New Locks for the Old Kernel

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Outline

1. **Background**: Locks, Locks in the Kernel, NUMA
2. **CNA**: Compact NUMA-aware Lock
3. **BRAVO**: Biased Reader-Writer Locking
4. **Conclusion**

ORACLE

Locks: Quick Background

- Protect access to the shared data

- Remain the most popular synchronization technique
- … and the topic of extensive research

- Performance of parallel software often depends on the efficiency of the locks it employs

ORACLE

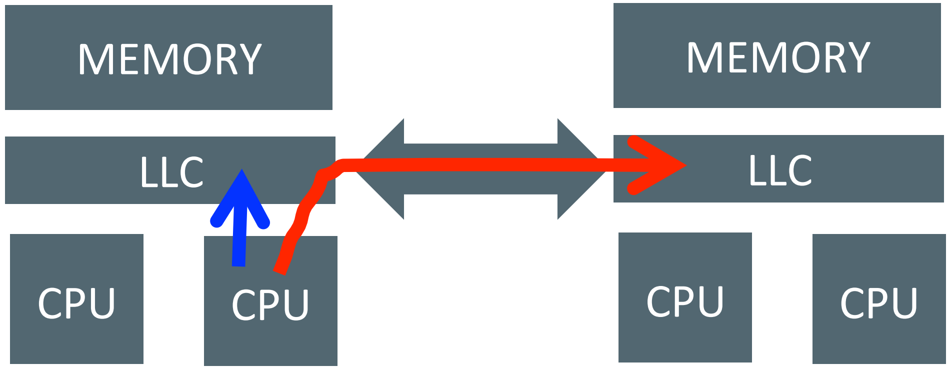
Locks: Quick Background (cont.)

- Many flavors:
  - exclusive / reader-writer
  - spinning / blocking
  - strictly fair / unfair / long-term fair
  - …

- Evolve with the evolution of computing architectures
  - we live in the era of multi-socket architectures with NUMA effects → we need *NUMA-aware* locks

ORACLE

NUMA-aware Locks



- Access by a core to a local memory or local cache is faster that accesses to a remote memory or remote cache
  - known as Non-Uniform Memory Access (NUMA) effect

- Keep the lock ownership ***within the same node***
  - decrease remote cache misses and inter-node communication
  - non-FIFO and unfair over the short term
  - ➢ trade-off short-term fairness for better performance

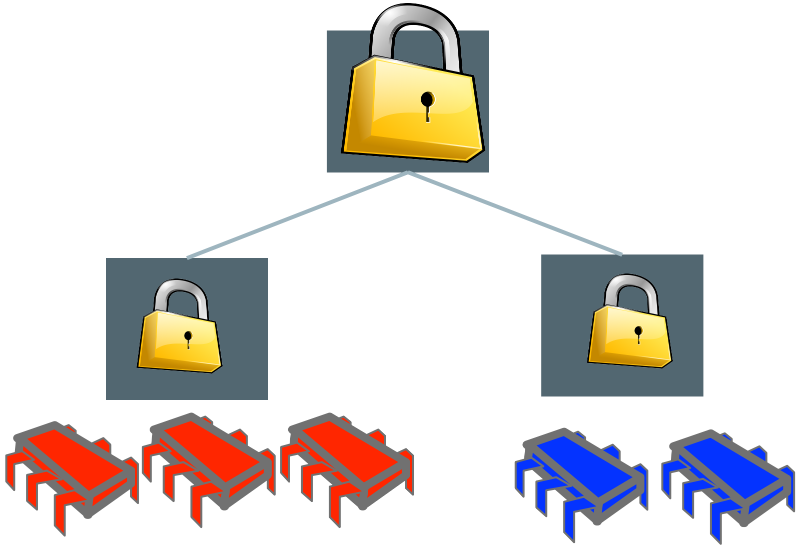
Locks in the Kernel

- Keep evolving
  - spinlocks: test-set → ticket → MCS (sort-of)
- Fail to keep up with the latest and greatest
  - e.g., spinlocks are not NUMA-aware, read-write locks use a shared counter
- Very specific requirements
  - compact
    - spinlock state must occupy at most 4 bytes
  - fair
  - good low thread-count performance

ORACLE

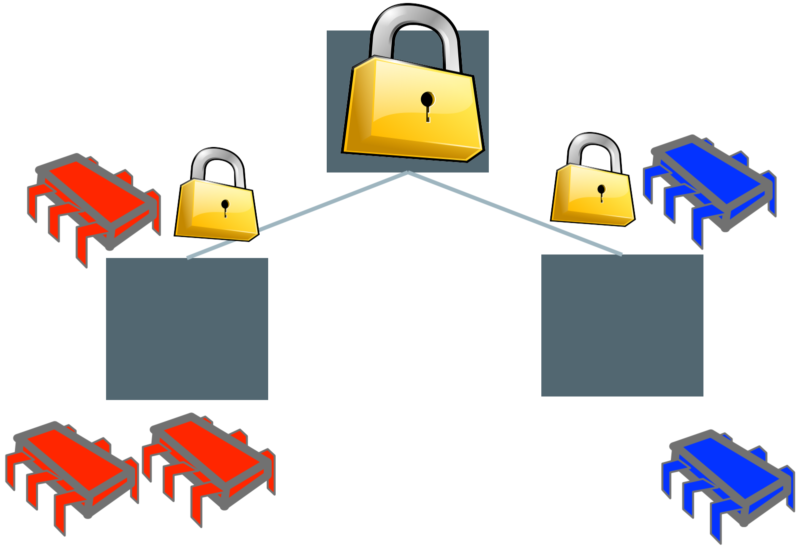
Hierarchical NUMA-aware Locks

- Multiple (2+) layers of lock hierarchy



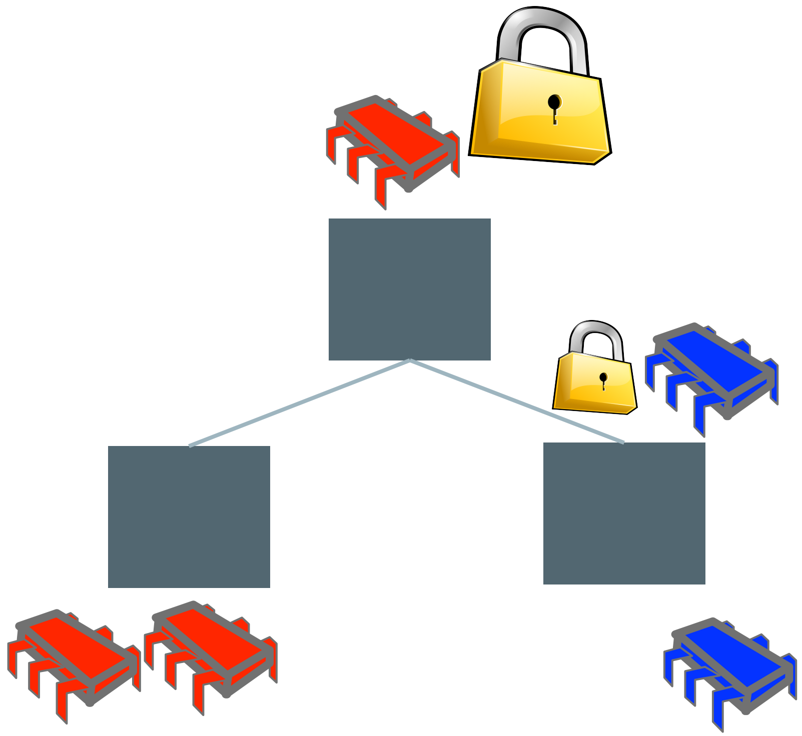
Hierarchical NUMA-aware Locks

- Multiple (2+) layers of lock hierarchy
- Acquire intra-node lock(s) first, then compete for the root lock



Hierarchical NUMA-aware Locks

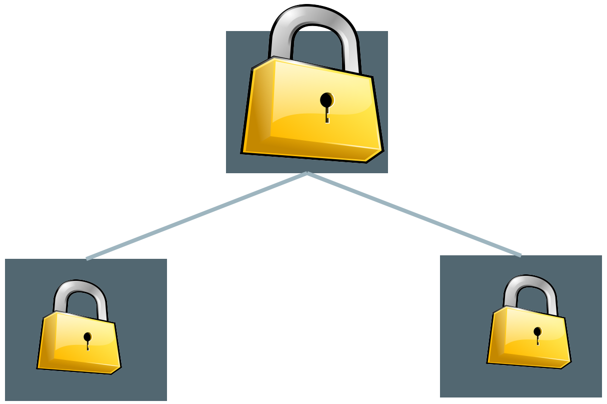
- Multiple (2+) layers of lock hierarchy
- Acquire intra-node lock(s) first, then compete for the root lock
- The root lock stays locked by threads running on the same socket
  - passing the intra-node lock passes the ownership



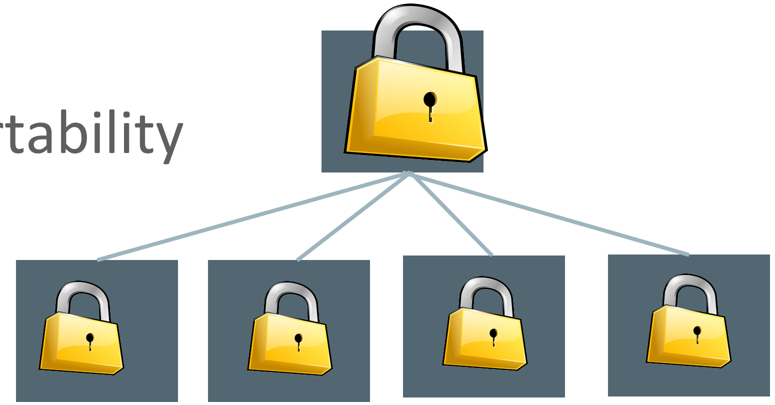
ORACLE

The Pitfalls of the Hierarchical Approach

- Longer acquisition path
  - multiple atomic instructions



- Require dynamic initialization to ensure portability
  - but the lack of standard API to query topology hinders portability



- SIZE: space proportional to #nodes
  - to make matters worse, each low-level lock has to be placed on a separate cache line

Where (and Why) Size Matters?

- Concurrent data structures with one lock-per-node/entry
  – E.g., binary search trees, linked lists, etc.

- Systems with millions+ of locks
  – E.g., Linux Kernel
    - spinlocks are embedded in every inode and page structure
    - ➢ one lock per file and per physical page
    - ➢ strict limit of 4 bytes (32bits) on the spinlock size

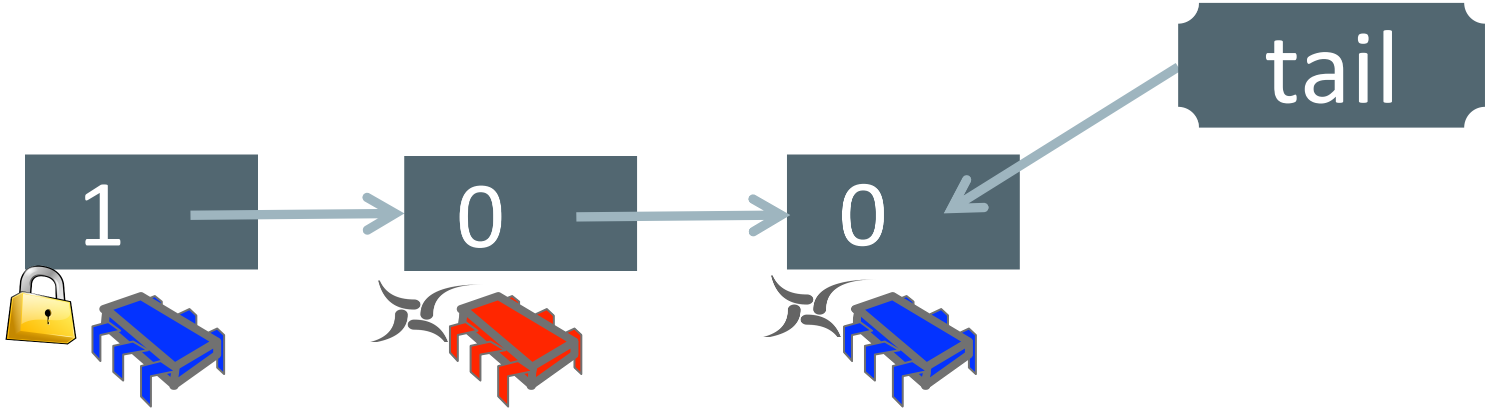
ORACLE

CNA: Compact NUMA-aware Lock

- Requires one word of memory
- Variant of a (NUMA-oblivious) MCS lock
  - inherits its performance features
    - local spinning, one atomic operation per acquisition, …
- Performance on-par with MCS under no contention, on-par with state-of-the-art hierarchical NUMA-aware locks when contended

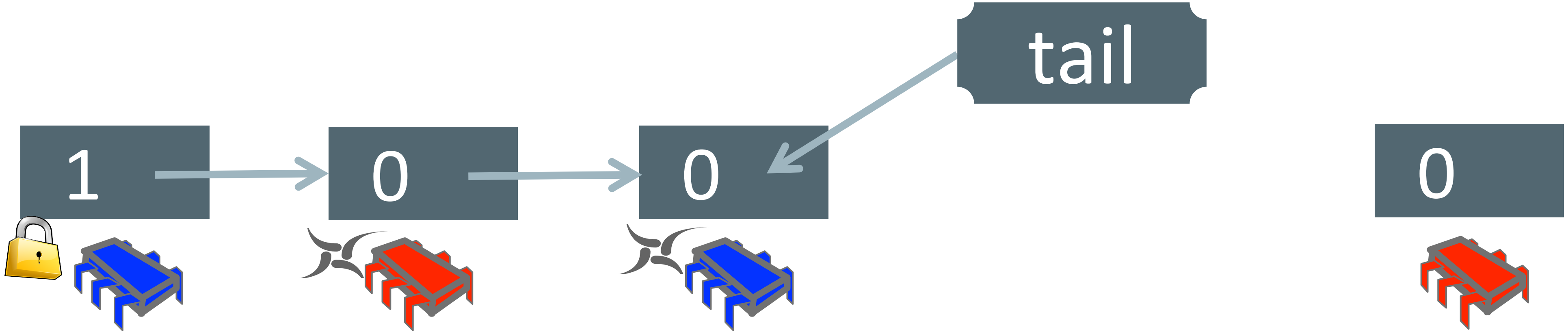
MCS (Mellor-Crummey and Scott) Lock

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record



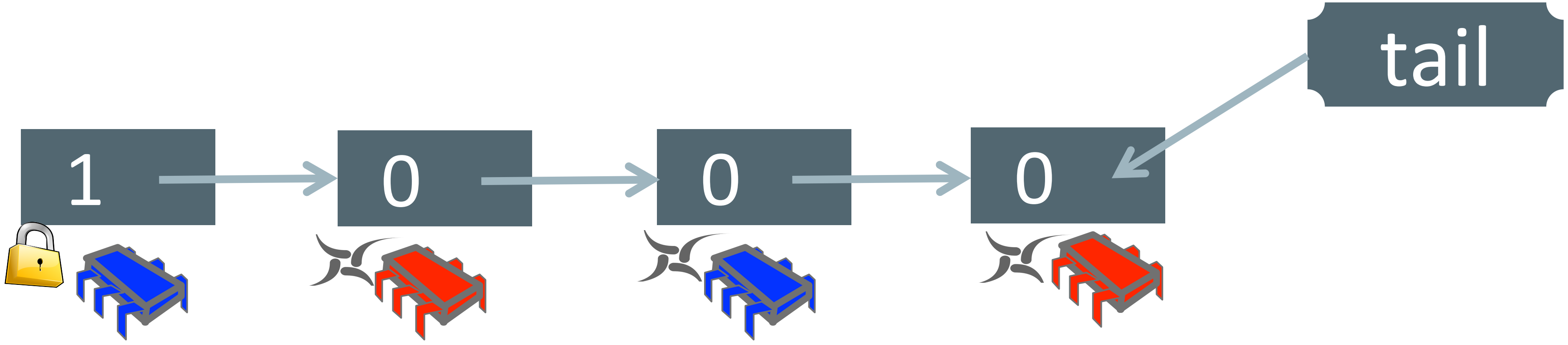
MCS (Mellor-Crummey and Scott) Lock

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record



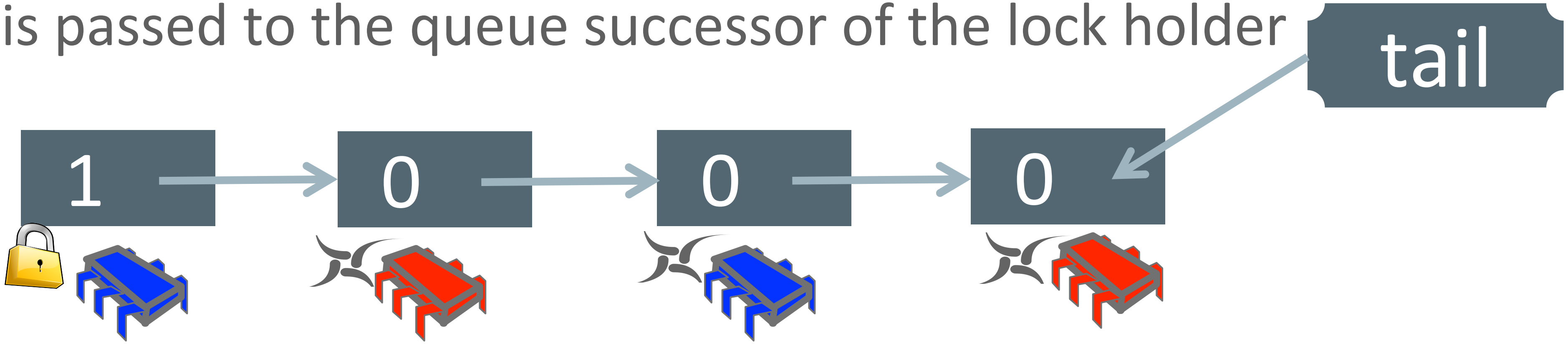
MCS (Mellor-Crummey and Scott) Lock

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record



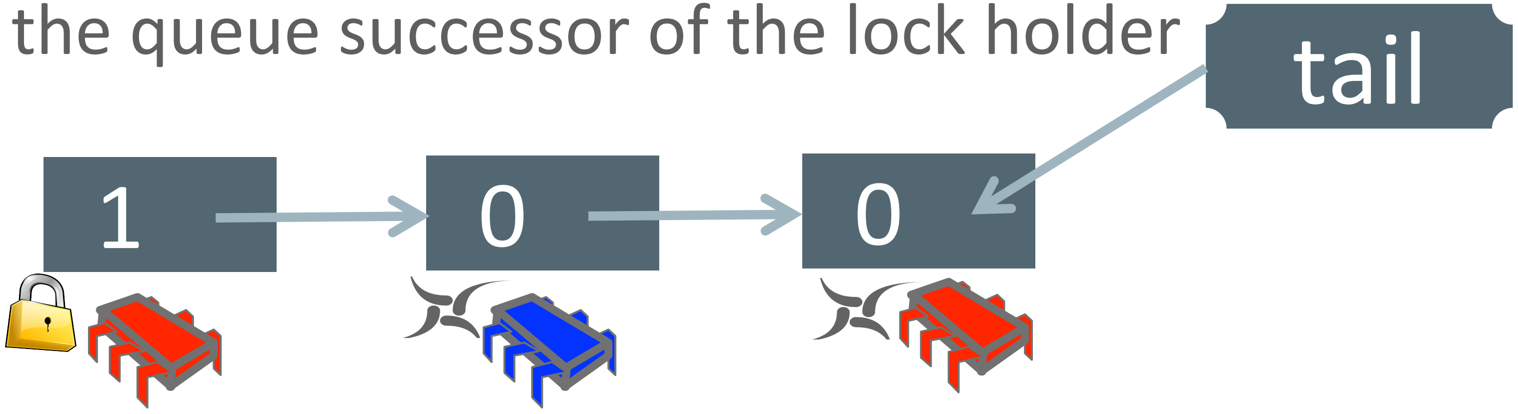
MCS (Mellor-Crummey and Scott) Lock

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder



MCS (Mellor-Crummey and Scott) Lock

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder



CNA: a NUMA-Aware Variant of MCS

- Organizes waiting threads in a FIFO queue
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder

CNA: a NUMA-Aware Variant of MCS

- Organizes waiting threads in **two queues (“main” and “secondary”)**
- The shared state is a pointer to the tail of the queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder

ORACLE

CNA: a NUMA-Aware Variant of MCS

- Organizes waiting threads in **two queues ("main" and "secondary")**
- The shared state is a pointer to the tail of the **main** queue
- Each thread has a record that it inserts into the queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder

ORACLE

CNA: a NUMA-Aware Variant of MCS

- Organizes waiting threads in **two queues ("main" and "secondary")**
- The shared state is a pointer to the tail of the **main** queue
- Each thread has a record that it inserts into the **main** queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor of the lock holder

