Scalable Synchronization

CSC 2/458

March 2026

Michael L. Scott

- Locks
- Barriers
- Read-mostly atomicity

Locks

- Centralized
  - (test and) test-and-set
  - ticket
- Queue-based
  - MCS
  - CLH
  - Hemlock
- Extensions

(ignoring RW-only locks, which are mostly of historic interest at this point)

(Test-and-)TAS lock

- Naive TAS lock thrashes the cache:

```
L.acquire():
    while L.TAS()   // spin
```

  - » Each TAS acquires the line in exclusive mode, stealing it from the thread that needs to release it and clogging the bus
- Partial solution: spin with an ordinary load [Rudolph & Segall, 1984]

```
L.acquire():
    while L.TAS()
        while L.load()   // spin
```

```
L.release():
    L.store(false)
```

  - » Still sees a flurry of traffic on release

TAS with exponential backoff

```
class lock
    atomic<bool> f := false
    const int base := ...                    // tuning parameters
    const int limit := ...
    const int multiplier := ...

lock.acquire():                              lock.release():
    int delay := base                            f.store(false)
    while f.TAS(‖)
        pause(delay)
        delay := min(delay × multiplier, limit)
     fence(R‖RW)
```

- Anderson et al. [1990]
- cf: Ethernet
- test-and- no longer needed
- NB: still *unfair*

Memory ordering

- All is ok if every shared location is `atomic` and `memory_order_seq_cst`
- Can relax (as on previous slide) if
  1. there's sufficient internally ordering to make sure the lock works correctly.
  2. there's a ‖RW ordering enforced within `acquire` and a RW‖ ordering enforced within `release`.

The ticket lock

- Fischer et al. [1979]; Reed and Kanodia [1979]
  - Inspired by “take a ticket” queueing at customer service counters
  - and by Lamport’s (load/store only) Bakery Lock [1974]

```
class lock
    atomic<int> next ticket := 0
    atomic<int> now_serving := 0
    const int base := ...   // tuning parameter

lock.acquire():
    int my_ticket := next_ticket.FAI()
    loop
        int ns := now_serving.load()
        if ns = my_ticket return
        pause(base × (my_ticket − ns))
```

```
lock.release():
    int t := now_serving.load() + 1
    now_serving.store(t)
```

Queued locks

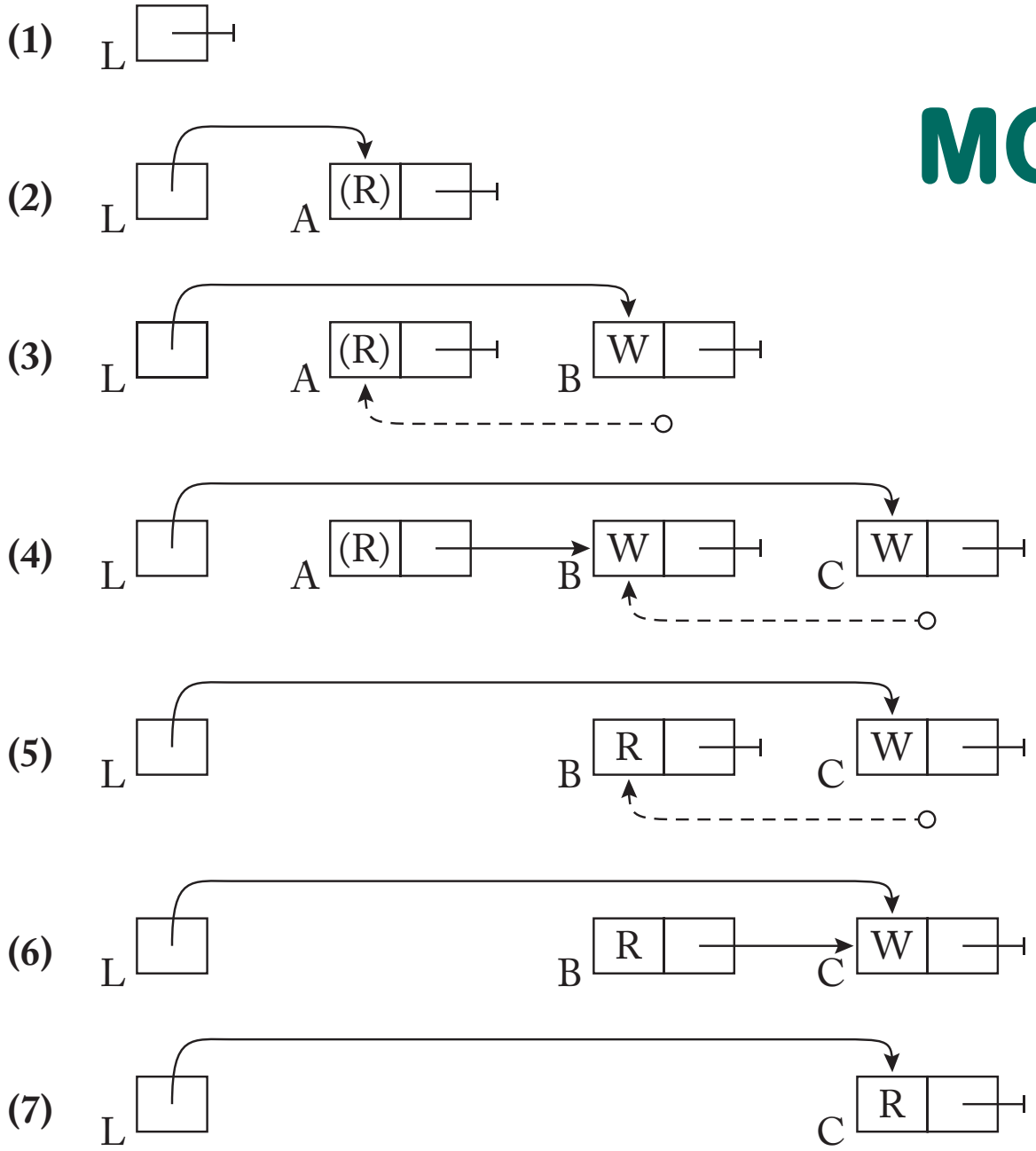
- Array-based (O(*tn*) space for *t* threads & *n* locks):
  - » Anderson [1990], Graunke & Thakkar [1990]
- Linked-list based (O(*t*+*n*) space for *t* threads & *n* locks):
  - » MCS [Mellor-Crummey & Scott 1991]
  - » CLH [Craig 1993, Landin & Hagersten 1994]
  - » Hemlock [Dice & Kogan 2021]
- All are fair, like the ticket lock
  - » which may be good or bad, depending…

