


## Lock Implementations

[§8.1] Recall the three kinds of synchronization from Lecture 6:

- Point-to-point
- Lock
- 

### *Performance metrics for lock implementations*

- Uncontended latency
  - Time to acquire a lock when there is no contention
- Traffic
  - Lock acquisition when lock is already locked
  - Lock acquisition when lock is free
  - Lock release
- Fairness
  - Degree in which a thread can acquire a lock with respect to others
- Storage
  - As a function of # of threads/processors

## The need for atomicity

This code sequence illustrates the need for atomicity. Explain.

```
void lock (int *lockvar) {
    while (*lockvar == 1) {} ; // wait until released
    *lockvar = 1;              // acquire lock
}

void unlock (int *lockvar) {
    *lockvar = 0;
}
```

In assembly language, the sequence looks like this:

```
lock: ld R1, &lockvar      // R1 = lockvar
      bnz R1, lock         // jump to lock if R1 != 0
```

```

        sti &lockvar, #1      // lockvar = 1
        ret                  // return to caller
unlock: sti &lockvar, #0     // lockvar = 0
        ret                  // return to caller

```

The `ld-to-sti` sequence must be executed atomically:

- The sequence appears to execute in its entirety
- Multiple sequences are serialized

### *Examples of atomic instructions*

- **test-and-set  $R_x, M$** 
  - read the value stored in memory location **M**, test the value against a constant (e.g. 0), and if they match, write the value in register **R<sub>x</sub>** to the memory location **M**.
- **fetch-and-op  $M$** 
  - read the value stored in memory location **M**, perform op to it (e.g., increment, decrement, addition, subtraction), then store the new value to the memory location **M**.
- **exchange  $R_x, M$** 
  - atomically exchange (or swap) the value in memory location **M** with the value in register **R<sub>x</sub>**.
- **compare-and-swap  $R_x, R_y, M$** 
  - compare the value in memory location **M** with the value in register **R<sub>x</sub>**. If they match, write the value in register **R<sub>y</sub>** to **M**, and copy the value in **R<sub>x</sub>** to **R<sub>y</sub>**.

How to ensure one atomic instruction is executed at a time:

1. Reserve the bus until done
  - Other atomic instructions cannot get to the bus

2. Reserve the cache block involved until done
  - Obtain exclusive permission (e.g. “M” in MESI)
  - Reject or delay any invalidation or intervention requests until done
3. Provide “illusion” of atomicity instead
  - Using load-link/store-conditional (to be discussed later)

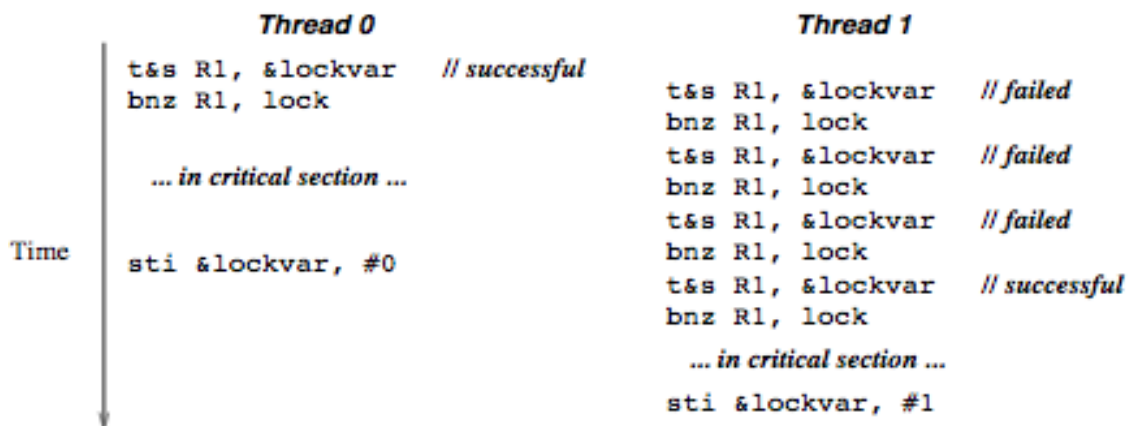
## Test and set

**test-and-set** can be used like this to implement a lock:

```
lock:    t&s R1, &lockvar    // R1 = MEM[&lockvar];
                                // if (R1==0) MEM[&lockvar]=1
        bnz R1, lock;        // jump to lock if R1 != 0
        ret                  // return to caller
unlock:  sti &lockvar, #0    // MEM[&lockvar] = 0
        ret                  // return to caller
```