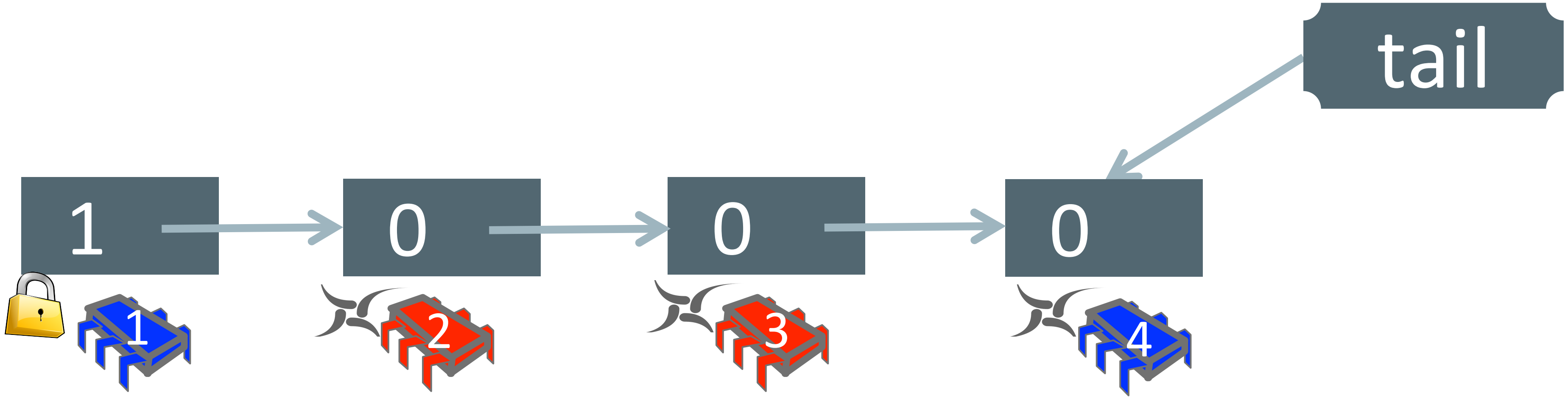
ORACLE

CNA: a NUMA-Aware Variant of MCS

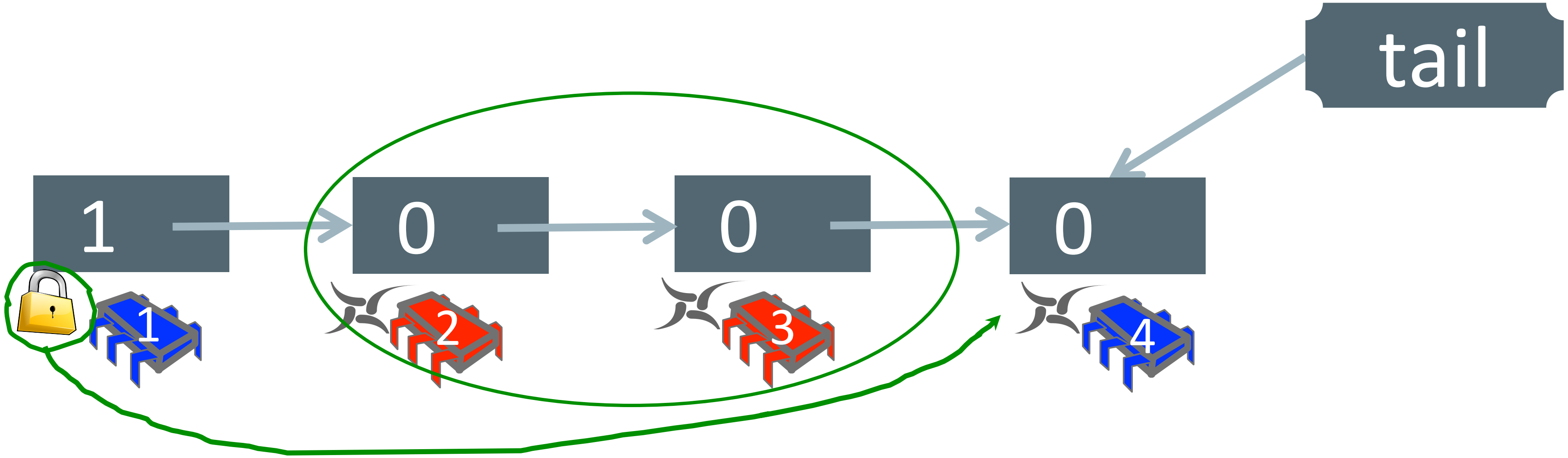
- Organizes waiting threads in **two queues (“main” and “secondary”)**
- The shared state is a pointer to the tail of the **main** queue
- Each thread has a record that it inserts into the **main** queue …
- … and then spins locally on a flag inside the record
- The lock is passed to the queue successor **running on the same node as the lock holder**
  - **waiting threads between the lock holder and its new successor are moved to the secondary queue so they do not interfere in subsequent lock handovers**

ORACLE

CNA in Action

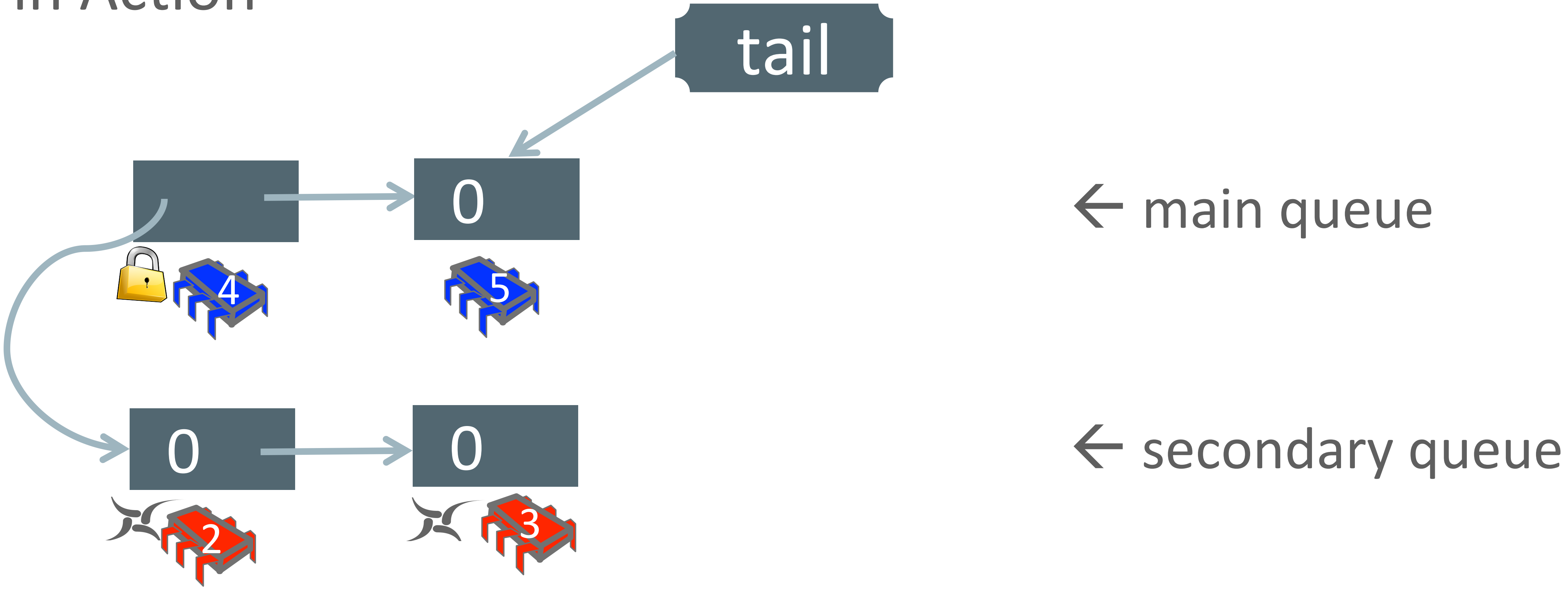


CNA in Action



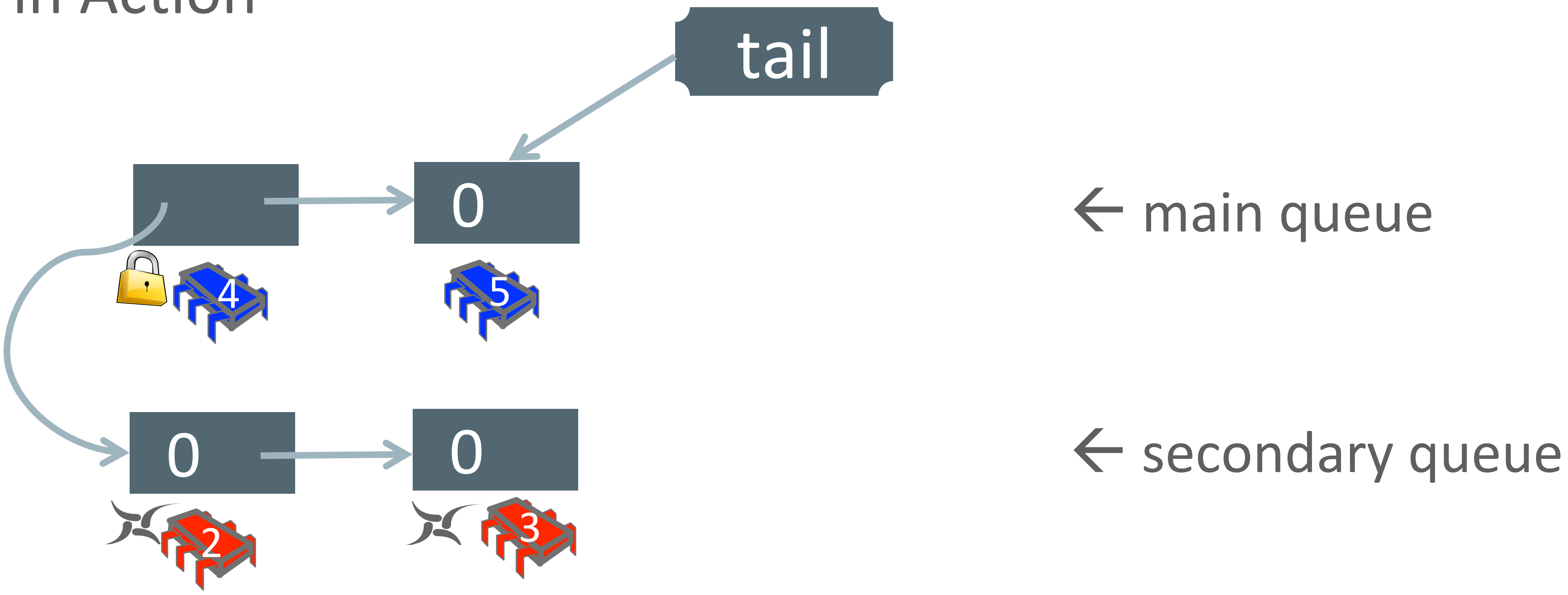
CNA in Action

CNA in Action



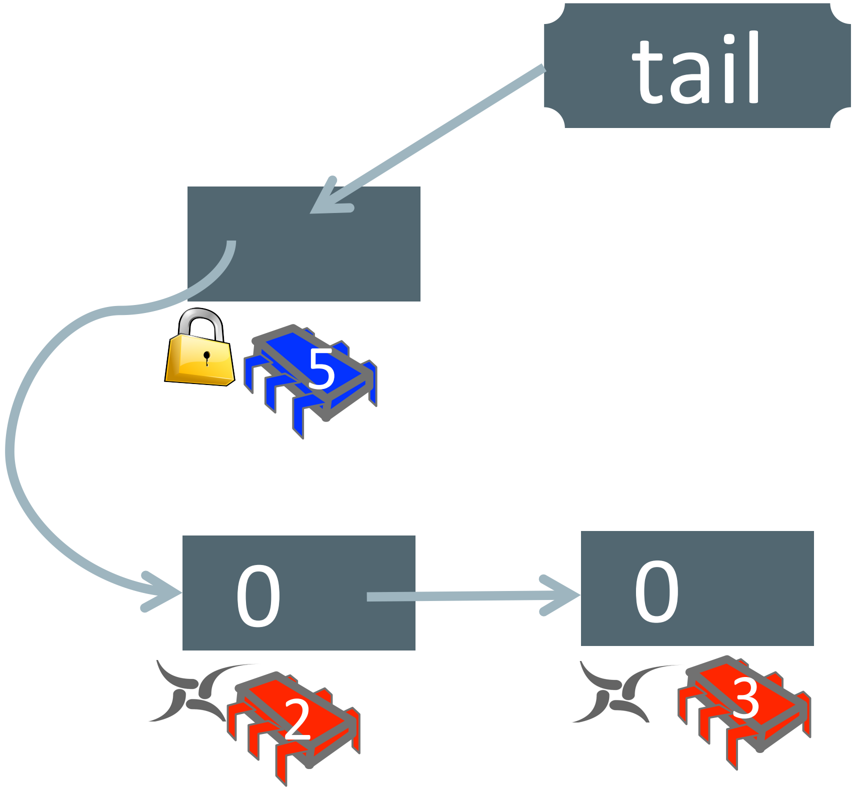
Thread 4: unlock()

CNA in Action



CNA in Action

CNA in Action

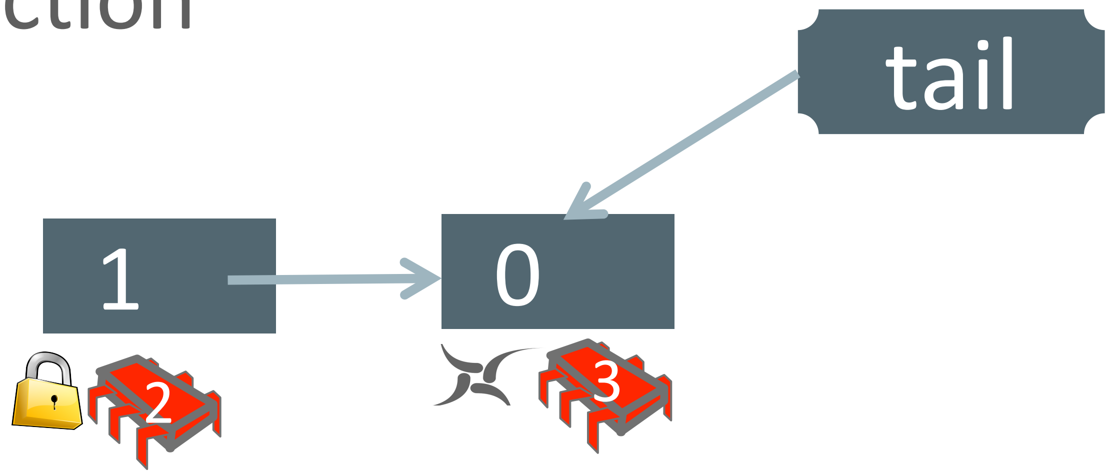


← main queue

← secondary queue

ORACLE

CNA in Action



Avoiding Starvation

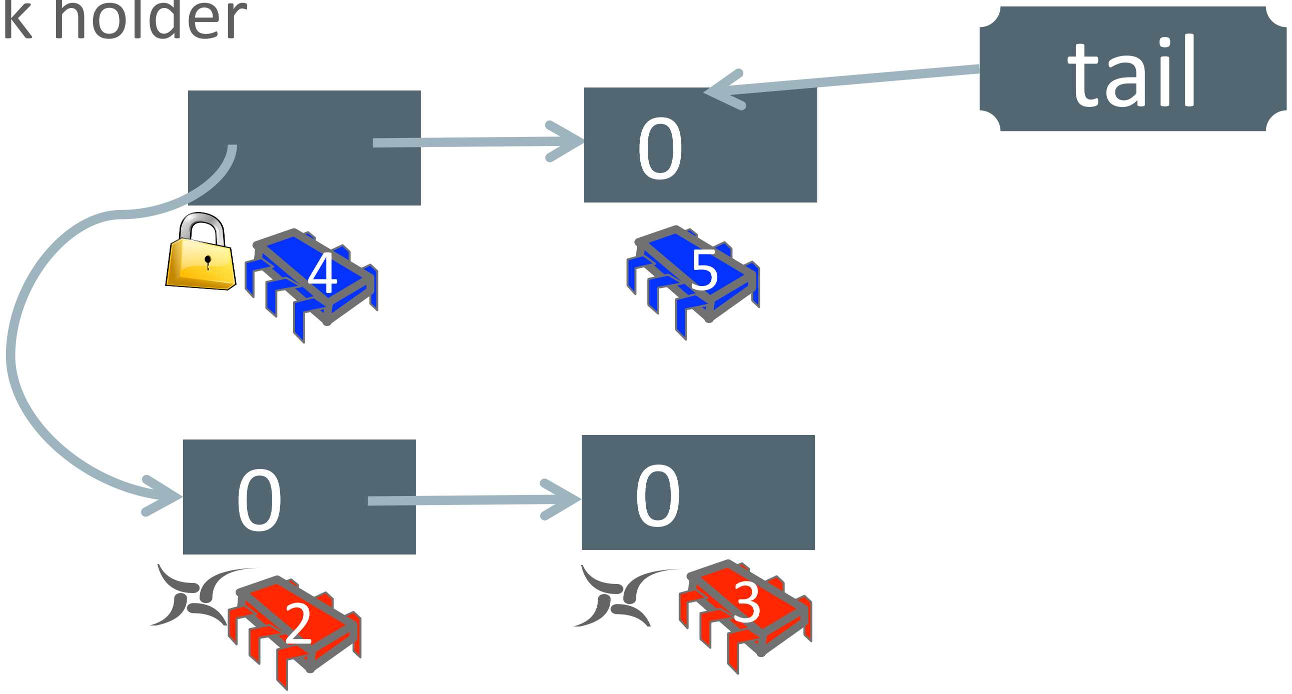
Move waiting threads back from secondary to main queue

1. When the main queue is empty / does not have threads on the same node as the lock holder

Avoiding Starvation

Move waiting threads back from secondary to main queue

1. When the main queue is empty / does not have threads on the same node as the lock holder



Avoiding Starvation

Move waiting threads back from secondary to main queue

1. When the main queue is empty / does not have threads on the same node as the lock holder
2. After a certain number of “intra-node” handovers
   - scan the main queue with high probability rather than always
   - can count deterministically, but incurs more overhead (cache misses to update count)
   - threshold controls fairness-VS-throughput trade-off

ORACLE

Performance Evaluation

User-space:

- Implemented CNA as a user-level library
- Compared to MCS, cohort locks (C-BO-MCS), HMCS lock

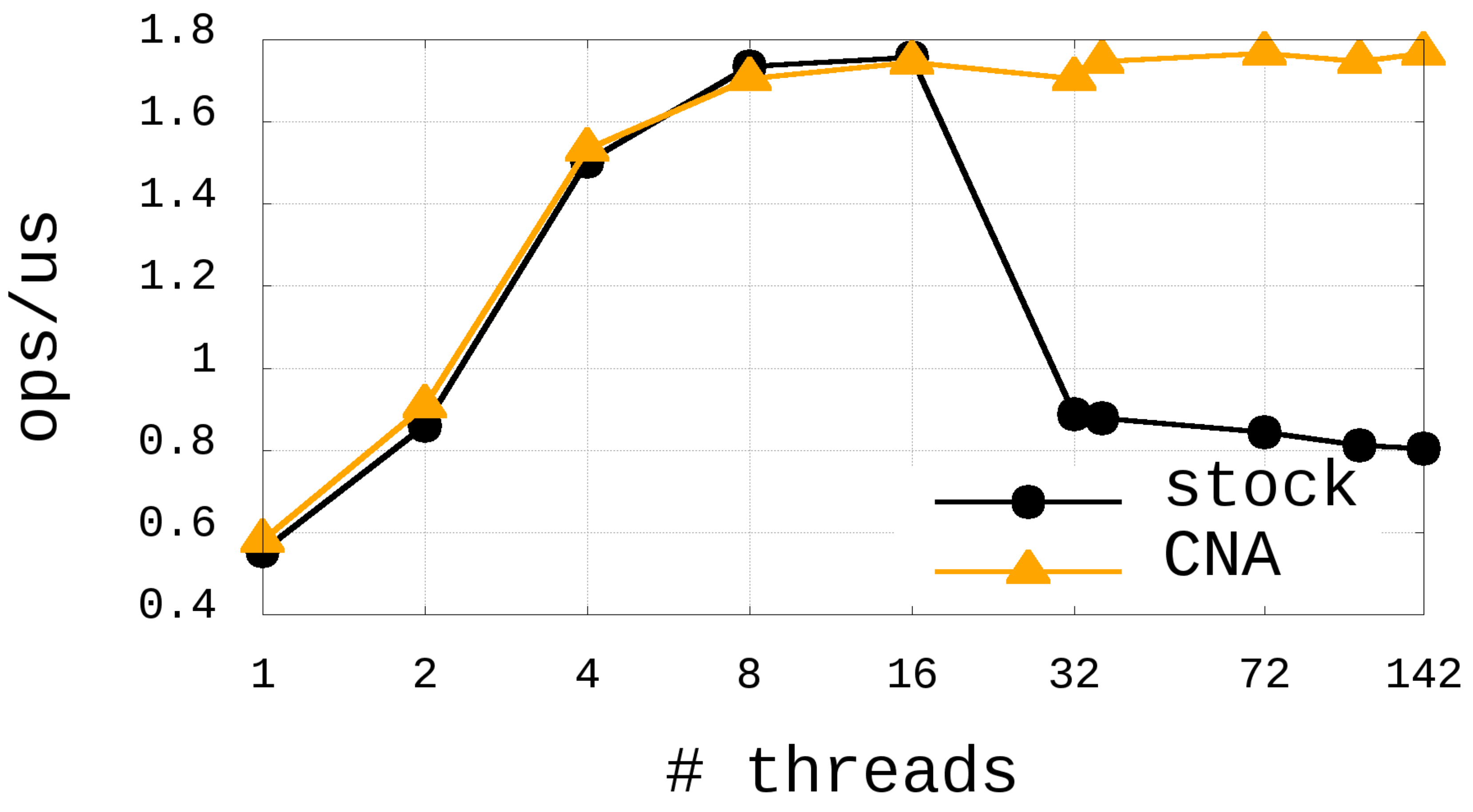
Kernel-space:

