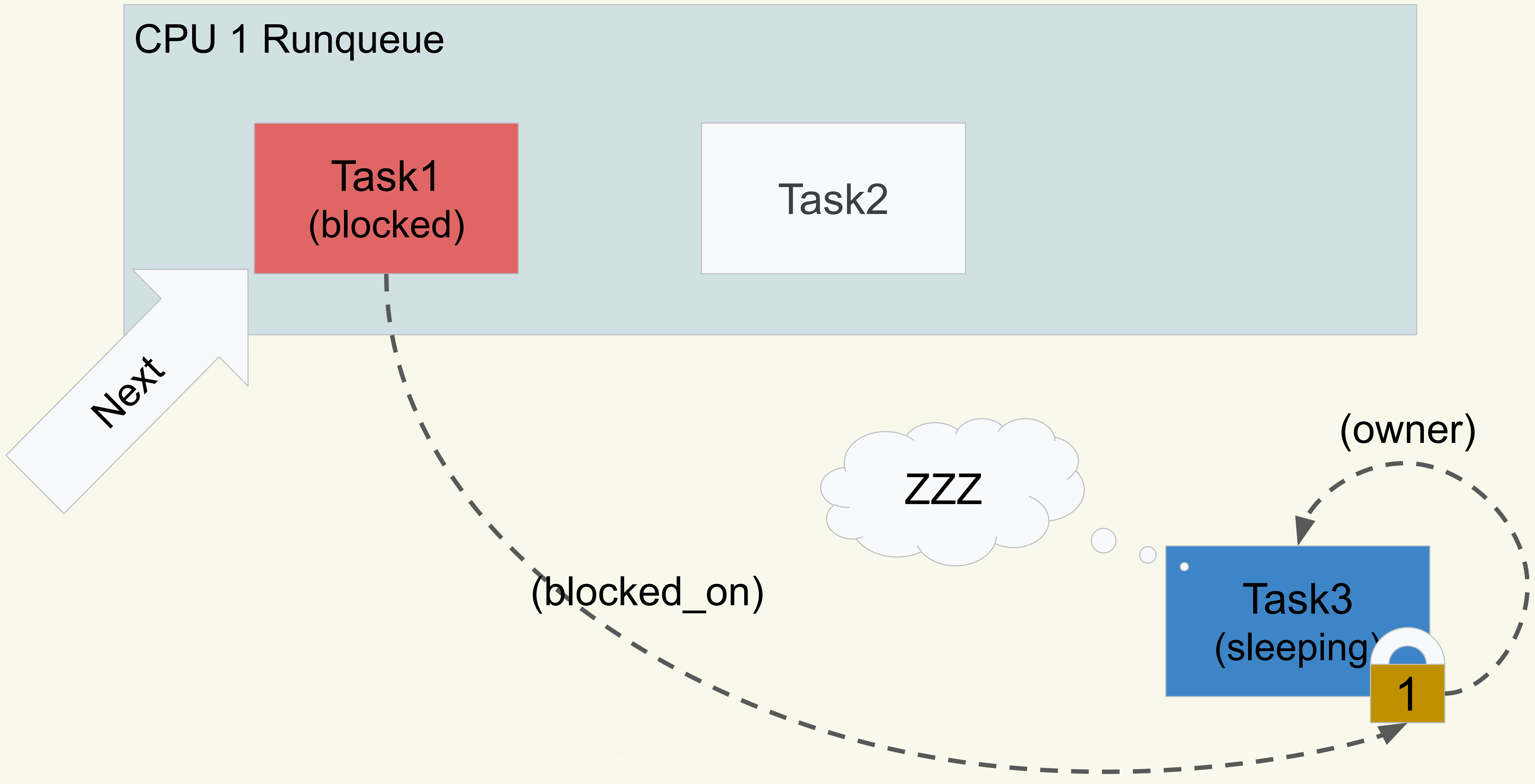
Tangent: ww_mutexes



Tangent: ww_mutexes

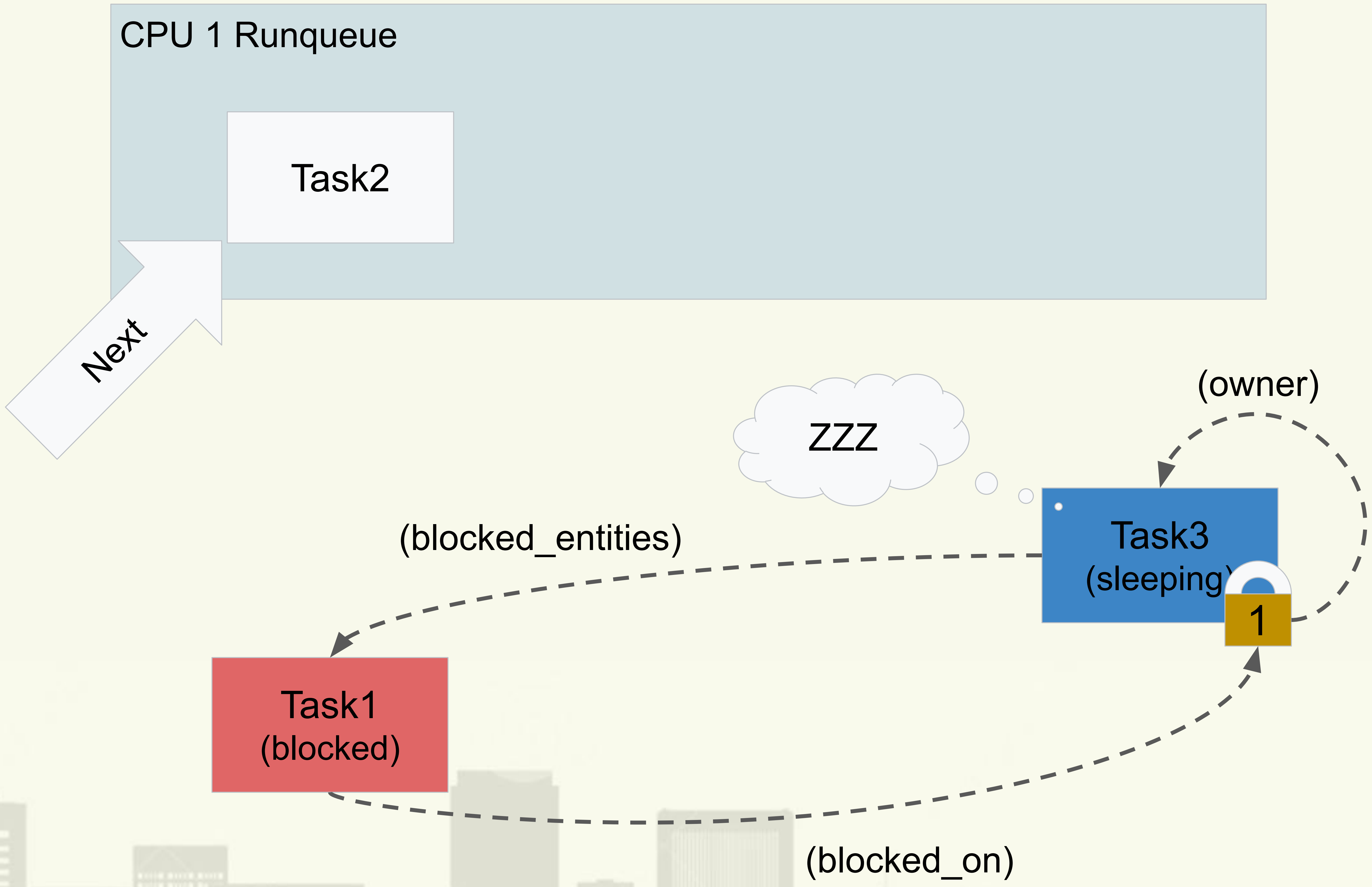


Sleeping Owner Enqueueing

