Shared-Memory Synchronization

Nima Honarmand

Why Synchronization?

- Concurrent threads are sequences of instructions
- Threads communicate by reading/writing shared memory locations
- Certain inter-thread interleaving of memory operations are not desirable

Synchronization is the art of precluding interleavings [of memory operations] that we consider incorrect

- Most common synchronization goals:
  - Atomicity
  - Condition synchronization

Common Forms of Synchronization

- Basic synchronization primitives
  - Mutual exclusion: locks, mutexes, semaphores, monitors, …
  - Consensus: barriers, eureka, …
  - Conditions: flags, condition variables, signals, …
- Advanced synchronization mechanisms
  - Transactional memory
  - Futures
  - Read-Copy Update (RCU)
  - Lock-free concurrent data structures
- Each can be implemented in hardware or software
  - Some hardware support is commonly needed for efficient software implementations

Basic Synchronization Primitives

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 condition until it changes
    - Releasing process sets the condition
    - 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
    - Requires interaction with scheduler (kernel or usermode)
  - Hybrids (e.g., spin for a while, then block)
- ***Release*** Method
  - Way to allow other processes to proceed

Atomic Read-Modify-Write (RMW) Primitives

- Assuming cache coherence, it is possible to devise synchronization algorithms using ordinary load/store instructions
  - Examples: Peterson and Bakery algorithms for mutual exclusion

- ✘ Time/space complexity rapidly increases with # threads
- ✘ Most algorithms need to know (maximum) # threads a priori

```
class lock
  bool choosing[T] := {false… }
  int number[T] := {0…}

lock.release():
  number[self] = 0
```

```
lock.acquire():
  choosing[self] = true
  int m := 1 + max(number)
  number[self] = m
  choosing[self] = false
  for i in 1..T
    while choosing[i]; // spin
    repeat
      int t := number[i]; // spin
    until t = 0 or <t,i> >= <m, self>
```

Example: Bakery Algorithm

Atomic Read-Modify-Write (RMW) Primitives

- Being able to read/modify/write memory locations atomically enables simpler, more scalable algorithms
  - RMW: Read old value; modify it; write the modified value back; return the old value
- Most modern synchronization implementations rely on RMW primitives
  - Any multiprocessor HW provides some form of atomic RMW
  - Software can implement the rest using HW ones

Examples of Atomic RMW Primitives

r: register
m[x]: memory location x

- Test&Set(r, x)

  `{r=m[x]; m[x]=1;}`

- Fetch&Op(r, x, op)

  `{r=m[x]; m[x]=op(r);}`

- Swap(r, x)

  `{temp=m[x]; m[x]=r; r=temp;}`

- Compare&Swap or CAS(r_{src}, r_{val}, x)

  {temp=m[x]; if temp==r_{val} then m[x]=r_{src}; return temp==r_{val}}

Implementing Atomic RMWs in HW

- Uniprocessors:
  - Disable interrupts before atomic inst (to avoid context switching)
  - Enable interrupts after atomic inst
- Bus-based multiprocessors:
  - Hold bus and issue load/store operations without any intervening accesses by other processors

  → ***Bus Locking***
- Scalable systems:
  - Acquire exclusive ownership via cache coherence
  - Perform load/store operations without allowing external coherence requests

  → ***Cache Locking***

Load Linked/Store Conditional (LL/SC)

- Bus/cache locking complicates HW implementation
- Alternative: Use 2 instructions; the second one's return value indicates whether the pair was executed "atomically"

1. Load Linked (LL)
   – Issues a normal load + starts monitoring the cache line (by setting a flag)
2. Store Conditional (SC)
   – If the flag is still set, then performs the store
   – **If successful return 1, else 0**

- Flag is cleared by
  – Invalidation
  – Cache eviction
  – Context switch
  – Interrupts (in some processors)

Load-Linked/Store Conditional (LL/SC)

- Effectively, SC's return value indicates whether the pair was executed atomically
- LL/SC is a ***universal*** primitive: all other ones can be implemented using LL/SC
  - So is CAS

```
int Fetch&OP(function OP, int *w):
  int old, new
  repeat
    old := LL(w)
    new := OP(old)
    until SC(w, new)
  return old
```

