Synchronization Primitives: Locks and Barriers

Srđan Milaković

03/26/2019

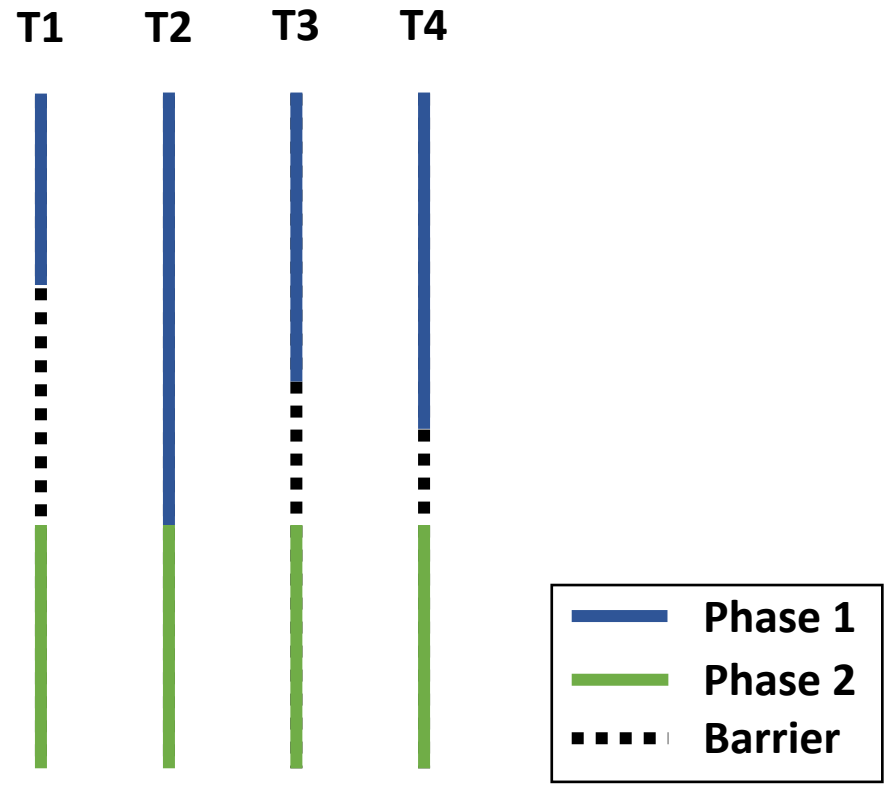
COMP 522

Synchronization Policies

- Blocking – deschedule waiting processes
- Busy-wait – repeatedly test shared variables
  - Scheduling overhead is greater than wait time
  - Processors resources are not needed for other tasks
  - Scheduler-based blocking is inappropriate or impossible

Spin Locks and Barriers

- Spin locks
  - Mutual exclusion
- Barriers
  - No processes advance beyond a particular point in computation until all have arrived at that point
  - Typically used to separate “phases” of an application program



Spin Locks and Barriers

- Performance is **very important**
  - Locks protect very small critical sections, and may be executed enormous number of times
- Agarwal and Cherian investigation
  - Synchronization accounted for as much as 49% of total network traffic
- Busy-waiting on a single synchronization variable
  - Why is this a problem?
- A lot of work for specialized harware

Atomic Operations

- Early algorithms used used only atomic reads and writes
  - E.g. Peterson's Algorithm
  - Costly in time and space – a lot of shared variables and a large number of operations used for coordination
- Modern processors support more sophisticated atomic operations
  - `fetch_and_φ` – Read-Modify-Write (RMW)
  - `test_and_set`, `fetch_and_store` (swap or exchange), `fetch_and_add`, `compare_and_swap`

Atomic Operations

- Modern processors support more sophisticated atomic operations
  - `fetch_and_Φ` – Read-Modify-Write (RMW)
    - `test_and_set()`
    - `fetch_and_store(T desired)`
    - `fetch_and_add(T arg)`
    - `fetch_and_increment() ≡ fetch_and_add(1)`
    - `compare_and_swap(T expected, T desired)`
- Load-link/store-conditional
  - fetch & square
  - ARM, RISC-V

Outline

- Locks
  - `test_and_set` Lock, The Ticket Lock, Array-Based Queuing Locks
  - The MCS Lock
  - Malthusian Locks
  - Compact NUMA-aware Locks
- Barriers
  - Centralized barriers, The software combining tree barrier, Dissemination barrier, and Tournament Barriers
  - A New Tree-Base Barrier

References

1. **Algorithms for scalable synchronization on shared-memory multiprocessors**. John Mellor-Crummey and Michael L. Scott (Feb. 1991)
2. **Malthusian Locks**. Dave Dice (Apr. 2017)
3. **Compact NUMA-Aware Locks**. Dave Dice, Alex Kogan (Oct. 2018)

The Simple `test_and_set` Lock

- The lock object have an atomic Boolean flag

```
1 typedef atomic_bool lock;
```

- Acquire – perform `test_and_set` until you flip the flag from **false** to **true**

```
1 void acquire_lock(lock *L)
2   while test_and_set(L) == true
3     NOP
```

- Release – set the flag to **false**

```
1 void release_lock(lock *L)
2   *L = false
```

The Simple `test_and_set` Lock

- Flag access contention
- `test_and_set` is relatively expensive
  - Particularly expensive on cache-coherent MPs
- Test-and-`test_and_set`
- Adding delay between consecutive probes of the lock
  - Exponential backoff

```
1 void acquire_lock(lock *L)
2   while true
3     while load(L) == true
4       NOP
5     if test_and_set(L) == false
6       break
```

```
1 void acquire_lock(lock *L)
2   delay = 1
3   while test_and_set(L) == true
4     pause(delay)
5     delay *= 2
```

The Ticket Lock

- Test-and-`test_and_set` – one RMW per waiting processor whenever locks becomes available
- The ticket lock – one RMW per lock acquisition
  - Lock acquisition happens in FIFO order – no starvation

The Ticket Lock

- The lock object have two counters
  - Next ticket – the number of requests to acquire the lock
  - Now serving – the number of times the lock has been released
- The counters
  - are initialized to 0
  - should be large enough to accommodate the maximum number of simultaneous requests for the lock

```
1 typedef struct lock
2   atomic_uint next_ticket = 0
3   atomic_uint now_serving = 0
```

The Ticket Lock

- Acquire – perform `fetch_and_increment` on the next ticket counter and busy wait until and wait until the the result (its ticket) is equal to the value of the now serving counter
- Release – increment the value of the now serving counter

```
1 void acquire_lock(lock *L)
2   my_ticket = fetch_and_increment(&L->next_ticket)
3   while load(&L->now_serving) != my_ticket
4     NOP
```

```
1 void release_lock(lock *L)
2   increment(&L->now_serving)
```

The Ticket Lock

- Still a lot of contention due to loads
- Add delay like in test-and-`test_and_set`
  - Exponential backoff?
    - **NO!**
  - Linear backoff based on how many processors are before me

```
1 void acquire_lock(lock *L)
2   my_ticket = fetch_and_increment(&L->next_ticket)
3   while true
4     pause(my_ticket - L->now_serving)
3     if load(&L->now_serving) == my_ticket
4       break
```

Array-Based Queuing Locks

- Ticket lock with proportional backoff requires non-constant number of network transactions
- The idea is to use an atomic operation to obtain the address of a location where to spin
- Array-based queuing locks require space per lock linear in the number of threads
- The maximum number of threads must be known before lock initialization

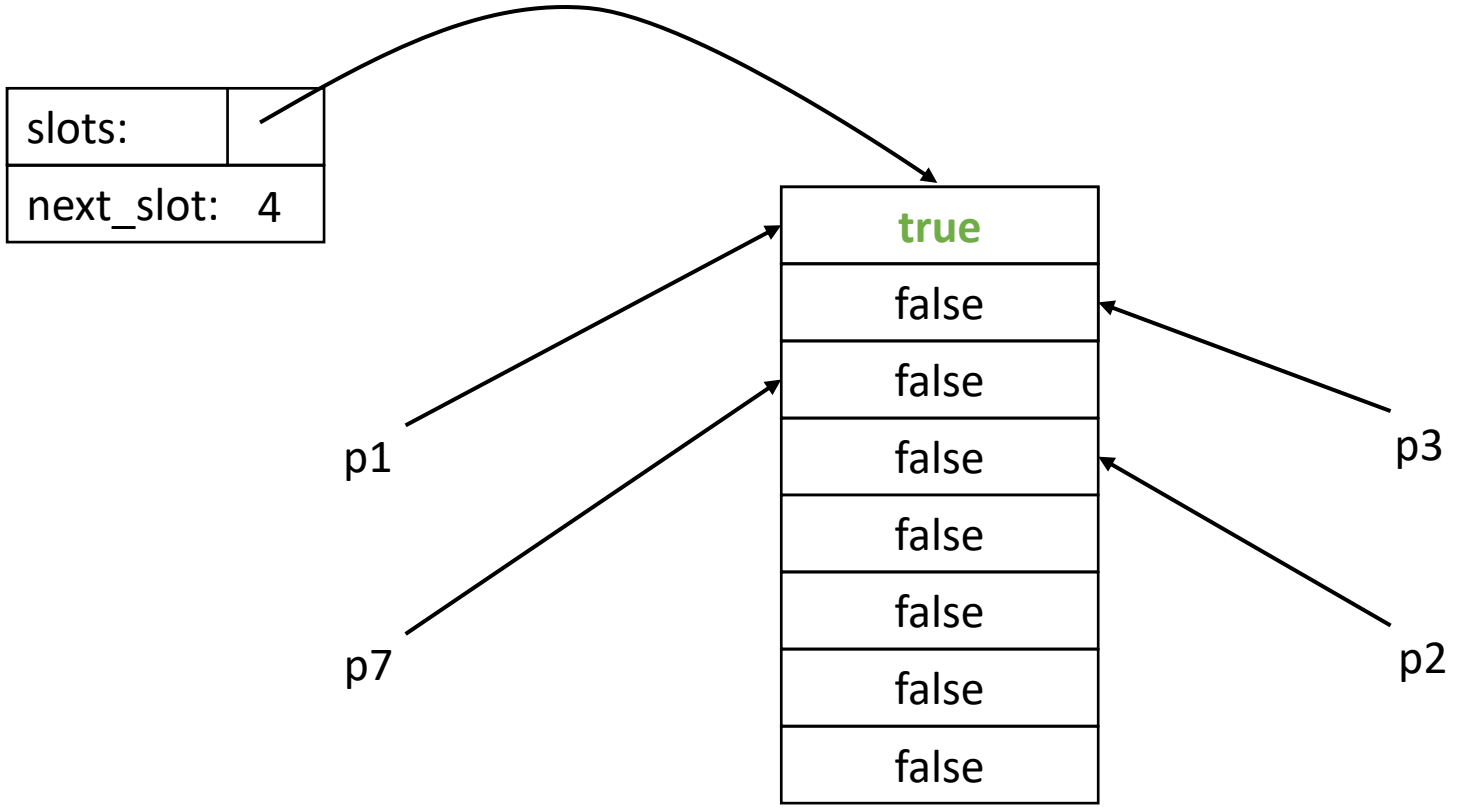
Anderson's Lock

```
1 typedef struct lock
2   atomic_bool slots[numprocs] = {true, false, …, false}
3   atomic_uint next_slot = 0
```

```
1 void acquire_lock(lock *L, uint *my_place)
2   *my_place = fetch_and_increment(&L->next_slot)
3   if *my_place mod numprocs == 0
4     atomic_add(&L->next_slot, -numprocs)
5   *my_place = *my_place mod numprocs
6   while load(&L->slots[*my_place]) == false
7     NOP
```

```
1 void release_lock(lock *L, uint *my_place)
2   L->slots[*my_place] = false
3   L->slots[(*my_place + 1) mod numprocs] =  true
```

Anderson's Lock



Graunke and Thakkar's Lock

```
1 typedef struct lock
2   atomic_bool slots[numprocs] = {true, true, …, true}
3   typedef atomic struct tail_t
4      atomic_bool *who_was_last = 0
5      this_means_locked = false
6   tail_t tail
7
8 processor private uint vpid // a unique virtual processor index
```

```
1 void acquire_lock(lock *L)
2   (who_is_ahead_of_me, what_is_locked) =
3       fetch_and_store(&L->tail, (&L->slots[vpid], L->slots[vpid]))
4   while load(who_is_ahead_of_me) == what_is_locked
5     NOP
```

```
1 void release_lock(lock *L)
2   &L->slots[vpid] = not L->slots[vpid]
```

Array-Based Queuing Locks

- Anderson's lock
  - Requires `fetch_and_increment`
- Graunke and Thakkar's lock
  - Requires `fetch_and_store`

A List-Base Queuing Lock - MCS

- Guarantees FIFO ordering of lock acquisitions[1]
  - The ticket lock✓, array-based queuing locks ✓, test_and_set lock ✗
- Requires O(1) space per lock
  - The ticket lock✓, array-based queuing locks ✗, test_and_set lock ✓
- Spins only on locally-accessible flag variables
- Works equally well on machines with and without cache coherence
  - Unique to the MCS lock ✓ ✓ ✓

[1] requires `compare_and_swap`

The MCS lock

- The lock object is a pointer to a qnode
- qnode has a pointer to a next qnode and a Boolean field locked
- Acquire – perform enqueue operation. If the queue was empty, the lock is acquired, otherwise spin on the locked field
- Release – if the queue is not empty, notify the next processor in the queue by setting the locked field to true

```
1 typedef struct qnode
2   qnode *next
3   atomic_bool locked
4
5 typedef qnode *lock
```

