Multicore Programming

Locks

8 Nov 2010 (Part 2)

Peter Sewell Jaroslav Ševčík Tim Harris

Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

Hierarchical locks

Parallel performance

Test and set (pseudo-code)

```
bool testAndSet(bool *b) {
  bool result;
  atomic {
    result = *b;
    *b = TRUE;
  }
}
```

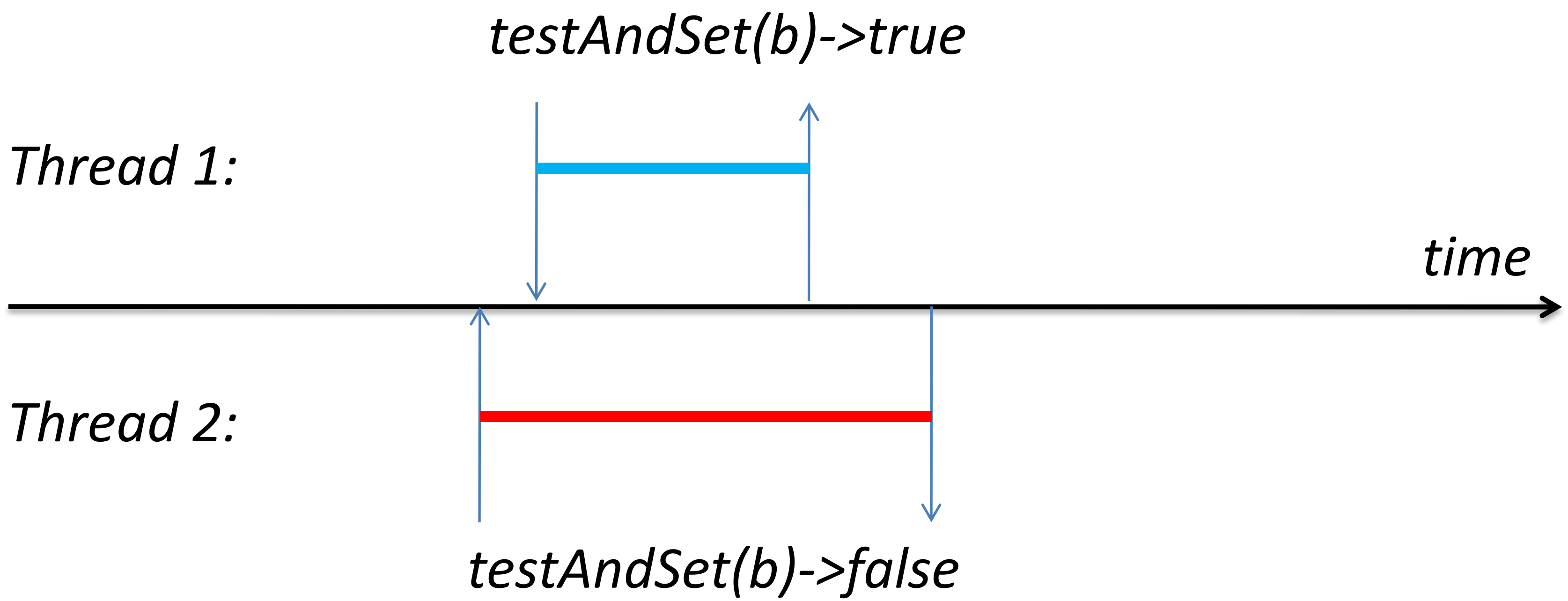
Pointer to a location holding a boolean value (TRUE/FALSE)

Read the current contents of the location b points to…

…set the contents of *b to TRUE

Test and set

- Suppose two threads use it at once



Test and set lock

lock: FALSE

FALSE => lock available
TRUE => lock held

```
void acquireLock(bool *lock) {
  while (testAndSet(lock)) {
    /* Nothing */
  }
}
```

Each call tries to acquire the lock, returning TRUE if it is already held

```
void releaseLock(bool *lock) {
  *lock = FALSE;
}
```

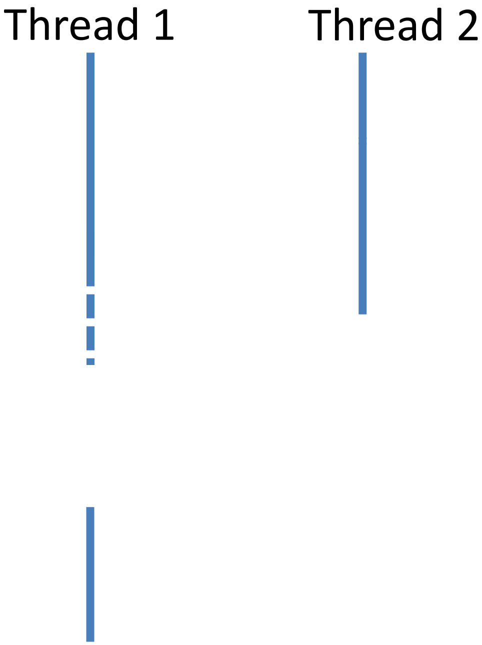
NB: all this is pseudo-code, assuming SC memory

Test and set lock

lock: TRUE

```
void acquireLock(bool *lock) {
   while (testAndSet(lock)) {
      /* Nothing */
   }
}
```

```
void releaseLock(bool *lock) {
   *lock = FALSE;
}
```



What are the problems here?

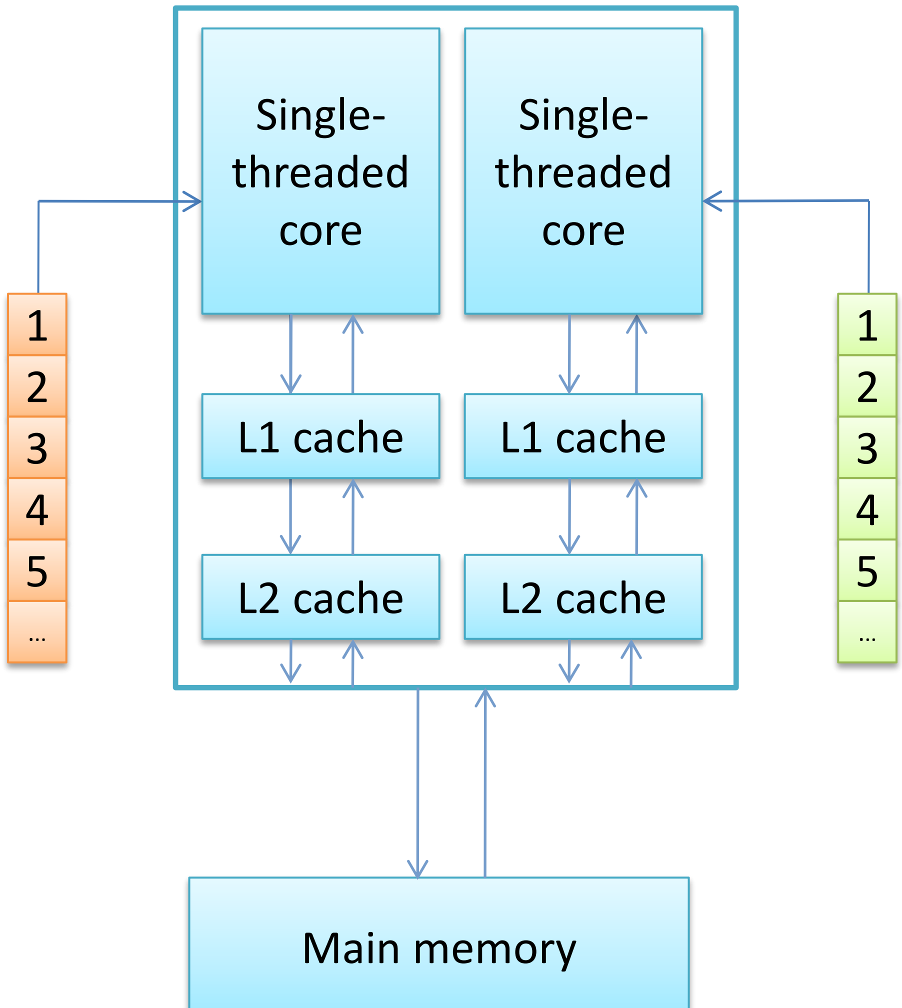
testAndSet implementation causes contention

