Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?

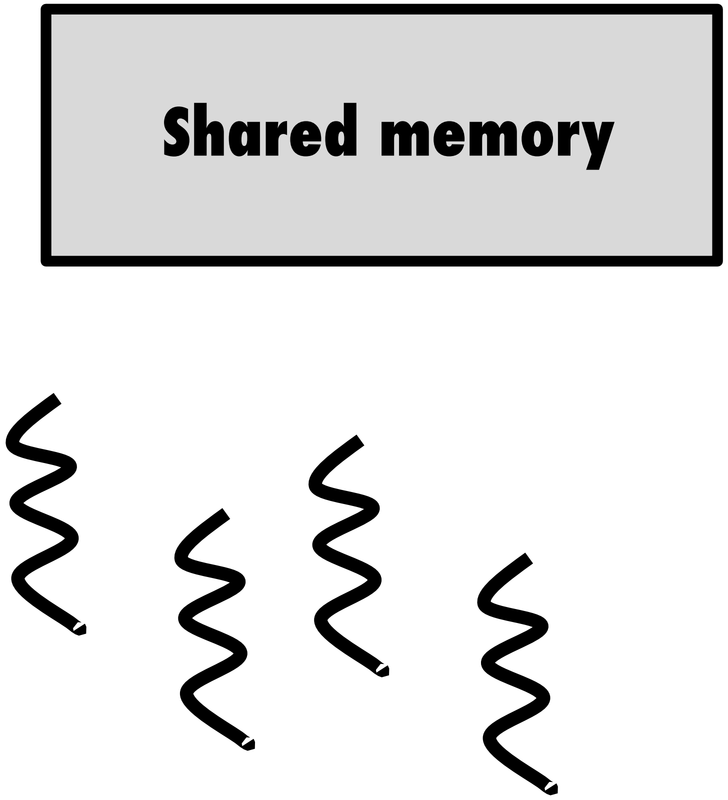
Jose Faleiro and Daniel Abadi
Yale University

Multi-core hardware is everywhere

- In the cloud
    - Amazon X1 instances with 64 physical cores
- On premise
    - Systems with > 50 cores are widely available
- Lots interest in building DBs for multi-core hardware

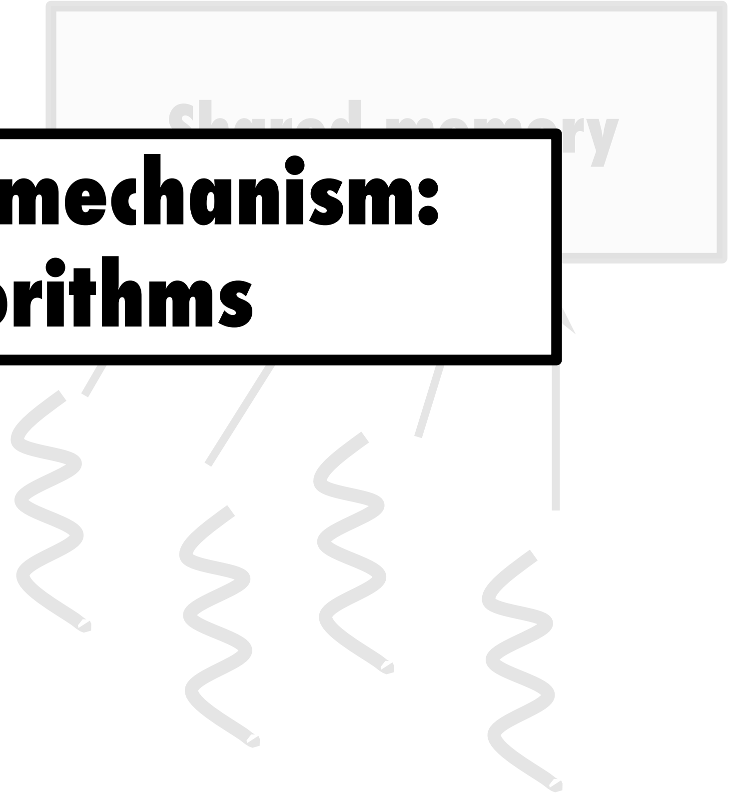
Multi-core systems need synchronization

- Parallelism via concurrent execution on CPU cores
- Communication via shared memory
- Access to shared memory must be **synchronized**
  - Prevents bugs due to race conditions



Multi-core systems need synchronization

- Parallelism via concurrent execution on CPU cores
- Communication via shared memory
- Access to shared memory must be **synchronized**
  - Prevents bugs due to race conditions



Two classes of synchronization mechanism:
Latches and latch-free algorithms

Latch-free synchronization: Silver bullet?

- Some recent research papers:

- “… fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well…” SIGMOD Workshop paper

- “ … latches are more likely to block, limiting scalability… Addressing [this] issue, XXX is latch-free.” ICDE paper

- “Scalability is often limited by contention on locks and latches … XXX is designed for high concurrency. To achieve this it uses only latch-free data structures” CIDR paper

Latch-free synchronization: Silver bullet?

- Some recent research papers:
- "… fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well…" SIGMOD Workshop paper
- " … latches are more likely to block, limiting scalability… Addressing [this] issue, XXX is latch-free." ICDE paper
- "Scalability is often limited by contention on locks and latches … XXX is designed for high concurrency. To achieve this it uses only latch-free data structures" CIDR paper

Latch-free synchronization: Silver bullet?

- Some recent research papers:
- "… fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well…" SIGMOD Workshop paper
- " … latches are more likely to block, limiting scalability… Addressing [this] issue, XXX is latch-free." ICDE paper
- "Scalability is often limited by contention on locks and latches … XXX is designed for high concurrency. To achieve this it uses only latch-free data structures" CIDR paper

Latch-free synchronization: Silver bullet?

- Some recent research papers:
- "… fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well…" SIGMOD Workshop paper
- " … latches are more likely to block, limiting scalability… Addressing [this] issue, XXX is latch-free." ICDE paper
- "Scalability is often limited by contention on locks and latches … XXX is designed for high concurrency. To achieve this it uses only latch-free data structures" CIDR paper

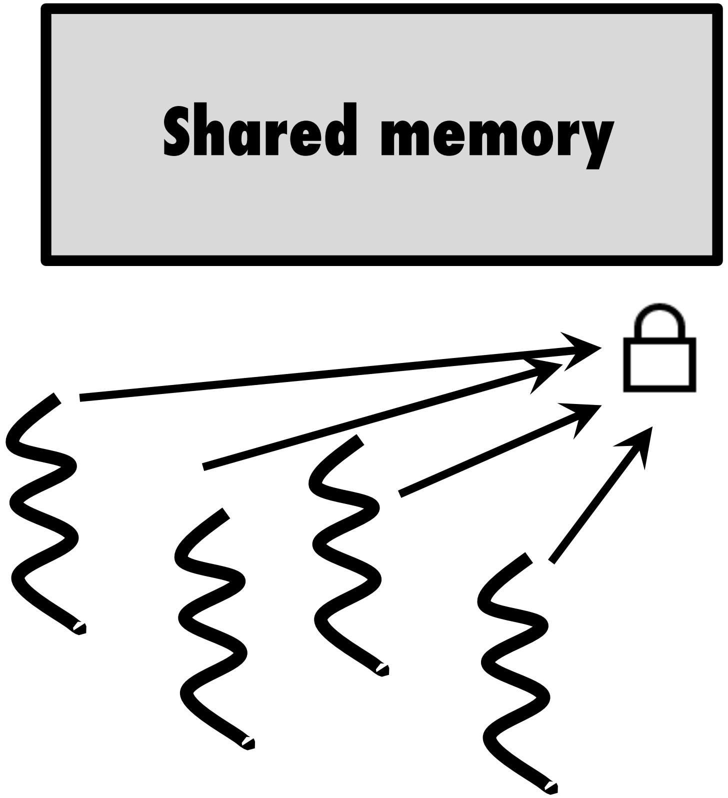
Latch-free synchronization: Silver bullet?

- Some recent research papers:
- "… fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well…" SIGMOD Workshop paper
- " … latches are more likely to block, limiting scalability… Addressing [this] issue, XXX is latch-free." ICDE paper
- "Scalability is often limited by contention on locks and latches … XXX is designed for high concurrency. To achieve this it uses only latch-free data structures" CIDR paper

Latches

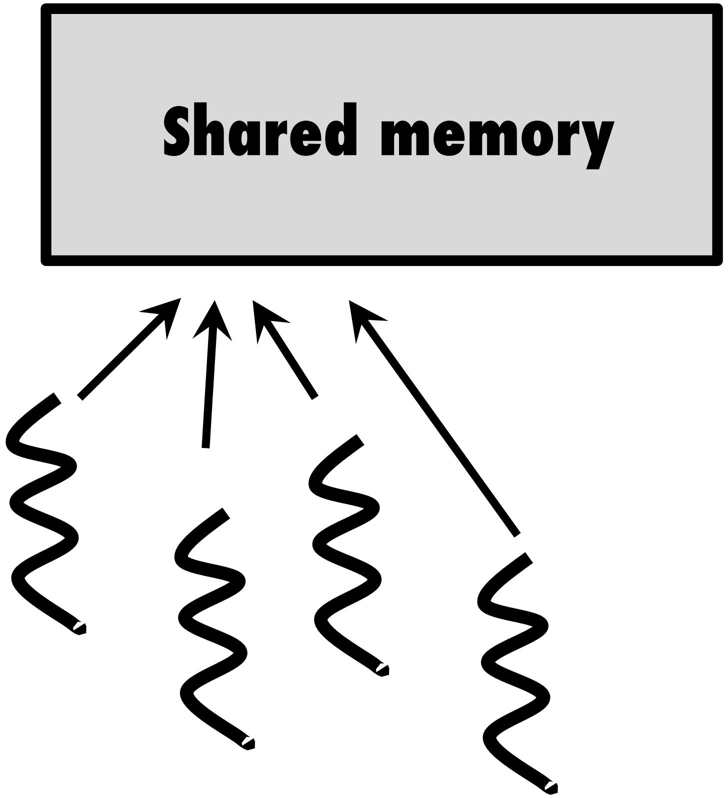
- Prevent conflicting threads from accessing shared data