The MCS lock [1991]

```
type qnode = record
    atomic<qnode*> next
    atomic<bool> waiting
class lock
    atomic<qnode*> tail := null
lock.acquire(qnode* p):
    p→next := null
    p→waiting := true
    qnode* prev := tail.swap(p)
    if prev ≠ null
        prev→next.store(p)
        while p→waiting.load();
```

```
lock.release(qnode* p):
    qnode* succ := p→next.load()
    if succ = null
        if tail.CAS(p, null) return
        repeat succ := p→next.load()
    until succ ≠ null
    succ→waiting.store(false)
```

MCS in operation



- NB: employs both CAS and swap
- may wait for successor in release
- works well on both CC-NUMA and NRC-NUMA machines

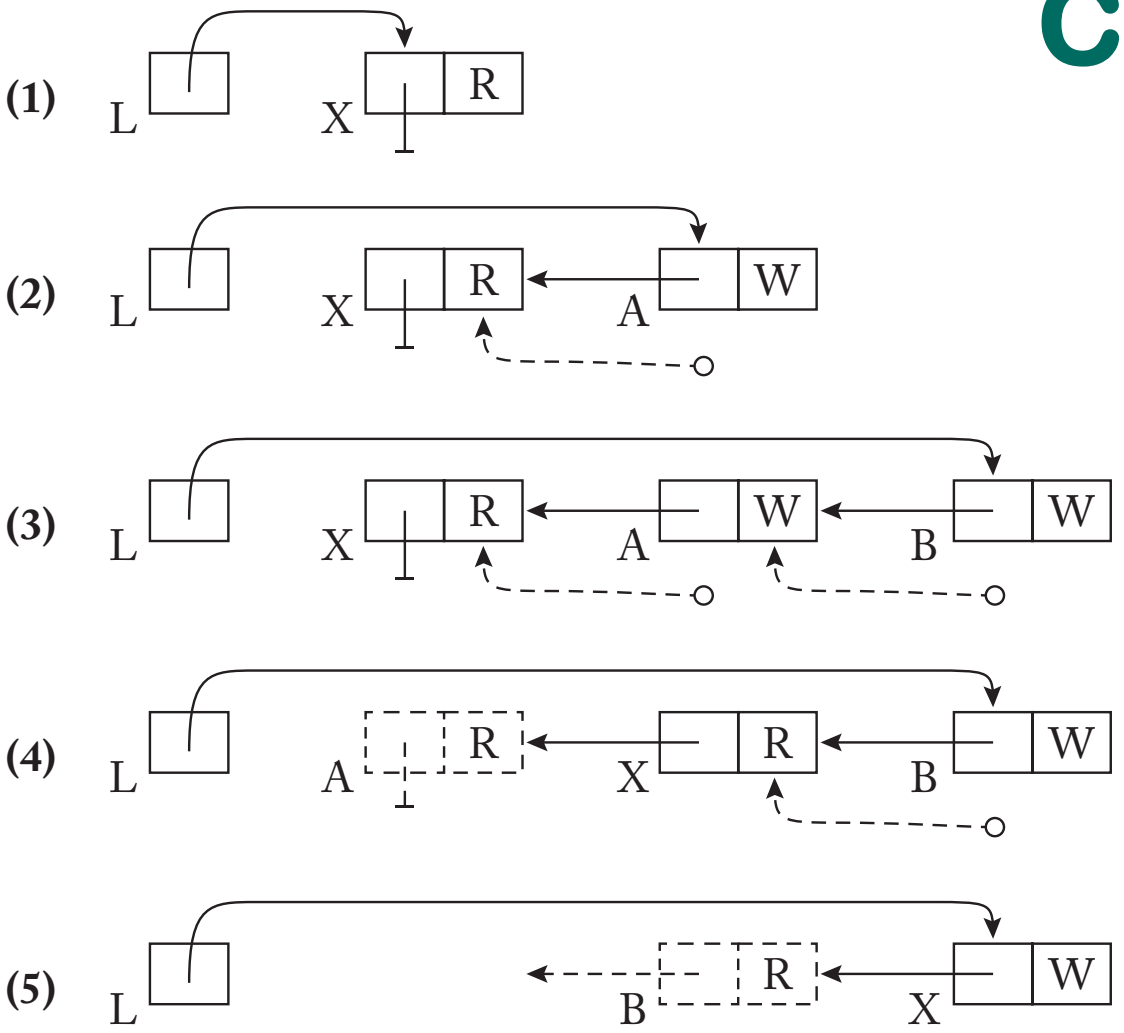
The CLH lock

```
type qnode = record
    qnode* prev
    atomic<bool> succ_must_wait
class lock
    atomic<qnode*> tail := new qnode(null, false)
    ~lock(): delete tail
lock.acquire(qnode* p):
    p→succ_must_wait.store(true)
    qnode* pred := p→prev := tail.swap(p)
    while pred→succ_must_wait.load()
```