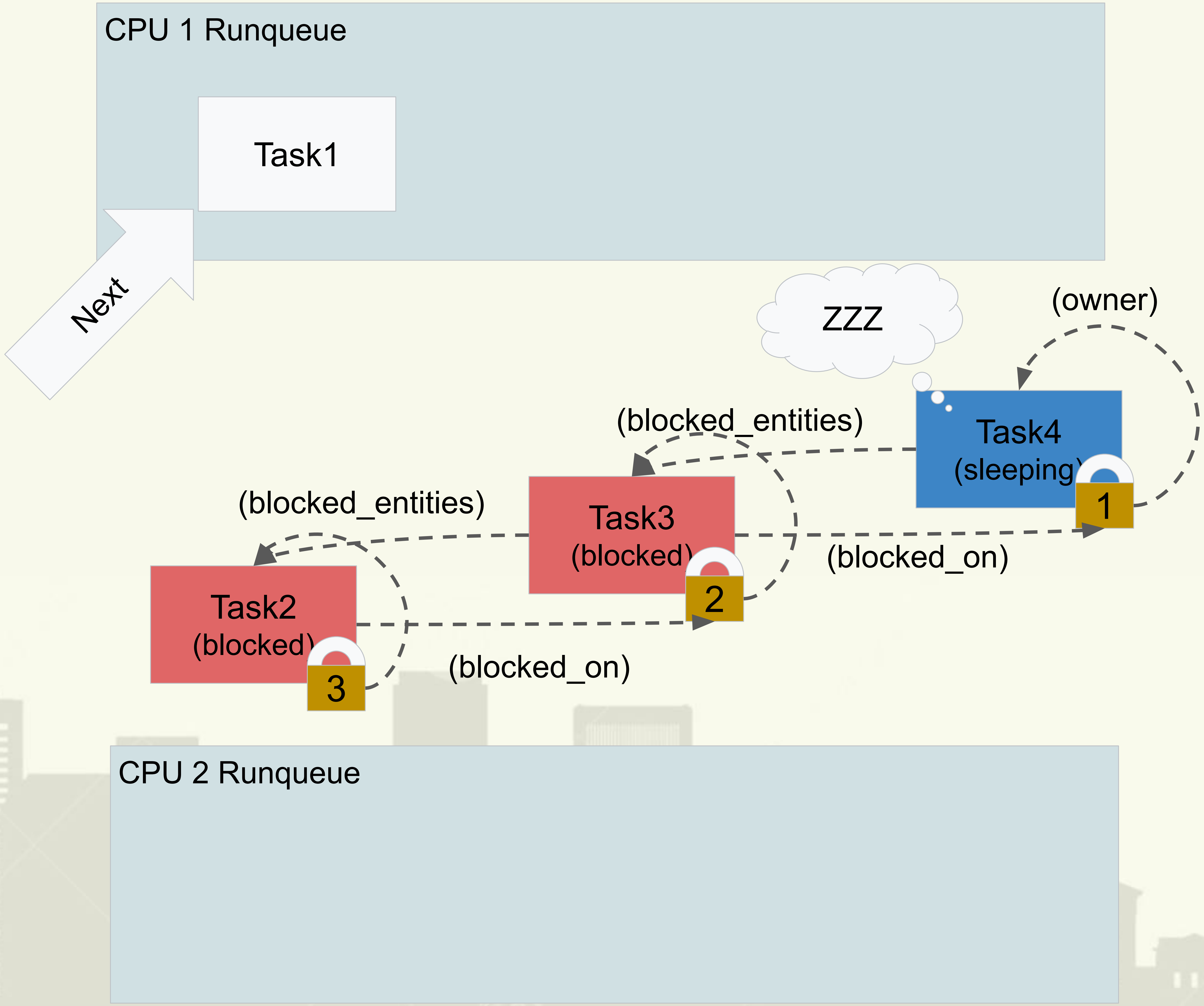


Quick Background
Proxy Execution
Recent Work
Current Issues
Discussion

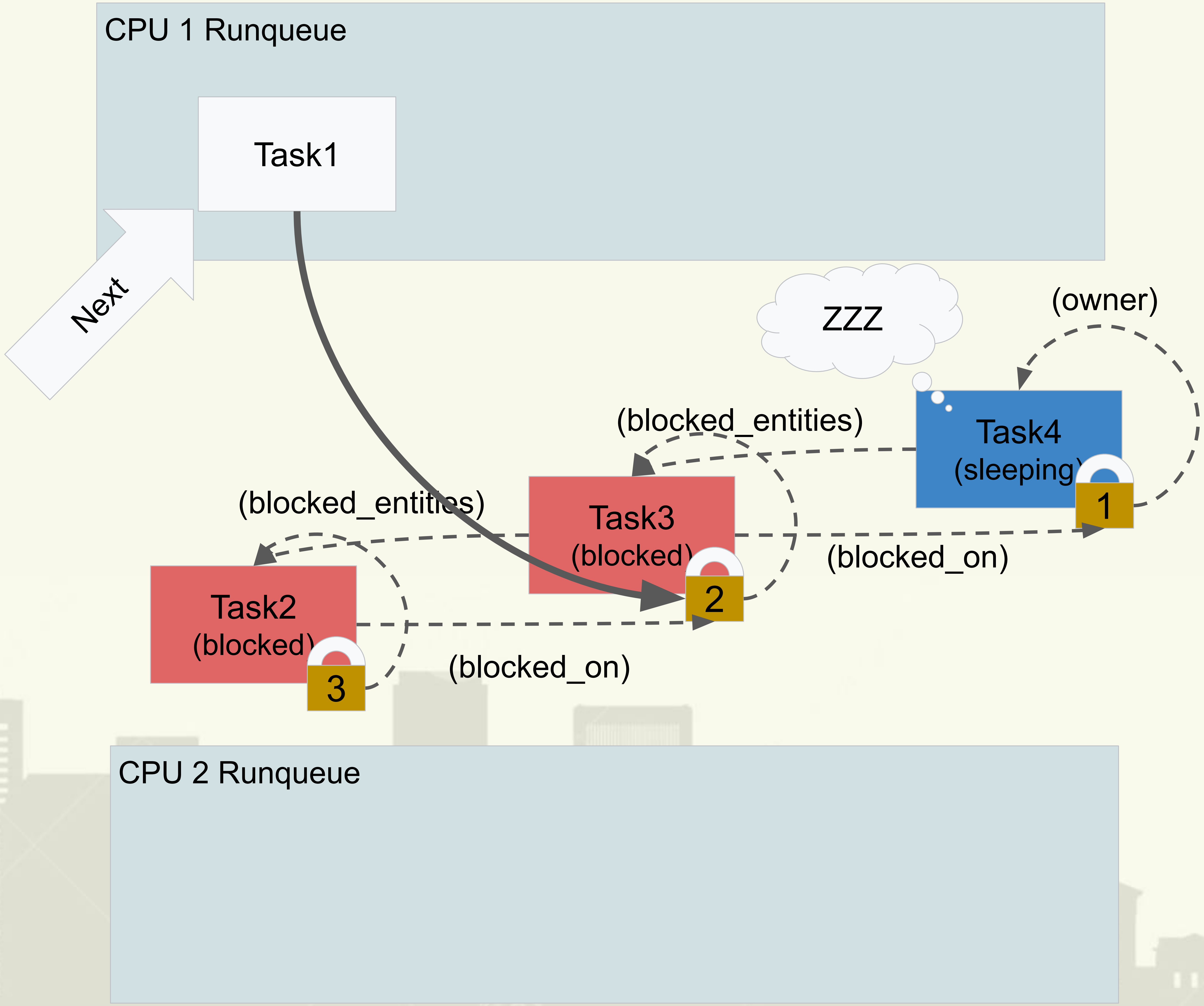
Sleeping Owner Enqueueing (cont)