The MCS lock – acquire lock

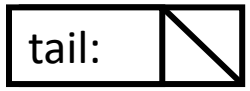
```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```

The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

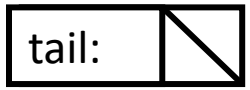
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

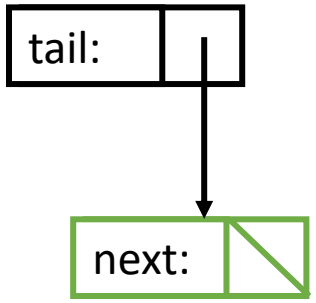
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

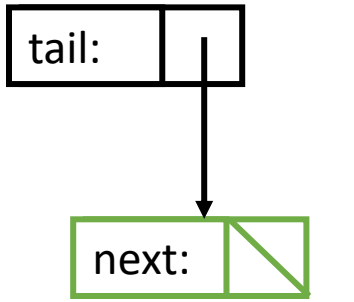
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```

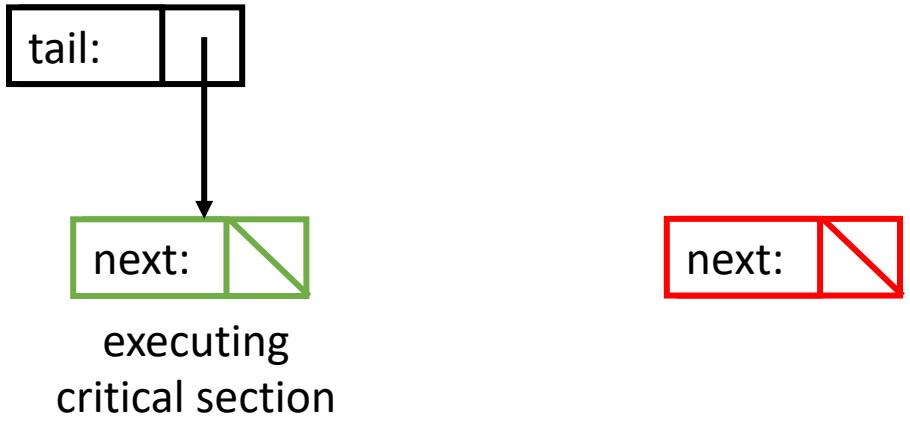


executing
critical section

The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

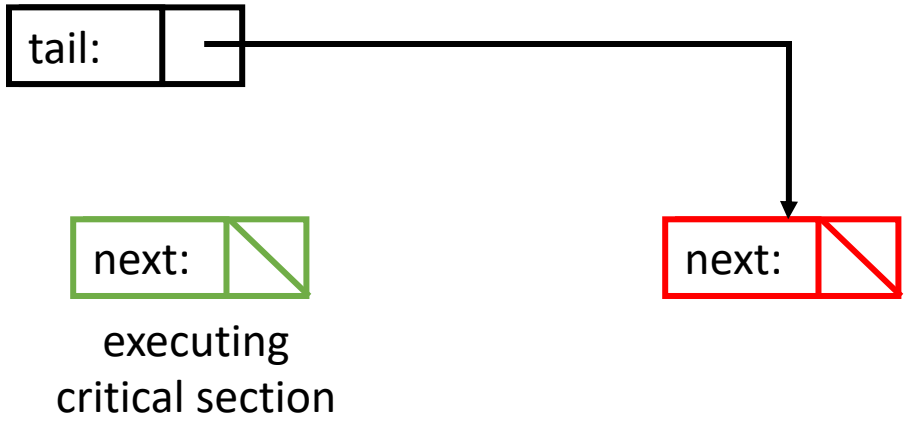
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

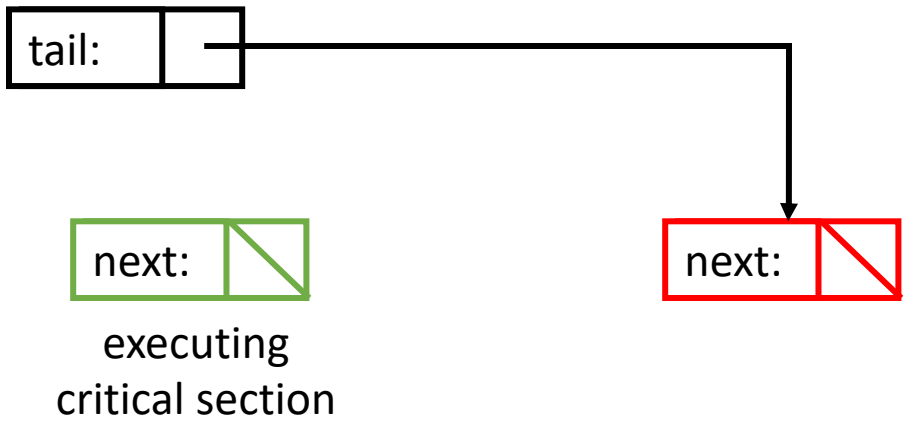
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

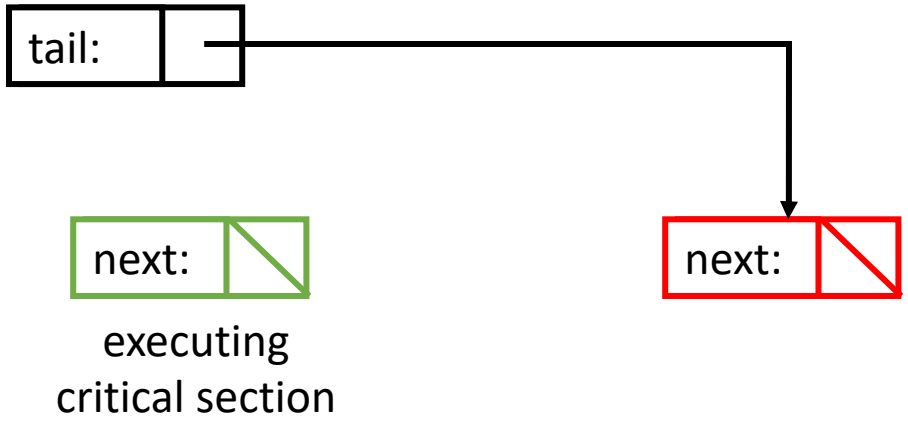
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

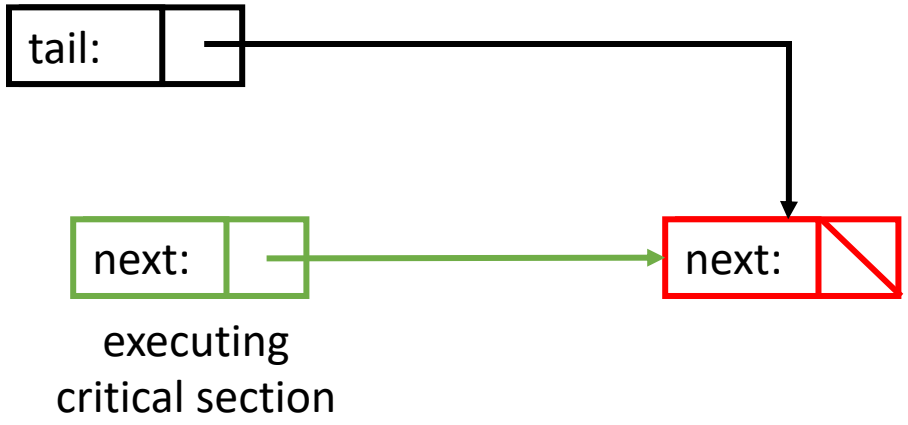
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

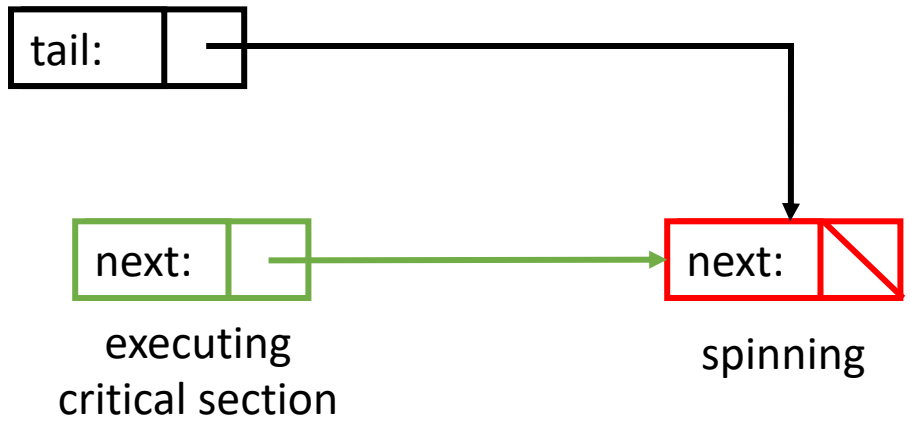
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

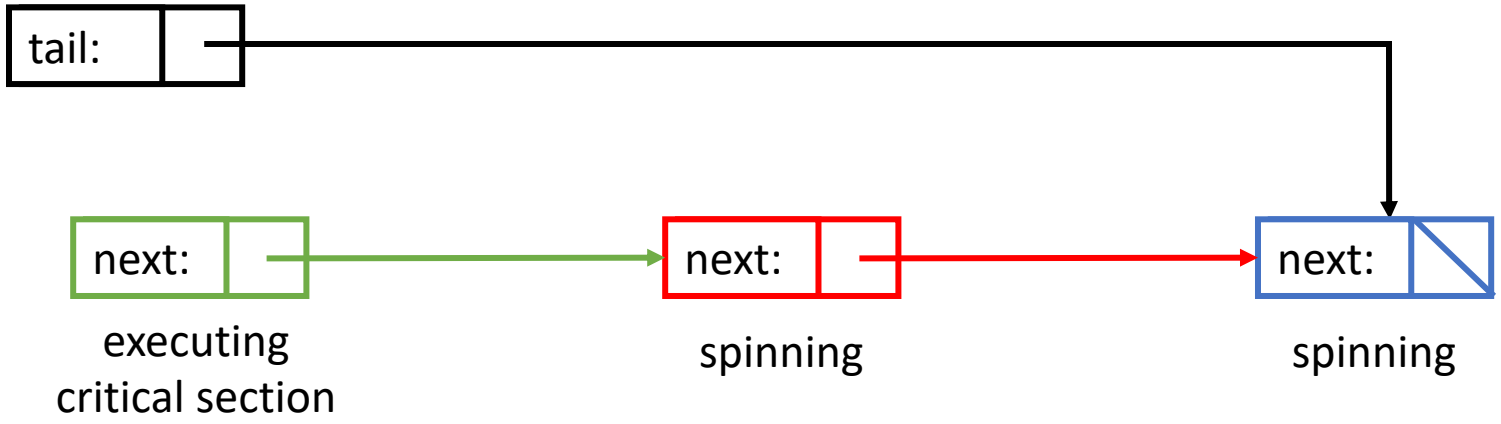
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

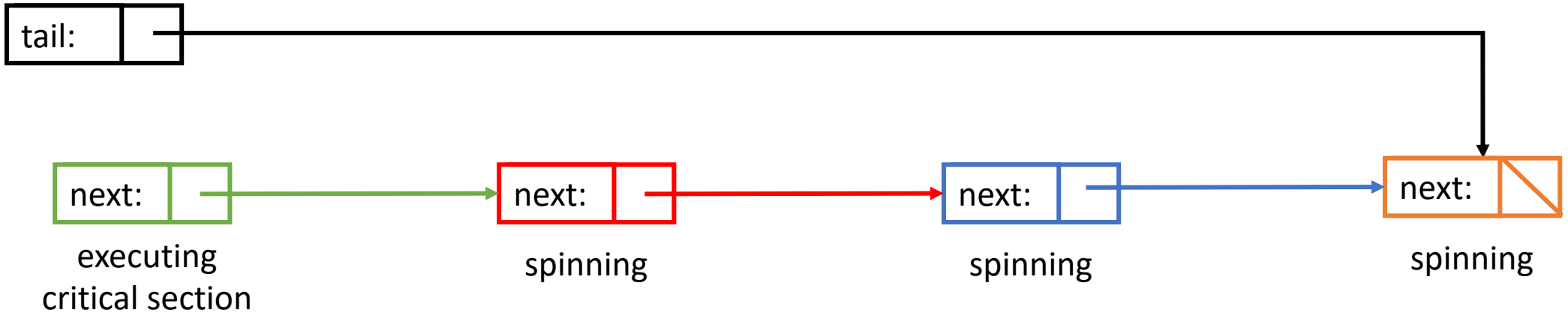
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – acquire lock

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

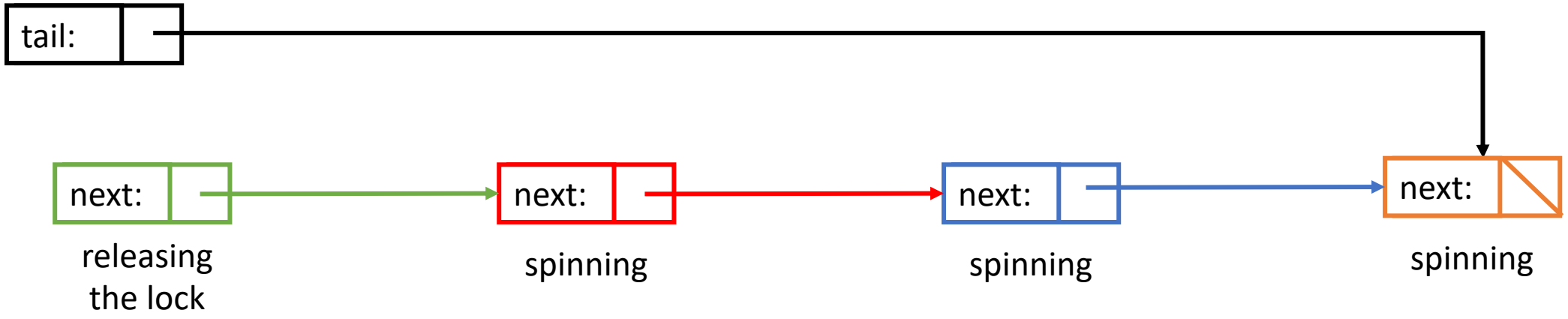
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 1

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

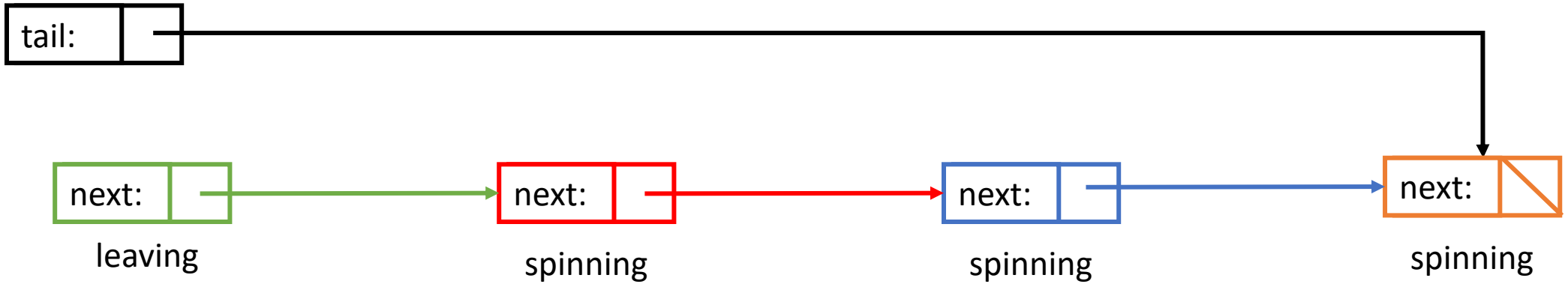
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 1

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

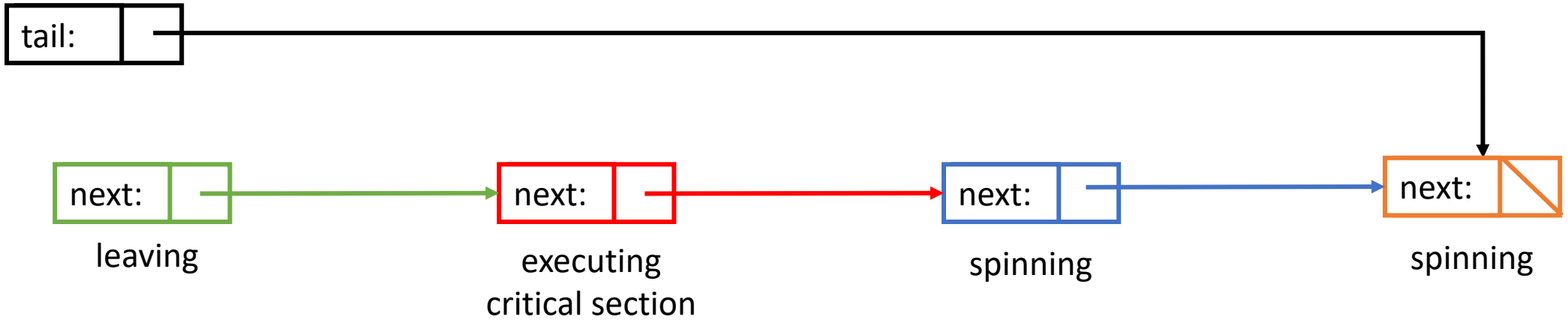
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 1

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

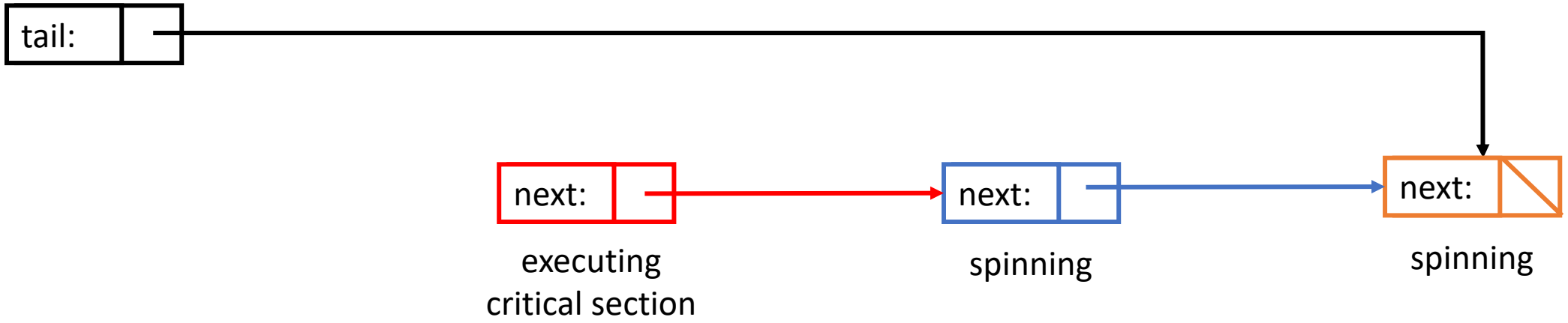
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 1

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 2

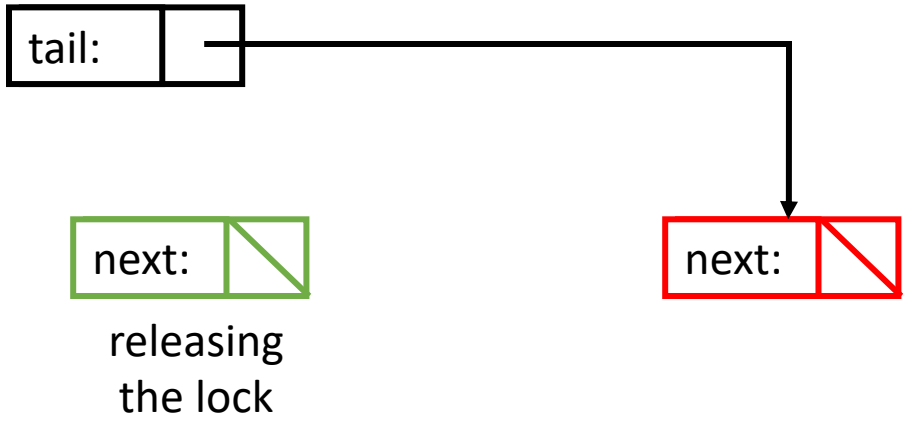
```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```

The MCS lock – release lock – case 2

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

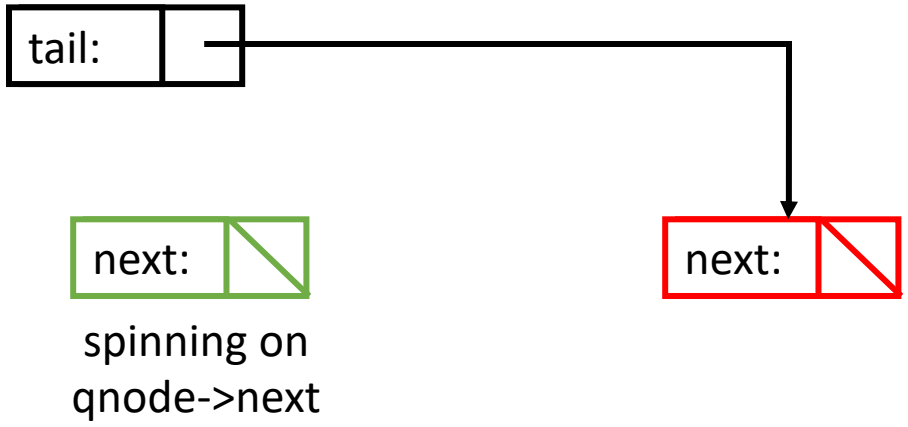
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null) ✗
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 2

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

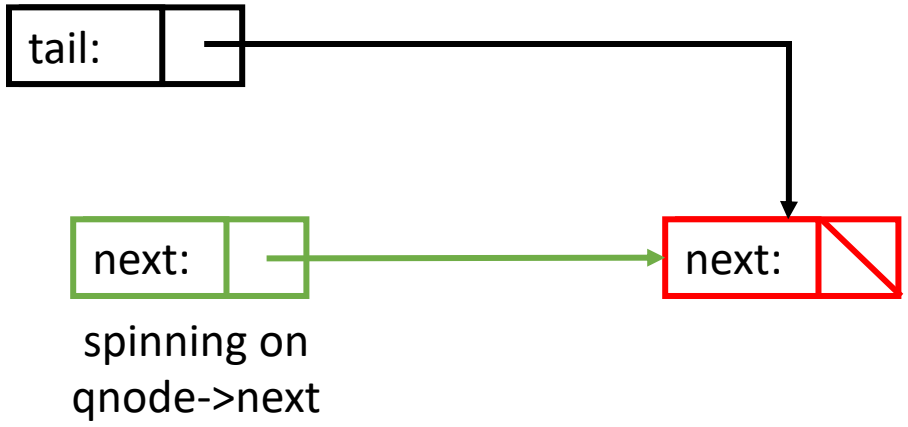
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 2

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

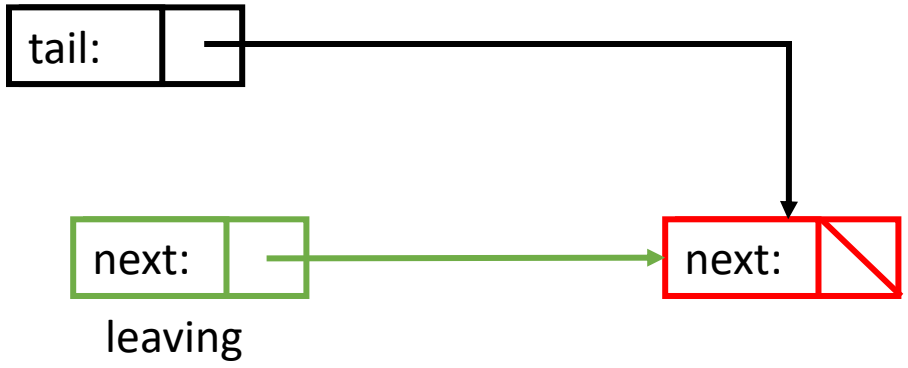
```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```



The MCS lock – release lock – case 2

```
1 void acquire_lock(lock *L, qnode *I)
2   I->next = null
3   predecessor = fetch_and_set(L, I)
4   if predecessor != null
5     I->locked = true
6     predecessor->next = I
7     while I->locked == true
8       NOP
```

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```

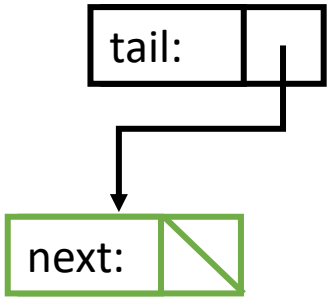


The MCS lock – release without CAS

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     old_tail = fetch_and_store(L, null)
4     if old_tail == null
5       return
6     usurper = fetch_and_store(L, null)
7     while I->next == null
8       NOP
9     if usurper != null
10      usurper->next = I->next
11    else
12       I->next->locked = false
13  else
14    I->next->locked = false
```