- Craig [1993] and Landin & Hagersten [1994]

```
lock.release(qnode** pp):
    qnode* pred := (*pp)→prev
    (*pp)→succ_must_wait.store(false)
    *pp := pred
```

CLH in operation

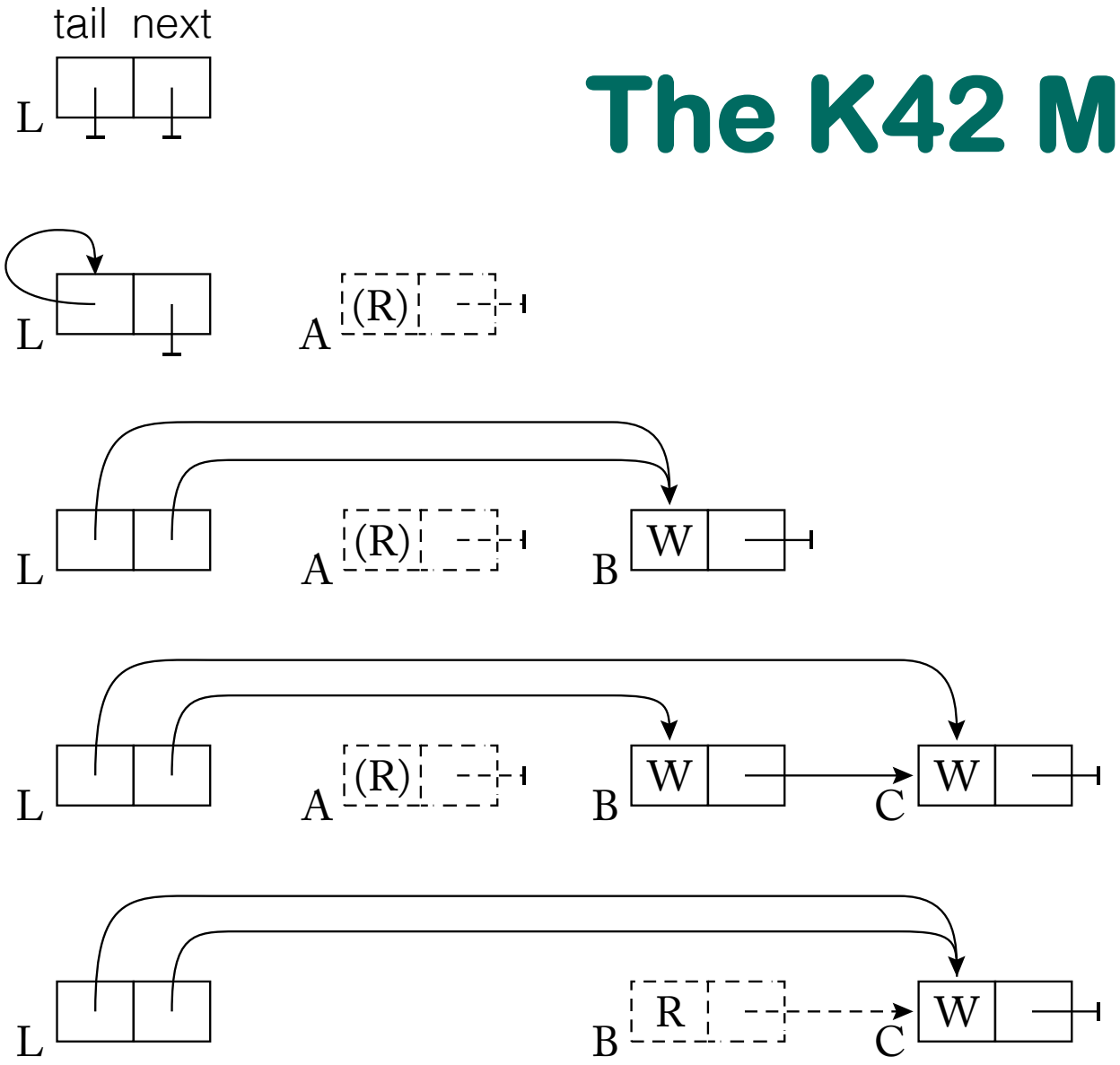


- avoids remote spinning only on a CC-NUMA machine
- uses swap but not CAS

Queued lock caveats

- qnode parameters (non-standard interface) can be avoided by keeping a bit more info in the tail node [Auslander et al. 2003 (MCS); Scott [2013], Scott & Brown 2024 (CLH)]
- Priority can be added by scanning a doubly-linked queue
- Timeout can be added by marking nodes as abandoned
  - » preemption tolerance, too, with some help from the operating system

The K42 MCS lock



- R: running (tail was null)
- W: waiting
- dotted qnodes can safely be discarded

The “K42” CLH lock

```
qnode* thread_qnode_ptrs[𝒯] := { i ∈ 𝒯 : new qnode(false) }
type qnode = record
    atomic<bool> succ_must_wait
class lock
    atomic<qnode*> tail := new qnode(false)
    qnode* head     // accessed only by owner
lock.acquire():
    qnode* p := thread_qnode_ptrs[self]
    p→succ_must_wait.store(true)
    qnode* pred := tail.swap(p)
    while pred→succ_must_wait.load();
    head := p
    thread_qnode_ptrs[self] := pred
```

```
~lock(): delete tail
```

```
lock.release():
    head->succ_must_wait.store(false)
```

Hemlock

```
type status = atomic<lock*>
status ts[𝒯] := { null ... }

class lock
    atomic<status*> tail := null

lock.acquire():
    status* pred := tail.swap(&ts[self])
    if pred ≠ null
        while pred⟶load() ≠ this;   // spin
        pred⟶store(null)            // handshake

lock.release():
    if ¬tail.CAS(&ts[self], null)
        ts[self].store(this)
        while ts[self].load() ≠ null;   // spin
```

- Dice & Kogan [2021]

- one single statically allocated qnode per thread
- wake successor by writing lock address to qnode
- *handshake* to free qnode for future use

Hemlock (optimized)

```
type status = atomic<lock*>
status ts[𝒯] := { null ... }

class lock
    atomic<status*> tail := null