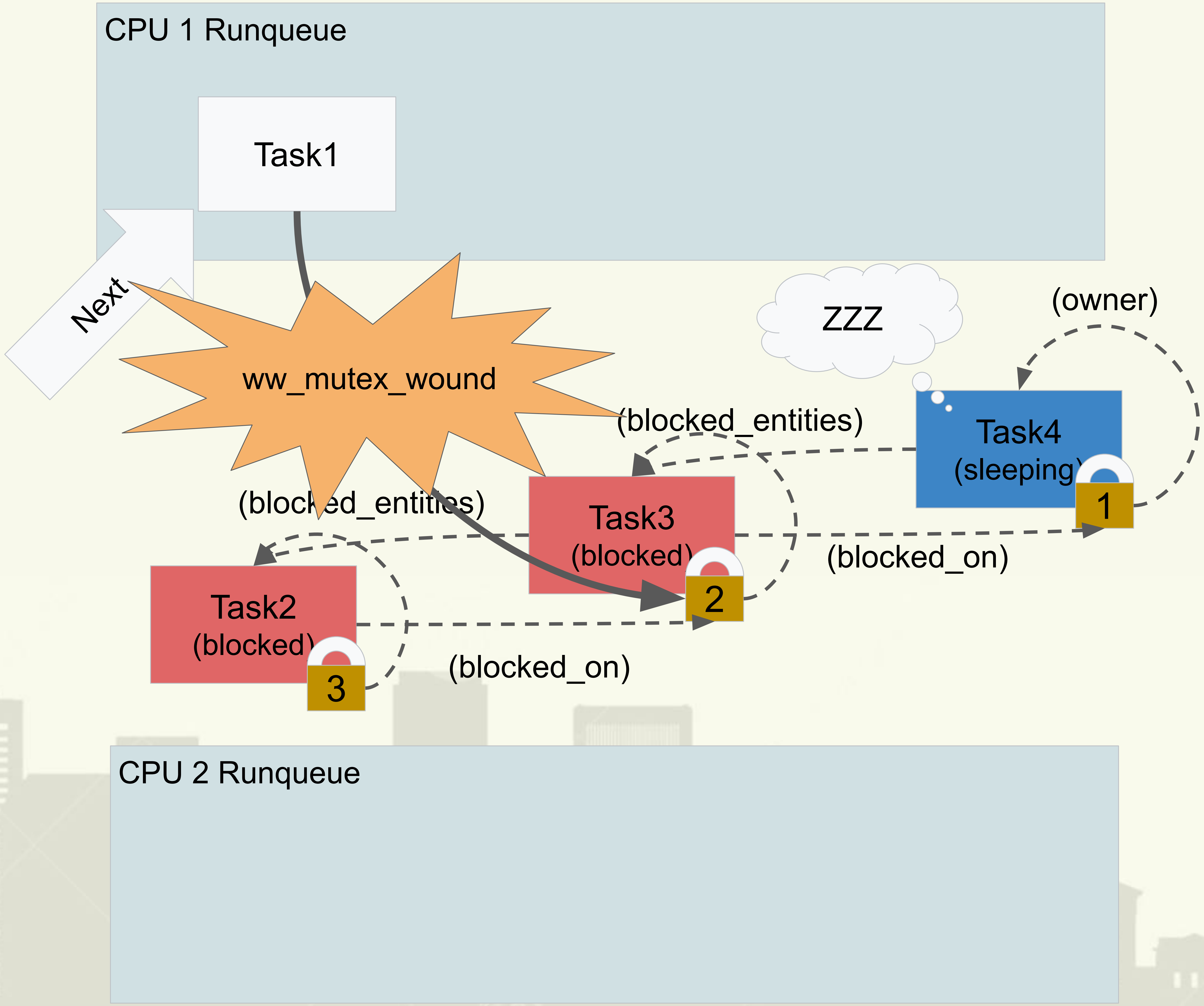
Sleeping Owner Midchain Wakeups



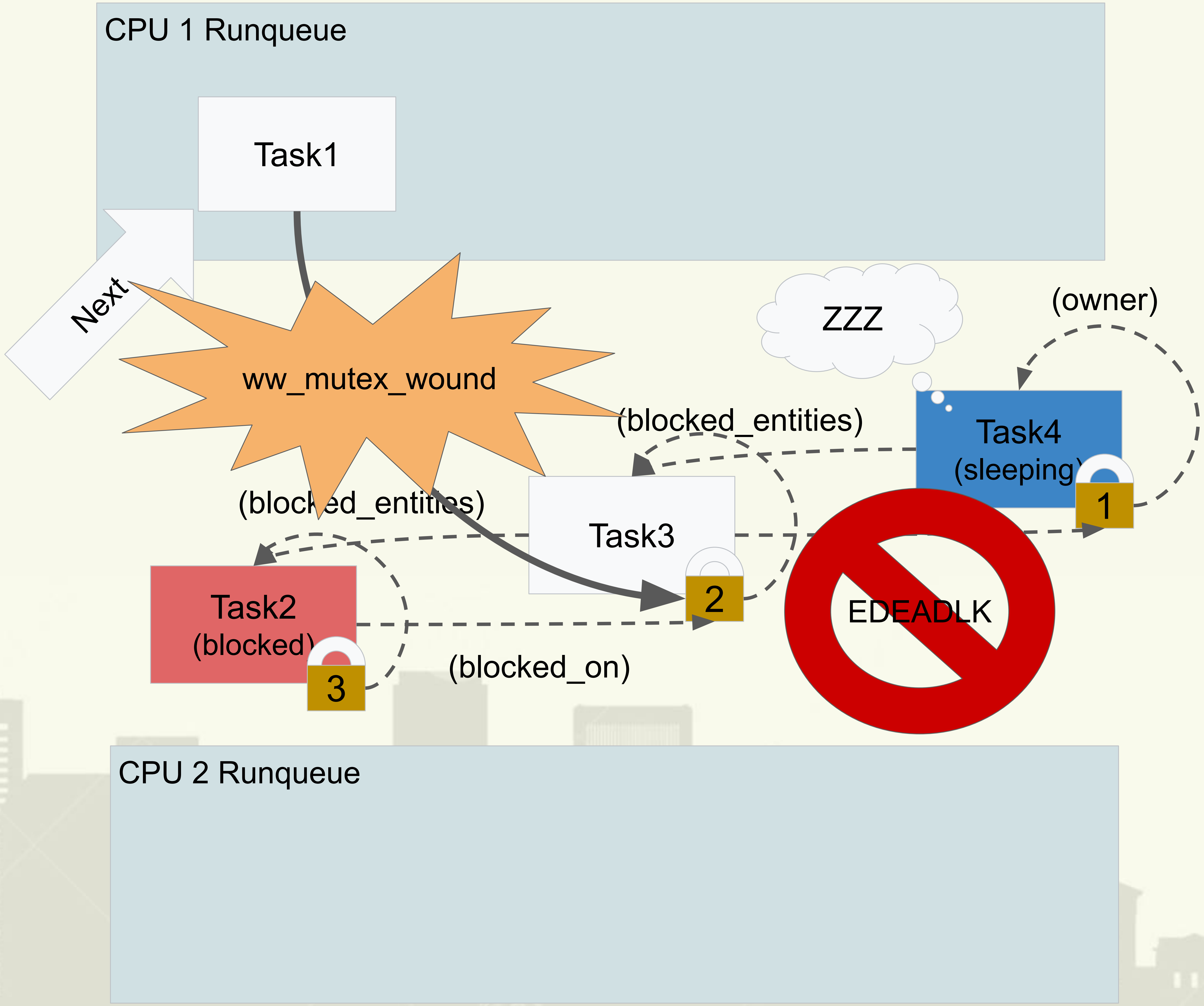
Sleeping Owner Midchain Wakeups



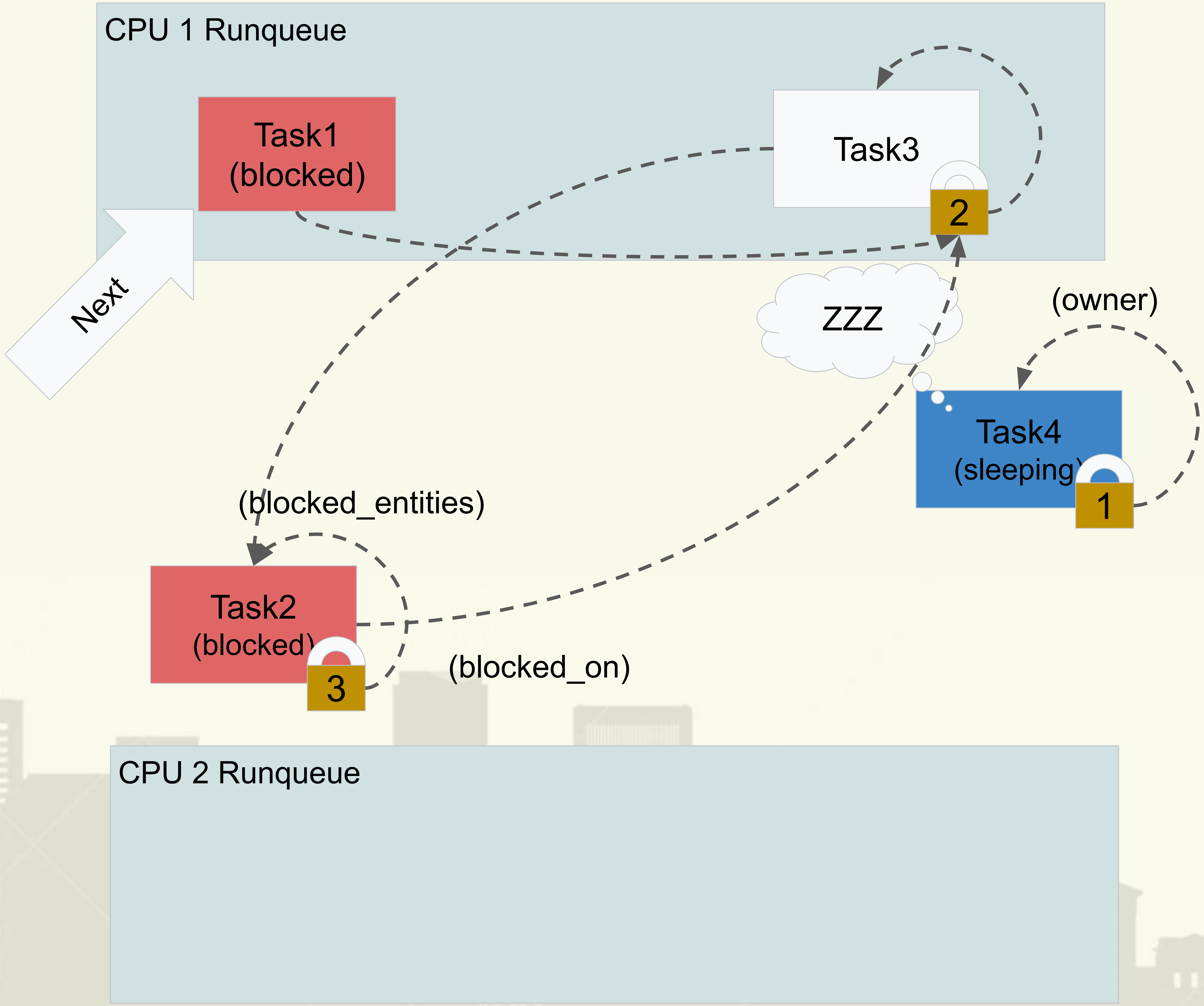
Sleeping Owner Midchain Wakeups



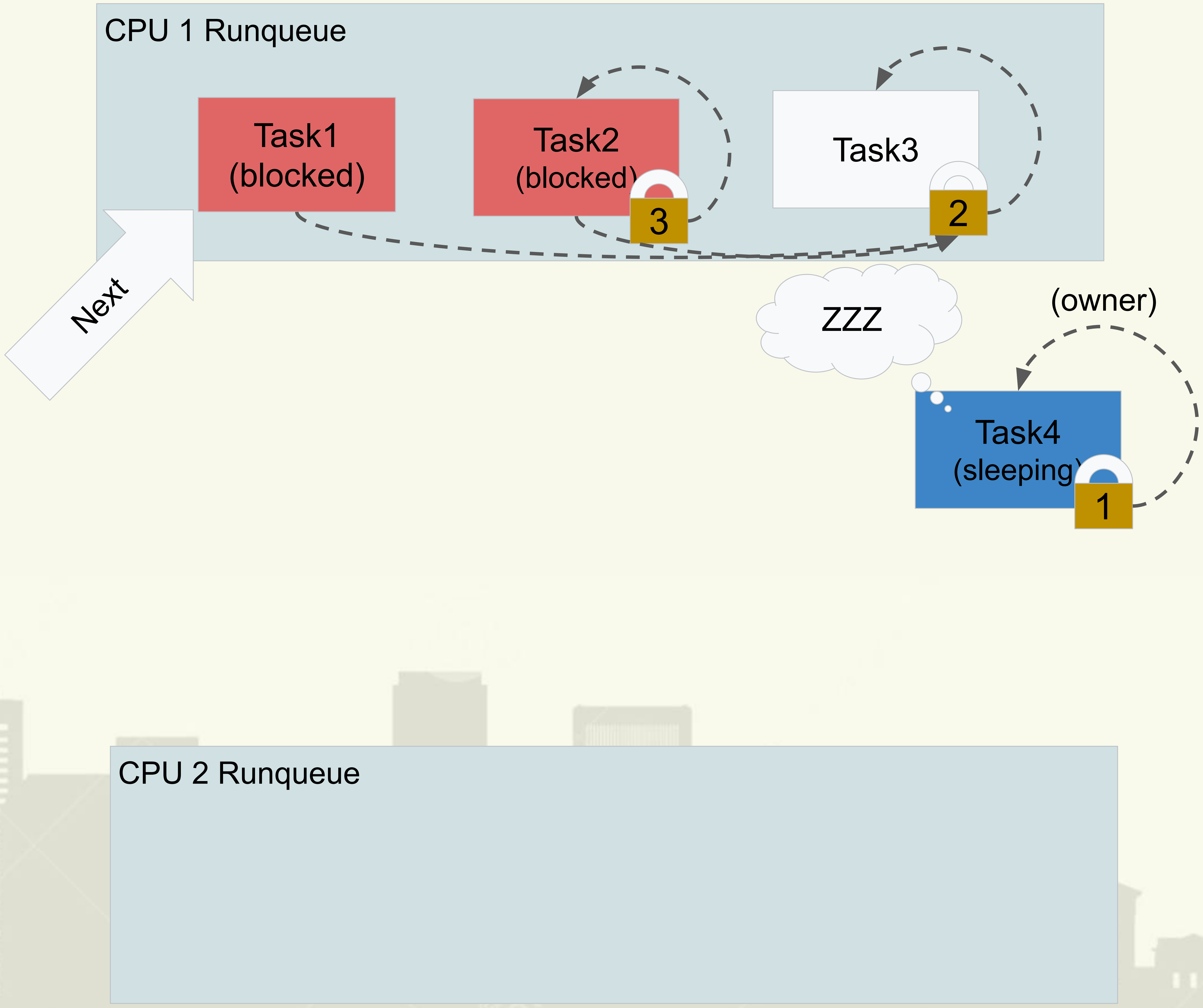
Sleeping Owner Midchain Wakeups



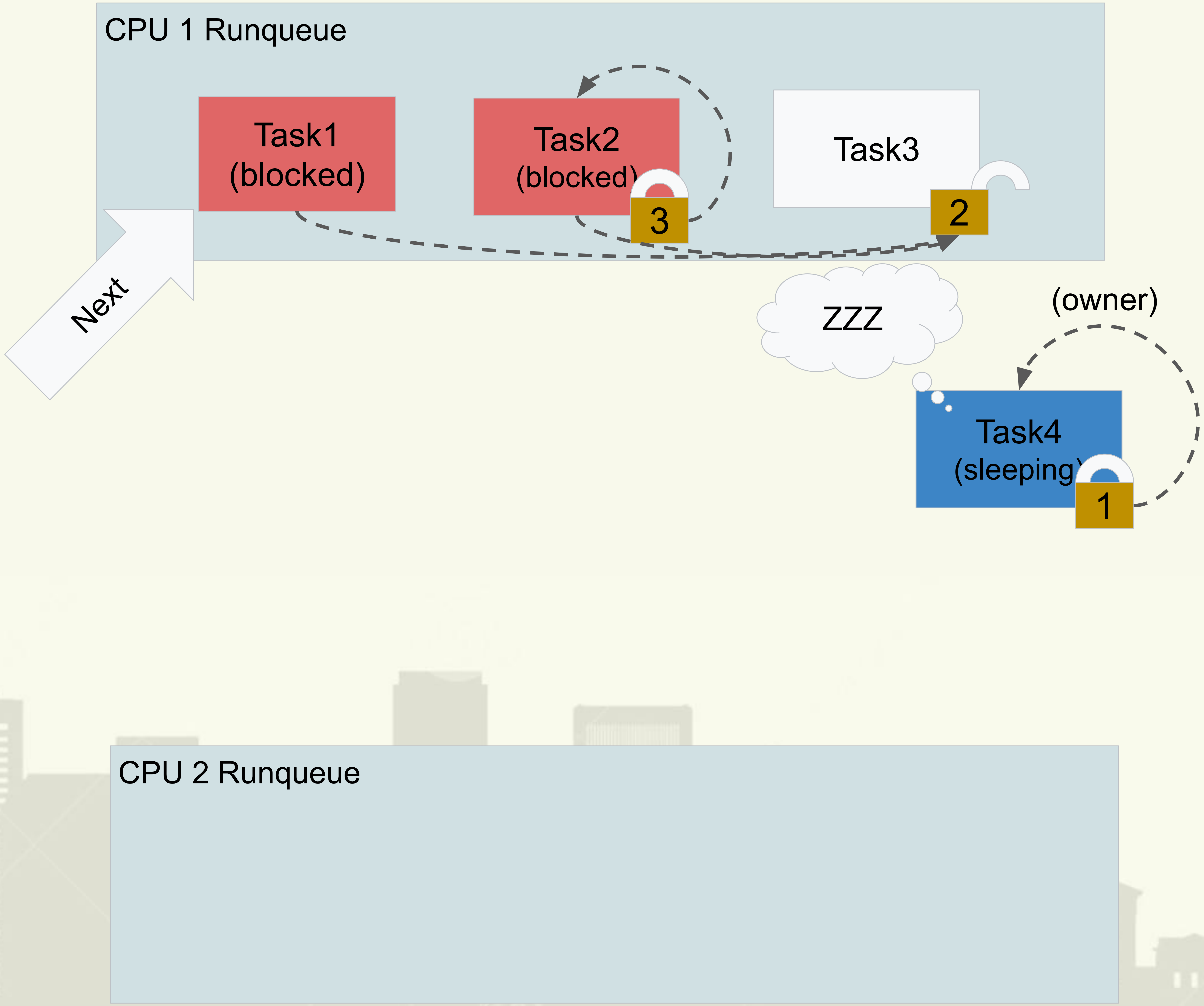
Sleeping Owner Midchain Wakeups



