EECS 570
Lecture 5
Synchronization

Winter 2020
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch. Some slides derived from Herlihy & Shavit “The Art of Multiprocessor Programming” used under http://creativecommons.org/licenses/by-sa/3.0/
Announcements

Programming Assignment 1 due Friday 2/7 11:59pm
• Upload zip in Canvas

Project Proposals due Feb 5th

Project kick-off meetings on Feb 6\textsuperscript{th}-7\textsuperscript{th}.

Post your group info on Piazza (have you all signed up? ...)
Readings

For Today:

- Michael Scott. *Shared-Memory Synchronization*. Morgan & Claypool Synthesis Lectures on Computer Architecture (Ch. 1, 4.0-4.3.3, 5.0-5.2.5).


For Monday, 3rd Feb:


Synchronization objectives

• Low overhead
  - Synchronization can limit scalability
    (E.g., single-lock OS kernels)

• Correctness (and ease of programmability)
  - Synchronization failures are extremely difficult to debug

• Coordination of HW and SW
  - SW semantics must be tightly specified to prove correctness
  - HW can often improve efficiency
Synchronization Forms

- Mutual exclusion (critical sections)
  - Lock & Unlock
- Event Notification
  - Point-to-point (producer-consumer, flags)
  - I/O, interrupts, exceptions
- Barrier Synchronization

- Higher-level constructs
  - Queues, software pipelines, (virtual) time, counters

- Next lecture: optimistic concurrency control
  - Transactional Memory
Anatomy of a Synchronization Op

• Acquire Method
  - Way to obtain the lock or proceed past the barrier

• Waiting Algorithm
  - Spin (aka busy wait)
    - Waiting process repeatedly tests a location until it changes
    - Releasing process sets the location
    - Lower overhead, but wastes CPU resources
    - Can cause interconnect traffic
  - Block (aka suspend)
    - Waiting process is descheduled
    - High overhead, but frees CPU to do other things
  - Hybrids (e.g., spin, then block)

• Release Method
  - Way to allow other processes to proceed
HW/SW Implementation Trade-offs

- User wants high-level (ease of programming)
  - LOCK(lock_variable); UNLOCK(lock_variable)
  - BARRIER(barrier_variable, numprocs)
- SW advantages: flexibility, portability
- HW advantages: speed
- Design objectives:
  - Low latency
  - Low traffic
  - Low storage
  - Scalability ("wait-free"-ness)
  - Fairness
Challenges

• Same sync may have different behavior at different times
  - Lock accessed with low or high contention
  - Different performance needs: low latency vs. high throughput
  - Different algorithms best for each, need different primitives

• Multiprogramming can change sync behavior
  - Process scheduling or other resource interactions
  - May need algorithms that are worse in dedicated case

• Rich area of SW/HW interactions
  - Which primitives are available?
  - What communication patterns cost more/less?
Locks
Lock-based Mutual Exclusion

Synchronization period

- Acquire starts
- Acquire done
- Release starts
- Release done

No contention:
- Want low latency

Contention:
- Want low period
- Low traffic
- Fairness
How Not to Implement Locks

• **LOCK**

  
  while (lock_variable == 1);

  lock_variable = 1;

• **UNLOCK**

  lock_variable = 0;
Solution: Atomic Read-Modify-Write

- **Test&Set(r,x)**
  
  \[
  \begin{array}{l}
  r = m[x]; \\
  m[x] = 1;
  \end{array}
  \]
  
  - \(r\) is register
  - \(m[x]\) is memory location \(x\)

- **Fetch&Op(r1,r2,x,op)**
  
  \[
  \begin{array}{l}
  r1 = m[x]; \\
  m[x] = \text{op}(r1, r2);
  \end{array}
  \]

- **Swap(r,x)**
  
  \[
  \begin{array}{l}
  \text{temp} = m[x]; \\
  m[x] = r; \\
  r = \text{temp};
  \end{array}
  \]