lock.acquire():
    status* pred := tail.swap(&ts[self])
    if pred ≠ null
        while pred→load() ≠ this;   // spin
        pred→store(null)            // handshake

lock.release():
    if ¬tail.CAS(&ts[self], null)
        ts[self].store(this)
        while ts[self].load() ≠ null;   // spin
```

- *handshake* to free qnode for future use — access alternates between owner and successor(s)

```
while !pred.CAS(this, null);
```

```
while ts[self].FAA(0) != null;
```

Extensions

- Trylocks
- Timeout
- Reentrant locks
- Locality-conscious locking
- Asymmetric locking
- ★ Concurrent readers
  - » centralized and queue-based *reader-writer locks*
  - » *sequence locks*
  - » *RCU*

Comparing spin locks

	TAS (w/ backoff)	ticket (w/ backoff)	MCS (original)	CLH (original)	MCS ("K42")	CLH ("K42")	Hem-lock
Fairness	–	+	+	+	+	+	+
Preemption tolerance	○	–	–	–	–	–	–
Scalability	○	○	+	+	+	+	+
Fast-path overhead	+	+	○	○	–	–	○
Interoperability	+	+	–	–	+	+	+
NRC-NUMA suitability	–	–	+	○	+	○	–
Space needs	n	$2n$	n	$2n$	$2n$	$2t + 3n$	$t + n$

t threads, n locks

Comparing spin locks

- Use ticket lock if the thread count is small (< 20?) and fairness is important or acceptable
- Use TATAS if need preemption tolerance is important and the thread count is very small (~single digits)
- Use a queue-based lock otherwise (MCS, CLH, or Hemlock, whichever performs best on your machine)
  - Use Henlock or a “K42” variant if you need a standard interface
- Consider various extensions, as needed

Barriers

- Centralized
- Tree
- Dissemination
- Fuzzy

Spin-based conditions

- We've already seen flags:

```
atomic<double> x
atomic<bool> f

thread 1:
    x := foo()
    f := true

thread 2:
    while ¬f;   // spin
    y := 1/x
```

- Several (scheduler-based) language mechanisms build on flags

- *Barriers* have both spin- and scheduler-based implementations

```
barrier b

in parallel for i ∈ 𝒯
    repeat
        // do i's portion of the work of a phase
        b.cycle()
    until terminating condition
```

