Lecture 18: Synchronization

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013
Follow up: Intel’s ring interconnect
Introduced in Sandy Bridge microarchitecture

- Four rings
  - request
  - snoop
  - ack
  - data (32 bytes)

- Six interconnect nodes:
  four “slices” of L3 cache + system agent + graphics

- Each bank of L3 connected to ring bus twice

- Theoretical peak BW from cores to L3 at 3.4 GHz is approx. 435 GB/sec
  - each core accessing local slice
Synchronization primitives

- For ensuring mutual exclusion
  - Locks
  - Atomic primitives (e.g., atomicAdd)
  - Transactions (next week)

- For event signaling
  - Barriers
  - Flags

Today’s topic: efficiently implementing synchronization primitives
Three phases of a synchronization event

1. Acquire method
   - How thread attempts to gain access to protected resource

2. Waiting algorithm
   - How thread waits for access to be granted to shared resource

3. Release method
   - How thread enables other threads to gain resource when its work in the synchronized region is complete
What you should know

- Performance issues related to various lock implementations (specifically their interaction with cache coherence)

- Performance issues related to barrier implementations
Busy waiting and blocking

- Busy waiting (a.k.a. "spinning")
  
  ```
  while (condition X not true) {} 
  logic that assumes X is true 
  ```

- In 15-213 or in OS, you have talked about synchronization
  - You were probably taught busy-waiting is bad: why?
"Blocking" synchronization

- If progress cannot be made because a resource cannot be acquired, free up execution resources for another thread (preempt the running thread)
  
  if (condition X not true)
      block until true;    // OS scheduler de-schedules process

- pthreads example

  pthread_mutex_t mutex;
  pthread_mutex_lock(&mutex);
Busy waiting vs. blocking

- Busy-waiting can be preferable to blocking if:
  - Scheduling overhead is larger than expected wait time
  - Processor’s resources not needed for other tasks
    - This often the case in a parallel program since we usually don’t oversubscribe a system when running a performance-critical parallel app (e.g., there aren’t multiple CPU-intensive programs running at the same time)
    - Clarification: be careful to not confuse the above statement with the value of multi-threading (interleaving execution of multiple threads/tasks to hiding long latency of memory operations) with other work within the same app.

Examples

```c
pthread_spinlock_t spin;
int lock;

pthread_spin_lock(&spin);
OSSpinLockLock(&lock);  // OSX spin lock
```
Locks
Warm up: a simple, but incorrect, lock

lock:  
  ld   R0, mem[addr]   // load word into R0
  cmp  R0, #0          // if 0, store 1
  bnz  lock            // else, try again
  st   mem[addr], #1   // store 0 to address

unlock:  st   mem[addr], #0   // store 0 to address

Problem: data race because LOAD-TEST-STORE is not atomic!
Test-and-set based lock

Test-and-set instruction:

\[
\text{ts R0, mem[addr]} \quad \text{// atomically load mem[addr] into R0}
\]
\[
\text{// and set mem[addr] to 1}
\]

---

lock: \[
\text{ts R0, mem[addr]} \quad \text{// load word into R0}
\]
\[
\text{bnz R0, #0} \quad \text{// if 0, lock obtained}
\]

unlock: \[
\text{st mem[addr], #0} \quad \text{// store 0 to address}
\]
Test & set lock: consider coherence traffic

Processor 1

BusRdX

Update line in cache (set to 0)

Invalidate line

[P1 is holding lock...]

BusRdX

Update line in cache (set to 1)

Invalidate line

Processor 2

BusRdX

Update line in cache (set to 1)

Invalidate line

Processor 3

BusRdX

Update line in cache (set to 1)

Invalidate line
Test-and-set lock performance

Benchmark: Total of N lock/unlock sequences (in aggregate) by P processors
Critical section time removed so graph plots only time acquiring/releasing the lock

Bus contention increases amount of time to transfer lock (lock holder must wait to acquire bus to release)

Not shown: bus contention also slows down execution of critical section

Ideal: one bus transaction per lock event

Benchmark executes:
lock(L);
critical-section(c);
unlock(L);
Desirable lock performance characteristics

- **Low latency**
  - If lock is free, and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly

- **Low traffic**
  - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible

- **Scalability**
  - Latency / traffic should scale reasonably with number of processors

- **Low storage cost**

- **Fairness**
  - Avoid starvation or substantial unfairness
  - One ideal: processors should acquire lock in the order they request access to it

Simple: test and set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness
Test-and-test-and-set lock

```c
void Lock(volatile int* lock) {
    while (1) {
        while (*lock != 0);  // while another processor has the lock...
        if (test&set(*lock) == 0) // when lock is released, try to acquire it
            return;
    }
}

void Unlock(volatile int* lock) {
    *lock = 0;
}
```
Test & test & set lock: coherence traffic

Processor 1

BusRdX

Update line in cache (set to 1)

[P1 is holding lock...]

