Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes

OSPP Chapter 6 – Part A

(adapted by Mark Smotherman from Tom Anderson’s slides on OSPP web site)

Multi-Object Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
  - Is locking modular?
- Synchronization performance
- Eliminating locks

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Overhead of creating threads, if not needed
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between cores
  - False sharing: communication between cores even for data that is not shared

Synchronization Performance Topics

- Multiprocessor cache coherence
- MCS locks (if locks are mostly busy)
- RCU locks (if locks are mostly busy, and data is mostly read-only)

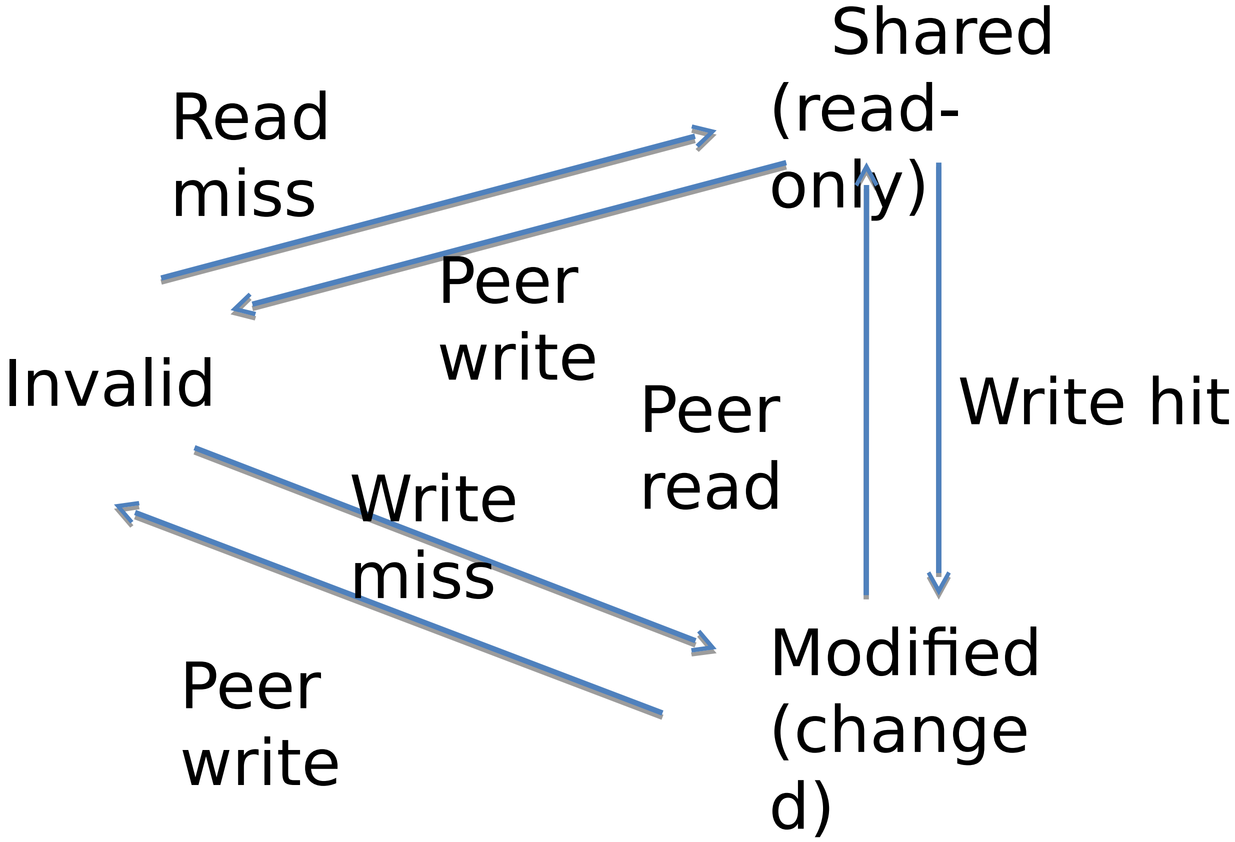
Multiprocessor Cache Coherence

- Scenario:
  - Thread A modifies data inside a critical section and releases lock
  - Thread B acquires lock and reads data
- Easy if all accesses go to main memory
  - Thread A changes main memory; thread B reads it
- What if new data is cached at processor A?
- What if old data is cached at processor B?

Write Back Cache Coherence

- Cache coherence = system behaves as if there is one copy of the data
  - If data is only being read, any number of caches can have a copy
  - If data is being modified, at most one cached copy
- On write: (get ownership)
  - Invalidate all cached copies, before doing write
  - Modified data stays in cache (“write back”)
- On read:
  - Fetch value from owner or from memory

Cache State Machine



Shared (read-only)

Read miss

Peer write

Invalid

Peer read

Write hit

Write miss

Peer write

Modified (changed)

Directory-Based Cache Coherence

- How do we know which cores have a location cached?
  - Hardware keeps track of all cached copies
  - On a read miss, if held exclusive, fetch latest copy and invalidate that copy or mark as shared
  - On a write miss, invalidate all copies
- Read-modify-write instructions
  - Fetch cache entry exclusive, prevent any other cache from reading the data until instruction completes

A Simple Critical Section

```
// A counter protected by a spinlock
Counter::Increment() {
    while (TestAndSet(&lock))
        ;

    value++;

    lock = FREE;
    memory_barrier();
}
```