- **Compare&Swap(r1,r2,x)**
  
  \[
  \begin{array}{l}
  \text{temp} = r2; \\
  r2 = m[x]; \\
  \text{if } r1 == r2 \text{ then } m[x] = \text{temp};
  \end{array}
  \]
Implementing RMWs

• Bus-based systems:
  - Hold bus and issue load/store operations without any intervening accesses by other processors

• Scalable systems
  - Acquire exclusive ownership via cache coherence
  - Perform load/store operations without allowing external coherence requests
Load-Locked Store-Conditional

- **Load-locked**
  - Issues a normal load...
  - ...and sets a flag and address field

- **Store-conditional**
  - Checks that flag is set and address matches...
  - ...only then performs store

- **Flag is cleared by**
  - Invalidation
  - Cache eviction
  - Context switch

```
lock: while (1) {
    load-locked r1, lock_variable
    if (r1 == 0) {
        mov r2 = 1
        if (SC r2, lock) break;
    }
}

unlock: st lock_variable, #0
```
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 Lock (MCS)

- A “lock” is a pointer to a linked list node
  - next node pointer
  - boolean must_wait
  - Each thread has its own local pointer to a node “I”

- acquire(lock):
  I->next = null;
  predecessor = fetch_and_store(lock,I)
  if predecessor != nil //some node holds lock
    I->must_wait = true
    predecessor->next = I //predecessor must wake us
  repeat while I->must_wait //spin till lock is free

- release(lock):
  if (I->next == null) //no known successor
    if compare_and_swap(lock,I,nil) //make sure...
      return //CAS succeeded; lock freed
  repeat while I->next = nil //spin to learn successor
  I->next->must_wait = false //wake successor
MCS Lock Example: Time 0

- acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
    pred->next = I
  repeat while I->must_wait

- release(lock):
  if (I->next == null)
    if CAS(lock,I,nil)
      return
  repeat while I->next == nil
  I->next->must_wait = false
MCS Lock Example: Time 1

- \( t_1 \): Acquire(L)

**acquire(lock):**
- \( I->next = \text{null} \)
- \( \text{pred} = \text{FAS}(\text{lock}, I) \)
- if \( \text{pred} \neq \text{nil} \)
  - \( I->must\_wait = \text{true} \)
  - \( \text{pred}->next = I \)
  - repeat while \( I->must\_wait \)

**release(lock):**
- if \( (I->next == \text{null}) \)
  - if \( \text{CAS}(\text{lock}, I, \text{nil}) \)
    - return
  - repeat while \( I->next == \text{nil} \)
  - \( I->next->must\_wait = \text{false} \)
MCS Lock Example: Time 2

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

acquire(lock):
- $I->next = \text{null}$;
- $\text{pred} = \text{FAS}(\text{lock}, I)$
  - if $\text{pred} \neq \text{nil}$
    - $I->must\_wait = \text{true}$
    - $\text{pred->next} = I$
    - repeat while $I->must\_wait$

release(lock):
- if ($I->next == \text{null}$)
  - if CAS(lock, I, nil)
    - return
  - repeat while $I->next == \text{nil}$
- $I->next->must\_wait = \text{false}$
MCS Lock Example: Time 3

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

### acquire(lock):

```
I->next = null;
pred = FAS(lock,I)
if pred != nil
    I->must_wait = true
pred->next = I
repeat while I->must_wait
```

### release(lock):

```
if (I->next == null)
    if CAS(lock,I,nil)
        return
    repeat while I->next == nil
I->next->must_wait = false
```
MCS Lock Example: Time 4

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

- **acquire(lock):**
  - $I\rightarrow$next = null;
  - pred = FAS(lock,$I$)
  - if pred != nil
    - $I\rightarrow$must_wait = true
  - pred->next = $I$
  - repeat while $I\rightarrow$must_wait

- **release(lock):**
  - if ($I\rightarrow$next == null)
    - if CAS(lock,$I$,nil)
      - return
    - repeat while $I\rightarrow$next == nil
  - $I\rightarrow$next->must_wait = false
MCS Lock Example: Time 5