No control over locking policy

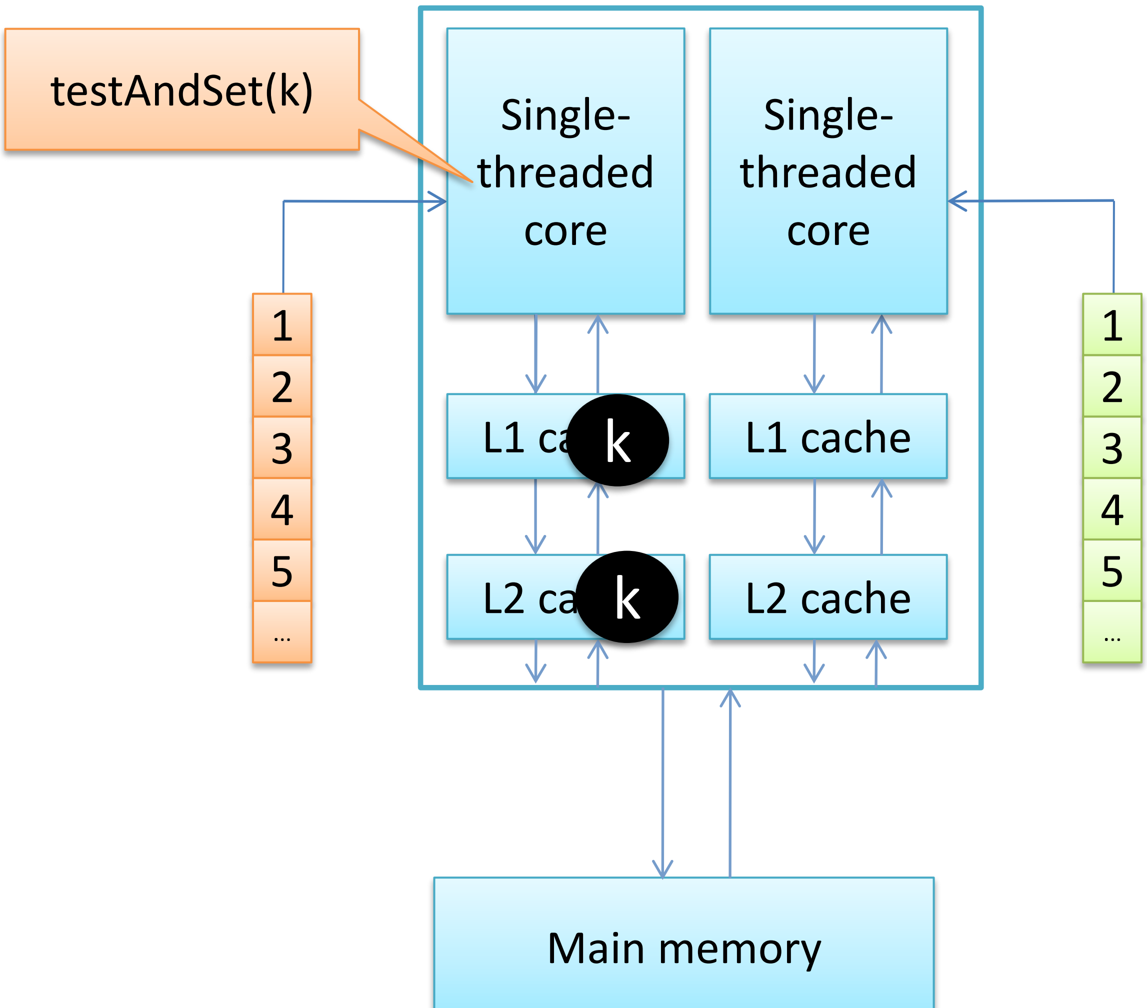
Only supports mutual exclusion: not reader-writer locking