What value does **lockvar** have when the lock is acquired? free?

Here is an example of **test-and-set** execution. Describe what it shows.



Let's look at how a sequence of test-and-sets by three processors plays out:

Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: t&s	M	–	–	BusRdX
P2: t&s	I	M	–	BusRdX
P3: t&s	I	I	M	BusRdX
P2: t&s	I	M	I	BusRdX
P1: unlock	M	I	I	BusRdX
P2: t&s	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: t&s	I	I	M	–
P2: unlock	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: unlock	I	I	M	–

[How does test-and-set perform](#) on the four metrics listed above?

- Uncontended latency
- Fairness
- Traffic
- Storage

### *Drawbacks of Test&Set Lock (TSL)*

What is the main drawback of test&set locks?

- 
- 

Without changing the lock mechanism, how can we diminish this overhead?

- : pause for awhile
  - by too little:

- \_\_\_\_\_ by too much: \_\_\_\_\_
- Exponential \_\_\_\_\_: Increase the \_\_\_\_\_ interval exponentially with each failure.

## Test and Test&Set Lock (TTSL)

- Busy-wait with ordinary read operations, not test&set.
  - Cached lock variable will be invalidated when release occurs
- When value changes (to 0), try to obtain lock with test&set
  - Only one attempter will succeed; others will fail and start testing again.

Let's compare the code for TSL with TTSL.

TSL:

```
lock:    t&s R1, &lockvar    // R1 = MEM[&lockvar];
                                // if (R1==0) MEM[&lockvar]=1
        bnz R1, lock;        // jump to lock if R1 != 0
        ret                  // return to caller
unlock:  sti &lockvar, #0    // MEM[&lockvar] = 0
        ret                  // return to caller
```

TTSL:

```
lock:    ld R1, &lockvar    // R1 = MEM[&lockvar]
        bnz R1, lock;        // jump to lock if R1 != 0
        t&s R1, &lockvar    // R1 = MEM[&lockvar];
                                // if (R1==0)MEM[&lockvar]=1
        bnz R1, lock;        // jump to lock if R1 != 0
        ret                  // return to caller

unlock:  sti &lockvar, #0    // MEM[&lockvar] = 0
        ret                  // return to caller
```

The `lock` method now contains two loops. What would happen if we removed the second loop?

Here's a trace of a TSL, and then TTSL, execution. Let's compare them line by line.

[Fill out](#) this table:

	TSL	TTSL
# BusReads		
# BusReadXs		
# BusUpgrs		
# invalidations		

(What's the proper way to count invalidations?)

TSL: Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: t&s	M	–	–	BusRdX
P2: t&s	I	M	–	BusRdX
P3: t&s	I	I	M	BusRdX
P2: t&s	I	M	I	BusRdX
P1: unlock	M	I	I	BusRdX
P2: t&s	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: t&s	I	I	M	–
P2: unlock	I	M	I	BusRdX
P3: t&s	I	I	M	BusRdX
P3: unlock	I	I	M	–

TTSL: Request	P1	P2	P3	Bus Request
Initially	–	–	–	–
P1: ld	E	–	–	BusRd
P1: t&s	M	–	–	–
P2: ld	S	S	–	BusRd
P3: ld	S	S	S	BusRd
P2: ld	S	S	S	–
P1: unlock	M	I	I	BusUpgr
P2: ld	S	S	I	BusRd
P2: t&s	I	M	I	BusUpgr
P3: ld	I	S	S	BusRd
P3: ld	I	S	S	–
P2: unlock	I	M	I	BusUpgr
P3: ld	I	S	S	BusRd
P3: t&s	I	I	M	BusUpgr
P3: unlock	I	I	M	–