A Simple Test of Cache Behavior

Array of 1K counters, each protected by a separate spinlock

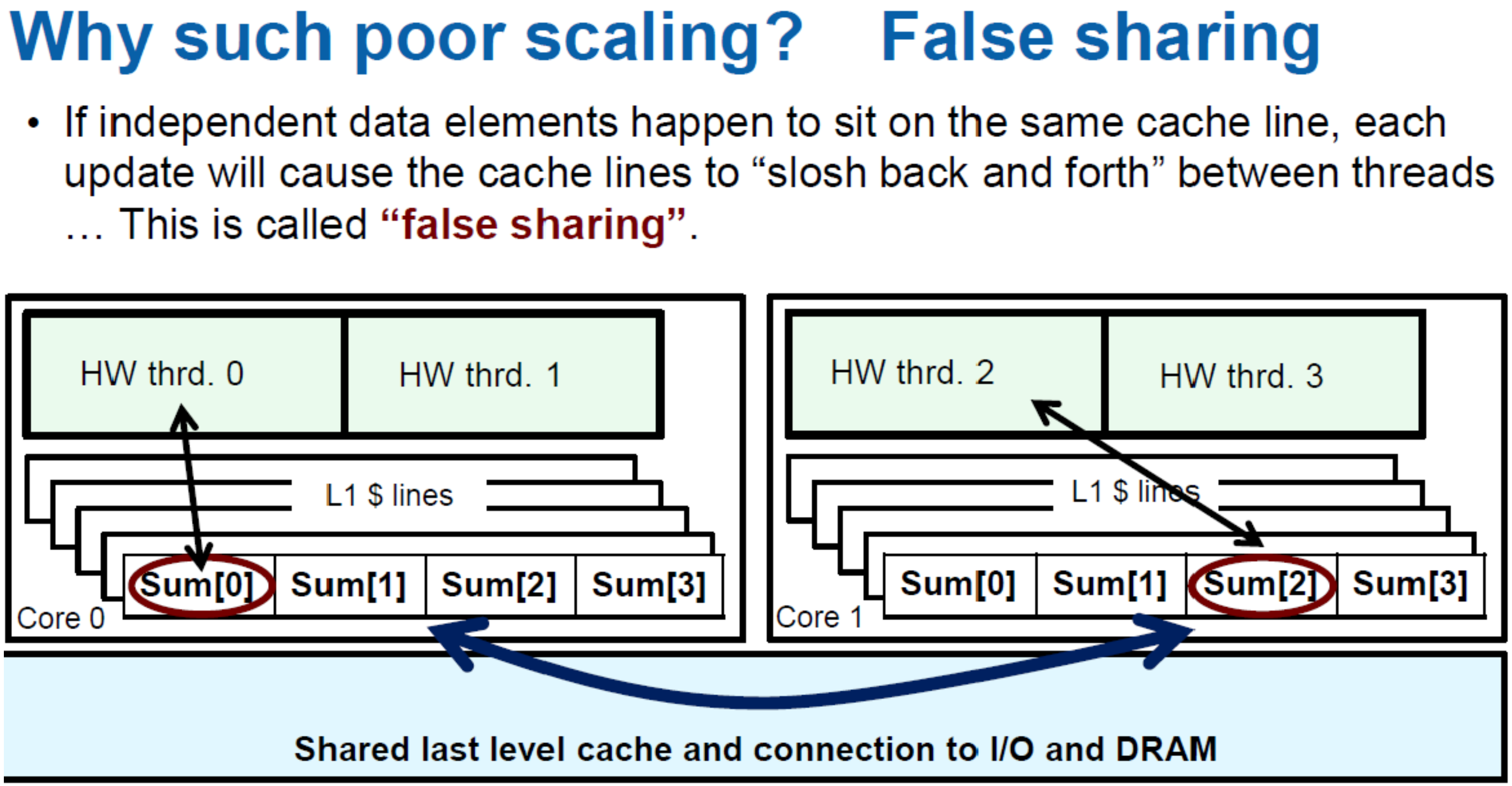
  - Array small enough to fit in cache

- Test 1: one thread loops over array
- Test 2: two threads loop over different arrays
- Test 3: two threads loop over single array
- Test 4: two threads loop over alternate elements in single array

Results (64 core AMD Opteron)

One thread, one array	51 cycles
Two threads, two arrays	52
Two threads, one array	197 (from contention)
Two threads, odd/even	127 (from false sharing)

False Sharing



Why such poor scaling? False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads … This is called **“false sharing”**.

Diagram is from Tim Mattson, “A ‘Hands On’ Introduction to OpenMP,”
https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Reducing Lock Contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
  - Example: hash table buckets
- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - Example: per-processor heap
- Ownership/Staged architecture
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads

Linus Shares His Opinion

You seem to have this blue-eyed belief that locking is simple. It's not.

... you talk about the locking cost as if something like a 12-20 cycles is "free". That's pure [BS]. Even if it's uncontended, you're dirtying cachelines in the L1. Guess what? If you have finegrained locking for lots of objects, the cost of all that extra cache traffic is really bad, and takes up a valuable resource.

End result: very few people actually do fine-grained locking at all. It's damn hard, and it easily eats up 50%+ of your CPU cycles if you do it wrong. You spend years getting it right for anything but the most trivial case.

[from realworldtech.com, June 6, 2013]

Locking Design Issues

Fine-grained locking comes at a cost, however. In a kernel with thousands of locks, it can be very hard to know which locks you need—and in which order you should acquire them—to perform a specific operation. Remember that locking bugs can be very difficult to find; more locks provide more opportunities for truly nasty locking bugs to creep into the kernel. Fine-grained locking can bring a level of complexity that, over the long term, can have a large, adverse effect on the maintainability of the kernel.

Locking in a device driver is usually relatively straightforward; you can have a single lock that covers everything you do, or you can create one lock for every device you manage. As a general rule, you should start with relatively coarse locking unless you have a real reason to believe that contention could be a problem. Resist the urge to optimize prematurely; the real performance constraints often show up in unexpected places.