- Spin-based versions can be centralized or distributed

Sense-reversing centralized barrier

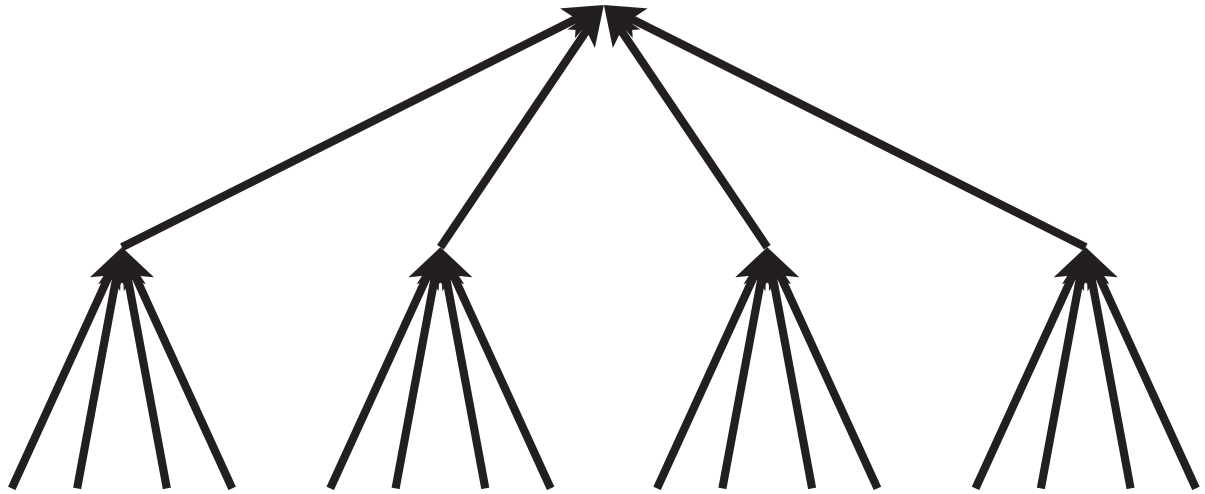
```
class barrier
    atomic<int> count := 0
    const int n := |𝒯|
    atomic<bool> sense := true
    bool local_sense[𝒯] := { true ... }
barrier.cycle():
    bool s := ¬local_sense[self]
    local_sense[self] := s        // each thread toggles its own sense
    if count.FAI() = n−1
        count.store(0)            // after FAI, last thread resets count
        sense.store(s)            // and then toggles global sense
    else
        while sense.load() ≠ s;   // spin
```

- This code is very easy to get wrong
  - » Threads reaching episode *k* must not interfere with threads that have not yet left episode *k*−1
- Takes linear time if all threads arrive at once

Tree barriers (multiple variants)

[Yew, Tseng, & Laurie; Hensgen, Finkel, & Manber; Lubachevsky; Mellor-Crummey & Scott]

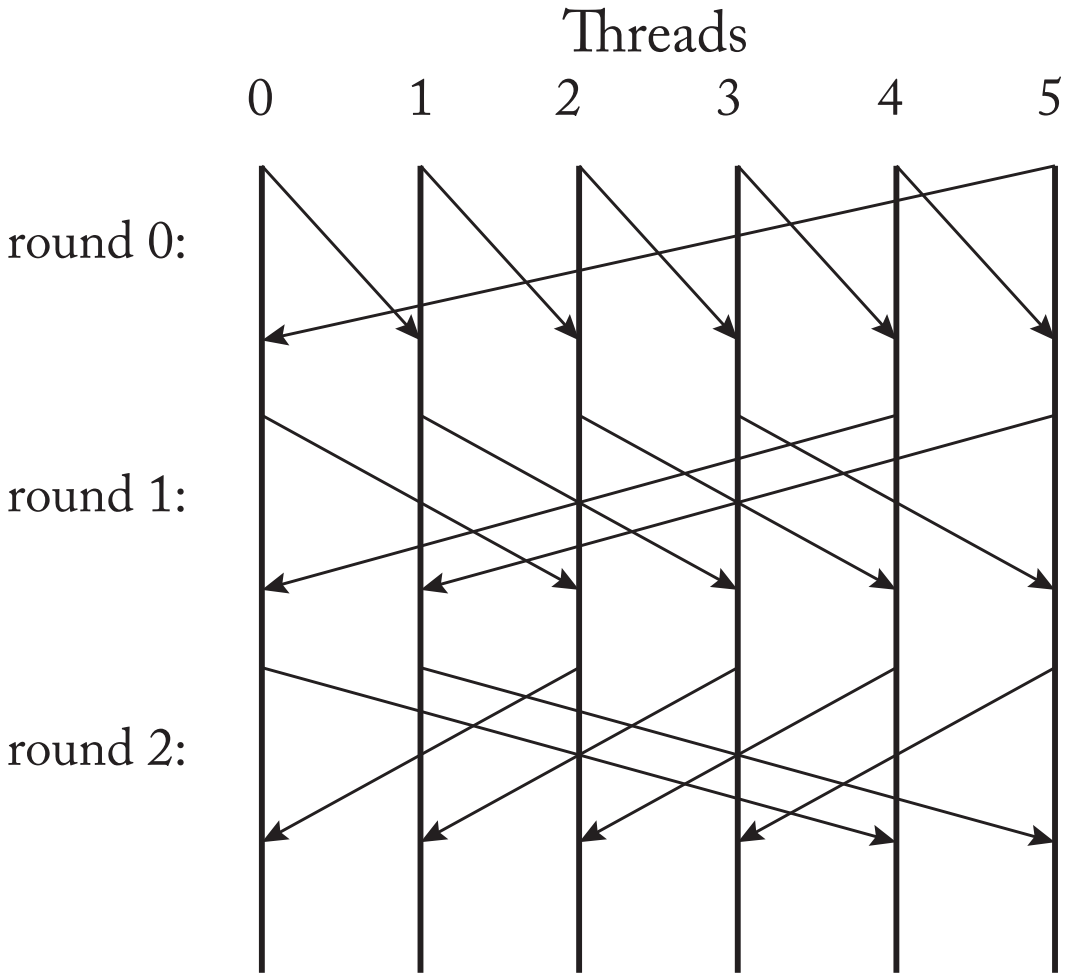
- Threads start at the leaves
- Once all threads have arrived at a node, one continues upward
- Winner at the root announces the barrier to others
  - Notification may propagate back down the tree
  - or may use a sense-reversing flag on a CC machine