- Integrated into the slow path of qspinlock, Linux kernel spin-lock

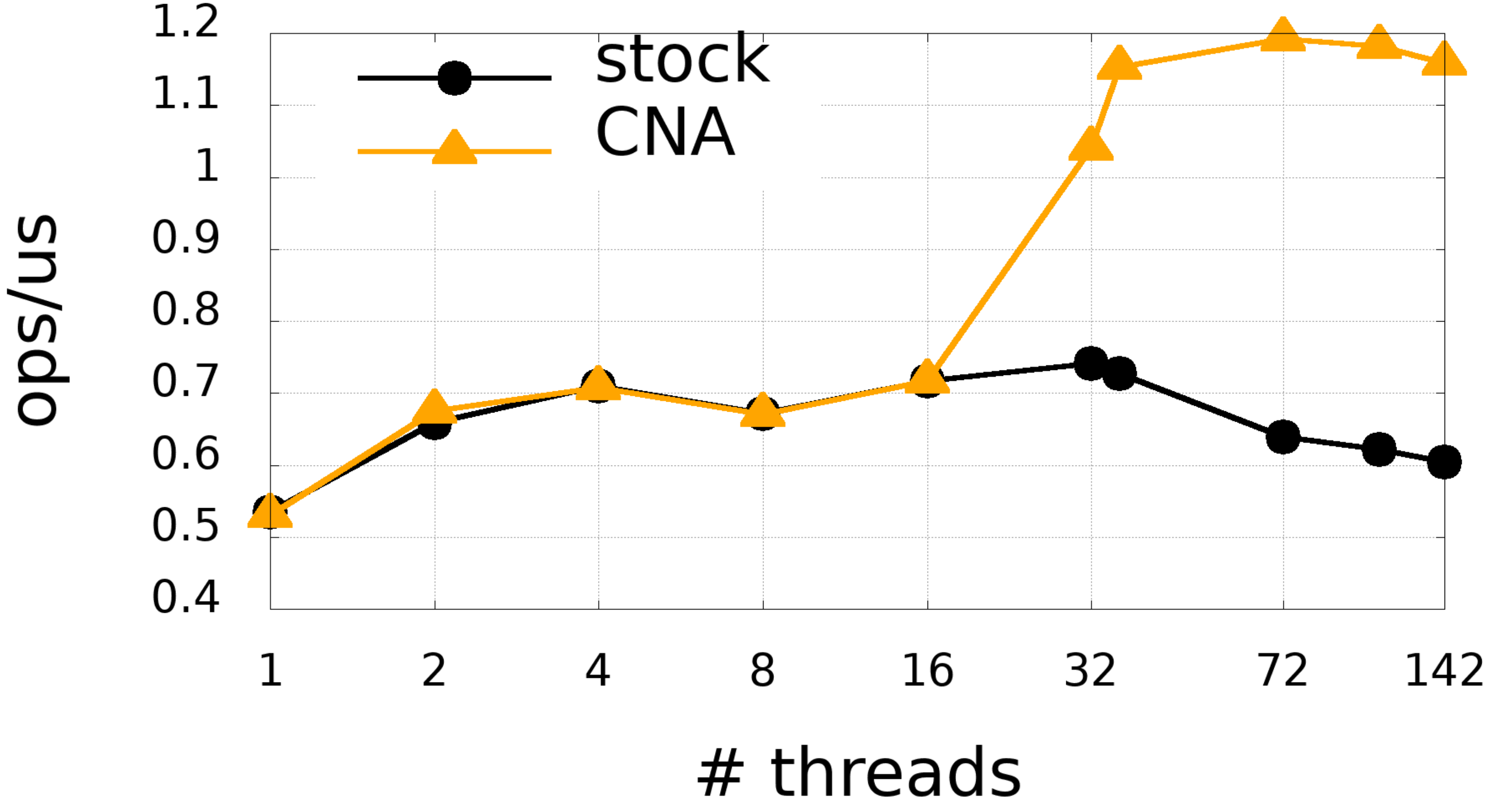
HW: 4-socket x86 machine, with 18 hyper-threaded cores per sockets

ORACLE

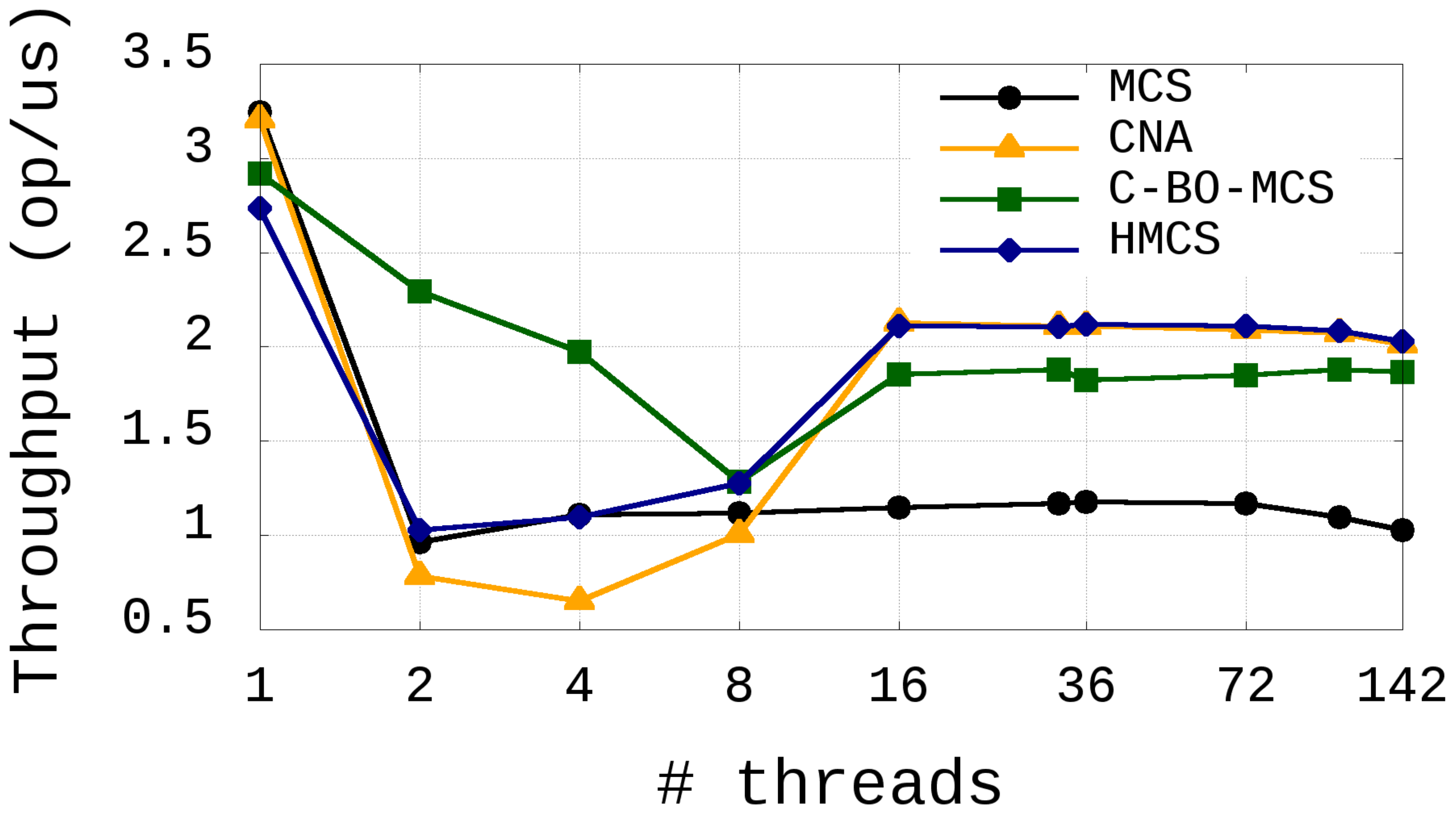
will-it-scale/open1_threads



LevelDB/readrandom



User-space: LevelDB

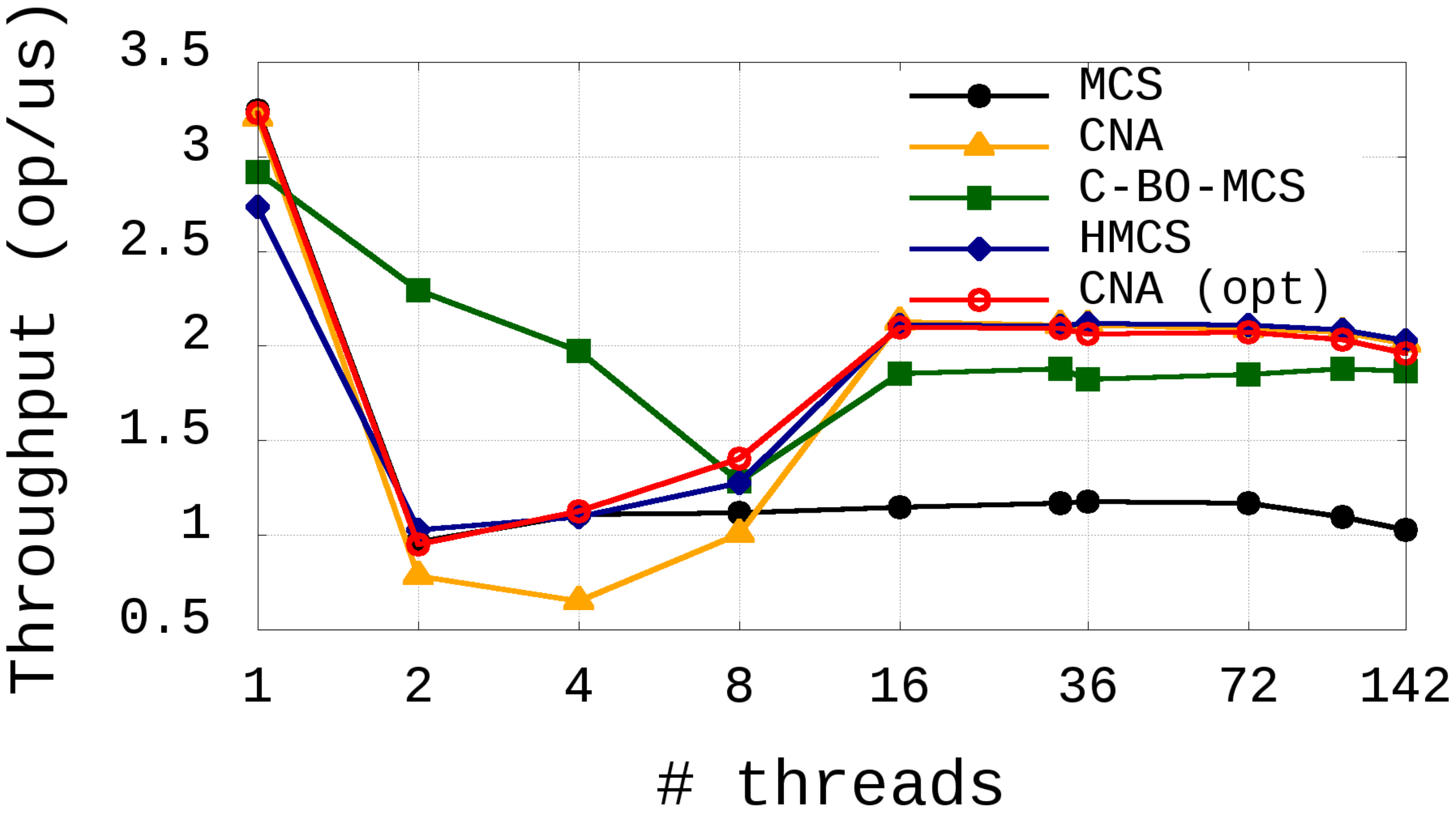


Shuffle Reduction Optimization

- Under light contention, waiting threads can be moved back and forth between two queues
  - creates overhead without reaping the benefit of locality
- Solution: when the secondary queue is empty, scan the main queue with low probability rather than always
  - reduces the amount of unnecessary shuffling when the contention is low, while responding fast enough when the contention is high

ORACLE

User-space: LevelDB



Wrap-up: CNA

CNA achieves the best of both worlds:

- as efficient as MCS at low contention
  - but better at high contention by 40-100%
- as performant as state-of-the-art NUMA-aware locks at high contention
  - but its state requires only one word of memory
- Reduces #remote cache misses while preserving long-term fairness

- Linux kernel patch is publicly available

ORACLE

Reader-Writer Locks: Quick Background

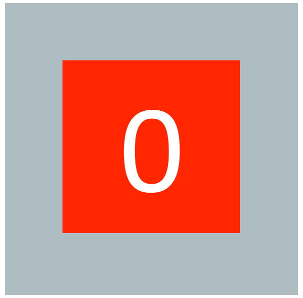
- Allow shared access for read-only use of a resource
- Ubiquitous in modern systems

Reader-Writer Locks: Quick Background

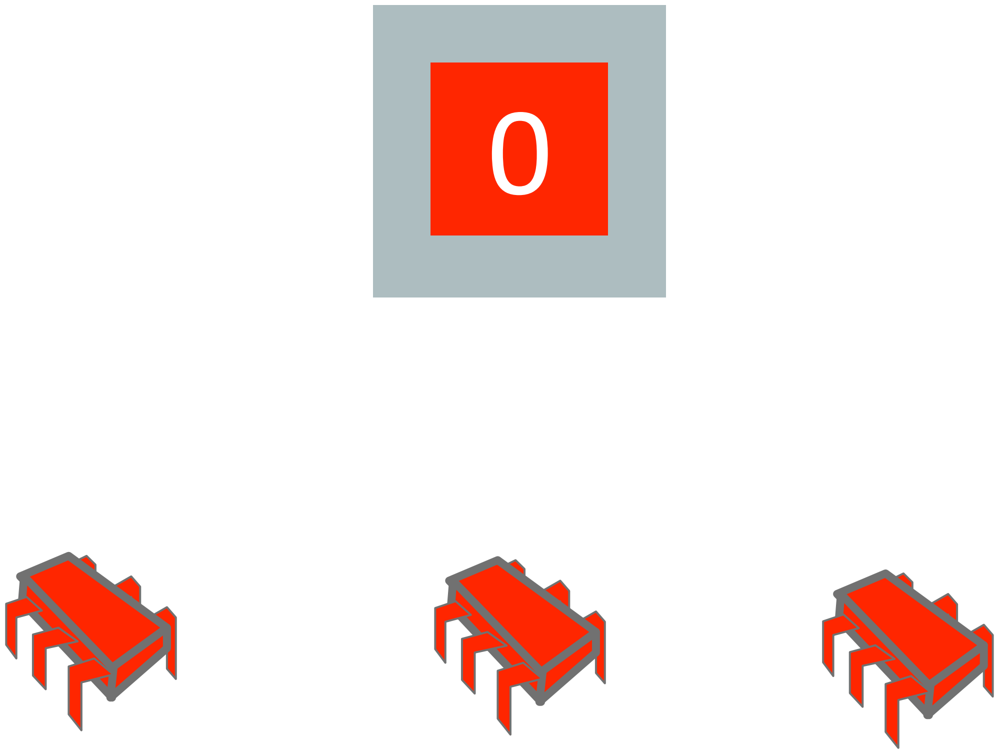
- Allow shared access for read-only use of a resource
- Ubiquitous in modern systems