*TSL vs. TTSL summary*

- Successful lock acquisition:
  - 2 bus transactions in TTSL
    - 1 BusRd to intervene with a remotely cached block
    - 1 BusUpgr to invalidate all remote copies
  - vs. only 1 in TSL
    - 1 BusRdX to invalidate all remote copies
- Failed lock acquisition:
  - 1 bus transaction in TTSL
    - 1 BusRd to read a copy
    - then, loop until lock becomes free
  - vs. unlimited with TSL
    - Each attempt generates a BusRdX

## LL/SC

- TTSL is an improvement over TSL.
- But bus-based locking
  - has a limited applicability (explain)
  - is not scalable with fine-grain locks (explain)
- Suppose we could lock a *cache block* instead of a bus ...
  - Expensive, must rely on buffering or NACK
- Instead of providing atomicity, can we provide an illusion of atomicity instead?
  - This would involve detecting a violation of atomicity.
  - If something “happens to” the value loaded, cancel the store (because we must not allow newly stored value to become visible to other processors)

- Go back and repeat all other instructions (load, branch, etc.).

This can be done with two new instructions:

- Load Linked/Locked (LL)
  - reads a word from memory, and
  - stores the address in a special LL register
  - The LL register is cleared if anything happens that may break atomicity, e.g.,
    - A context switch occurs
    - The block containing the address in the LL register is invalidated.
- Store Conditional (SC)
  - tests whether the address in the LL register matches the store address
  - if so, store succeeds: store goes to cache/memory;
  - else, store fails: the store is canceled, 0 is returned.

Here is the code.

```
lock: LL R1, &lockvar // R1 = lockvar;
      // LINKREG = &lockvar
      bnz R1, lock    // jump to lock if R1 != 0
      add R1, R1, #1  // R1 = 1
      SC R1, &lockvar // lockvar = R1;
      beqz R1, lock   // jump to lock if SC fails
      ret             // return to caller

unlock: sti &lockvar, #0 // lockvar = 0
        ret             // return to caller
```

Note that this code, like the TTSL code, consists of two loops. Compare each loop with its TTSL counterpart.

- The first loop
- The second loop

Here is a trace of execution. [Compare it](#) with TTSL.

Request	P1	P2	P3	BusRequest
Initially	–	–	–	–
P1: LL	E	–	–	BusRd
P1: SC	M	–	–	–
P2: LL	S	S	–	BusRd
P3: LL	S	S	S	BusRd
P2: LL	S	S	S	–
P1: unlock	M	I	I	BusUpgr
P2: LL	S	S	I	BusRd
P2: SC	I	M	I	BusUpgr
P3: LL	I	S	S	BusRd
P3: LL	I	S	S	–
P2: unlock	I	M	I	BusUpgr
P3: LL	I	S	S	BusRd
P3: SC	I	I	M	BusUpgr
P3: unlock	I	I	M	–

- Similar bus traffic
  - Spinning using loads  $\Rightarrow$  no bus transactions when the lock is not free
  - Successful lock acquisition involves two bus transactions. What are they?
- But a failed SC does not generate a bus transaction (in TTSL, all test&sets generate bus transactions).
  - Why don't SCs fail often?

### *Limitations of LL/SC*

- Suppose a lock is highly contended by  $p$  threads
  - There are  $O(p)$  attempts to acquire and release a lock

- A single release invalidates  $O(p)$  caches, causing  $O(p)$  subsequent cache misses
- Hence, each critical section causes  $O(p^2)$  bus traffic
- Fairness: There is no guarantee that a thread that contends for a lock will eventually acquire it.

These issues can be addressed by two different kinds of locks.