If you do suspect that lock contention is hurting performance, you may find the lockmeter tool useful. This patch (available at http://oss.sgi.com/projects/lockmeter/) instruments the kernel to measure time spent waiting in locks. By looking at the report, you are able to determine quickly whether lock contention is truly the problem or not.

From "Locking Traps," sec. 5.6 in J. Corbet, G. Kroah-Hartman, and A. Rubin, Linux Device Drivers,
www.makelinux.net/ldd3/chp-5-sect-6

What If Locks are Still Mostly Busy?

- MCS Locks
  - Optimize lock implementation for when lock is contended
- RCU (read-copy-update)
  - Efficient readers/writers lock used in Linux kernel
  - Readers proceed without first acquiring lock
  - Writer ensures that readers are done
- Both rely on atomic read-modify-write instructions

The Problem with Test and Set

```
Counter::Increment() {
    while (TestAndSet(&lock))
        ;
    value++;
    lock = FREE;
    memory_barrier();
}
```

What happens if many processors try to acquire the lock at the same time?

- Hardware doesn’t prioritize FREE

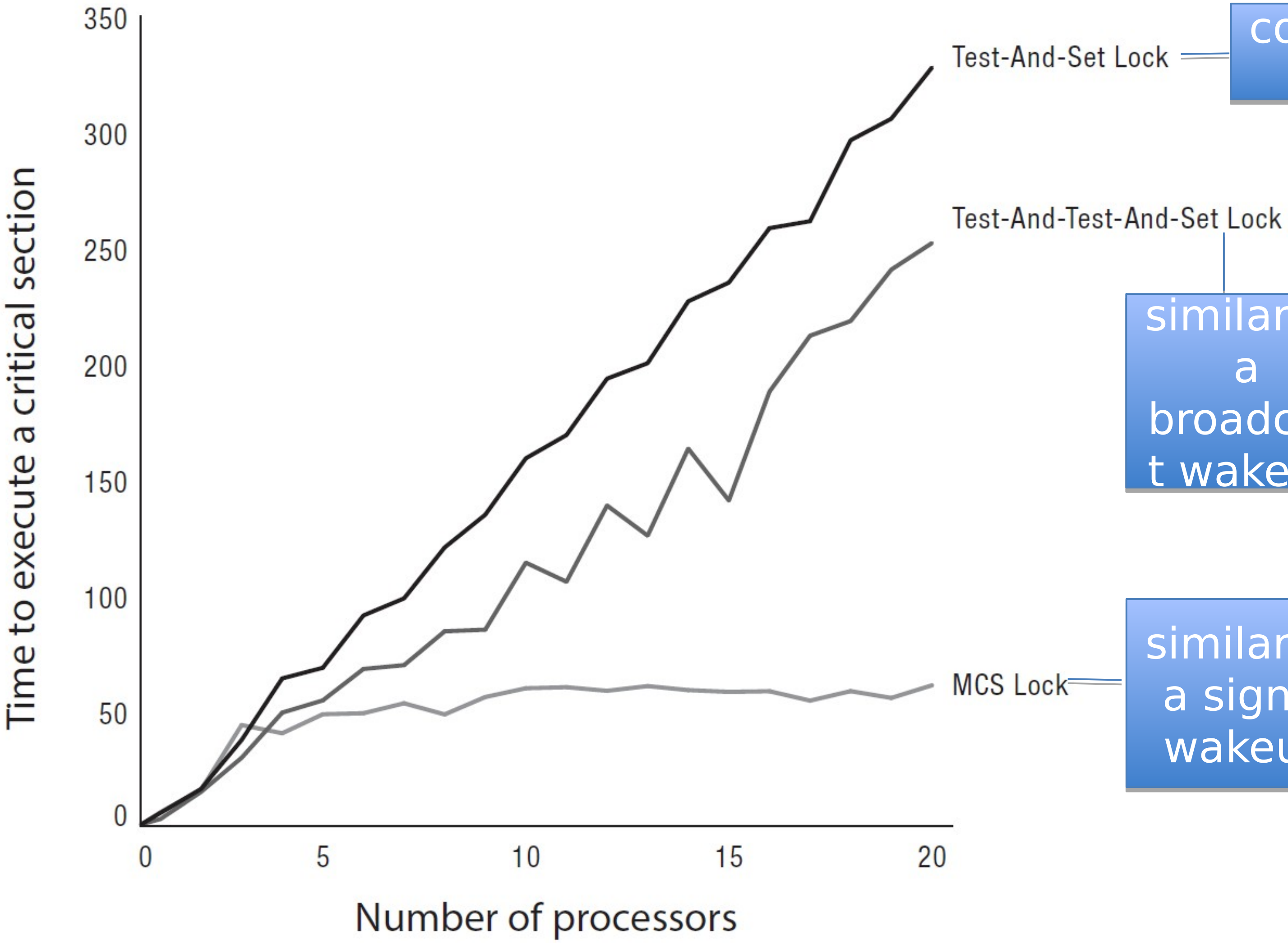
The Problem with Test and Test and Set

```
Counter::Increment() {
    while ( lock == BUSY  ||  TestAndSet(&lock) )
        ;
    value++;
    lock = FREE;
    memory_barrier();
}
```

What if many processors try to acquire the lock?