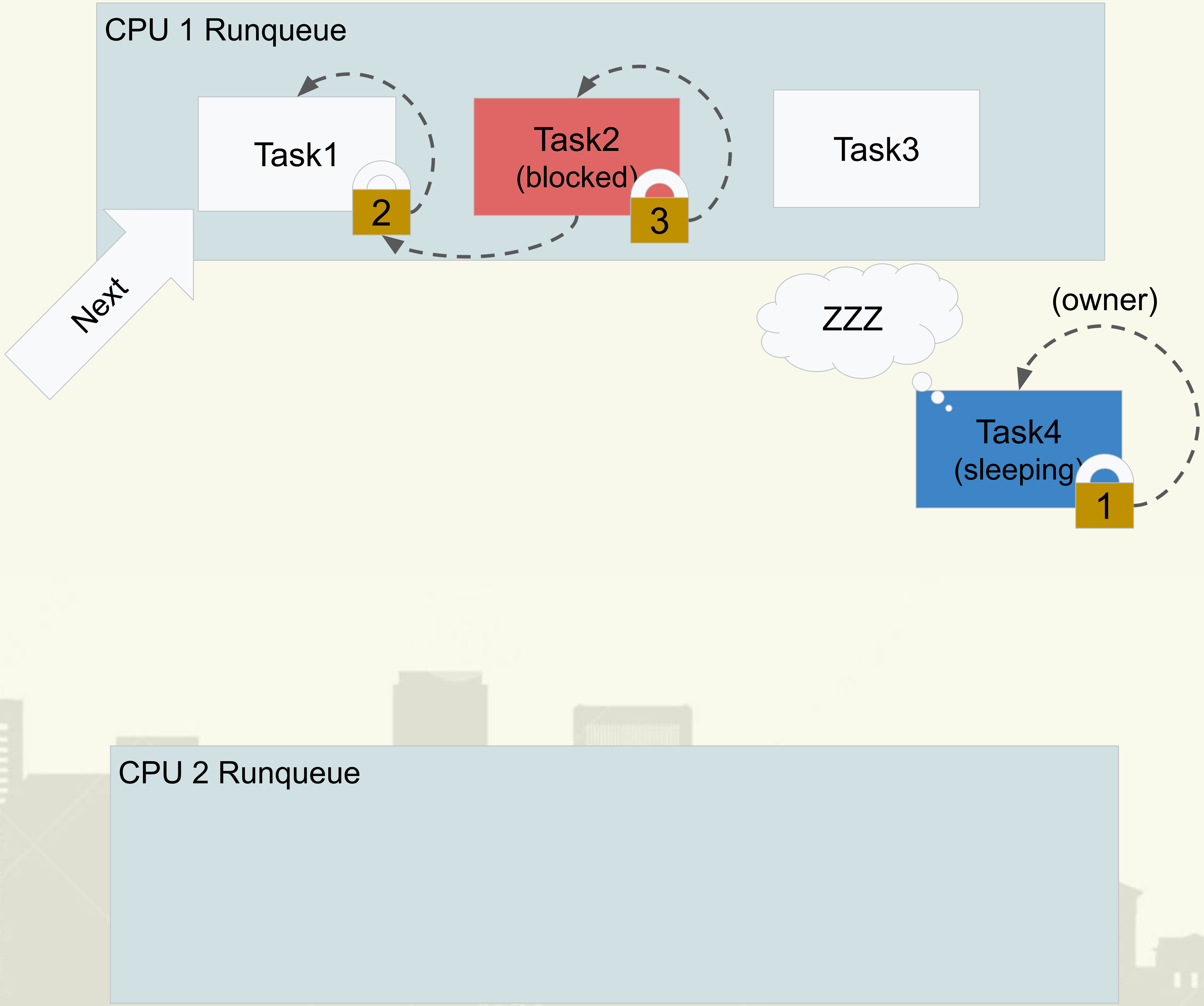
Sleeping Owner Midchain Wakeups



Sleeping Owner Midchain Wwakeups

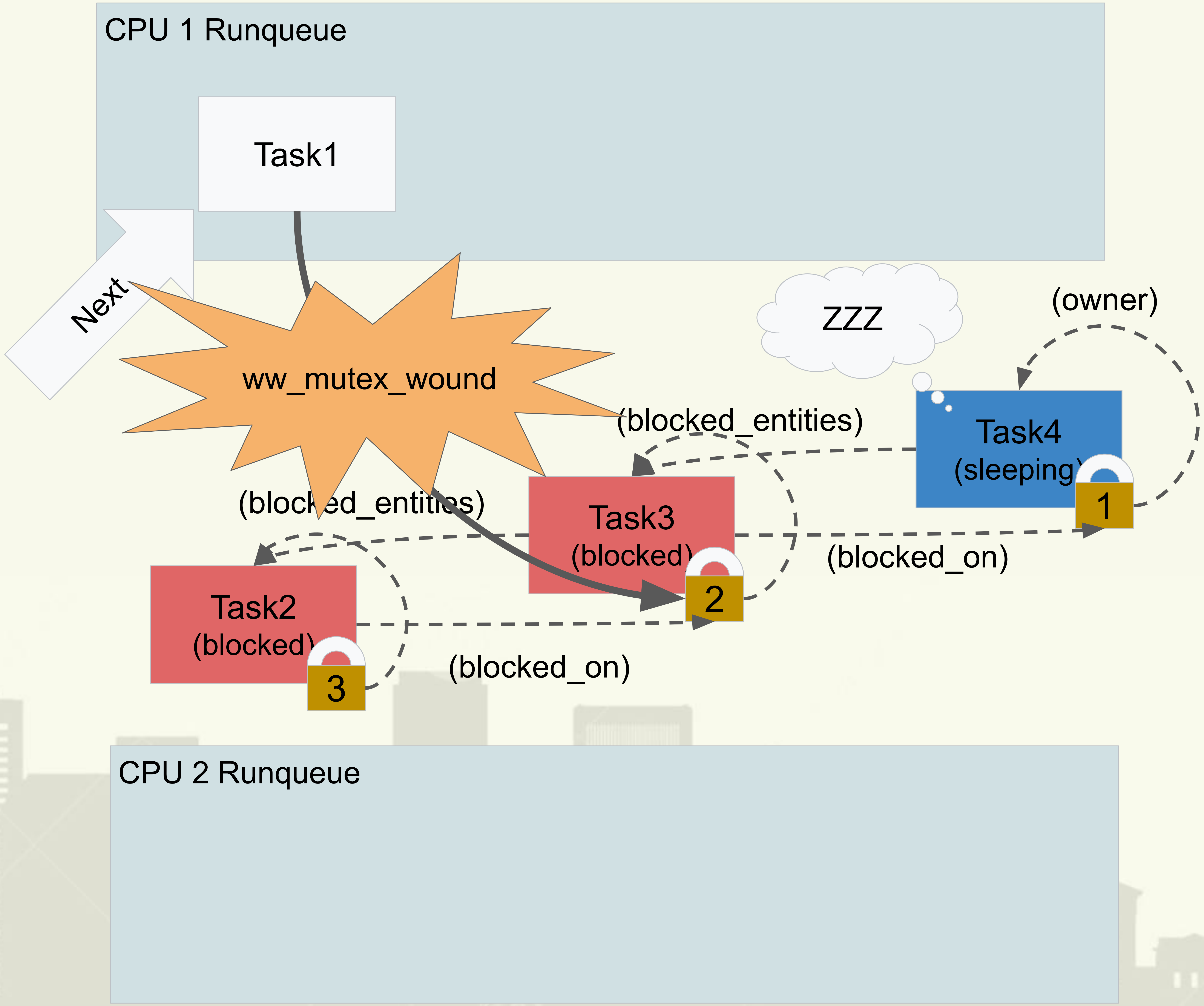


Sleeping Owner Midchain Wakeups

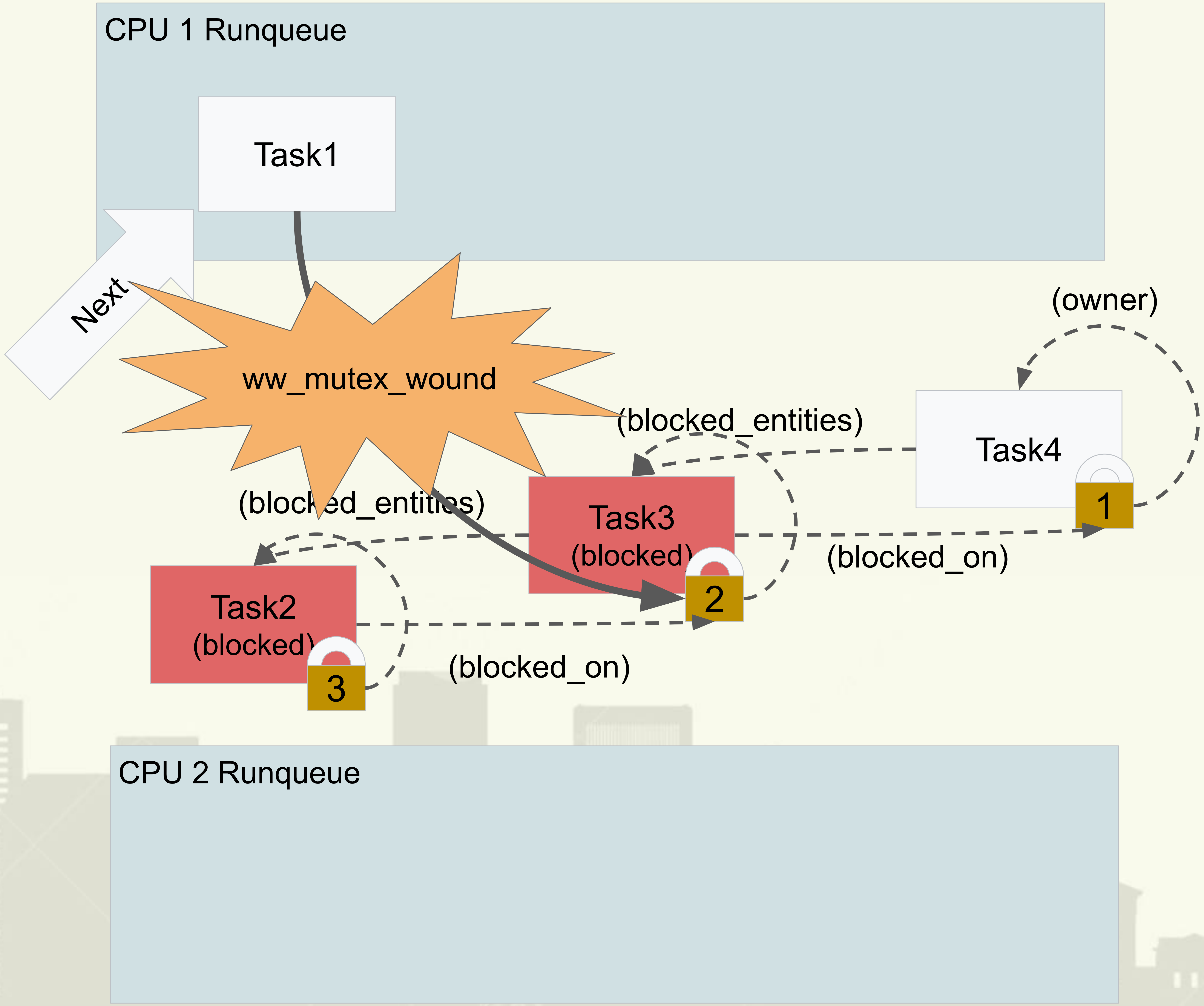


But There's a Race

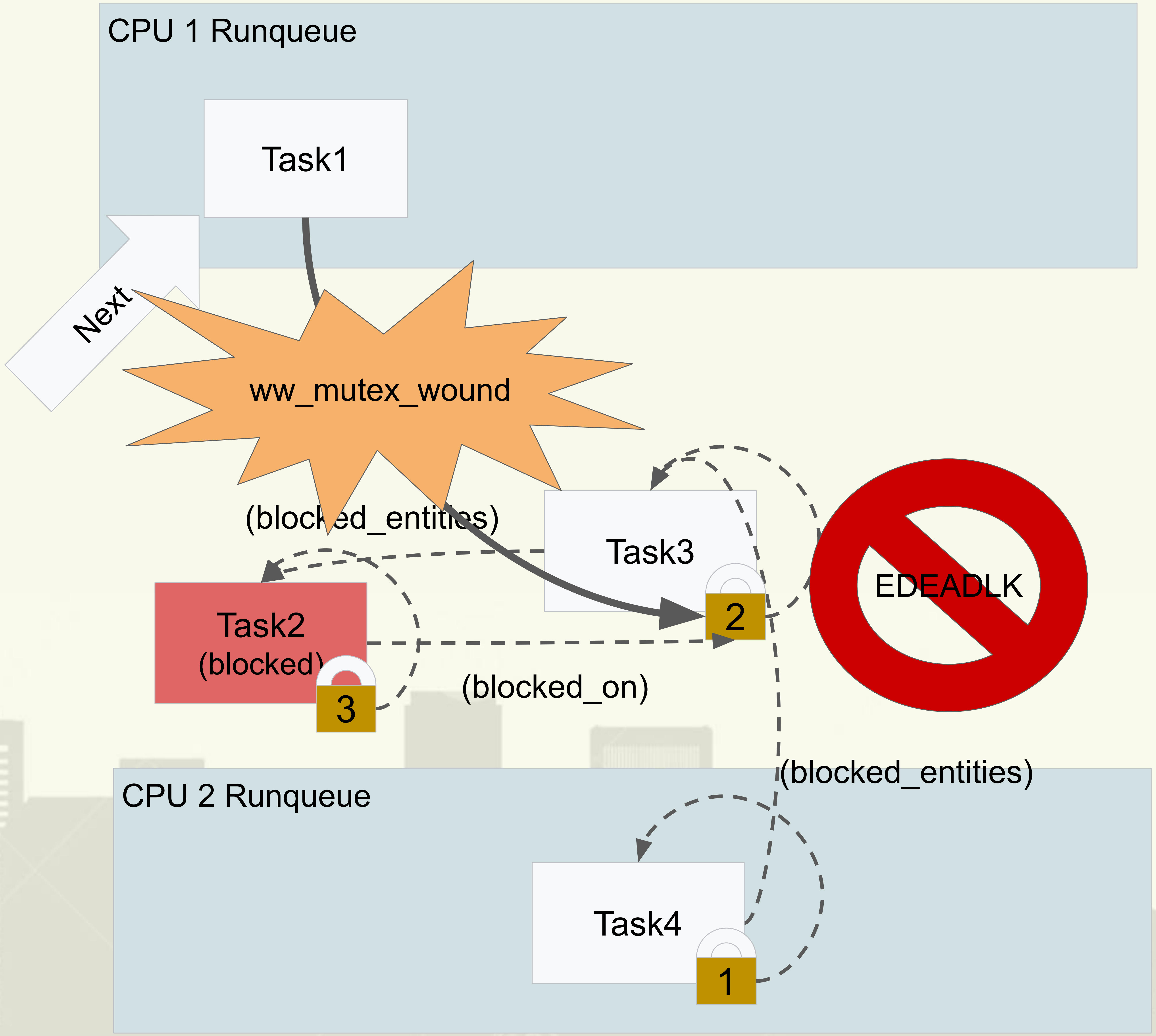
Sleeping Owner Midchain Wakeups



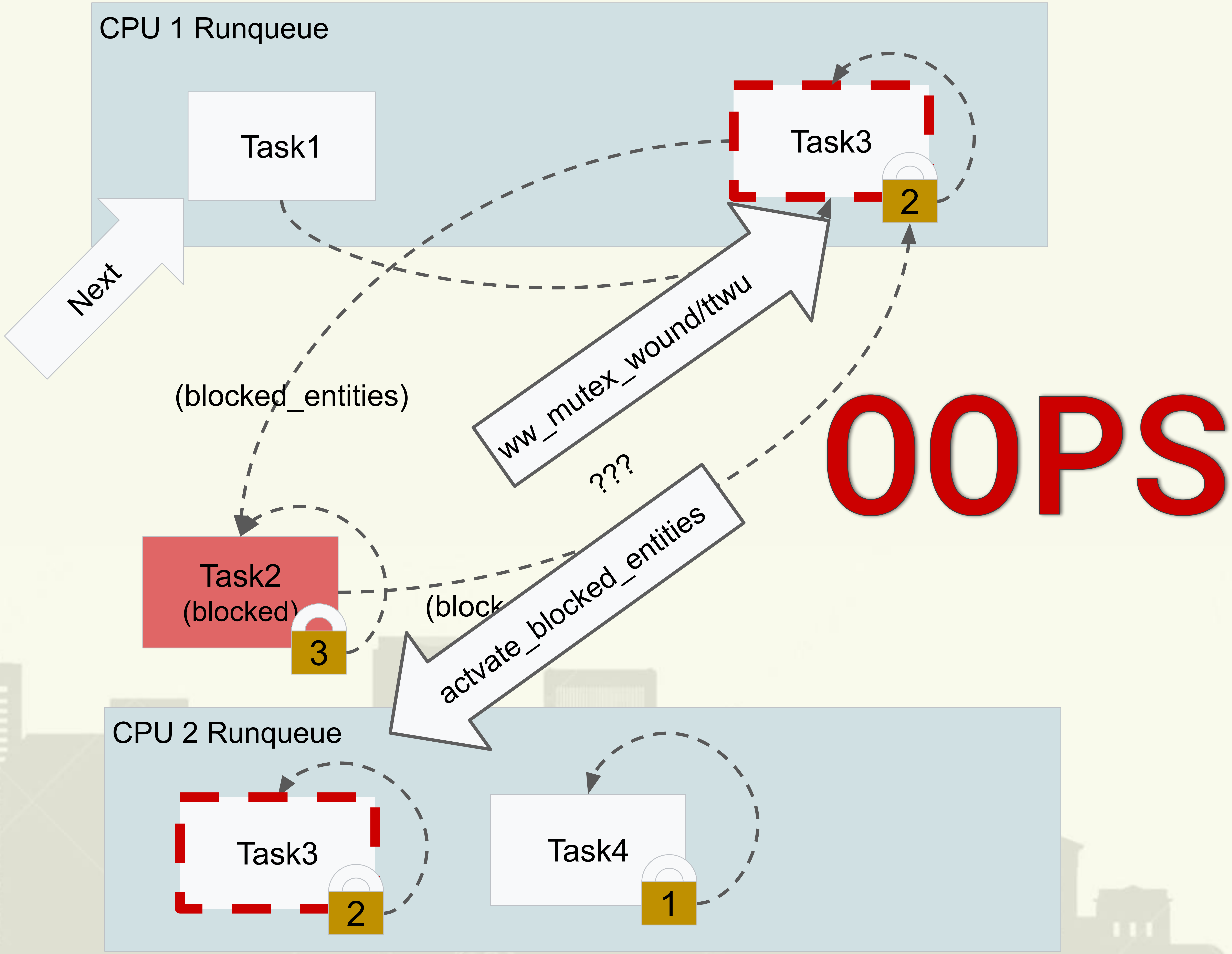
