Parallelism: Synchronization Revisited

Todd C. Mowry & Dave Eckhardt

(Contributions also from Angela Demke Brown)

I. Synchronization on a Parallel Machine

II. Transactional Memory

Recall: Intel's `xchg` Instruction

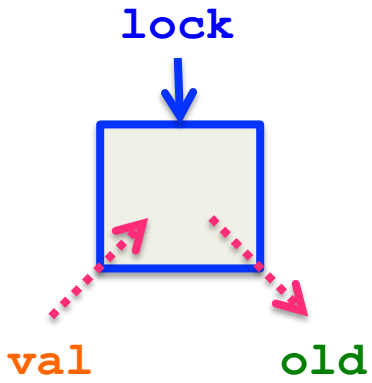
In assembly:

```
xchg (%esi), %edi
```

Functionality:

```
int32 xchg(int32 *lock, int32 val) {
  register int old;
  old = *lock;
  *lock = val;
  return (old);
}
```

} occur *atomically*



- *atomically* read the old value and store a new value
  - at the location pointed to by the lock pointer
- returns the old value
  - (by storing it in the register that contained the new value)

Recall: Using **xchg** to Implement a Lock

- Initialization:

```
int lock_available = 1;        // initially available
```

- Grabbing the lock:
  - "Try-lock" version:

```
i_won = xchg(&lock_available, 0);    // unavailable after this
```

  - Spin-wait version:

```
while (!xchg(&lock_available, 0)     // unavailable after this
    continue;
```

- Unlock:

```
xchg(&lock_available, 1);      // make lock available again
```

How Does **`xchg`** Actually Work?

- Complication:
  - fundamentally, this involves both a load and a store to a memory location
    - and these things can't occur simultaneously!
- How x86 processors handle complex instructions:
  - the hardware translates x86 instructions into simpler µop instructions
  - e.g., "**`add (%esi), %edi`**" actually turns into 3 µops:
    1. load (**`%esi`**) into a hardware register
    2. add **`%edi`** to that hardware register
    3. store result into (**`%esi`**)
- Hence at the µop level, "**`xchg (%esi), %edi`**" turns into:
  1. load (**`%esi`**) into a hardware register
     - *(through hardware register renaming, this eventually ends up in **`%edi`**)*
  2. store **`%edi`** into (**`%esi`**)
- Question: how do you think cache coherence handles this sequence?
  - Answer: need to get & hold the block *exclusively* throughout the sequence

If Lock Is Not Available, Should We Spin or Block?

Spin-Waiting:

```
while (!xchg(&lock_available, 0)
  continue;
```

Blocking:

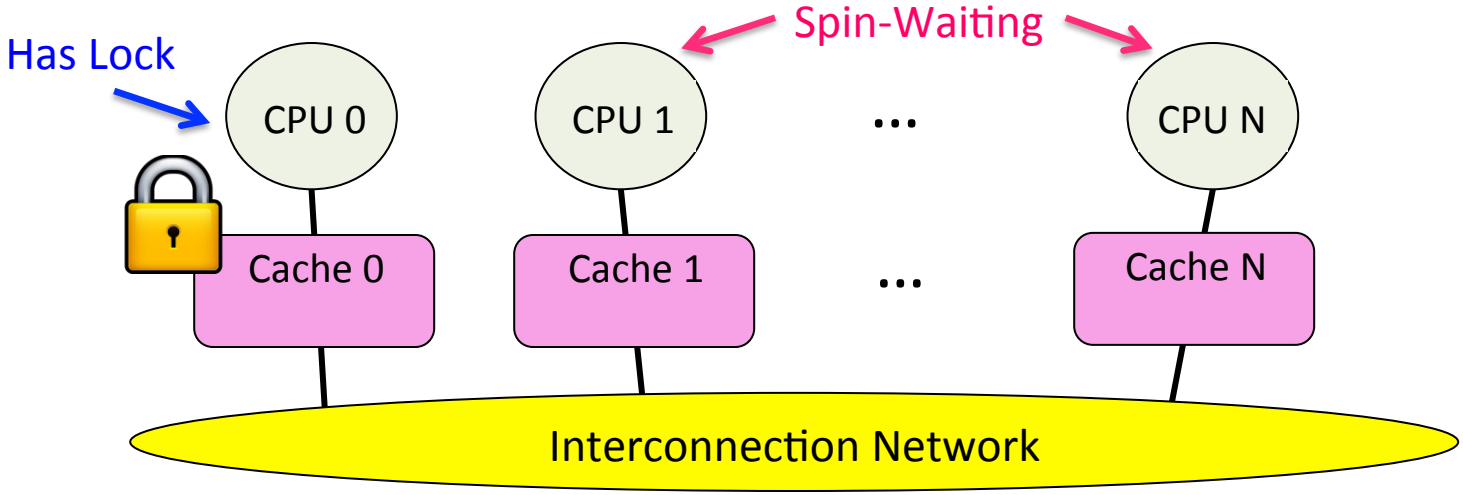
```
while (!xchg(&lock_available, 0)
  // go to sleep until someone does an unlock
  // (or at least yield)
  cond_wait(&unlock_event, …);
```

