CS758
Synchronization

Shared Memory Primitives

Shared Memory Primitives

- **Create thread**
  - Ask operating system to create a new “thread”
  - Threads starts at specified function (via function pointer)
- Memory regions
  - Shared: globals and heap
  - Per-thread: stack
- Primitive memory operations
  - Loads & stores
  - Word-sized operations are atomic (if “aligned”)
- **Various “atomic” memory instructions**
  - Swap, compare-and-swap, test-and-set, atomic add, etc.
  - Perform a load/store pair that is guaranteed not to interrupted

Thread Creation

- Varies from operating system to operating system
  - POSIX threads (P-threads) is a widely supported standard (C/C++)
  - Lots of other stuff in P-threads we're **not** using
    - Why? really design for single-core OS-style concurrency
- pthread_create(id, attr, start_func, arg)
  - "id" is pointer to a "pthread_t" object
  - We're going to ignore "attr"
  - "start_func" is a function pointer
  - "arg" is a void *, can pass pointer to anything (ah, C…)

Thread Creation Code Example I

- **Caveat: C-style pseudo code**
  - **Examples like wont work without modification**
- pthread_create(id, attr, start_func, arg)
  - “start_func” is a function pointer
  - “arg” is a void *, can pass pointer to anything (ah, C…)

```
void* my_start(void *ptr)
{
    printf(“hello world\n”);
    return NULL;  // terminates thread
}
int main()
{
    pthread_t id;
    int error = pthread_create(&id, NULL, my_start, NULL);
    if (error) { … }
    pthread_join(id, NULL);
    return 0;
}
```

Thread Creation Code Example II

```
void* my_start(void *ptr)
{
  int* tid_ptr = (int *) ptr;     // cast from void*
  printf("hello world: %d\n", *tid_ptr);
  return NULL;  // terminates thread
}

void spawn(int num_threads)
{
  pthread_t* id = new pthread_t[num_threads];
  int* tid = new int[num_threads];

  for (int i=1; i<num_threads; i++) {
    tid[i] = i;
    void *ptr = (void *) &i;
    int error = pthread_create(&id[i], NULL, my_start, &tid[i]);
    if (error) { … }
  }
  tid[0] = 0;
  my_start(&tid[i]);   // "start" thread zero

  for (int i=1; i<num_threads; i++) {
    pthread_join(id[i], NULL);
  }
}
```

Compare and Swap (CAS)

- Atomic Compare and Swap (CAS)
  - Basic and universal atomic operations
  - Can be used to create all the other variants of atomic operations
  - Supported as instruction on both x86 and SPARC

- Compare_and_swap(address, test_value, new_value):
  - Load [Address] -> old_value
  - if (old_value == test_value):
    - Store [Address] <- new_value
  - Return old_value

- Can be included in C/C++ code using "inline assembly"
  - Becomes a utility function

Inline Assembly for Atomic Operations

- x86 inline assembly for CAS
  - From Intel's TBB source code `machine/linux_intel64.h`

```
static inline int64 compare_and_swap(volatile void *ptr,
                                     int64 test_value,
                                     int64 new_value)
{
  int64 result;
  __asm__
  __volatile__("lock\ncmpxchgq %2,%1"
            : "=a"(result), "=m"(*(int64 *)ptr)
            : "q"(new_value), "0"(test_value), "m"(*(int64 *)ptr)
            : "memory");
  return result;
}
```

- Black magic
  - Use of **`volatile`** keyword disable compiler memory optimizations

Fetch-and-Add (Atomic Add)

- Another useful “atomic” operation
- atomic_add(address, value)
  - Load [address] -> temp
  - temp2 = temp + value
  - store [address] <- temp2
- Some ISAs support this as a primitive operation (x86)
- Or, can be synthesized out of CAS:

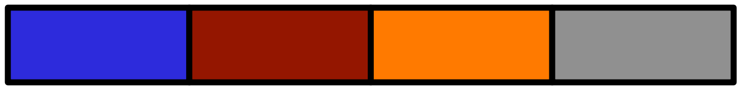
```
int atomic_add(int* ptr, int value)
{
     while(true) {
       int original_value = *ptr;
       int new_value = original_value + value;
       int old = compare_and_swap(ptr, original_value, new_value);
       if (old == original_value) {
         return old;     // swap succeeded
       }
     }
}
```

Thread Local Storage (TLS)

- Sometimes having a non-shared global variable is useful
  - A per-thread copy of a variable
- Manual approach:
  - Definition: **`int accumulator[NUM_THREADS];`**
  - Use: **`accumulator[thread_id]++;`**
  - Limited to **`NUM_THREADS`**, need to pass **`thread_id`** around false sharing
- Compiler supported:
  - Definition: **`__thread int accumulator = 0;`**
  - Use: **`accumulator++;`**
  - Implemented as a per-thread "globals" space
    - Accessed efficiently via **`%gs`** segment register on x86-64
  - More info: http://people.redhat.com/drepper/tls.pdf

Simple Parallel Work Decomposition

Static Work Distribution



- Sequential code

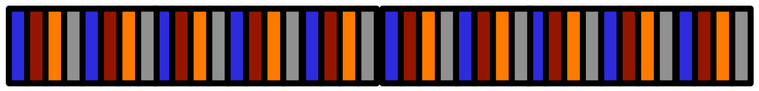
```
for (int i=0; i<SIZE; i++):
   calculate(i, …, …, …)
```

- Parallel code, for each thread:

```
void each_thread(int thread_id):
   segment_size = SIZE / number_of_threads
   assert(SIZE % number_of_threads == 0)
   my_start = thread_id * segment_size
   my_end = my_start + segment_size
   for (int i=my_start; i<my_end; i++)
      calculate(i, …, …, …)
```

- Hey, its a parallel program!

