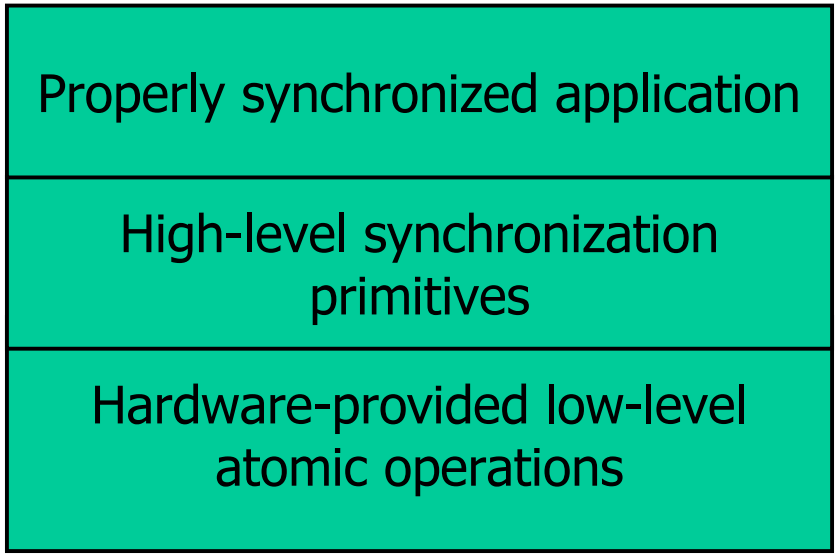
Recall: Layered approach to synchronization

- Hardware provides simple low-level atomic operations, upon which we can build high-level, synchronization primitives, upon which we can implement critical sections and build correct multi-threaded/multi-process programs



"Blocking" synchronization

- **Idea: if progress cannot be made because a resource cannot be acquired, it is desirable to free up execution resources for another thread (preempt the running thread)**

```
if (condition X not true)
   block until true;   // OS scheduler de-schedules thread
                       // (let's another thread use the processor)
```

- **pthreads mutex example**

```
pthread_mutex_t mutex;
pthread_mutex_lock(&mutex);
```

Busy waiting vs. blocking

- **Busy-waiting can be preferable to blocking if:**
  - Scheduling overhead is larger than expected wait time
  - A processor's resources not needed for other tasks
    - This is often the case in a parallel program since we usually don't oversubscribe a system when running a performance-critical parallel app (e.g., there aren't multiple CPU-intensive programs running at the same time)
    - Clarification: be careful to not confuse the above statement with the value of multi-threading (interleaving execution of multiple threads/tasks to hiding long latency of memory operations) with other work within the same app.

- **Examples:**

```
pthread_spinlock_t spin;
pthread_spin_lock(&spin);
```

```
int lock;
OSSpinLockLock(&lock);   // OSX spin lock
```

Implementing Locks

Warm up: a simple, but incorrect, lock

```
lock:       ld   R0, mem[addr]      // load word into R0
            cmp  R0, #0             // compre R0 to 0
            bnz  lock               // if nonzero jump to top
            st   mem[addr], #1

unlock:     st   mem[addr], #0      // store 0 to address
```

Problem: data race because LOAD-TEST-STORE is not atomic!