- Lock value pings between caches

Test (and Test) and Set Performance



contentio n

similar to a broadcas t wakeup

similar to a signal wakeup

Some Approaches

- Insert a delay in the spin loop
  - Helps but acquire is slow when not much contention
- Spin adaptively
  - No delay if few waiting
  - Longer delay if many waiting
  - Guess number of waiters by how long you wait
- MCS
  - Create a linked list of waiters using CompareAndSwap
  - Spin on a per-processor location

Atomic CompareAndSwap

- Operates on a memory word
- Check that the value of the memory word hasn’t changed from what you expect
  - E.g., no other thread did CompareAndSwap first
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value

MCS Lock

- Maintain a list of threads waiting for the lock
  - Front of list holds the lock
  - MCSLock::tail is last thread in list
  - New thread uses CompareAndSwap to add to the tail
- Lock is passed by setting next->needToWait = FALSE;
  - Next thread spins while its needToWait is TRUE

```
TCB {
        TCB *next;                  // next in line
         bool needToWait;
}
MCSLock {
        Queue *tail = NULL; // end of line
}
```

MCS Lock Implementation

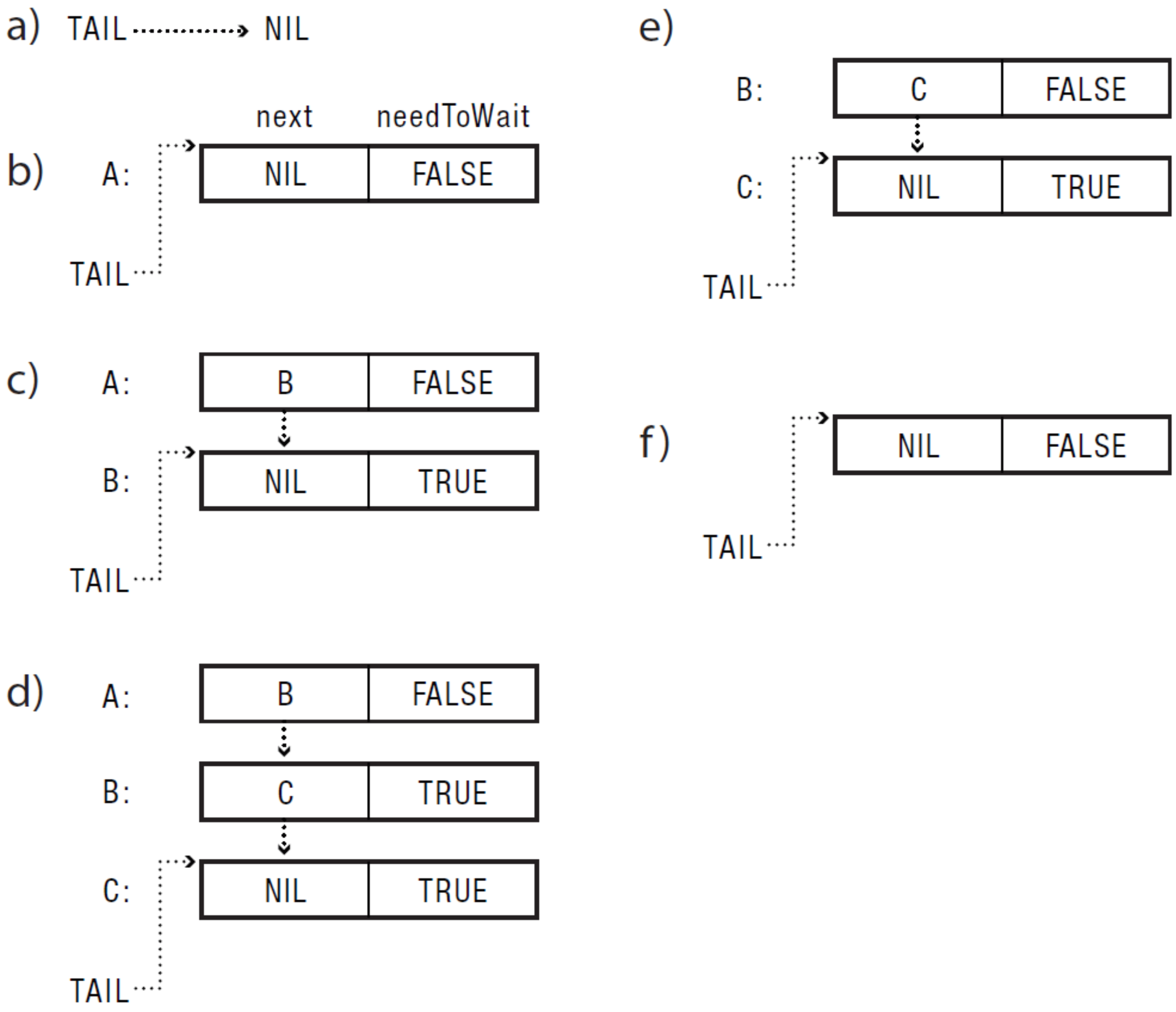
```
MCSLock::acquire() {
    Queue ∗oldTail = tail;

    myTCB−>next = NULL;
    myTCB−>needToWait = TRUE;
    while (!CompareAndSwap(&tail,
   oldTail, &myTCB)) {
         oldTail = tail;
    }
    if (oldTail != NULL) {
        oldTail−>next = myTCB;
      memory_barrier();
        while (myTCB−>needToWait)
            ;
    }
}
```

```
MCSLock::release() {
     if (!
    CompareAndSwap(&tail,
    myTCB, NULL)) {
           while (myTCB−>next
    == NULL)
              ;

    myTCB−>next−>needT
    oWait=FALSE;
     }
}
```

MCS In Operation (1)



MCS In Operation (2)