Example: Fetch&OP using LL/SC

Locks

Test-and-Set Spin Lock (TS)

- Lock is "acquire", Unlock is "release"

```
acquire(lock_ptr):
  while (true):
    // Perform "test-and-set"
    old = CAS(lock_ptr, UNLOCKED, LOCKED)
    if (old == UNLOCKED):
      break   // lock acquired!

release(lock_ptr):
  *lock_ptr = UNLOCKED
```

- Could have used other RMW primitives instead of CAS
  - Like Test&Set or Fetch&Inc
- **Performance problem**
  - RMW is both a read and write → spinning causes lots of invalidations
  - Lots of traffic and cache misses

Test-and-Test-and-Set Spin Lock (TTS)

```
acquire(lock_ptr):
  while (true):
    // Perform "test"
    original_value = *lock_ptr
    if (original_value == UNLOCKED):
      // Perform "test-and-set"
      old = CAS(lock_ptr, UNLOCKED, LOCKED)
      if (old == UNLOCKED):
        break   // lock acquired!

release(lock_ptr):
  *lock_ptr = UNLOCKED
```

- Now "spinning" is read-only, on local cached copy
  - Reduces invalidations and cache misses compared to TS lock

TTS Lock Performance Issues

- Performance issues remain
  - Suppose N processors are spin-waiting with TTS
  - Bus traffic for all N processors to gain access to lock:
    - **$O(N^2)$**
  - Why?
    - Each time lock is unset, all processors issue an access, but only 1 is successful
- One solution: ***backoff***
  - Instead of spinning constantly, check less frequently
  - Exponential backoff works well in practice
- TTS lock is unfair
  - Threads can starve waiting for the lock

Ticket Locks

- Locks have two counters: `next_ticket`, `now_serving`
  - Deli counter

```
acquire(lock_ptr):
  // take a ticket
  my_ticket = Fetch&Inc(lock_ptr->next_ticket)
  // spin while waiting for ticket
  while(lock_ptr->now_serving != my_ticket);

release(lock_ptr):
  // next ticket
  lock_ptr->now_serving = lock_ptr->now_serving + 1
```

- `release` is 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

- Desirable properties
  - Only one RMW per acquire (probing is done with reads only)
  - FIFO scheme: grant lock to processors in order they requested it
    - Fair, no starvation
- Undesirable properties
  - Still a lot of cache or network contention through polling

Ticket Lock Implementation Issues

- Padding
  - Allocate `now_serving` and `next_ticket` on different cache blocks
  - Two locations reduces interference
- Backoff mechanism
  - Introduce delay on each processor between probes
  - Not exponential backoff
    - Better have a delay = f (number of processors waiting)

```
acquire(lock_ptr):
  my_ticket = Fetch&Inc(lock_ptr->next_ticket)
  while(lock_ptr->now_serving != my_ticket)
    pause(f(my_ticket - lock_ptr->now_serving));
```


Array-Based Queue Locks

- Why not give each waiter its own location to spin on?
  - Avoid coherence storms altogether!
- Idea: `slots` array of size N of `go_ahead` or `must_wait`
  - Padded one slot per cache block
  - Initialize first slot to `go_ahead`, all others to `must_wait`
  - Keep a `next_slot` counter

```
acquire(lock_ptr):
  my_slot = Fetch&Inc(lock_ptr->next_slot) % num_threads
  // spin
  while (lock_ptr->slots[my_slot] == must_wait) ;
  // reset for the next person
  lock_ptr->slots[my_slot] = must_wait

release(lock_ptr):
  // Unblock the next in line
  lock_ptr->slots[my_slot+1 % N] = go_ahead
```

Array-Based Queue Locks

- 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 queue locks*** address the O(N) storage problem
  - Several variants of list-based locks: ***MCS*** 1991, ***CLH*** 1993/1994