Dynamic Work Distribution



- Sequential code

```
for (int i=0; i<SIZE; i++):
   calculate(i, …, …, …)
```

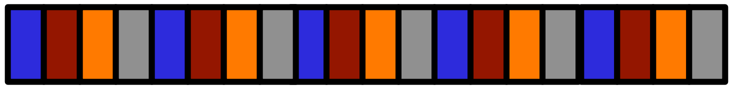
- Parallel code, for each thread:

```
int counter = 0    // global variable
void each_thread(int thread_id):
   while (true)
      int i = atomic_add(&counter, 1)
      if (i >= SIZE)
         return
      calculate(i, …, …, …)
```

- Dynamic load balancing, but high overhead

Coarse-Grain Dynamic Work Distribution

- Parallel code, for each thread:



```
int num_segments = SIZE / GRAIN_SIZE
int counter = 0     // global variable
void each_thread(int thread_id):
   while (true)
      int i = atomic_add(&counter, 1)
      if (i >= num_segments)
         return
      my_start = i * GRAIN_SIZE
      my_end = my_start + GRAIN_SIZE
      for (int j=my_start; j<my_end; j++)
         calculate(j, …, …, …)
```

- Dynamic load balancing with lower (adjustable) overhead

Barriers

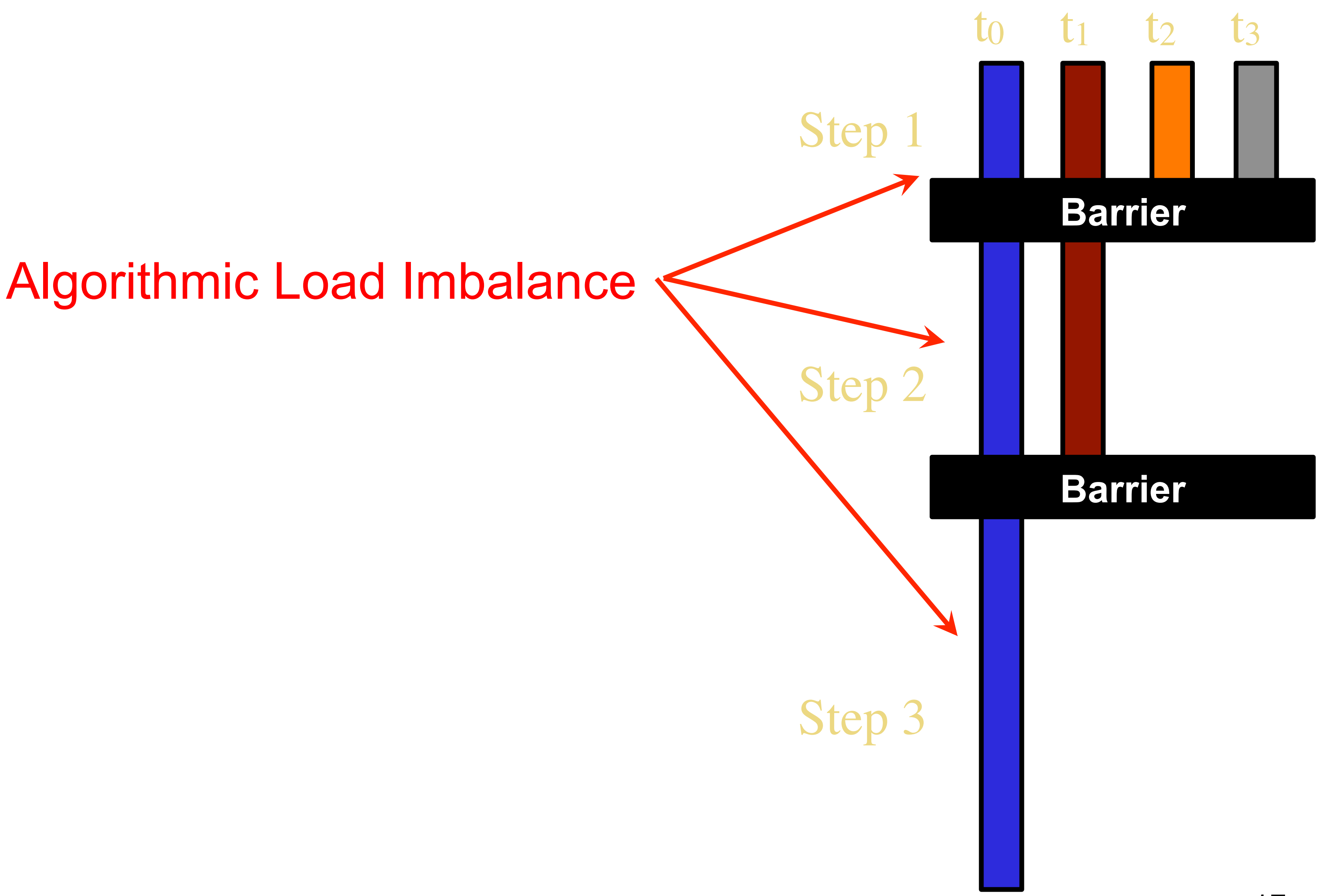
Common Parallel Idiom: Barriers

- Physics simulation computation
  - Divide up each timestep computation into N independent pieces
  - Each timestep: compute independently, synchronize
- Example: each thread executes:

```
segment_size = total_particles / number_of_threads
my_start_particle = thread_id * segment_size
my_end_particle =  my_start_particle + segment_size - 1
for (timestep = 0; timestep += delta; timestep < stop_time):
    calculate_forces(t, my_start_particle, my_end_particle)
    barrier()
    update_locations(t, my_start_particle, my_end_particle)
    barrier()
```

- Barrier? All threads wait until all threads have reached it

Example: Barrier-Based Merge Sort



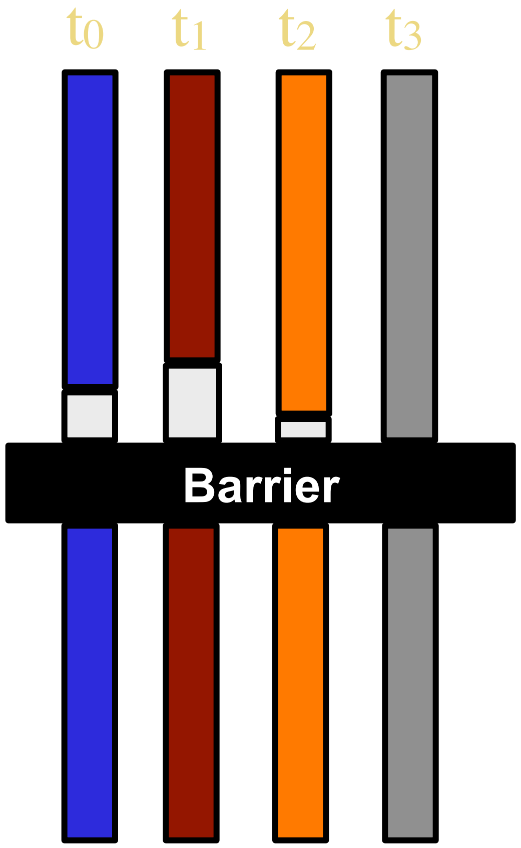
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it
- Strawman implementation (**wrong!**)

```
global (shared) count : integer := P

procedure central_barrier
  if fetch_and_decrement(&count) == 1
    count := P
  else
    repeat until count == P
```

- What is wrong with the above code?