Processor 0 loads address X, observes 0

Processor 1 loads address X, observes 0

Processor 0 writes 1 to address X

Processor 1 writes 1 to address X

Test-and-set based lock

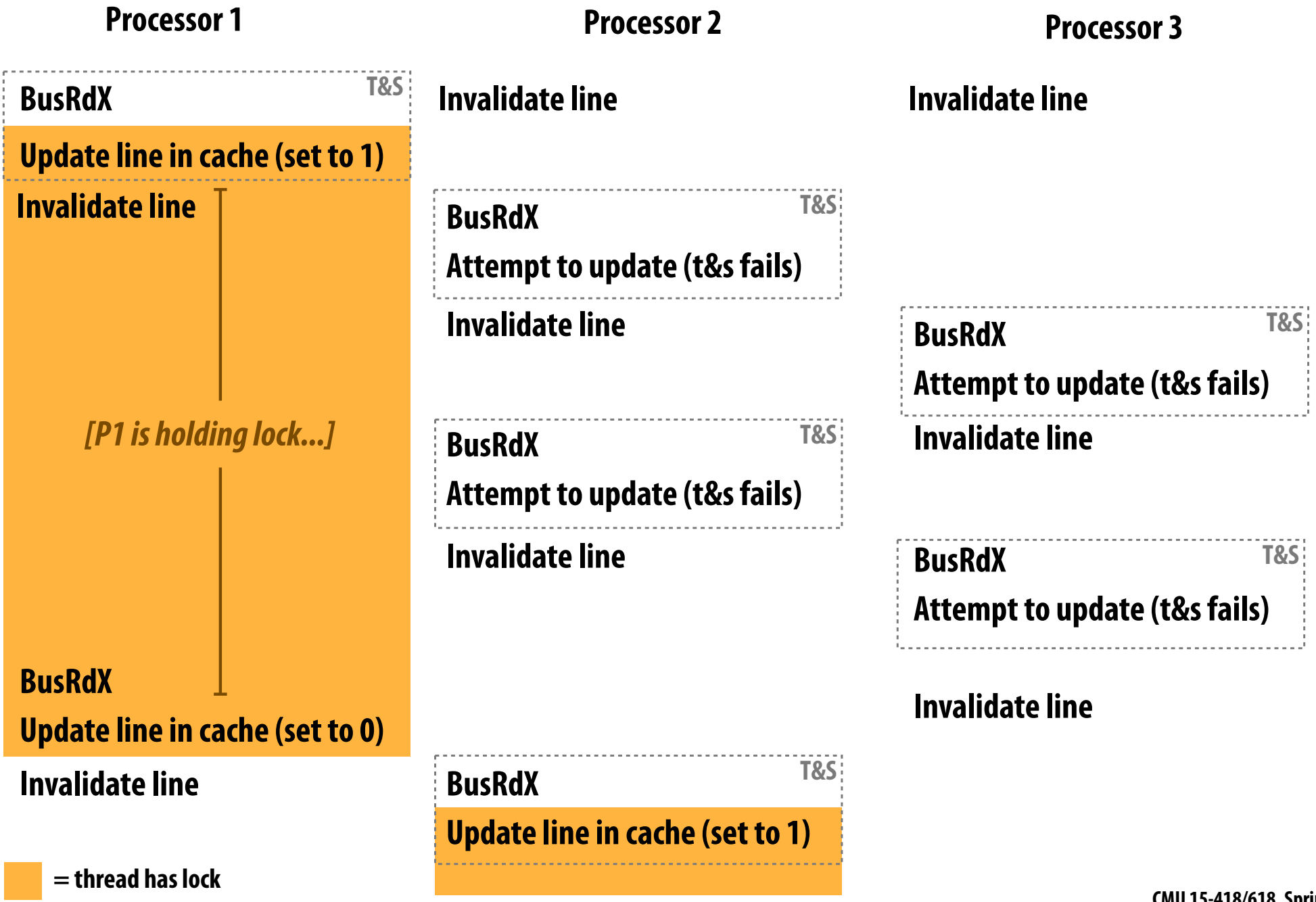
Atomic test-and-set instruction:

```
ts R0, mem[addr]        // load mem[addr] into R0
                        // if mem[addr] is 0, set mem[addr] to 1
```

```
lock:       ts   R0, mem[addr]        // load word into R0
            bnz  R0, lock             // if 0, lock obtained

unlock:     st   mem[addr], #0        // store 0 to address
```

Test-and-set lock: consider coherence traffi



Test-and-set lock performance

Benchmark: execute a total of N lock/unlock sequences (in aggregate) by P processors
Critical section time removed so graph plots only time acquiring/releasing the lock