- Have to keep track of the presence of active readers

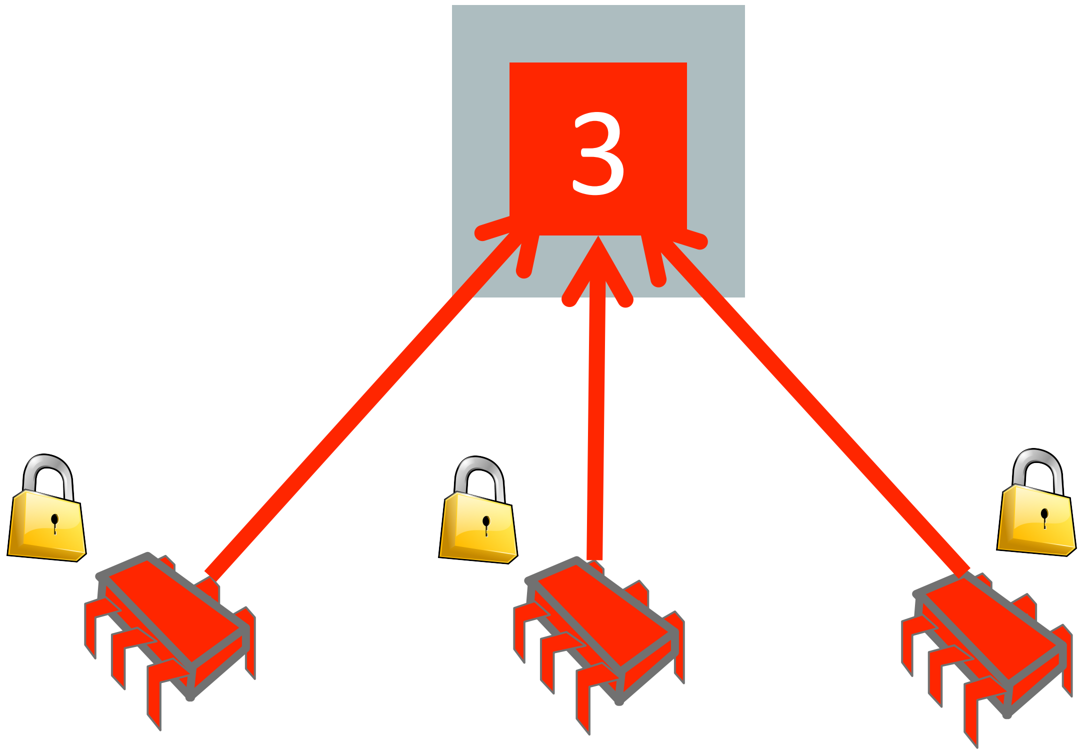
The “shared counter” approach



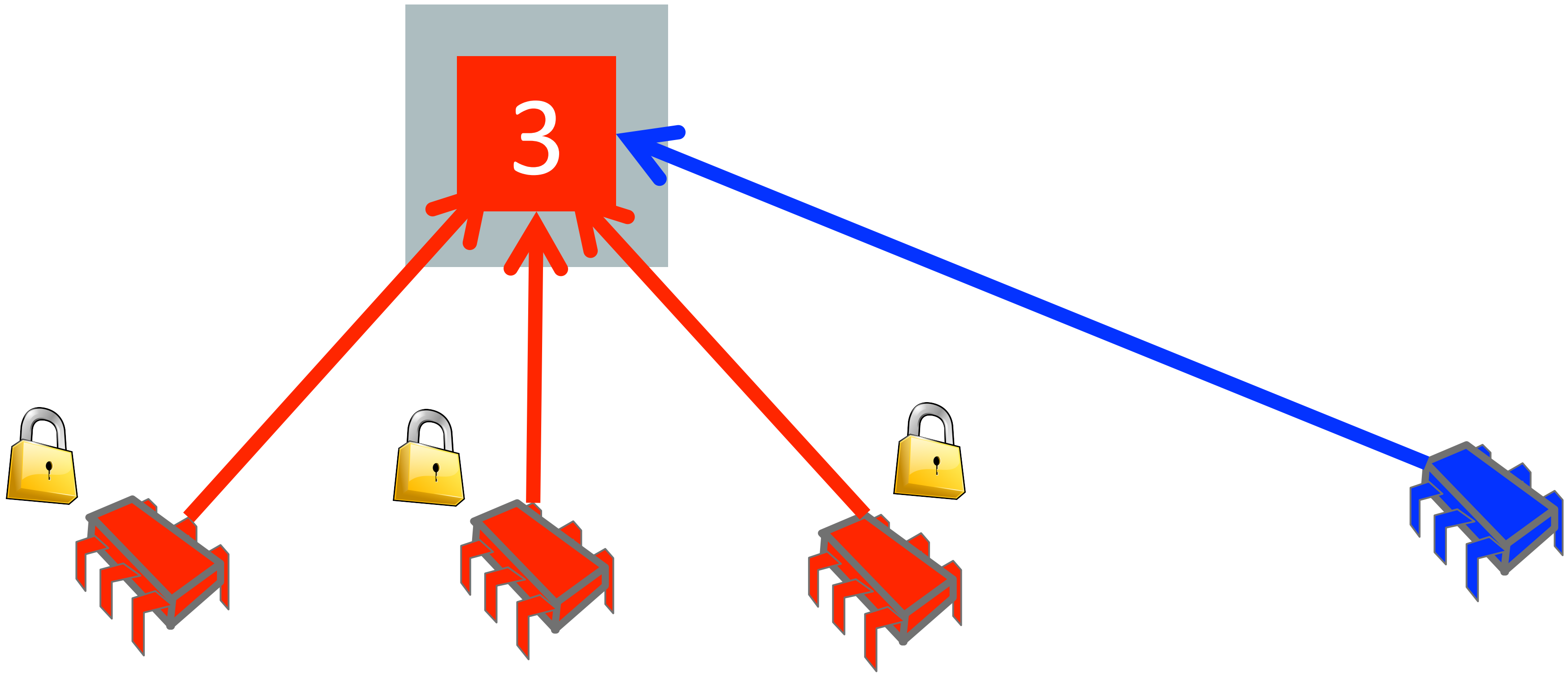
The “shared counter” approach



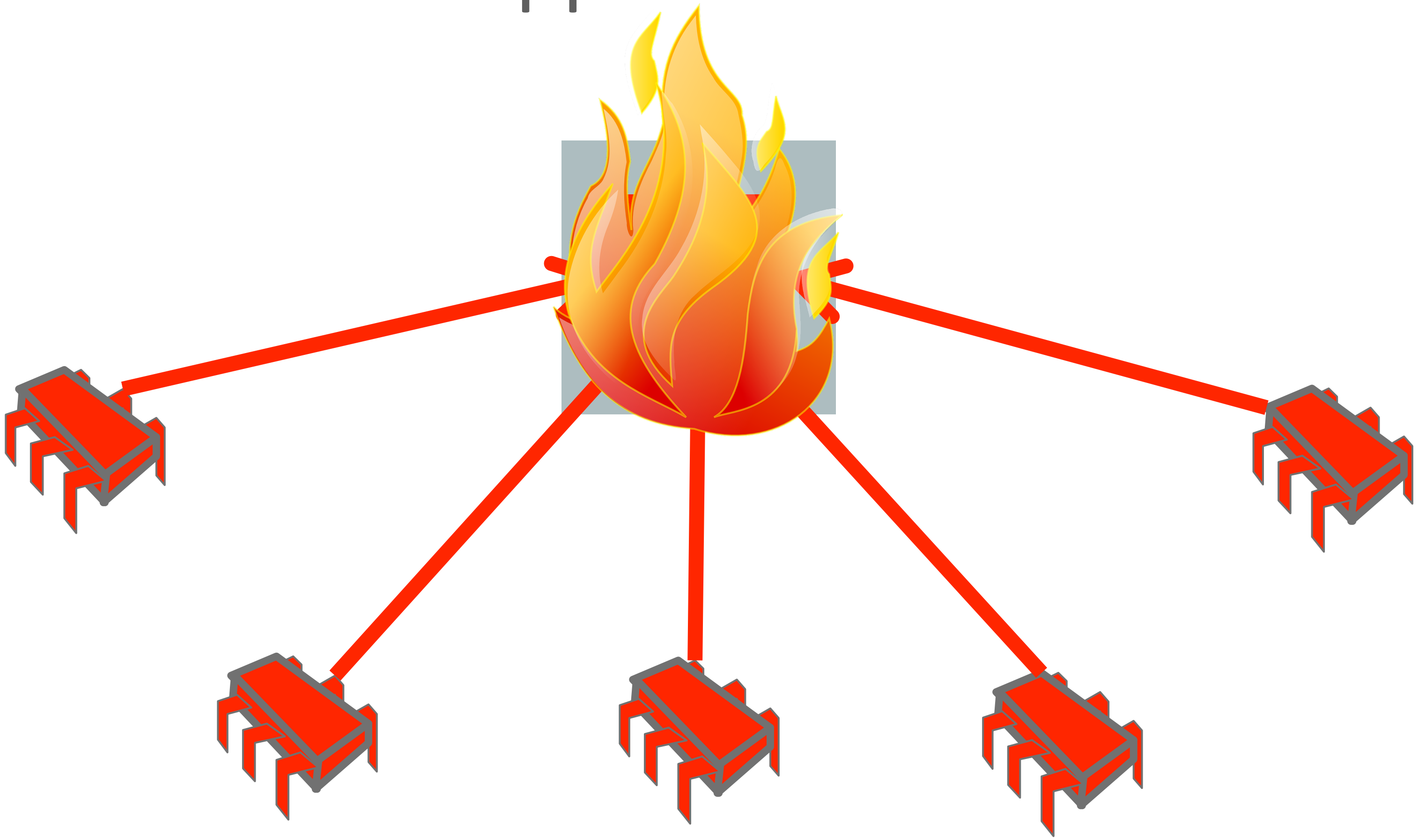
The “shared counter” approach



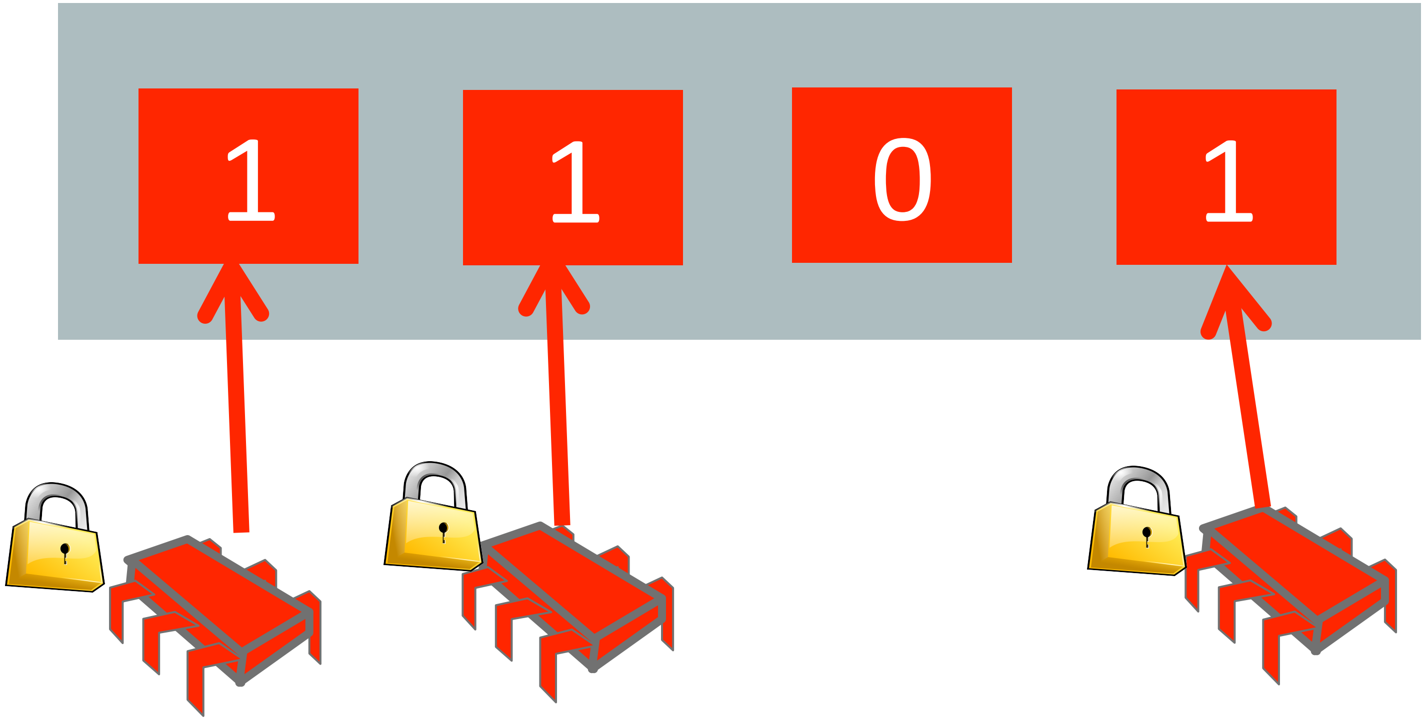
The “shared counter” approach



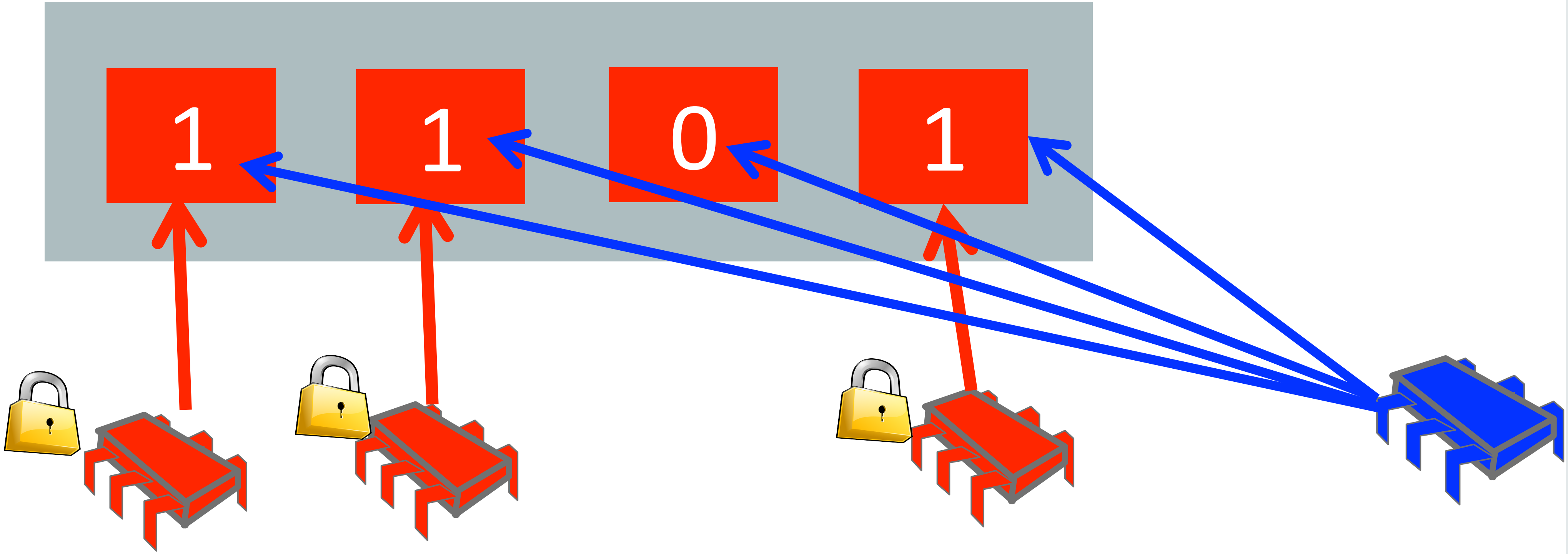
The “shared counter” approach



The "distributed" approach

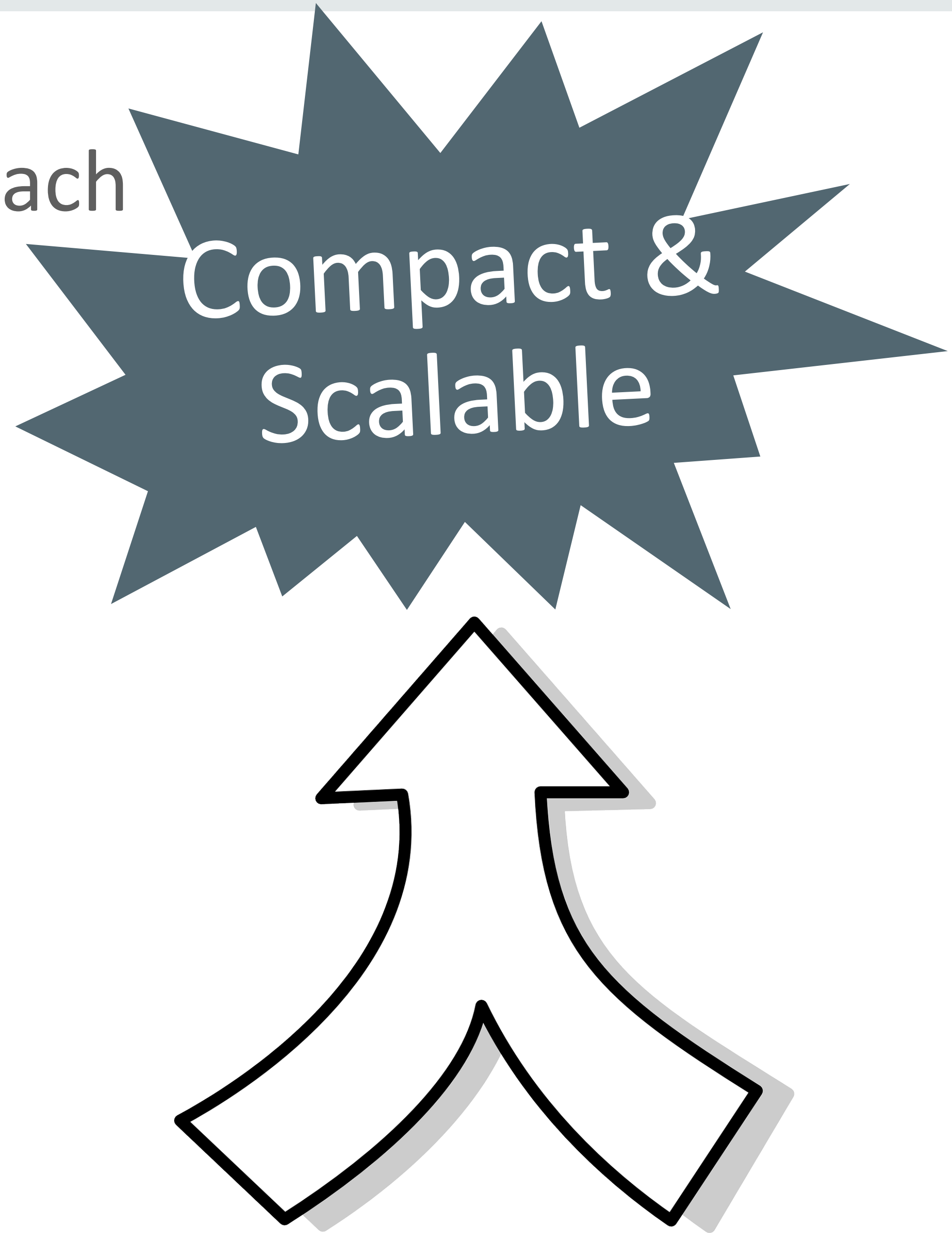


The “distributed” approach

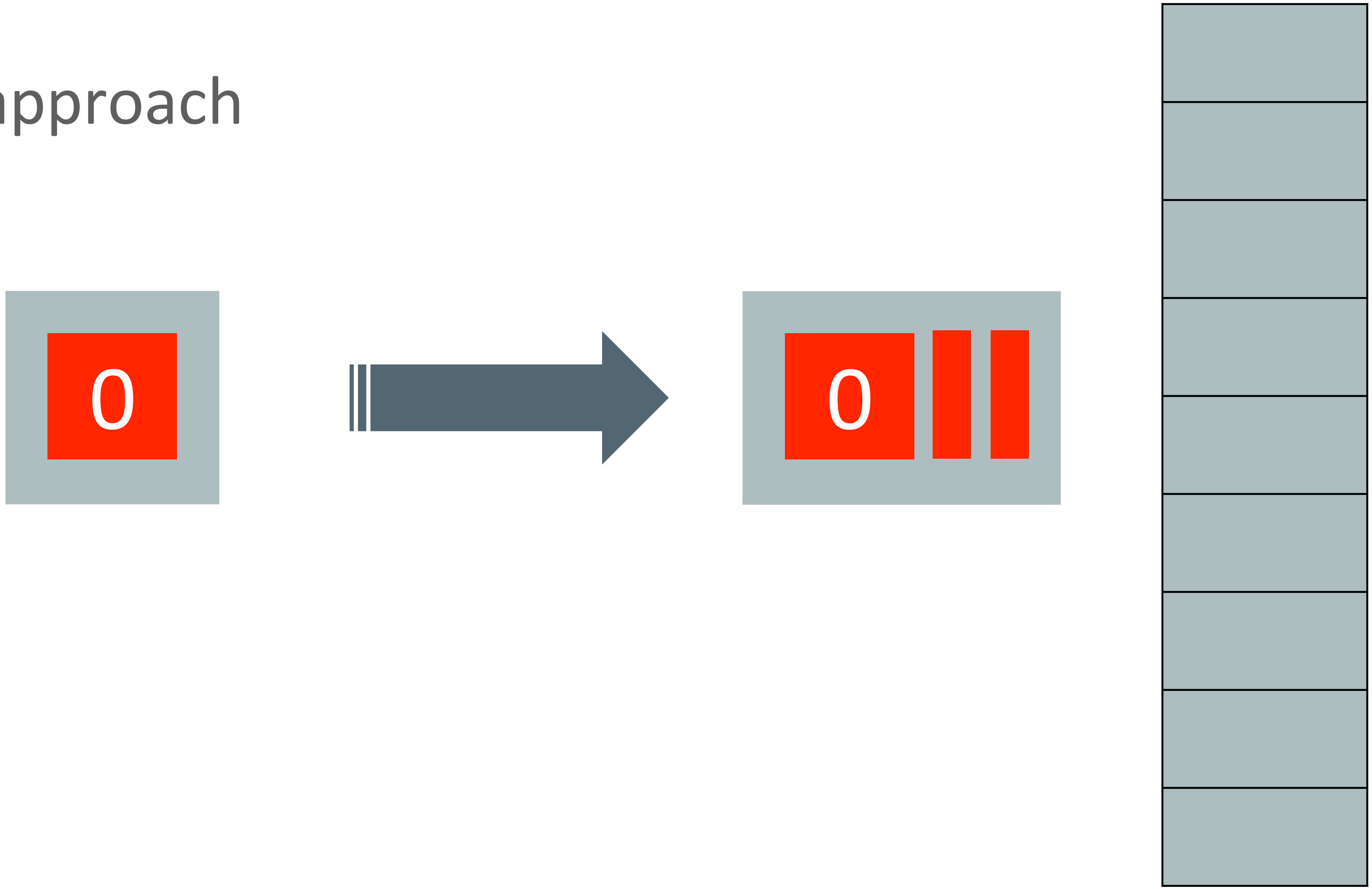


The Scalable "Reader Indicator" Dilemma

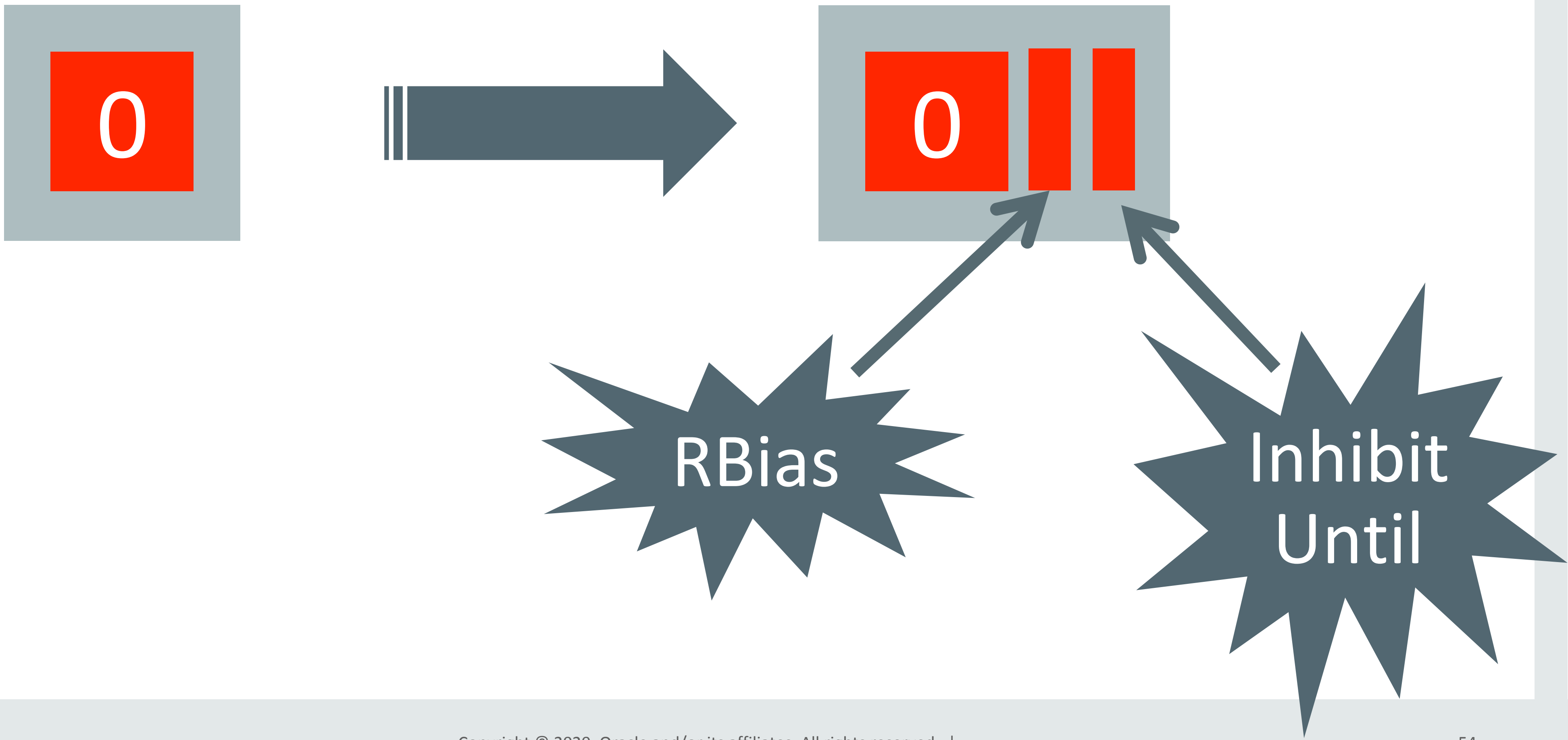
The BRAVO approach



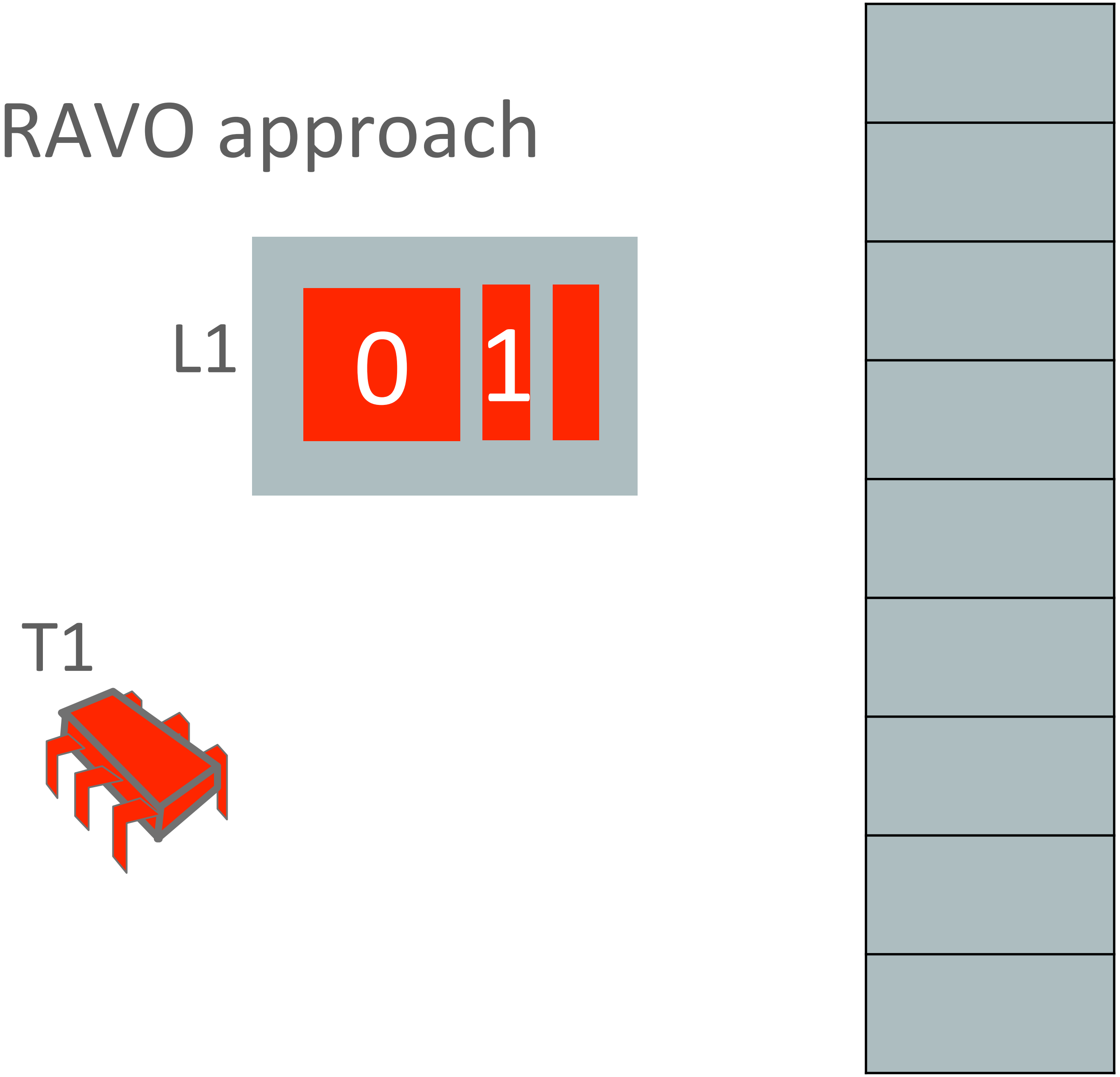
The BRAVO approach



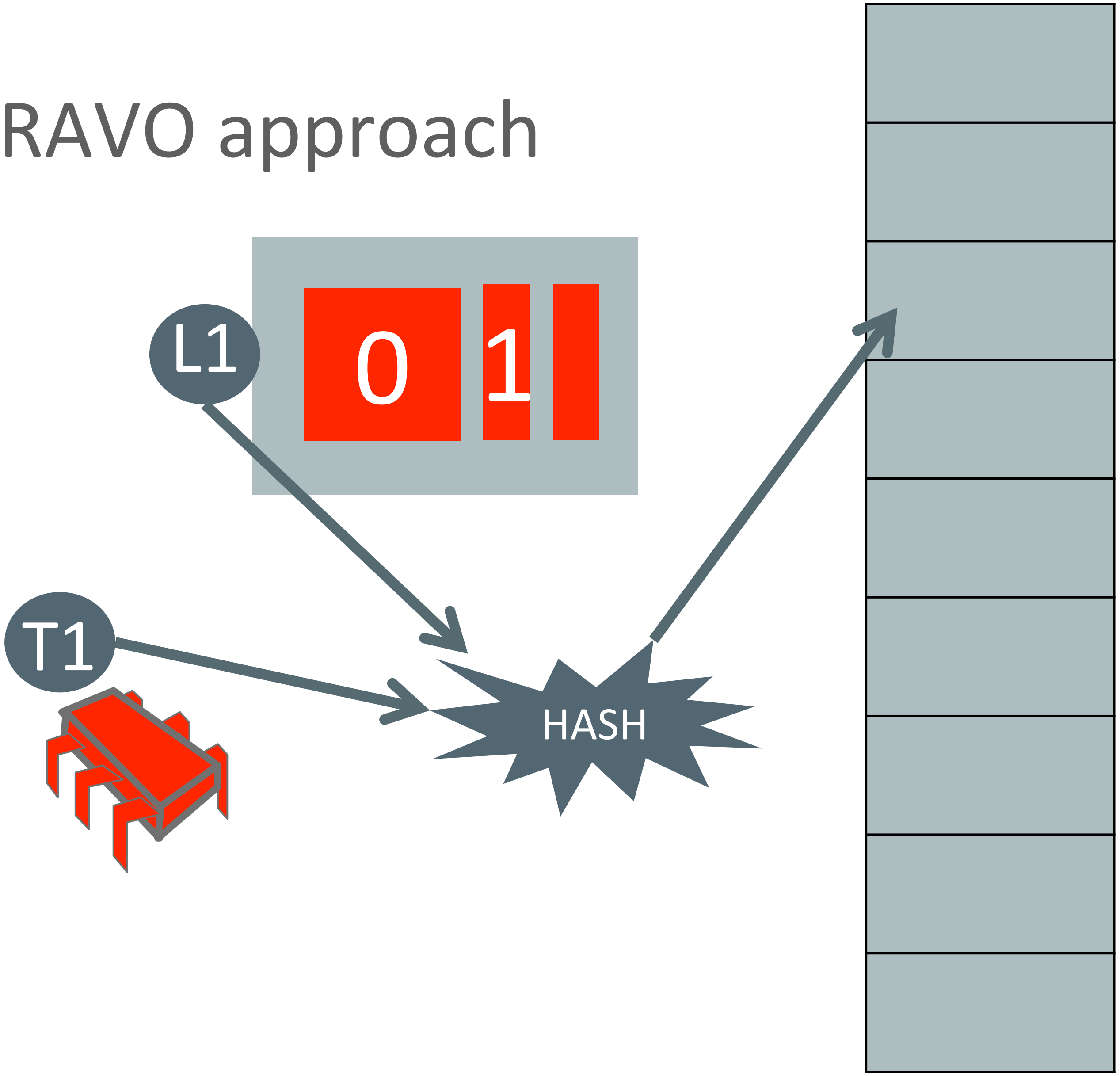
The BRAVO approach



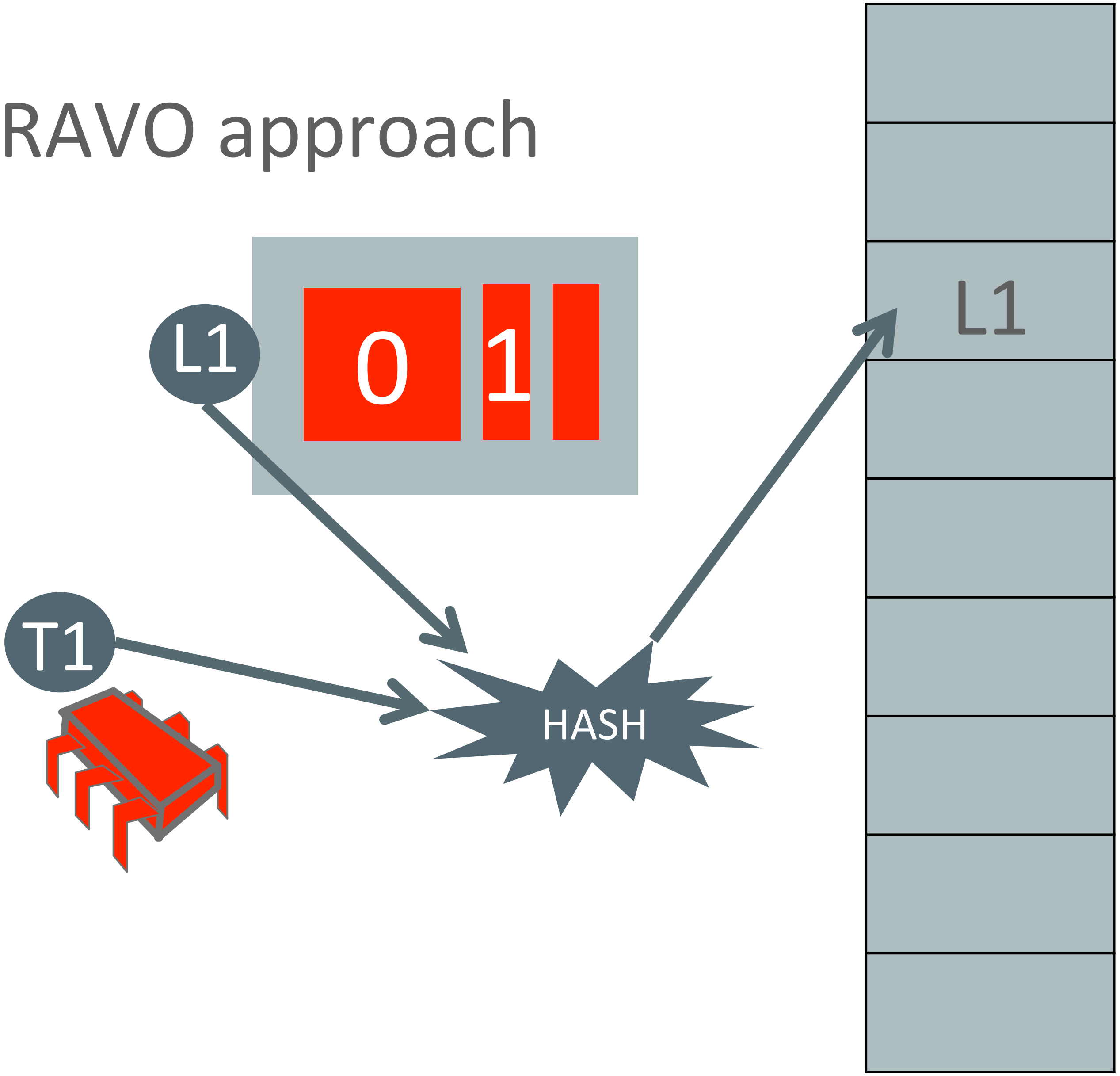