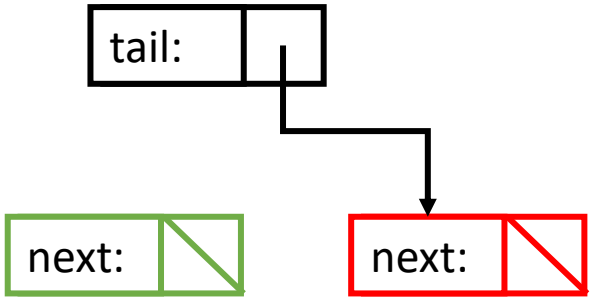
The MCS lock – release without CAS

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     old_tail = fetch_and_store(L, null)
4     if old_tail == null
5       return
6     usurper = fetch_and_store(L, old_tail)
7     while I->next == null
8       NOP
9     if usurper != null
10      usurper->next = I->next
11    else
12       I->next->locked = false
13  else
14     I->next->locked = false
```



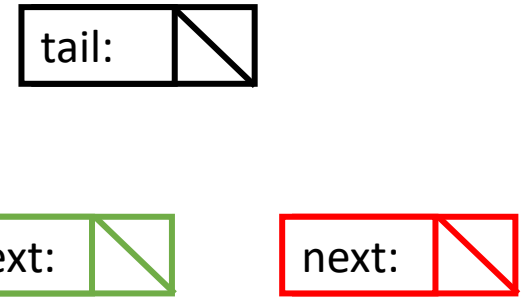
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



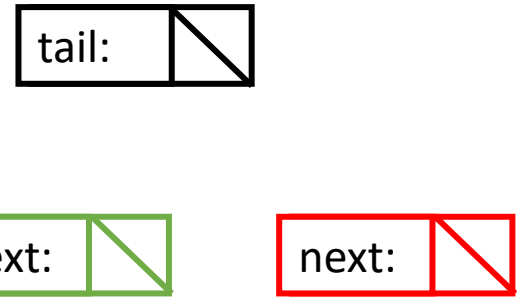
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



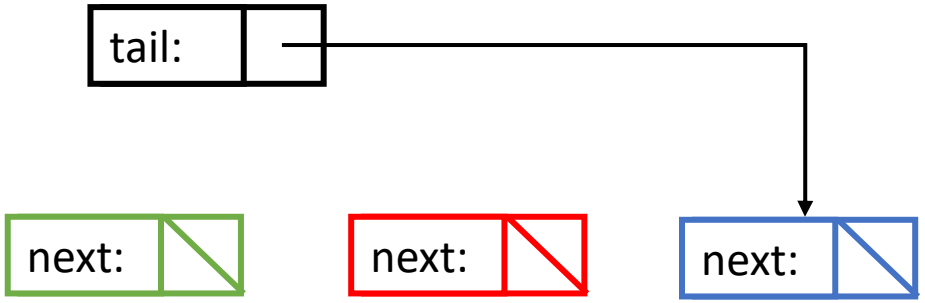
The MCS lock – release without CAS

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     old_tail = fetch_and_store(L, null)
4     if old_tail == null
5       return
6     usurper = fetch_and_store(L, old_tail)
7     while I->next == null
8       NOP
9     if usurper != null
10      usurper->next = I->next
11    else
12       I->next->locked = false
13  else
14    I->next->locked = false
```



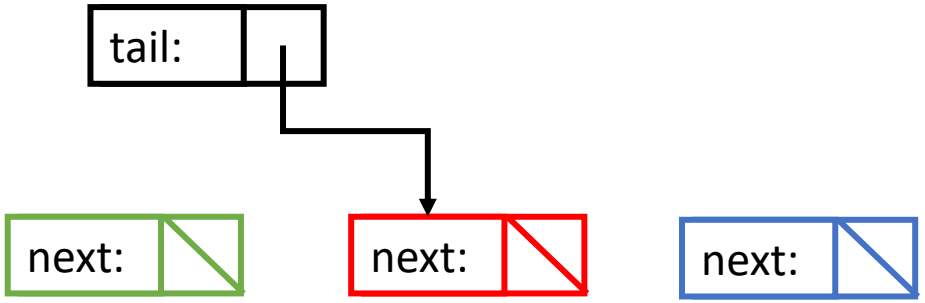
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



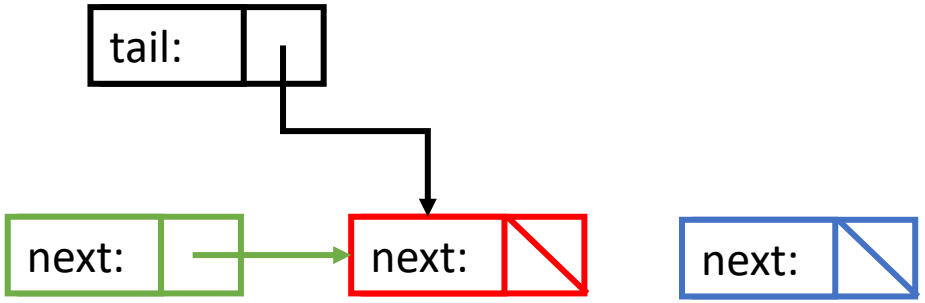
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



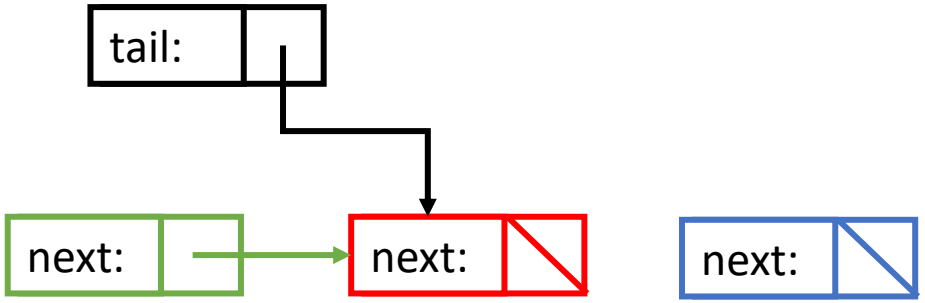
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



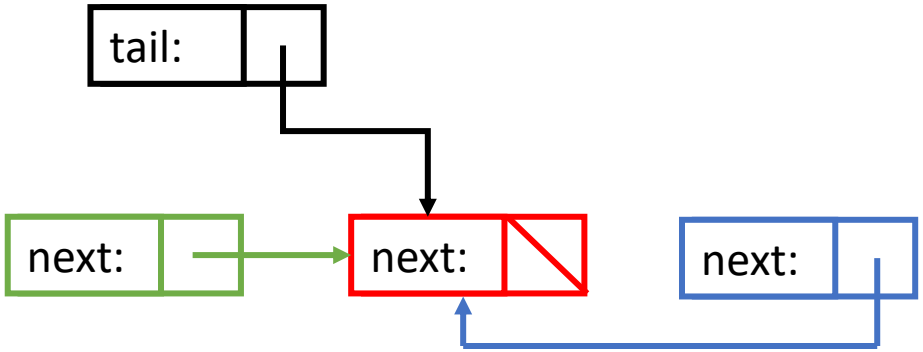
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



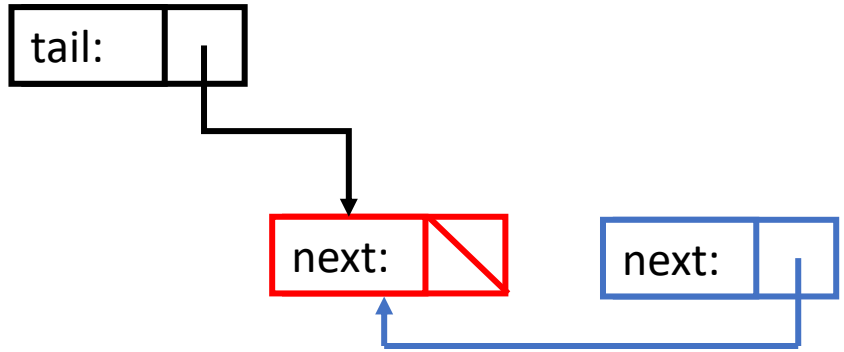
The MCS lock – release without CAS

```
1  void release_lock(lock *L, qnode *I)
2    if I->next == null
3      old_tail = fetch_and_store(L, null)
4      if old_tail == null
5        return
6      usurper = fetch_and_store(L, old_tail)
7      while I->next == null
8        NOP
9      if usurper != null
10       usurper->next = I->next
11     else
12       I->next->locked = false
13   else
14     I->next->locked = false
```



The MCS lock – release without CAS

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     old_tail = fetch_and_store(L, null)
4     if old_tail == null
5       return
6     usurper = fetch_and_store(L, old_tail)
7     while I->next == null
8       NOP
9     if usurper != null
10      usurper->next = I->next
11    else
12       I->next->locked = false
13  else
14     I->next->locked = false
```



Performance – Hardware description

- BBN Butterfly 1 – a distributed shared memory multiprocessor
- Sequent Symmetry Model B – a cache coherent shared-bus multiprocessor

BBN Butterfly 1

- Shared-memory multiprocessor
- Up to 256 nodes
  - 8MHz and 1-4 MB
- Each processor has local memory
- Access to remote memory goes through $\log_4$-depth switching network
- Remote memory read takes 5 µs (no contention) which is roughly 5x compared to local read

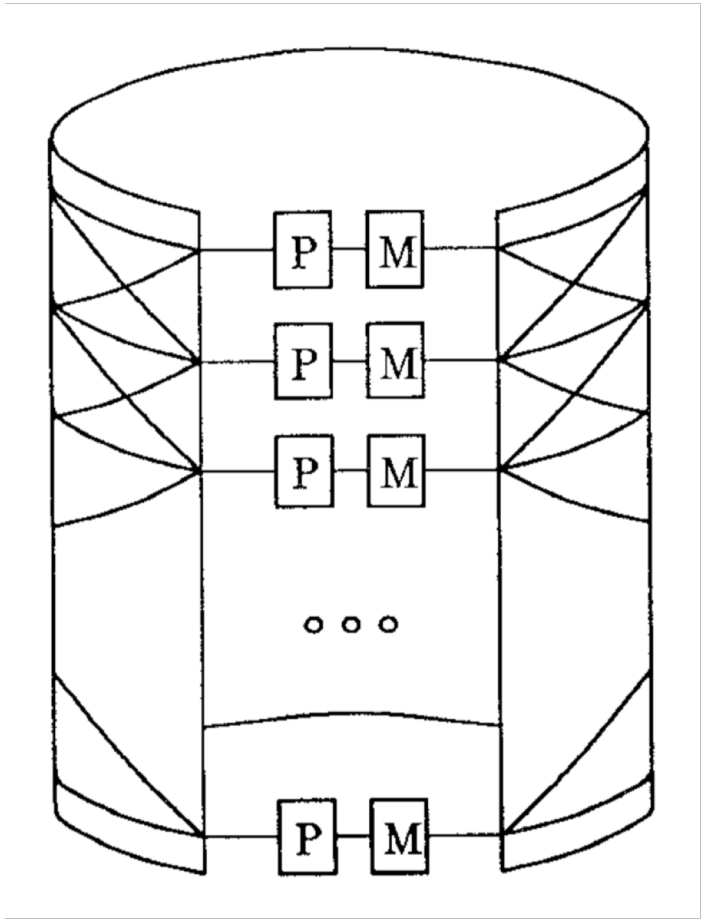


Figure credit: [1]

BBN Butterfly 1 – atomic operations

- Two operations:
  - `fetch_and_clear_then_add`
  - `fetch_and_clear_then_xor`
- Three arguments:
  - dst – the address of the 16-bit destination operand
  - mask – 16-bit mask
  - src – 16-bit source operand
- `*dst = (*dst` **`AND`** `!mask) Φ src`
- Used to implement `fetch_and_store, fetch_and_add`, …

The Sequent Symmetry Model B

- Shared-bus multiprocessor
- Up to 30 processor nodes
- 16 MHz Intel 80386 and 64 KB two-way set associative cache

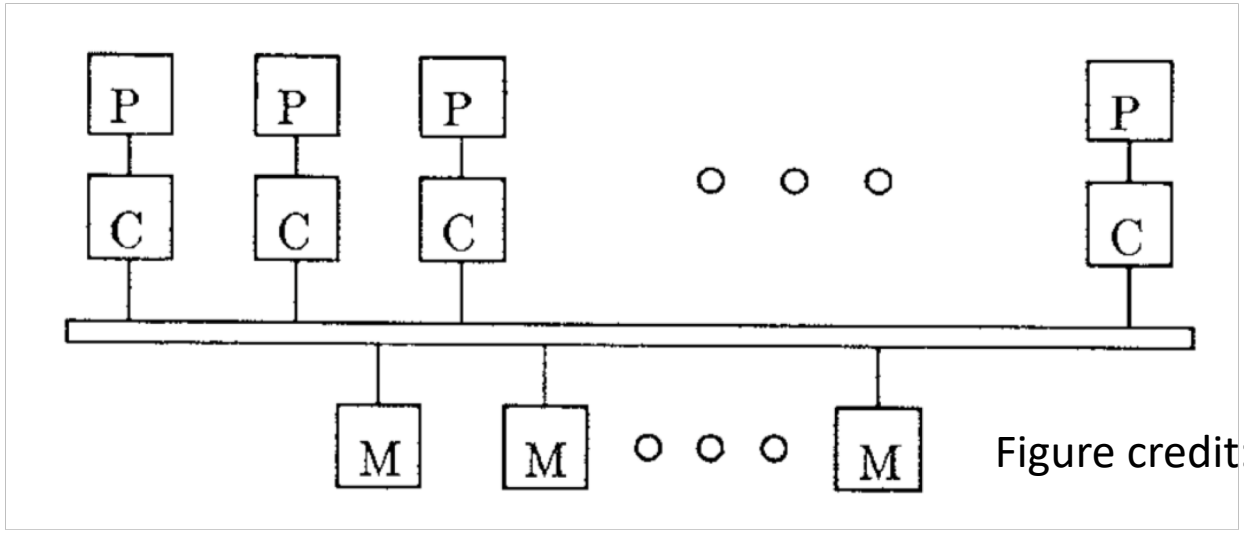


Figure credit: [1]

The Sequent Symmetry Model B

- Supported atomic operations:
  - `fetch_and_store`
  - various logical and arithmetic operations
- Can be applied to 1, 2, or 4 byte quantity
- The logical and arithmetic operations do not return the previous value
  - less useful compared to `fetch_and_Φ`
- No support for `compare_and_swap`

Butterfly – empty critical section

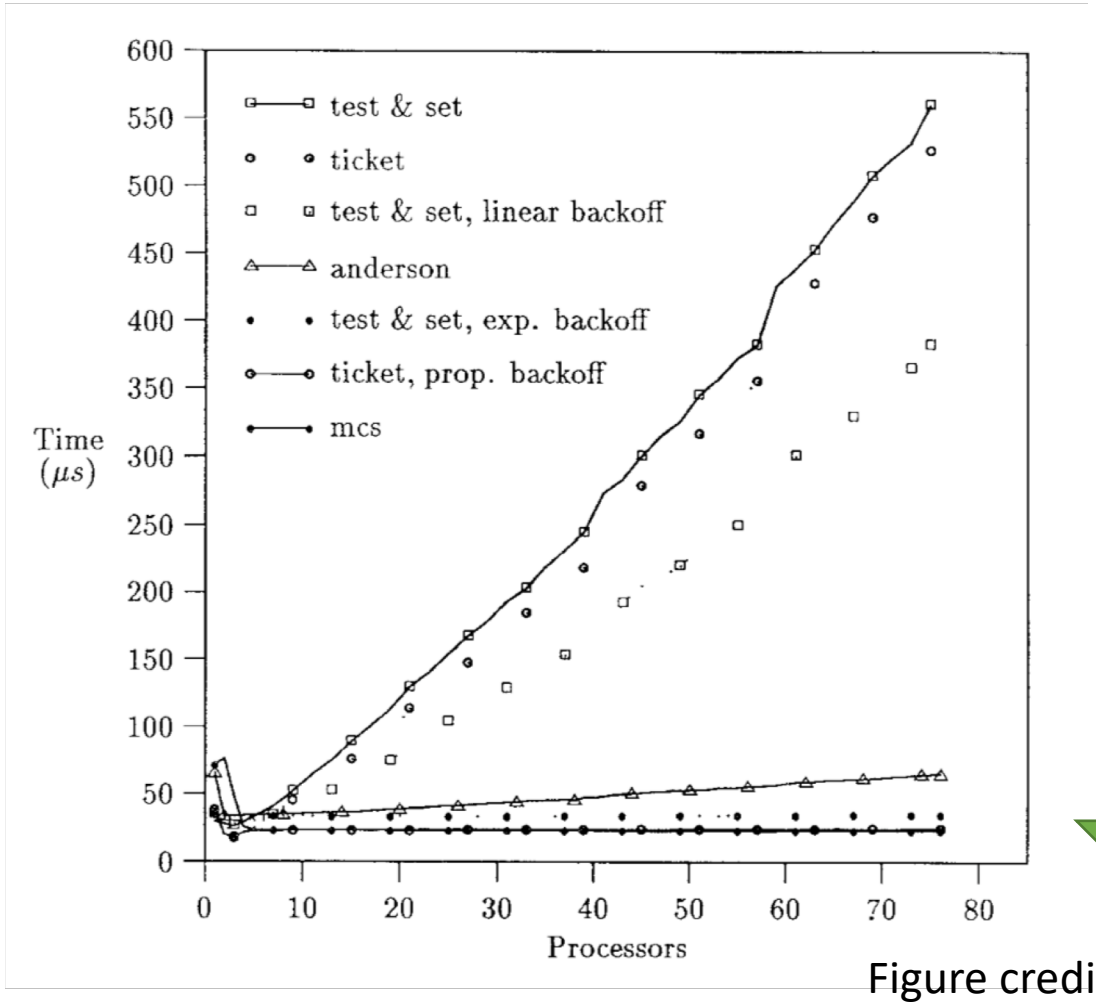


Figure credit: [1]

Butterfly – empty critical section

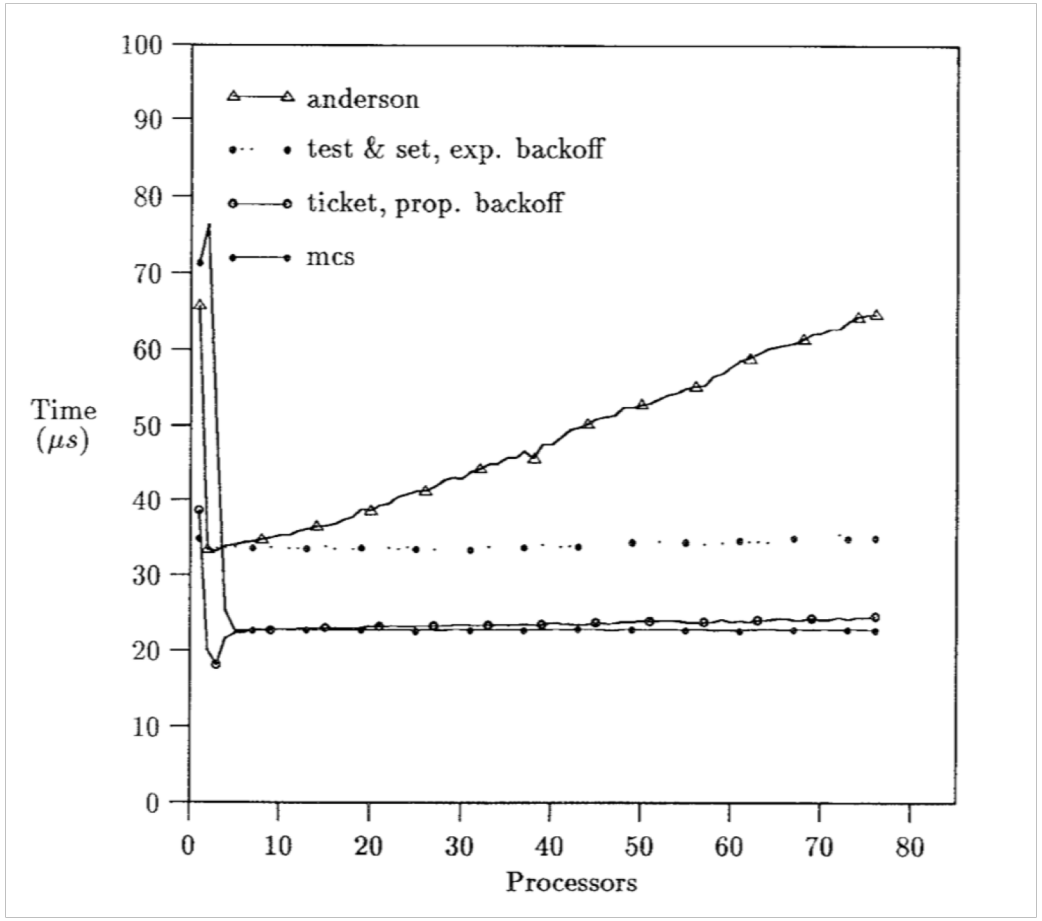


Figure credit: [1]

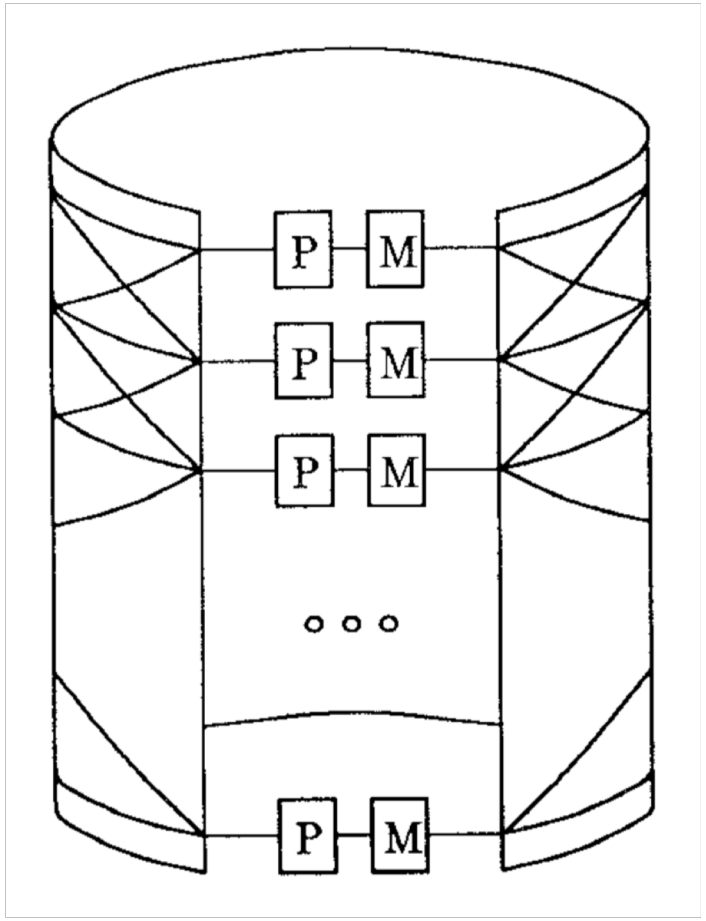


Figure credit: [1]

Butterfly – empty critical section

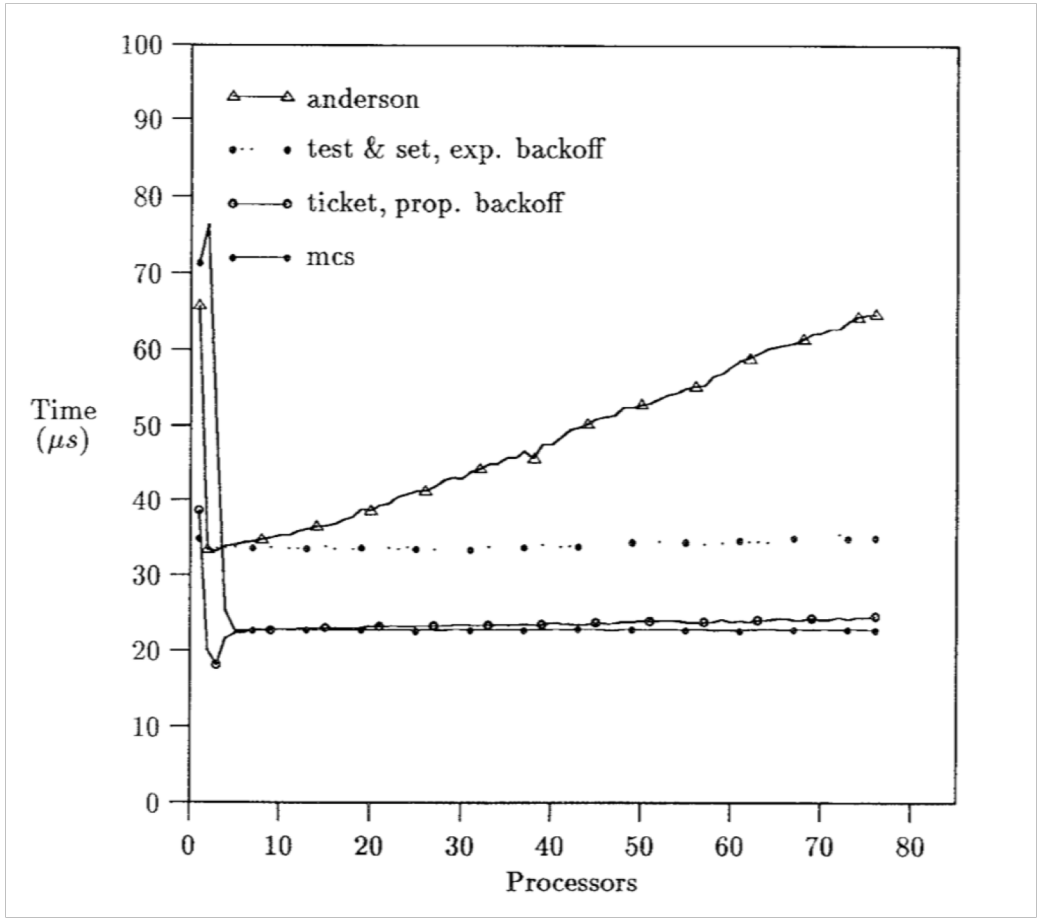


Figure credit: [1]