Sense-Reversing Barrier

- Correct barrier implementation:

```
global (shared) count : integer := P
global (shared) sense : Boolean := true
local (private) local_sense : Boolean := true

procedure central_barrier
  // each processor toggles its own sense
  local_sense := !local_sense
  if fetch_and_decrement(&count) == 1
    count := P
    // last processor toggles global sense
    sense := local_sense
  else
    repeat until sense == local_sense
```

- Single counter makes this a “centralized” barrier

Other Barrier Implementations

- Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - O(n) if threads arrive simultaneously, slow for lots of processors
- Combining Tree Barrier
  - Build a $\log_k(n)$ height tree of counters (one per cache block)
  - Each thread coordinates with **k** other threads (by thread id)
  - Last of the **k** processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - O(log n) in best case
- Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release
- Also, hardware support possible (e.g., Cray T3E)

Barrier Performance (from 1991)

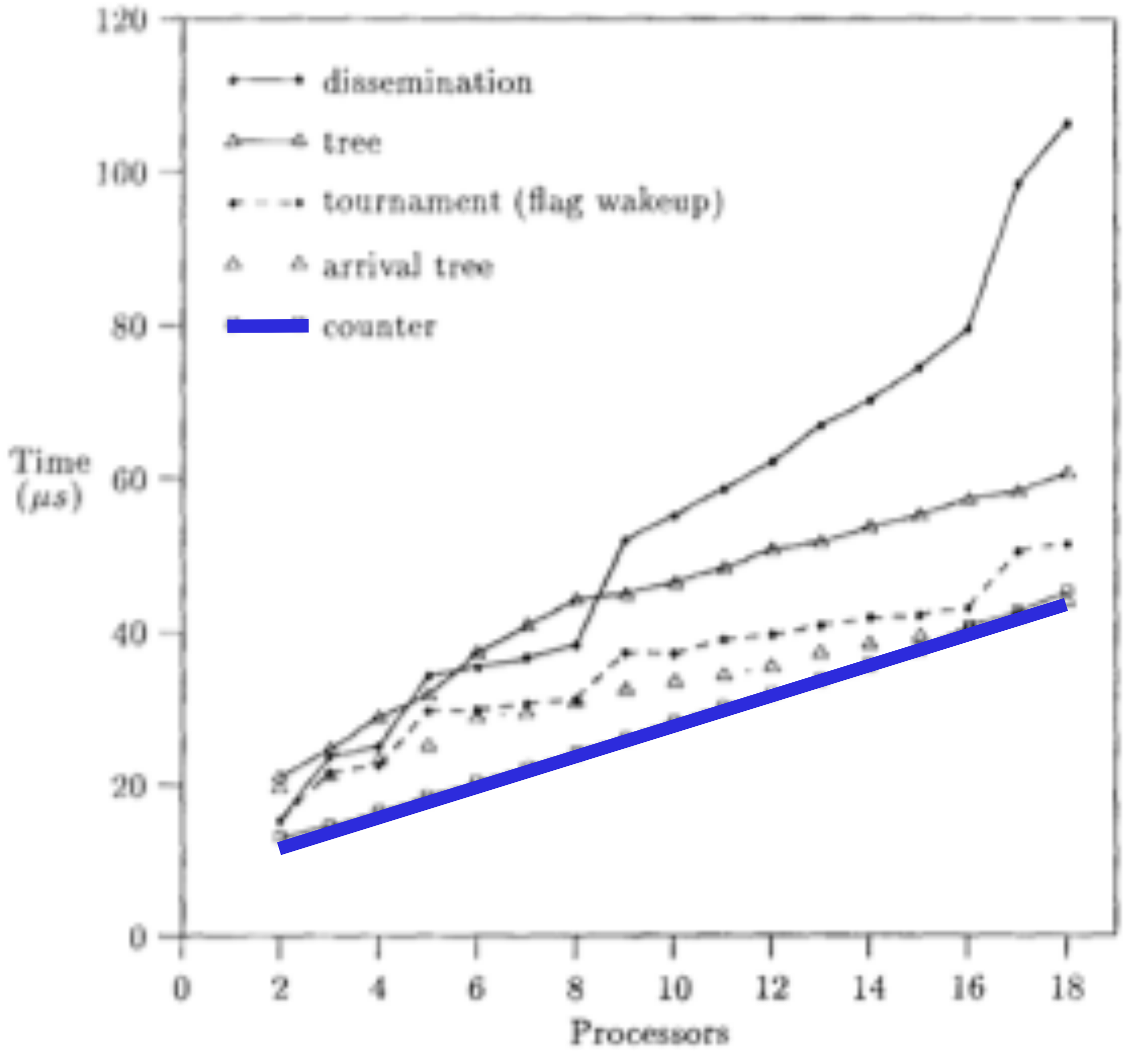


Fig. 21. Performance of barriers on the Symmetry

From Mellor-Crummey & Scott, ACM TOCS, 1991

Locks

Common Parallel Idiom: Locking

- Protecting a shared data structure
- Example: parallel tree walk, apply f() to each node
    - Global "set" object, initialized with pointer to root of tree

```
Each thread, while (true):
    node* next = set.remove()
    if next == NULL: return     // terminate thread
    func(code->value)     // computationally intense function
    if (next->right != NULL):
        set.insert(next->right)
    if (next->left != NULL):
        set.insert(next->left)
```

- How do we protect the "set" data structure?
    - Also, to perform well, what element should it "remove" each step?