MCS List-Based Queue Lock

- Processors waiting on the lock are stored in a linked list
- Every processor using the lock allocates a queue node (I) with two fields
  - Boolean must_wait
  - Pointer to next node in the queue
- Lock variable is a pointer to the tail of the queue

```
acquire(lock):
  I->next = null;
  predecessor = Swap(lock, I)
  if predecessor != NULL                  // 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 CAS(lock, I, null)                 // make sure…
      return                              // CAS succeeded; lock freed
    repeat while I->next = null           // spin to learn successor
  I->next->must_wait = false              // wake successor
```

Barriers

Barriers

- Used to create a ***rendezvous*** point between parallel entities (tasks, threads, …)
  - All threads wait until all threads have reached it
- Often used in loop bodies
- Example: N-body simulation

```
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()
```

Centralized Barrier

- A globally-shared piece of state keeps track of thread arrivals
  - *e.g.*, a counter
- Each thread
  - updates shared state to indicate its arrival
  - polls that state and waits until all threads have arrived
- Then, it can leave the barrier
- Since barrier has to be used repeatedly:
  - state must end as it started

An Incorrect Implementation

```
global (shared) count : integer = P;

procedure central_barrier:
  if Fetch&Dec(&count) == 1
    // last arrival; reset the state
    count = P
  else
    repeat until (count == P)
```

- What is ***wrong*** with the above code?
- Naïve solution: use two back-to-back barriers
  - The first one ensures that all threads have arrived
  - The second one ensures that all threads have left the first one

Better Solution: Sense-Reversing Barrier

- Idea: decouple spinning from the counter

```
global (shared)  count : integer = P
global (shared)  sense : Boolean = true
local  (per-thread) local_sense : Boolean = true

procedure central_barrier:
  // each processor toggles its own sense
  local_sense = not local_sense
  if Fetch&Dec(&count) == 1
    count = P
    // last processor toggles global sense
    sense = local_sense
  else
    repeat until (sense == local_sense)
```

"count" tracks arrivals

"sense" controls spinning

Centralized Barriers

- ✗ Disadvantages
  - – All processors must increment the counter
  - – Each RMW is a serialized coherence action and invalidates others
    - Each one is a cache miss
  - – O(N) if threads arrive simultaneously, slow for lots of processors
  - – "sense" is widely shared → Writing to it can cause broadcast in a limited-pointer directory

- Question: does it make sense to add backoff policy to a sense-reversing centralized barrier?

Software Combining Tree Barrier

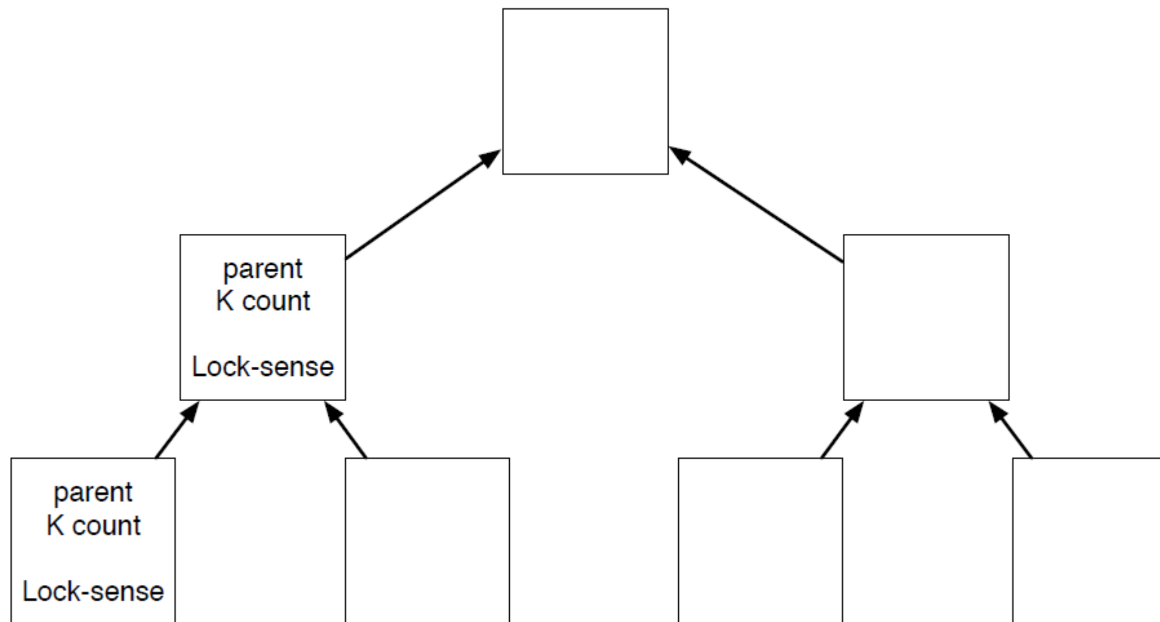
- Shared variable represented as a tree of variables
  - Each node of the tree in a different cache line
- Processors divided into groups
  - Each group assigned to a leaf of the tree