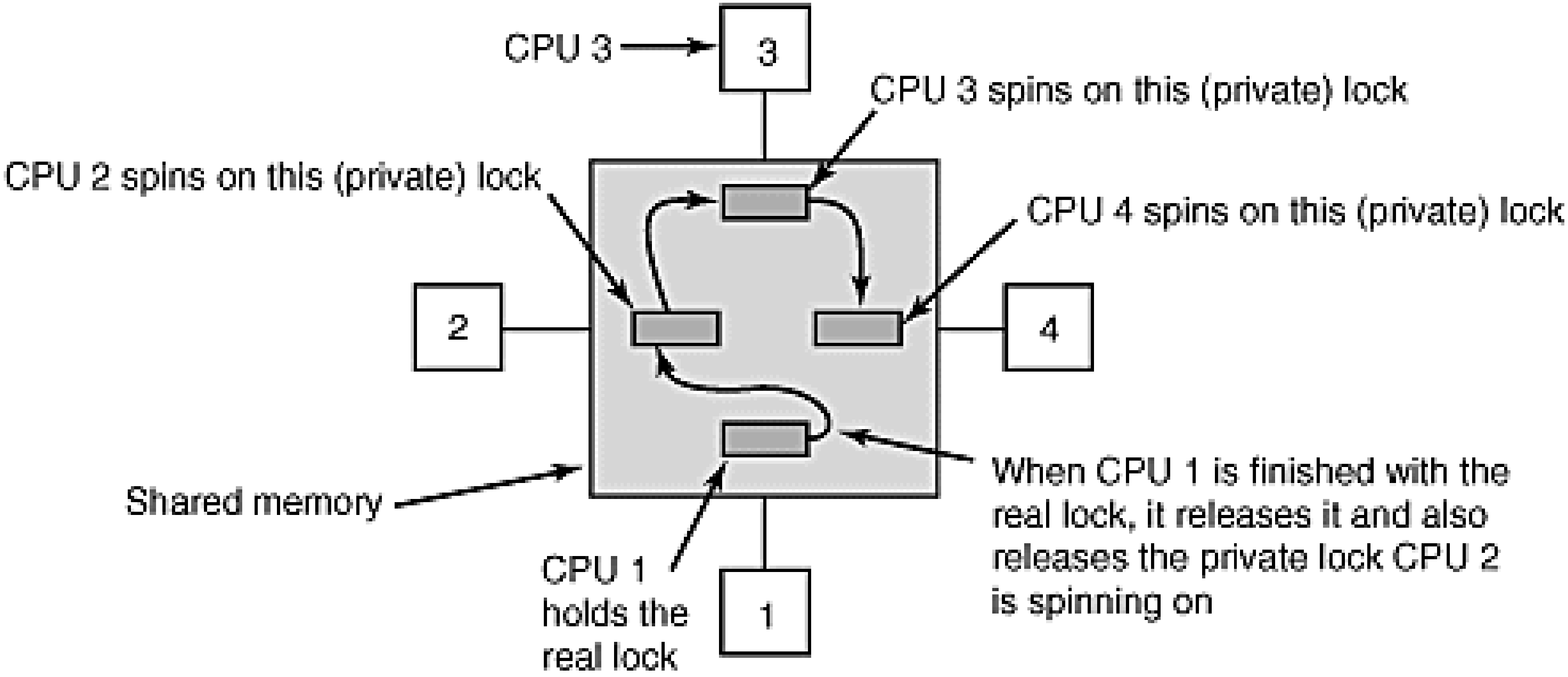
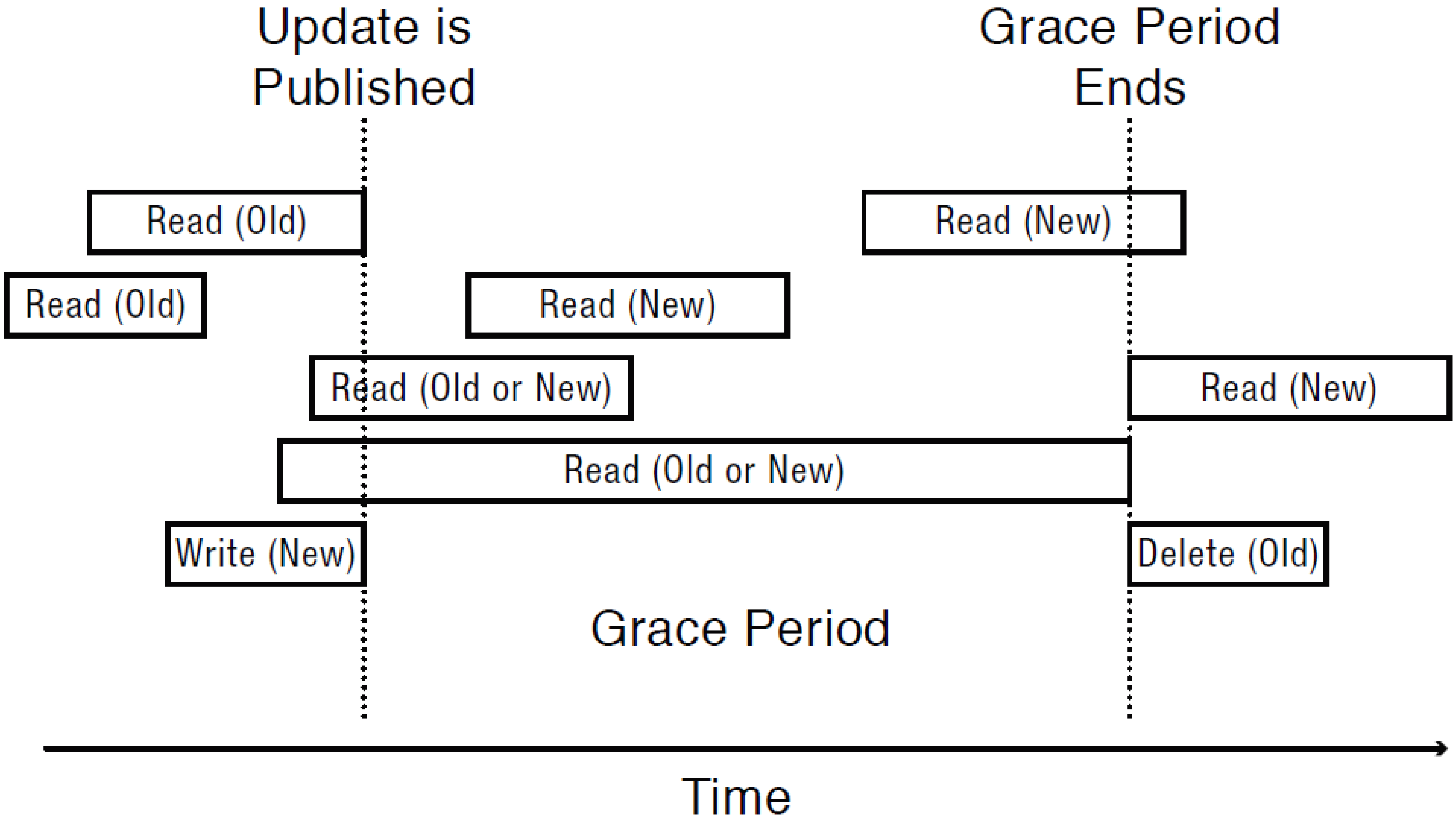