Common Parallel Idiom: Locking

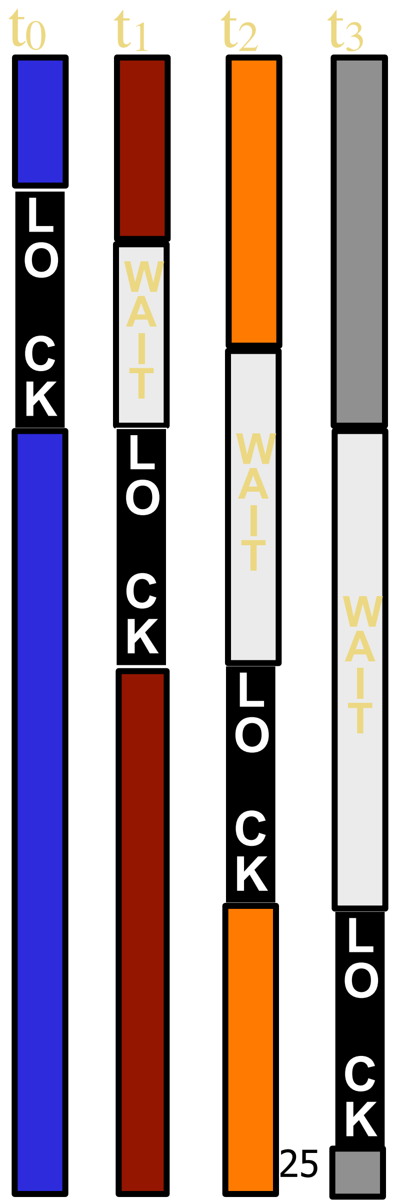
- Parallel tree walk, apply f() to each node
  - Global “set” object, initialized with pointer to root of tree
  - Each thread, while (true):

```
acquire(set.lock_ptr)
node* next = set.pop()
release(set.lock_ptr)
if next == NULL:
    return    // terminate thread
func(node->value)    // computationally intense
acquire(set.lock_ptr)
if (next->right != NULL)
    set.insert(next->right)
if (next->left != NULL)
    set.insert(next->left)
release(set.lock_ptr)
```

Put lock/unlock into **pop**() method?

Put lock/unlock into **insert**() method?

Lock-Based Mutual Exclusion



Simple Boolean Spin Locks

- Simplest lock:
  - Single variable, two states: **locked**, **unlocked**
  - When unlocked: atomically transition from unlocked to locked
  - When locked: keep checking (spin) until the lock is unlocked
- Busy waiting versus “blocking”
  - In a multicore, **busy wait** for other thread to release lock
    - Likely to happen soon, assuming critical sections are small
    - Likely nothing “better” for the processor to do anyway
  - In a single processor, if trying to acquire a held lock, **block**
    - The only sensible option is to tell the O.S. to context switch
    - O.S. knows not to reschedule thread until lock is released
  - Blocking has high overhead (O.S. call)
    - IMHO, rarely makes sense in multicore (parallel) programs

Test-and-Set Spin Lock (T&S)

- Lock is "acquire", Unlock is "release"
- acquire(lock_ptr):

```
While (true):
    // Perform "test-and-set"
    old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
    if (old == UNLOCKED):
        break   // lock acquired!
    // keep spinning, back to top of while loop
```

- release(lock_ptr):

```
Store[lock_ptr] <- UNLOCKED
```

- Performance problem
  - CAS is both a read and write, spinning causes lots of invalidations

Test-and-Test-and-Set Spin Lock (TTS)

- acquire(lock_ptr):

```
While (true):
    // Perform "test"
    Load [lock_ptr] -> original_value
    if (original_value == UNLOCKED):
        // Perform "test-and-set"
        old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
        if (old == UNLOCKED):
            break   // lock acquired!
    // keep spinning, back to top of while loop
```

- release(lock_ptr):

  Store[lock_ptr] <- UNLOCKED

- Now "spinning" is read-only, on local cached copy

TTS Lock Performance Issues

- Performance issues remain
    - Every time the lock is released…
    - All the processors load it, and likely try to CAS the block
    - Causes a storm of coherence traffic, clogs things up badly
- One solution: backoff
    - Instead of spinning constantly, check less frequently
    - Exponential backoff works well in practice
- Another problem with spinning
    - Processors can spin really fast, starve threads on the same core!
    - Solution: x86 adds a "PAUSE" instruction
        - Tells processor to suspend the thread for a short time
- (Un)fairness

Ticket Locks

- To ensure fairness and reduce coherence storms
- Locks have two counters: next_ticket, now_serving
  - Deli counter
- acquire(lock_ptr):

```
my_ticket = fetch_and_increment(lock_ptr->next_ticket)
while(lock_ptr->now_serving != my_ticket); // spin
```

- release(lock_ptr):

```
lock_ptr->now_serving = lock_ptr->now_serving + 1
```

  (Just a normal store, not an atomic operation, why?)

- Summary of operation
  - To "get in line" to acquire the lock, CAS on next_ticket
  - Spin on now_serving

Ticket Locks

- Properties
  - Less of a “thundering herd” coherence storm problem
    - To acquire, only need to read new value of now_serving
  - No CAS on critical path of lock handoff
    - Just a non-atomic store
  - FIFO order (fair)
    - Good, but only if the O.S. hasn’t swapped out any threads!
- Padding
  - Allocate now_serving and next_ticket on different cache blocks
    - struct { int now_serving; char pad[60]; int next_ticket; } …
  - Two locations reduces interference
- Proportional backoff
  - Estimate of wait time: (my_ticket - now_serving) * average hold time

Array-Based Queue Locks

- Why not give each waiter its own location to spin on?
  - Avoid coherence storms altogether!
- Idea: “slot” array of size N: “go ahead” or “must wait”
    - Initialize first slot to “go ahead”, all others to “must wait”
    - Padded one slot per cache block,
  - Keep a “next slot” counter (similar to “next_ticket” counter)
- Acquire: “get in line”
  - my_slot = (atomic increment of “next slot” counter) mod N
  - Spin while slots[my_slot] contains “must_wait”
  - Reset slots[my_slot] to “must wait”
- Release: “unblock next in line”
  - Set slots[my_slot+1 mod N] to “go ahead”

