# Advanced Computer Architecture

## Synchronization

**Fall 2016**

**Pejman Lotfi-Kamran**

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi and Wenisch of CMU, EPFL, Michigan, Wisconsin



## Where Are We?

| Fr | Sa | Su | Mo | Tu |
|---|---|---|---|---|
| | 27-Shahrivar | | 29-Shahrivar | |
| | 3-Mehr | | 5-Mehr | |
| | 10-Mehr | | 12-Mehr | |
| | 17-Mehr | | 19-Mehr | |
| | 24-Mehr | | 26-Mehr | |
| | 1-Aban | | 3-Aban | |
| | 8-Aban | | 10-Aban | |
| | 15-Aban | | 17-Aban | |
| | 22-Aban | | 24-Aban | |
| | 29-Aban | | 1-Azar | |
| | 6-Azar | | 8-Azar | |
| | 13-Azar | | 15-Azar | |
| | 20-Azar | | 22-Azar | |
| | 27-Azar | | 29-Azar | |
| | 4-Dey | | 6-Dey | |

- This Lecture
  - Synchronization
- Next Lecture:
  - Synchronization



## 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



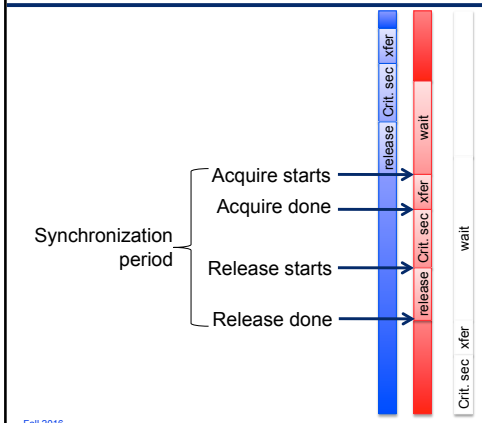
## Challenges

- Same sync may have different behavior at different times
  - 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




No contention:
- Want low latency

Contention:
- Want low period
- Low traffic
- Fairness



## HW/SW Implementation Tradeoffs

- User wants:
  - high level (ease of programming)
    - ▲LOCK(lock_variable), UNLOCK(lock_variable)
    - ▲BARRIER(barrier_variable, Num_Procs)
  - low cost:
    - ▲Performance: latency, traffic
    - ▲Implementation: storage overhead
  - scalability
  - fairness
- Hardware
  - for speed (it's fast)
- Software
  - for flexiblilty



## How Not to Implement Locks

- LOCK

```
while (lock_variable == 1);
lock_variable = 1
```


- UNLOCK

```
lock_variable = 0;
```

## Solution: Atomic Read-Modify-Write

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

- Test&Set(r,x)
  `{r=m[x]; m[x]=1;}`
- 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;}`



## Test&Set in SPARC

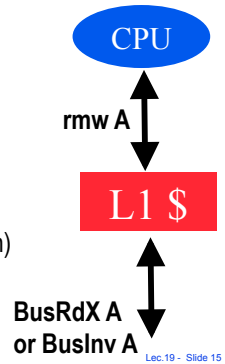
```
test_and_set()
   L1:  ldstub  [%l1], %l0      ; lock = 255 (set)
        bne L1

unset()
        st       %g0, [%l1]     ; lock = 0
```



## What does Test&Set do to the system?

- rmw = generic name for all read-modify-write instructions
- rmw requires write permission
- If no contention
  - To lock, if hit rmw
  - If miss, get A with write permission, then rmw
  - To unlock, if hit, store
  - If miss, get A with write permission, then store
- With contention (multiple processors spin)
  - Every test is a store miss!!!!
  - Every unlock is a store miss






## Better Lock Implementations

- Two choices:
  - Don't execute test&set so much
  - Spin without generating bus traffic
- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Exponential seems good ($k*c^i$)
  - Not fair
- Test&Test&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue test&set
  - Intuition: No point in trying to set the location until we know that it's not set, which we can detect when it get invalidated...
  - Still contention after invalidate
  - Still not fair

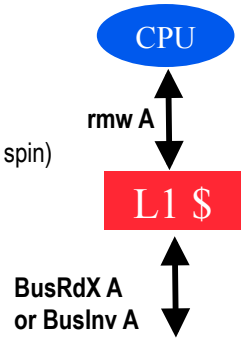
## Test&Set with Backoff

```
test_and_set_withbackoff()
L1:     ldstub  [%l1], %l0      ; lock = 255 (set)
        beq     continue        ; if (!z) not set, loop
        call    _sleep          ; loops an exponential # of cycles
        br      L1
continue:
                                ; lock acquired
unset()
        st      %g0, [%l1]      ; lock = 0
```