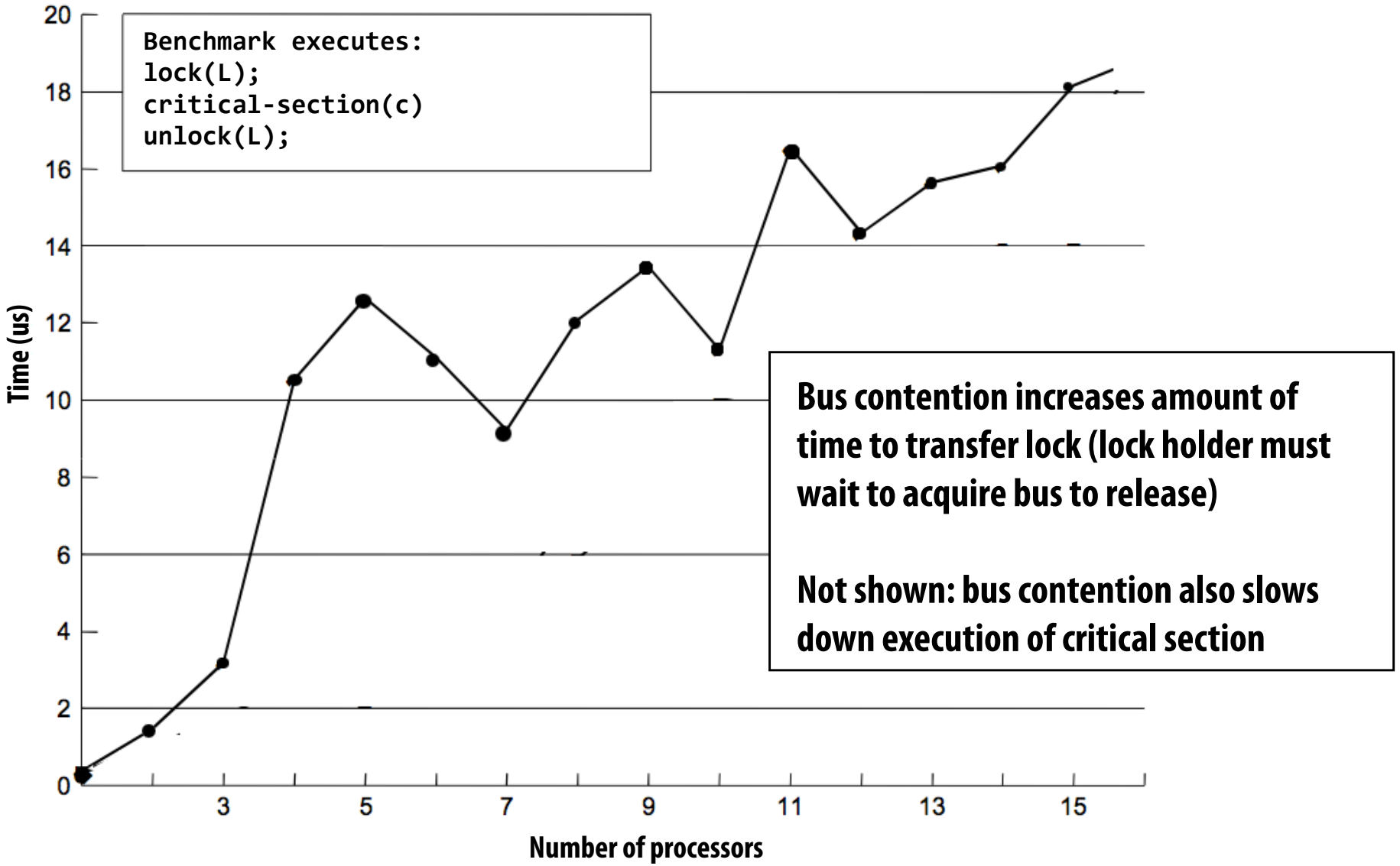


Figure credit: Culler, Singh, and Gupta

Desirable lock performance characteristics

- Low latency
  - If lock is free and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- Low interconnect traffic
  - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- Scalability
  - Latency / traffic should scale reasonably with number of processors
- Low storage cost
- Fairness
  - Avoid starvation or substantial unfairness
  - One ideal: processors should acquire lock in the order they request access to it