CPU 3
3
CPU 3 spins on this (private) lock
CPU 2 spins on this (private) lock
CPU 4 spins on this (private) lock
2
4
When CPU 1 is finished with the real lock, it releases it and also releases the private lock CPU 2 is spinning on
Shared memory
CPU 1 holds the real lock
1

Diagram from A.S. Tanenbaum, Modern Operating Systems, 2nd ed., as appears in A.S. Tanenbaum, “Multiprocessor Operating Systems,” InformIT, March 22, 2002, http://www.informit.com/articles/article.aspx?p=26027

Read-Copy-Update

- Goal: very fast reads to shared data
  - Reads proceed without first acquiring a lock
  - OK if write is (very) slow
- Restricted update
  - Writer computes new version of data structure
  - Publishes new version with a single atomic instruction
- Multiple concurrent versions
  - Readers may see old or new version
- Integration with thread scheduler
  - Guarantee all readers complete within grace period, and then garbage collect old version

Read-Copy-Update



Read-Copy-Update Implementation

- Readers ask kernel to disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time slice on each CPU
  - Only then, garbage collect old version of data structure

Non-Blocking Synchronization

- Goal: data structures that can be read/modified without acquiring a lock
  - No lock contention!
  - No deadlock!
- General method using CompareAndSwap
  - Create copy of data structure
  - Modify copy
  - Swap in new version iff no one else has
  - Restart if pointer has changed

Optimistic Concurrency Control

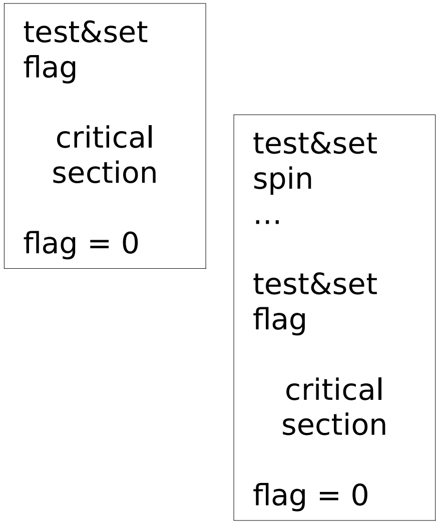
- Allows overlapped execution of updates
  - With forward progress for at least one thread
- If the updates apply to different fields in a protected object, then all can succeed
  - Gives benefit of fine-grain locking without all the locks
  - If same field is changed by another thread between the time a thread reads and writes, then write attempt fails and update has to be restarted using the new value
  - In essence, a critical section has been moved into the body of a busy wait loop

Lock-Free Bounded Buffer

```
tryget() {
    do {
        copy = ConsistentCopy(p);
        if (copy->front == copy->tail)
            return NULL;
        else {
            item = copy->buf[copy->front % MAX];
            copy->front++;
     } while (CompareAndSwap(&p, p, copy));
    return item;
}
```

Comparison of Three Approaches

Spin lock



test&set flag

critical section

flag = 0

test&set spin …

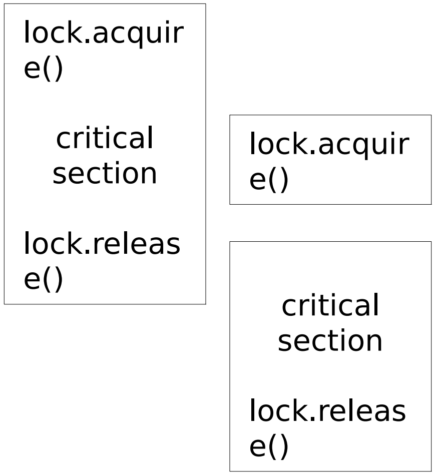
test&set flag

critical section

flag = 0

Second thread busy waits until successful test&set

Queuing lock



lock.acquire()

critical section

lock.release()

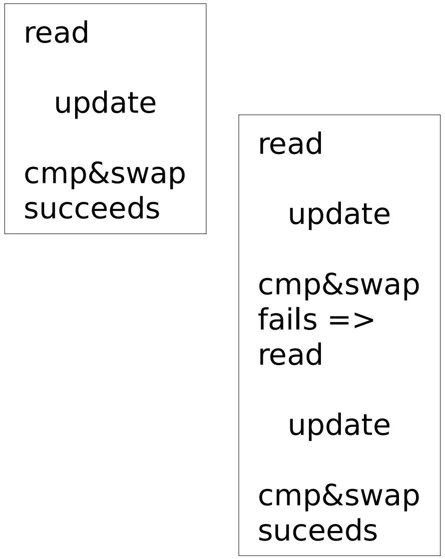
lock.acquire()

critical section

lock.release()