## What does Test&Set with Backoff do?

- If no contention
  - Acts like Test & Set
- With contention (multiple processors spin)
  - Every test is a store miss!!!!
  - Every unlock is a store miss
  - But the tests are exponentially distributed






## Test&Test&Set in SPARC

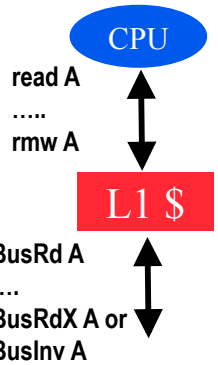
```
test_and_test_and_set()
L1:     ld          [%l1], %l0      ; check first
        bne         L1              ; if (z) not set, loop
        ldstub      [%l1], %l0      ; lock = 255 (set)
        bne         L1

unset()
        st          %g0, [%l1]      ; lock = 0
```



## What does Test&Test&Set do?

- If no contention
  - To lock, read, if hit and available then rmw
  - If miss or not available, spin reading
  - To unlock, if hit, store
  - If miss, get A with write permission, then store
- With contention (multiple processors spin)
  - Most tests are read hits
  - Writes only when acquiring the lock
  - And releasing the lock
  - Wait using reads



## Load-Locked Store-Conditional (Alpha, ARM)

- 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



## Test&Set in Alpha (with Local Spinning)

```
test_and_set()
L1: ldl_l   t0, 0(t1)     ; load locked, t0 = lock
    bn      t0, L1        ; if not free, loop
    lda     t0, 1(0)      ; t0 = 1
    stl_c   t0, 0(t1)     ;conditional store,
                          ; lock = 1
    beq     t0, L1        ; if failed, loop

unset()
    stl     0, 0(t1)
```



## Performance of Test&Set

LOCK

while (test&set(x) == 1);

UNLOCK

x = 0;

- High contention (many processes want lock)
- Remember the CACHE!
- Each Test&Set is a read miss and a write miss
  - Not fair
- Problem is?
  - Waiting Algorithm!



## Ticket Locks

- To ensure fairness and reduce coherence storms
- Locks have two counters: `next_ticket, now_serving`
  - Waiting in line in banks

```
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: (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

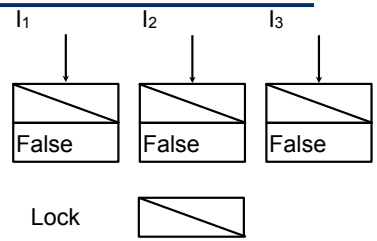


## List-Based Queue Lock (MCS)

- 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



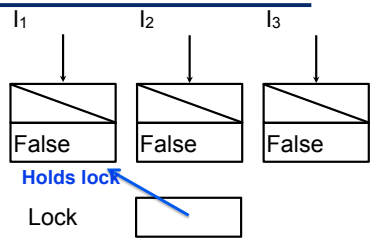

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)



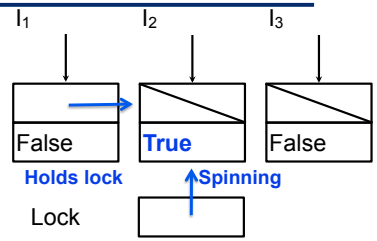

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)



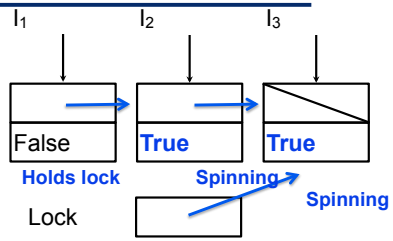

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)

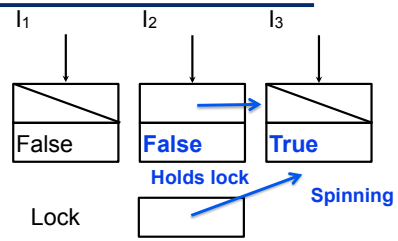



```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)



