EECS 570
Lecture 6
Synchronization I
Winter 2021
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.
Readings

For Wednesday:


Synchronization
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
  r Lock accessed with low or high contention
  r Different performance needs: low latency vs. high throughput
  r Different algorithms best for each, need different primitives

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

• Rich area of SW/HW interactions
  r Which primitives are available?
  r 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**
  ```c
  while (lock_variable == 1);
  lock_variable = 1;
  ```

• **UNLOCK**
  ```c
  lock_variable = 0;
  ```

Context switch!
Solution: Atomic Read-Modify-Write

• Test&Set(r,x)
  \{r=m[x]; m[x]=1;\}

  • r is register
  • m[x] is memory location x

• Fetch&Op(r1,r2,x,op)
  \{r1=m[x]; m[x]=op(r1,r2);\}

• Swap(r,x)
  \{temp=m[x]; m[x]=r; r=temp;\}

• Compare&Swap(r1,r2,x)
  \{temp=r2; r2=m[x]; if r1==r2 then m[x]=temp;\}
Implementing RMWs

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

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

• Loadlocked
  r Issues a normal load...
  r ...and sets a flag and address field

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

• Flag is cleared by
  r Invalidation
  r Cache eviction
  r 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
  r Less of a “thundering herd” coherence storm problem
    m To acquire, only need to read new value of now_serving
  r No CAS on critical path of lock handoff
    m Just a non-atomic store
  r FIFO order (fair)
    m Good, but only if the O.S. hasn’t swapped out any threads!

• Padding
  r Allocate now_serving and next_ticket on different cache blocks
    m struct { int now_serving; char pad[60]; int next_ticket; } ...
  r Two locations reduces interference

• Proportional backoff
  r 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):**
  1. \( \text{I->next} = \text{null;} \)
  2. \( \text{pred} = \text{FAS(lock,I)} \)
  3. if \( \text{pred} \neq \text{nil} \)
     1. \( \text{I->must_wait} = \text{true} \)
     2. \( \text{pred->next} = \text{I} \)
     3. repeat while \( \text{I->must_wait} \)

- **release(lock):**
  1. if \( \text{(I->next == null)} \)
     1. if \( \text{CAS(lock,I,nil)} \)
     2. return
     3. repeat while \( \text{I->next == nil} \)
  2. \( \text{I->next->must_wait} = \text{false} \)
MCS Lock Example: Time 1

• $t_1$: 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 2

- $t_1$: Acquire(L)
- $t_2$: 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 3

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

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

- \textbf{release(lock):}
  
  $$\text{if } (I->next == \text{null})$$
  $$\quad \text{if } \text{CAS(lock,I,nil)}$$
  $$\quad \text{return};$$
  $$\quad \text{repeat while } I->next == \text{nil};$$
  $$\quad I->next->must\_wait = \text{false};$$
MCS Lock Example: Time 4

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

\[
\begin{align*}
\text{I}_1 & \rightarrow \text{False} \\
\text{I}_2 & \rightarrow \text{False} \\
\text{I}_3 & \rightarrow \text{True}
\end{align*}
\]

- 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 5

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

```python
• 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 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):**

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

**release(lock):**

- if $(I->next == \text{null})$
  - if CAS(lock, I, nil)
  - return
- repeat while $I->next == \text{nil}$
  - $I->next->\text{must}_\text{wait} = \text{false}$

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\(^{st}\) 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>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>Optional</td>
<td>No</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>Yes</td>
<td>No</td>
<td>Optional</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>Yes</td>
<td>Optional</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

![Graph showing relative sync period vs number of CPUs for different synchronization methods: T&S, T&T&S, MCS, QOLB. The graph indicates the performance of these methods as the number of CPUs increases. The reference is Kägi 97.](image)
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); \quad \text{while} \ (\text{flag} == 0);
  
flag = 1; \quad b = g(a);

• If we know initial conditions
  
a = f(x); \quad \text{while} \ (a == 0);
  
\quad b = g(a);

• Assumes Sequential Consistency!

• Full/Empty Bits
  
  r Set on write
  
  r Cleared on read
  
  r 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?
Sense-Reversing Barrier

• Correct barrier implementation:

global (shared) count : integer := P

global (shared) sense : Boolean := true

local (private) local_sense : Boolean := true

procedure central_barrier

    // each processor toggles its own sense

    local_sense := !local_sense

    if fetch_and_decrement(&count) == 1
        count := P

        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense == local_sense
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