Ticket lock: A fair lock

```
number = next = 0      // all threads share
Acquire( ) {
   int ticket = FetchAndAdd(number, 1)
   while (ticket != next)
       ;
}
Release( ) {
   next++
}
```

Simple algorithm, but causes significant network traffic because of polling of common location

Array-based queuing lock code

L: shared ptr to a record w/ array and index

Array element values either "wait" or "go"

```
Acquire (L, ref place)  {
  place = f&i(L→nextslot) //overflow?
  place = place mod P
  while (L→slots[place] == wait)
      ;
  L→slots[place] := wait;
}
```

```
Release(L, place) {
   next = (place+1) mod P;
   L→slots[next] = go;
}
```

Disadvantage: Linear space

MCS lock code

I: ptr to a node with boolean and next ptr

L: shared ptr to tail of list

```
Acquire (L, I)  {
  I→next = null
  pred = f&s(L, I)
  if (pred != null)  {
    I→locked = true
    pred→next = I
    while (I→locked)
      ;
  }
}
```

```
Release(L, I) {
  if (I→next == null) {
    if c&s(L, I, null)
      return
    while (I → next == null)
      ;
  }
  I→next→locked = false;
}
```

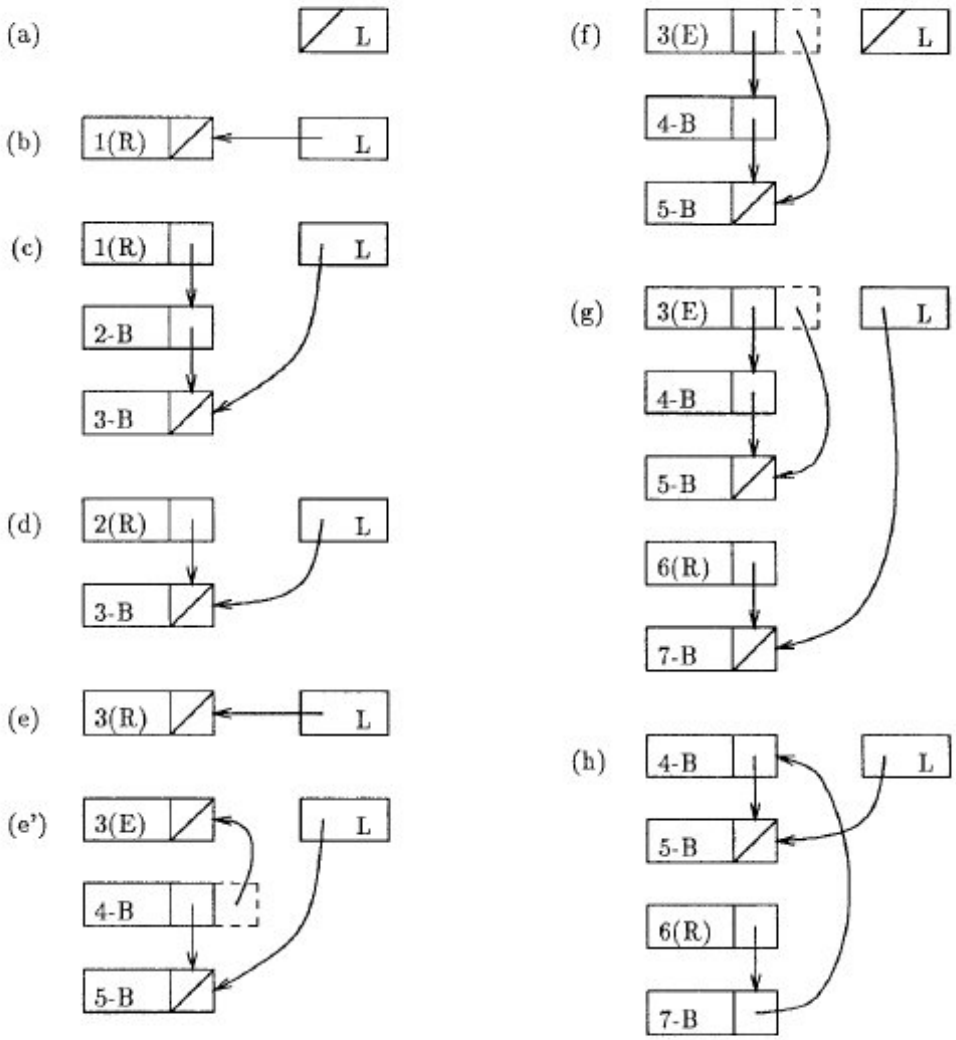

Fig. 6. Pictorial example of MCS locking protocol in the presence of competition.