- Latch is just a memory word
    - Combinations of reads & writes acquire the latch



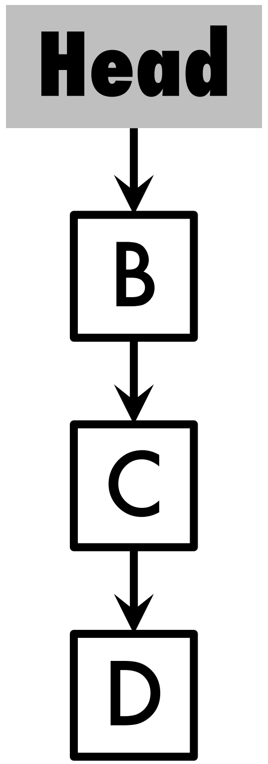
Latch-free

- Implicit synchronization
  - Directly operate on shared data
- Use atomic instructions for correctness
  - Compare-and-swap
- Cores concurrently attempt updates on one word



Latch-free list

```
Insert(A):

while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```

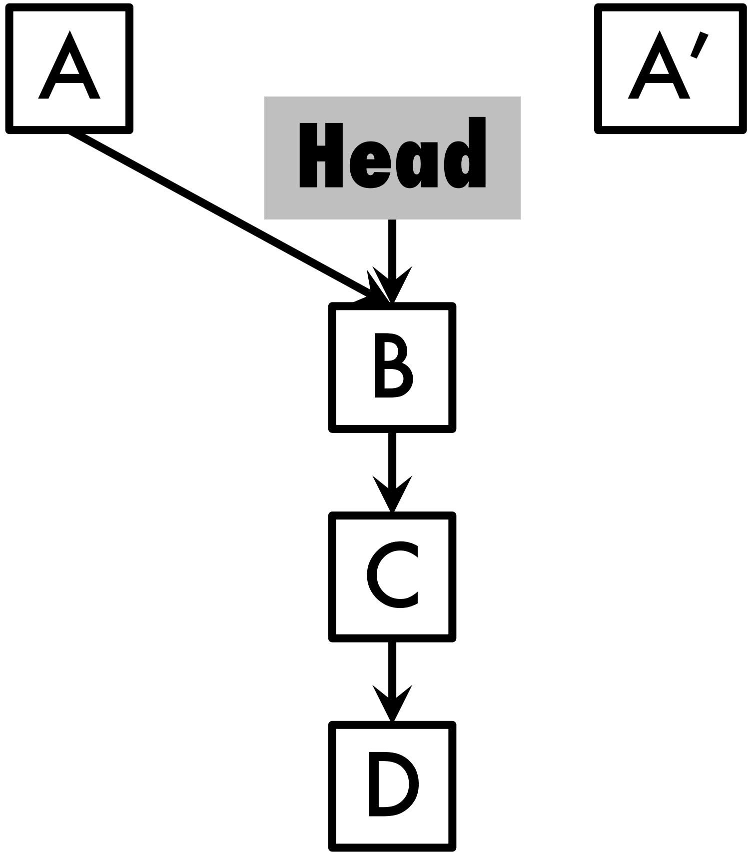


Latch-free list

```
Insert(A):
while true
```

1. A.next = Head

```
Insert(A’):
while true
```



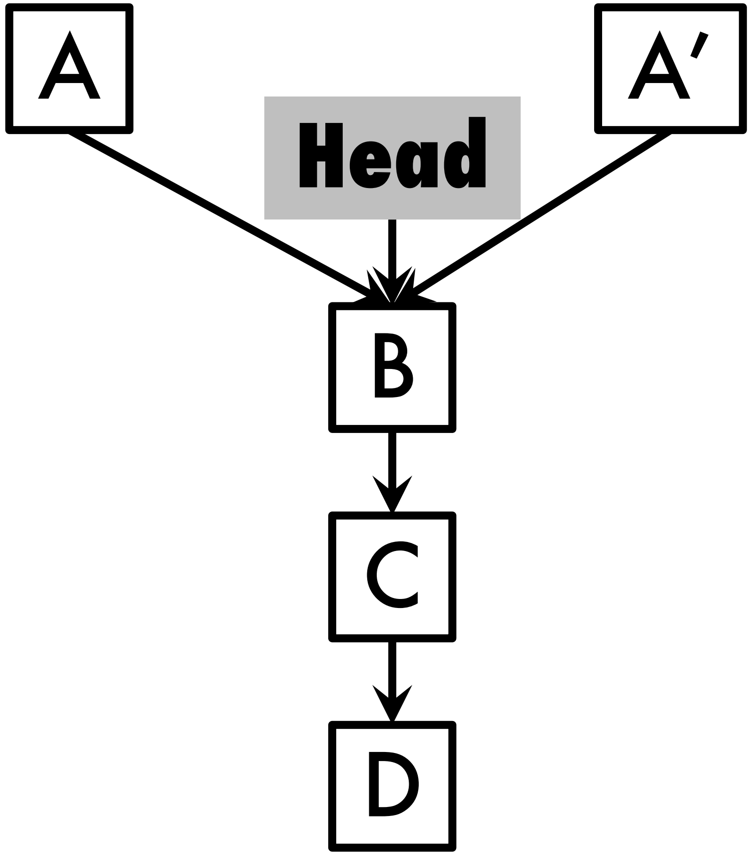
Latch-free list

```
Insert(A):
while true
```

1. A.next = Head

```
Insert(A'):
while true
```

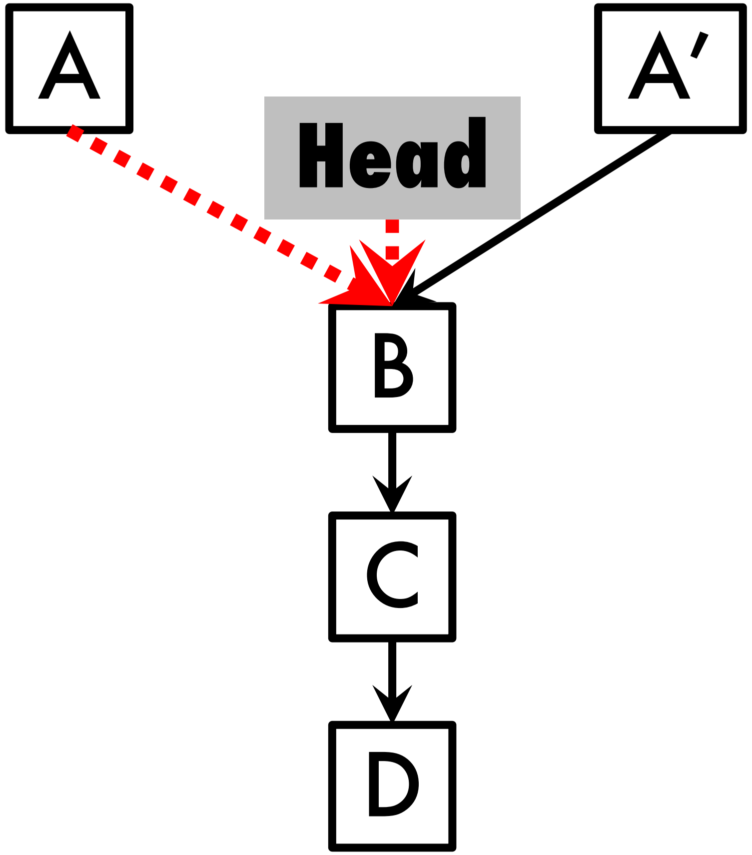
2. A'.next = Head



Latch-free list

```
Insert(A):
while true
1. A.next = Head

3. if
   cmp_n_swap(&Head,
          A.next, A)
          break
```

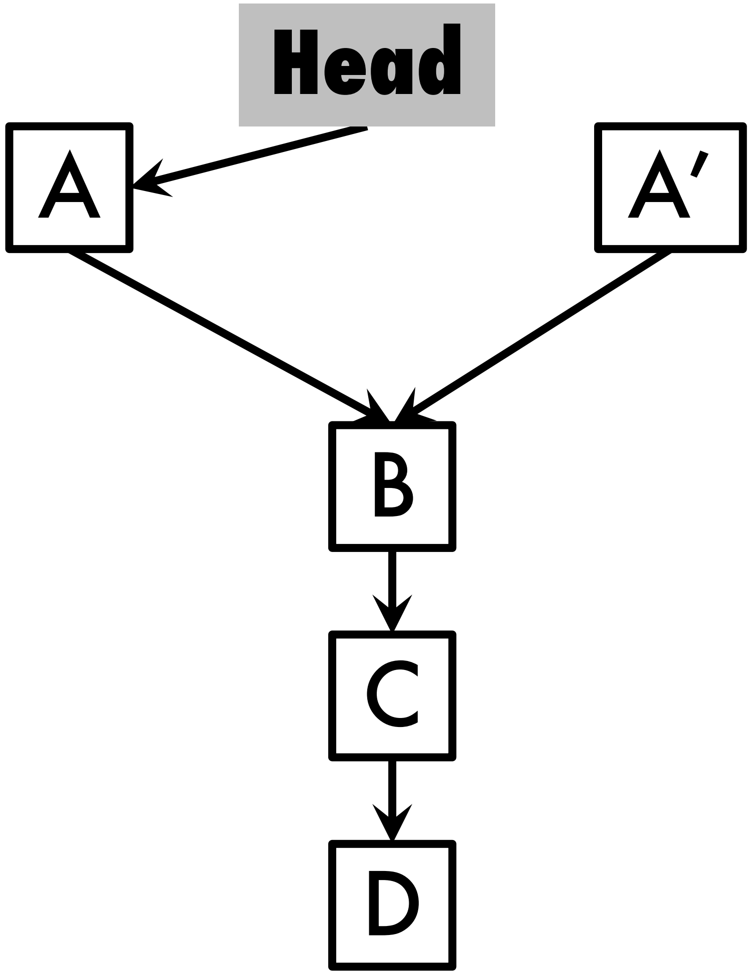
```
Insert(A’):
while true