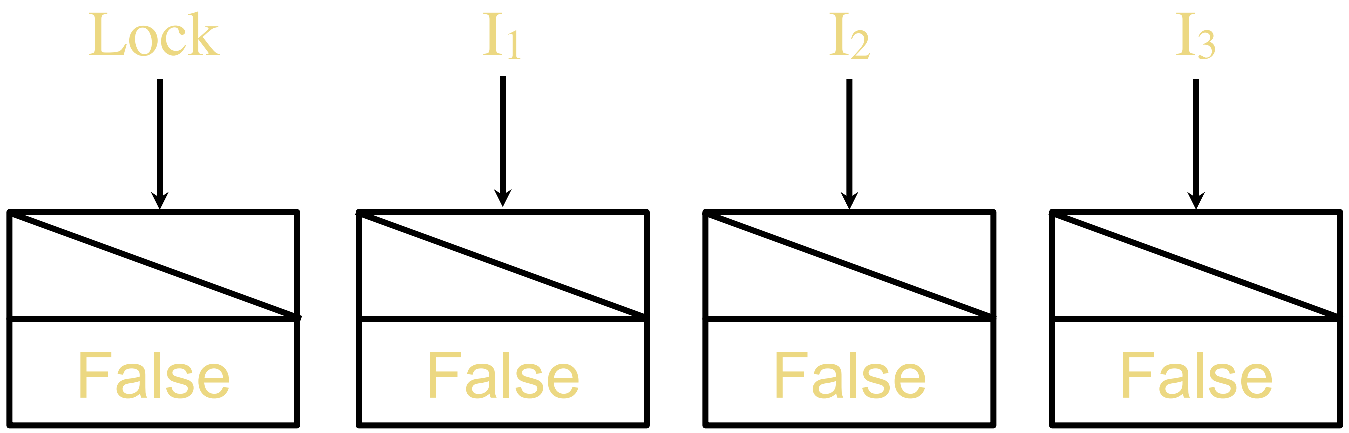
Array-Based Queue Locks

- Variants: Anderson 1990, Graunke and Thakkar 1990
- Desirable properties
  - Threads spin on dedicated location
    - Just two coherence misses per handoff
    - Traffic independent of number of waiters
  - FIFO & fair (same as ticket lock)
- Undesirable properties
  - Higher uncontended overhead than a TTS lock
  - Storage O(N) for each lock
    - 128 threads at 64B padding: 8KBs per lock!
    - What if N isn't known at start?
- List-based locks address the O(N) storage problem
  - Several variants of list-based locks: MCS 1991, CLH 1993/1994

List-Based Queue Locks (CLH lock)

- Link list node:
  - Previous node pointer
  - bool must_wait
- A “lock” is a pointer to a link list node
  - Each thread has a local pointer to a node “I”
- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (prev->must_wait) **// spin**
- Release(L):
  - pred = I->prev
  - I->must_wait = false **// wakeup next waiter, if any**
  - I = pred **// take pred’s node**

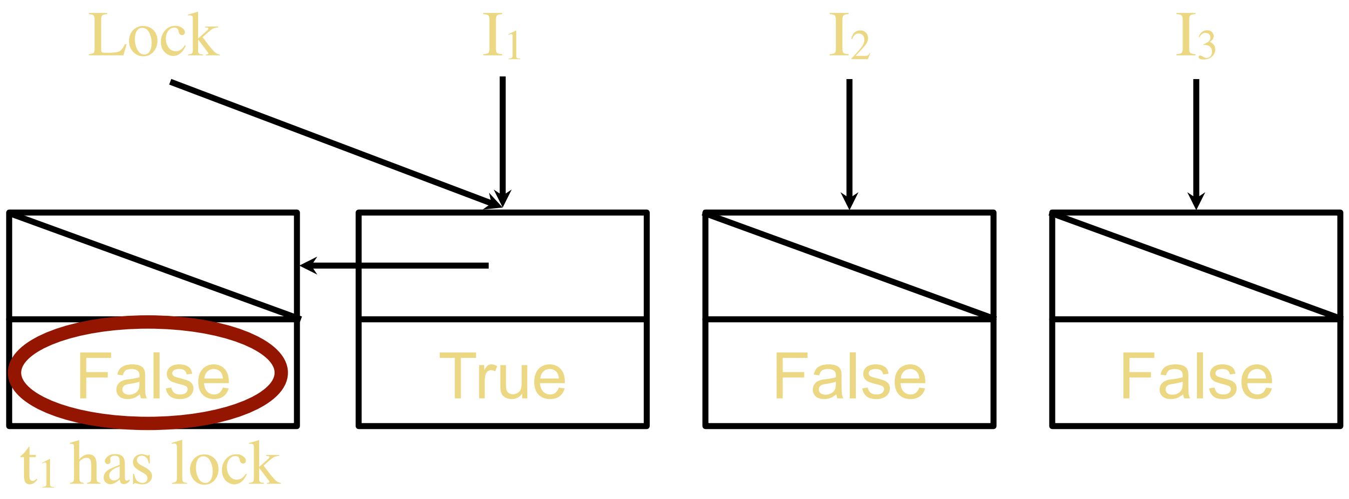
Queue Locks Example: Time 0



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release (L):
  - pred = I->prev
  - I->must_wait = false