- Uniprocessor (review):
  - Who has the lock?
    - Another thread, that is currently blocked!
  - So spinning would be a waste of time.
- Multiprocessor:
  - Who has the lock?
    - Another thread, that is hopefully currently running (on a different processor)!
  - Also, parallel programmers try not to hold locks for very long
    - so hopefully it will become available soon, and we want to grab it ASAP
  - Spin-waiting looks attractive!
    - unless of course the thread holding the lock gets de-scheduled for some reason
    - common approach: spin for a while, and eventually block

Memory System Behavior for High-Contention Locks

- What if all processors are trying to grab the same lock at the same time:

```
while (!xchg(&lock_available, 0)
  continue;
```



- What will the coherence traffic across the interconnect network look like?
- As each processor spin-waits, it repeatedly:
  - requests an exclusive copy of the cache block, invalidating the other caches
  - checks whether the lock is available, and finds that it is not

    → constant stream of cache misses and coherence traffic: very bad!

Improved Version: "Test and Test-and-Set" Lock

```
do {
  while (lock_available == 0)          // "Test" loop
    continue;
} while (!xchg(&lock_available, 0);    // "Test-and-Set" check
```

- *(In the synchronization literature, our* ***xchg*** *lock is called a "test-and-set" lock.)*
- Note that the "test" loop uses a normal memory load (not an **xchg**)
- How does the coherence traffic change with this modification?
  - while the lock is held, the other processors spin locally in their caches
    - using normal read operations in the "test" loop, which hit on the "Shared" block
  - so there is no longer a flood of coherence traffic while the lock is held
- So are we completely happy now? Or is there still a problem…
  - what happens when the lock is released?

    → a sudden burst of "test-and-set" attempts, with all but one of them failing

Avoiding the Burst of Traffic When a Lock is Released

- One approach: use backoff
  - upon failing to acquire lock, delay for a while before retrying
    - either constant delay or exponential backoff

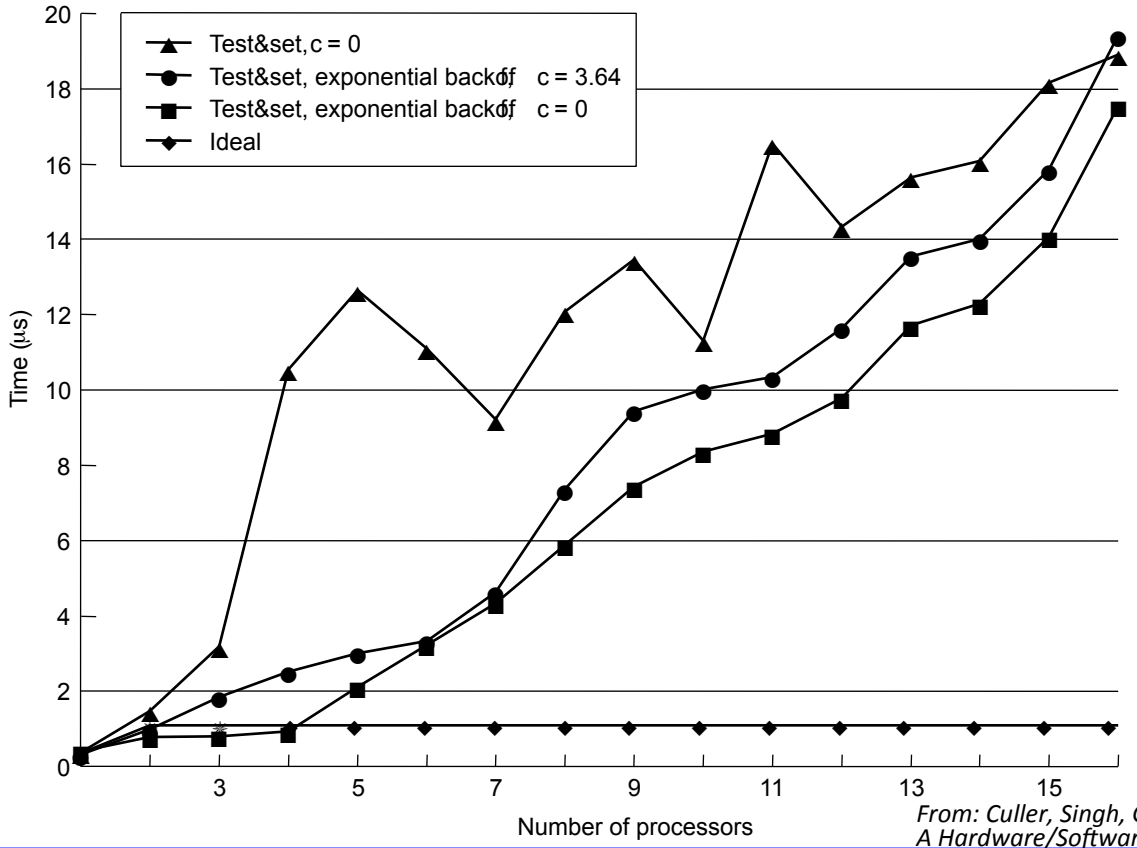
- The good news:
  - significantly reduces interconnect traffic

- The bad news:
  - exponential backoff can lead to starvation for high-contention locks
    - new requestors back off for shorter times

- Exponential backoff seems to work well in practice.