```
1 void release_lock(lock *L, qnode *I)
2   if I->next == null
3     if compare_and_swap(L, I, null)
4       return
5     while I->next == null
6       NOP
7   I->next->locked = false
```

Symmetry – empty critical section

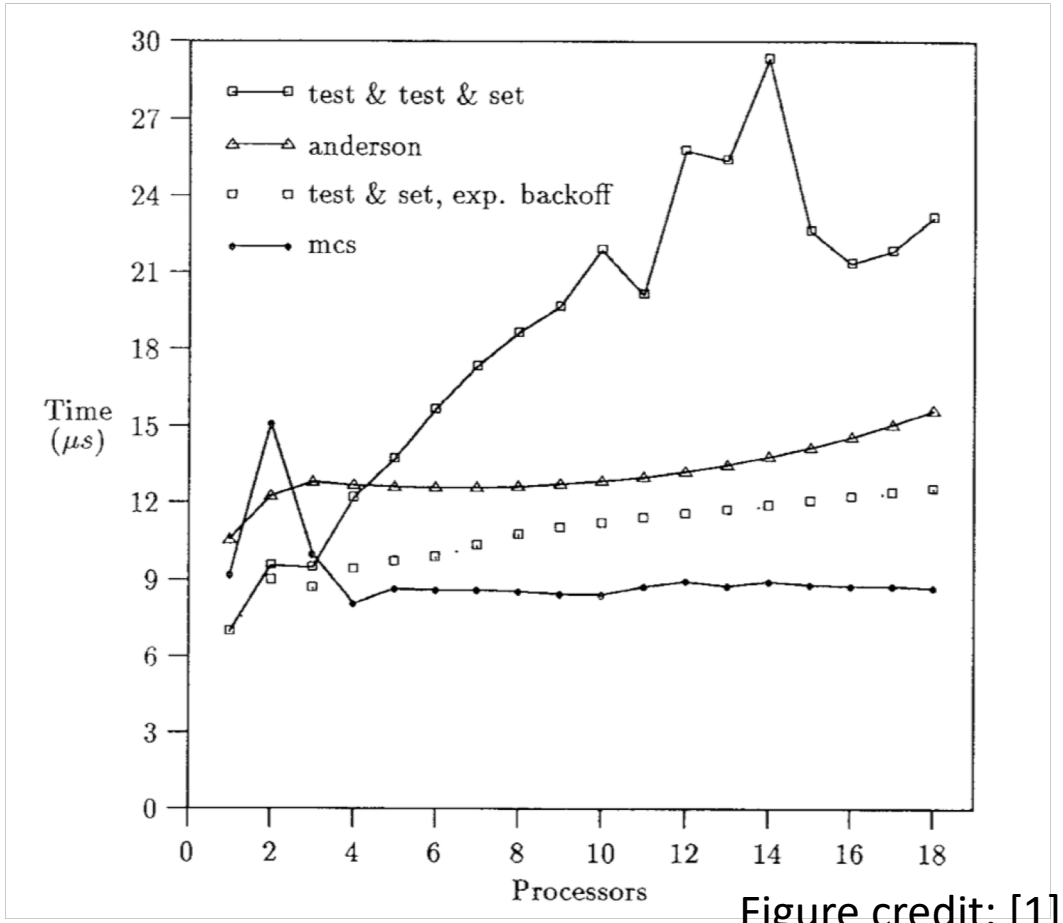


Figure credit: [1]

Symmetry – small critical section

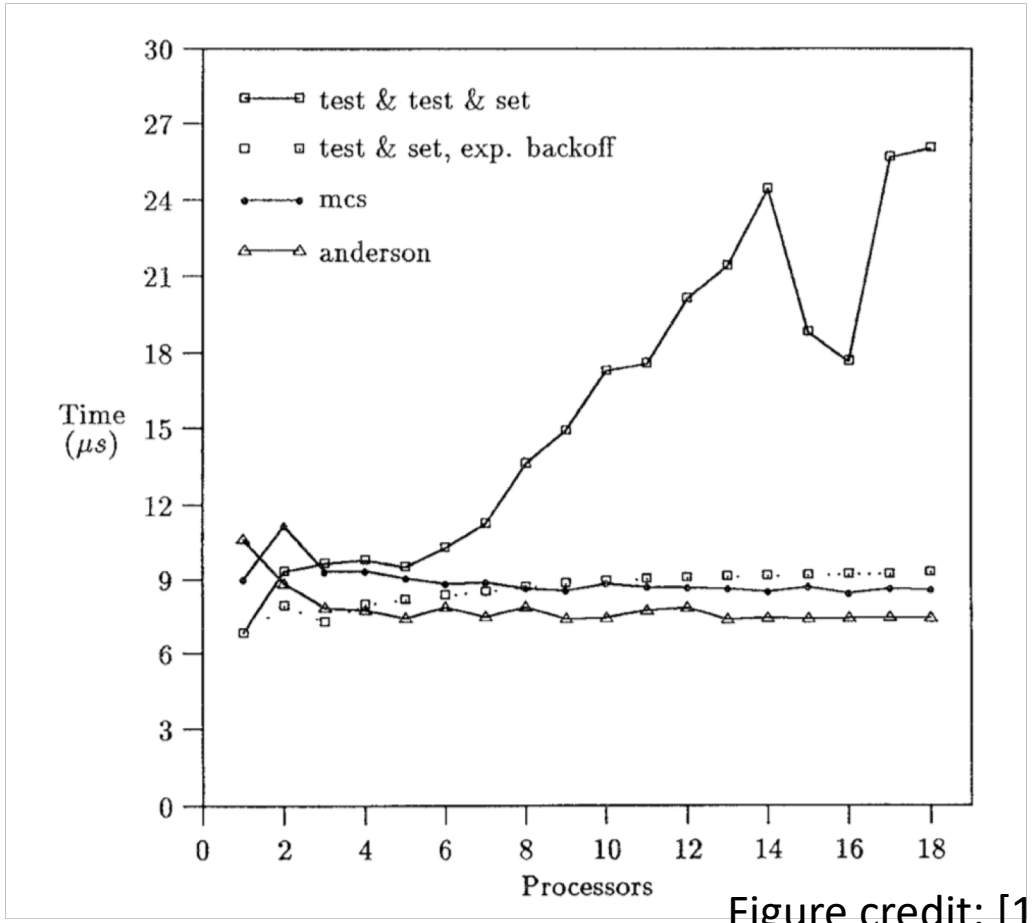


Figure credit: [1]

Malthusian Locks

Dave Dice (April 2017)

Malthusian Locks

- A lot of work was done to improve the performance of lock methods
- Can we improve critical section performance?
- Applications running in modern multithreaded environments are sometimes overthreaded
- The excess of threads does not improve performance
- In fact, it can degrade performance

Malthusian Locks – Motivation

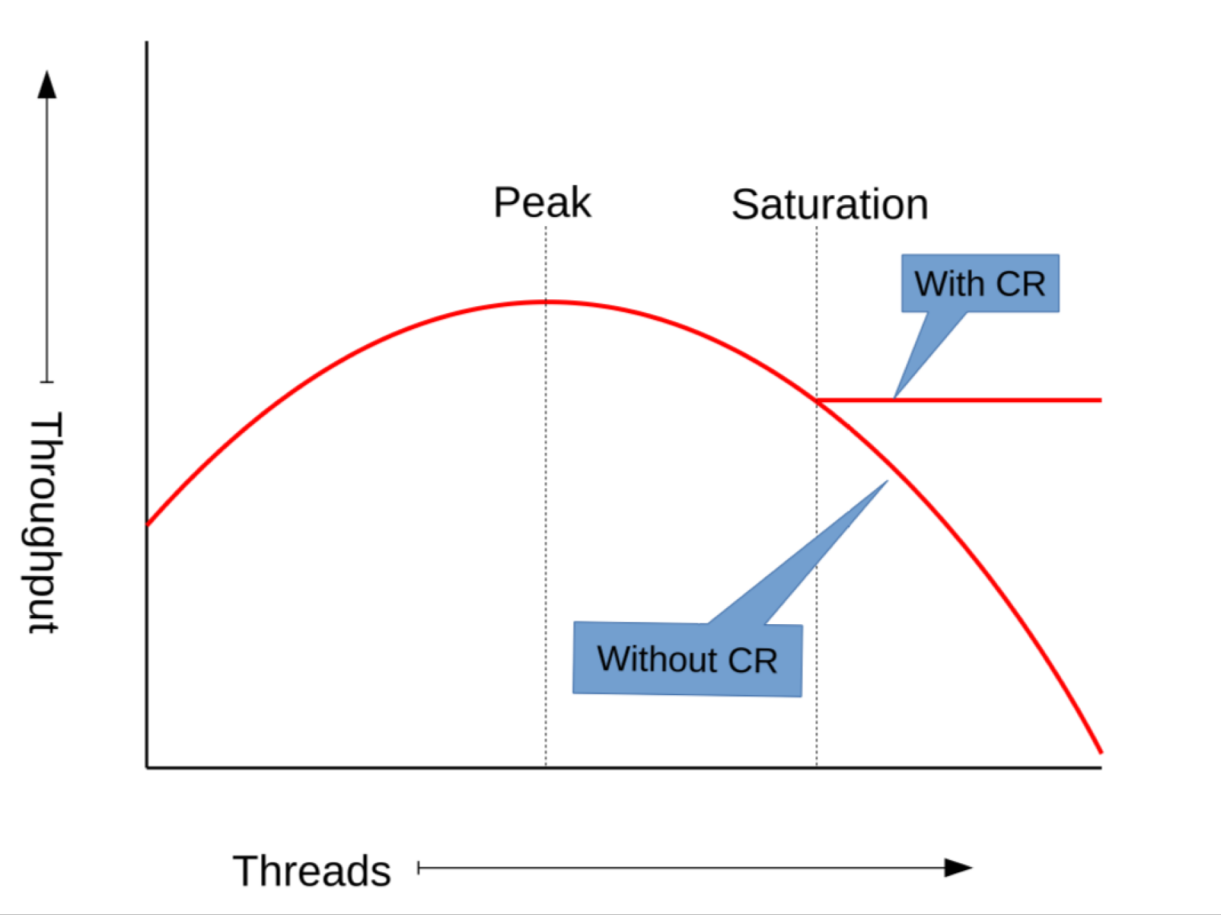


Figure credit: [2]

Malthusian Locks – Motivation

- Single-socket processor
  - 16 cores
  - LLC (L3) is shared and has 8 MB
- Customer database has 1 MB

Malthusian Locks – Motivation

- Duration of the non-critical sections is 4 time longer than the duration of CS
- Memory footprint of NCS is 1 MB
- FIFO lock & 16 threads ⇒ we have 17 MB footprint > 8 MB of LLC
- Threads limited to 5 ⇒ we have 6 MB footprint < 8 MB LLC

Malthusian Locks

- Intentionally limit the number of threads circulating over the lock
- Concurrency restriction (CR)
- The lock acquisition order
  - Unfair during short term
  - Fair over long-term
- Tradeoff fairness and throughput

The MCSCR lock

- Based on the MCS lock
- Two queues
  - Active circulating set (ACS) – enabled threads
  - Passive set (PS) – disabled threads

The MCSCR lock

- ACS should minimal set of threads that saturate lock
- At lock release-time:
  - If there are nodes between the current lock owner and tail, a node from ACS is moved to PS
  - If the ACS is empty, a top node from PS is moved to ACS
- Long-term fairness
  - Periodically move a node from PS to ACS
  - Once every 1000 unlock operations

The MCSCR lock

- The size of ACS is determined automatically
  - No tuning required
- All changes are implemented in the lock release method
  - Effectively, the length of the critical section is increased