2. A’.next = Head
```



Latch-free list

```
Insert(A):
while true
1. A.next = Head


3.  if
    cmp_n_swap(&Head,
          A.next, A)
          break
```

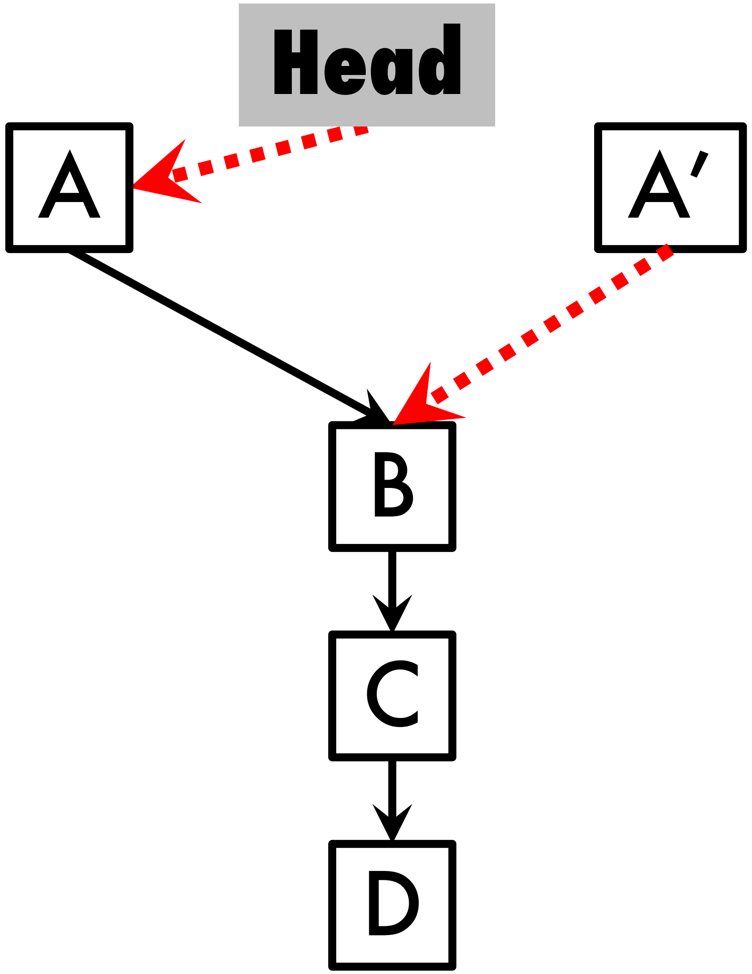
```
Insert(A'):
while true

2. A'.next = Head
```



Latch-free list

```
Insert(A):
while true
1. A.next = Head
3. if
   cmp_n_swap(&Head,
         A.next, A)
         break
```

```
Insert(A’):
while true
2. A’.next = Head
4. if
   cmp_n_swap(&Head,
         A’.next, A’)
         break
```



Latching vs latch-free algorithms

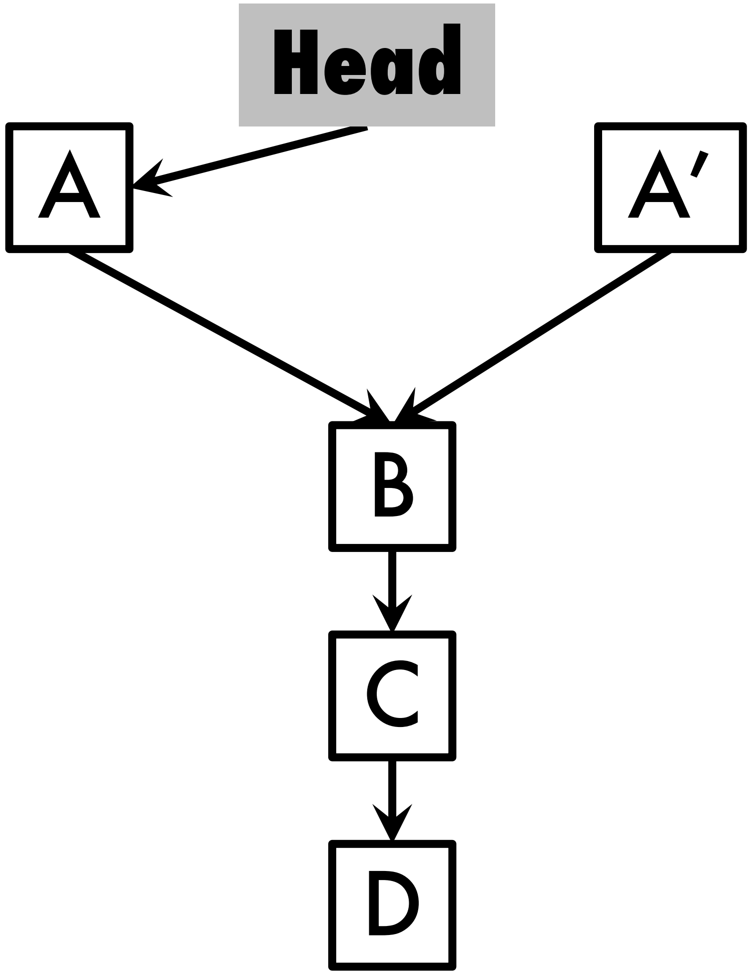
- Latch-free: strong progress guarantees
    - A thread is never blocked due to other threads

- Latches make no guarantees
    - If latch holder is delayed, another thread cannot acquire the latch

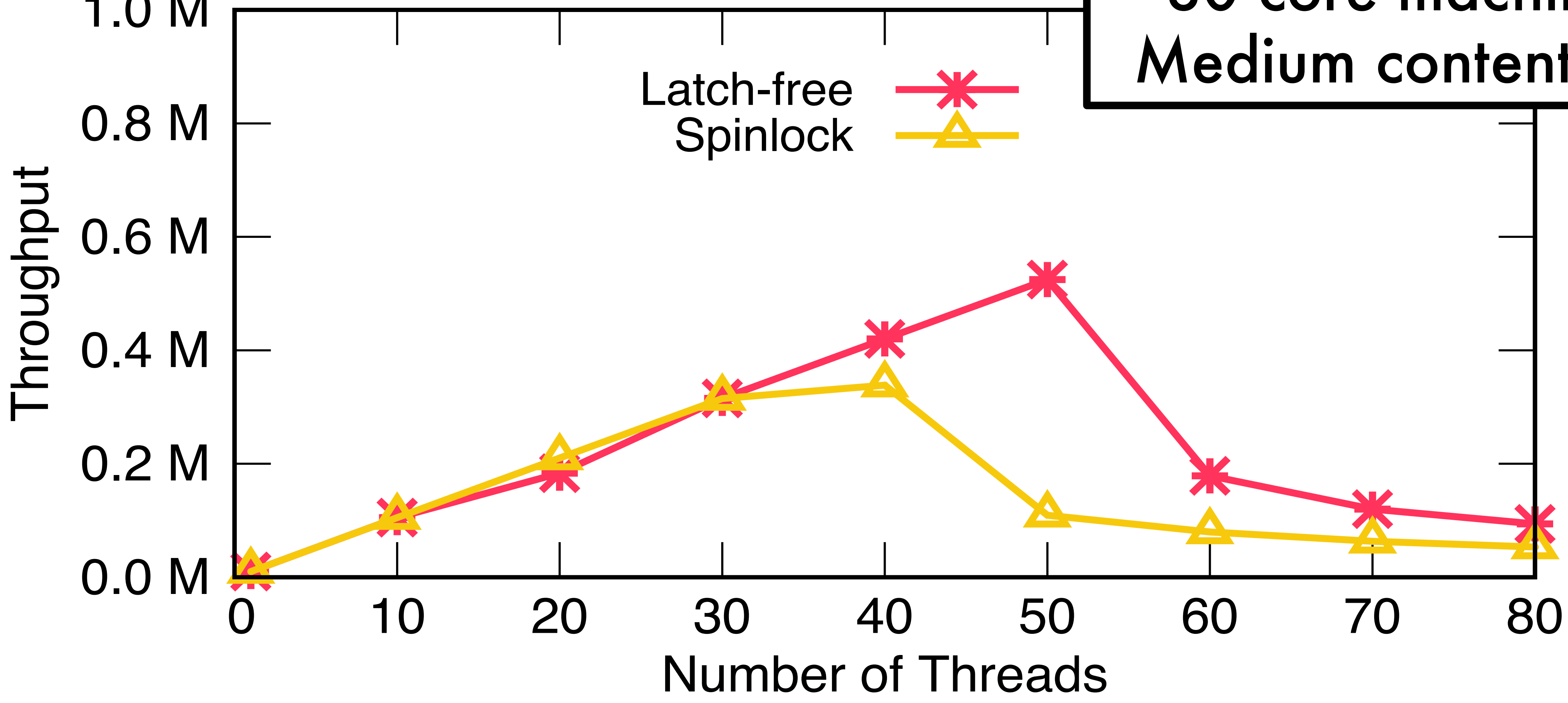
Latch-free list

```
Insert(A):
while true
1. A.next = Head

3. if
   cmp_n_swap(&Head,
          A.next, A)
          break
```

```
Insert(A’):
while true
2. A’.next = Head

4. if
   cmp_n_swap(&Head,
          A.next, A)
          break
```



Latching vs latch-free scalability



80 core machine
Medium contention

Rules of thumb

- Writes: Concurrent writes to the same location are processed serially
  - Expected time to write proportional to #concurrent writers

- Reads: Concurrent readers “notice” a change in a word’s value serially
  - Expected time to “notice” proportional to #concurrent readers
  - Prevents optimizations such as “test-and-test-and-set”
  - **Not discussed in this talk**

Looping while doing these is not a good idea

Latch implementations

- Spinlocks
    - Cores repeatedly attempt to read or write a global location
    - Test-and-set, Test-and-test-and-set, ticket latches

- Acquire latch

```
while (test-and-set(&word, 1) == 1)
             ;
```

- Release latch

```
atomic-set(&word, 0)
```

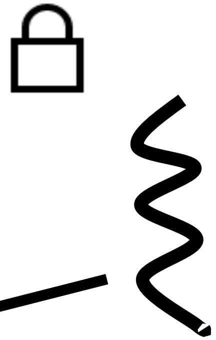
Latch implementations

- Acquire latch