Sleeping Owner Midchain Wakeups



Sleeping Owner Midchain Wakeups



Sleeping Owner Midchain Wakeups

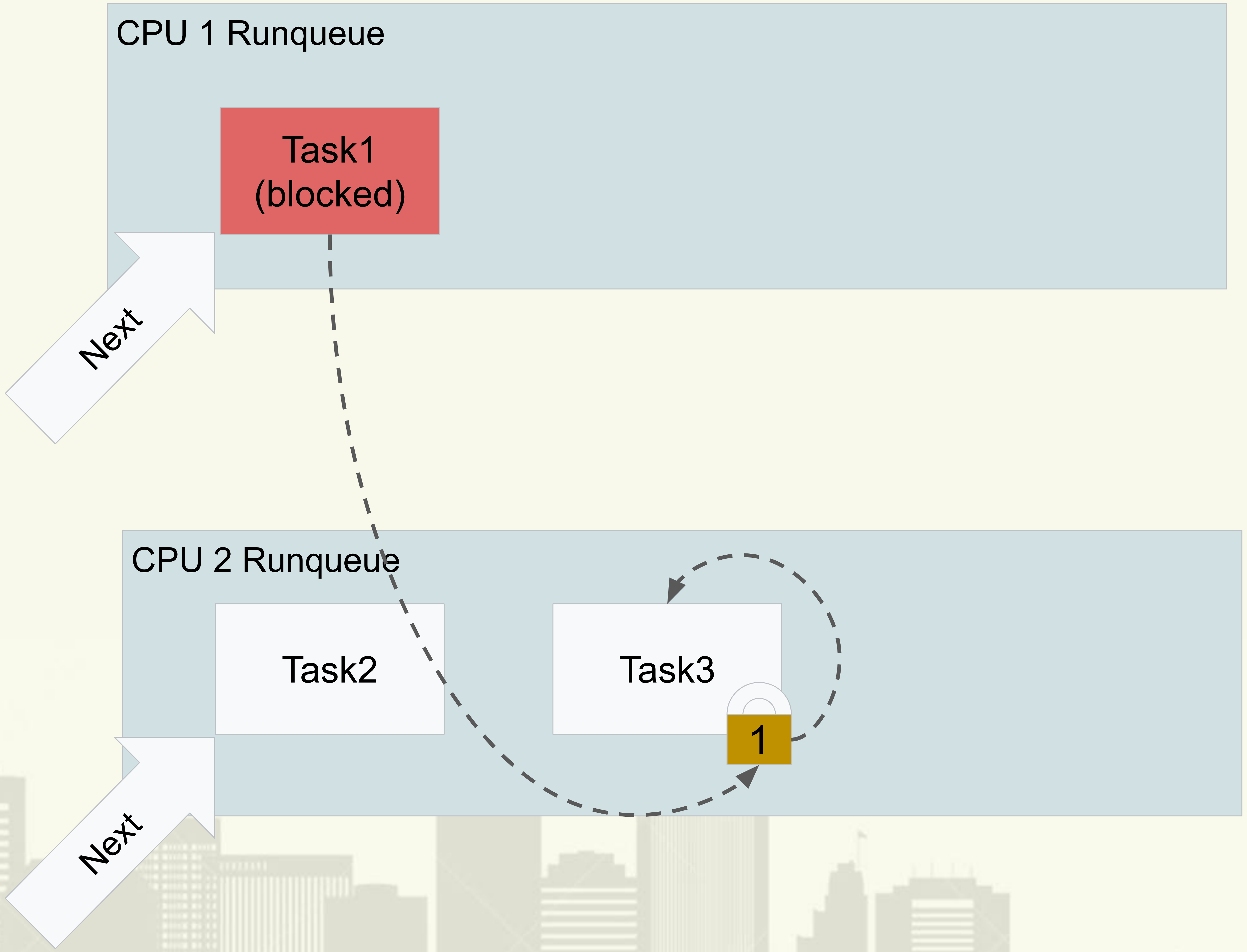


Complications

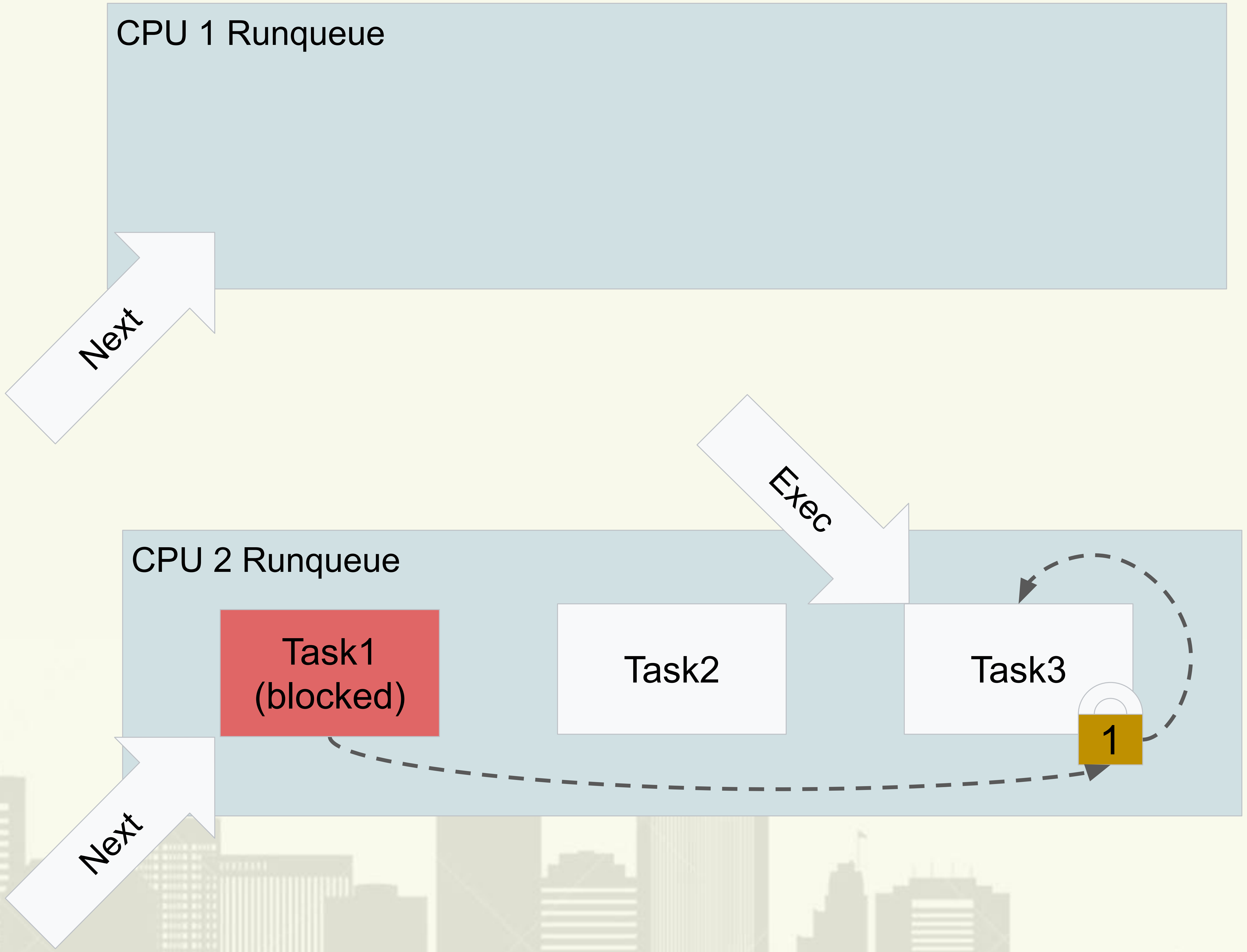
- **Lock order**: task.pi_lock -> rq.lock -> mutex.wait_lock -> task.blocked_lock
- From ww_mutex_wound() we call try_to_wake_up(), and hold task.pi_lock
- From activate_task() where we'd activate blocked_entities, we're already holding the owner's pi_lock & local rq lock.
  - Have to drop and pick up other locks in the middle of things
- With 100s of blocked entities, dropping and taking all the locks to activate them all can take time.
  - In the meantime, the owning task might migrate to other cpus
  - Might go to sleep
  - Might add new blocked entities!