  - The lock acquire method is same as in the MCS lock

Waiting policies

- What to do if we don’t have a lock?
- Unbounded spinning
  - Consume pipeline resources and energy
  - Increases and possibly preventing other threads to use turbo mode
  - Polite spinning – PAUSE instruction (or equivalent)
  - Low resume time

Waiting policies

- Parking
  - Voluntary context switching
  - Potentially reducing power consumption and enabling turbo mode
  - Long resume time
- Spin-Then-Park
  - Hybrid approach
  - Limit the maximum spin period to the length of context-switch round trip

Performance - Hardware description

- Oracle SPARC T5-2
  - 2 sockets (1 disabled)
  - 16 cores per socket
  - 8 logical cores
  - **128 logical cores per socket**
  - Cache
    - 16KB private L1 - unified
    - 128KB private L2 - unified
    - **8MB shared L3 - unified**

Performance - Random Access Array

- N concurrent threads
- 10 seconds interval
- Total number of iterations
- NCS – 400 iterations that randomly fetch a value from a thread private array of 256K 32-bit integers
- CS – 100 iterations that randomly fetch a value form a shared array of 256K 32-bit integers
- The ideal speedup is 5x

Performance - Random Access Array

MCS-S – the classical MCS lock with a polite instruction inside spin loop
MCS-STP – MCS lock with spin-then-park wait policy
MCSCS-S – MCSCS lock with a polite instruction inside spin loop
MCSCS-STP – MCSCS lock with spin-than-park wait policy
null – empty lock method

120x throughput for 256 threads

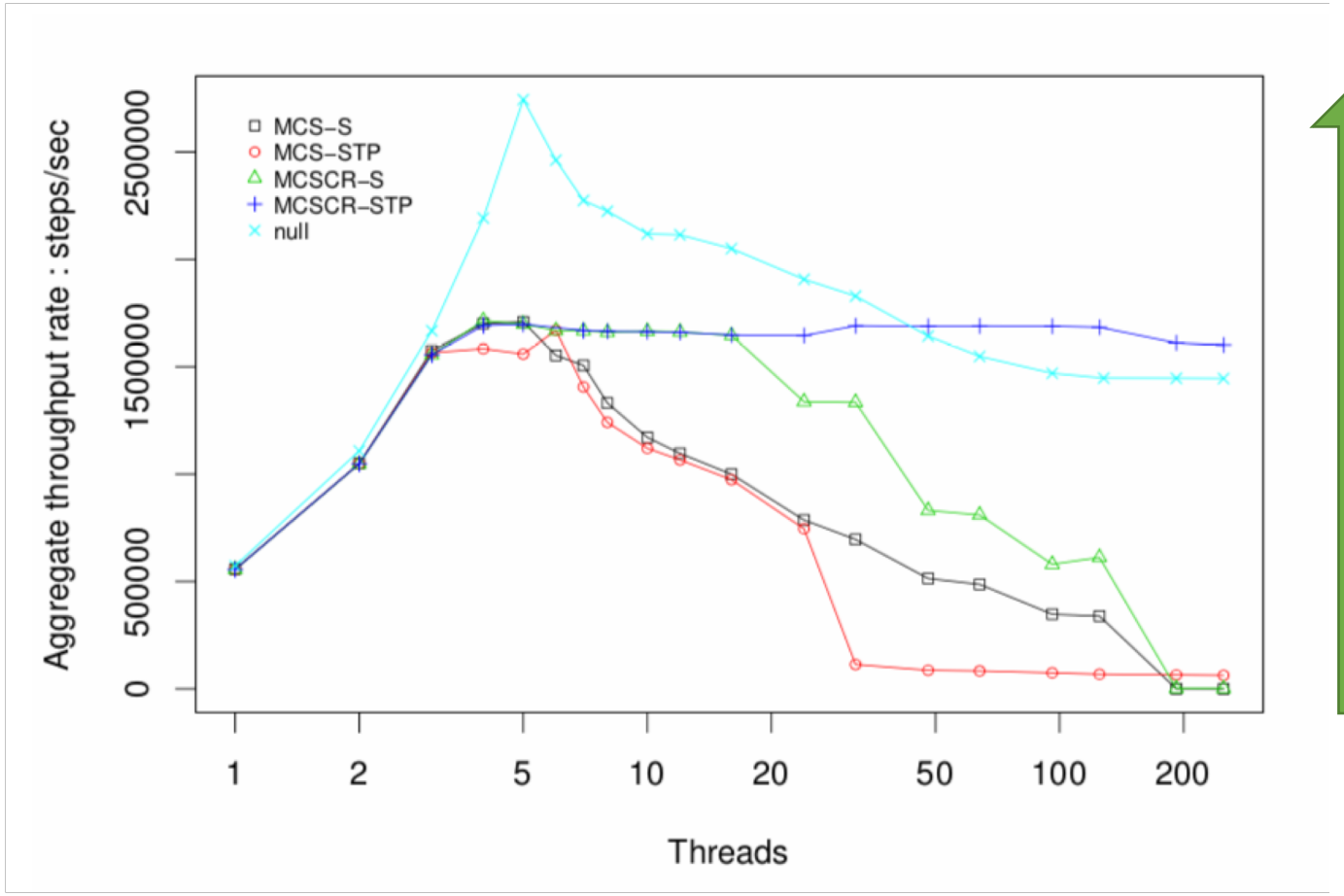


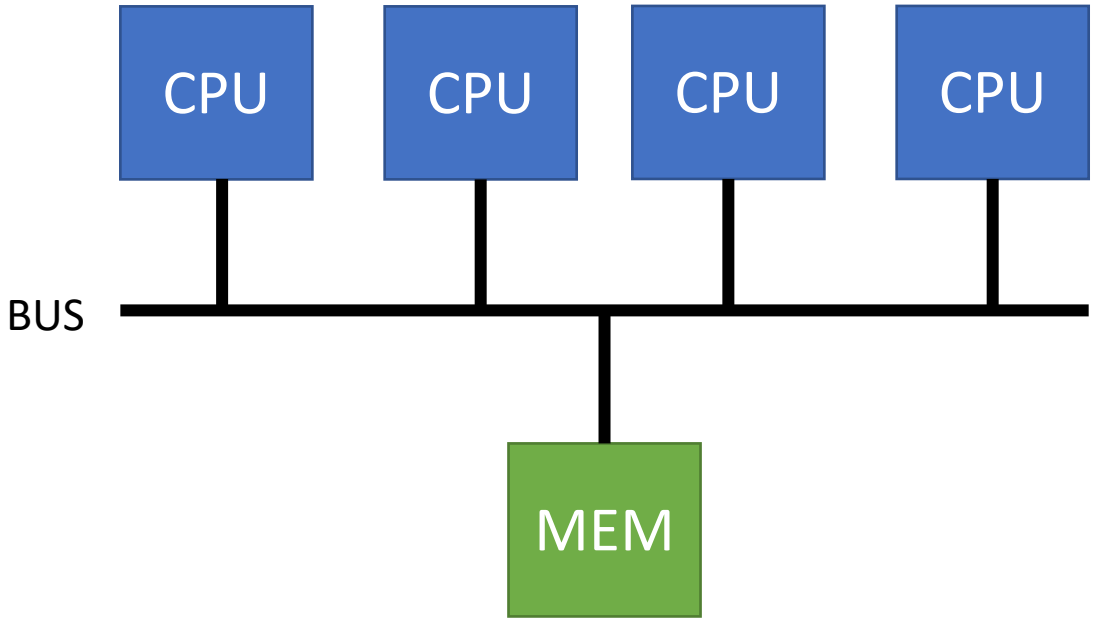
Figure credit: [2]

Compact NUMA-Aware Locks

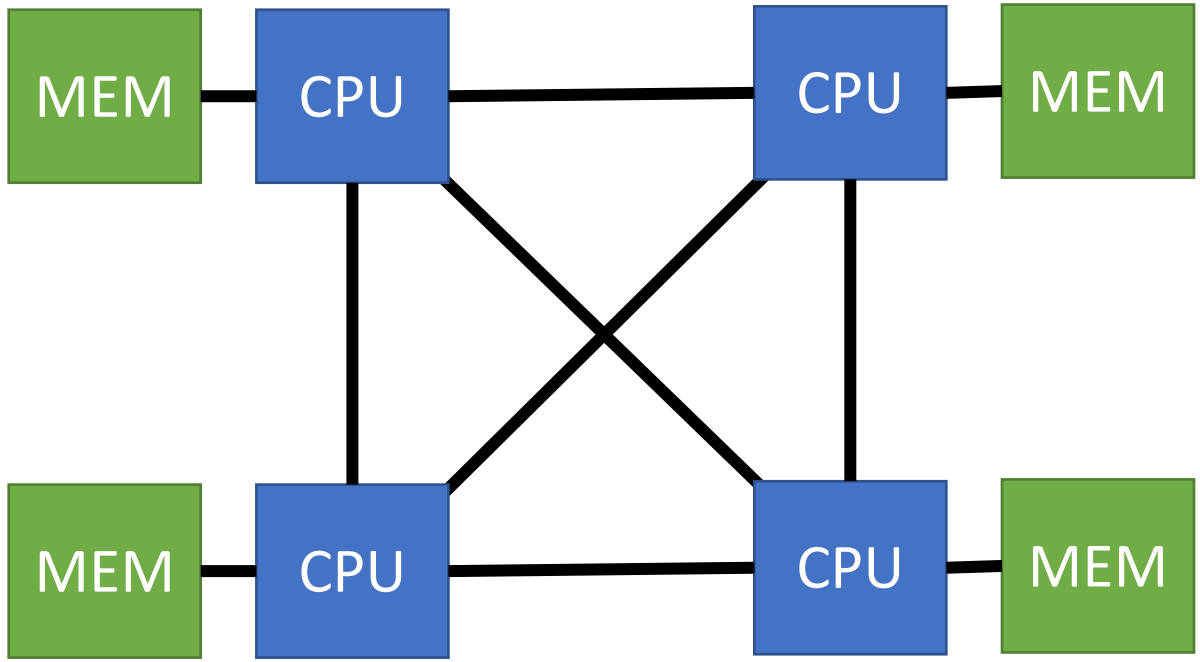
Dave Dice, Alex Kogan (October 2018)

Shared Memory Model

Uniform Memory Access (UMA)



Non-Uniform Memory Access (NUMA)



Compact NUMA-aware Locks

- Previous NUMA-aware Locks use hierarchy
  - Requires space linear to the number of sockets
  - Linux kernel allows only 4 bytes per lock
  - Databases and data structures that use fine grain locking
- Single-thread performance was not so good

Previous work

- Hierarchical backoff `test-and-set` lock (HBO)