```
while (test-and-set(&word, 1) == 1)
            ;
```

- Release latch

```
atomic-set(&word, 0)
```



A thread currently holds the latch

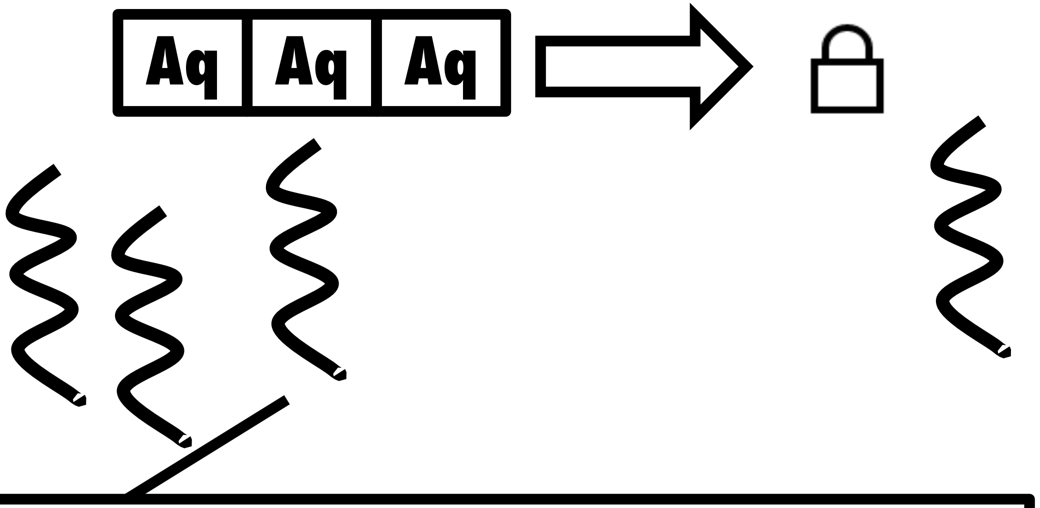
Latch implementations

- Acquire latch

```
while (test-and-set(&word, 1) == 1)
             ;
```

- Release latch

```
atomic-set(&word, 0)
```



Others threads attempt to acquire the latch (unsuccessfully)

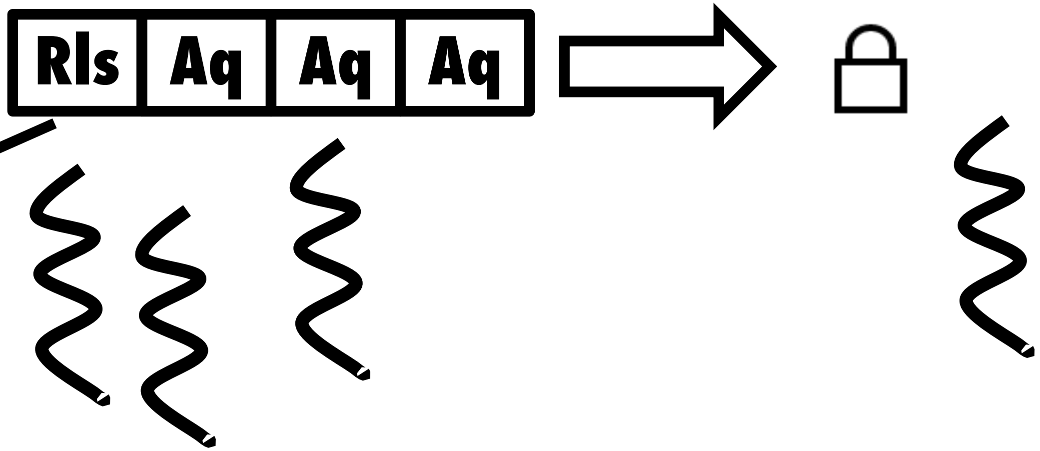
Latch implementations

- Acquire latch

```
while (test-and-set(&word, 1) == 1)
             ;
```

- Release latch

```
atomic-set(&word, 0)
```



Unsuccessful acquires compete with release
Increases critical section length

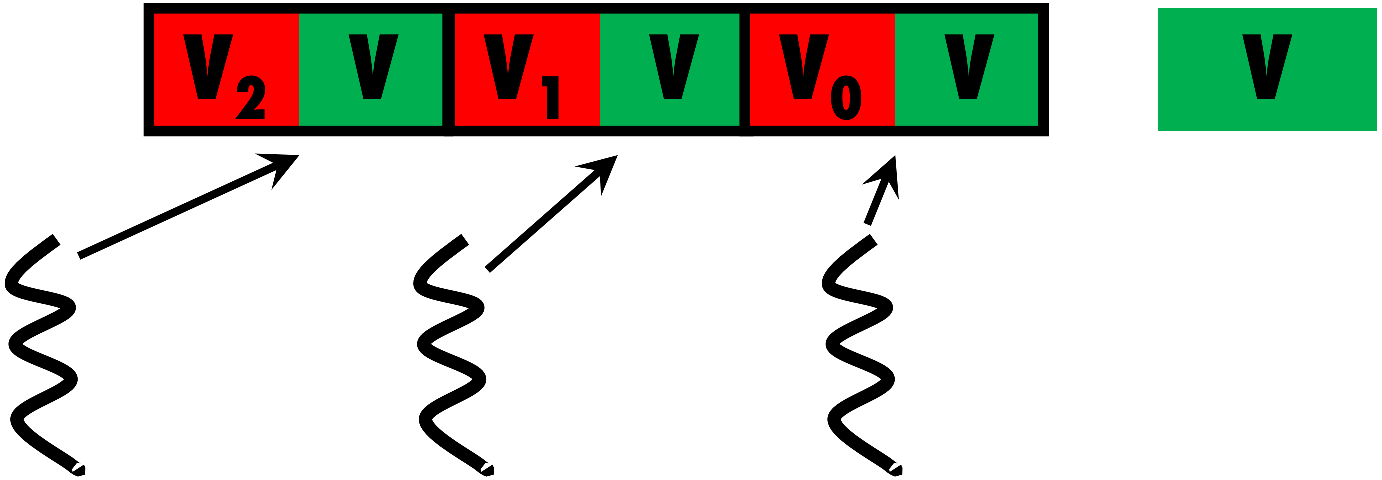
Latch-free algorithms

- Structure of a latch-free algorithm
  - Read value of word
  - Perform some computation
  - “Commit” via compare-and-swap on previously read word
  - If commit fails, retry

```
Insert(A):
while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```

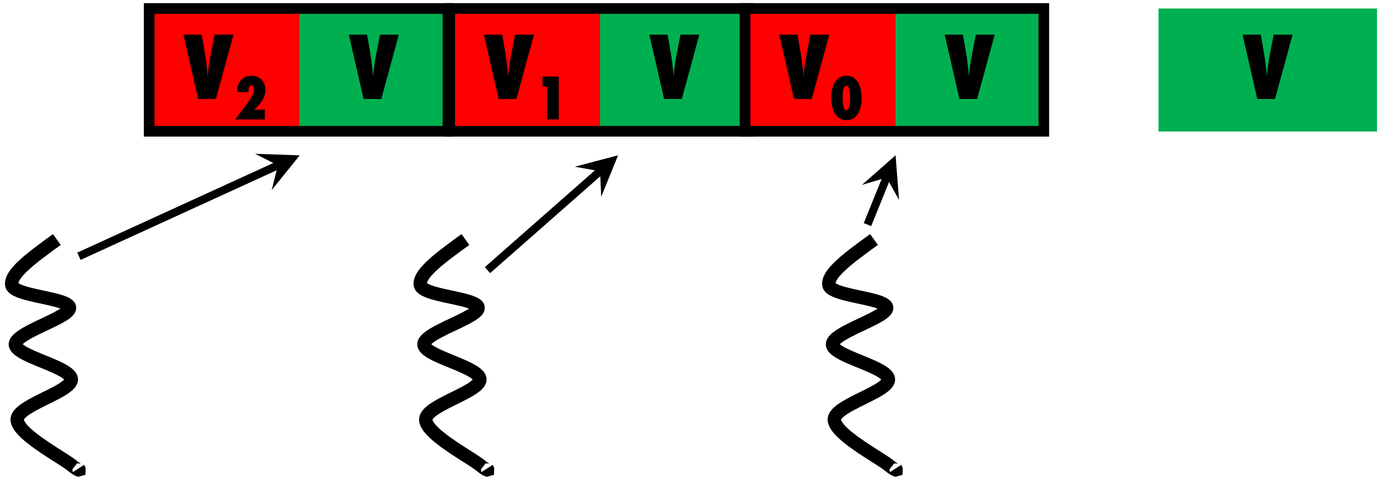
Latch-free algorithms

```
Insert(A):
while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```



Latch-free algorithms

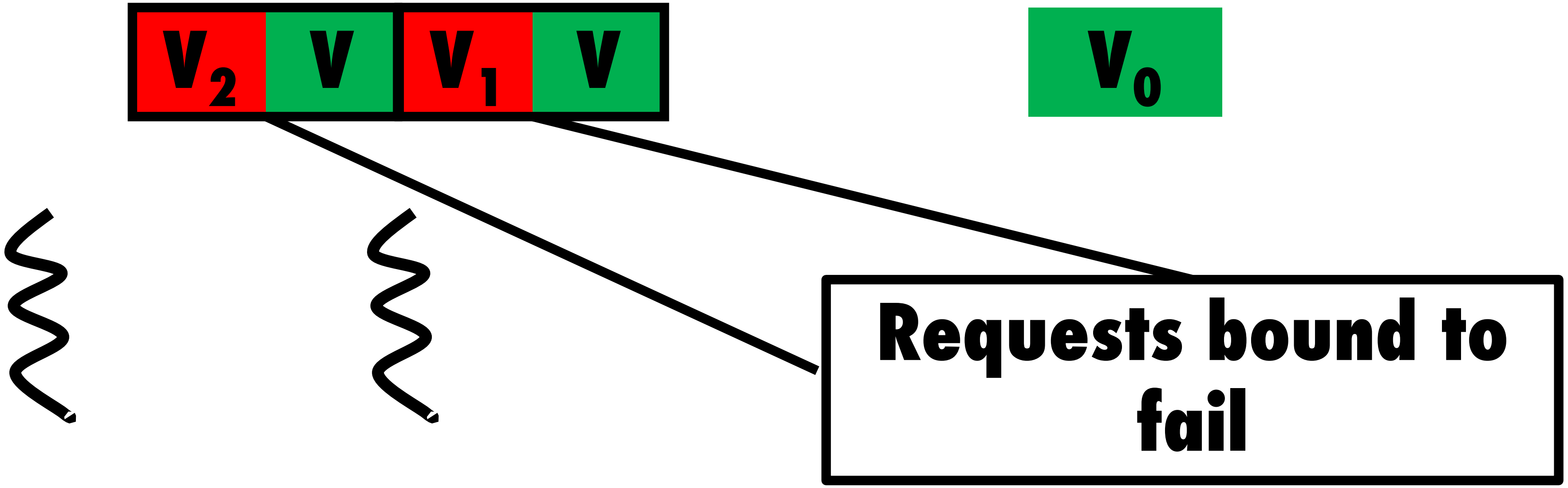
```
Insert(A):
while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```