Spinning may waste resources while waiting

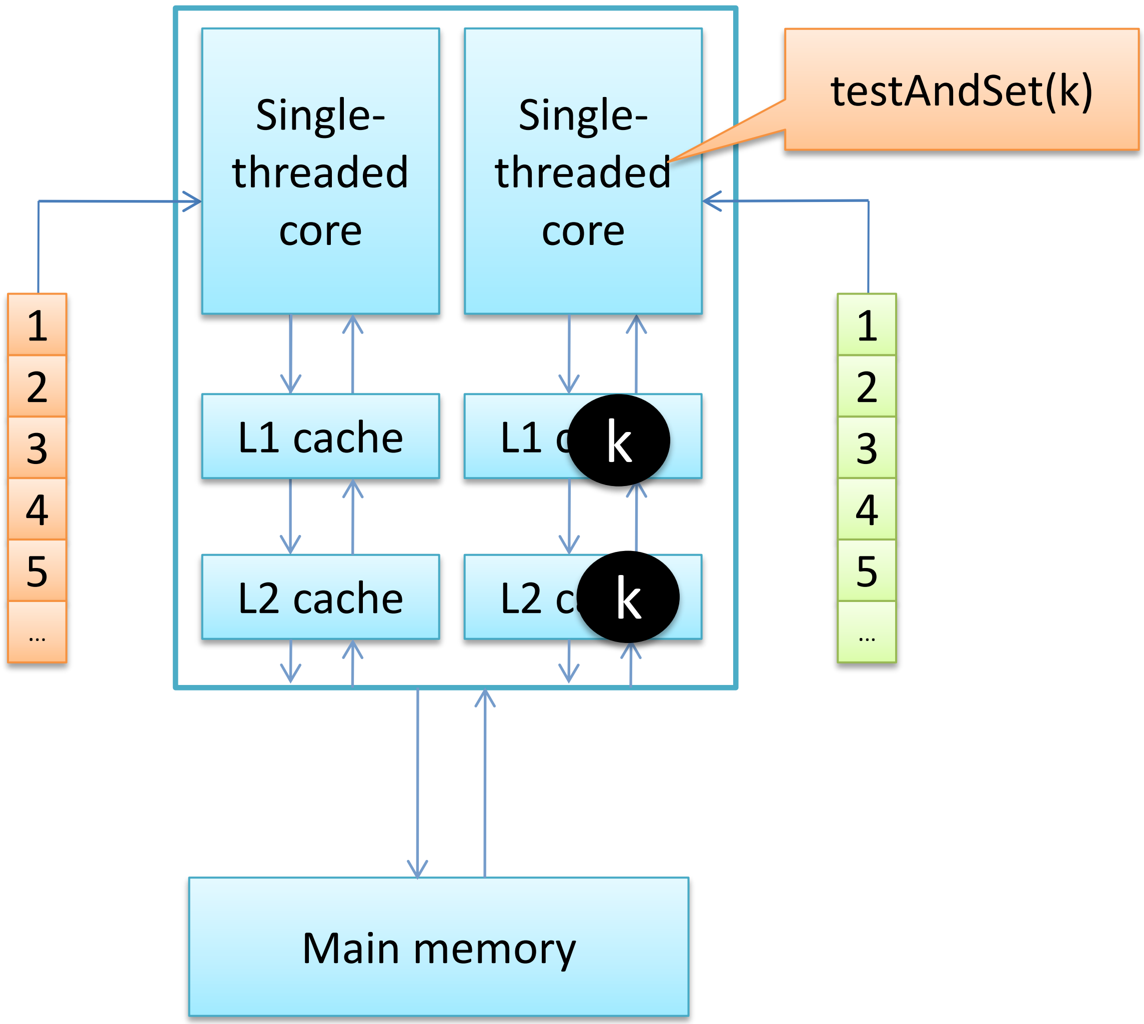
Contention from testAndSet



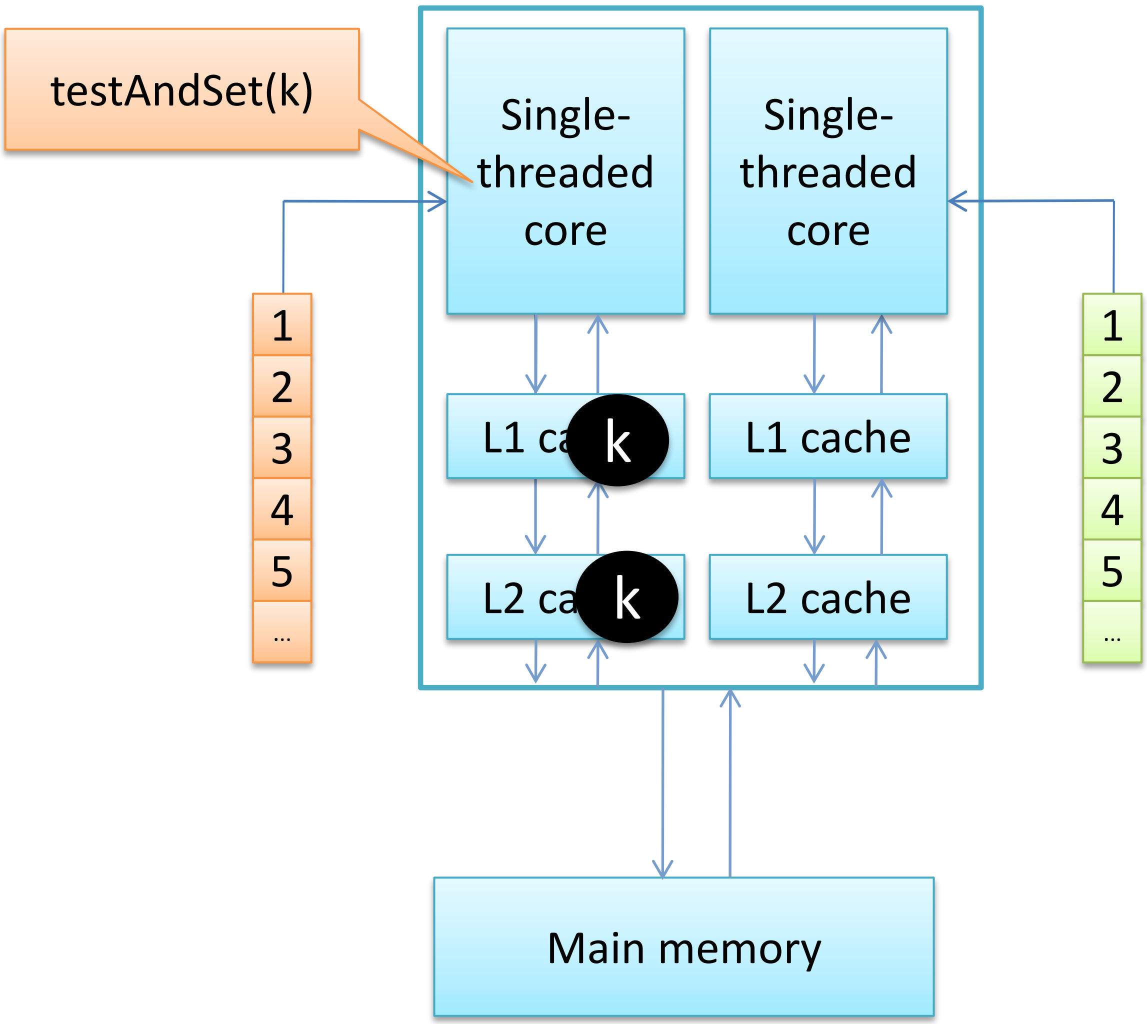
Multi-core h/w – separate L2



Multi-core h/w – separate L2



Multi-core h/w – separate L2



General problem

- No *logical conflict* between two failed lock acquires
- Cache protocol introduces a *physical conflict*
- For a good algorithm: only introduce physical conflicts if a logical conflict occurs

Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

Hierarchical locks

Parallel performance

Test and test and set lock

lock: FALSE

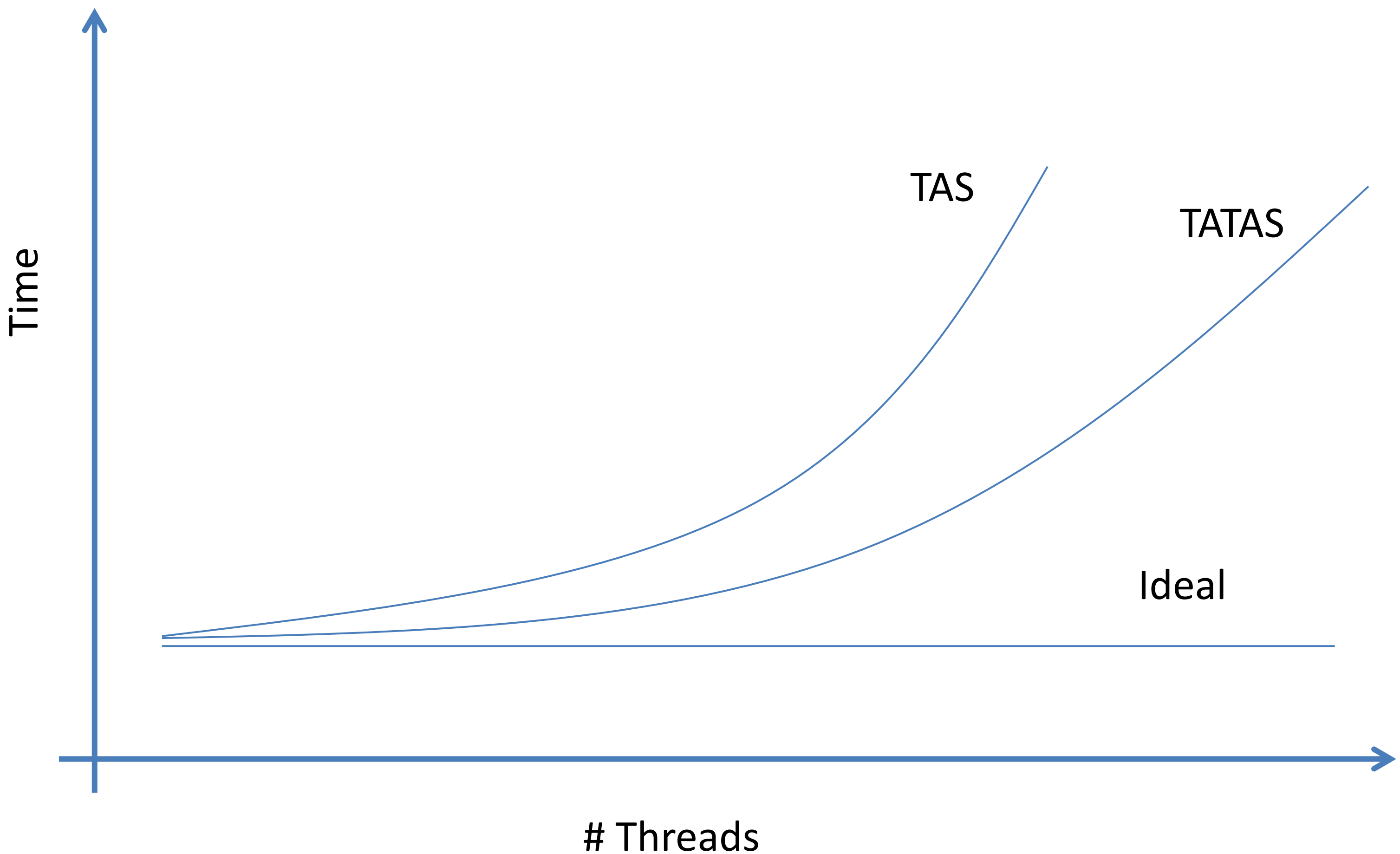
FALSE => lock available
TRUE => lock held

```
void acquireLock(bool *lock) {
  do {
    while (*lock) { }
  } while (testAndSet(lock));
}
```

Spin while the lock is held… only do testAndSet when it is clear

```
void releaseLock(bool *lock) {
  *lock = FALSE;
}
```

Performance

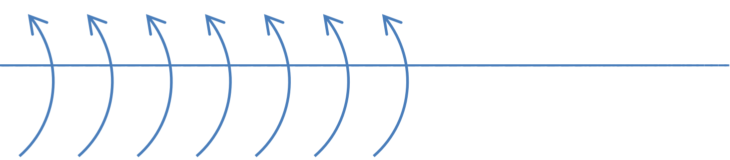


Based on Fig 7.4, Herlihy & Shavit, “The Art of Multiprocessor Programming”

Stampedes

lock: TRUE

```
void acquireLock(bool *lock) {
  do {
    while (*lock) { }
  } while (testAndSet(lock));
}
```



```
void releaseLock(bool *lock) {
   *lock = FALSE;
}
```

Back-off algorithms

1. Start by spinning, watching the lock
2. After an interval *c* spin locally for *s* *(without watching the lock)*

What should "c" be?
What should "s" be?

Time spent waiting "c"

- Lower values:
  - Less time to build up a set of threads that will stampede
- Higher values:
  - Less likelihood of a delay between a lock being released and a waiting thread noticing

Spinning time “s”

- Lower values:
  - More responsive to the lock becoming available
- Higher values:
  - If the lock doesn’t become available then the thread makes fewer accesses to the shared variable

Methodical approach

- For a given workload and performance model:
  - What is the best that an oracle could do (e.g. given perfect knowledge of lock demands)?
  - How does a practical algorithm compare with this?
- Look for an algorithm with a bound between its performance and that of the oracle
- “Competitive spinning”

Rule of thumb

- Spin for a duration that's comparable with the shortest back-off interval
- Exponentially increase the per-thread back-off interval (resetting it when the lock is acquired)
- Use a maximum back-off interval that is large enough that waiting threads don't interfere with the system's performance

Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

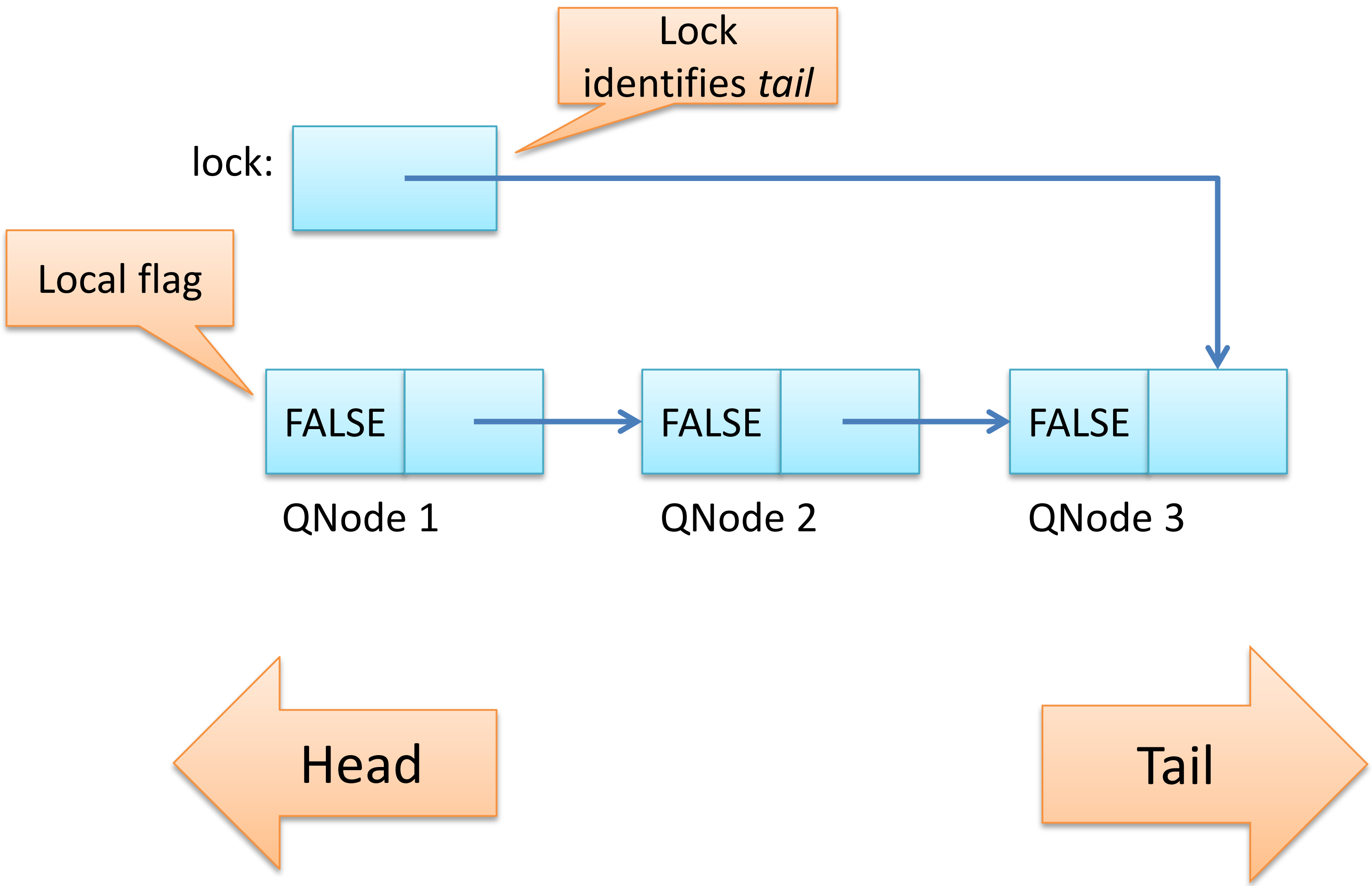
Hierarchical locks

Parallel performance

Queue-based locks

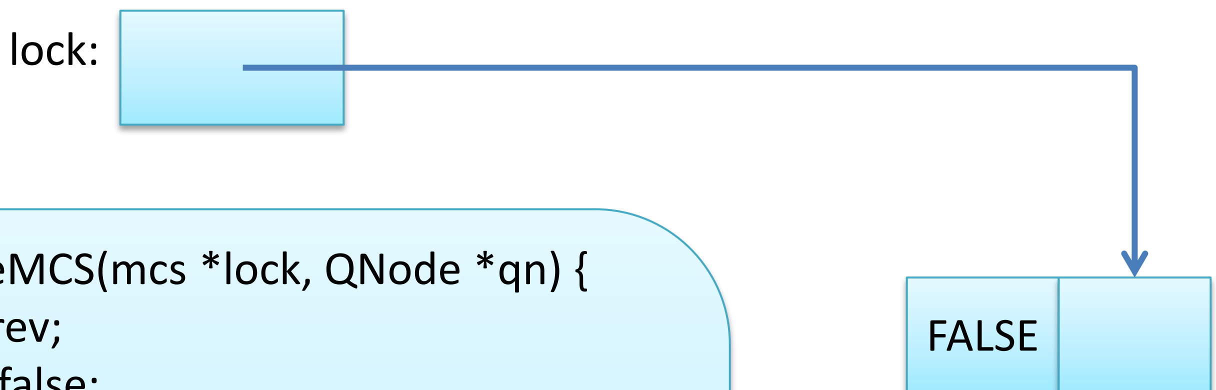
- Lock holders queue up: immediately provides FCFS behavior
- Each spins *locally* on a flag in their queue entry: no remote memory accesses while waiting
- A lock release wakes the next thread directly: no stampede

MCS locks



Lock identifies *tail*
lock:
Local flag
FALSE
FALSE
FALSE
QNode 1
QNode 2
QNode 3
Head
Tail

MCS lock acquire




```
void acquireMCS(mcs *lock, QNode *qn) {
  QNode *prev;
  qn->flag = false;
  qn->next = NULL;
  while (true) {
    prev = lock->tail;
    /*Label 1*/
    if (CAS(&lock->tail, prev, qn)) break;
  }
  if (prev != NULL) {
    prev->next = qn; /*Label 2*/
    while (!qn->flag) { } // Spin
} }
```

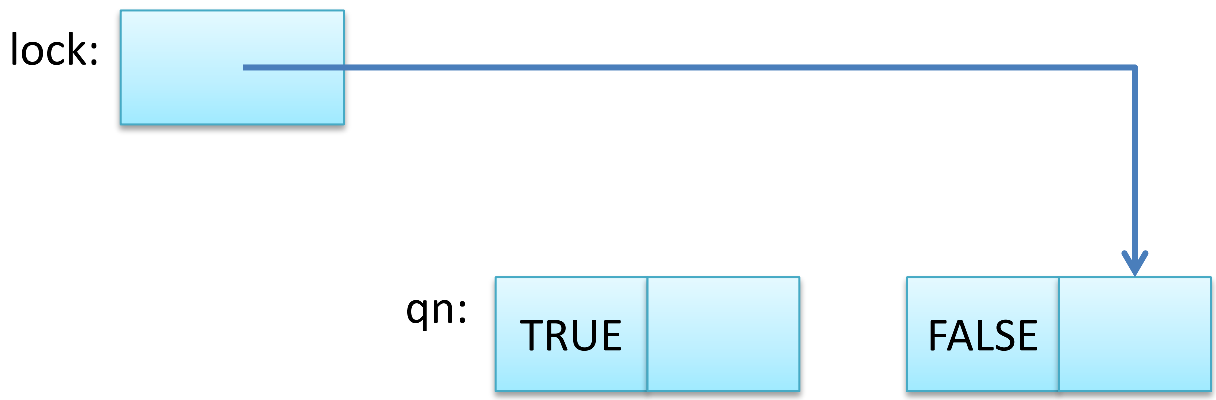
Find previous tail node

Atomically replace “prev” with “qn” in the lock itself

Add link within the queue

NB: the two labels in the source code are referred to in the exercise sheet; they are not necessary for the algorithm

MCS lock release




```
void releaseMCS(mcs *lock, QNode *qn) {
  if (lock->tail = qn) {
    if (CAS(&lock->tail, qn, NULL)) return;
  }
  while (qn->next != NULL) { }
  qn->next->flag = TRUE;
}
```