Performance of Test-and-Set Lock with Exponential Backoff

- Micro-benchmark loop body: `lock; delay(c); unlock;`
- Same total number of lock calls as the number of processors increases
- Y-Axis: measured time per transfer (on SGI Origin)



Backoff helps, but still far from ideal.

From: Culler, Singh, Gupta. "Parallel Computer Architecture: A Hardware/Software Approach." Morgan Kauffman.

Ticket Lock

Two counters:

Both initialized to 0.

`next_ticket`
(# of requestors)

`now_serving`
(# of releases that have happened)

Lock:

```
my_ticket = atomic_fetch_and_increment(&next_ticket);
while (now_serving != my_ticket)
   continue;
```

Can be implemented with Load-Linked/Store-Conditional

Unlock:

```
now_serving++;
```

Regular (non-atomic) operations.

- What is the coherence traffic like upon an unlock?
  - an invalidation, and then a read miss for each spinning processor
- Possible solution: use delay while spinning (but by how much?)

Ticket Lock Tradeoffs

The good news:

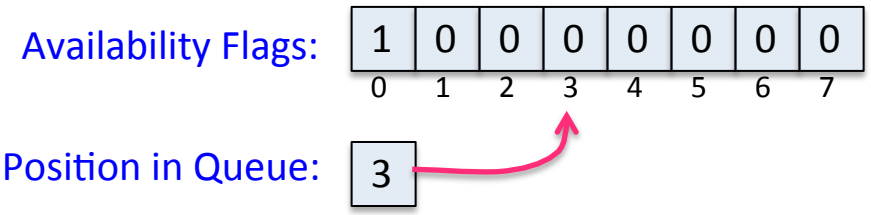
- guaranteed FIFO order
  - so starvation is not possible
- traffic can be quite low

But could it be better still?

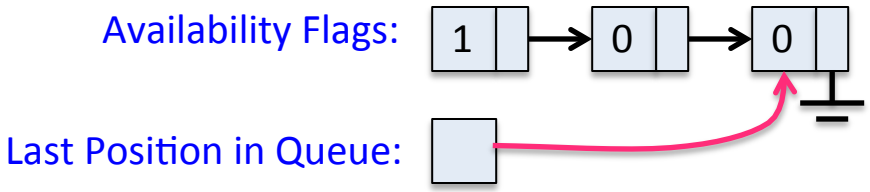
- traffic is not guaranteed to be O(1) per lock acquire

Achieving O(1) Traffic: Queueing Locks

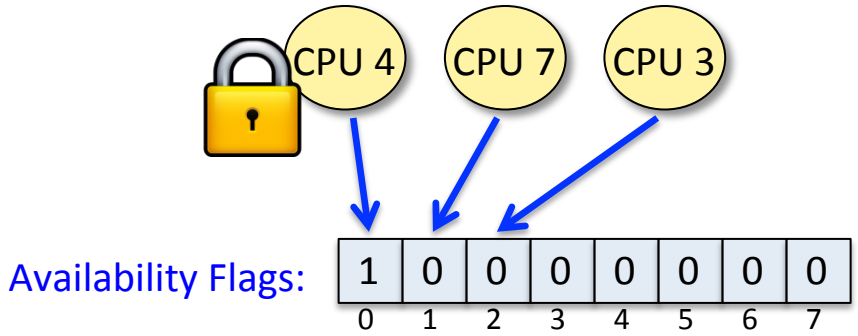
- Basic Idea:
  - pre-determine the order of lock handoff via a queue of waiters
  - during an unlock, the next processor in the queue is directly awakened
    - set a flag variable corresponding to the next waiter (spins locally in cache)
- Implementations:
  - Array-Based Queueing Locks:



  - List-Based Queueing Locks:



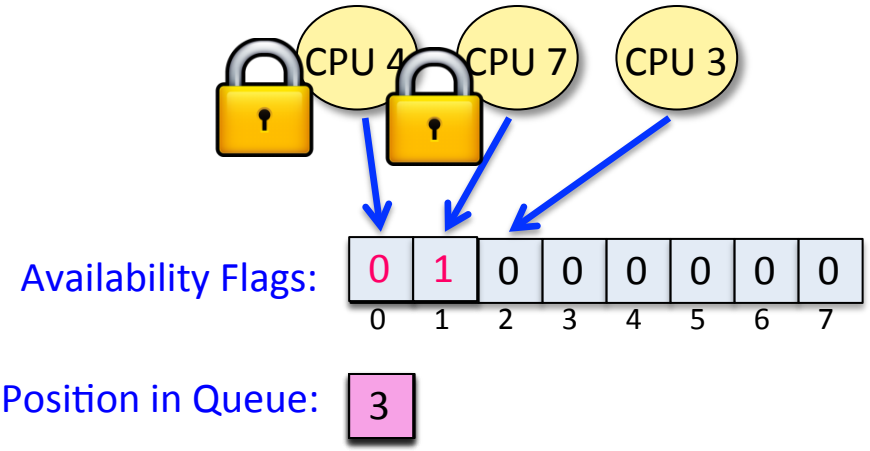
Array-Based Queueing Locks



Lock:

```
my_index = atomic_fetch_and_increment(&next_position)%NUM_PROCESSORS;
while (!lock_available[my_index])
   continue;
```