Only first request succeeds

Latch-free algorithms

```
Insert(A):
while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```

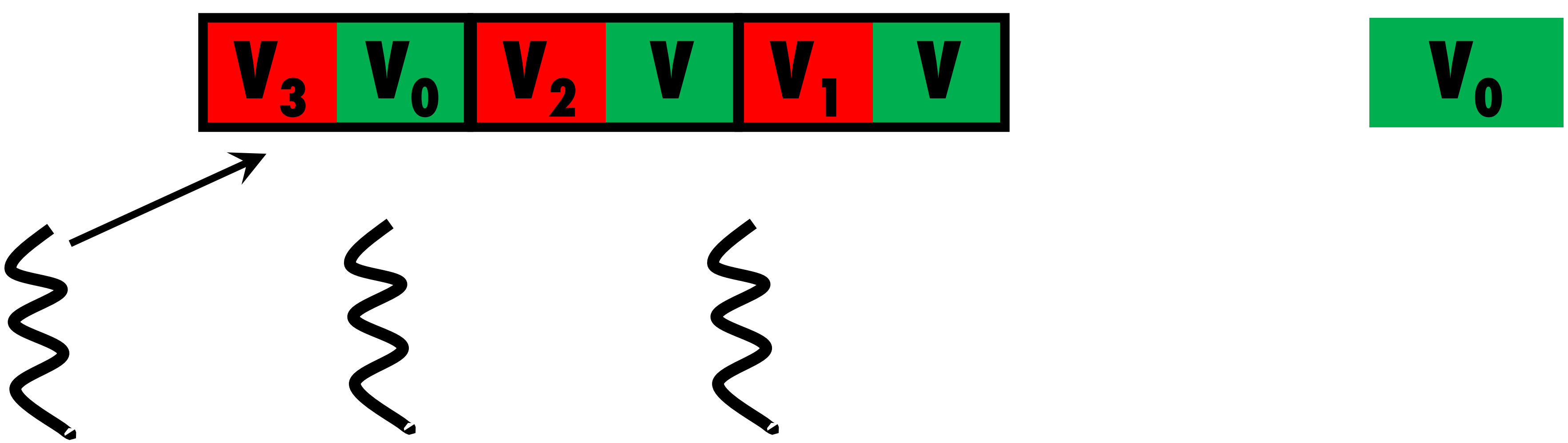


Requests bound to fail

Latch-free algorithms