If we were at the tail then remove us

Wait for next lock holder to announce themselves; signal them

Test-and-set (TAS) locks

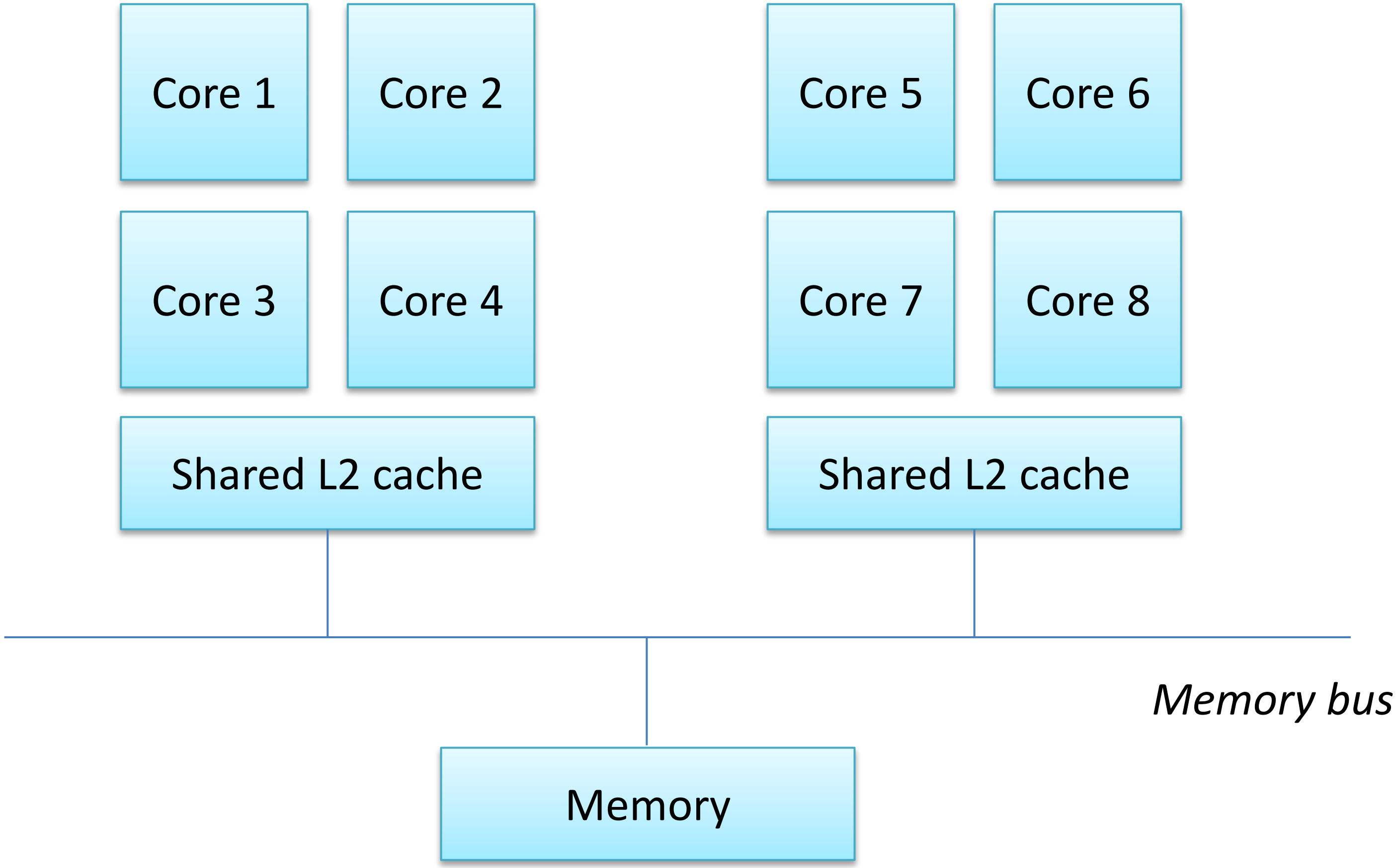
TATAS locks & backoff

Queue-based locks

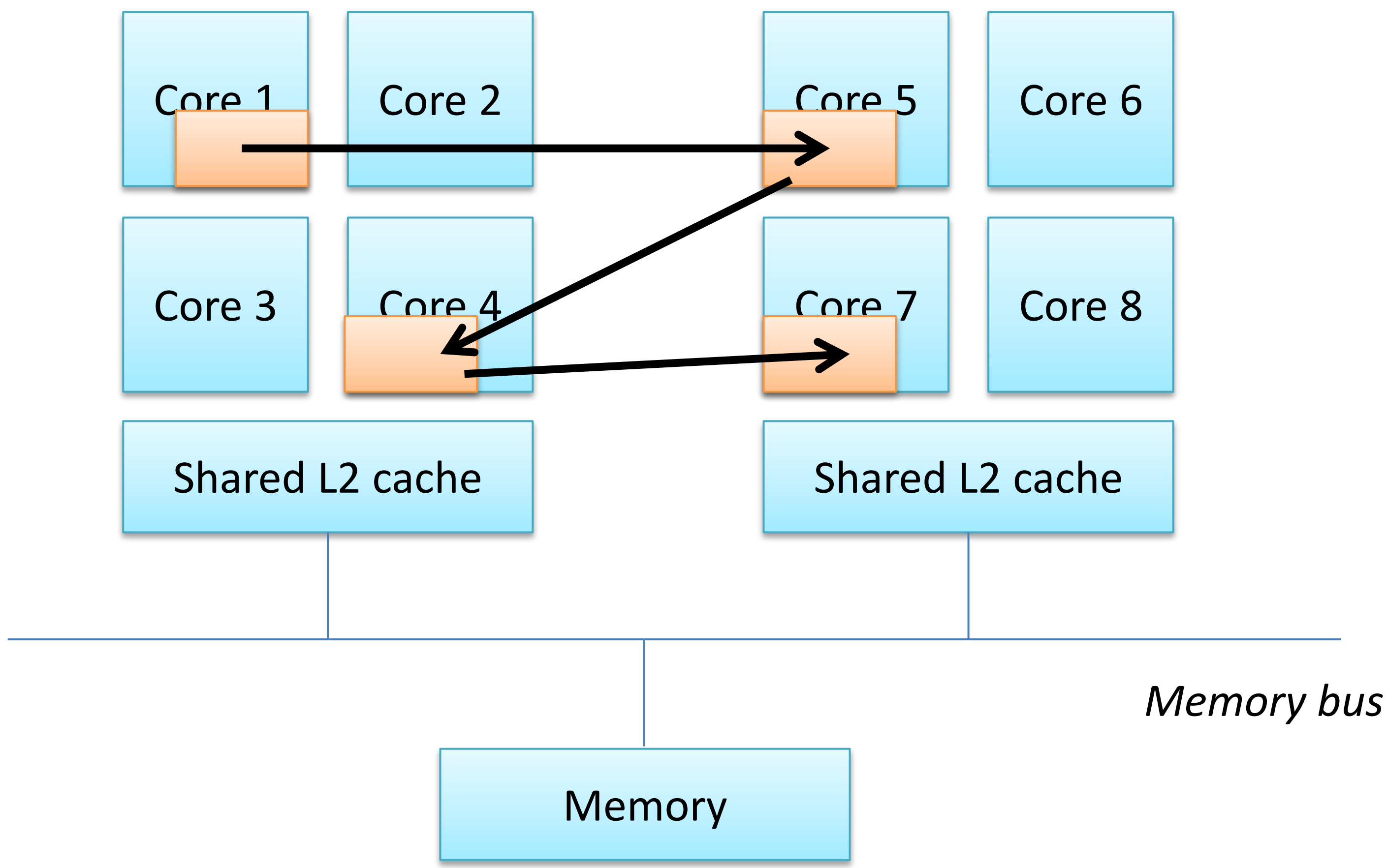
Hierarchical locks

Parallel performance

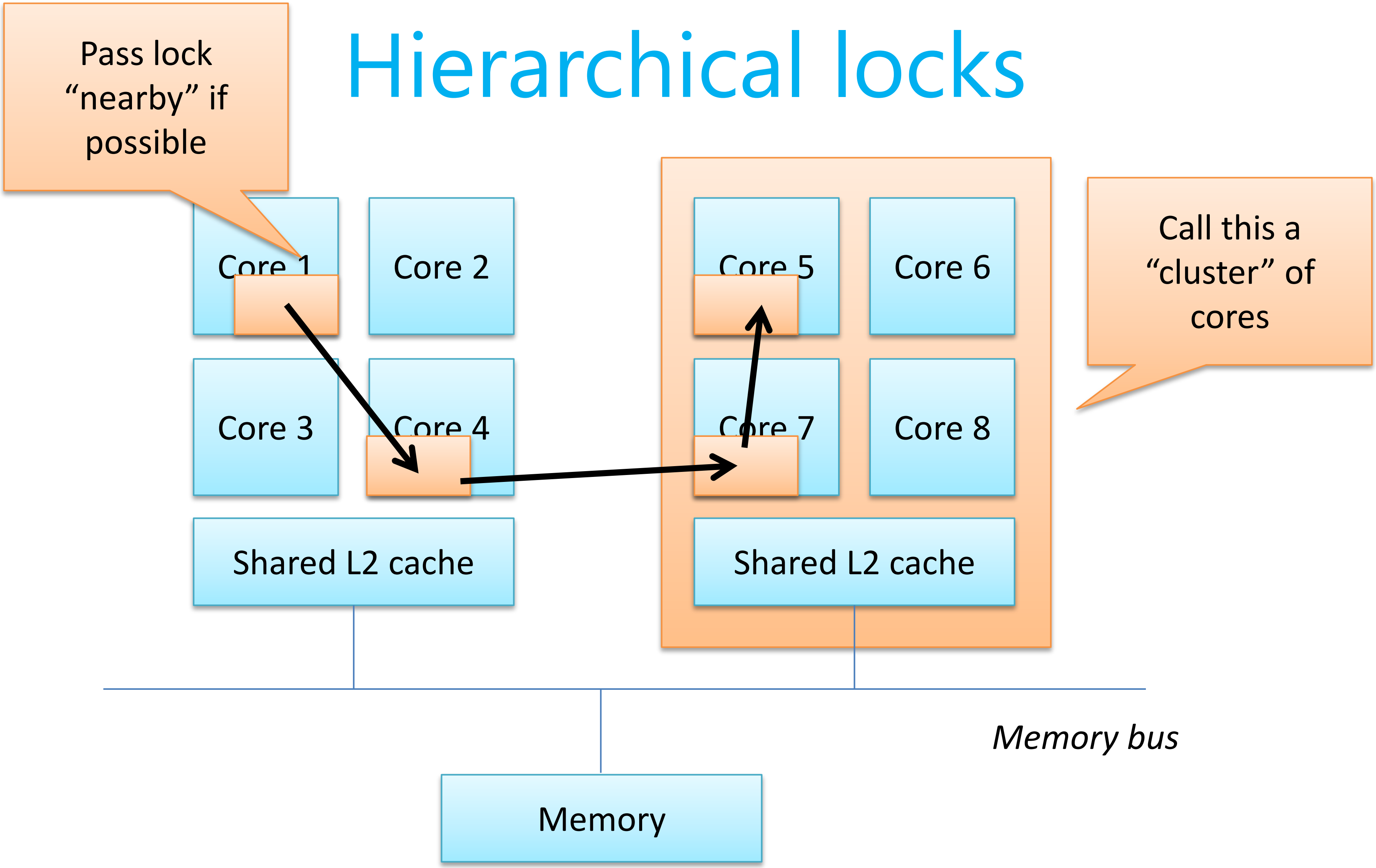
Hierarchical locks



Hierarchical locks



Hierarchical locks