Queue Locks Example: Time 1

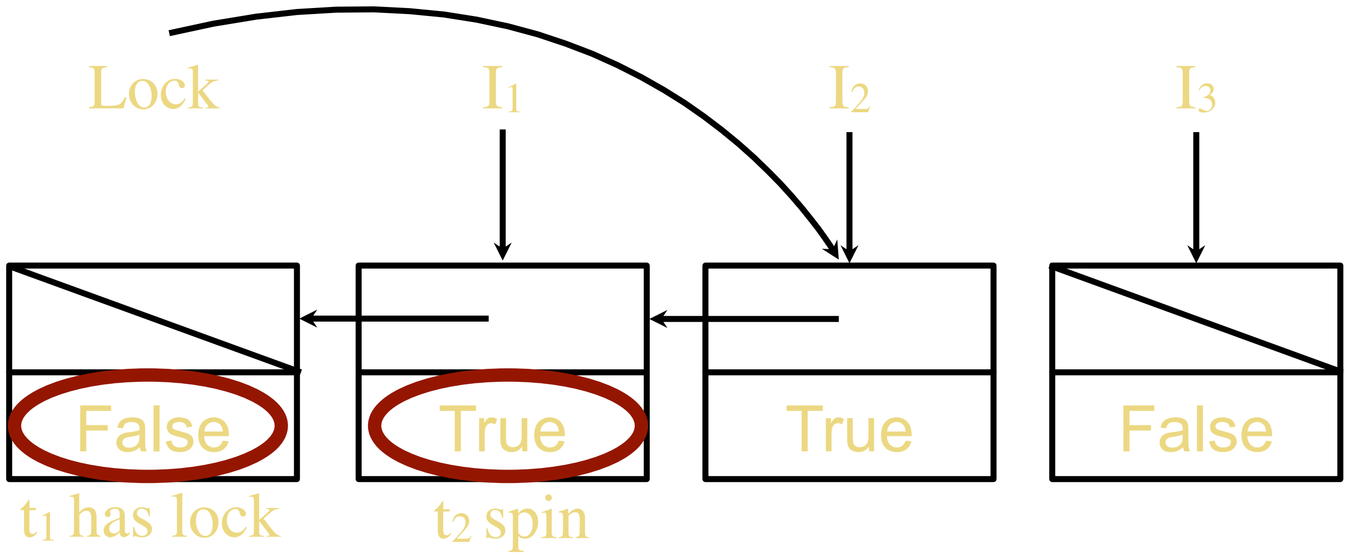
- t_1: Acquire(L)



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release(L):
  - pred = I->prev
  - I->must_wait = false

Queue Locks Example: Time 2

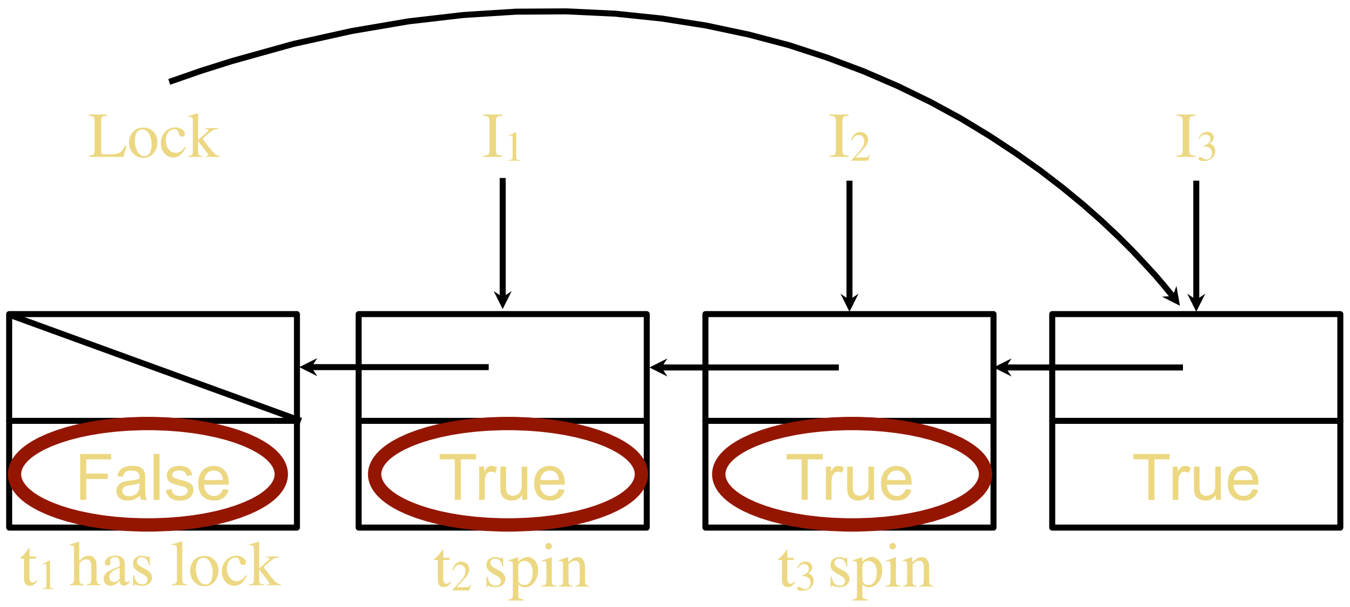
- t_1: Acquire(L)
- t_2: Acquire(L)



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release(L):
  - pred = I->prev
  - I->must_wait = false

Queue Locks Example: Time 3

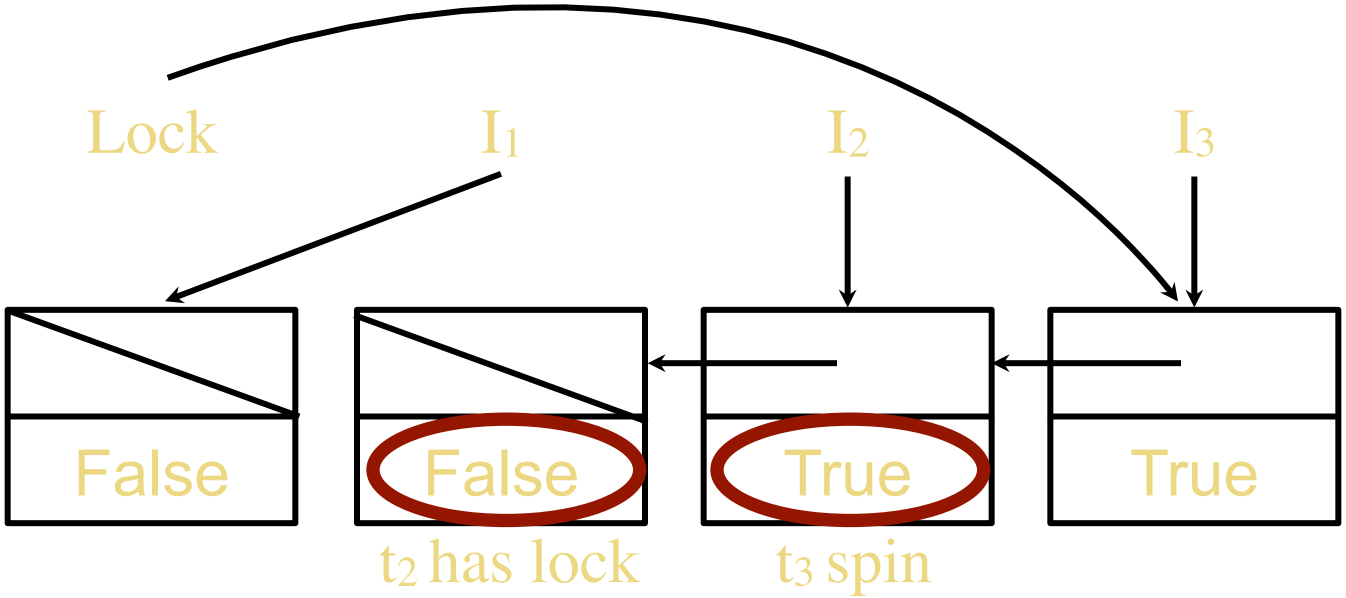
- t_1: Acquire(L)
- t_2: Acquire(L)
- t_3: Acquire(L)