```
Insert(A):
while (true)
  A.next = Head
  if cmp_n_swap(&Head, A.next, A)
    break
```

Later requests delayed by those bound to fail



But we’re not done …

- Some latches do not busy-wait on a single word

- “Scalable” latches
    - Each core spins on a different word

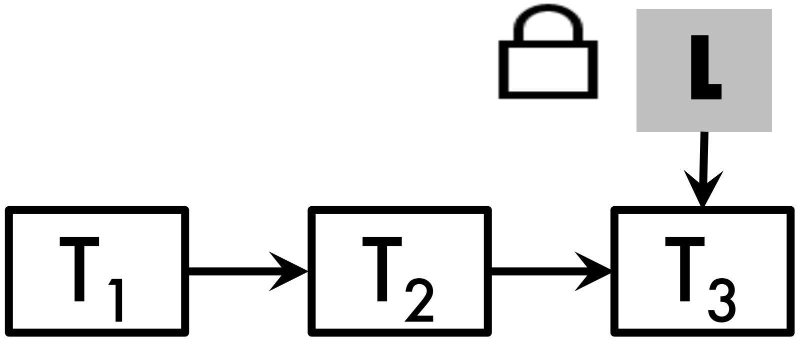
Scalable latches

- MCS latches

- To acquire latch, thread appends a queue node
  - Single non-failing instruction

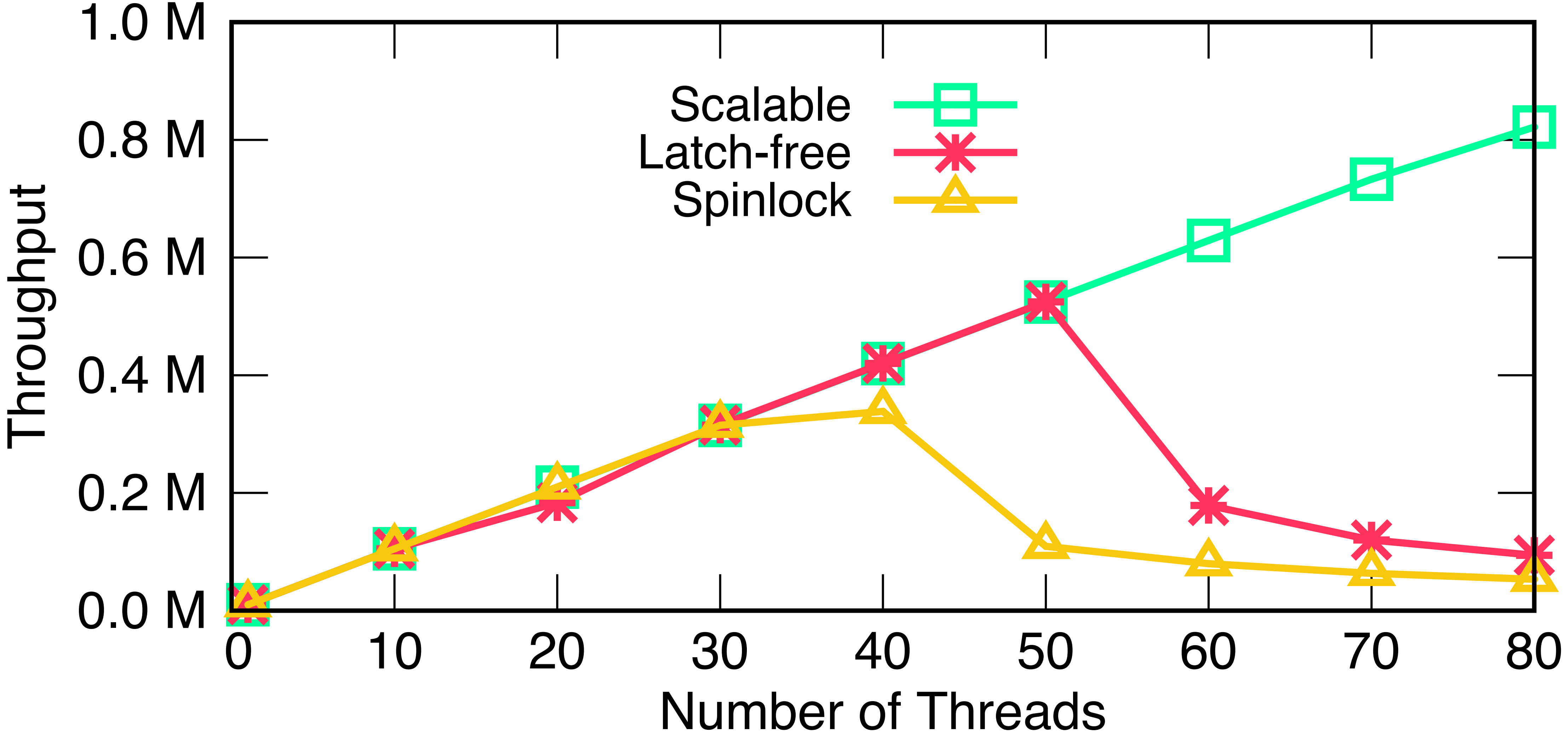
- First node is latch holder

- Latch holder signals the next thread
  - T_1 will signal T_2

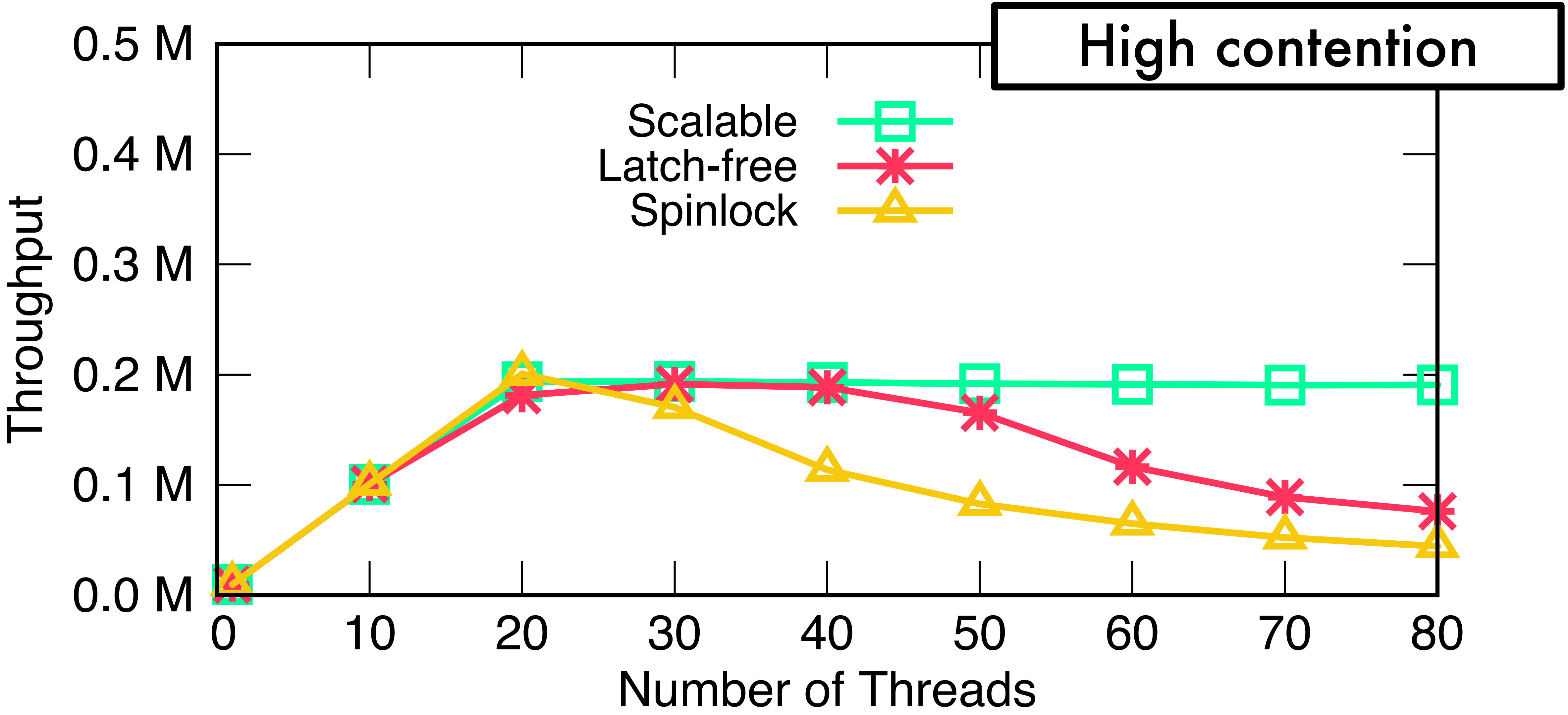


No busy waiting on a single word

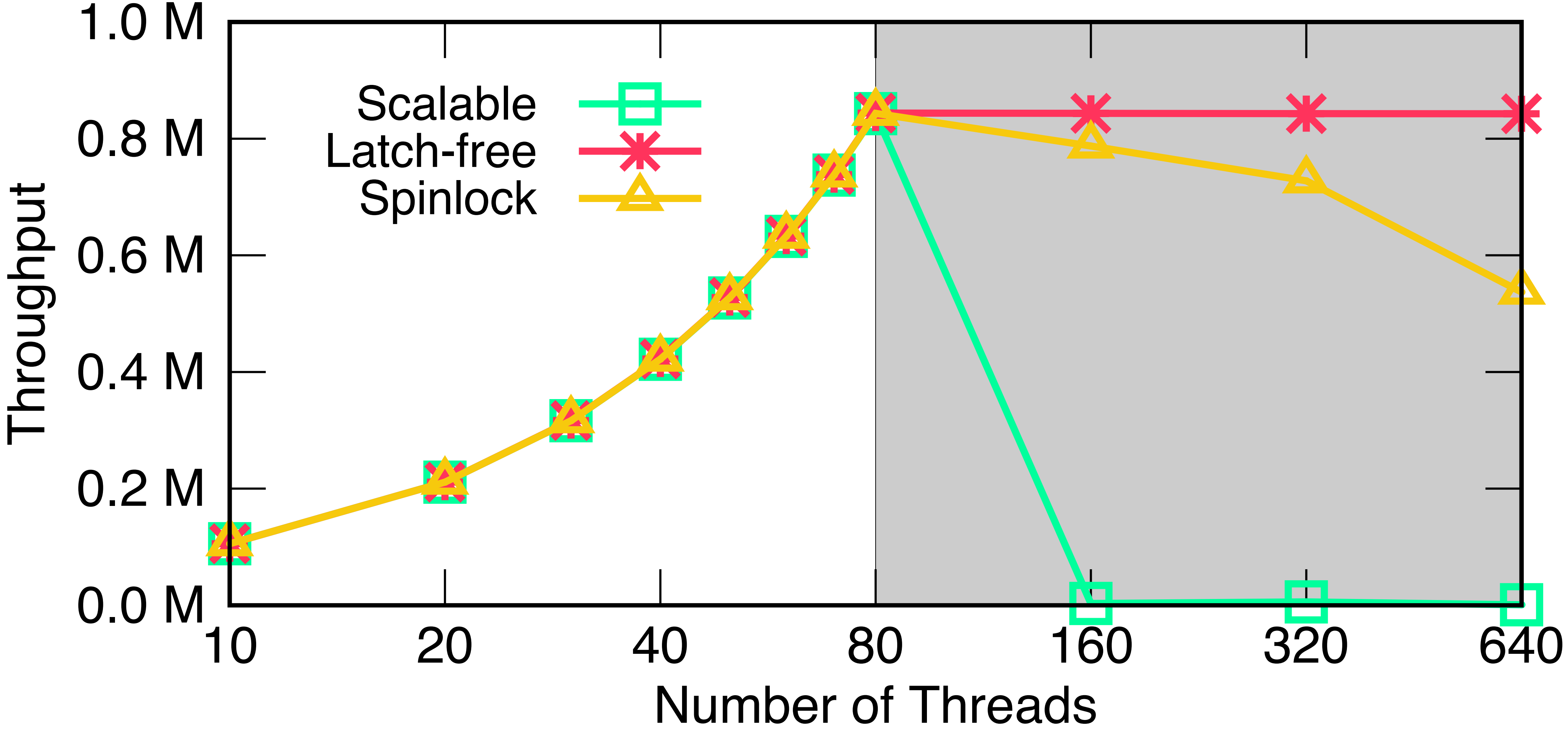
Latching vs latch-free; medium contention



Latching vs latch-free; high contention



Latching vs latch-free; low contention



Thread/process allocation

- # threads > # cores?
  - Bad for latches
  - Latch holder preemption in spinlocks
  - Preemption of any waiting thread in scalable latches

- Latch-free algorithms are robust to preemption
  - One thread is never delayed due to other threads

Scheduling requests