The dissemination barrier

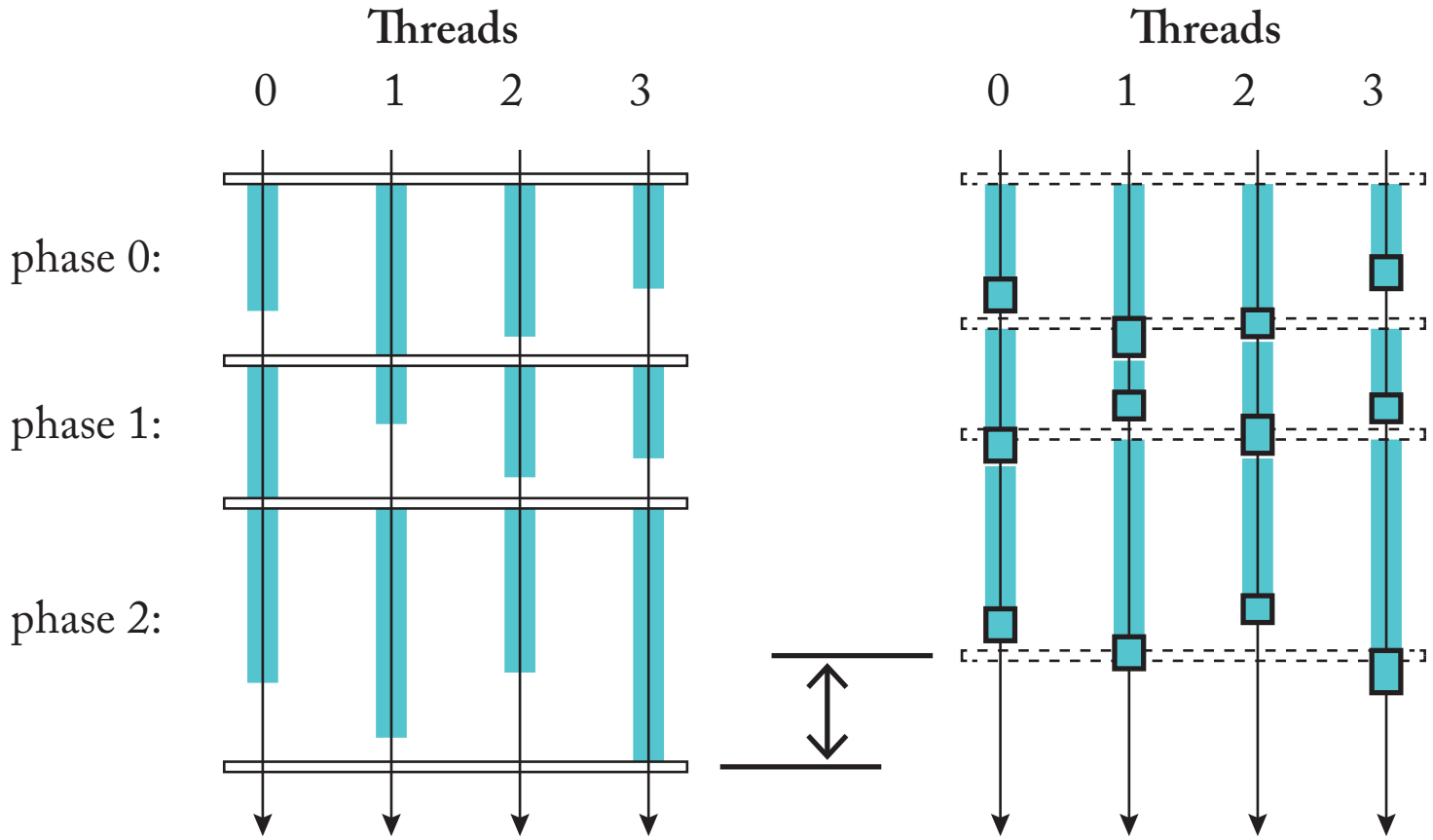
[Brooks; Hensgen, Finkel, & Manber]