- Each processor updates the state of its leaf
- The ***last*** one to arrive continues up the tree to update parent

- Two logical trees:
  - **Arrival tree**: to determine that all processors have reached the barrier
    - Replaces the “count” variable
  - **Departure tree**: allow the processors to continue past barrier
    - Replaces the “sense” variable
- The two trees can be combined

How It Works

- The thread that reaches the root of the tree begins a reverse wave of updates to lock-sense flags
- As soon as it awakes, each threads retraces its path through the tree unblocking its siblings



Software Combining Tree Barrier

```
struct node {
  k : integer                     // fan-in of this node
  count : integer                 // count of this node (initialized to k)
  lock_sense: boolean             // initially false
  parent: node*                   // pointer to parent node (NULL if root)
}

// each element of nodes allocated in a different cache line
global (shared) nodes[P];
local (per-thread) local_sense : boolean = true
local (per-thread) my_node: node*         // my group's leaf in the tree

procedure combining_barrier:
  combining_barrier_aux(my_node)          // join the barrier
  local_sense = not local_sense           // for next barrier

procedure combining_barrier_aux(np : node*)
  if Fetch&Dec(&np->count) == 1           // last one to reach this node
    if parent != NULL
      combining_barrier_aux(parent)
    np->count = k                         // prepare for next barrier
    np->lock_sense = ! np->lock_sense     // release waiting processors
  repeat until (np->lock_sense == local_sense)
```

And More…

- Software combining is a general technique to reduce contention over ***reduction*** variables
  - Like a shared counter
- There are many other forms of non-centralized barriers
  - Dissemination barrier: reduces latency by eliminating the separation between arrival and departure
  - Tournament barrier: avoids Fetch&Dec() by selecting the winner (who goes up) statically
  - Fuzzy barriers, adaptive barriers, …
- See the "Synchronization" Synthesis Lecture for details.

Advanced Hardware Support

Full/Empty Bits (HEP machine)

- Used in HEP, Cray MTA, Cray XMT, …
- Each word in memory has Full/Empty (F/E) bit
- Bit is tested in hardware before special RD/WR ops
- The RD/WR blocks until the test succeeds:
  - RD until full
  - WR until empty
- If test succeeds, the bit is negated indivisibly with the RD/WR
- Advantages and disadvantages
  - ✓ Very efficient for low level dependences (compare to locks)
  - ✗ F/E bits and logic to initialize them
  - ✗ Support to queue a process if test fails
  - ✗ Logic to implement indivisible ops

Queue-based locks in HW: QOLB

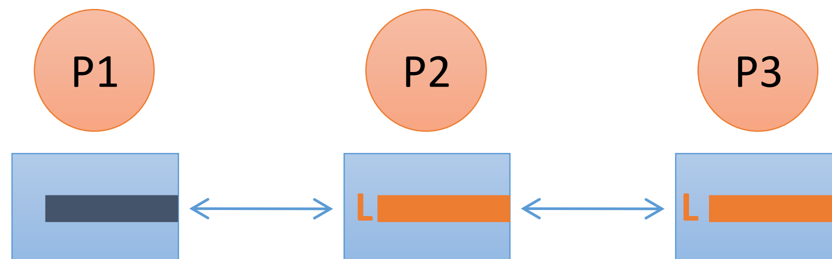
[Goodman et al., ASPLOS 1989]