Second thread calls acquire() and is put on the lock’s waiting list; it is put back on the ready list when first thread calls release()

Optimistic concurrency



read

update

cmp&swap succeeds

read

update

cmp&swap fails => read

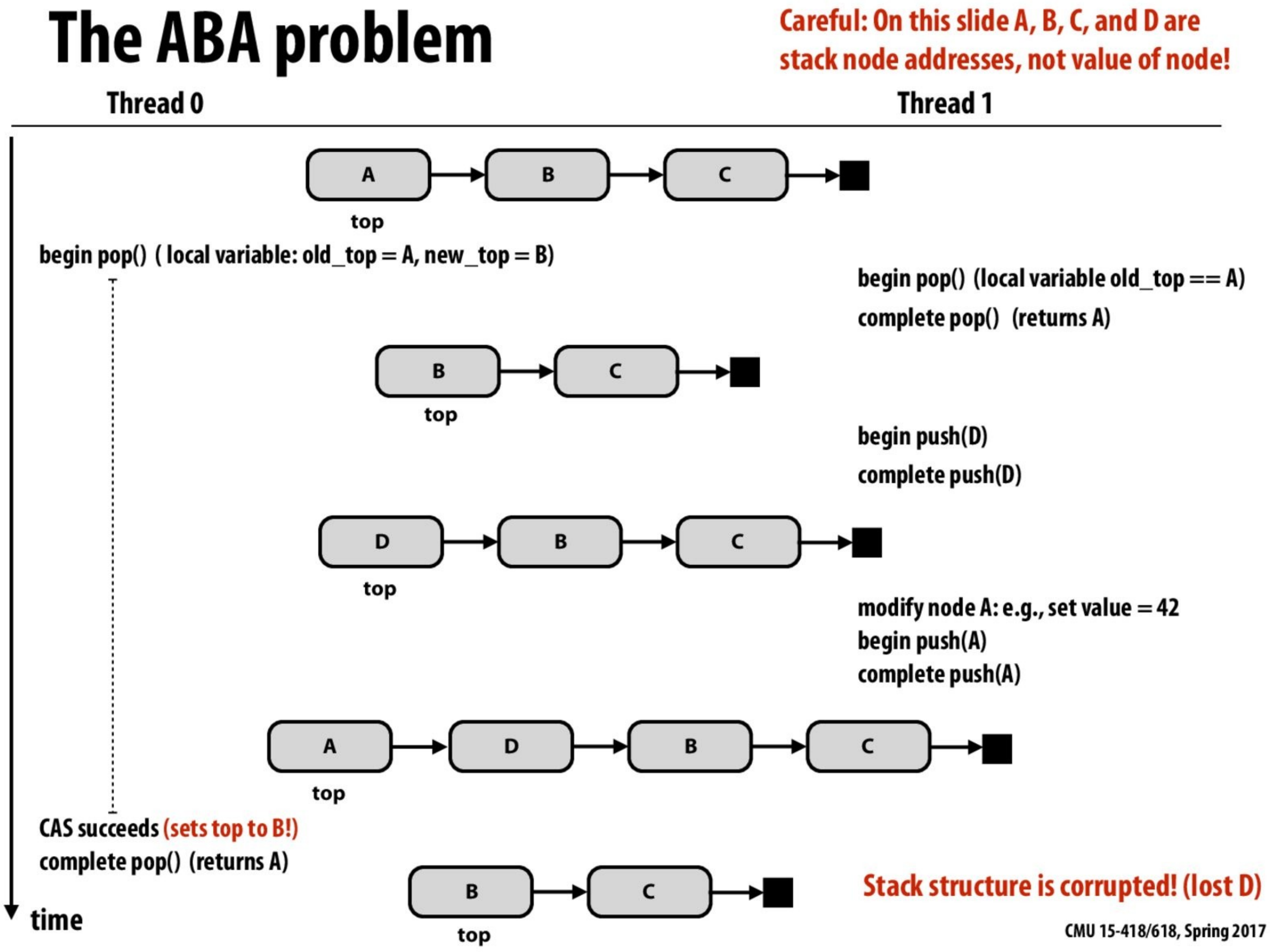
update

cmp&swap suceeds

Second thread fails on cmp&swap and then repeats the update using the new global value

CompareAndSwap ABA Problem

- Intervening actions between copy and update of list head with reuse of an address
- Needs a second word to act as a counter
  - IBM S/370 implemented a single-wide CS and double-wide CDS
  - Intel x86 implemented a single-wide CMPXCHG8B and double-wide CMPXCHG16B



The ABA problem

Careful: On this slide A, B, C, and D are stack node addresses, not value of node!

Thread 0 | Thread 1

begin pop() (local variable: old_top = A, new_top = B)

begin pop() (local variable old_top == A)

complete pop() (returns A)

begin push(D)

complete push(D)

modify node A: e.g., set value = 42

begin push(A)

complete push(A)

CAS succeeds (sets top to B!)

complete pop() (returns A)

Stack structure is corrupted! (lost D)

CMU 15-418/618, Spring 2017

Diagram from
http://15418.courses.cs.cmu.edu/spring2017/lecture/lockfree/slide_027