- Hensgen et al. [1998]
- Log number of rounds
- In round i, thread t of n notifies thread $t + 2^i \bmod n$
- Total communication increases from $O(n)$ to $n\lceil \log_2 n \rceil$
- But latency drops by a factor of 2



Fuzzy barriers [Gupta 1989]

- Skew in arrival times leads to idle cores
- Sometimes idle time can be used for work — e.g., logging — that doesn't depend on work from the last cycle and that no one depends on for the next cycle
- Requires separate arrival and departure algorithms; doesn't work for dissemination or most tree barriers



Comparing barriers

	central	dissemination	static tree
space needs			
CC-NUMA	$t + 1$	$t + 2t \lceil \log_2 t \rceil$	$4t + 1$
NRC-NUMA			$(5 + d)t$
critical path length			
CC-NUMA	$t + 1$	$\lceil \log_2 t \rceil$	$\lceil \log_a t \rceil + 1$
NRC-NUMA	∞		$\lceil \log_a t \rceil + \lceil \log_d t \rceil$
total remote refs			
CC-NUMA	$t + 1 .. 2t$	$t \lceil \log_2 t \rceil$	t
NRC-NUMA	∞		$2t - 2$
fuzzy barrier suitability	+	–	–
tolerance of changes in n	+	–	–

t threads
arrival fan-in a
departure fan-out d

Comparing barriers

Use

- centralized when
  - # of participants is modest
  - # of participants isn't static
- dissemination when
  - latency matters, bandwidth is plentiful, and you don't have cache coherence
  - fuzziness isn't desired
- static tree OW

Reader-Writer locks

- Allow concurrent readers
  - various priority/fairness options
- Centralized v. queued implementations

- Sequence locks
- RCU

Central reader-preference RW lock

```
class rw_lock
    atomic<int> n := 0   // low bit: writer indicator; other bits: reader count
    const int WA_flag = 1;    const int RC_inc = 2
    const int base, limit, mult = ...  // tuning parameters

rw_lock.writer_acquire():
    int delay := base
    while ¬n.CAS(0, WA_flag)
        pause(delay)
        delay := min(delay × mult, limit)

rw_lock.writer_release():
    (void) n.FAA(−WA_flag)
```

```
rw_lock.reader_acquire():
    (void) n.FAA(RC_inc)
    while (n.load() & WA_flag) = 1;   // spin

rw_lock.reader_release():
    (void) n.FAA(−RC_inc)
```

RW variants

- reader-preference: reader can always join
- writer-preference: reader can join only if no writers waiting
- fair: readers wait for earlier-arriving writers only
- phase-fair: writers alternate with all waiting readers

- queued locks to eliminate remote spinning (fair version simplest)

Central writer-preference RW lock

```
class rw_lock
    atomic<⟨short, short, bool⟩> n := ⟨0, 0, false⟩
    // active readers, waiting writers, writer active?
    const int base, limit, mult = ...  // tuning parameters

rw_lock.reader_acquire():
    loop
        ⟨short ar, short ww, bool aw⟩ := n.load()
        if ww = 0 and aw = false
            if n.CAS(⟨ar, 0, false⟩, ⟨ar+1, 0, false⟩) break
            pause(ww × base)
```

```
rw_lock.reader_release():
    short ar, ww;  bool aw
    repeat    // fetch-and-phi
        ⟨ar, ww, aw⟩ := n.load()
    until n.CAS(⟨ar, ww, aw⟩,
        ⟨ar−1, ww, aw⟩)
```

```
rw_lock.writer_acquire():
    int delay := base
    loop
        ⟨short ar, short ww, bool aw⟩ := n.load()
        if aw = false and ar = 0   // nobody active
            if n.CAS(⟨ar, ww, aw⟩, ⟨ar, ww+1, aw⟩)
                // I'm registered as waiting
                loop                        // spin
                    ⟨ar, ww, aw⟩ := n.load()
                    if aw = false and ar = 0
                        if n.CAS(⟨ar, ww, false⟩, ⟨ar, ww−1, true⟩)
                            break outer loop
                    pause(delay);  delay := min(delay × mult, limit)
```

```
rw_lock.writer_release():
    short ar, wr;  bool aw
    repeat
        ⟨ar, ww, aw⟩ := n.load()
    until n.CAS(⟨ar, ww, aw⟩,
        ⟨ar, ww, false⟩)
```

Central fair RW lock

```
class rw_lock
    atomic<⟨short, short⟩> requests := ⟨0, 0⟩
    atomic<⟨short, short⟩> completions := ⟨0, 0⟩
    // top half of each word counts readers; bottom half writers
    const int base = ...   // tuning parameter
```