  4. Hierarchical backoff locks for nonuniform communication architectures. Radovic, Hagersten (2003)
- Requires only one word of memory
  - Store the socket number of the lock holder
  - Same node acquire – small delay
  - Different node acquire – large delay
- Not fair
- Starvation is possible

Previous work

5. High performance locks for multi-level NUMA systems. Chabbi, Fagan, and Mellor-Crummey (2015)
6. Contention-conscious, locality-preserving locks. Chabbi, Mellor-Crummey (2016)

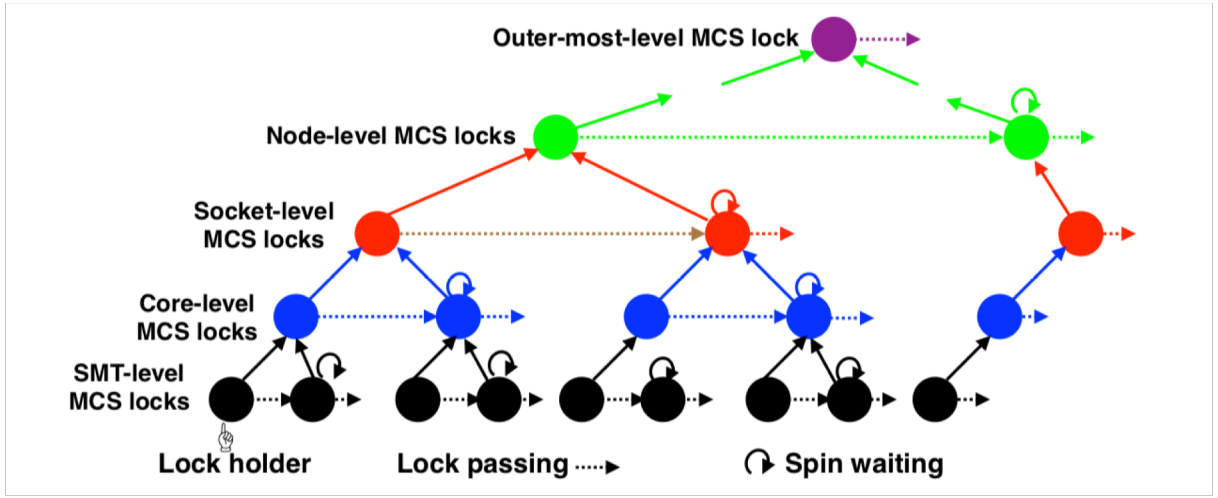


Image credit: [5]

Background (Linux Kernel Spin Lock)

- Multi-path approach
  - fast path – `test_and_set`
  - slow path – MCS lock
- Four-byte lock word is divided
  - 1 bit – lock value
  - 1 bit – pending
  - 30 bits – queue tail

Background (Linux Kernel Spin Lock)

- Acquire lock
  - try to flip the lock value from 0 to 1
    - successful ⇒ we acquired the lock
    - otherwise ⇒ check for contention (the remaining bits)
  - In case of contention ⇒ slow path ⇒ MCS lock
  - Head of the queue spins on the pending bit
- Release lock
  - Set the lock bit to 0
- No need to carry a queue node from lock to unlock

Compact NUMA-aware (CNA) lock

- Two queues
  - Main queue – threads running on the **same socket** as the lock holder
  - Secondary queue – threads running of a **different socket**
- Acquire lock – join the main queue
- Release lock – notify the first thread in the queue that is on the same socket

CNA Lock - one word requirement

- Always traverse the queue – too expensive
- Move the traversed threads to the secondary queue
- How?
- Add an extra field to the lock
  - Lock requires 2 words ✗
- Add an extra field to the queue node
  - an extra store instruction – possible cache miss - okay
- Pass to the locked value ✓

Performance – key-value map

- AVL tree
- Single lock
  - insert
  - remove
  - lookup

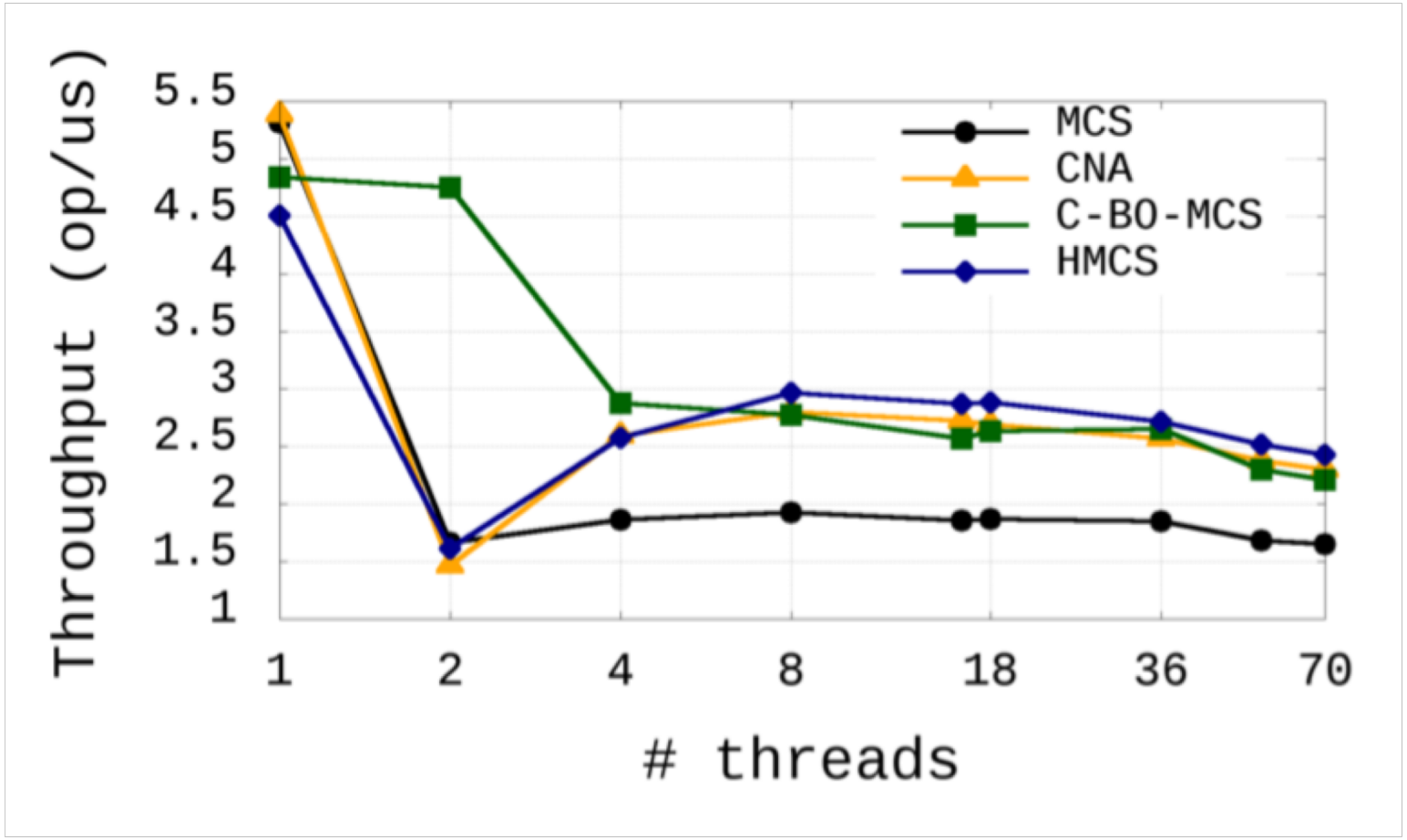


Figure credit: [3]

Barriers

Centralized barrier

- Each processor updates a small amount of space
  - **single counter** and Boolean flag
- Most barriers are designed to used repeatedly
  - separate phases of many-phase algorithms