The BRAVO approach



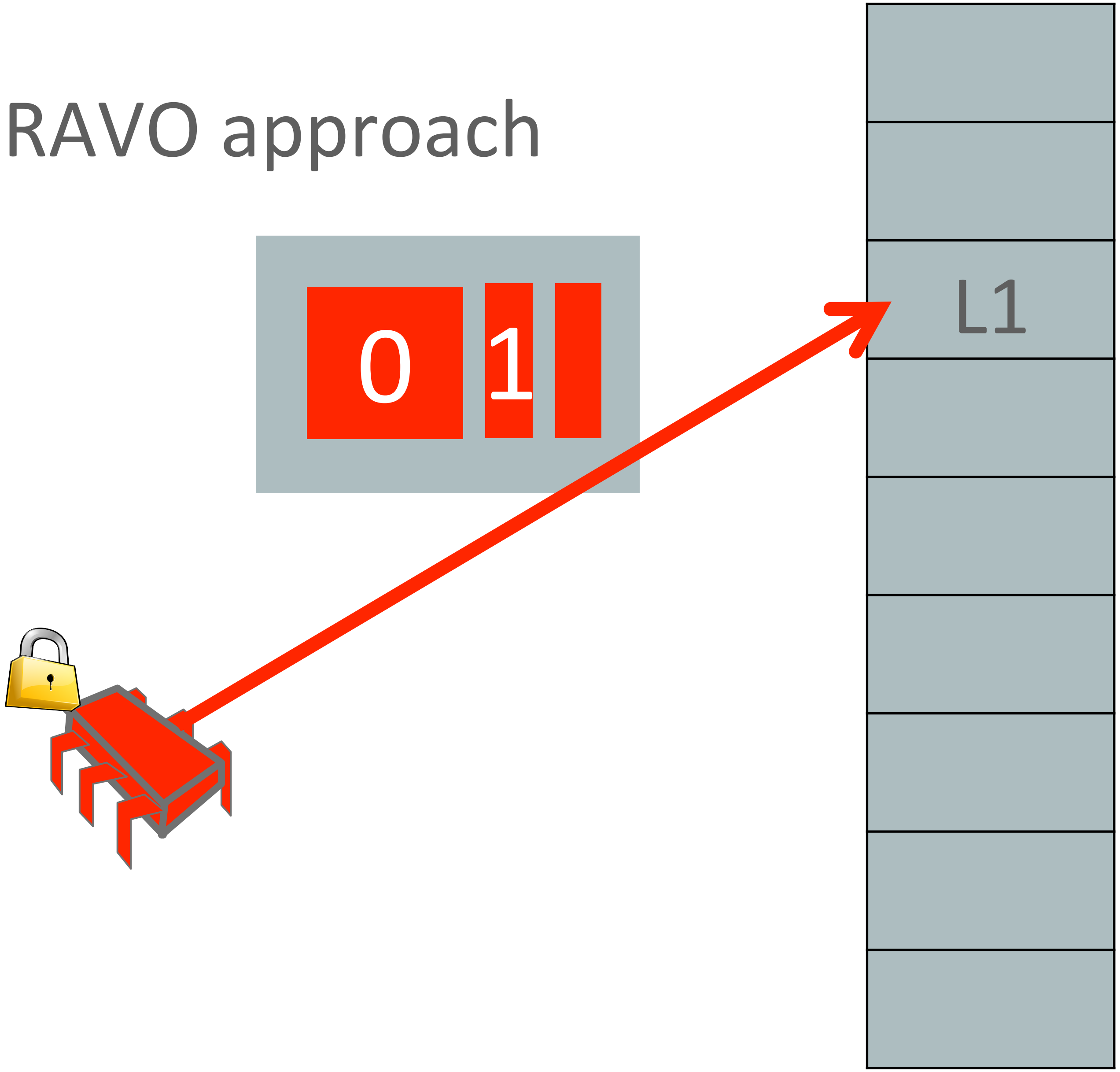
The BRAVO approach



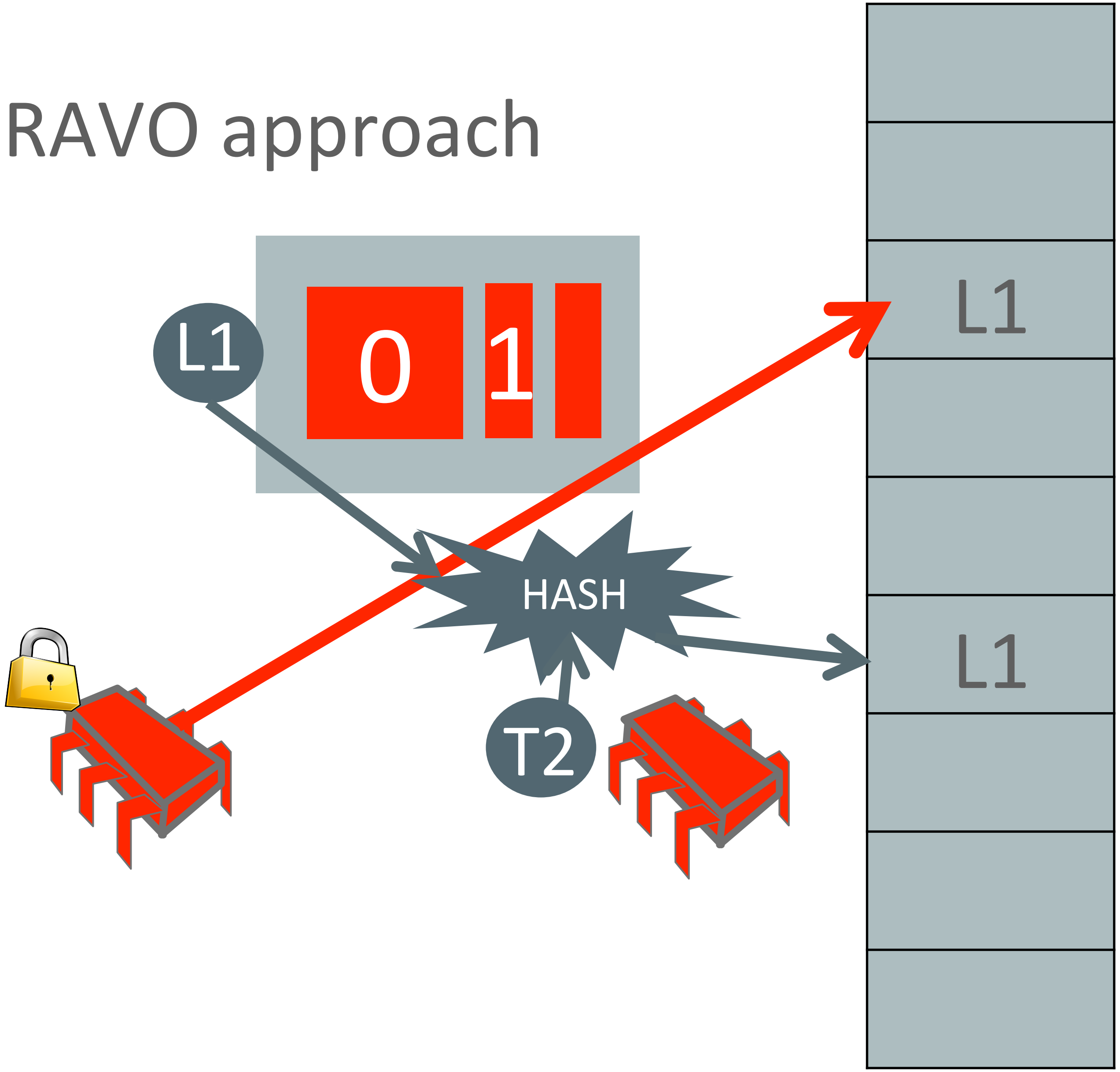
The BRAVO approach



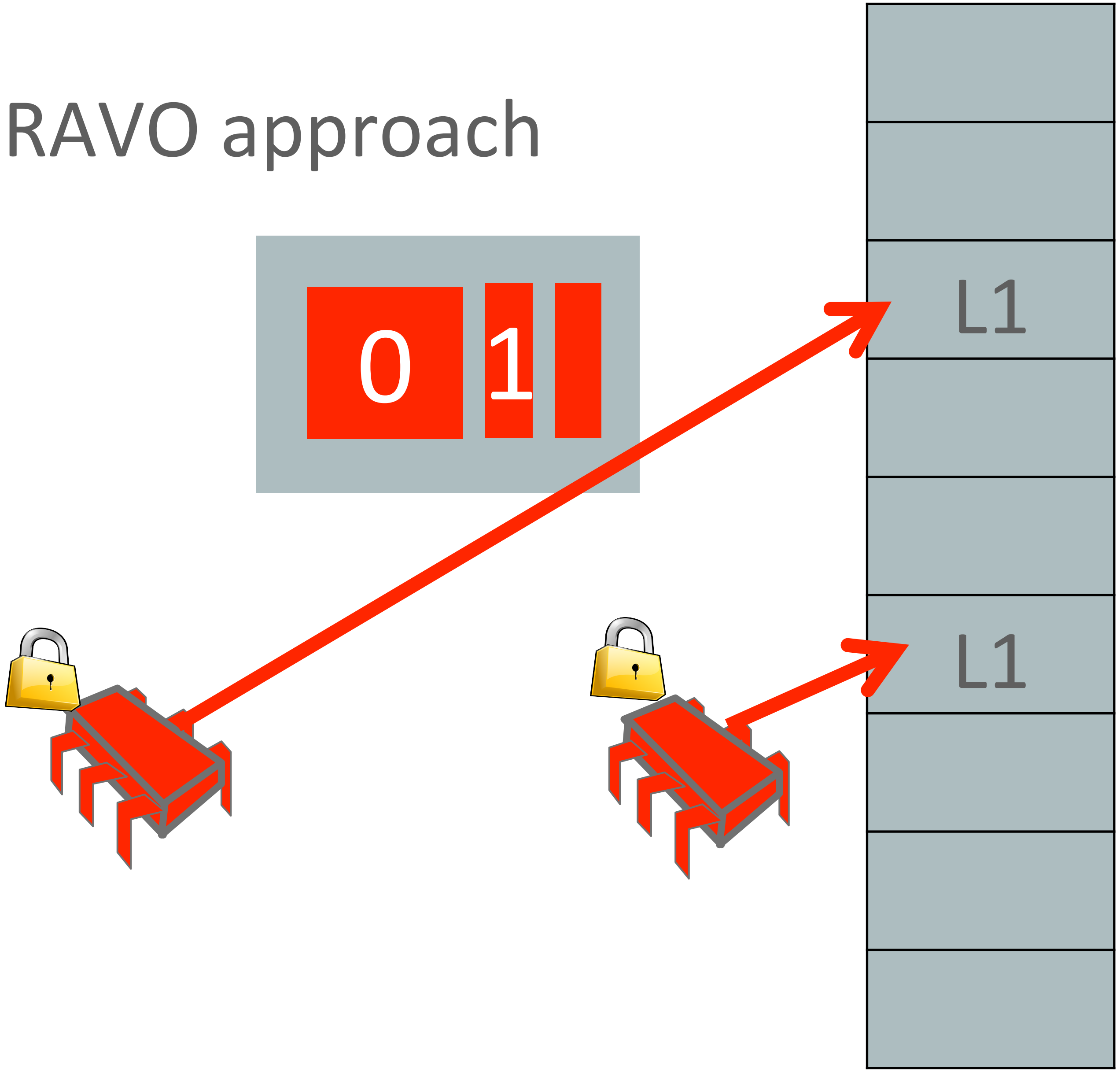
The BRAVO approach



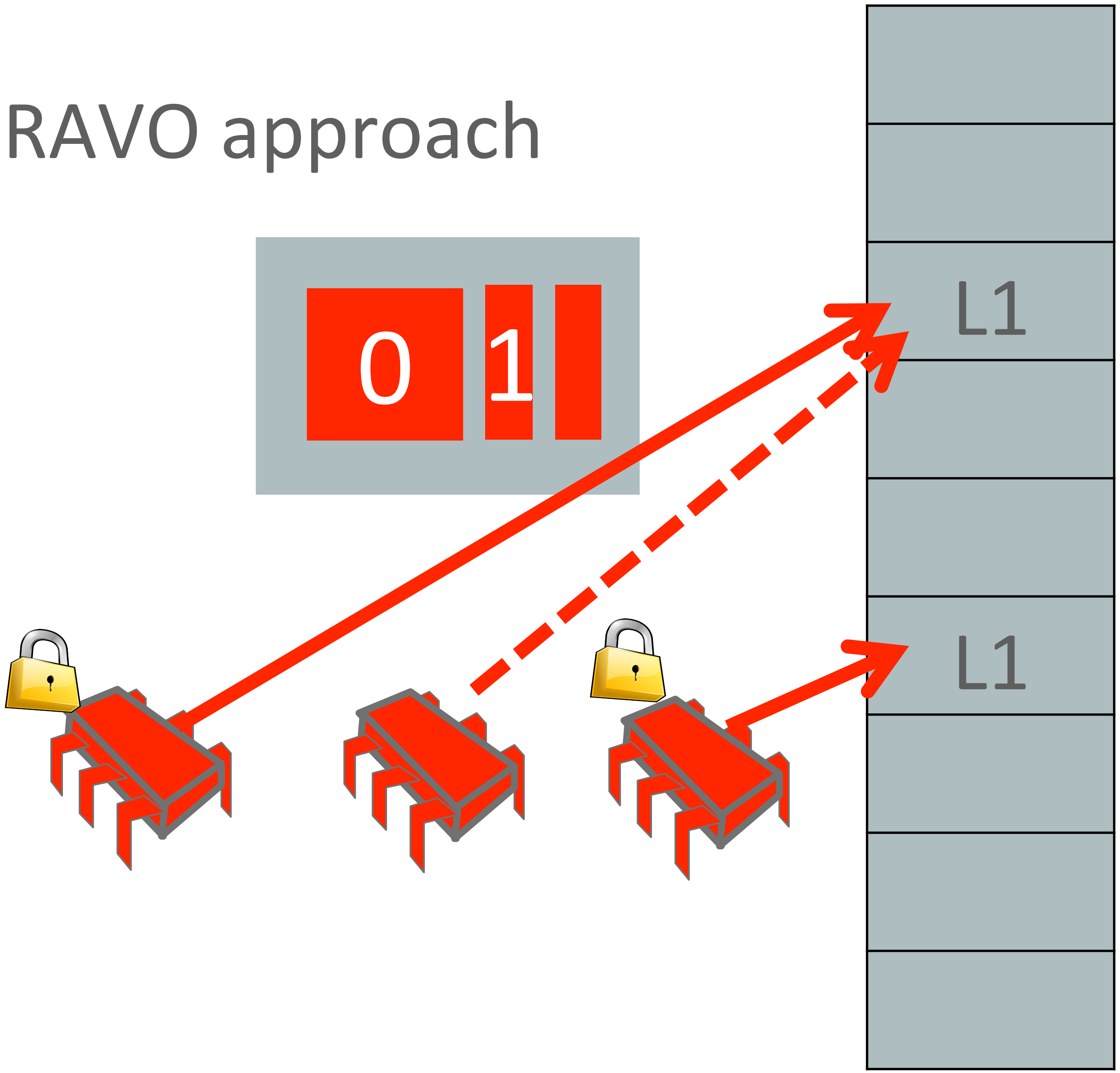
The BRAVO approach



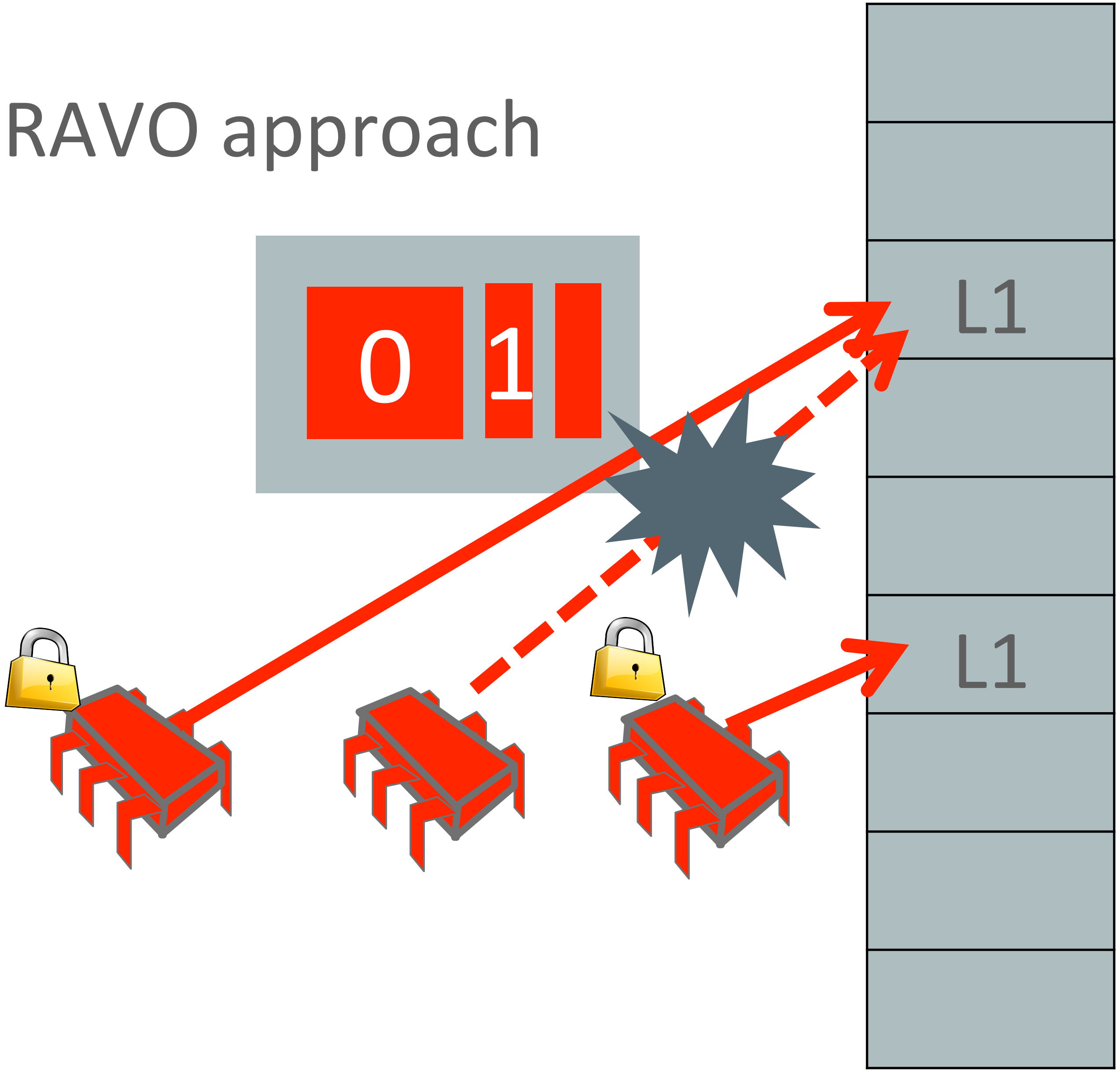
The BRAVO approach



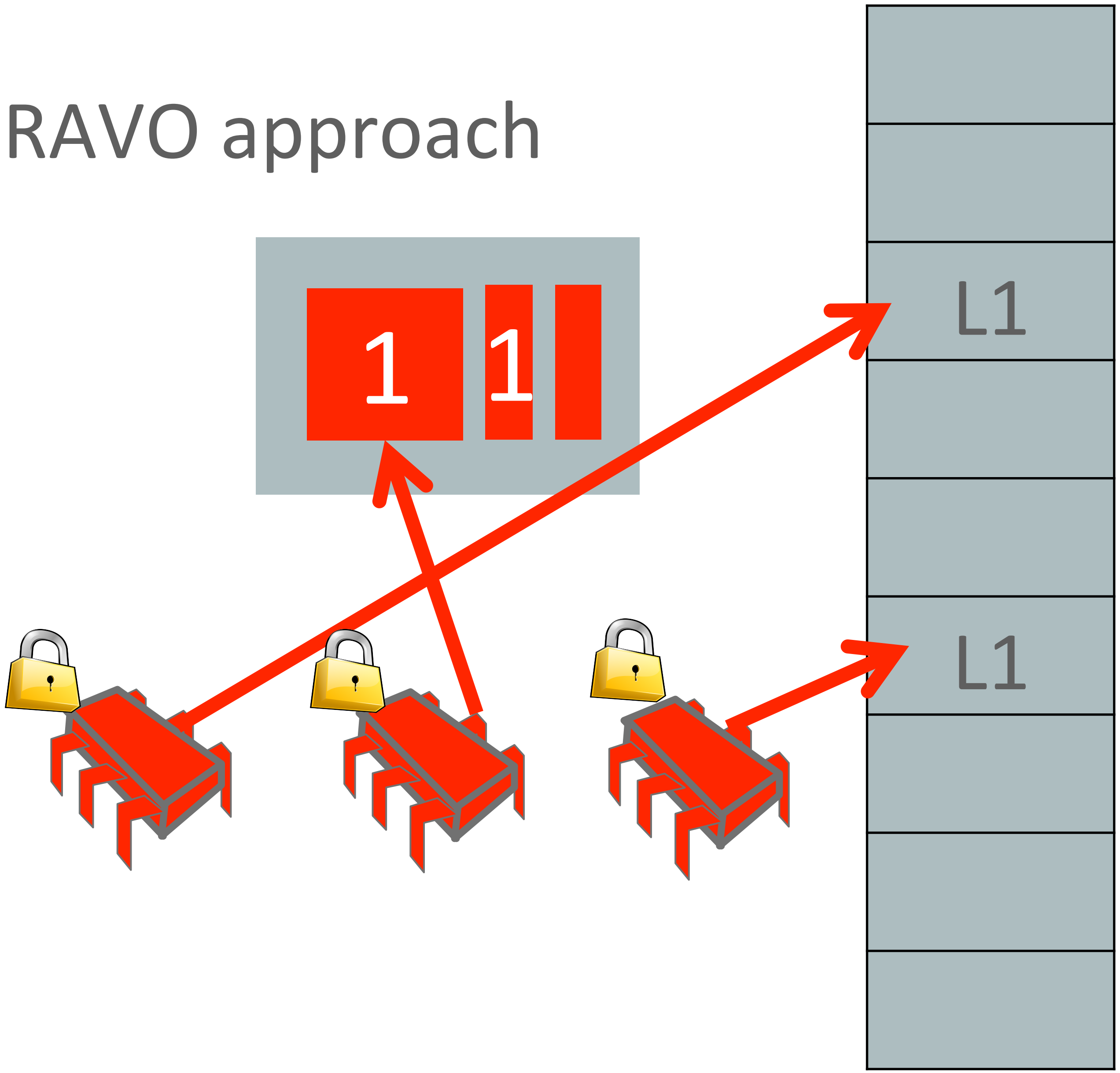
The BRAVO approach



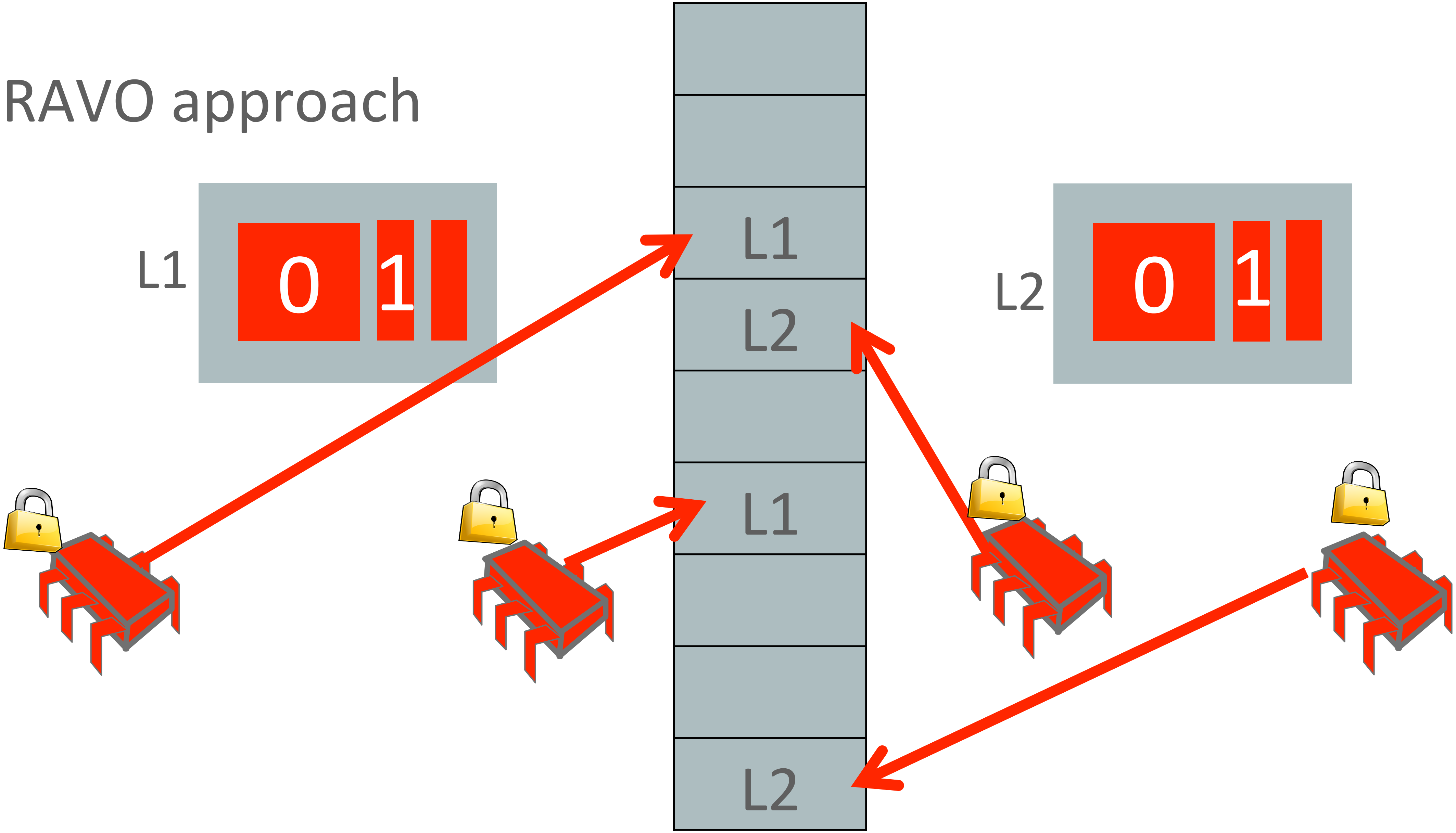
The BRAVO approach



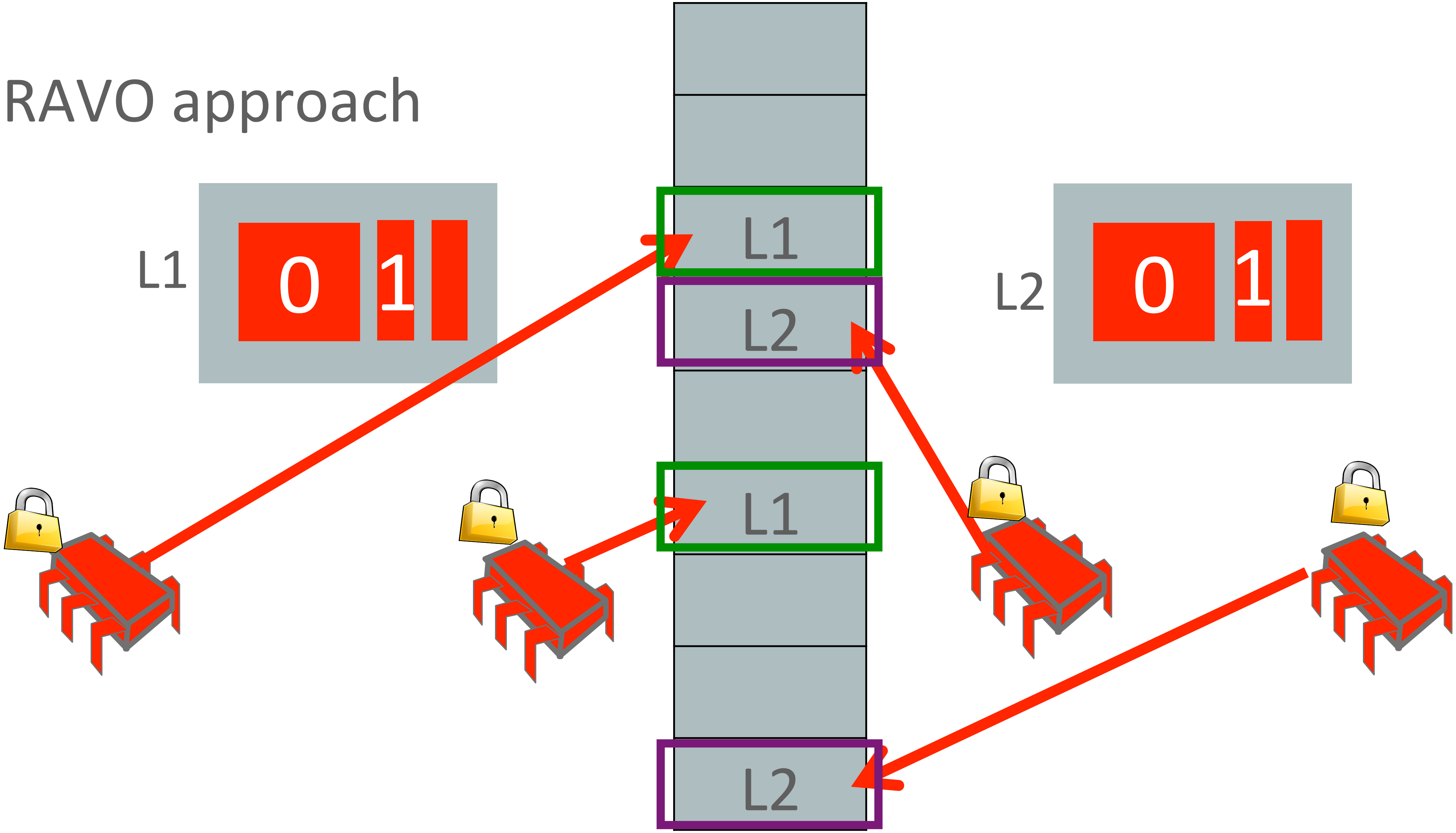
The BRAVO approach



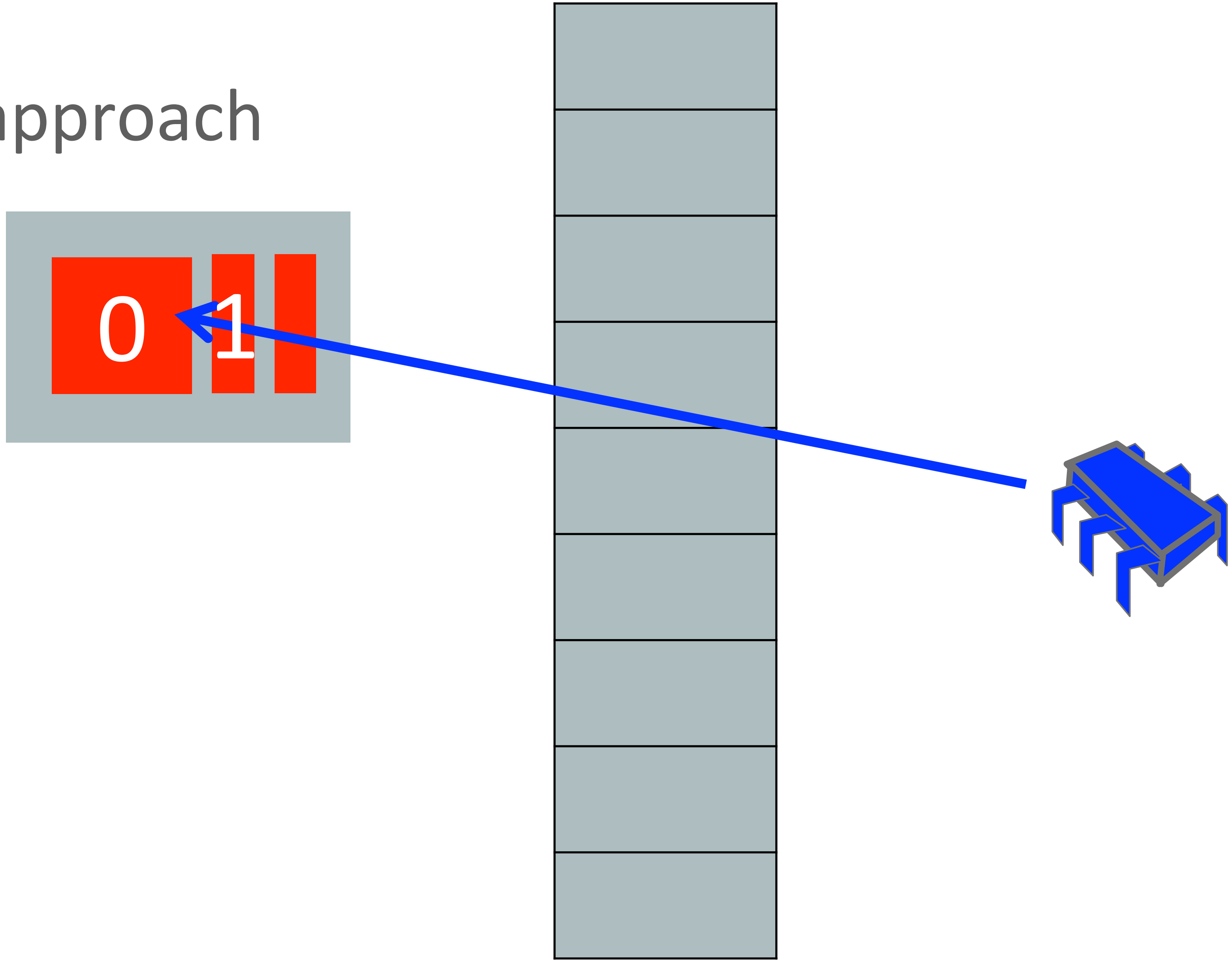
The BRAVO approach



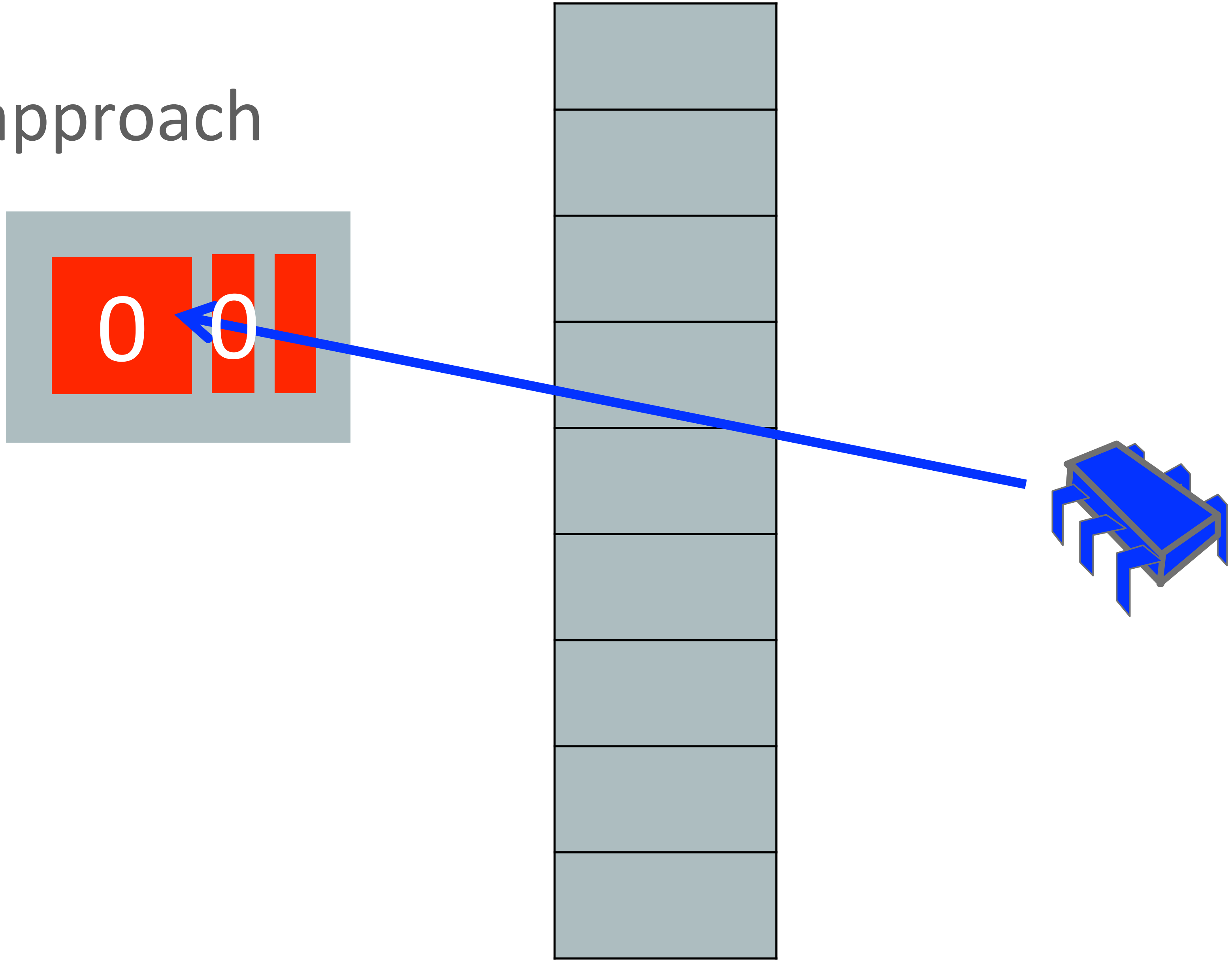
The BRAVO approach



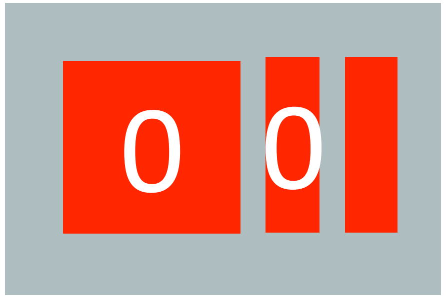
The BRAVO approach

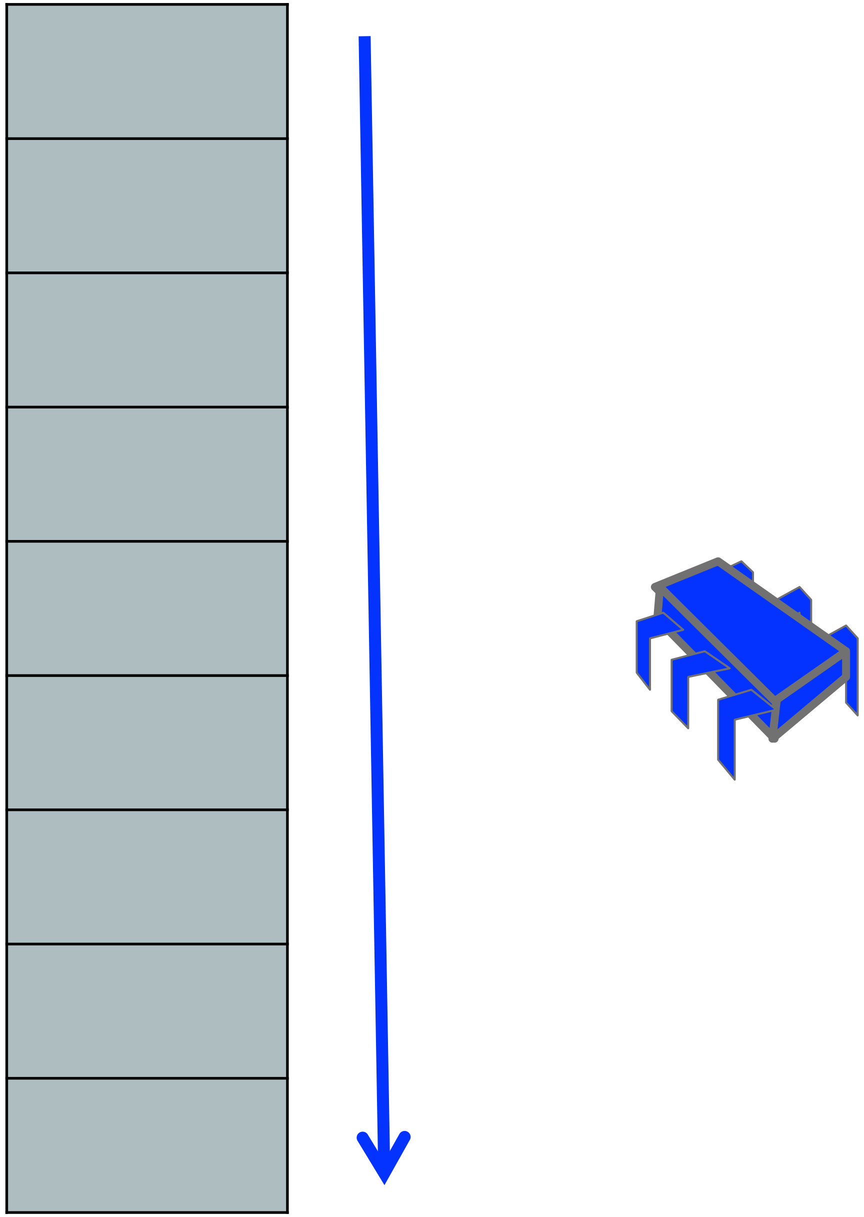


The BRAVO approach

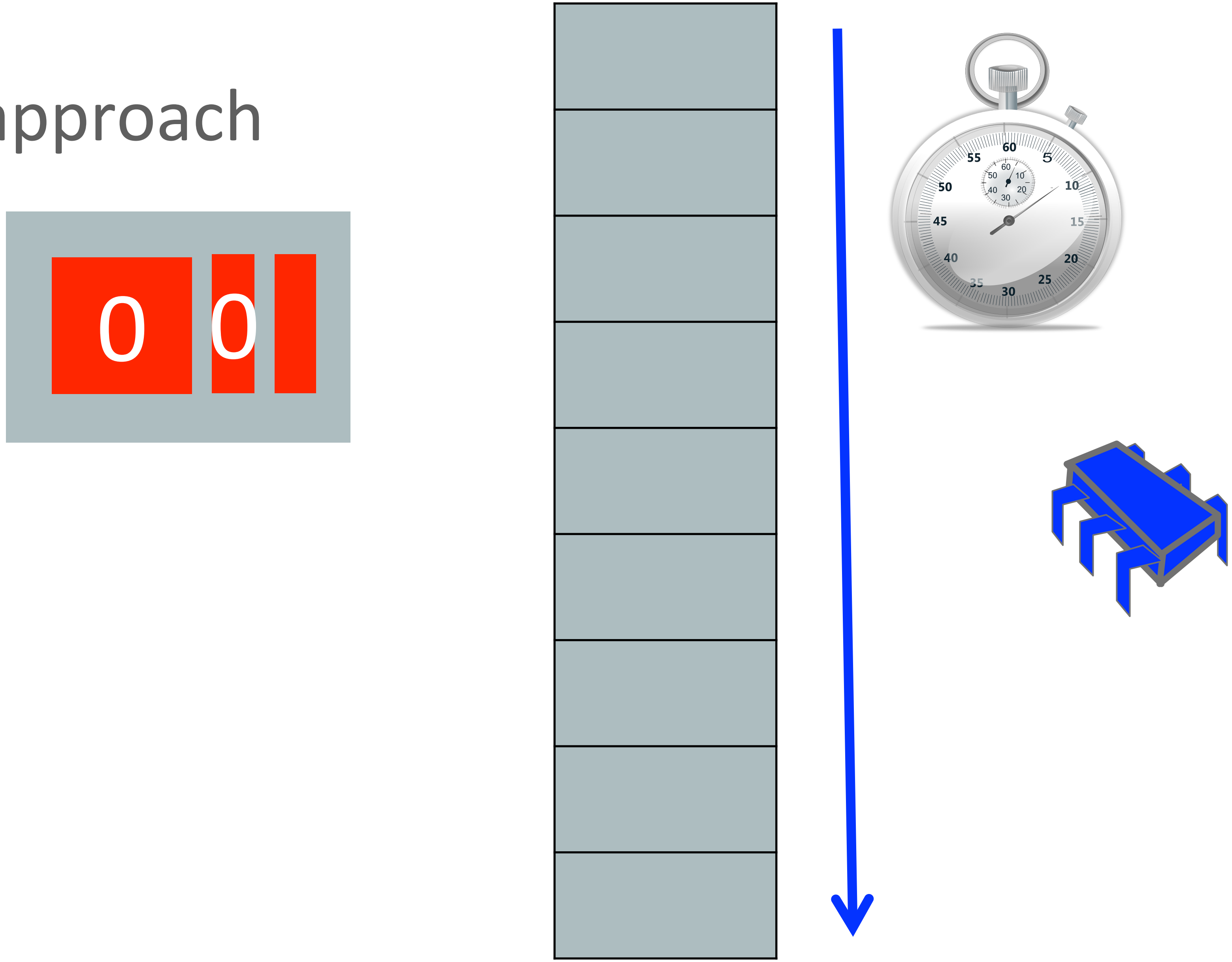


The BRAVO approach