Update line in cache (set to 0)

Invalidate line

Processor 2

BusRd

[Many reads from local cache]

Invalidate line

BusRdX

Update line in cache (set to 1)

Invalidate line

Processor 3

BusRd

[Many reads from local cache]

Invalidate line

BusRdX

Update line in cache (set to 1)
Test & test & set characteristics

- Higher latency than test & set in uncontended case
  - Must test... then test and set
- Generates much less bus traffic
  - One invalidation per waiting processor per lock release
    - $O(P)$ invalidations = $O(P^2 \text{ traffic})$
    - Recall: test & set generated one invalidation per waiting processor per test
- More scalable (due to less traffic)
- Storage cost unchanged
- Still no provisions for fairness

$P =$ number of waiting processors
Test-and-set lock with back-off

Upon failure to acquire lock, delay for awhile before retrying

```c
void Lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (test&set(*l) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

- Same **uncontended** latency as test and set, but potentially higher latency under contention. Why?
- Generates less traffic than test and set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged
- Exponential back-off can cause severe unfairness
  - Newer requesters back off for shorter intervals
Ticket lock

Main problem with test & set style locks: upon release, all waiting processors attempt to acquire lock using test & set

```
struct lock {
    volatile int next_ticket;
    volatile int now_serving;
};

void Lock(lock* l) {
    int my_ticket = atomicIncrement(l->next_ticket);
    while (my_ticket != l->now_serving);
}

void unlock(lock* l) {
    l->now_serving++;
}
```

No atomic operation needed to acquire the lock (only a read)
Result: only one invalidation per lock release (traffic is $O(P)$)
Array-based lock

Each processor spins on a different memory address

Use fetch+op (e.g., atomicIncrement) to assign address on attempt to acquire

```c
struct lock {
    volatile padded_int status[P];  // padded to keep off same cache line
    volatile int head;
};

int my_element;

void Lock(lock* l) {
    my_element = atomicIncrement(l->head);  // assume circular increment
    while (l->status[my_element] == 1);
}

void unlock(lock* l) {
    l->status[next(my_element)] = 0;
}
```

O(1) traffic per release, but requires space linear in P
Implementing atomic fetch and op

// atomicCAS: atomic compare and swap
int atomicCAS(int* addr, int compare, int val)
{
    int old = *addr;
    *addr = (old == compare) ? val : old;
    return old;
}

- Exercise: how can you build an atomic fetch+op out of atomicCAS()?
  - try: atomicIncrement()

- See definition of atomicCAS() in NVIDIA programmer’s guide
Barriers
Implementing a centralized barrier
Based on shared counter

```c
struct Bar_t {
    int counter; // initialize to 0
    int flag;
    LOCK lock;
};

// barrier for p processors
void Barrier(Bar_t* b, int p) {
    lock(b->lock);
    if (b->counter == 0) {
        b->flag = 0; // first arriver clears flag
    }
    int arrived = ++(b->counter);
    unlock(b->lock);

    if (arrived == p) { // last arriver sets flag
        b->counter = 0;
        b->flag = 1;
    } else {
        while (b->flag == 0); // wait for flag
    }
}
```

**Does it work? Consider:**

```c
do stuff ...
Barrier(b, P);
do more stuff ...
Barrier(b, P);
```
Correct centralized barrier

```c
struct Bar_t {
    int arrive_counter;   // initialize to 0
    int leave_counter;    // initialize to P
    int flag;
    LOCK lock;
};

// barrier for p processors
void Barrier(Bar_t* b, int p) {
    lock(b->lock);
    if (b->arrive_counter == 0) {
        if (b->leave_counter == P) {    // no other threads “in barrier”
            b->flag = 0;                 // first arriver clears flag
        } else {
            unlock(lock);
            while (b->leave_counter != P);  // wait for all to leave before clearing
            lock(lock);
            b->flag = 0;                   // first arriver clears flag
        }
    }
    int arrived = ++(b->counter);
    unlock(b->lock);

    if (arrived == p) {    // last arriver sets flag
        b->arrive_counter = 0;
        b->leave_counter = 1;
        b->flag = 1;
    } else {
        while (b->flag == 0);     // wait for flag
        lock(b->lock);
        b->leave_counter++;
        unlock(b->lock);
    }
}
```

Main idea: wait for all processes to leave first barrier, before clearing flag for entry into the second
Centralized barrier with sense reversal

```c
struct Bar_t {
    int counter;  // initialize to 0
    int flag;   // initialize to 0
    LOCK lock;
};

int local_sense = 0; // private per processor

// barrier for p processors
void Barrier(Bar_t* b, int p) {
    local_sense = (local_sense == 0) ? 1 : 0;
    lock(b->lock);
    int arrived = ++(b->counter);
    if (b->counter == p) { // last arriver sets flag
        unlock(b->lock);
        b->counter = 0;
        b->flag = local_sense;
    } else {
        unlock(b->lock);
        while (b->flag != local_sense); // wait for flag
    }
}
```

Sense reversal optimization results in one spin instead of two
Centralized barrier: traffic

- $O(P)$ traffic on a bus:
  - 2P write transactions to obtain barrier lock and update counter
  - 2 write transactions to write flag + reset counter
  - P-1 transactions to read updated flag

- But there is still serialization on a single shared variable
  - Latency is $O(P)$
  - Can we do better?
Combining trees make better use of parallelism in interconnect topologies
- \( \log(P) \) latency
- Strategy makes less sense on a bus (all traffic still serialized on single shared bus)

- **Acquire:** when processor arrives at barrier, performs `atomicIncr()` of parent counter
  - Process recurses to root
- **Release:** beginning from root, notify children of release
Next week

- What if you have a shared variable for which contention is low enough that it is unlikely two processors will enter the critical section at the same time?

- You could avoid the overhead of taking the lock since it is likely ensuring mutual exclusion is not needed for correctness - Classic optimize for the common case situation.

- What happens if you take this approach and you’re wrong: in the middle of the critical region, another process enters the same region?
Preview: transactional memory

atomic
{
  // begin transaction

  perform atomic computation here ...

} // end transaction

Instead of ensuring mutual exclusion via locks, system will proceed as if no synchronization was necessary (it speculates!).

System provides hardware/software support for “rolling back” all loads and stores from critical region if it detects (at runtime) that another thread has entered same region.