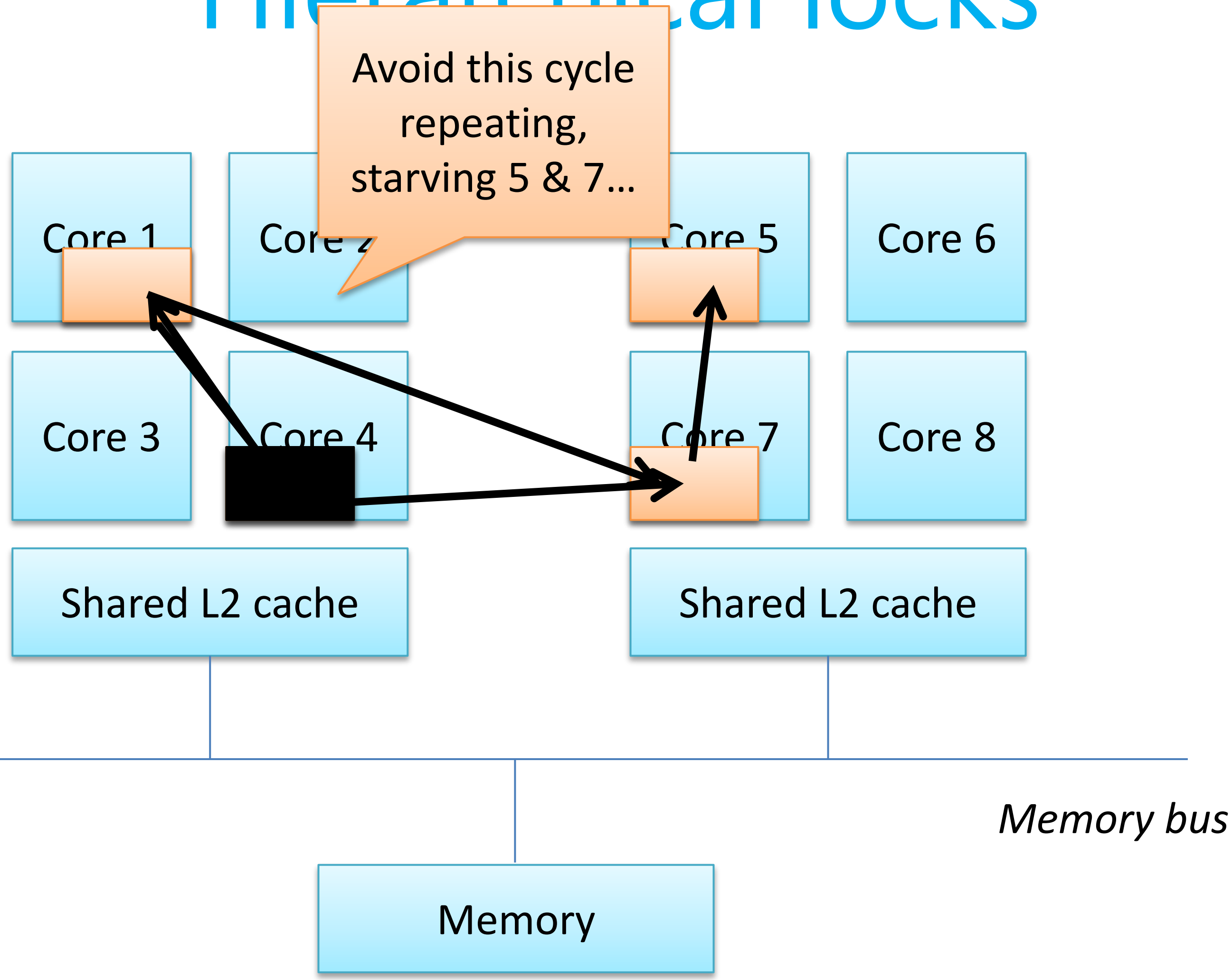
Hierarchical TATAS with backoff

lock: -1

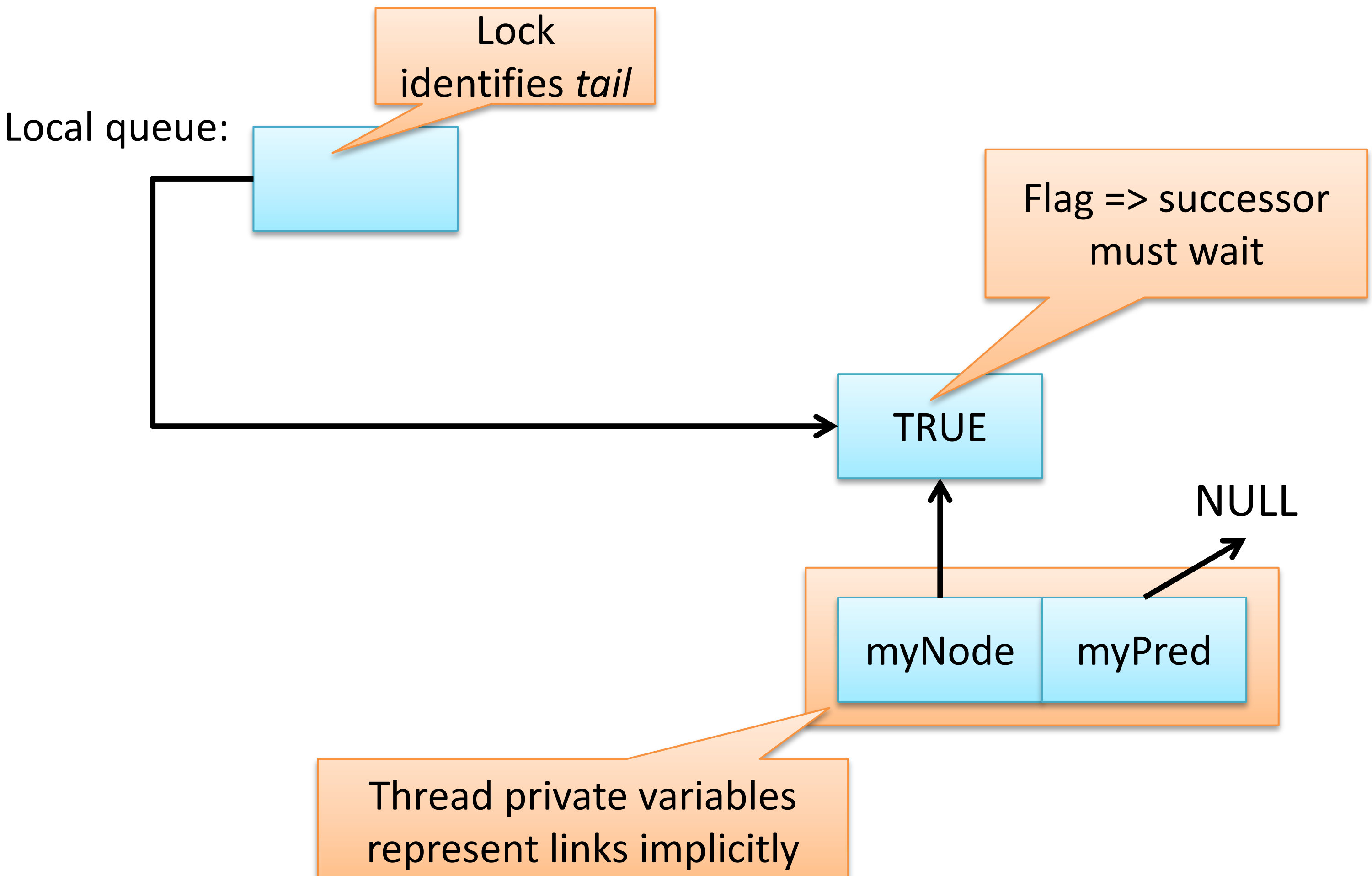
-1 => lock available
n => lock held by cluster n

```
void acquireLock(bool *lock) {
  do {
    holder = *lock;
    if (holder != -1) {
      if (holder == MY_CLUSTER) {
        BackOff(SHORT);
      } else {
        BackOff(LONG);
      }
    }
  } while (!CAS(lock, -1, MY_CLUSTER));
}
```