Centralized barrier

- spin twice per barrier instance
  - all processors have left the previous barrier
  - all processors have arrived at the current barrier
- use a counter and a Boolean flag (sense)
  - last thread flips the sense
  - threads spin on sense
  - on broadcast-based cache-coherent multiprocessor, spinning on sense is not a problem

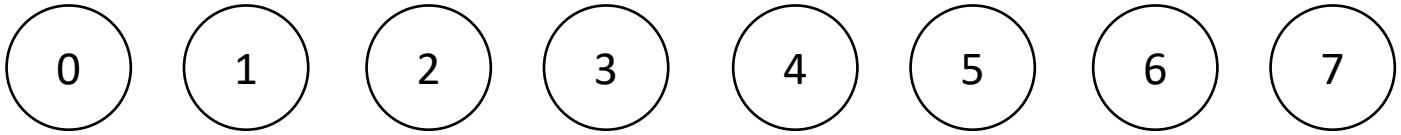
Centralized barrier

- Adaptive backoff schemes
  - latency increase
  - departure is delayed ⇒ arrival is delayed
- Centralized barriers will not scale well

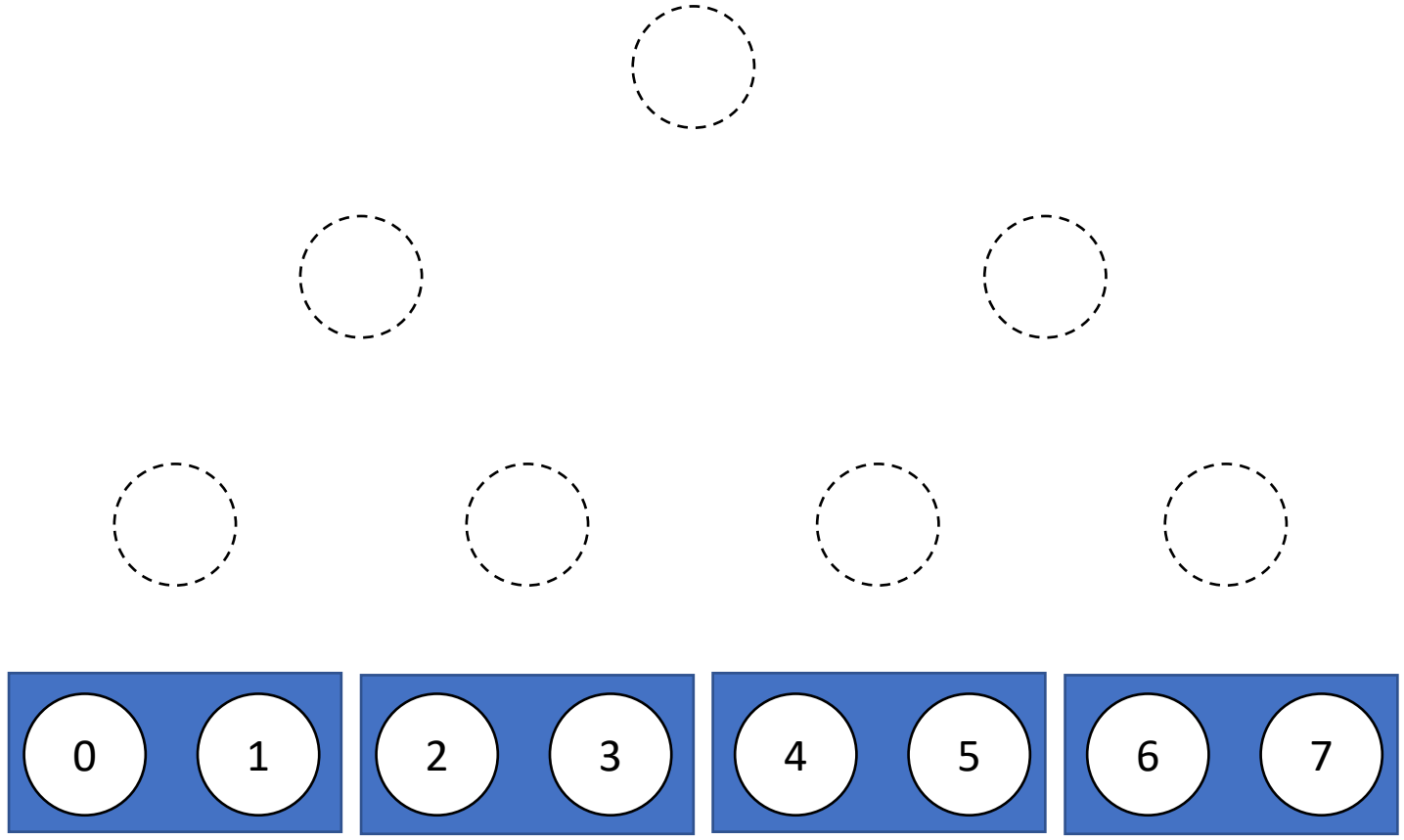
The Software Combining Tree Barrier

- Reduce hot-spot contention
- Processors are divided into groups
- One group is assigned to each leaft of the tree
- Last processor continues up the tree

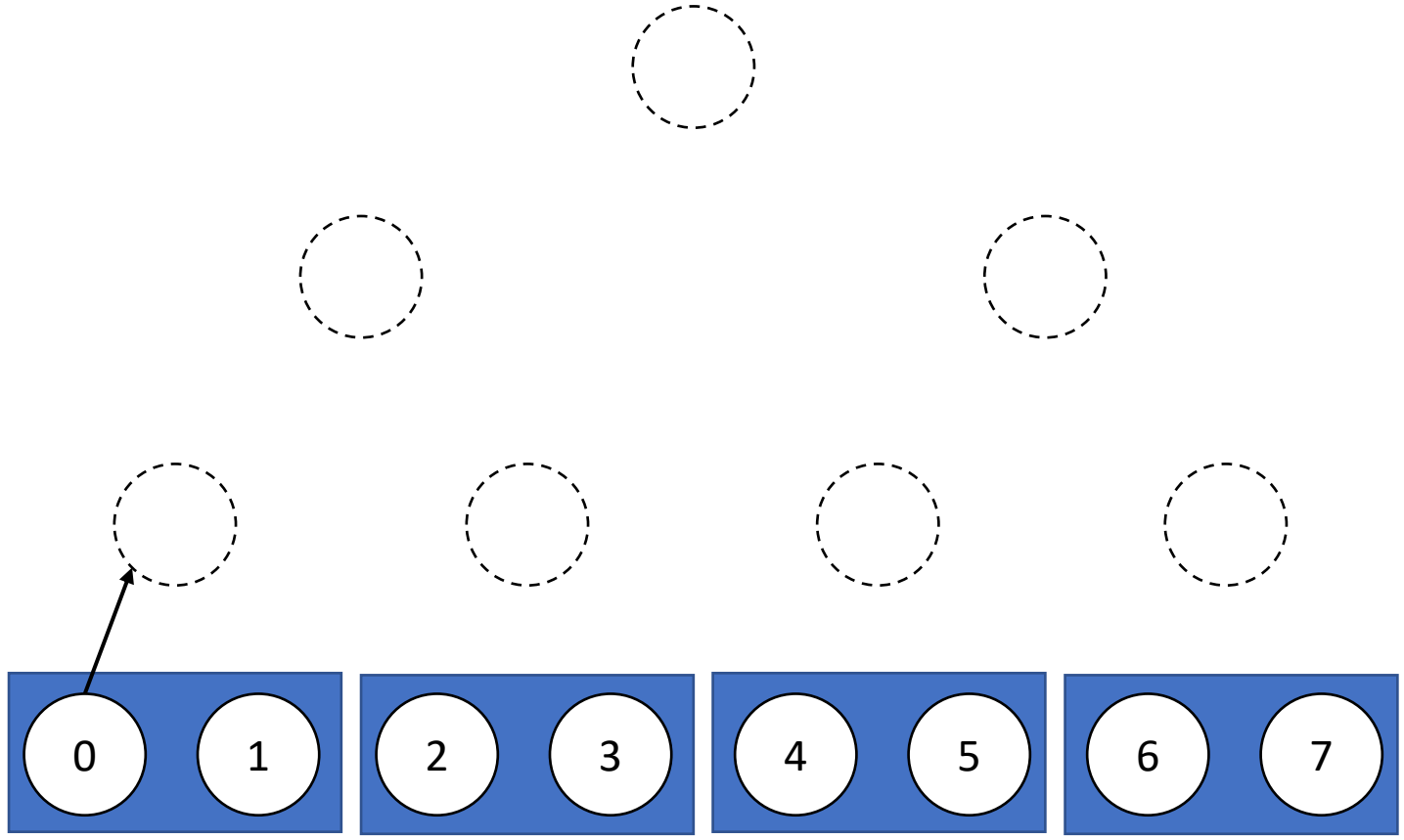
The Software Combining Tree Barrier



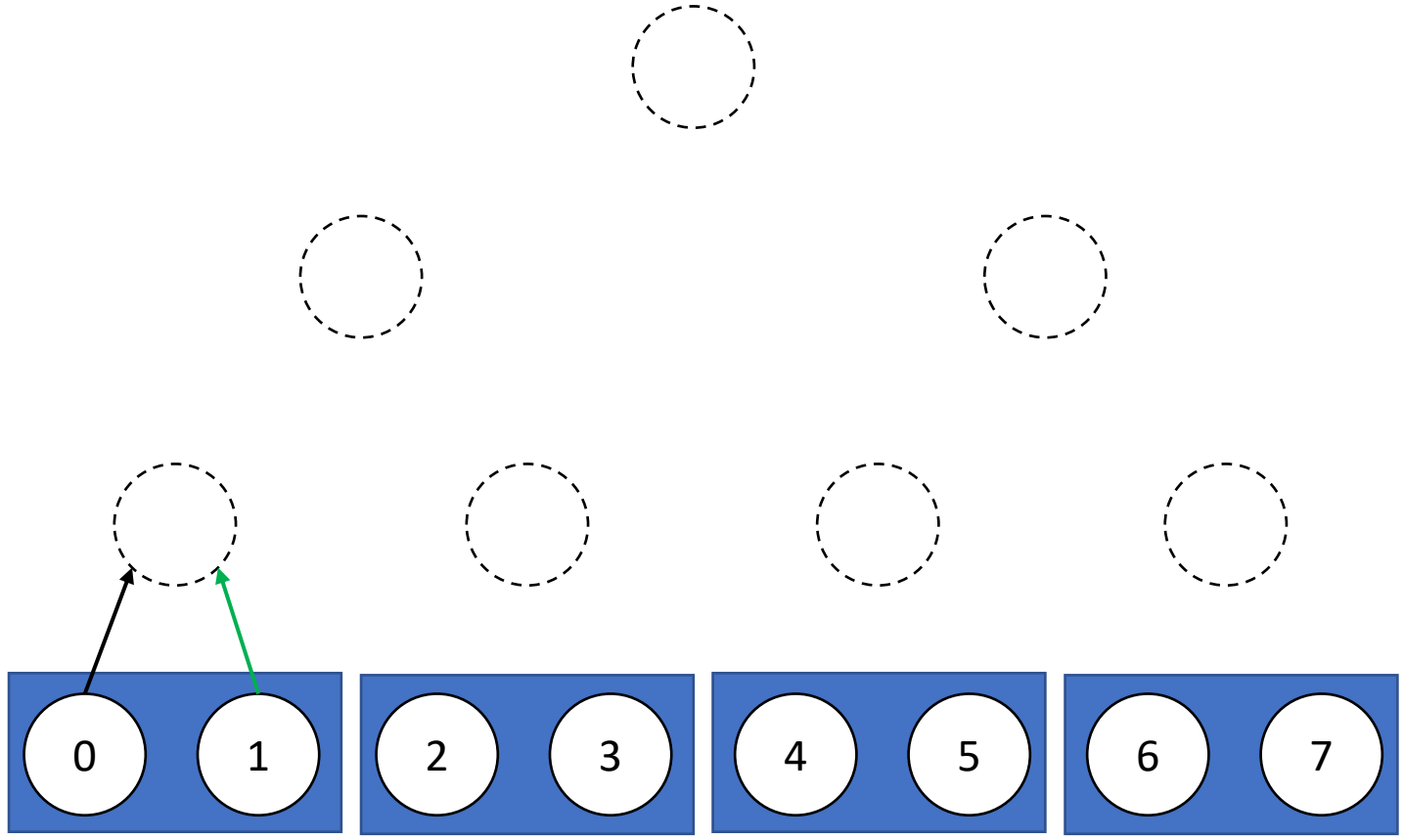
The Software Combining Tree Barrier



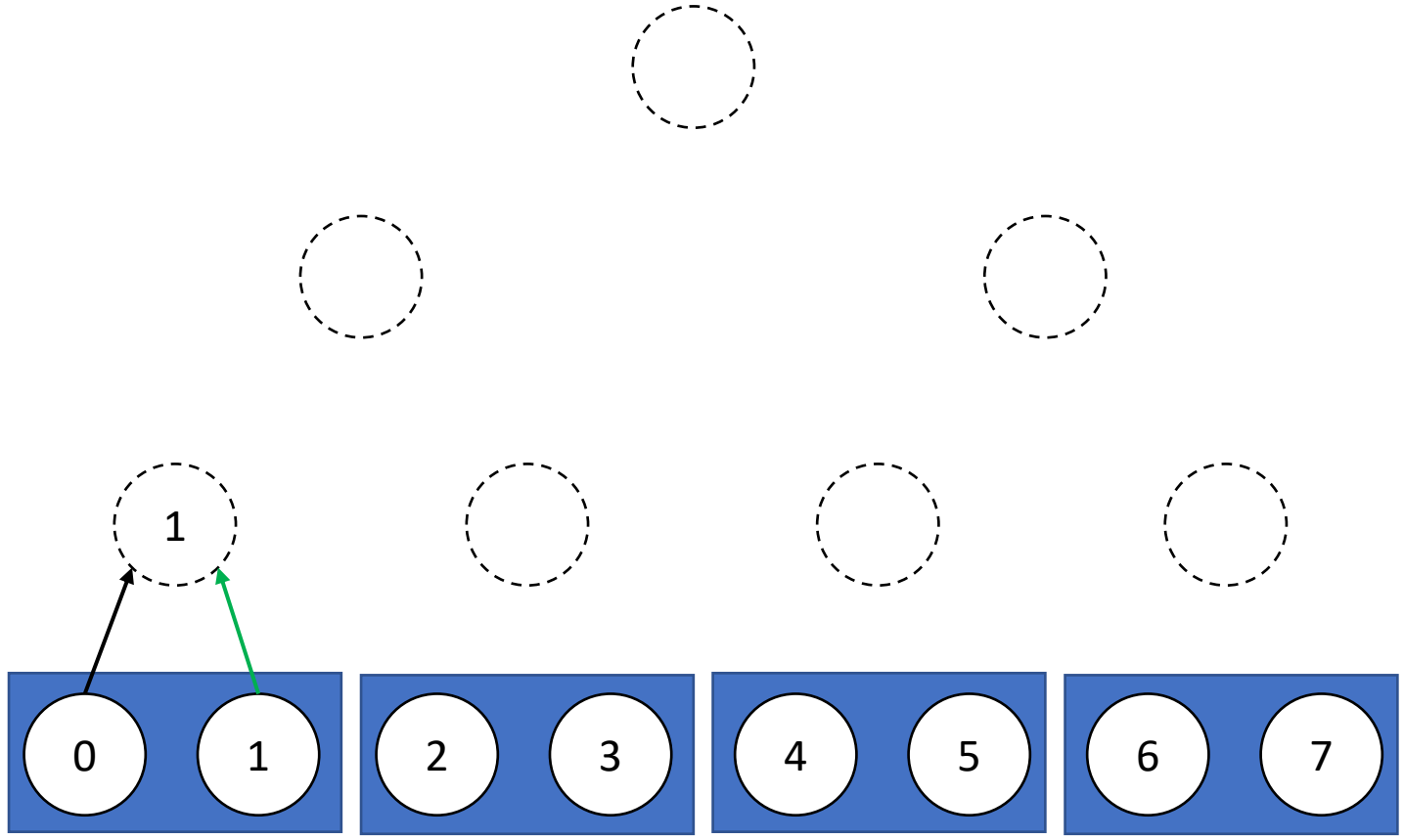
The Software Combining Tree Barrier



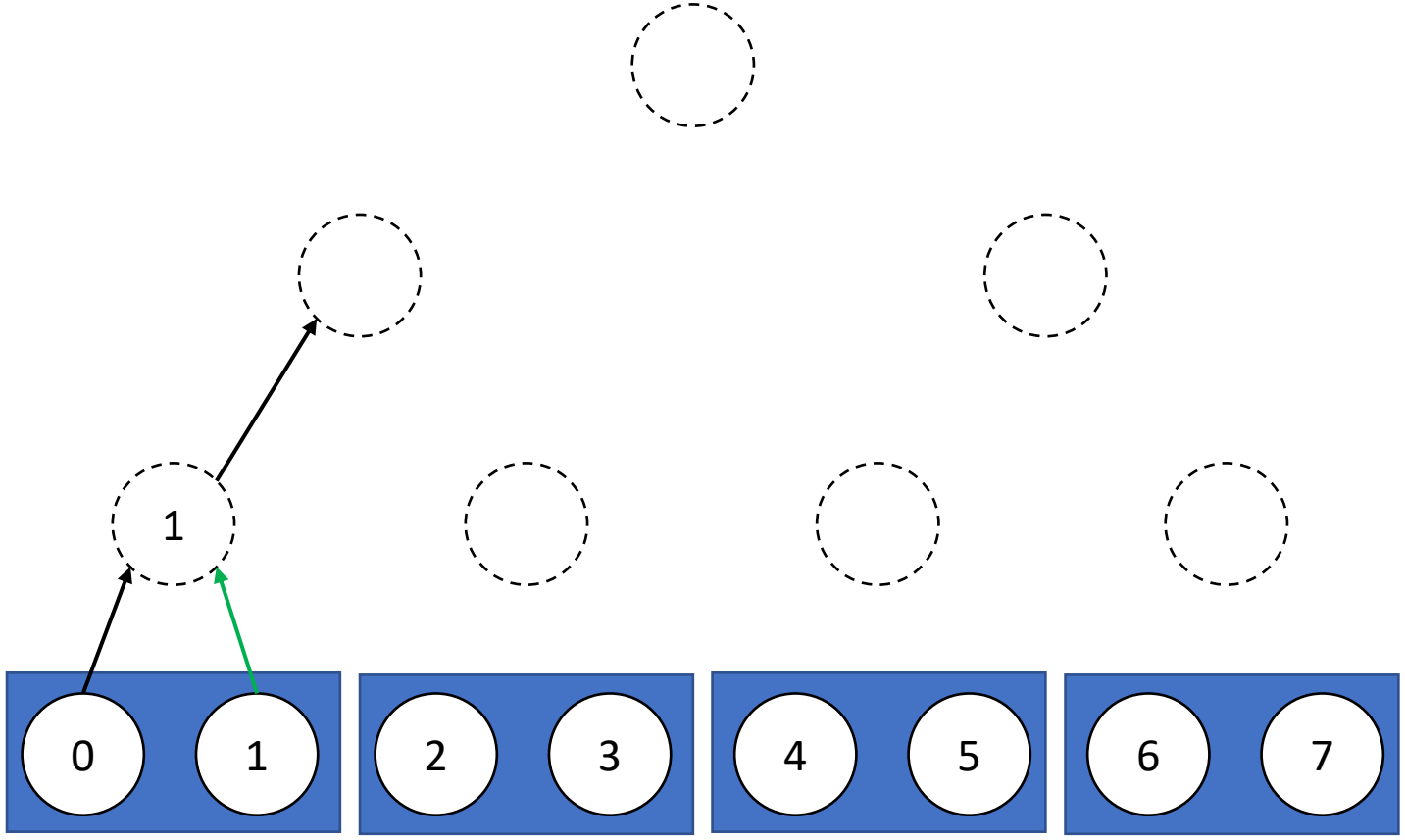
The Software Combining Tree Barrier



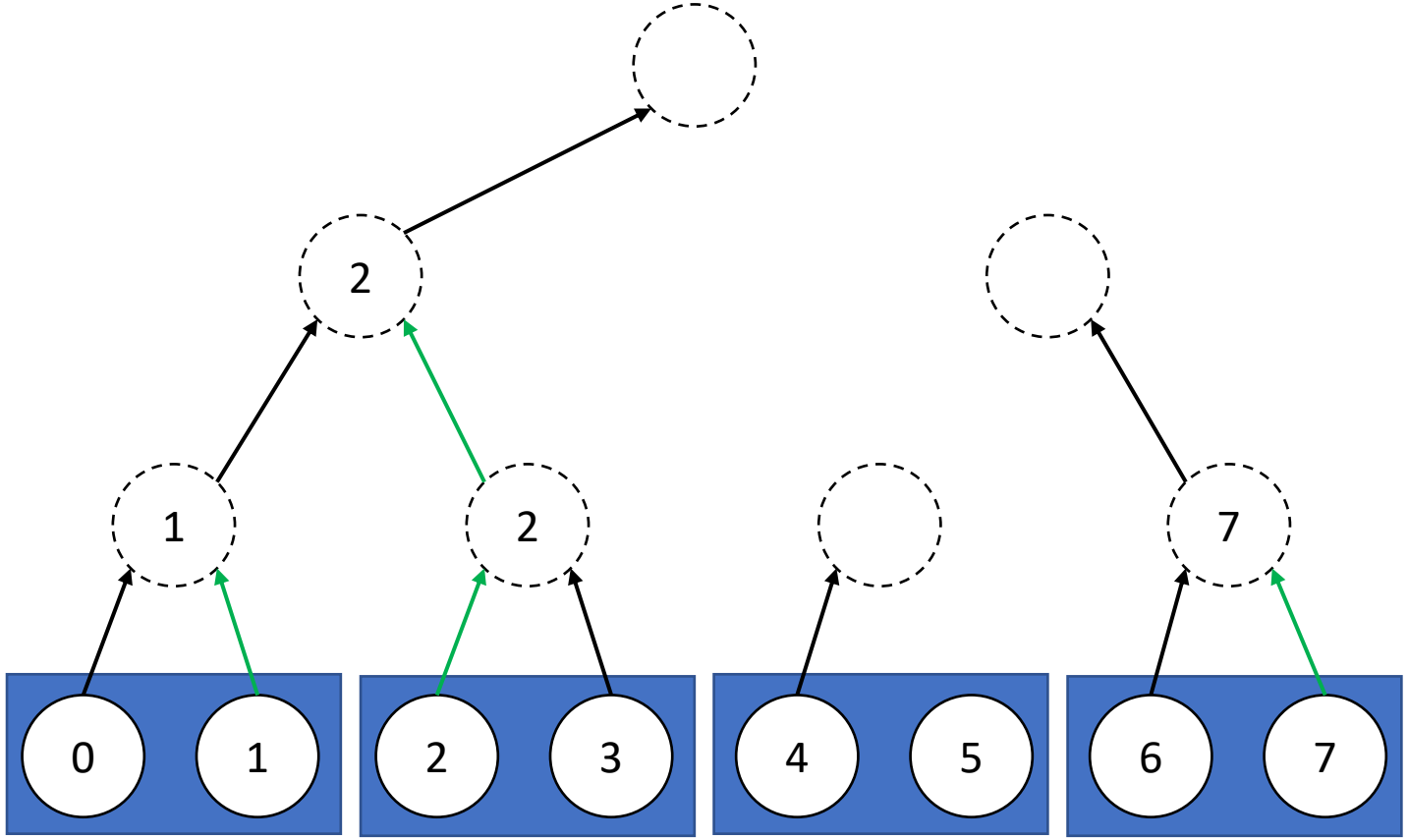
The Software Combining Tree Barrier



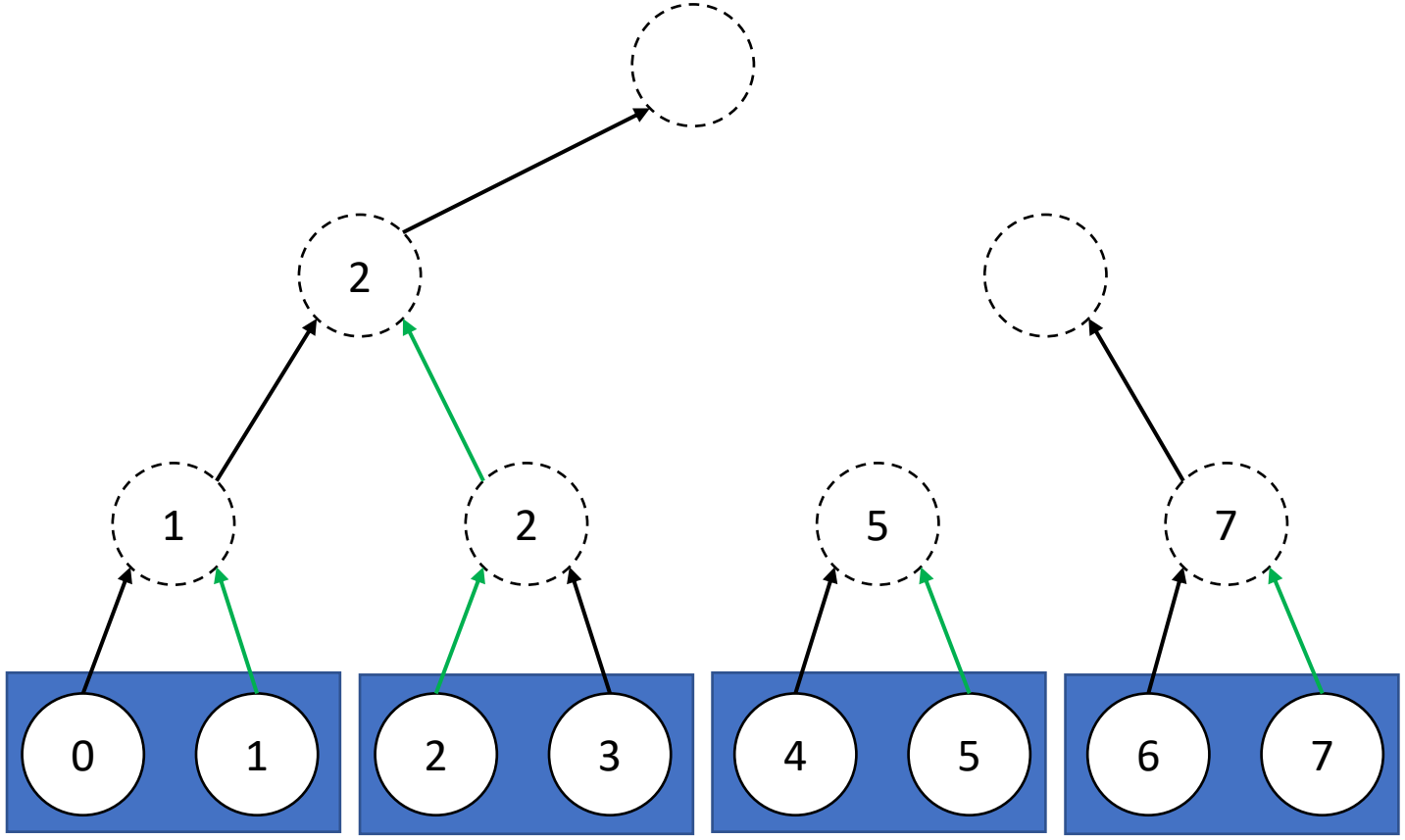
The Software Combining Tree Barrier



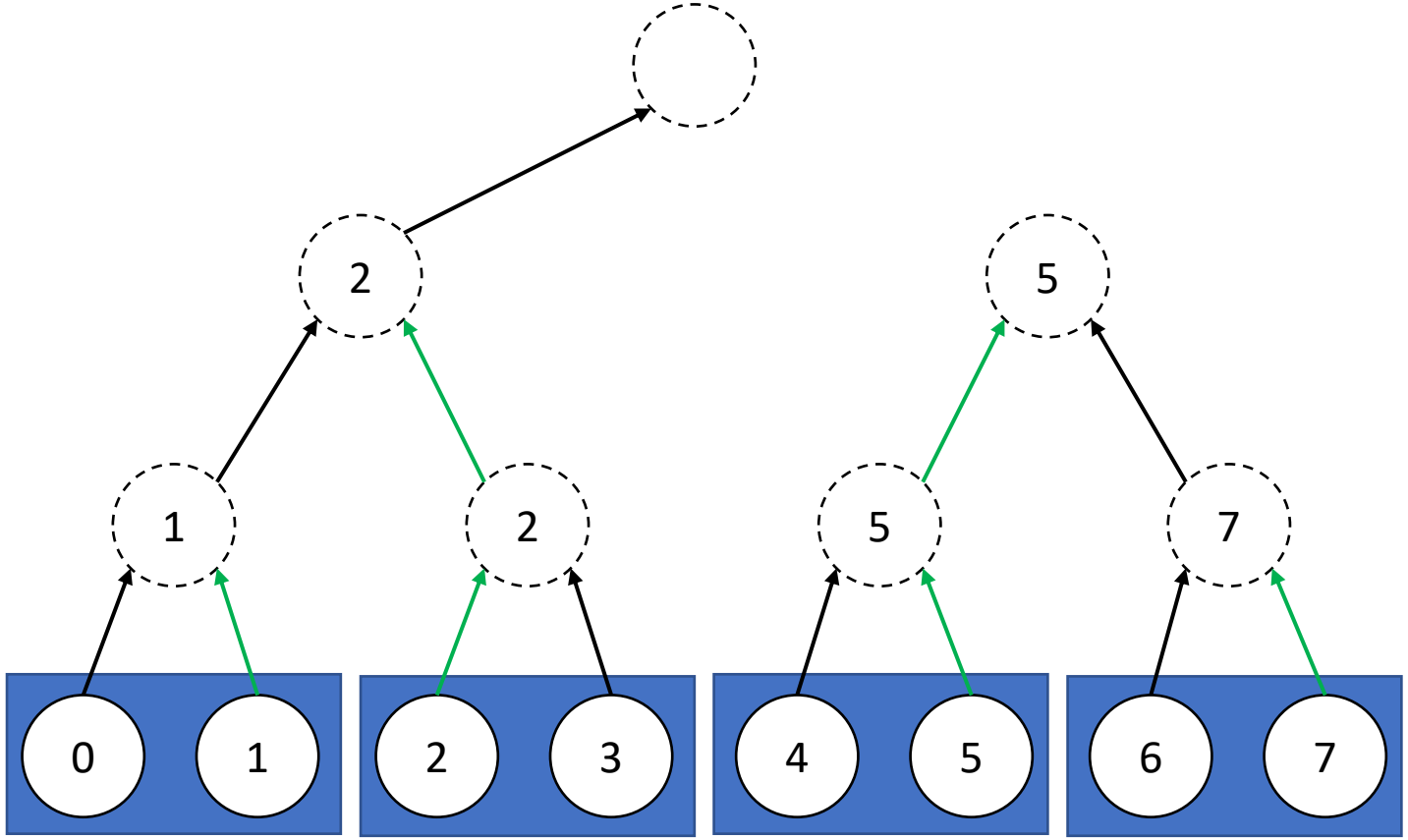
The Software Combining Tree Barrier



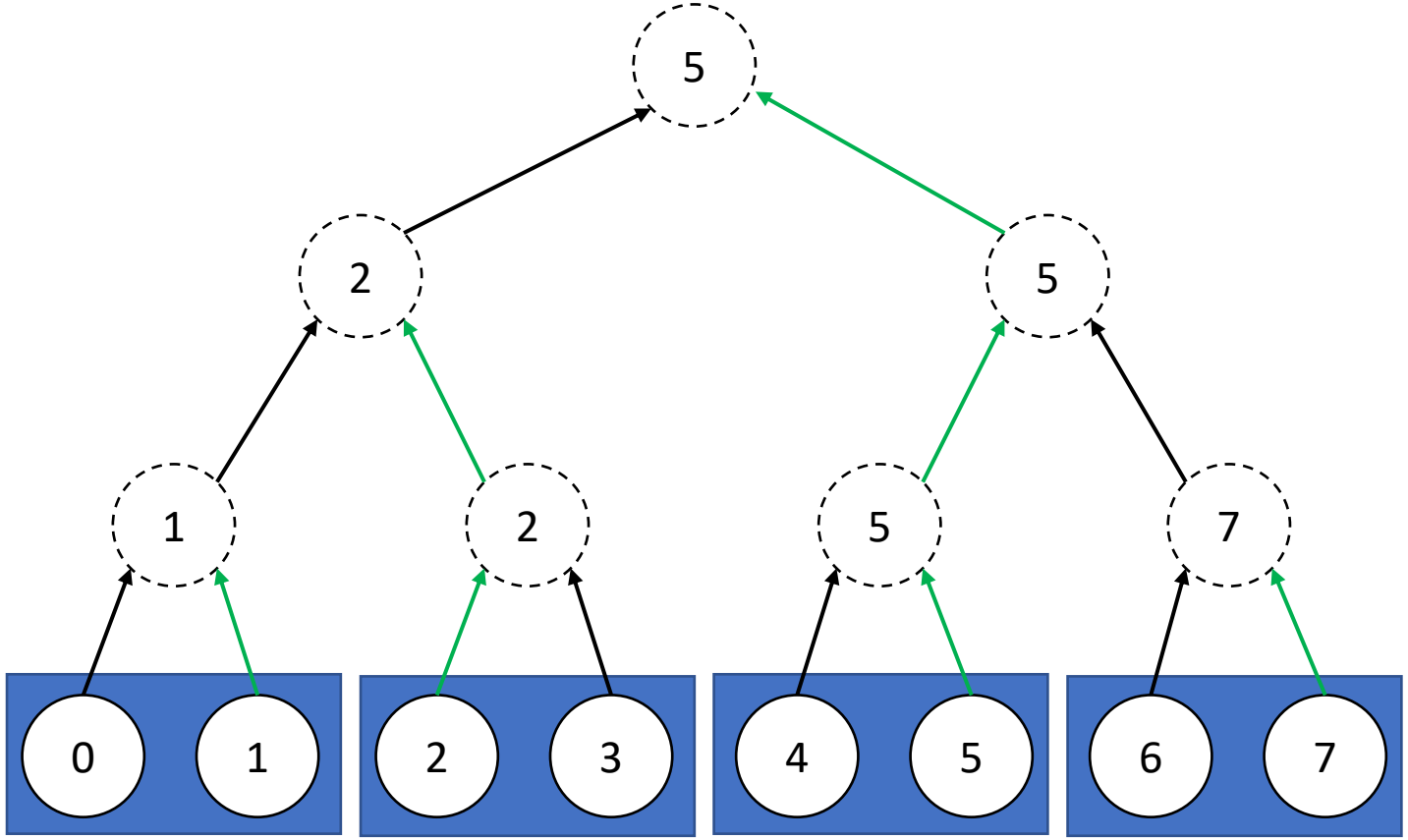
The Software Combining Tree Barrier



The Software Combining Tree Barrier



The Software Combining Tree Barrier



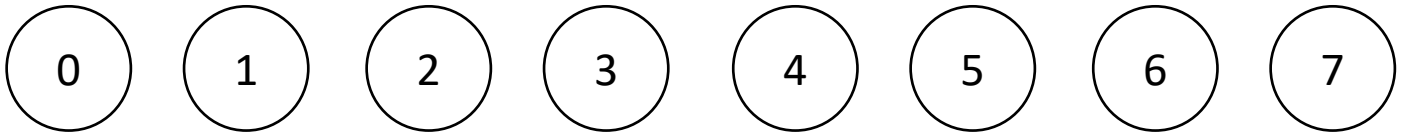
The Software Combining Tree Barrier

- Can significantly decrease memory contention
- Spin location cannot be statically determined
- Multiple processors can spin on same location in different barrier instances
- Not a problem on broadcast-based cache-coherent machines

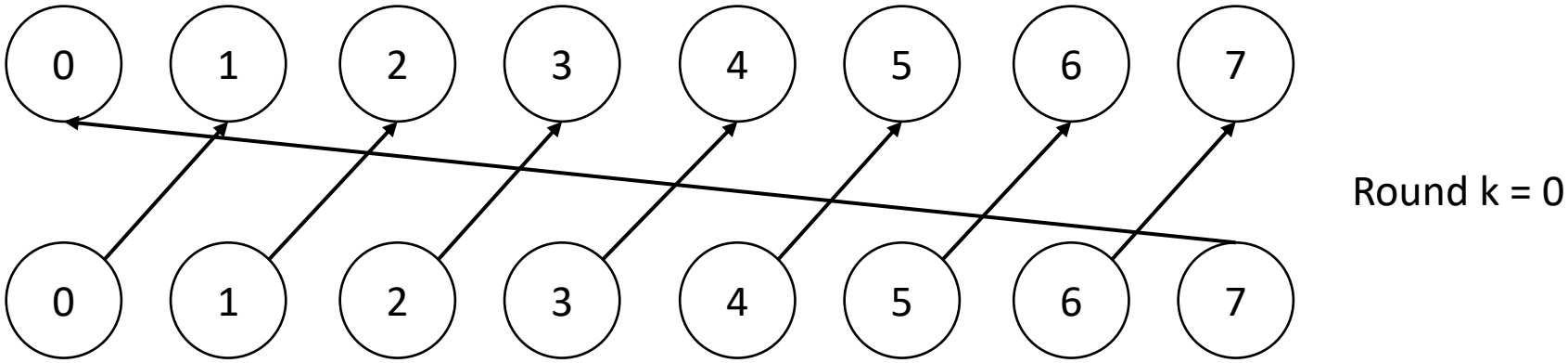
The Dissemination Barrier

- $\lceil \log_2 P \rceil$ rounds
- In round k (counting from 0), processor i, signals processor $(i + 2^k) \bmod P$
- $\lceil \log_2 P \rceil$ synchronization operations on the critical path
- $P * \lceil \log_2 P \rceil$ signals

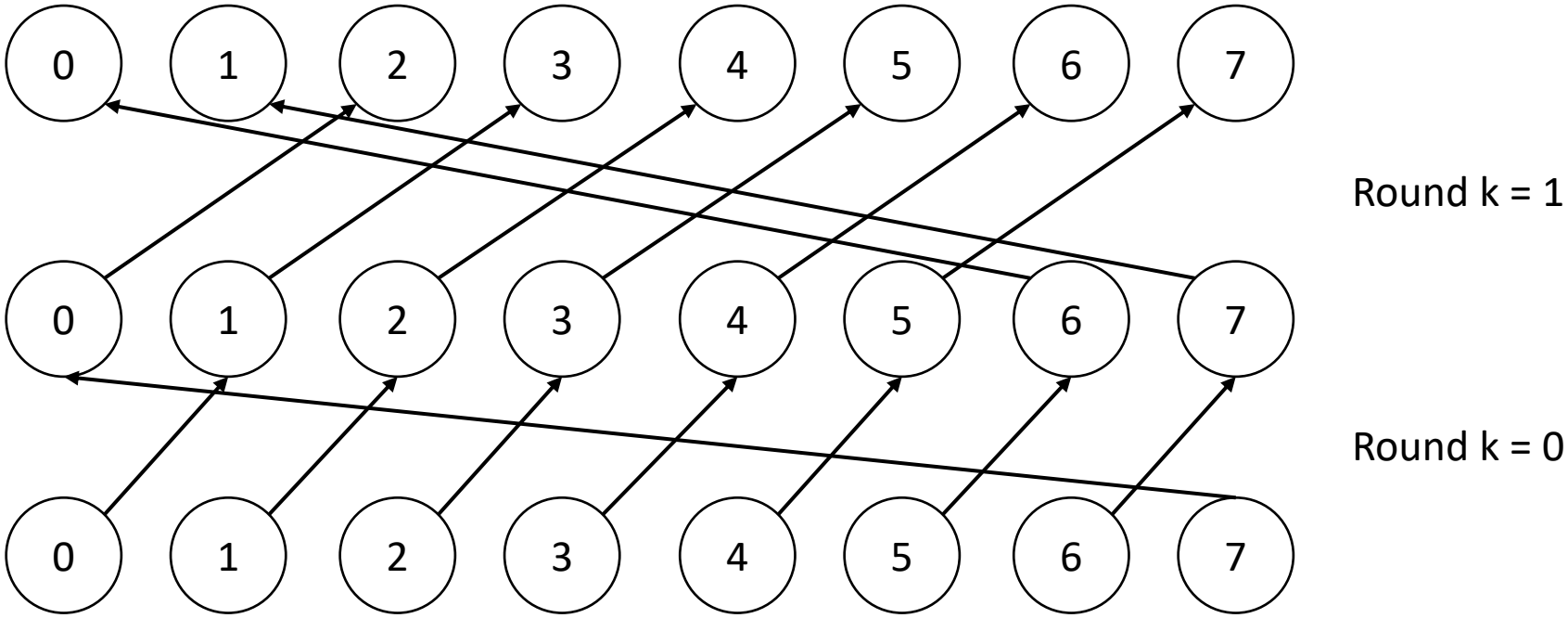
The Dissemination Barrier