- Most leading DBMSs => request to OS thread (or process)
- Admission control typically allows significantly more requests than cores
  - Inevitably end up with more OS threads than cores

Scheduling requests

- No fundamental reason to assign request to an OS thread

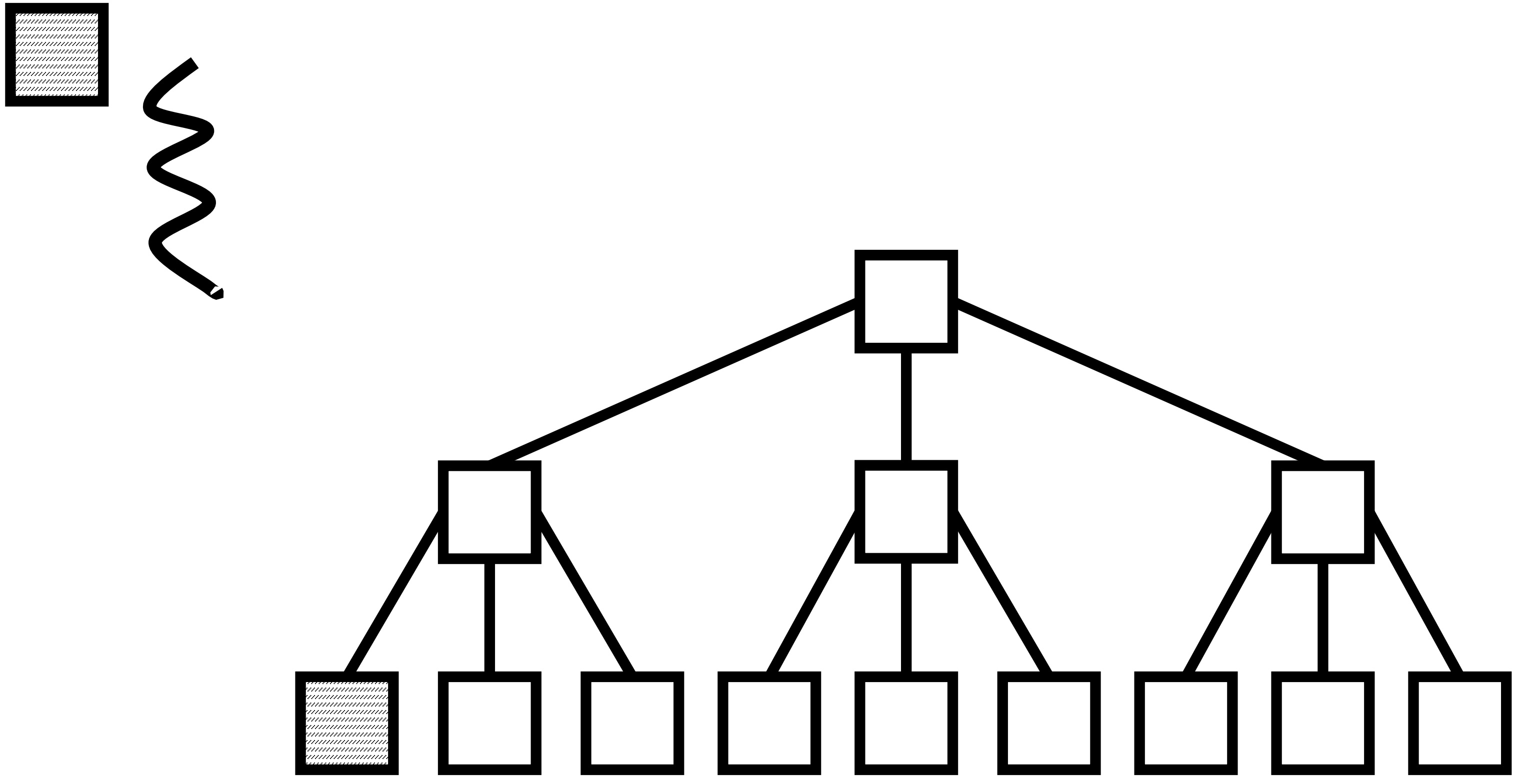
- DBMS can multiplex requests across fixed set of OS threads
  - Commercial systems – VoltDB, recent Microsoft products
  - DBMSs have been doing this for decades

- User-level scheduling mechanisms
  - Scheduler activations
  - User mode scheduling in Windows

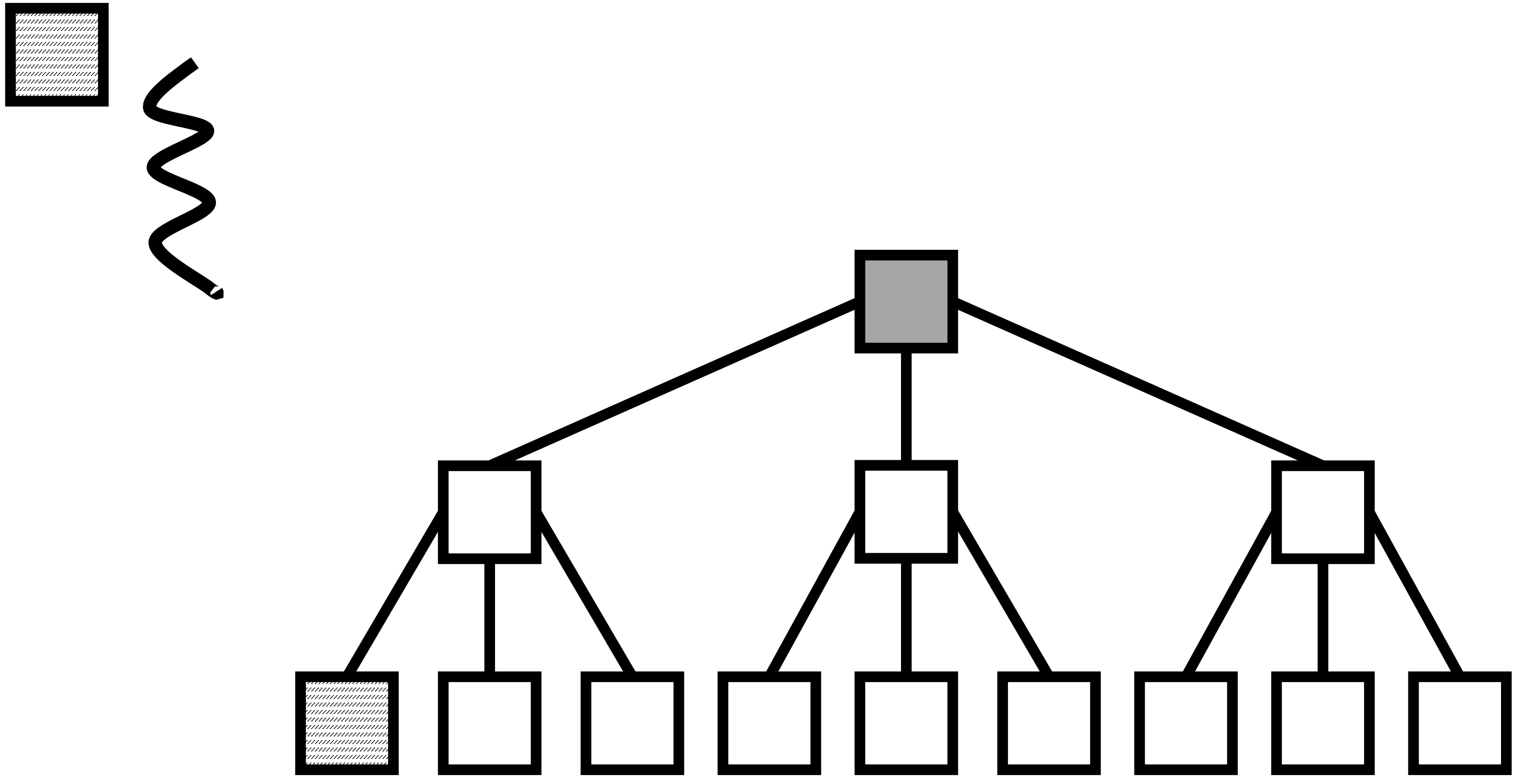
B^+tree example

- B^+tree concurrency control implemented via latch-coupling to leaves in shared-mode
    - ARIES/IM, B^{link} trees
- Root latch is acquired on **every** descent
    - Even if acquired in shared mode, latch meta-data is contended

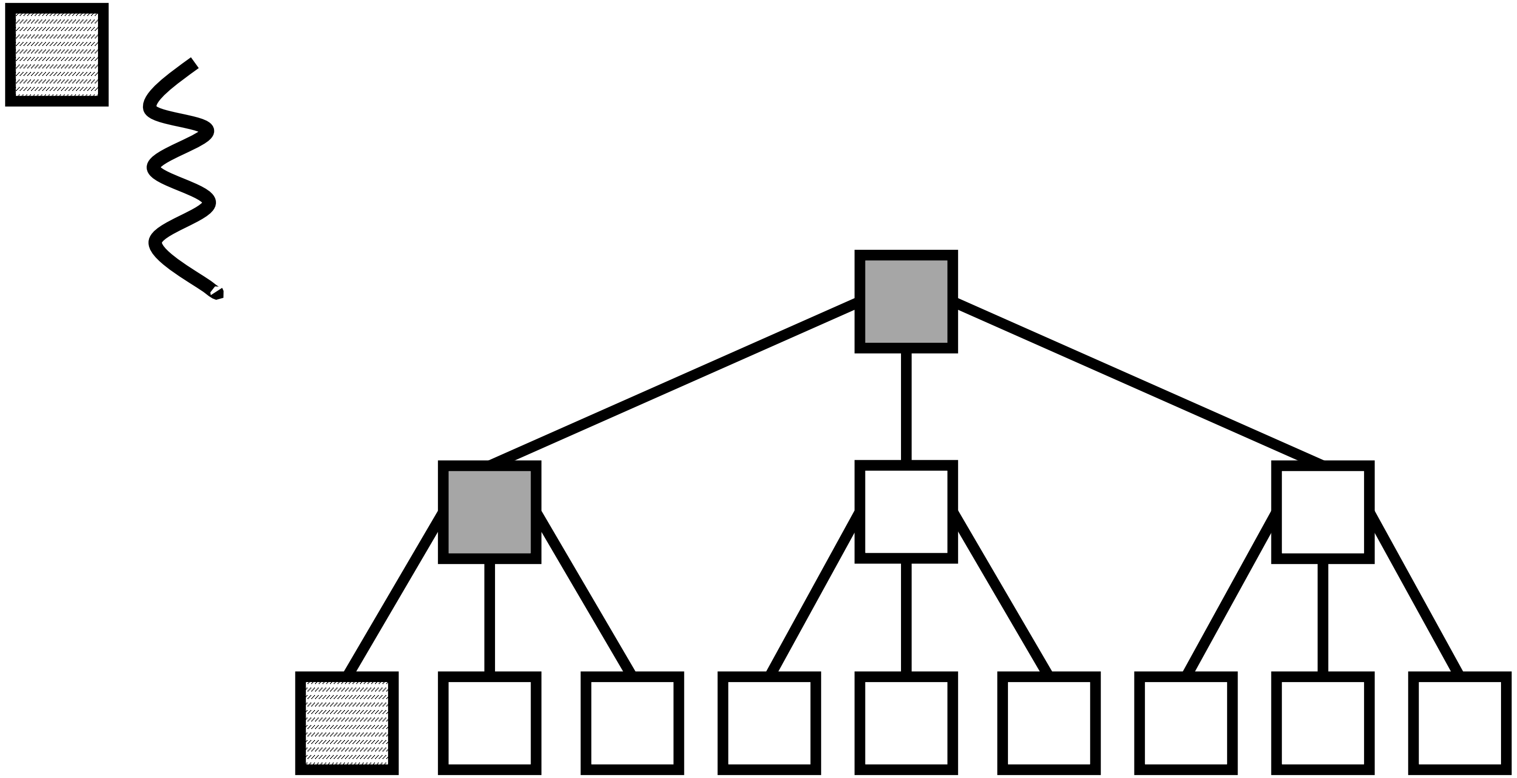
B$^+$tree example



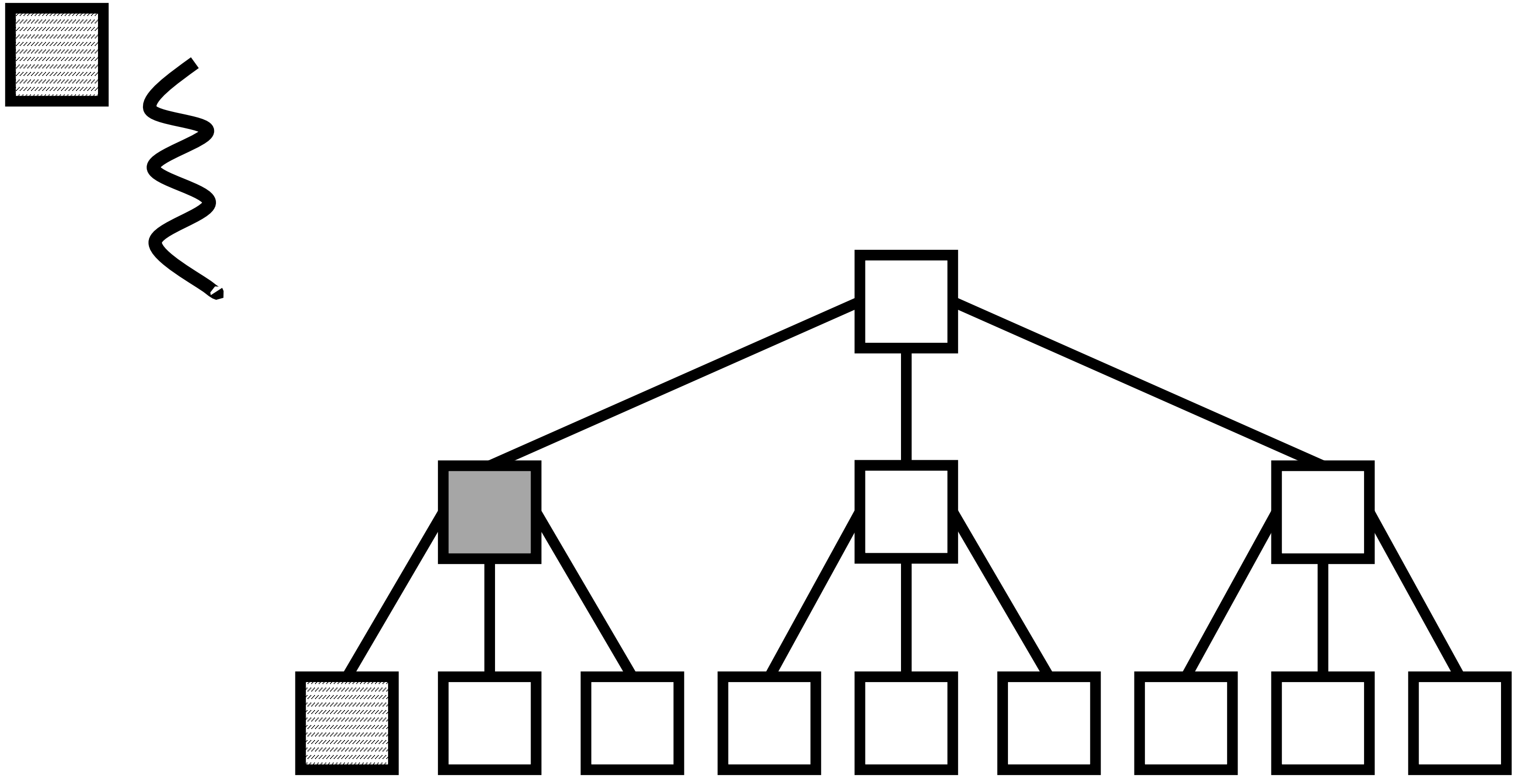
B+tree example



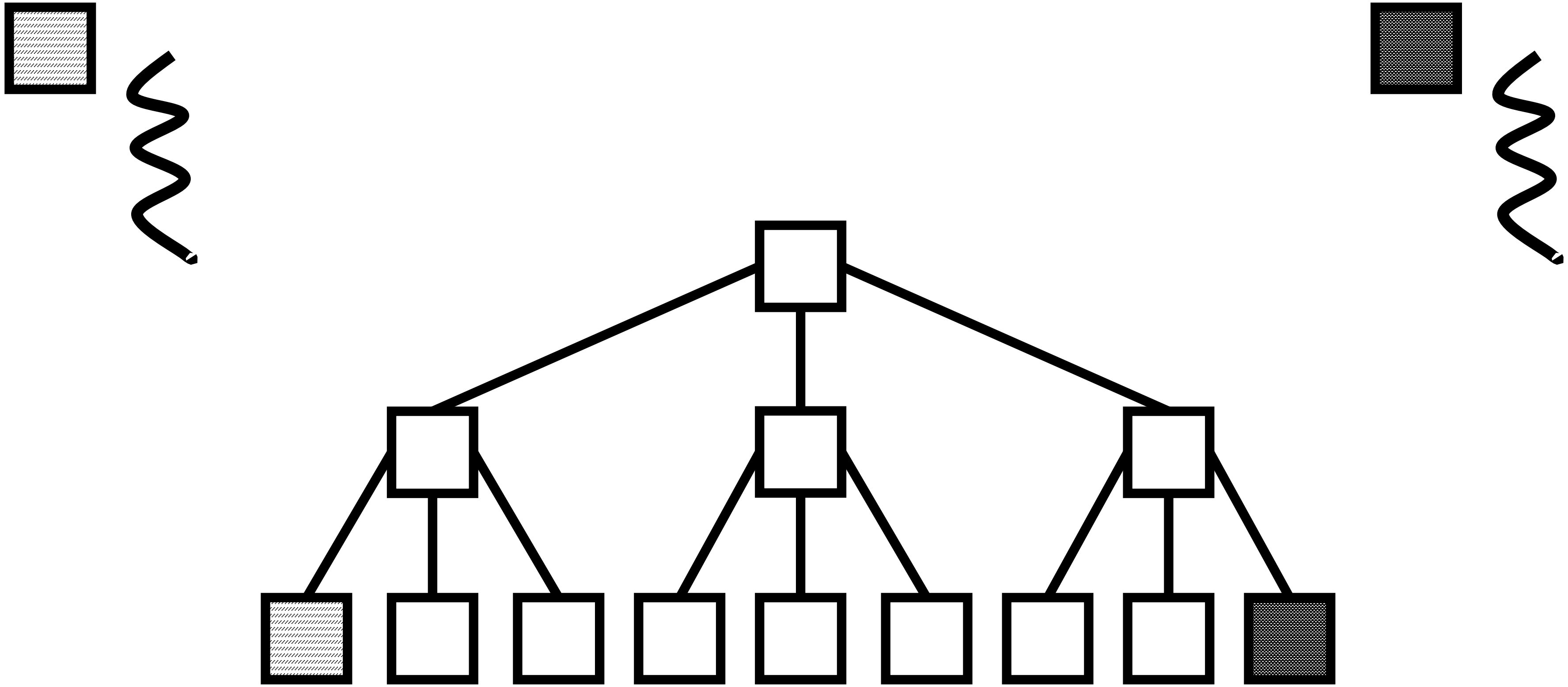
B^+tree example



B^+tree example

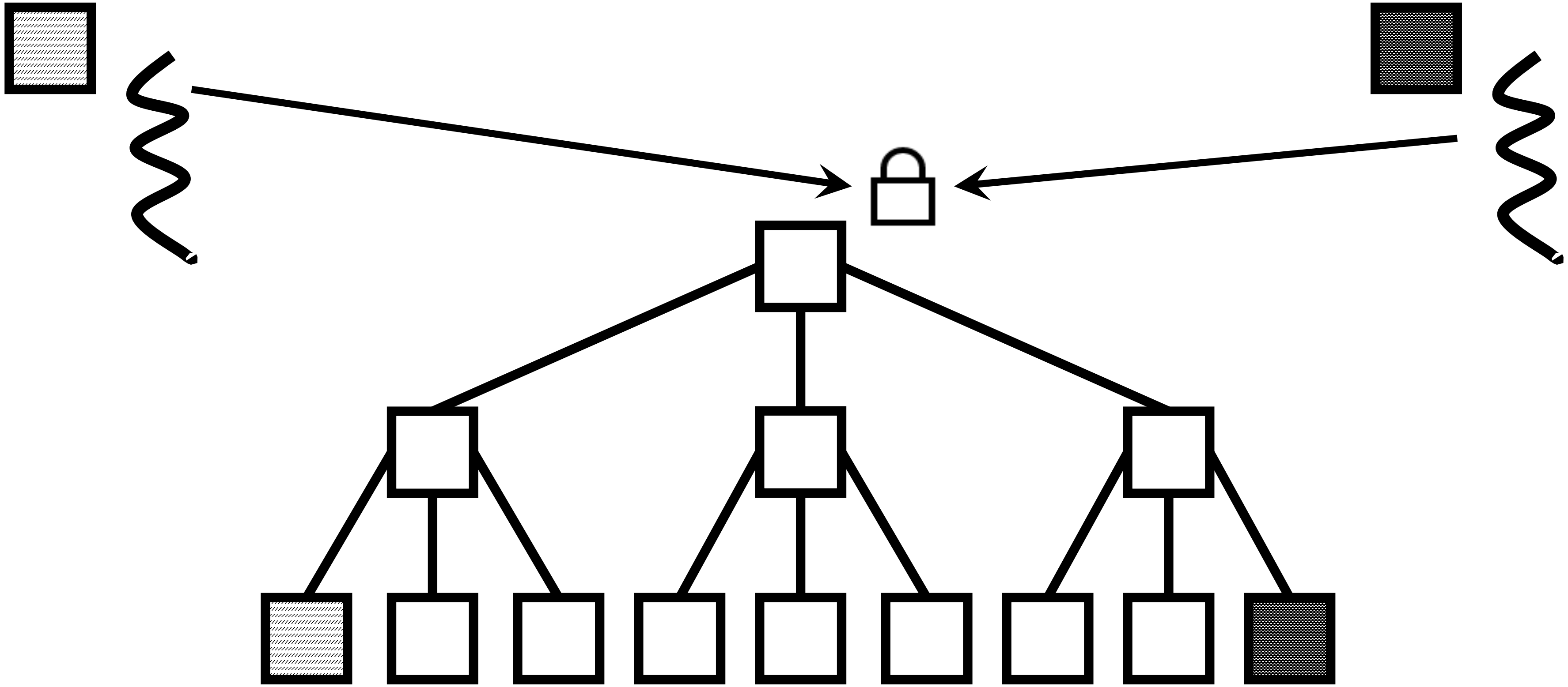


B$^+$tree example



Two threads make non-conflicting accesses

B⁺tree example



Both threads must update the same meta-data

Scalable B^+trees

- High level goal: Avoid frequent synchronization on the root

- Multi-core specific algorithms — OLFIT, Bw-tree
  - OLFIT, latch-based [Cha et al. VLDB 2001]
  - Bw-tree, latch-free [Levandoski et al. ICDE 2013]
  - Both avoid synchronization on tree descent

What **not** to take away from this talk

- Stop designing latch-free algorithms

- Latch-based algorithms will always perform just as well

- Scalable latches will solve all your problems

What to take away

- “Latch-free” is not a synonym for scalable
  - It’s not a synonym for synchronization-free either

- Latch-free and latching algorithms are subject to similar issues under contention

- Focus on how to play well with hardware
  - Keep the performance of concurrent reads and writes in mind
  - Better indicator of scalability than “latch-free” or “latch-based”

- Latch-free algorithms’ theoretical guarantees are mostly irrelevant
  - Assigning requests to OS threads is not fundamental
  - Multiplex requests on a fixed set of threads