- Arriving threads increment the appropriate request count and snapshot the other
- Readers wait for the writer completion count to match the snapshotted writer request count; writers wait for both completion counts to match the snapshot
- Departing threads increment the appropriate completion count

(Actual code is kinda long)

Queued RW locks

- Reader-preference, writer-preference, and fair variants [Mellor-Crummey and Scott, 1991]
  - » Central counter can be a target of contention (though never spinning)
- Fair variant w/ no central counter [Krieger 1993; Dice 2013]
  - » Doubly linked list w/ mutex (TATAS) lock in each qnode
  - » Arbitrary delays necessitate SMR
- Phase-fair variant [Brandendurg and Anderson, 2010]

- Asymmetric variant: $|\mathcal{T}|$ "reader locks" and one "writer lock"
  - » Used for certain operations in the Linux kernel

Sequence locks [Lameter 2005]

- Use *speculation* to avoid updating the read lock:

```
repeat
    int s := SL.reader_start()
    // critical section
until SL.reader_validate(s)
```

  - mis-speculation needs to be harmless
  - tolerate races with memory_order_relaxed
  - canonical use case: consistent multi-word reads

- Allow readers to *upgrade* to writing:

```
loop
    int s := SL.reader_start()
    ...
    if unlikely_condition
        if ¬SL.become_writer(s) continue
        ...
        SL.writer_release()
        break
    else
        ...
        if SL.reader_validate(s) break
```

Sequence lock implementation

```
class seqlock
    atomic<int> n := 0

int seqlock.reader_start():
    int seq
    repeat      // spin until even
        seq := n.load()
    until seq ≡ 0 mod 2
    return seq

bool seqlock.reader_validate(int seq):
    return (n.load() = seq)
```

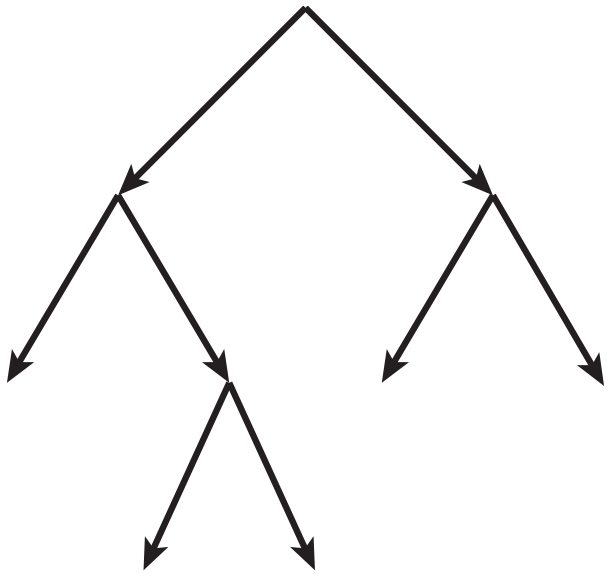
```
bool seqlock.become_writer(int seq):
    return n.CAS(seq, seq+1)

seqlock.writer_acquire():
    int seq
    repeat     // spin
        seq := n.load()
    until seq ≡ 0 mod 2
        and n.CAS(seq, seq+1)

seqlock.writer_release():
    int seq := n.load()
    n.store(seq+1)
```

Fast userspace mutex (futex) mechanism

- Provided by Linux and several other OSes
- Allows user-space-only synchronization (even across process boundaries) when waiting is not required
- Essentially, an atomic compare-and-block
  - » if ⟨location has expected value⟩ yield
  - » separate call to wake one or more waiters (if any)
- Kernel finds metadata via hash(paddr) lookup
- Very tricky to use correctly (Yifan Zhu has experience 🙂)

RCU

- Advanced technique for highly read-heavy workloads
  - » reads as close to free as possible; writes may be very expensive
  - » mostly used in systems software; correctness can be very subtle
- Wide variety of implementations; most have
  - » no shared updates by readers
  - » single-pointer updates
  - » unidirectional data traversal
  - » delayed reclamation based on *grace periods*
- Many variants; see SMS or papers



Grace periods

- In a nonpreemptive OS kernel: wait for a (voluntary) context switch on every core.
  - » e.g., migrate to every core in turn
- In user space, maintain a global counter *C* and an array *S* of per-thread counters
  - » increment *C* at the end of each write op
  - » set *S*[*i*] := C at the beginning of each read op (partial violation of the rules…); *S*[*i*] := 0 at the end
  - » grace period ends when *S*[*i*] = 0 or *S*[*i*] > end-of-write-op value ∀ *i*

Memory ordering

- Need ||R store to S at beginning of read op, R|| store at end.
- Need R|| ordering when reading a pointer that might have been updated by a writer.
- Can avoid the ||R store (the expensive one) by kicking all readers (as in asymmetric locking) at the end of each write op: signal handler handshakes w/ writer to
  - » make each reader's S update visible to the writer
  - » make the writer's updates visible to each reader
- Need seq-cst stores on writer pointer updates