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release(L):
  - pred = I->prev
  - I->must_wait = false

Queue Locks Example: Time 4

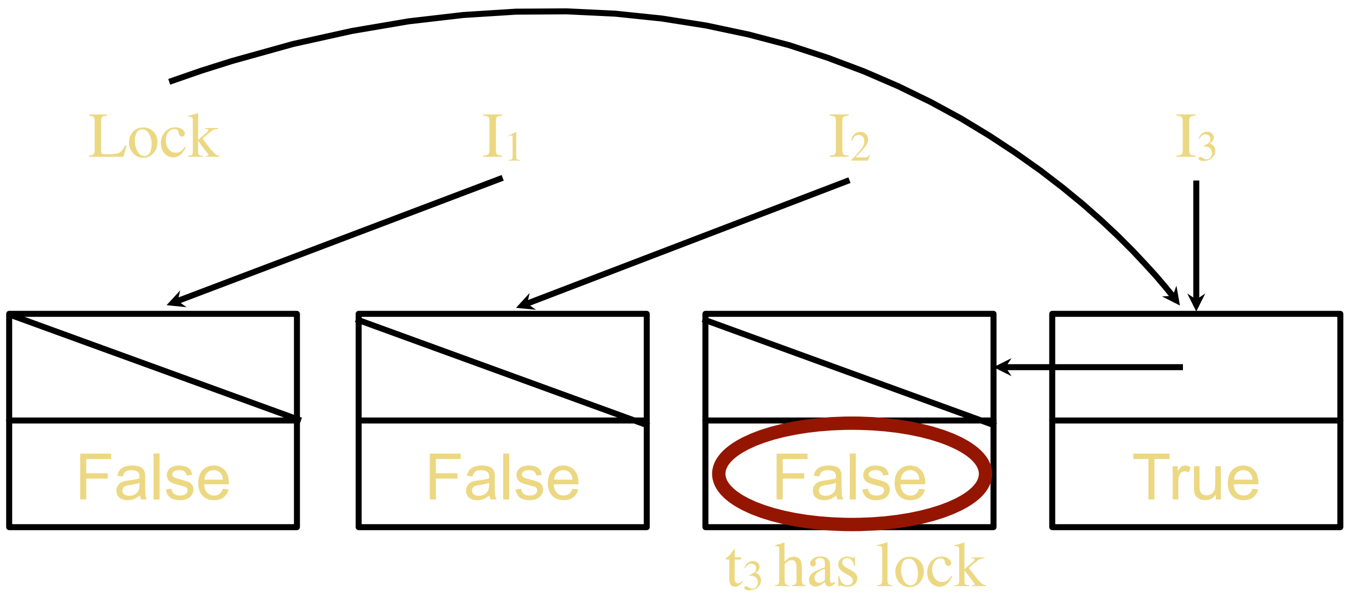
- t_1: Acquire(L)
- t_2: Acquire(L)
- t_3: Acquire(L)
- t_1: Release(L)



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release(L):
  - pred = I->prev
  - I->must_wait = false

Queue Locks Example: Time 5

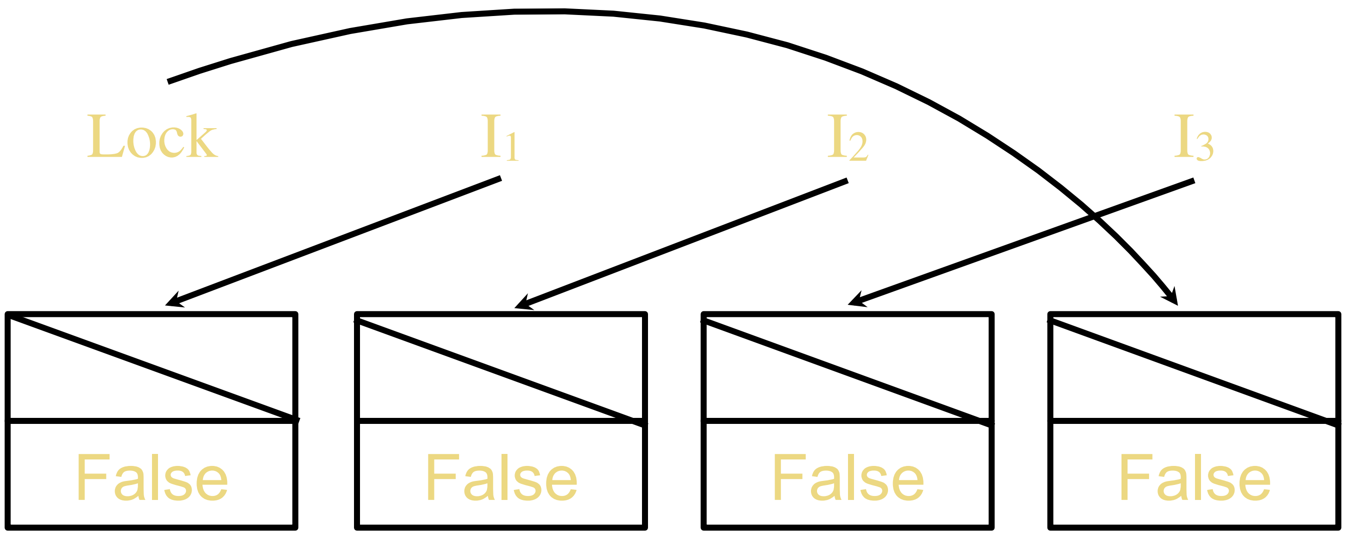
- t_1: Acquire(L)
- t_2: Acquire(L)
- t_3: Acquire(L)
- t_1: Release(L)
- t_2: Release(L)