Array-Based Queueing Locks



Lock:

```
my_index = atomic_fetch_and_increment(&next_position)%NUM_PROCESSORS;
while (!lock_available[my_index])
   continue;
```

Unlock:

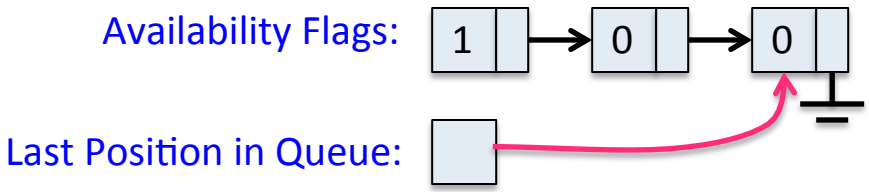
```
lock_available[my_index] = 0;
next_index = (my_index+1)%NUM_PROCESSORS;
lock_available[next_index] = 1;
```

Tradeoffs:

- Good: FIFO order; O(1) traffic (with cache coherence)
- Bad: requires space per lock proportional to P (x cache line size)

List-Based Queueing Locks

- Proposed by Mellor-Crummey and Scott (called "MCS" locks)



- Same basic idea, but insertions occur at the tail of a linked list.
- Space is allocated on-demand
  - aside from head pointers per lock, need only O(P) space for all locks in total
- Slightly more computation for lock/unlock operations

Which Performs Better: Test-and-Test-and-Set or Queue Locks?

- It depends on the amount of lock contention.
- Low-contention locks:
  - test-and-test-and-set is faster
    - less work to acquire the lock
- High-contention locks:
  - queue-based locks may be faster
    - less communication traffic, especially on large-scale systems

- Hybrid approaches have been proposed
  - switch from one to the other, depending on observed contention

Disadvantages of Locks (Even Ones that Scale Well)

- Functionality:
  - locks have their own addresses
    - separate from the data that they protect
  - mapping between locks and protected data is not explicit
    - usually only exists in programmer's head
  - locks are *not composable*
    - concurrent threads often need to know locking behavior to avoid breaking code
      - in contrast with usual modular code encapsulation in single-threaded code
    - e.g., stack modules with abstract push/pop operations:
      - moving item from one stack to another requires exposing locking mechanism

- Performance:
  - locks are a *pessimistic* way to achieve mutual exclusion
    - need to get permission first, before starting the critical section code
  - if a collision is unlikely, a more optimistic approach may perform better

Wouldn't it be nice if…

- Programmer simply specifies desired outcome:
  - "This code sequence should appear to execute *atomically.*"

```
void remove_node(Node_type *node) {
    atomic {
        if (node->prev != NULL)
            node->prev->next = node->next;
        if (node->next != NULL)
            node->next->prev = node->prev;
    }
}
```

- The system (e.g., language, run-time software, OS, hardware) makes this happen
  - hopefully *optimistic* (rather than pessimistic) to achieve high performance
  - while enabling *composability* of implementations within abstract objects, etc.

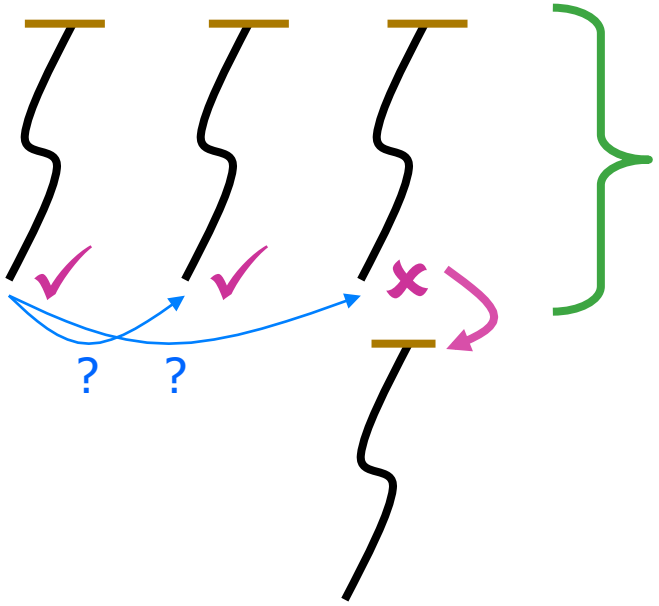
Transactional Memory (TM)

Source Code:

```
...
atomic {
 ...
 access_shared_data();
 ...
}
...
```

TM System

Transactions:



Programmer: Specifies threads/transactions in source code

TM System: Executes transactions optimistically in parallel

1) Checkpoints execution

2) Detects conflicts

3) (i) Commits or (ii) aborts and re-executes

Implementations of Transactional Memory (TM)

- Hardware TM (HTM)
  - Changes to computer system and ISA:
    - extend caches to buffer writes
    - extended coherence protocol to track conflicts
    - special transaction instructions
  - Support for limited number of memory locations
- Software TM (STM)
  - Language runtime (or library) + extensions to specify transaction
  - Exploit current commodity hardware (multicores)
  - Get experience with transactional programming model
  - Java: DSTM (Marathe et al.), ASTM (Herlihy et al.)
  - C/C++: McRT-STM (Saha et al.), TL2 (Dice et al.), RSTM
- Hybrid TM (HyTM)