Proxy & Return Migration Locking

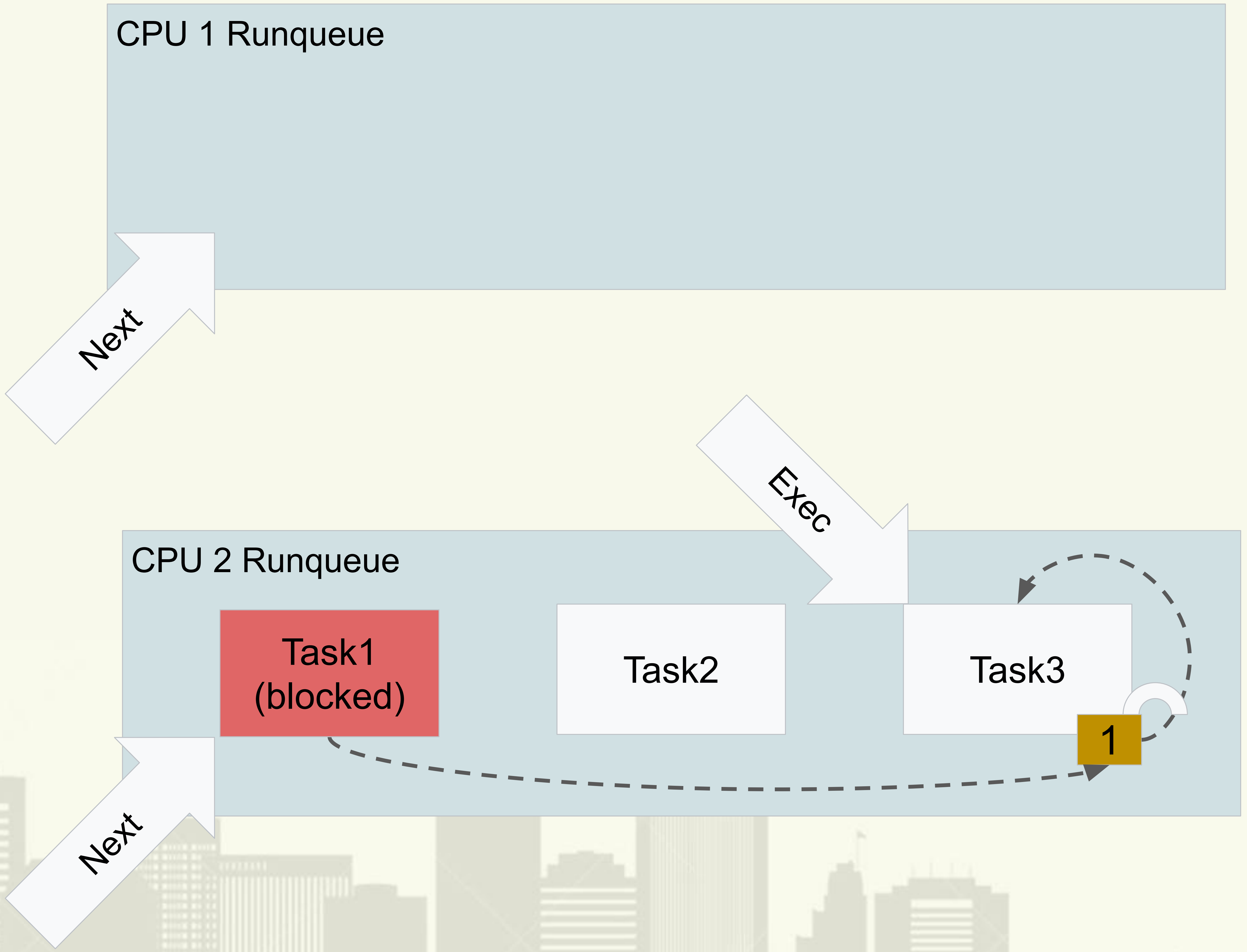
Proxy migration



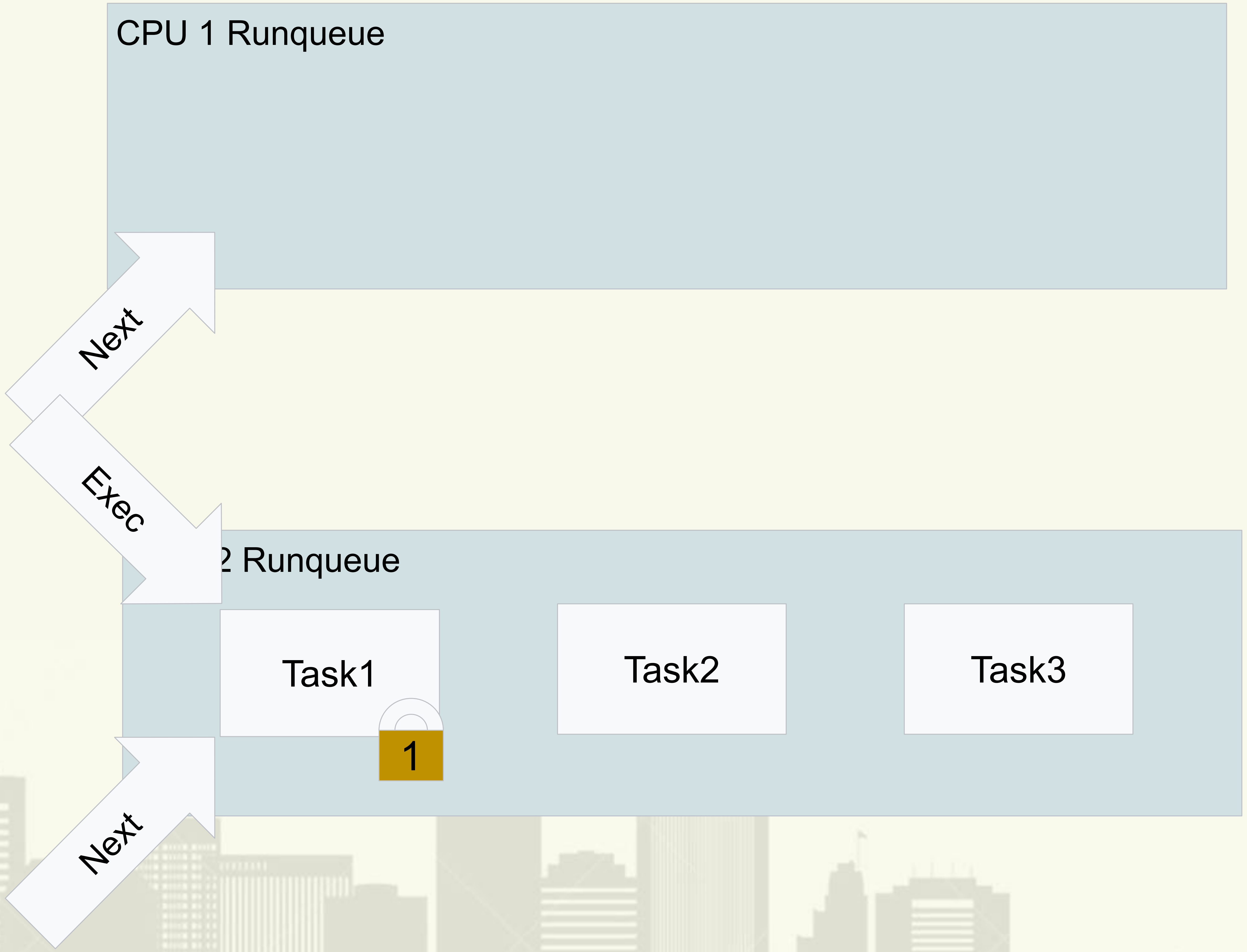
Proxy migration



Proxy migration

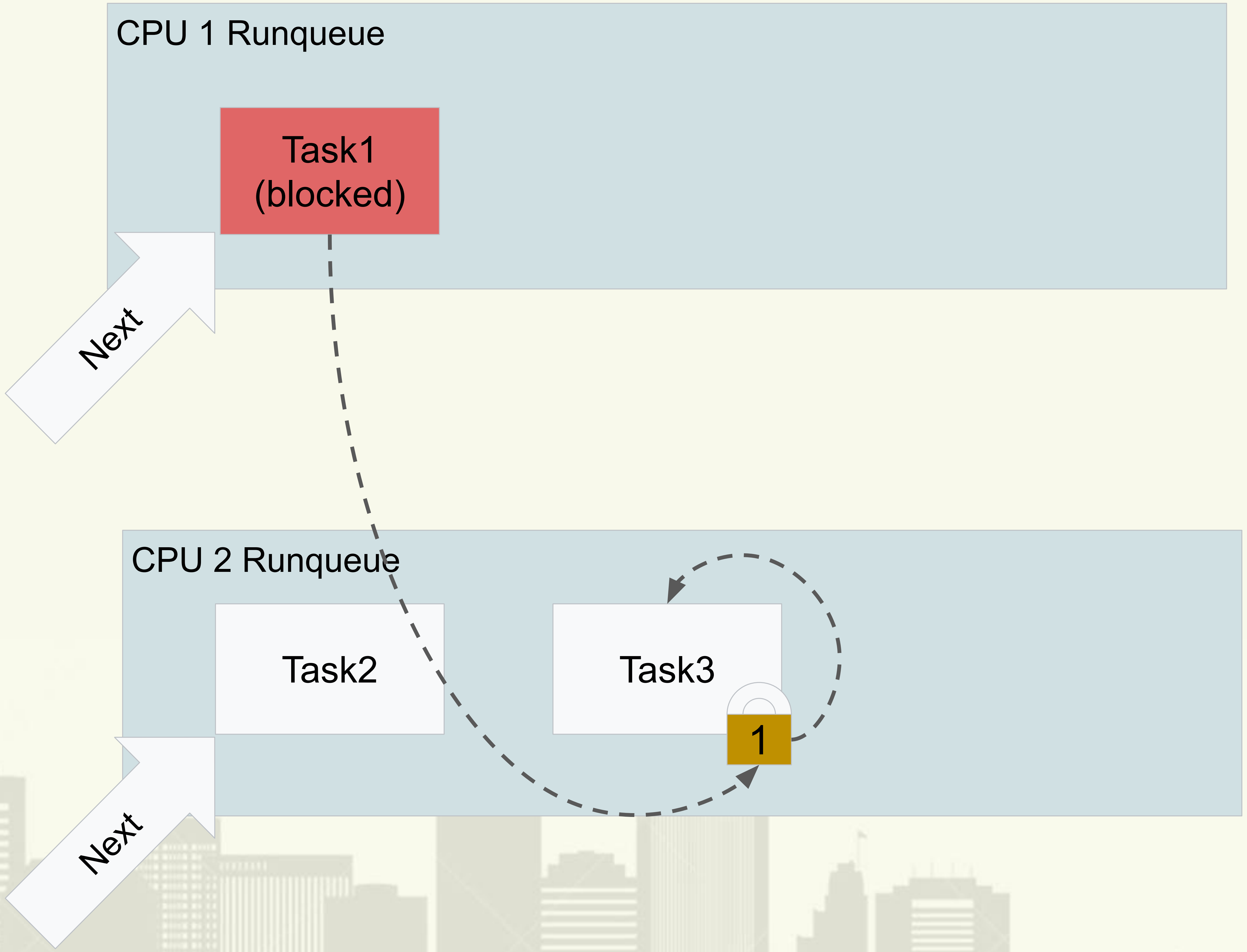


Proxy migration

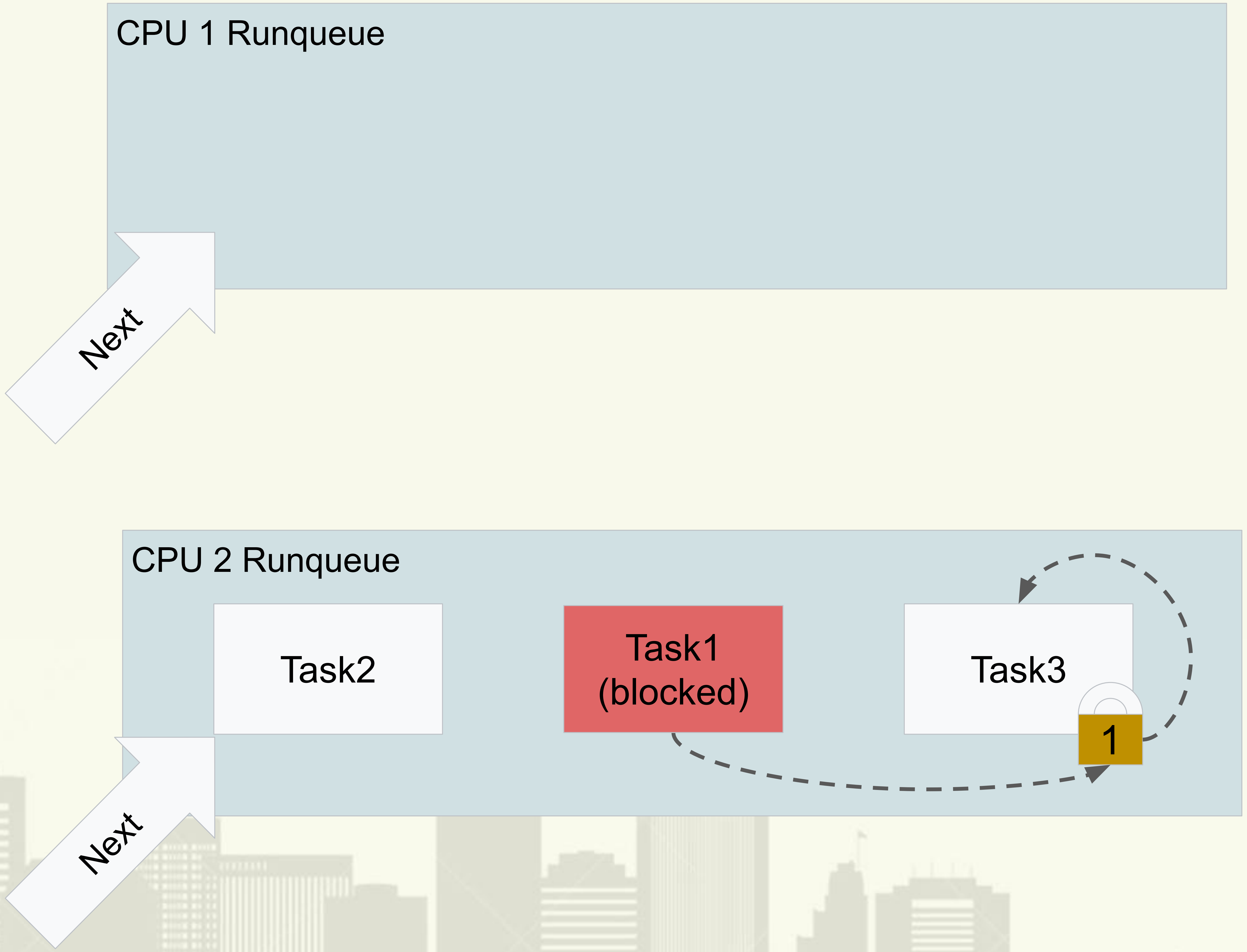


But Task1 might not be able to run on CPU2!