What if there is no Compare-and-Swap?

- Makes release code much more complicated
  - Essentially, without C&S, inspection and update of the tail pointer cannot occur atomically
  - Can have situation in which we believe the last thread is releasing, only after that is determined, Acquires are performed
    - Leads to L being set to NULL, but extra nodes on list; this leads to the potentially disastrous situation that a new Acquire jumps ahead of the previous Acquires along with the previous Acquires being “lost”
    - So, we need to grab the old end of list, and grab the next element after releaser, and make the list consistent

Sense-reversing centralized barrier

```
shared count = P, sense = true
Barrier( )
  static localSense = true
  localSense = not localSense
  if FetchAndAdd(count, -1) == 1   // returns old value
     count = P
     sense = localSense
  else
     while (sense != localSense)
        ;
```

Sense-reversing centralized barrier

- Problem: spinning on a global flag (*sense*)
  - On a multicore machine without broadcast-based cache coherence, significant network traffic
    - Many-core machines may not have a broadcast
  - Contention at atomic instruction?

Combining tree barrier

```
type node = record
    k : integer              // fan-in of this node
    count : integer          // initialized to k
    locksense : Boolean      // initially false
    parent : ^node           // pointer to parent node; nil if root

shared nodes : array [0..P-1] of node
    // each element of nodes allocated in a different memory module or cache line
processor private sense : Boolean  := true
processor private mynode : ^node    // my group's leaf in the combining tree

procedure combining_barrier
    combining_barrier_aux (mynode)      // join the barrier
    sense := not sense                  // for next barrier

procedure combining_barrier_aux (nodepointer: ^node)
    with nodepointer^ do
        if fetch_and_decrement (&count) = 1     // last one to reach this node
            if parent != nil
                combining_barrier_aux (parent)
            count := k                          // prepare for next barrier
            locksense := not locksense          // release waiting processors
        repeat until locksense = sense
```

Fig. 9. A software combining tree barrier with optimized wakeup.

Tournament (tree) barrier

- Advantages: know ahead of time who partner is, no atomic instructions
  - Similar to dissemination barrier
  - Form tree, where number of leaves is P
  - Assign winners and losers all the way up the tree
    - Loser exits and spins on global flag waiting for root of tree to set the flag
    - E.g., at leaves, all even numbered threads “win”; at next level up, all threads divisible by 4 “win”, etc.
    - Uses extra storage (one array row per round), but could use “count up” trick from symmetric barrier