How Transactional Memory Is Implemented

- For all (non-stack) write instructions:
  - track write addresses and values (write set)
- For all (non-stack) read instructions:
  - track read addresses and values (read set)

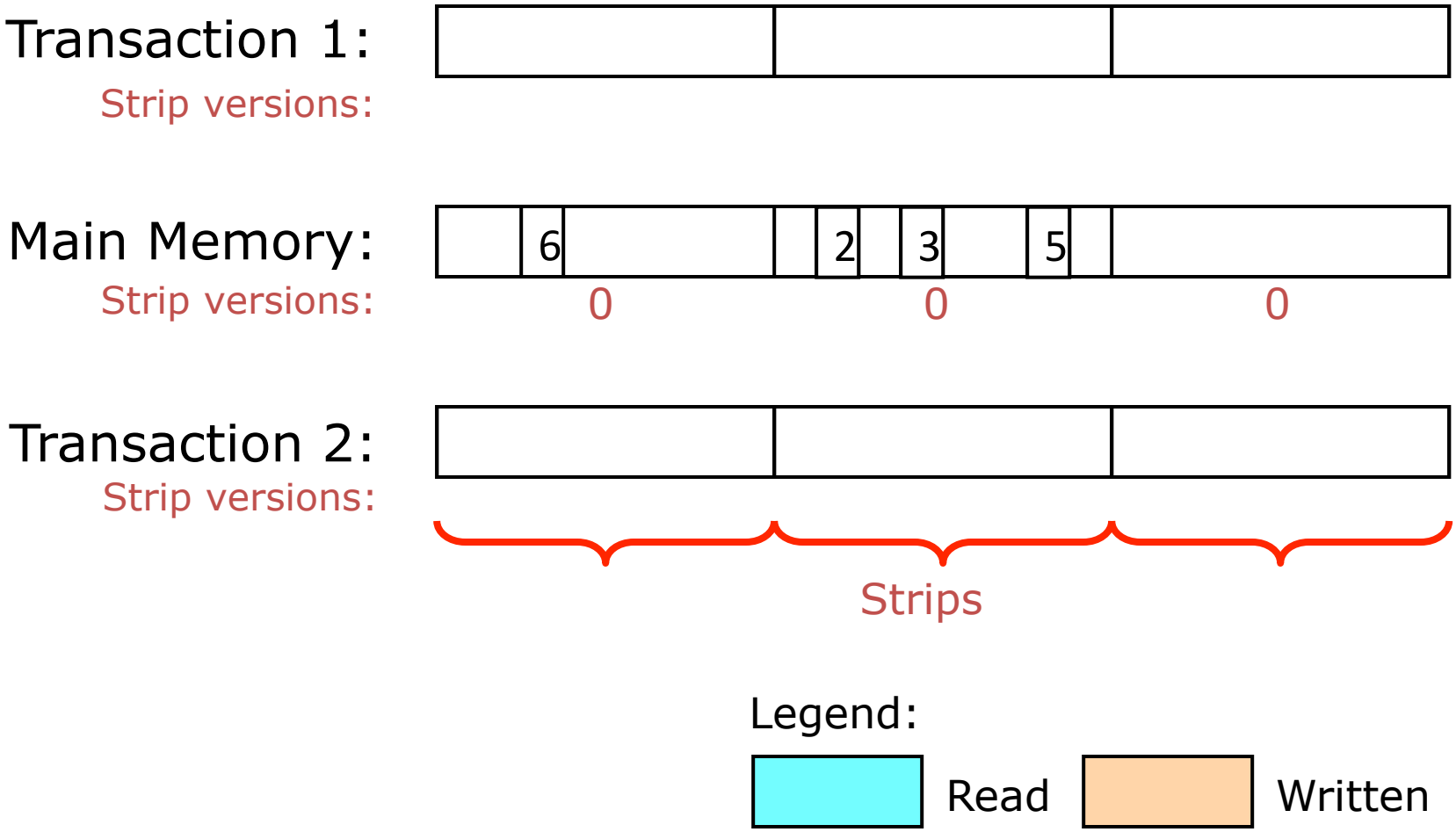
- When a transaction completes:
  - Atomically:
    - Validate read set (conflict detection)
    - Commit write set

Implementation Options

- Granularity: unit of storage over which conflicts are detected
  - Hardware TM: usually cache block or word
  - Software TM: usually per object
    - extend object-oriented languages

- Direct vs. Deferred Updates:
  - Direct: transaction directly modifies the object itself
    - must log previous value for undo in case of abort
  - Deferred: modify private copy, propagate at commit
  - Both get complicated in the presence of data races

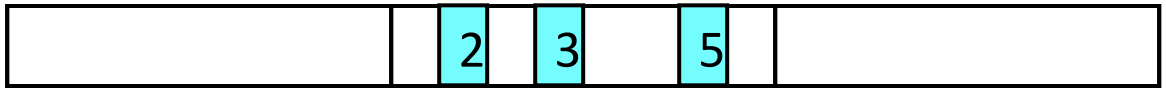
Location-Based Conflict Detection



Location-Based Conflict Detection

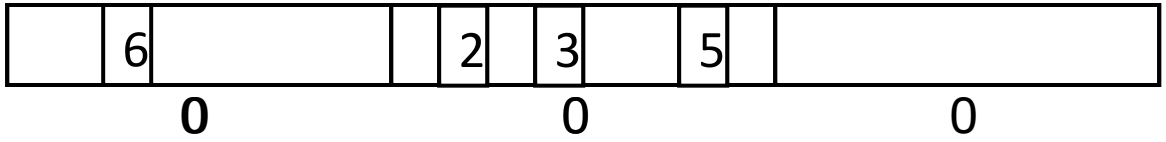
Transaction 1:

Strip versions:



Main Memory:

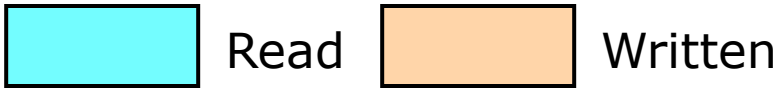
Strip versions:



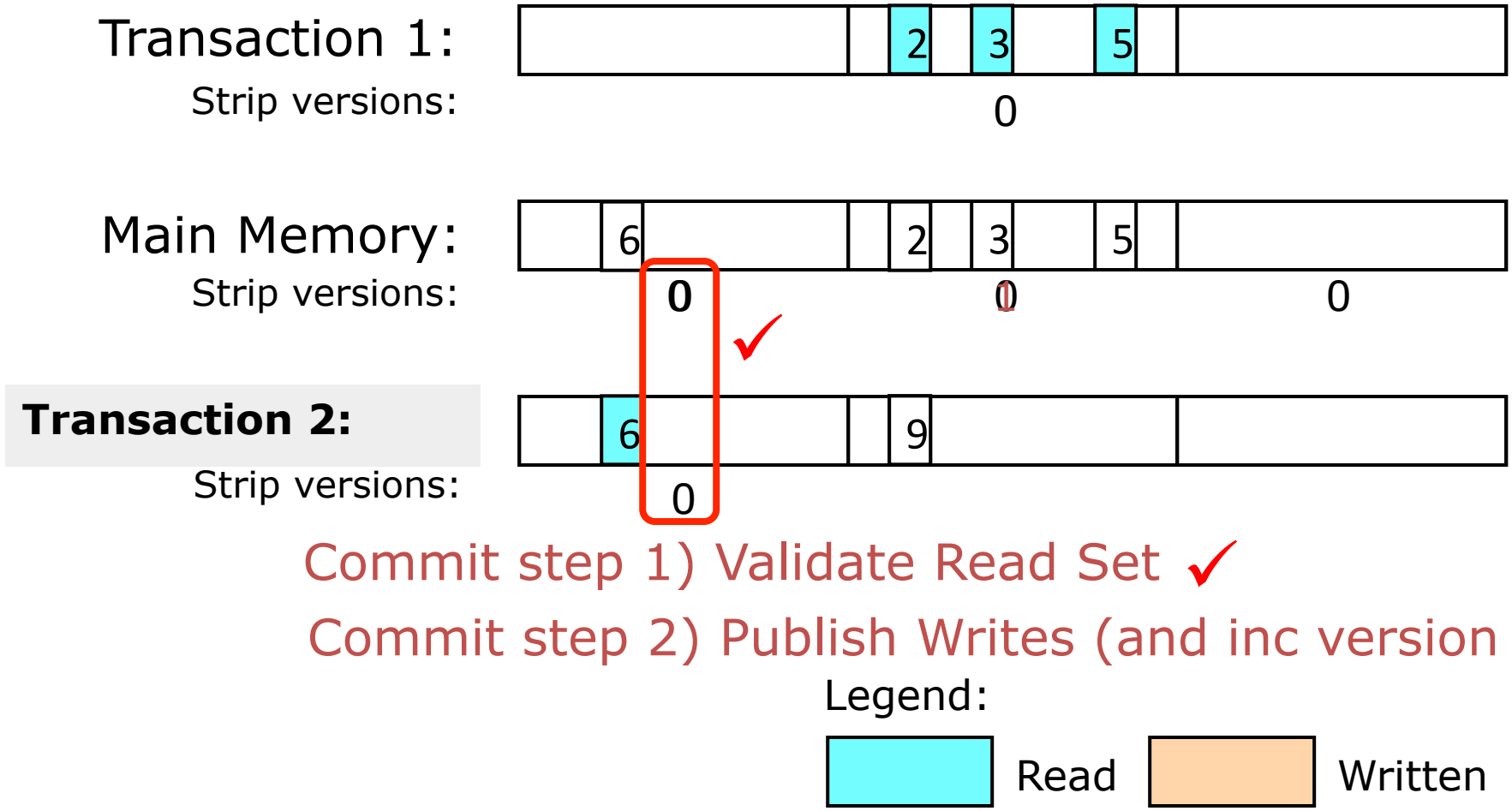
Transaction 2:

Strip versions:

Legend:



Location-Based Conflict Detection



Commit step 1) Validate Read Set ✓

Commit step 2) Publish Writes (and inc version #s)

Legend:

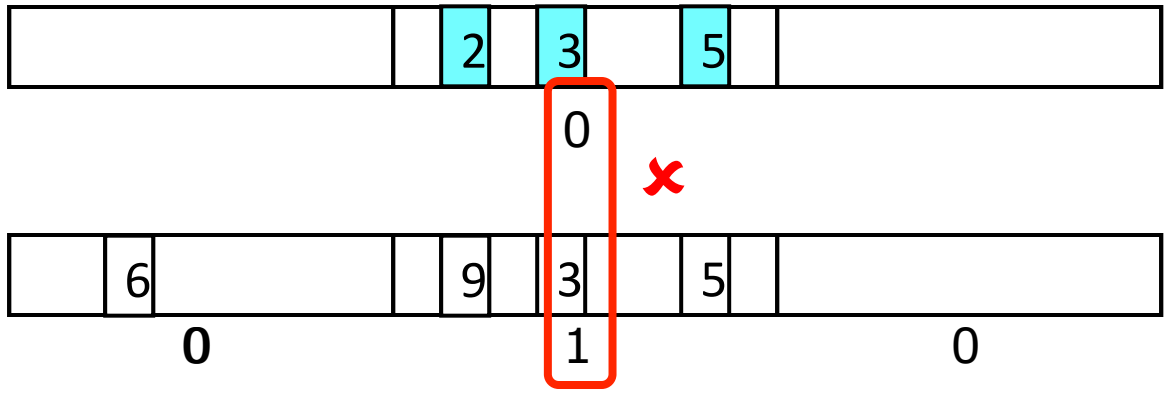
Read

Written

Location-Based Conflict Detection

Transaction 1:

Strip versions:



Commit step 1) Validate Read Set ✗ Abort!

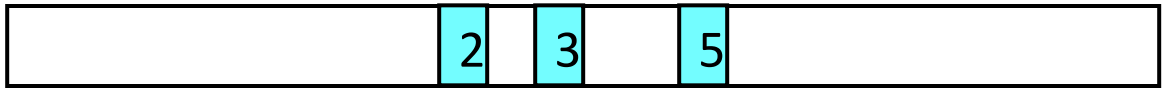
Note: all transactions must maintain strip version #s

Legend:

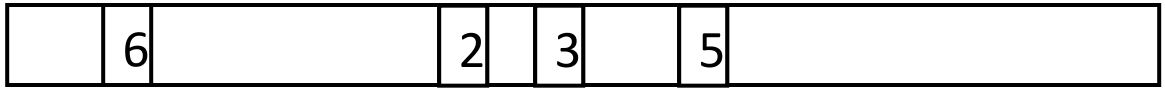
Read Written

Value-Based Conflict Detection

Transaction 1:

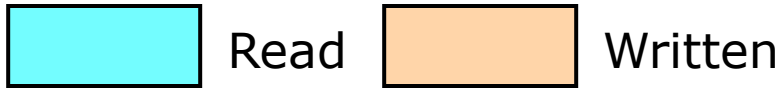


Main Memory:

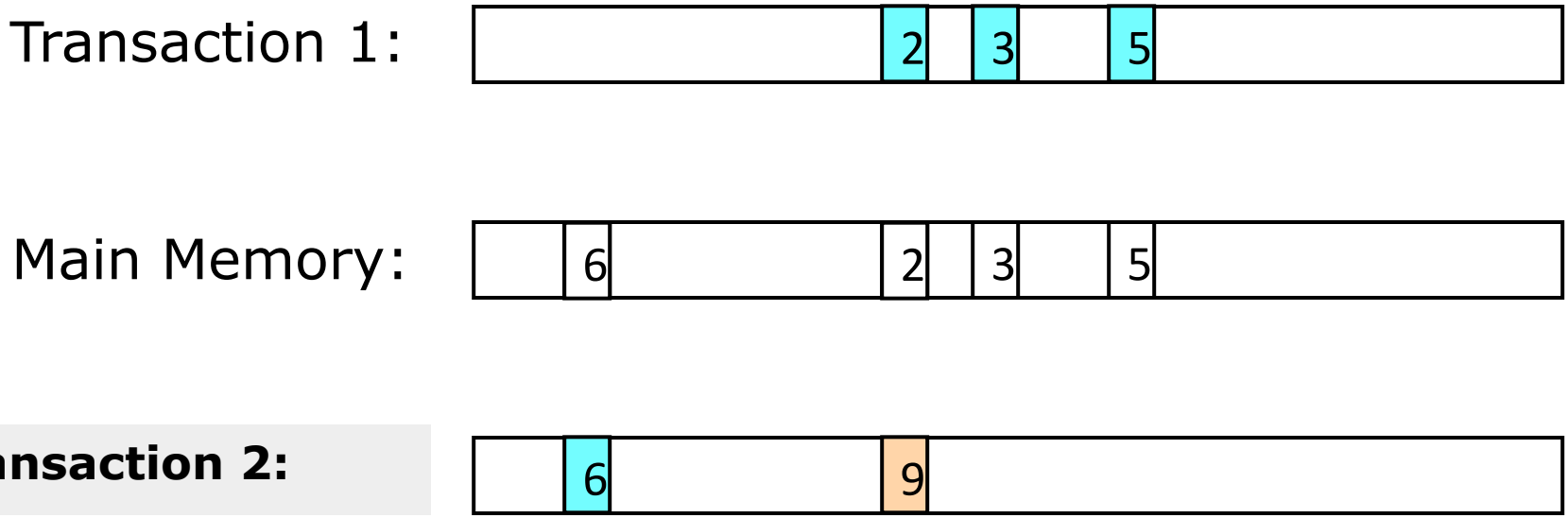


Transaction 2:

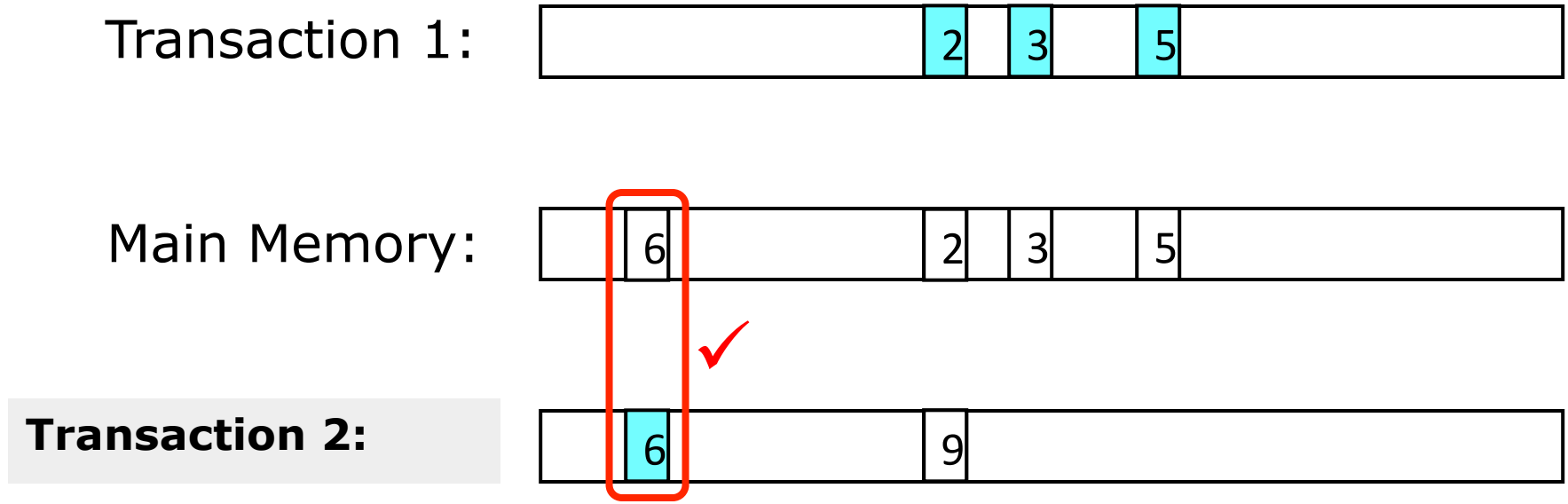
Legend:



Value-Based Conflict Detection



Value-Based Conflict Detection



Commit step 1) Validate Read Set ✓

Commit step 2) Publish Writes

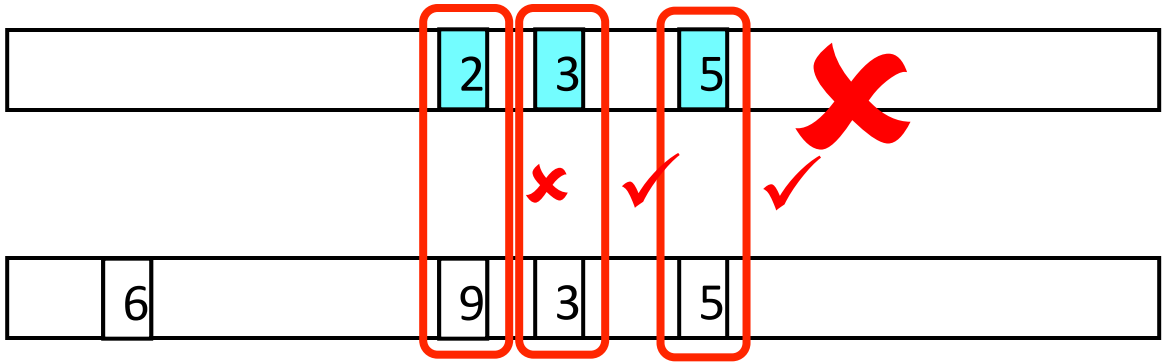
Legend:

Read

Written

Value-Based Conflict Detection

Transaction 1:



Transaction 2:

Commit step 1) Validate Read Set Abort!

Note: no version information to maintain

Legend:

Read Written

Transactional Memory Weaknesses

- Some operations are hard to abort/retry
  - essentially anything not idempotent (e.g., I/O)
- In practice, TM does not interact well with locking
- Some variables are prone to high conflict rates
  - frequent true sharing and dependences
- Conflict resolution needs to avoid starving long-running, large transactions

TM Status

- Hardware TM is now a reality
  - Sun's Rock processor was killed after acquisition by Oracle (2009)
  - Azul Systems has HTM in their Java appliance hardware (circa 2009)
  - IBM BlueGene/Q (2011)
  - Intel Haswell's *Transactional Sync Extensions (TSX)*
- Software TM has performance problems
  - But some applications are a nice fit
    - E.g. parallel game server