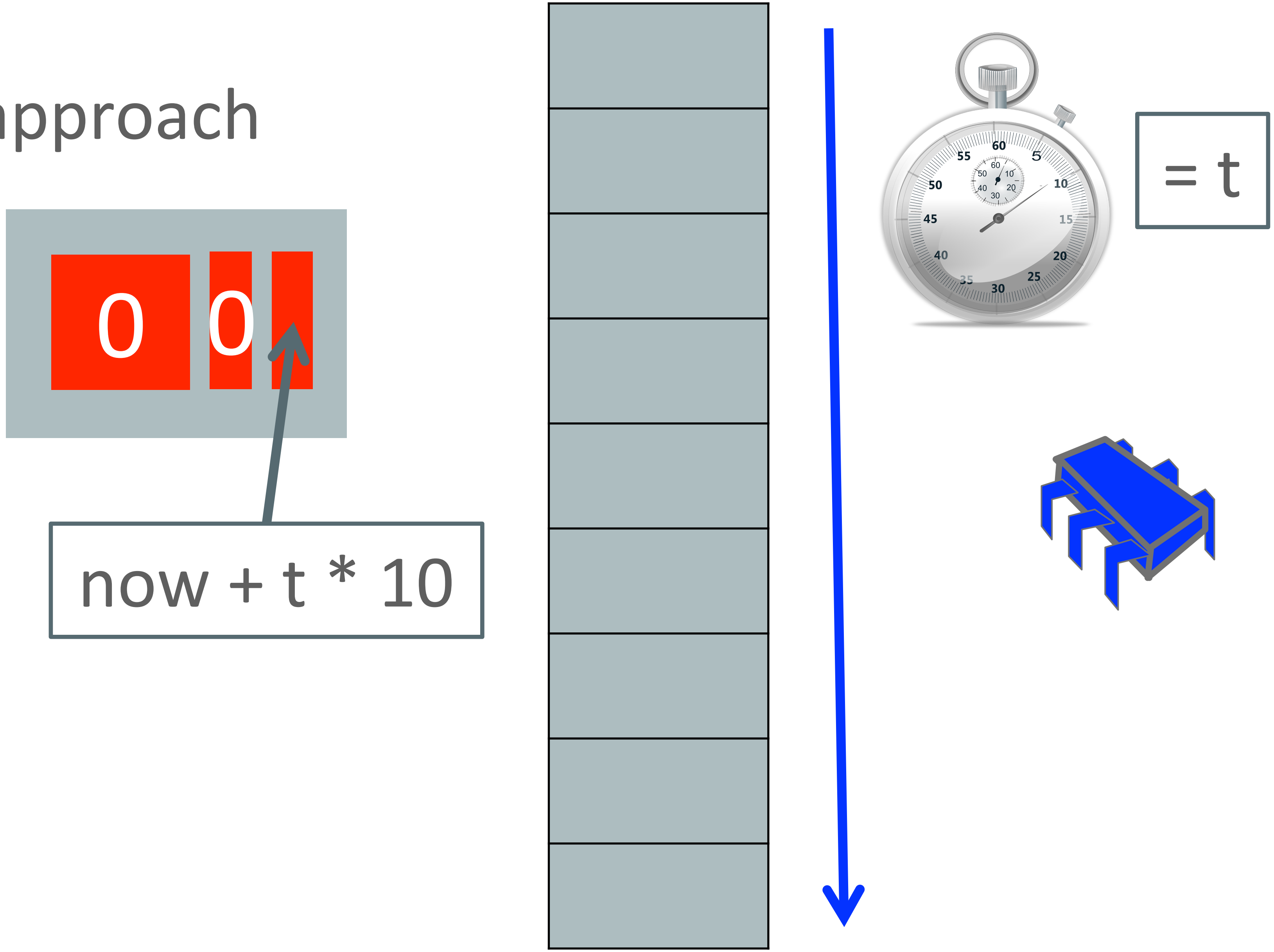




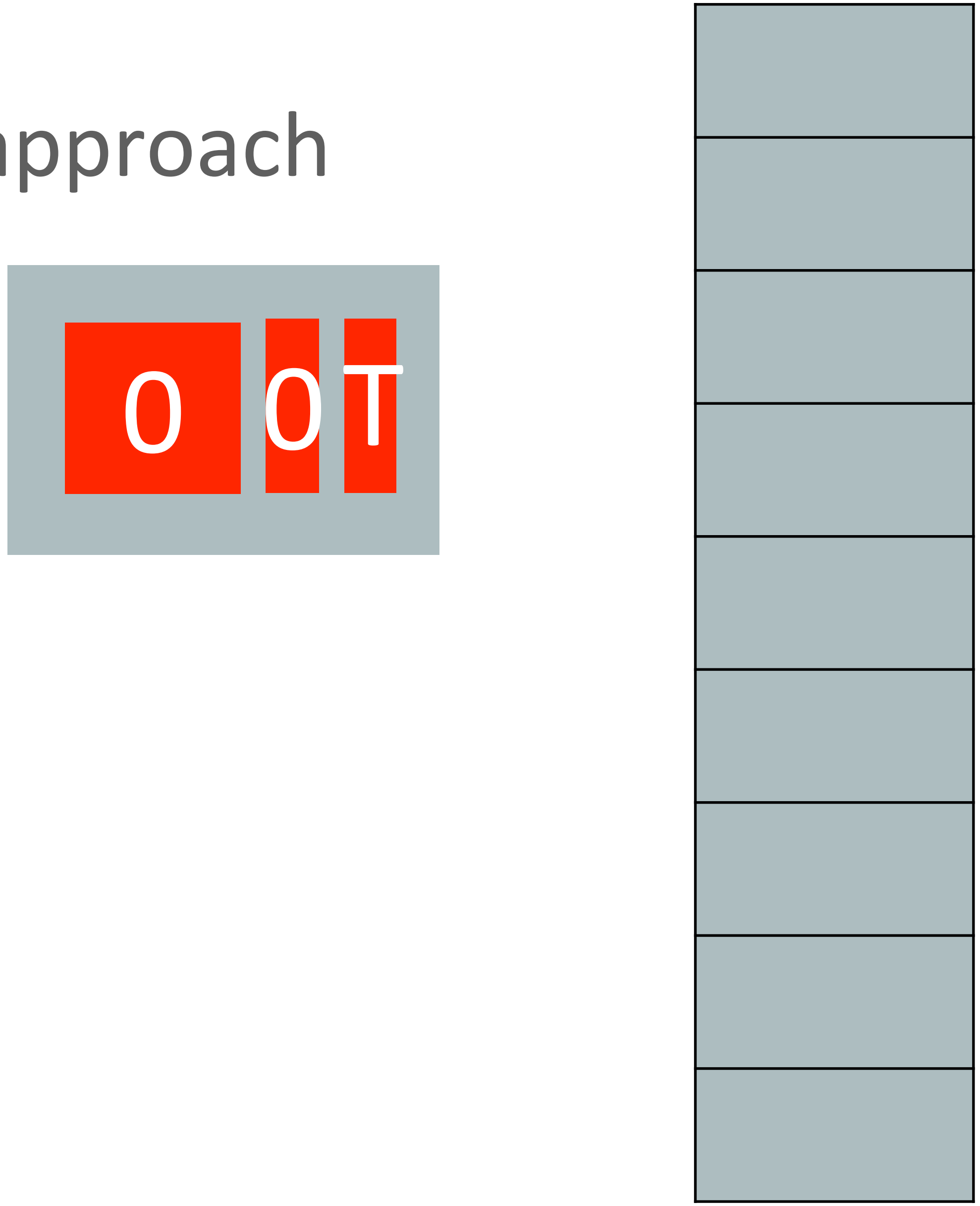
The BRAVO approach



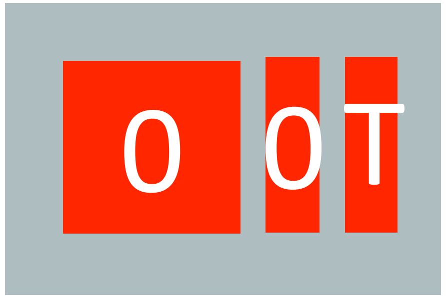
The BRAVO approach

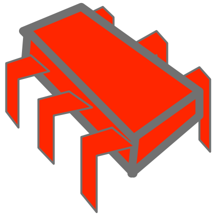


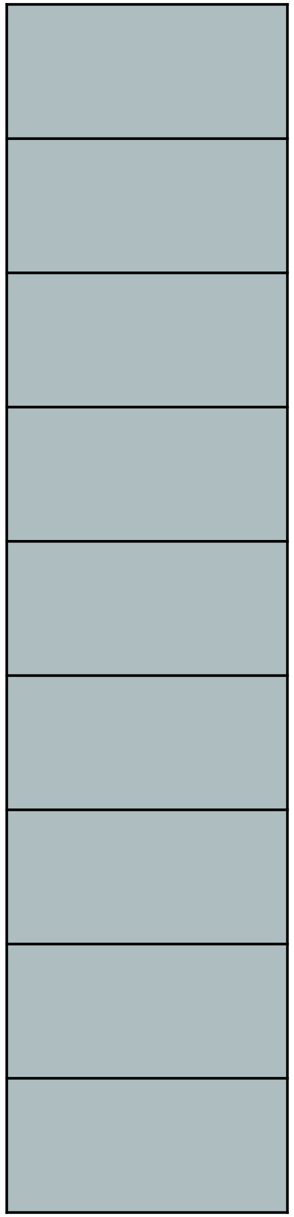
The BRAVO approach



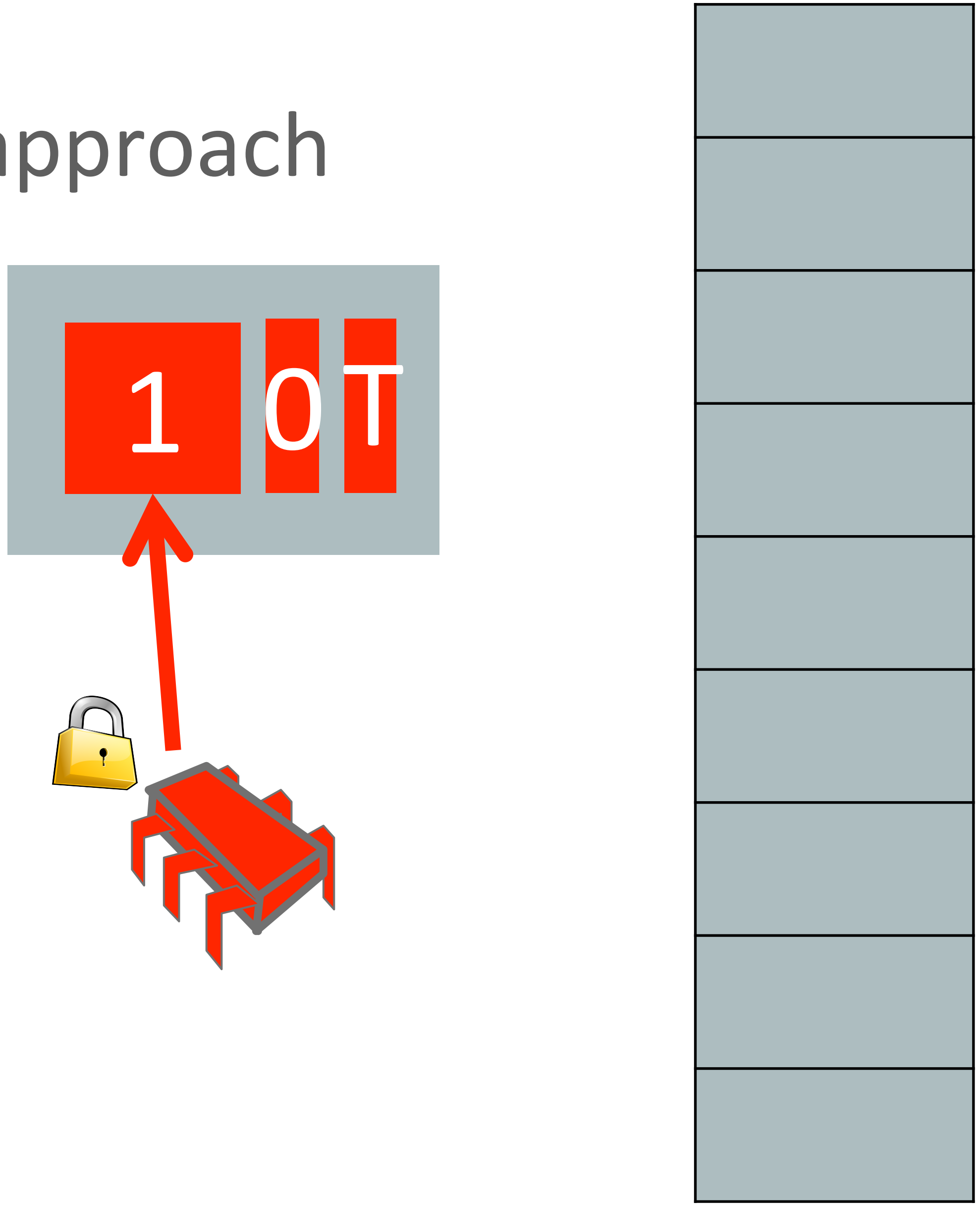
The BRAVO approach



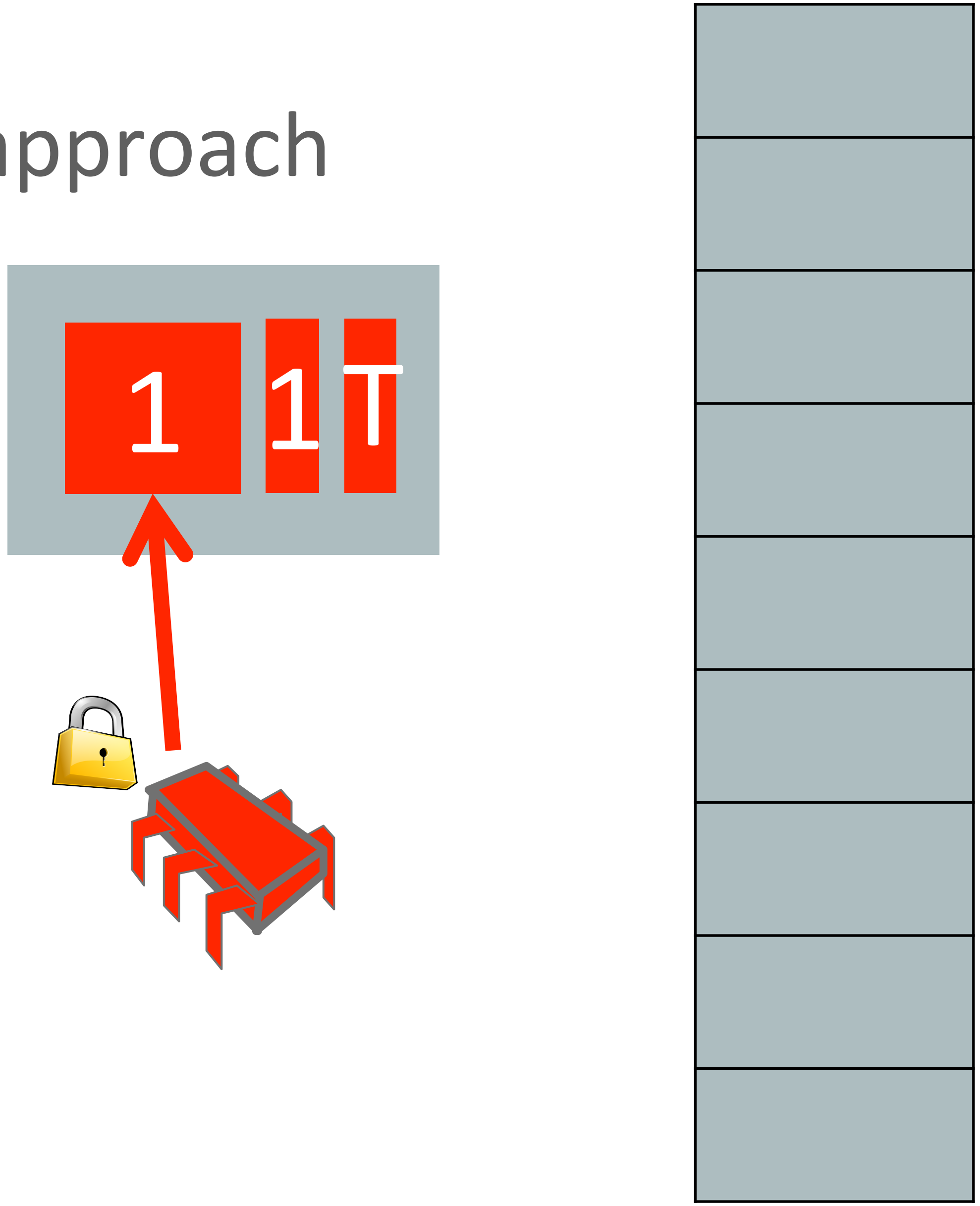




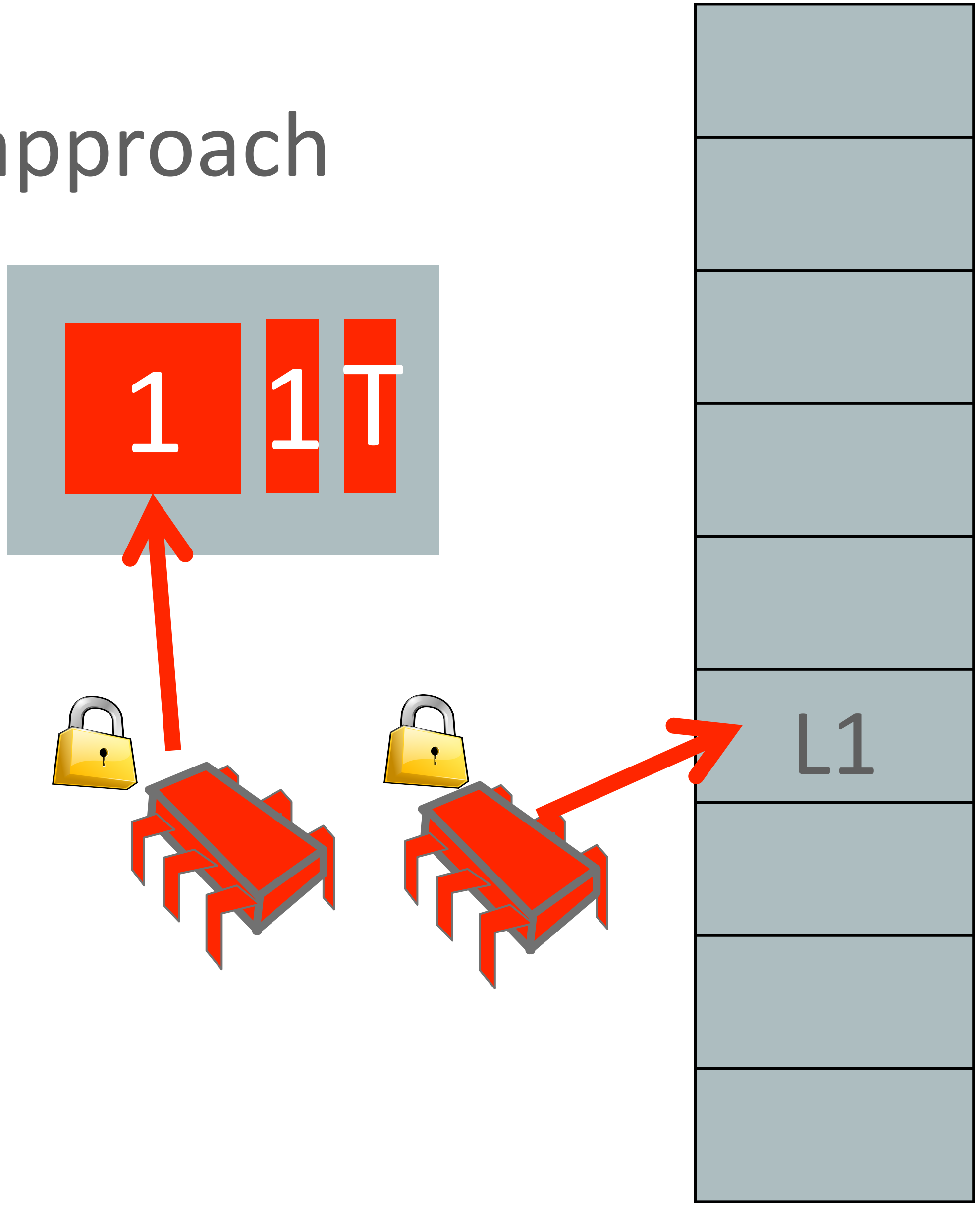
The BRAVO approach



The BRAVO approach



The BRAVO approach



Evaluation

- Easy to integrate with existing locks
- Compact
- Accelerates reads
- Handles writes gracefully

Evaluation: Easy to integrate

- Brandenburg-Anderson (BA) reader-writer lock
- POSIX Pthread reader-writer lock

- Linux kernel rwsem

Evaluation: Compact

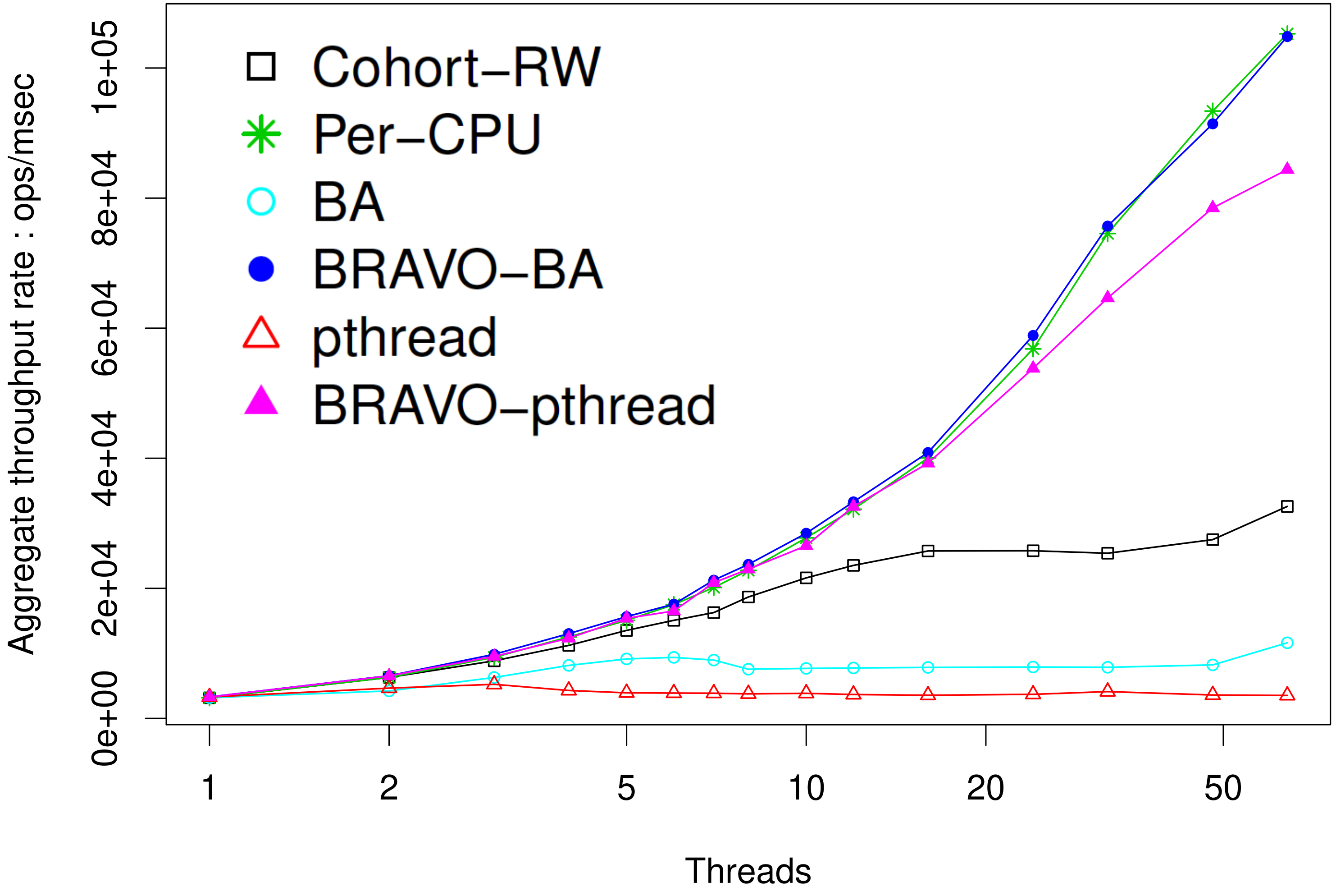
Locks	Memory footprint
BA	40
BA + BRAVO	40 + 12 + 32KB (for a table)