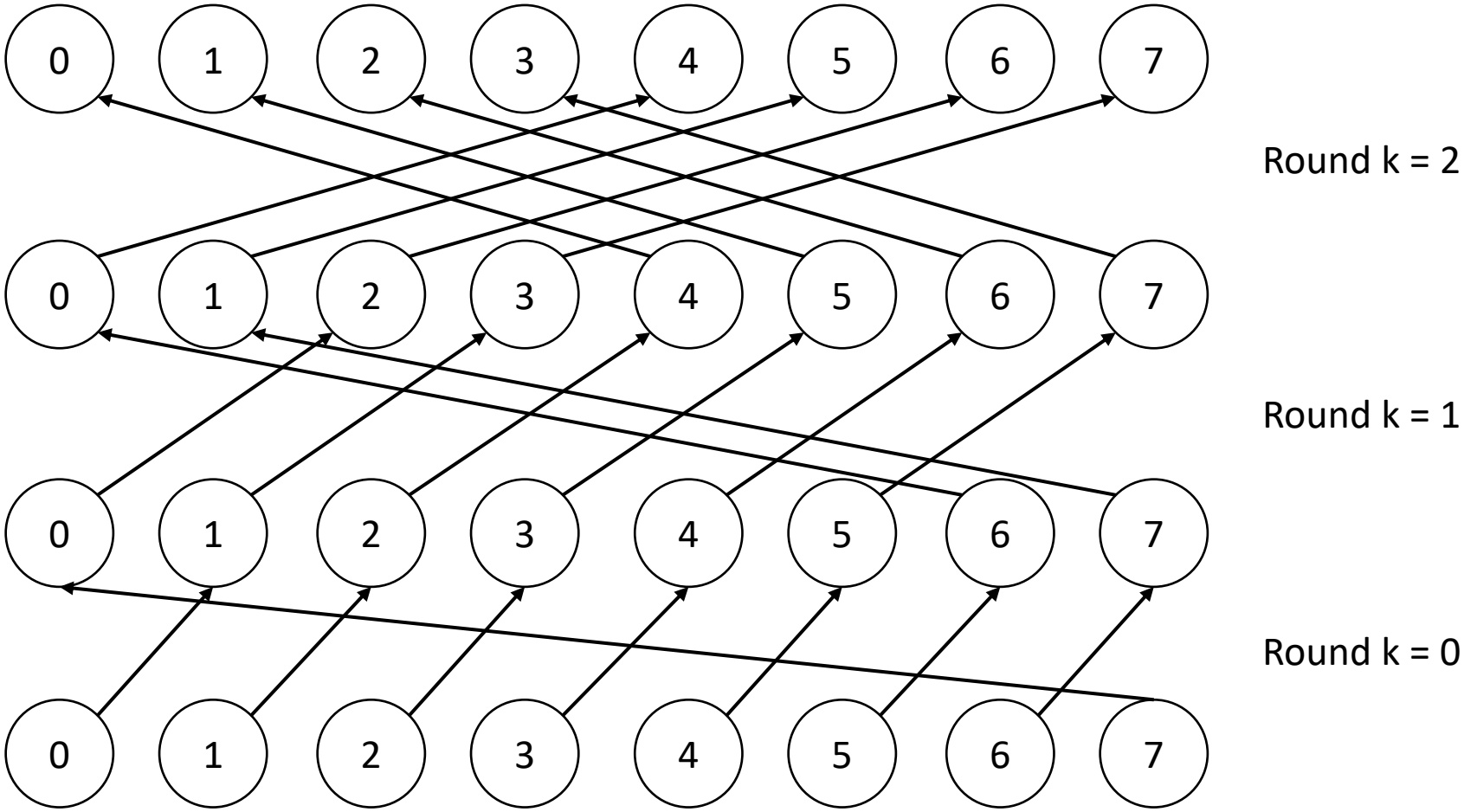
The Dissemination Barrier



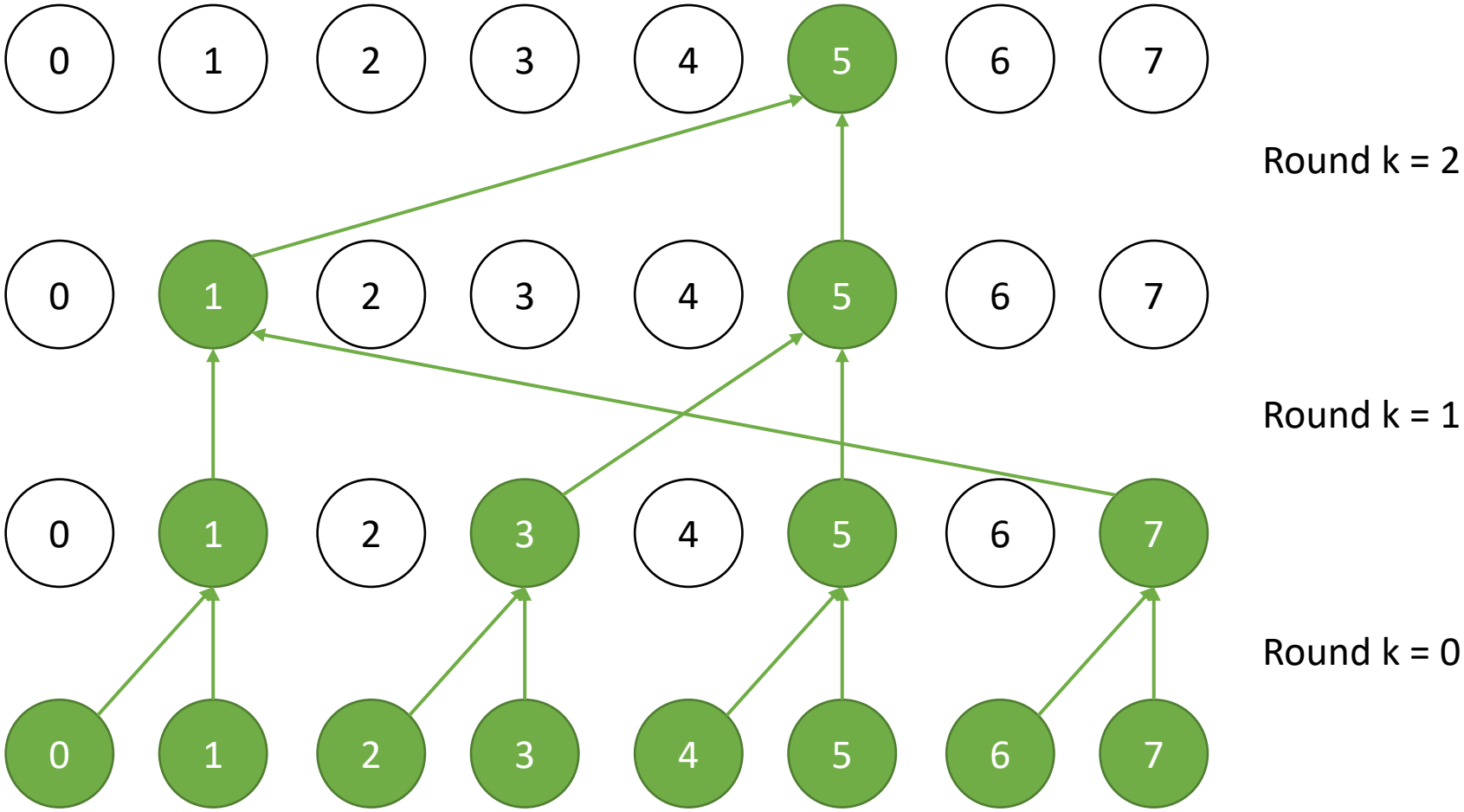
The Dissemination Barrier



The Dissemination Barrier



The Dissemination Barrier



Tournament barrier

- Processors begin at the leaves of a binary tree
- One processor from each node continues up
- ”Winning” processor is statically determined
  - no need for `fetch_and_Φ`
- In round k (counting from zero), processor i sets a flag awaited by processor j
  - $i \equiv 2^k \pmod{2^{k+1}}$, $j = i - 2^k$
- Processor i drops from the tournament

Tournament barrier

- Concurrent read, exclusive write (CREW)
  - spinning on a global flag
- Exclusive read, exclusive write (EREW)
  - spinning on separate flags - similar to combining tree

A new Tree-Based Barrier

- Spins only on locally accessible flags
- Requires O(P) space
- Performs theoretical minimum number of network transactions (2P-2)
- Performs O(log P) network transactions on its critical path

A new Tree-Based Barrier

- A pair of P-node trees
  - each processor is assigned a unique tree node
  - arrival tree – link to a parent
    - fan-in = 4
    - packing 4 bytes in a word (inspect status for all children)
  - wakeup tree – a set of child links
    - fan-out = 2
    - shortest critical path to resume P processors

A new Tree-Based Barrier

- Processor arrival
  - set the flag in its parent node
  - P - 1 network transactions
  - $\lceil \log_4 P \rceil$ critical path
- Processor wakeup
  - notify children by setting a flag in each of their nodes
  - P - 1 network transactions
  - $\lceil \log_2 P \rceil$ rounds

Butterfly - Performance

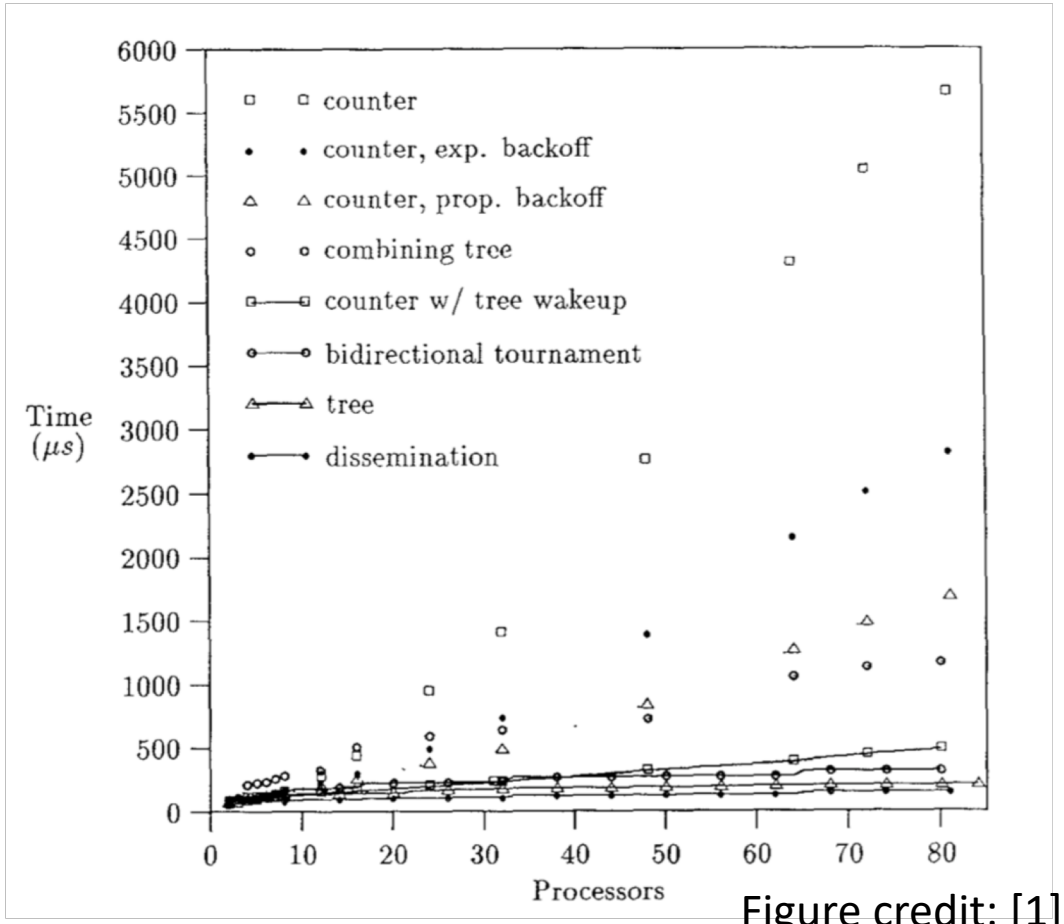


Figure credit: [1]

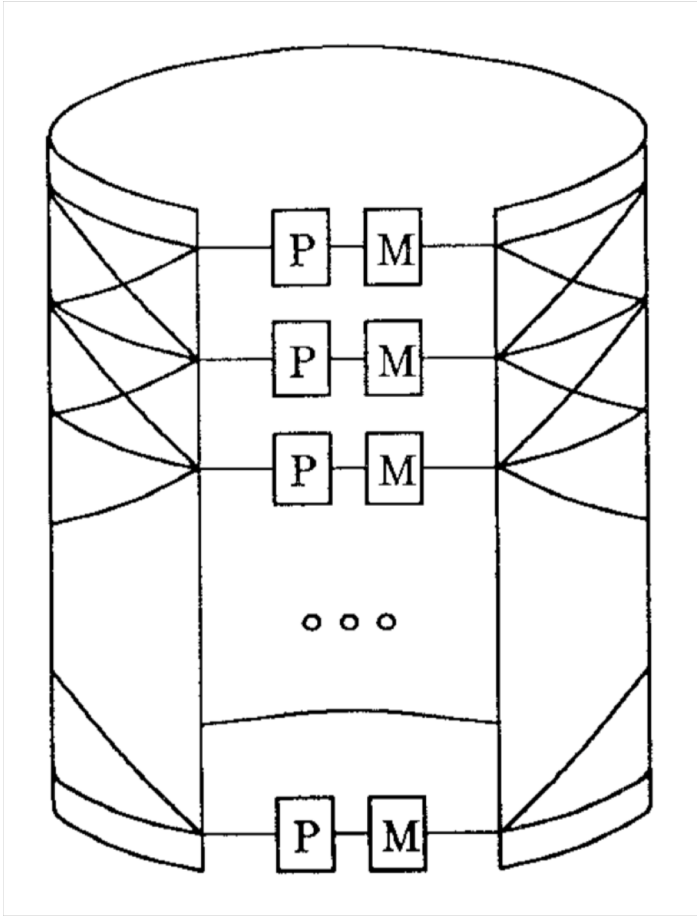


Figure credit: [1]

Butterfly - Performance

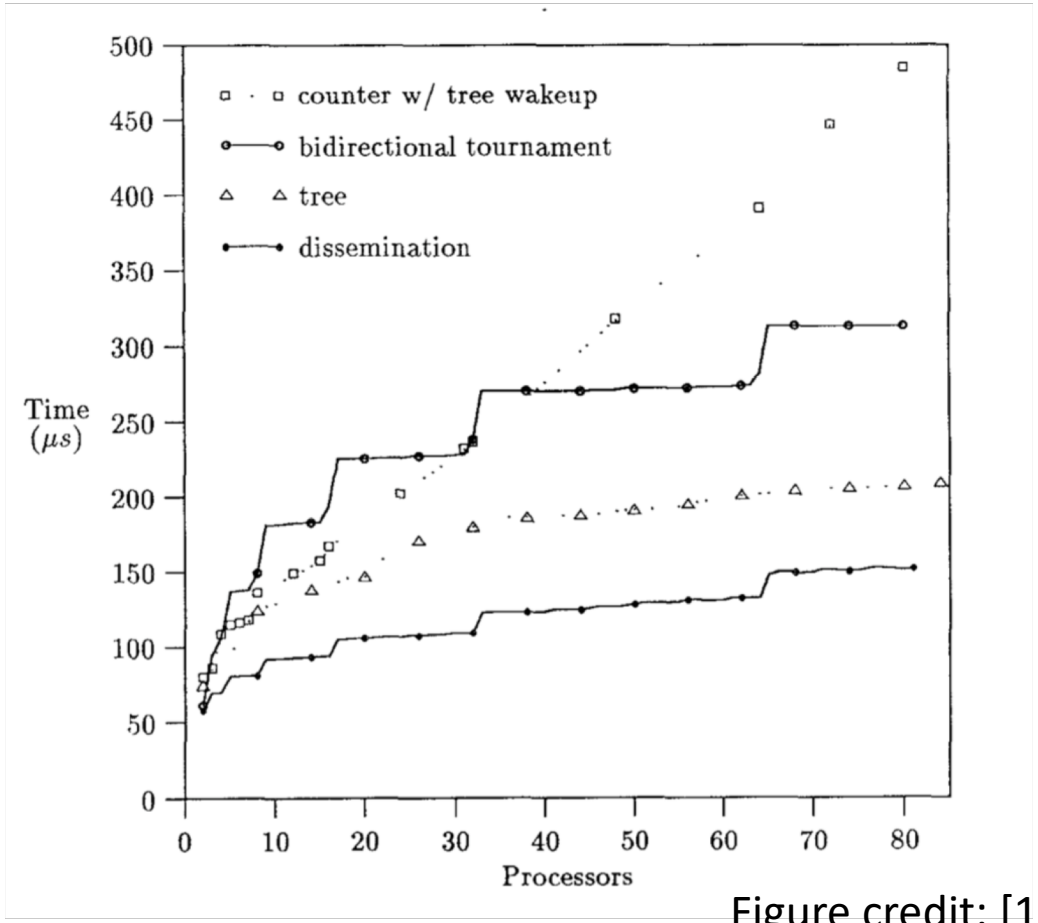


Figure credit: [1]

Performance

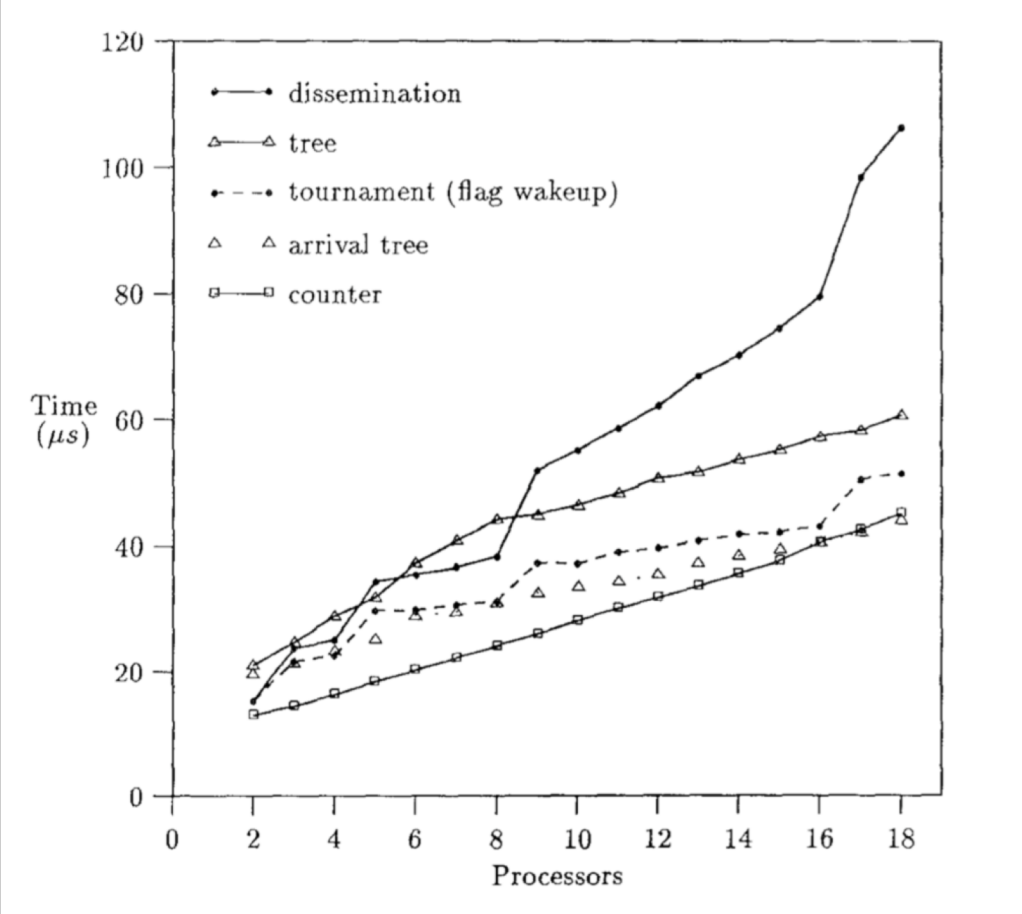


Figure credit: [1]

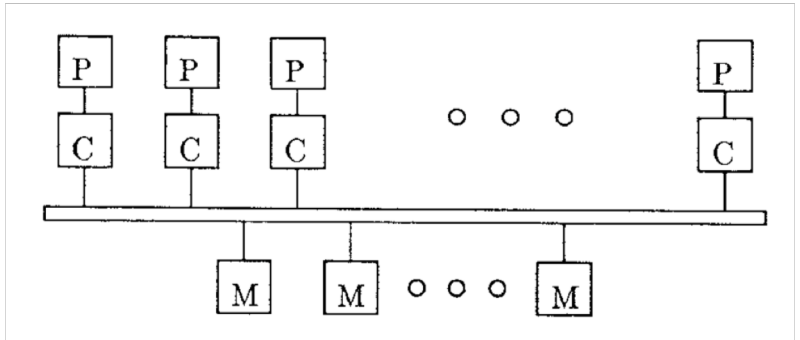


Figure credit: [1]

Takeaway

- Hardware support not always required
- If possible, perform local spinning
- Scalable synchronization primitives are important for applications performance