- ***Queue On Lock Bit*** (originally: Queue On SyncBit (QOSB))
  - Used in Wisconsin Multicube and later adapted for “Scalable Coherent Interface” (SCI)
- ***QOLB instruction*** adds a cache to the queue for the cache line
  - Allocates a shadow copy of the line locally and marks it as “not available”
  - HW maintains a linked list between requesters
- Waiting processor spins locally until line available
- Upon release, lock holder sends line to next cache in the queue

✓ Lock handoff only requires one message on interconnect

✓ After QOLB, processor can do other work before cecking the line

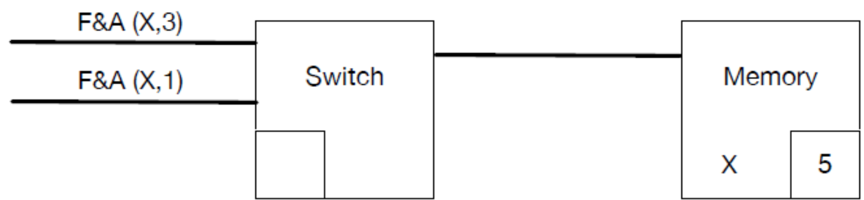
  - A form of prefetching

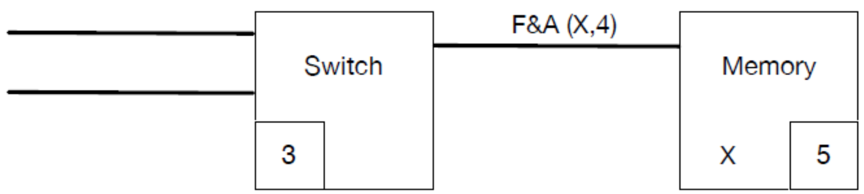


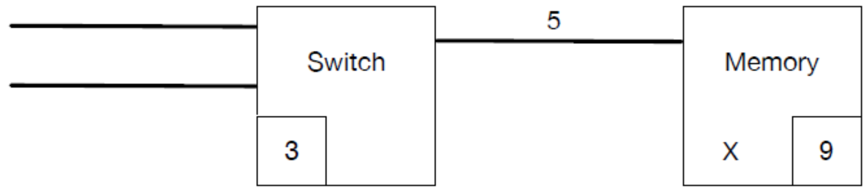
Network Combining in NYU Ultra

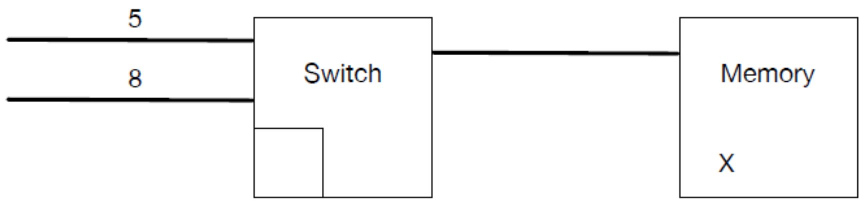
- Atomic Fetch&Add: Send a message to a memory location with a constant
  - *e.g.*, Useful to get the next iteration of a parallel loop
- Network has hardware to combine messages to the same location to tolerate contentions
- Advantages and disadvantages:
  - ✓ Multiple requests in parallel
  - ✓ Less traffic (scalable)
  - ✗ Very complex network
  - ✗ Slows down the rest of the messages

Network Combining in NYU Ultra









Illinois Cedar

- General atomic instruction that operates on synch vars
- Synch var is 2 words: Key and Value
- Synch instruction:

```
{faddr; (condition); op on key; op on value}
```

if * in condition: spin until true

Example:

```
{X; (X.key == 1)*; decrement; fetch}
```

this is like an F/E bit test for a read option

- Implemented using a special processor at every memory module