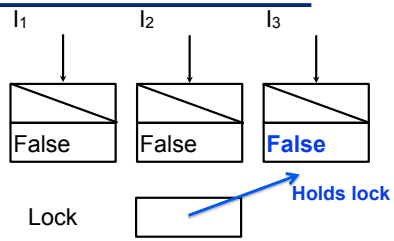

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)



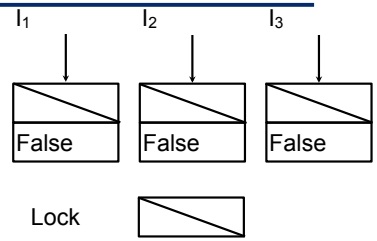

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

```
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)



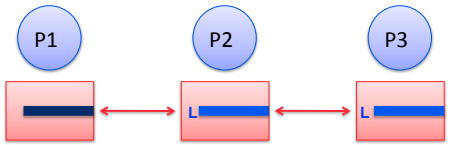

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
      repeat while I-
      >must_wait
```

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



## 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"
  - Augment cache with "locked" bit
    - ▲Waiting caches spin on local "locked" cache line
  - Upon release, lock holder sends line to 1st requester
    - ▲Only requires one message on interconnect



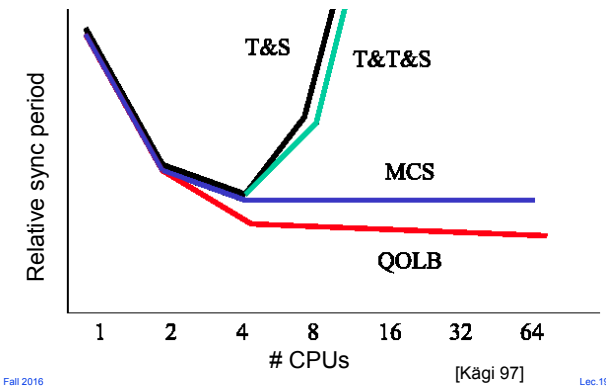
## Fundamental Mechanisms to Reduce Overheads
[Kägi, Burger, Goodman ASPLOS 97]

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

| | Local Spin | Queue | Collocation | Prefetch |
|---|---|---|---|---|
| T&S | No | No | Optional | No |
| T&T&S | Yes | No | Optional | No |
| MCS | Yes | Yes | Partial | No |
| QOLB | yes | Yes | Optional | Yes |



## Microbenchmark Analysis




[Kägi 97]



## 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);                    while (flag == 0);
flag = 1;                    b = g(a);
```

- If we know initial conditions

```
a = f(x);                    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

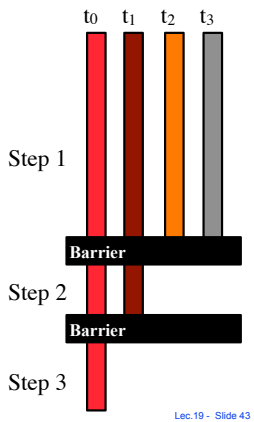
- Physics simulation computation
  - Divide up each timestep computation into N independent pieces
  - Each timestep: compute independently, synchronize
- Example: each thread executes:

```
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





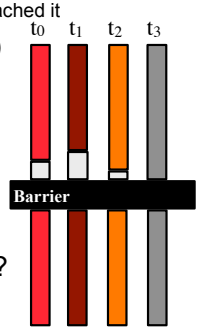

## Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it
- Strawman implementation (**wrong!**)

```
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
```

- Single counter makes this a "centralized" barrier

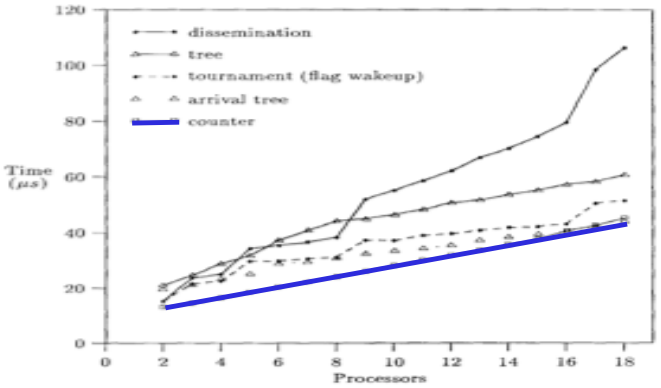


## Other Barrier Implementations

- 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



## Barrier Performance (from 1991)




[Mellor-Crummey & Scott, ACM TOCS, 1991]