Hardware Transactional Memory

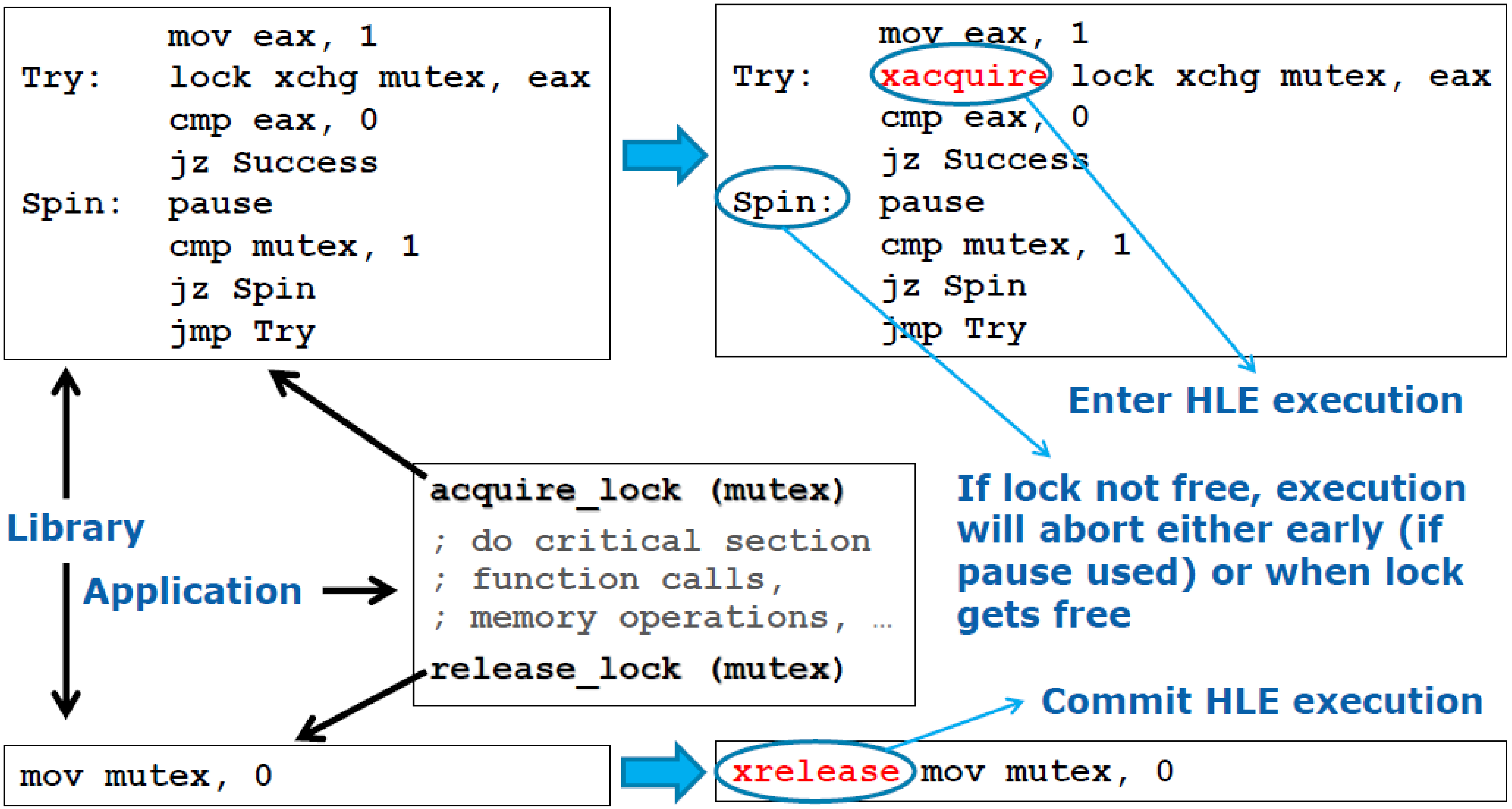
- Execute without acquiring a lock and commit all writes at once
  - Track reads and writes of current processor and any interfering accesses from other processors at cache line granularity
  - If no interfering accesses, commit all updates in a single transaction
  - Otherwise abort and follow a non-transactional recovery path

Hardware Transactional Memory

- Intel TSX (Transactional Synchronization Extensions)
  - HLE – Hardware Lock Elision
    - Prefix bytes added to instructions in legacy code
  - RTM – Restricted Transactional Memory

- IBM z/Architecture TX (Transactional Execution)
  - Constrained transactions guarantee forward progress for at least one thread

Intel® TSX Interface: HLE



```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

```
        mov eax, 1
Try:    xacquire lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

Enter HLE execution

If lock not free, execution will abort either early (if pause used) or when lock gets free

```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

Commit HLE execution

```
mov mutex, 0
```

```
xrelease mov mutex, 0
```

Slide from IDF 2012 presentation by Ravi Rajwar and Martin Dixon, as appears in Johan De Gelas and Cara Hamm, "Making Sense of the Intel Haswell Transactional Synchronization eXtensions," AnandTech, Sept. 20, 2012

(if time permits)

Historical Progression of Transactional Memory

- CompareAndSwap instruction
  - tests a value in a memory location
- LoadLinked and StoreConditional instructions
  - track interfering writes to a memory location
- HW transactional memory
  - tracks interfering accesses to cache lines

Load Linked and Store Conditional

- LL loads the addressed word from memory and places the address into a special register with which the processor bus-snoops

- SC conditionally stores a word into memory
  - The address must be same as that loaded by the last LL
  - The store will succeed (i.e., modify memory and signal success) only if the location has not been modified since it was loaded by the LL
  - The processor is bus-snooping on this address, any writes from other processors to this address will be detected
  - The store will fail (i.e., not modify memory, signal failure) if the location has been modified since the LL or if the processor has context-switched
  - Success indicated by 1 in register; 0 otherwise

- You can build higher-level synchronization constructs out of these primitives

LL/SC Example

```
// increment counter example – optimistic concurrency

    // r1 points to counter
lp: ll   r2,(r1)     // load linked counter into r2
    addi r3,r2,1     // r3 gets counter plus 1
    sc   r3,(r1)     // conditionally update counter
    beq  r3,0,lp     // test if sc successful
    nop
```