- Acquire(L):
    - I->must_wait = true
    - I->prev = fetch_and_store(L, I)
    - pred = I->prev
    - while (pred->must_wait) // spin
- Release(L):
    - pred = I->prev
    - I->must_wait = false

Queue Locks Example: Time 6

- t_1: Acquire(L)
- t_2: Acquire(L)
- t_3: Acquire(L)
- t_1: Release(L)
- t_2: Release(L)
- t_3: Release(L)



- Acquire(L):
  - I->must_wait = true
  - I->prev = fetch_and_store(L, I)
  - pred = I->prev
  - while (pred->must_wait) // spin
- Release(L):
  - pred = I->prev
  - I->must_wait = false

Lock Review & Performance

- Test-and-set
- Test-and-test-and-set
  - With or without exponential backoff
- Ticket lock
- Array-based queue lock
  - “Anderson”
- List-based queue lock
  - “MCS”

Lock Performance

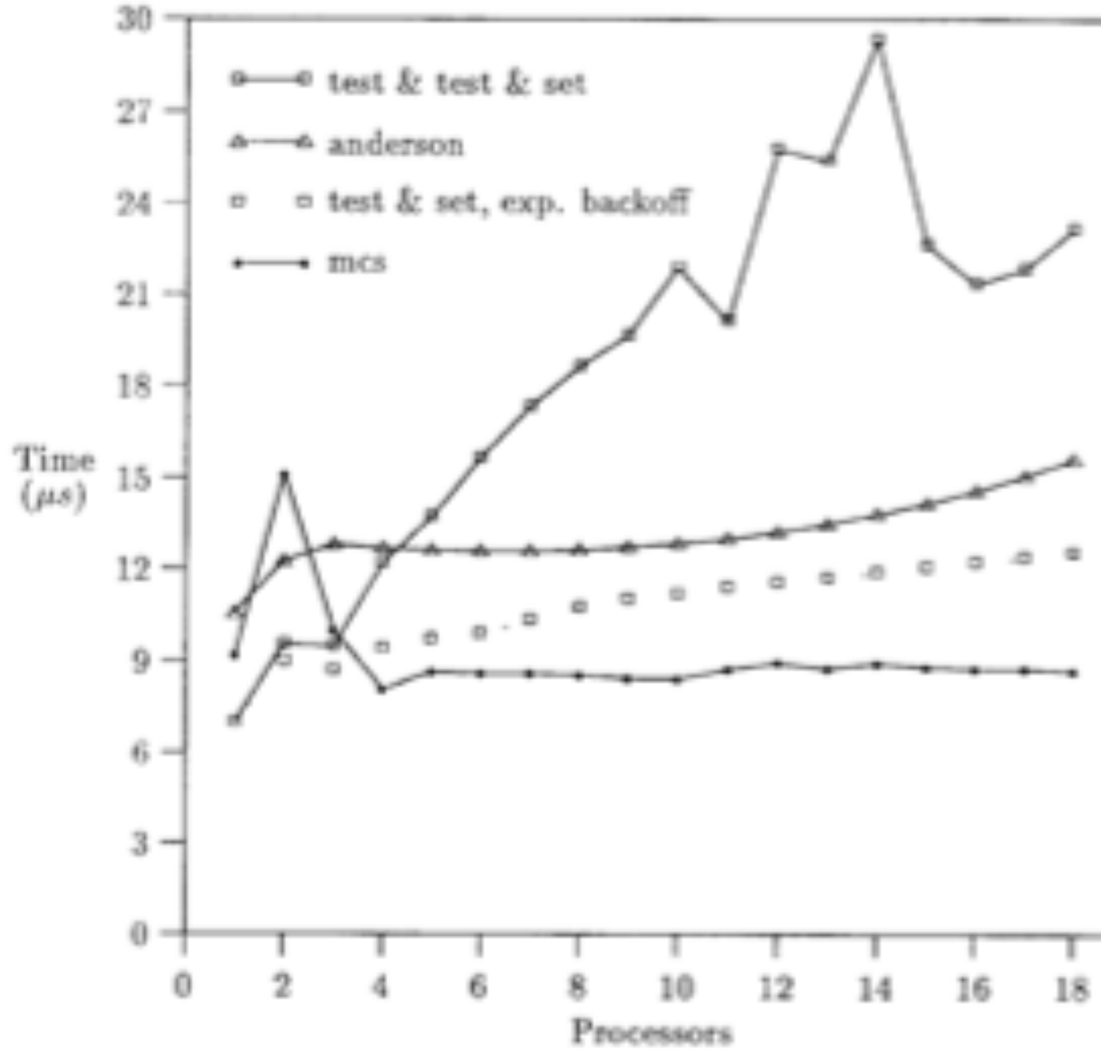


Fig. 17. Performance of spin locks on the Symmetry (empty critical section).

From Mellor-Crummey & Scott, ACM TOCS, 1991

Other Lock Concerns

- Supporting "bool trylock(timeout)"
  - Attempt to get the lock, give up after time
  - Easy for simple TTS locks; harder for ticket and queue-based locks
- Reader/Writer locks
  - lock_for_read(lock_ptr)   lock_for_write(lock_ptr)
  - Many readers can all "hold" lock at the same time
  - Only one writer (with no readers)
  - Reader/Writer locks sound great! **Two problems:**
    - Acquiring a read lock requires read-modify-write of shared data
    - Acquiring a read/write lock is slower than a simple TTS lock
- Other options: topology/hierarchy aware locks, adaptive hybrid TTS/queue locks, biased locks, etc.