Hierarchical locks

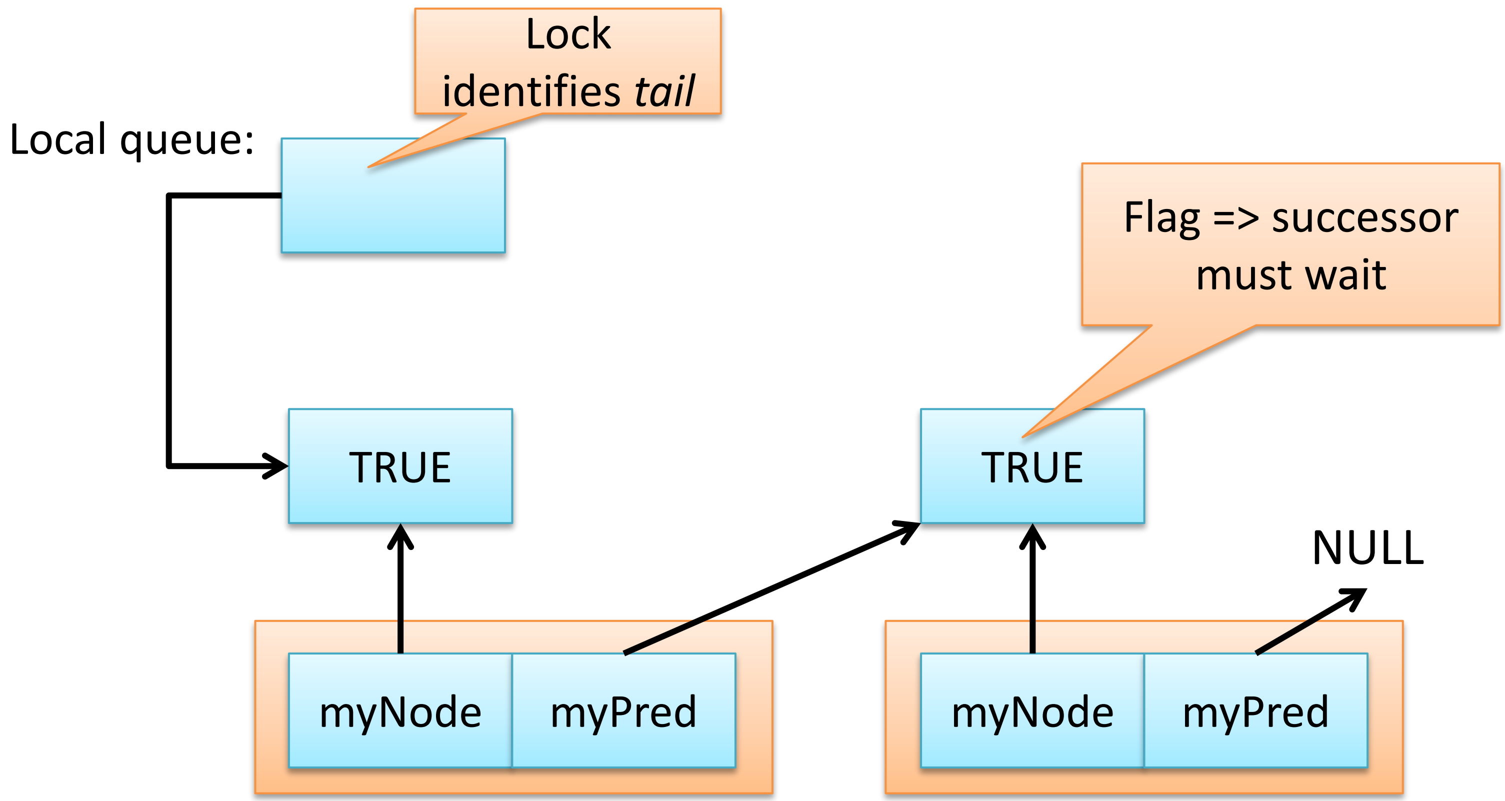


Hierarchical CLH queue lock



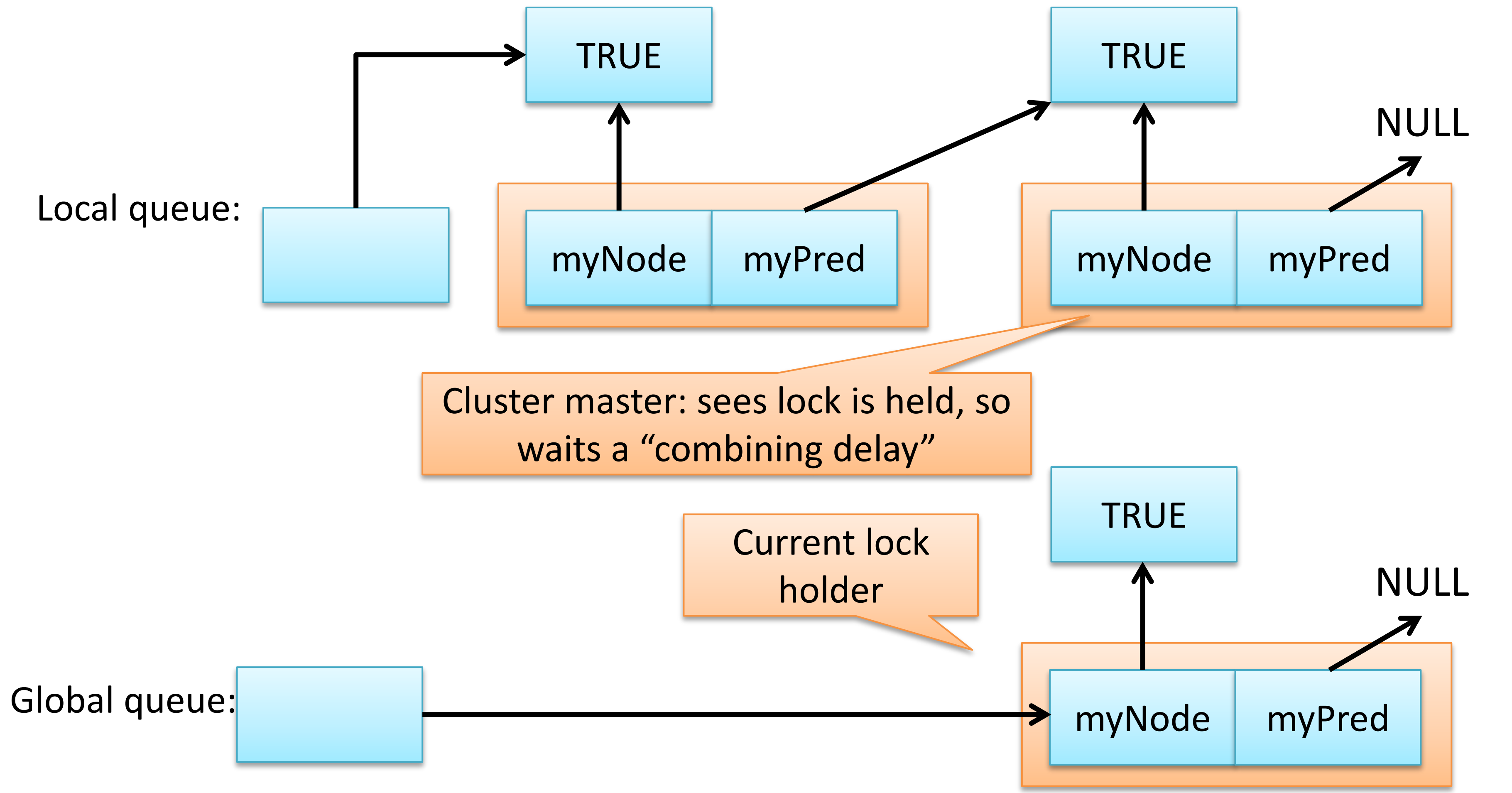
Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit

Hierarchical CLH queue lock



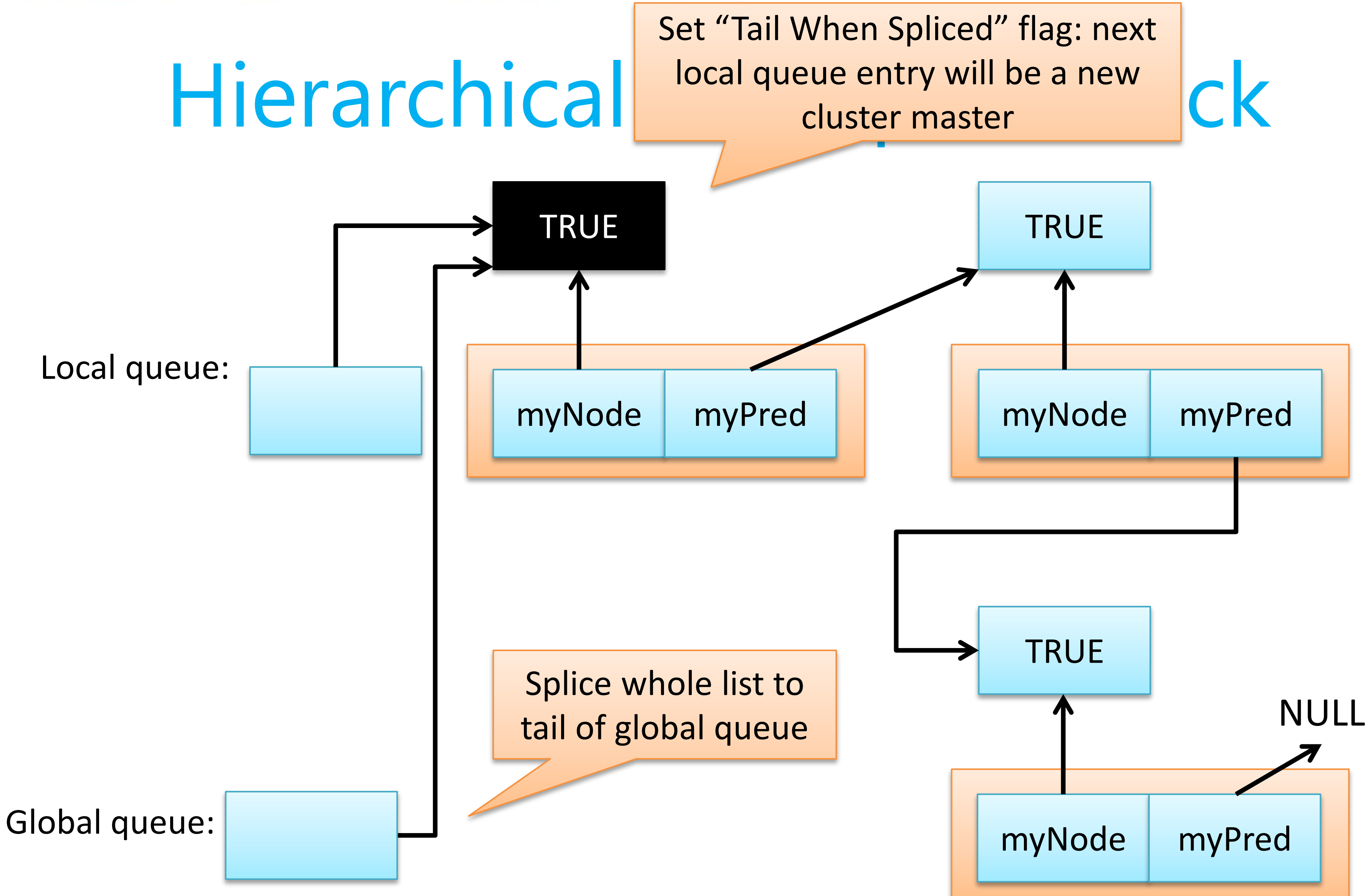
Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit

Hierarchical CLH queue lock



Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit

Hierarchical ck



Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit

Test-and-set (TAS) locks

TATAS locks & backoff

Queue-based locks

Hierarchical locks

Parallel performance

An aside: is this a better algorithm?

- How fast does it run without contention?
  - Each thread acquires and releases different locks
  - Threads acquire and release the same lock… but not at the same time
- How fast does it run with contention?
  - *n* threads trying to acquire the same lock at the same time
  - How does performance scale as *n* varies?

An aside: is this a better algorithm?

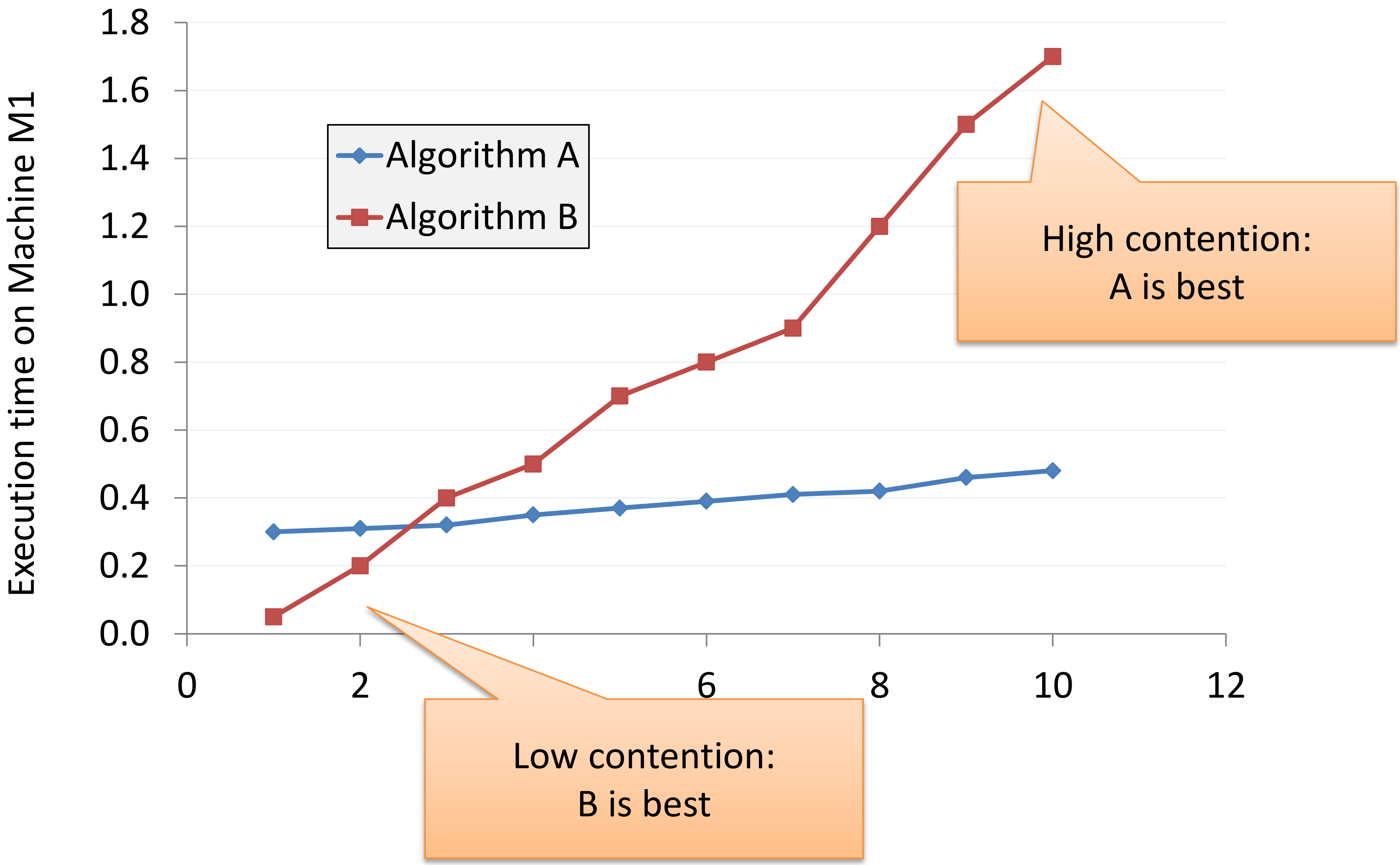
Wall-clock time?

Memory accesses?

Instruction count?

Cache lines read/written?

An aside: is this a better algorithm?



An aside: is this a better algorithm?