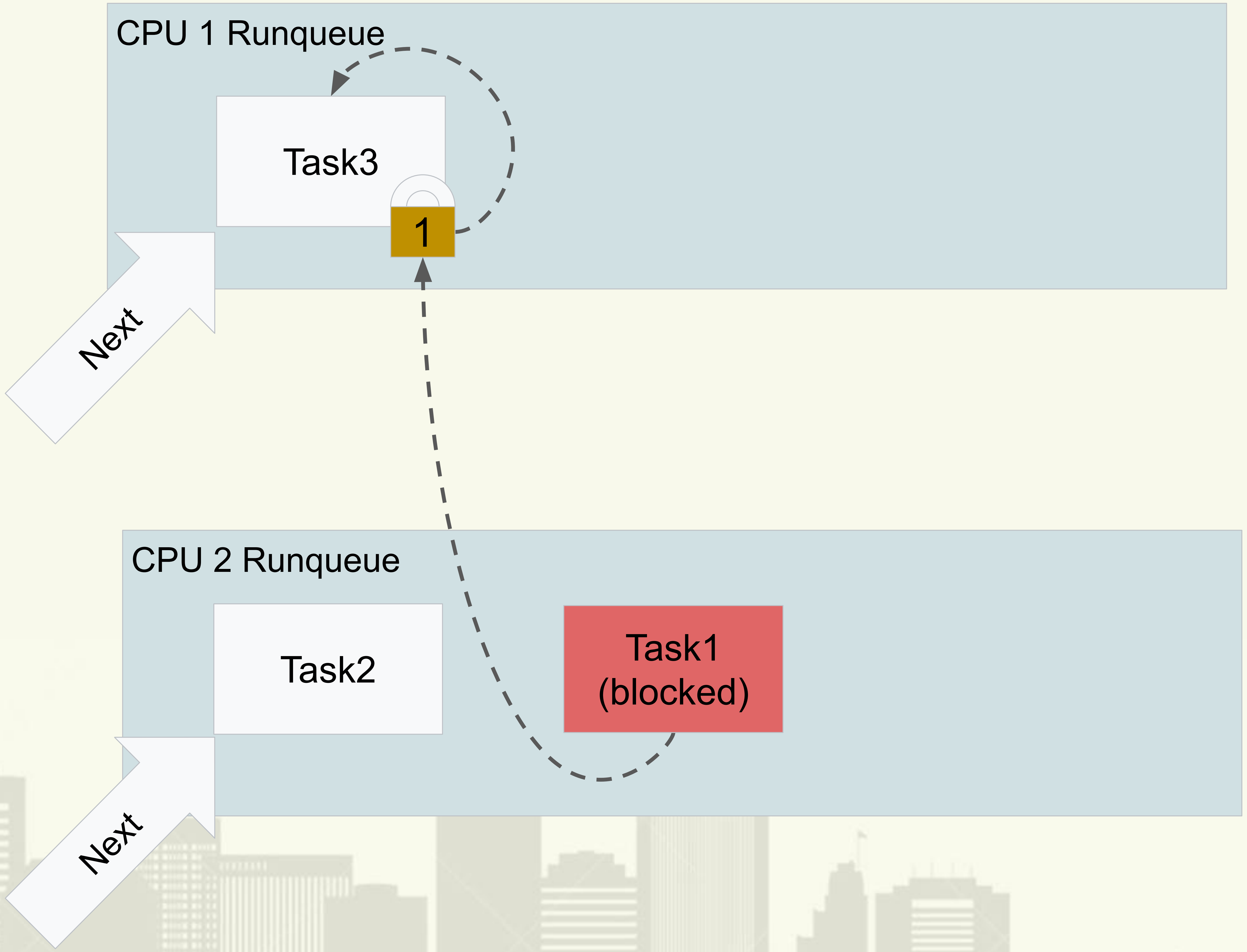
Proxy migration



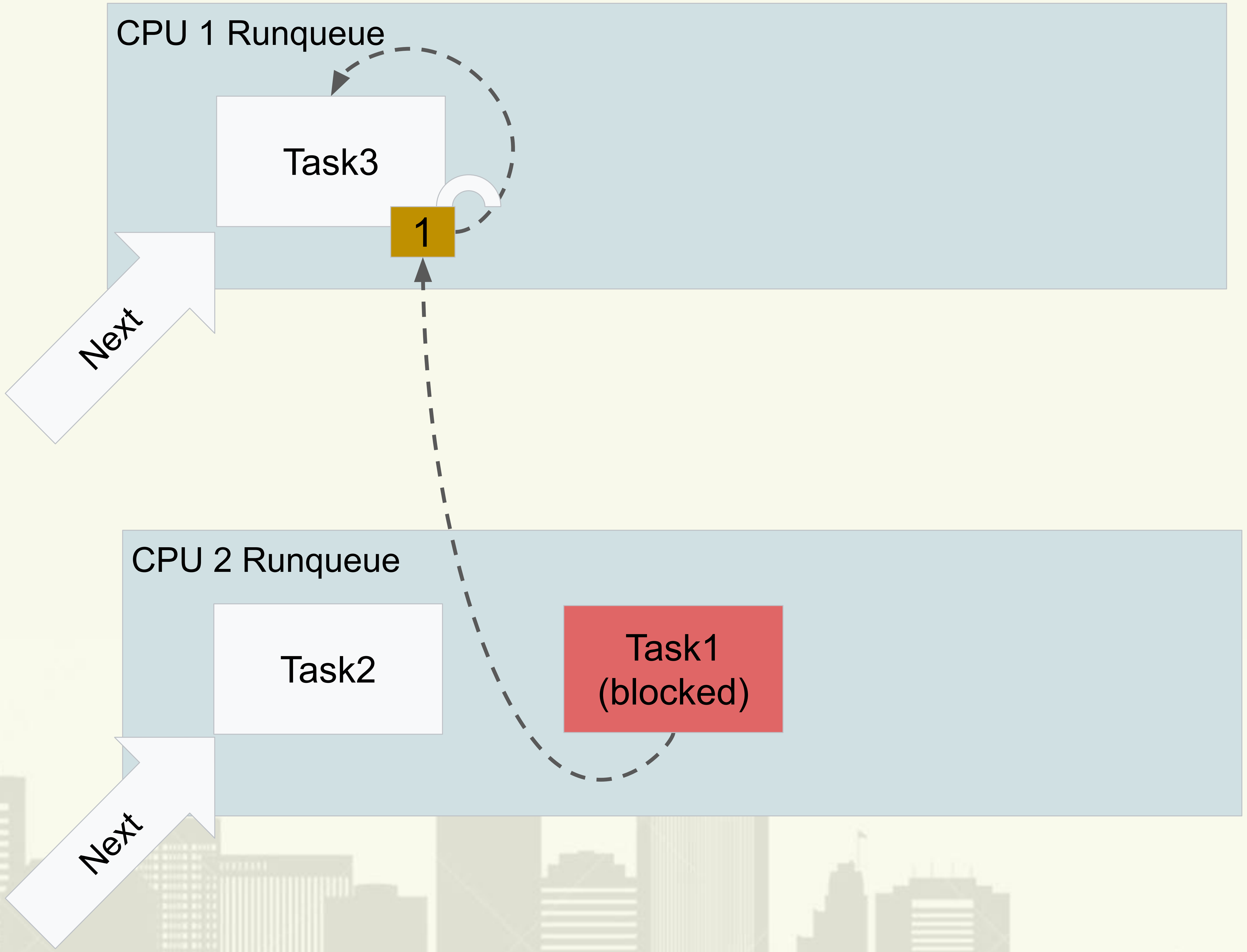
Proxy migration



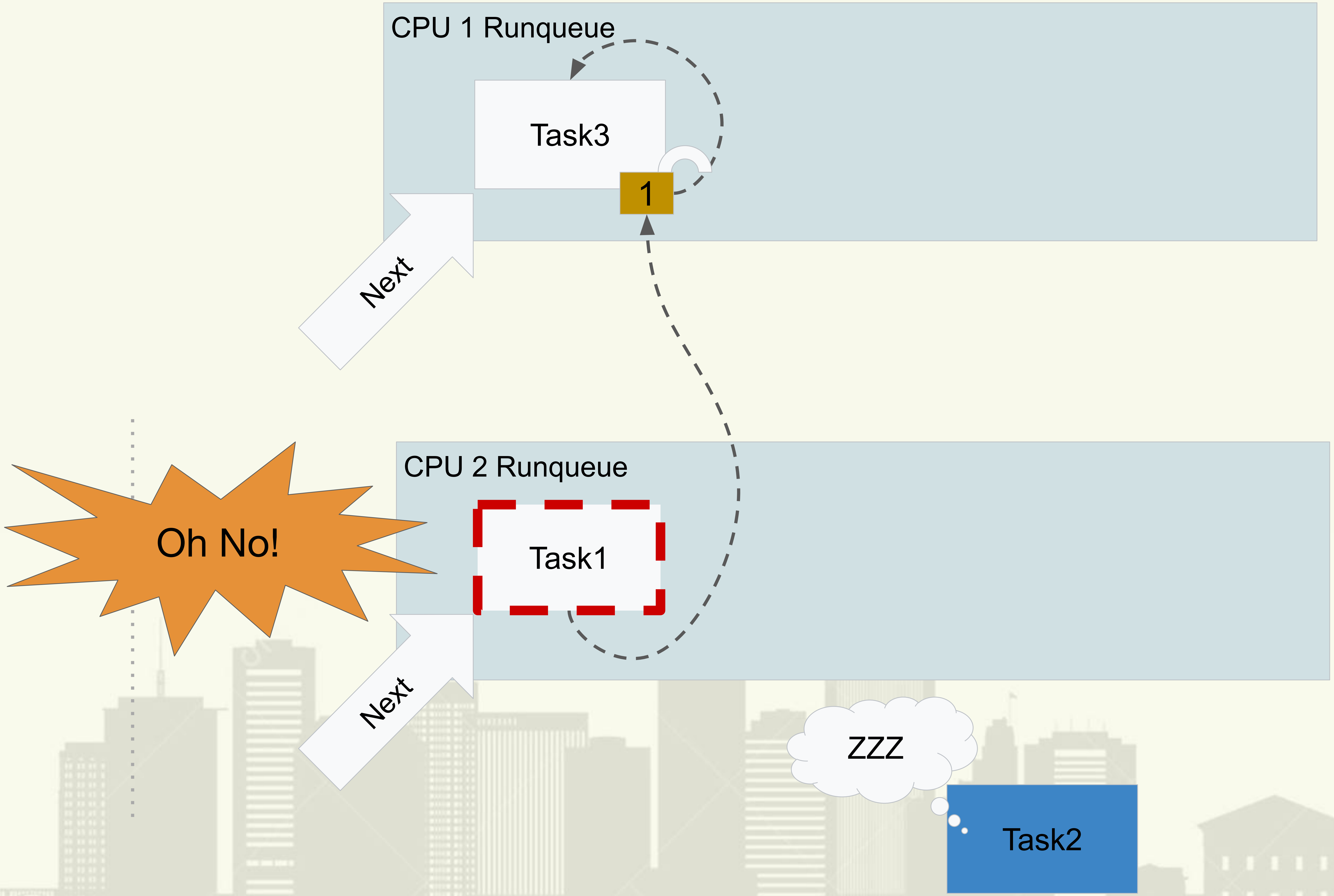
Proxy migration



Proxy migration



Proxy migration



Complications

- In v4 and earlier, we cleared the blocked_on state in try_to_wakeup() called from mutex_unlock_slowpath() on from lock handoff
  - This would deactivate the task, set_task_cpu() back to a runnable cpu and activate it.
  - But multiple migrations can happen, so its possible we hand the lock off & clear the blocked_on relationship while waiter was on a different cpu
  - This makes it immediately runnable, possibly on a cpu it can not run on!

Complications

- In v5 I moved this racy return migration logic out of try_to_wakeup() and into __schedule(). When we have selected a task to run, we double check its runnable on the current cpu, and if not migrate it back.
  - Problem: In __schedule() we hold the *current cpu* rq lock
  - We need task->pi_lock to set_task_cpu() and we also need rq lock for destination cpu.
  - unlock current cpu rqlock, take task->pi lock, take current cpu rqlock, deactivate task, set_task_cpu(), drop current cpu rqlock, take dest rqlock activate task, drop dest rqlock, take current cpu rqlock, drop task->pi lock.
  - Terrible amount of lock juggling!

Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023