### **Ticket Lock**

- Ensures fairness, but still incurs  $O(p^2)$  traffic
- Uses the concept of a “bakery” queue
- A thread attempting to acquire a lock is given a ticket number representing its position in the queue.
- Lock acquisition order follows the queue order.

Implementation:

```
ticketLock_init(int *next_ticket, int *now_serving) {
    *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving) {
    my_ticket = fetch_and_inc(next_ticket);
    while (*now_serving != my_ticket) {};
}

ticketLock_release(int *next_ticket, int *now_serving) {
    *now_serving++;
}
```

Trace:

Steps	next_ticket	now_serving	my_ticket		
			P1	P2	P3
Initially	0	0	–	–	–
P1: fetch&inc	1	0	0	–	–
P2: fetch&inc	2	0	0	1	–
P3: fetch&inc	3	0	0	1	2
P1:now_serving++	3	1	0	1	2
P2:now_serving++	3	2	0	1	2
P3:now_serving++	3	3	0	1	2

Note that fetch&inc can be implemented with LL/SC.

### Array-Based Queueing Locks

With a ticket lock, a release still invalidates  $O(p)$  caches.

*Idea:* Avoid this by letting each thread wait for a unique variable. Waiting processes poll on different locations in an array of size  $p$ .

Just change **now\_serving** to an array! (renamed “**can\_serve**”).

A thread attempting to acquire a lock is given a ticket number in the queue.

Lock acquisition order follows the queue order

- Acquire
  - fetch&inc obtains the address on which to spin (the next array element).
  - We must ensure that these addresses are in different cache lines or memories
- Release
  - Set next location in array to 1, thus waking up process spinning on it.

Advantages and disadvantages:

- $O(1)$  traffic per acquire with coherent caches
  - And each release invalidates only one cache.
- FIFO ordering, as in ticket lock, ensuring fairness

- But,  $O(p)$  space per lock
- Good scalability for bus-based machines

Implementation:

```
ABQL_init(int *next_ticket, int *can_serve) {
    *next_ticket = 0;
    for (i=1; i<MAXSIZE; i++)
        can_serve[i] = 0;
    can_serve[0] = 1;
}

ABQL_acquire(int *next_ticket, int *can_serve) {
    *my_ticket = fetch_and_inc(next_ticket) % MAXSIZE;
    while (can_serve[*my_ticket] != 1) {};
}

ABQL_release(int *next_ticket, int *can_serve) {
    can_serve[*my_ticket + 1] = 1;
    can_serve[*my_ticket] = 0; // prepare for next time
}
```

Trace:

Steps	next_ticket	can_serve[]	my_ticket		
			P1	P2	P3
Initially	0	[1, 0, 0, 0]	–	–	–
P1: f&i	1	[1, 0, 0, 0]	0	–	–
P2: f&i	2	[1, 0, 0, 0]	0	1	–
P3: f&i	3	[1, 0, 0, 0]	0	1	2
P1: can_serve[1]=1	3	[0, 1, 0, 0]	0	1	2
P2: can_serve[2]=1	3	[0, 0, 1, 0]	0	1	2
P3: can_serve[3]=1	3	[0, 0, 0, 1]	0	1	2

Let's compare array-based queueing locks with ticket locks.

[Fill out](#) this table, assuming that 10 threads are competing:

	Ticket locks	Array-based queueing locks
#of invalidations		
# of subsequent cache misses		

## Comparison of lock implementations

Criterion	TSL	TTSL	LL/SC	Ticket	ABQL
Uncontested latency	Lowest	Lower	Lower	Higher	Higher
1 release max traffic	$O(p)$	$O(p)$	$O(p)$	$O(p)$	$O(1)$
Wait traffic	High	Low	–	–	–
Storage	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(p)$
Fairness guaranteed?	No	No	No	Yes	Yes

Discussion:

- Design must balance latency vs. scalability
  - ABQL is not necessarily best.
  - Often LL/SC locks perform very well.
  - Scalable programs rarely use highly-contended locks.
- Fairness sounds good in theory, but
  - Must ensure that the current/next lock holder does not suffer from context switches or any long delay events