Simple test-and-set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness

Test-and-test-and-set lock

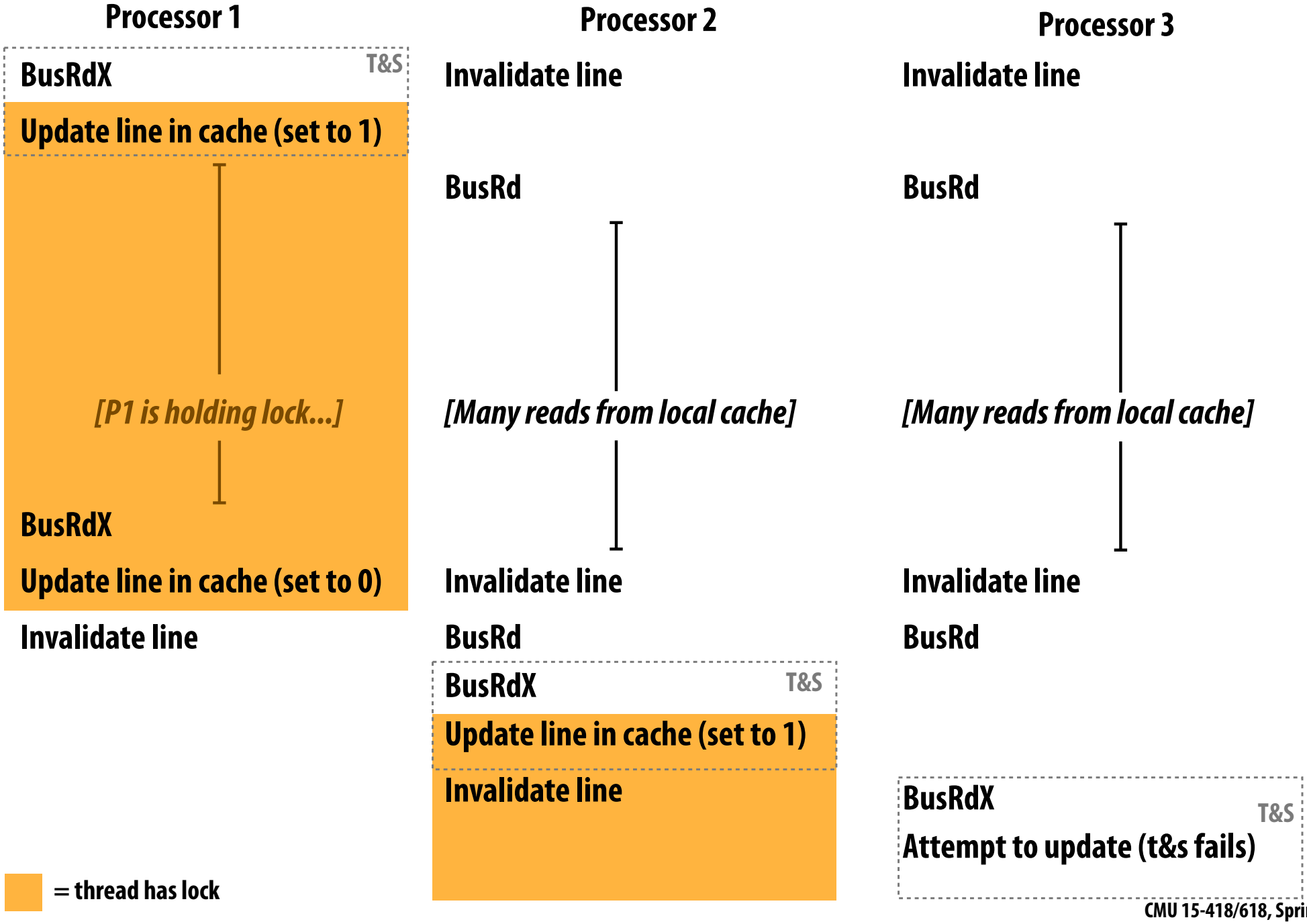
```
void Lock(int* lock) {
  while (1) {

    while (*lock != 0);            // while another processor has the lock…
                                   // (assume *lock is NOT register allocated)


    if (test_and_set(*lock) == 0)  // when lock is released, try to acquire it
      return;
  }
}

void Unlock(int* lock) {
   *lock = 0;
}
```