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

\[\text{acquire(lock)}: \]
\[
\text{I->next = null;}
\]
\[
\text{pred = FAS(lock,I)}
\]
\[
\text{if pred != nil}
\]
\[
\text{I->must_wait = true}
\]
\[
\text{pred->next = I}
\]
\[
\text{repeat while I->must_wait}
\]

\[\text{release(lock)}: \]
\[
\text{if (I->next == null)}
\]
\[
\text{if CAS(lock,I,nil)}
\]
\[
\text{return}
\]
\[
\text{repeat while I->next == nil}
\]
\[
\text{I->next->must_wait = false}
\]
MCS Lock 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(lock):
  
  I->next = null;
  
  pred = FAS(lock, I)
  
  if pred != nil
    
    I->must_wait = true
    p

  pred->next = I
  repeat while I->must_wait

• release(lock):
  
  if (I->next == null)
    
    if CAS(lock, I, nil)
      return
      repeat while I->next == nil
      I->next->must_wait = false

release() w/o CAS is more complex; see paper
Queue-based locks in HW: QOLB

• **Queue On Lock Bit**
  - HW maintains doubly-linked list between requesters
    - This is a key idea of “Scalable Coherence Interface”, see Unit 3
  - Augment cache with “locked” bit
    - Waiting caches spin on local “locked” cache line
  - Upon release, lock holder sends line to 1\textsuperscript{st} requester
    - Only requires one message on interconnect
Fundamental Mechanisms to Reduce Overheads

[Kägi, Burger, Goodman ASPLOS 97]

- **Basic mechanisms**
  - Local Spinning
  - Queue-based locking
  - Collocation
  - Synchronous Prefetch

<table>
<thead>
<tr>
<th></th>
<th>Local Spin</th>
<th>Queue</th>
<th>Collocation</th>
<th>Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>No</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>Yes</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>Yes</td>
<td>Optional</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

![Graph showing the relative sync period against the number of CPUs for T&S, T&T&S, MCS, and QOLB. The graph is based on data from Kägi (1997).]
Performance of Locks

• Contention vs. No Contention
  - Test-and-Set best when no contention
  - Queue-based is best with medium contention
  - Idea: switch implementation based on lock behavior
    - Reactive Synchronization – Lim & Agarwal 1994
    - SmartLocks – Eastep et al 2009

• High-contention indicates poorly written program
  - Need better algorithm or data structures
Point-to-Point Event Synchronization

• Can use normal variables as flags
  
  \[
  a = f(x); \\
  \text{flag} = 1;
  \]

  while (flag == 0);

  b = g(a);

• If we know initial conditions
  
  \[
  a = f(x); \\
  \text{while (a == 0);}
  \]

  b = g(a);

• Assumes Sequential Consistency!

• Full/Empty Bits
  
  - Set on write
  - Cleared on read
  - Can’t write if set, can’t read if clear
Barriers
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

Step 1

Step 2

Step 3
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it

- Strawman implementation (**wrong!**)

```plaintext
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?
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {

            }
        }
        threadSense.set(!mySense)}}}
Sense-Reversing Barriers

```java
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} } else {
                while (sense != mySense) {} } } }]
```

Completed odd or even-numbered phase?
Sense-Reversing Barriers

```java
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    threadSense.set(!mySense)
}
```
Sense-Reversing Barriers

public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<>

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    }

    threadSense.set(!mySense)
}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)})();
}
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
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}...
        }...
    }}
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent;
    Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
```

Parent barrier in tree
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }...}
public class Node {
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }
    }
}

Proceed to parent barrier
Combining Tree Barrier

public class Node{
    AtomicInteger count; int size,
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            } else {
                while (sense != mySense) {} ...
            }
            count.set(size);
            sense = mySense
        }
    }...
}
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {…
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}{
        }…}}

**Notify others at this node**
Combining Tree Barrier

```java
public class Node {
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense;
        } else {
            while (sense != mySense) {}
        }
    }
```

*I'm not last, so wait for notification*
Combining Tree Barrier

- No sequential bottleneck
  - Parallel getAndDecrement() calls

- Low memory contention
  - Same reason

- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for NUMA