Per-CPU	9216 (on a system with 72 CPUs)
Cohort-RW	896 (dual-socket)

Intel Xeon E5-2699 v3 CPU
2 sockets
72 logical CPUs in total

ORACLE

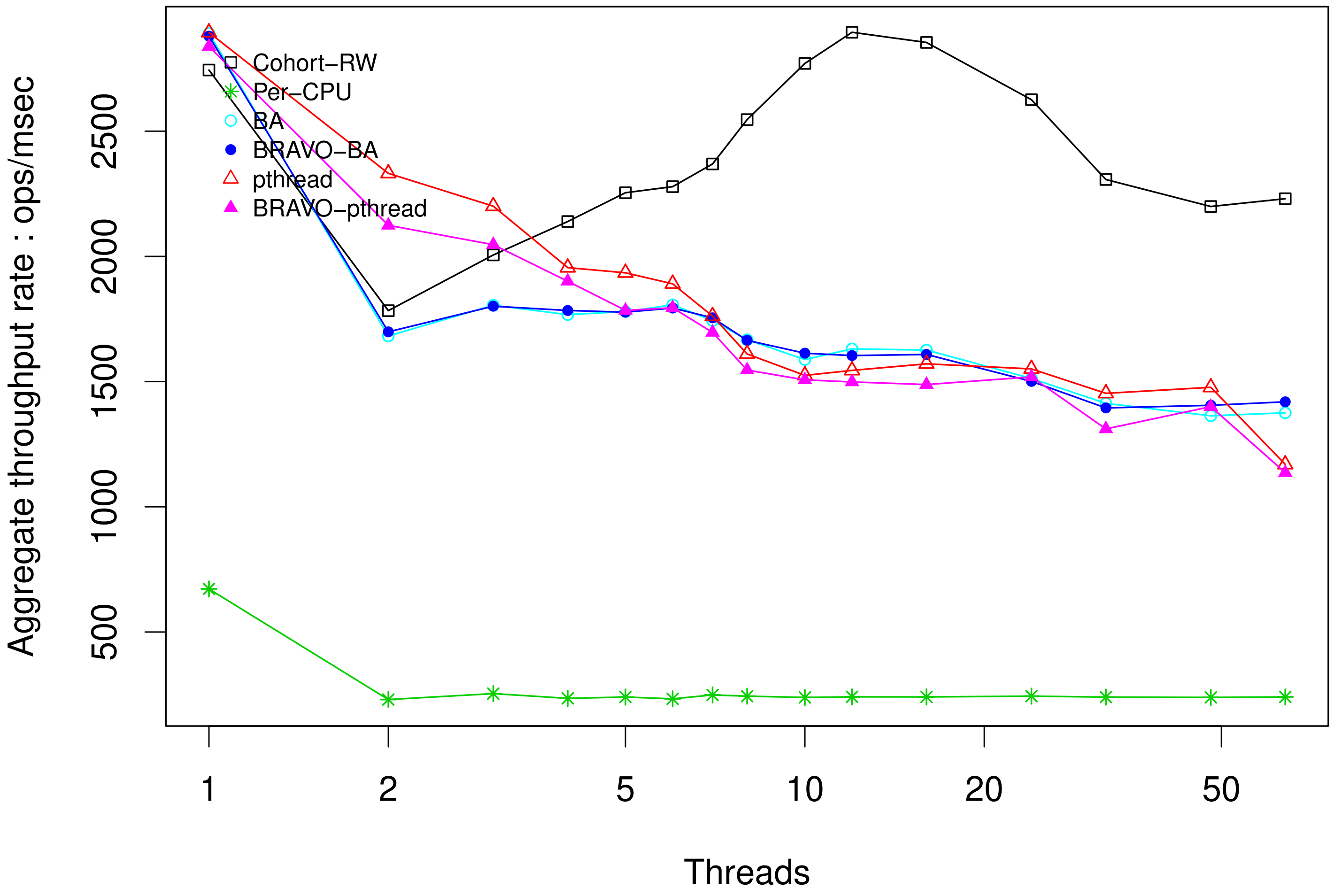
Evaluation: Accelerates reads

RWBench with 1 out of every 10000 are writes



Evaluation: Handles writes gracefully

RWBench with 9 out of every 10 are writes



Linux Kernel rwsem

- Counter + waiting queue protected by a spin lock
- Reader atomically increments the counter and checks its value

Linux Kernel rwsem

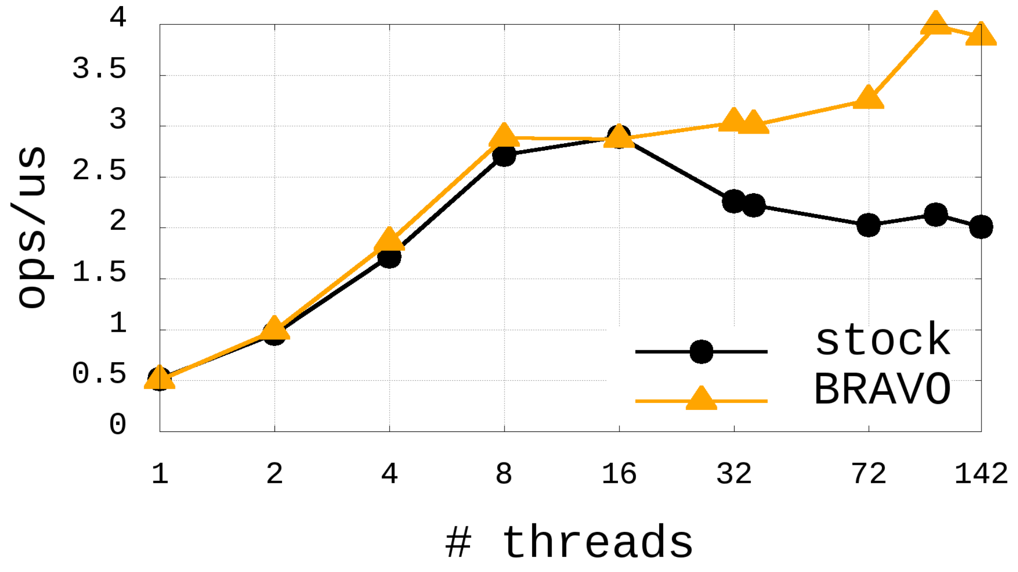
- Counter + waiting queue protected by a spin lock
- Reader atomically increments the counter and checks its value
- Synchronization bottleneck in the kernel (mmap_sem)

Linux Kernel rwsem

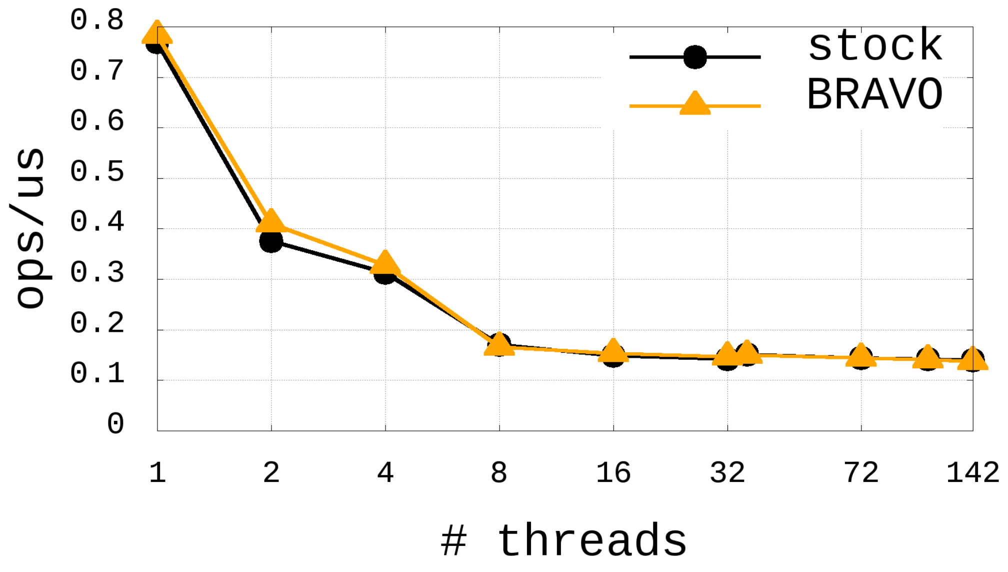
- Counter + waiting queue protected by a spin lock