Performance

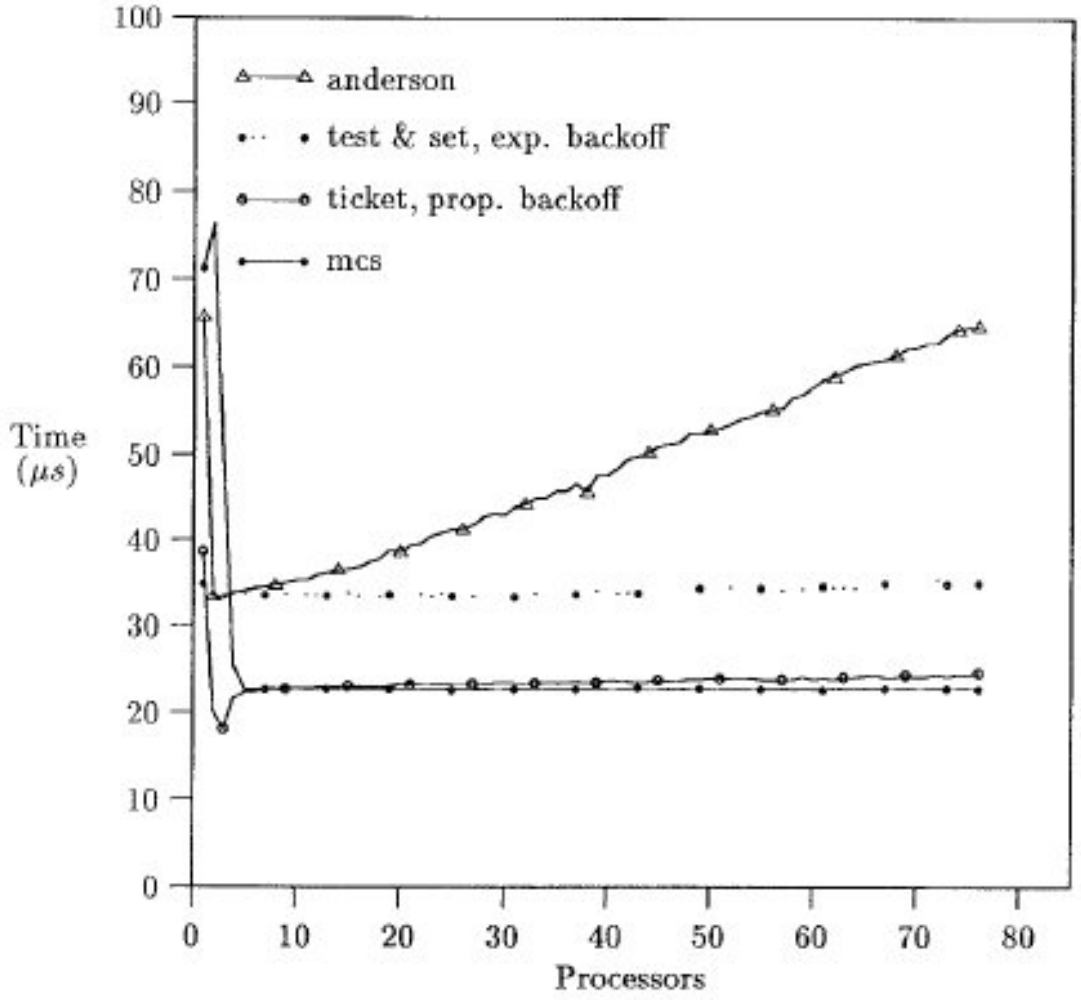

Fig. 16. Performance of selected spin locks on the Butterfly (empty critical section)