Test-and-test-and-set lock: coherence traffic



Test-and-test-and-set characteristics

- Slightly higher latency than test-and-set in uncontended case
  - Must test... then test-and-set
- Generates much less interconnect traffic
  - One invalidation, per waiting processor, per lock release (O(P) invalidations)
  - This is $O(P^2)$ interconnect traffic if all processors have the lock cached
  - Recall: test-and-set lock generated one invalidation per waiting processor per test
- More scalable (due to less traffic)
- Storage cost unchanged (one int)
- Still no provisions for fairness

Test-and-set lock with back off

Upon failure to acquire lock, delay for awhile before retrying

```
void Lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (test_and_set(*l) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

- Same uncontended latency as test-and-set, but potentially higher latency under contention. Why?
- Generates less traffic than test-and-set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged (still one int for lock)
- Exponential back-offcan cause severe unfairness
  - Newer requesters back offfor shorter intervals

Load-linked, store conditional (LL/SC)

- Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)
  - load_linked(x): load value from address
  - store_conditional(x, value): store value to x, if x hasn't been written to since corresponding LL

- Corresponding ARM instructions: LDREX and STREX

- How might LL/SC be implemented on a cache coherent processor?

Load-Linked and Store Conditional

- LL-SC is an implementation of atomic read-modify-write with very high flexibility
- LL: read a value and update a table indicating you have read this address, then perform any amount of computation
- SC: attempt to store a result into the same memory location, the store will succeed only if the table indicates that no other process attempted a store since the local LL (success only if the operation was “effectively” atomic)
- SC implementations do not generate bus traffic if the SC fails – hence, more efficient than test&test&set

Spin Lock with Low Coherence Traffic

```
lockit:  LL       R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ     R2, lockit    ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC       R2, 0(R1)    ; store-conditional succeeds if no one
                                ; updated the lock since the last LL
         BEQZ     R2, lockit    ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

Spin Lock with Low Coherence Traffic

```
lockit:  LL      R2, 0(R1)    ; load linked, generates no coherence traffic
         BNEZ    R2, lockit    ; not available, keep spinning
         DADDUI R2, R0, #1 ; put value 1 in R2
         SC      R2, 0(R1)    ; store-conditional succeeds if no one
                              ; updated the lock since the last LL
         BEQZ    R2, lockit    ; confirm that SC succeeded, else keep trying
```

- If there are i processes waiting for the lock, how many bus transactions happen?

  1 write by the releaser + i read-miss requests + i responses + 1 write by acquirer + 0 (i-1 failed SCs) + i-1 read-miss requests + i-1 responses

Further Reducing Bandwidth Needs

- Ticket lock: every arriving process atomically picks up a ticket and increments the ticket counter (with an LL-SC), the process then keeps checking the now-serving variable to see if its turn has arrived, after finishing its turn it increments the now-serving variable

- Array-Based lock: instead of using a “now-serving” variable, use a “now-serving” array and each process waits on a different variable – fair, low latency, low bandwidth, high scalability, but higher storage

- Queueing locks: the directory controller keeps track of the order in which requests arrived – when the lock is available, it is passed to the next in line (only one process sees the invalidate and update)

Ticket lock

Main problem with test-and-set style locks: upon release, all waiting processors attempt to acquire lock using test-and-set

```
struct lock {
   int next_ticket;
   int now_serving;
};

void Lock(lock* l) {
  int my_ticket = atomic_increment(&l->next_ticket);   // take a "ticket"
  while (my_ticket != l->now_serving);                 // wait for number
}                                                      // to be called

void unlock(lock* l) {
  l->now_serving++;
}
```

No atomic operation needed to acquire the lock (only a read)

Result: only one invalidation per lock release (O(P) interconnect traffic)

Array-based lock

Each processor spins on a different memory address

Utilizes atomic operation to assign address on attempt to acquire

```
struct lock {
   padded_int status[P];     // padded to keep off same cache line
   int head;
};

int my_element;

void Lock(lock* l) {
  my_element = atomic_circ_increment(&l->head);    // assume circular increment
  while (l->status[my_element] == 1);
}

void unlock(lock* l) {
  l->status[my_element] = 1;
  l->status[circ_next(my_element)] = 0;            // next() gives next index
}
```

O(1) interconnect traffic per release, but lock requires space linear in P

Also, the atomic circular increment is a more complex operation (higher overhead)