- Reader atomically increments the counter and checks its value
- Synchronization bottleneck in the kernel (mmap_sem)
- Stress-test with will-it-scale: page_fault and mmap

Evaluation with will-it-scale

Intel Xeon E7-8895 v3 CPU
4 sockets
144 logical CPUs in total



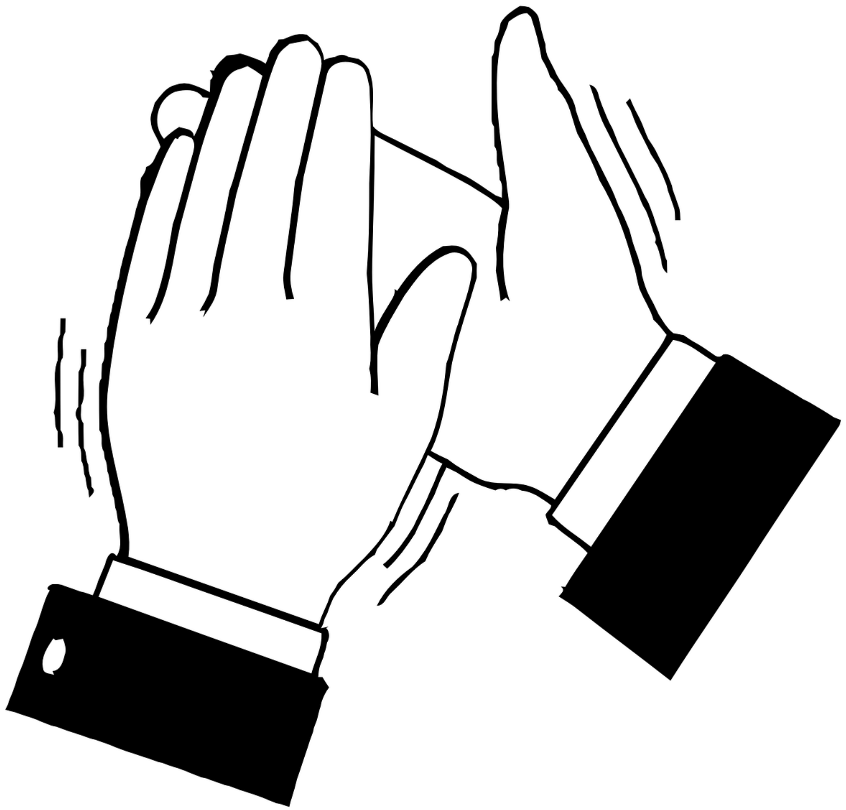
page_fault



mmap

BRAVO: wrap-up

- Builds into any existing lock
- Reads are accelerated
- Avoids write overhead
- Very compact
- Overall, takes the “reader indicator” dilemma away



Conclusion

- Kernel requirements impede adaptation of user-space locks
  - some considerations we did not talk about: real-time, paravirt
- Same techniques can still be used
  - trading (some) fairness for performance
  - eliminating contention bottlenecks
  - reducing #cache misses in lock handover
- Kernel locks remain hot

Thank you!

QUESTIONS?

ORACLE