Performance

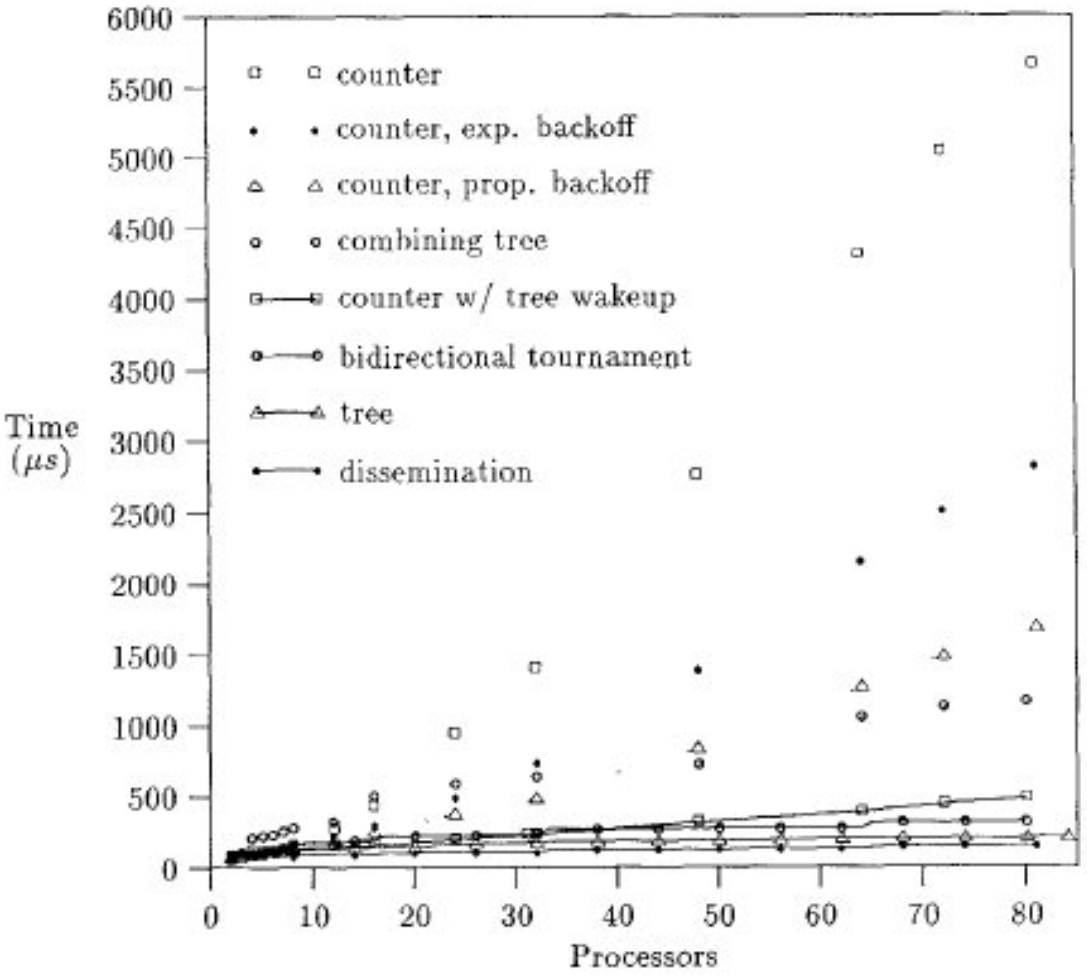

Fig. 19 Performance of barriers on the Butterfly.

Performance

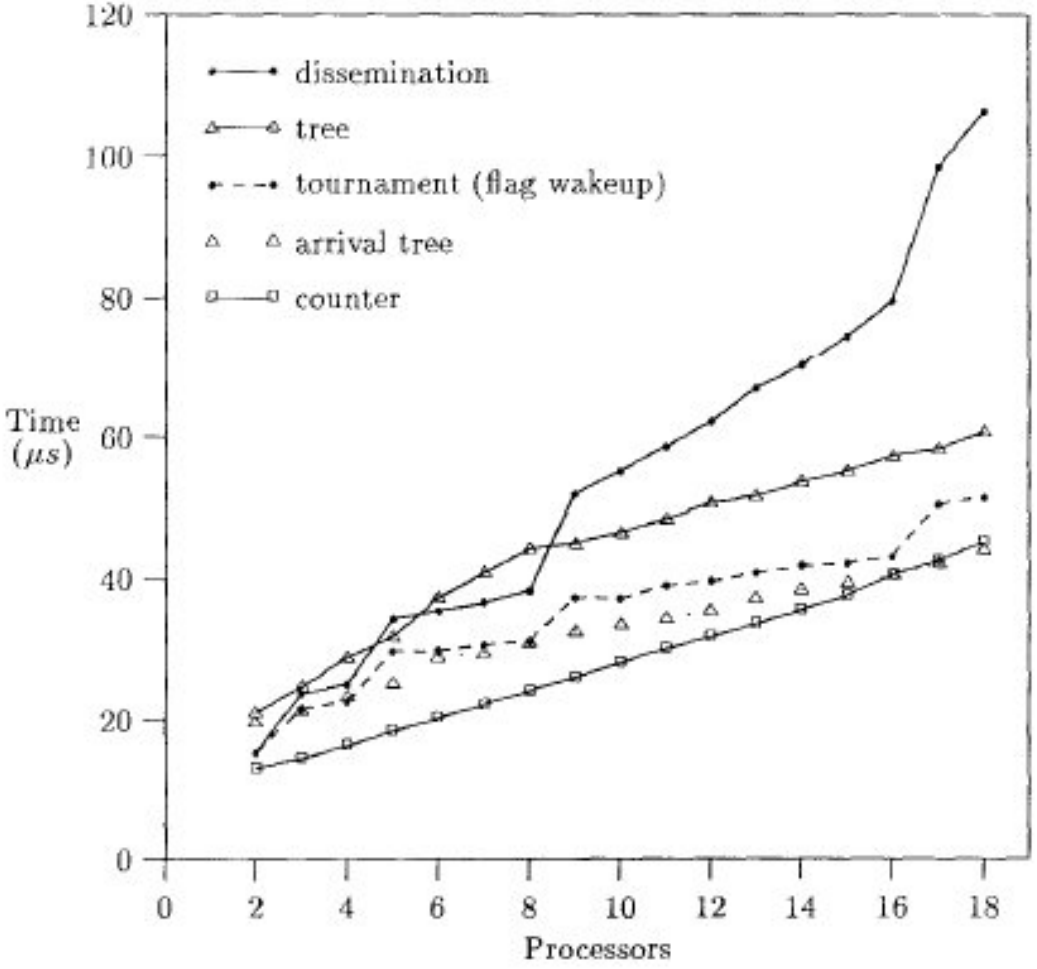

Fig. 21. Performance of barriers on the Symmetry