Lecture 19: Synchronization
Announcements

- Assignment 4 due tonight at 11:59 PM
Synchronization primitives
(that we have or will soon see in class)

- For ensuring mutual exclusion
  - Locks
  - Basic atomic operations (e.g., atomicAdd)
  - Transactions (next time)

- For event signaling
  - Barriers
  - Flags

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

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

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

3. Release method
   - How process enables other processes 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 various barrier implementations
Busy waiting and blocking

- **Busy waiting (a.k.a. “spinning”)**
  
  ```
  while (condition X not true) {} 
  proceed with logic that assumes X is true
  ```

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

- **“Blocking”**
  - If progress cannot be made, free up resources for someone else (pre-emption)
    ```
    if (condition X not true)
    block;  // OS scheduler kicks in, de-schedules process from processor
    ```
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 programs running at the same time)
    - Clarification: be careful to not confuse this idea with the clear value of multi-threading (interleaving execution of multiple threads/tasks to hiding long latency of memory operations) with other work within the same app.
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
```

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:

ts R0, mem[addr]  // atomically load mem[addr] into R0
                  // and set mem[addr] to 1

lock:     ts  R0, mem[addr]  // load word into R0
           bnz R0, #0   // if 0, lock obtained

unlock:   st  mem[addr], #0 // 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

BusRdX
Update line in cache (set to 1)
Invalidate line

Processor 3

BusRdX
Update line in cache (set to 1)
Invalidate line

BusRdX
Update line in cache (set to 1)
Invalidate line

BusRdX
Update line in cache (set to 1)
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

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

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
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 it quickly

- **Low traffic**
  - If all processors are trying to acquire lock at once, they should acquire the lock in suggestion 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 holding lock...]

Update line in cache (set to 0)

Invalidate line

Processor 2

BusRd

[Many reads from local cache]

Invalidate line

Processor 3

BusRd

[Many reads from local cache]

Invalidate line

BusRdX

Update line in cache (set to 1)

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
- More scalable (due to less traffic)
- Storage cost unchanged
- Still no provisions for fairness
Test-and-set lock with backoff

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
- Generates less traffic than test and set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged
- Exponential backoff 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

```c
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++;
}
```
Array-based lock

Each processor spins on a different memory address

Use fetch&op (below: atomicIncrement) to assign address on attempt to acquire

```c
struct lock {
    volatile int status[P];
    volatile int head;
};

int my_element;

void Lock(lock* l) {
    my_element = atomicIncrement(l->head); // assume circular inc
    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 programmers guide
Barriers
Implementing a centralized barrier
Based on shared counter

struct Bar {
    int counter;     // initialize to 0
    int flag;
    LOCK lock;
};

// barrier for p processors
void Barrier(Bar* 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:

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

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

// barrier for p processors
void Barrier(Bar* b, int p) {
  lock(b->lock);
  if (b->arrive_counter == 0) {
    while (b->leave_counter != P); // wait for all to leave before clearing
    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 = 0;
    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 the second
Correct centralized barrier: sense reversal

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

int local_sense = 0;  // private per processor

// barrier for p processors
void Barrier(Bar* b, int p) {
    local_sense != local_sense;
    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
    }
}
```

One spin instead of two
Centralized barrier: traffic

- 0(p) traffic on a bus:
  - 2p transactions to obtain barrier lock and update counter
  - 2 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

- 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 time

- 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 very likely ensuring mutual exclusion is not needed for correctness.

- What happens if you take this approach and you’re wrong: in the middle of the critical region, another process enters the same region?
Next time: 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 (speculation).

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.