Lecture 10
Barriers and Thread Pools

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Outline

• Homework 2 Review
• Reasoning about mutex/semaphore-based schemes
  • Read-Write Locks
  • Barriers
• Lab 2
  • Thread Pools
  • Genetic Algorithms
Homework 2 Q1

Warm-up: As we discussed in class, explain at least two important differences between semaphores and condition variables.
Warm-up: As we discussed in class, explain at least two important differences between semaphores and condition variables.

1. Blocking
   **Semaphores:** Block only if decrementing past 0
   **Condition Variables:** wait() always blocks

2. Memory
   **Semaphores:** Current state depends on initial count, sequence of up() and down() operations
   **Condition Variables:** Past signal() / broadcast() calls don’t affect a current wait()

3. Awakening
   **Semaphores:** Can only release one waiter at a time.
   **Condition Variables:** Can release all waiters (via broadcast())
What criterion or criteria must a correct mutex fulfill? Without looking it up, reason out which of these criteria (and/or possibly others) must be true for other thread synchronization primitive (namely condition variables and semaphores), and explain why.
What criterion or criteria must a correct mutex fulfill? Without looking it up, reason out which of these criteria (and/or possibly others) must be true for other thread synchronization primitive (namely condition variables and semaphores), and explain why.

Mutex:
1. Mutual exclusion
2. Starvation-free
3. Deadlock-free
Homework 2 Q2

What criterion or criteria must a correct mutex fulfill? Without looking it up, reason out which of these criteria (and/or possibly others) must be true for other thread synchronization primitive (namely condition variables and semaphores), and explain why.

Condition Variables:
1. Linearizability (no lost wait()s)
2. (Proper wait()ing: no spurious wakeups)
3. Starvation-free: FIFO wait()ing
4. Deadlock-free: wait() must not prevent signal()
Homework 2 Q2

What criterion or criteria must a correct mutex fulfill? Without looking it up, reason out which of these criteria (and/or possibly others) must be true for other thread synchronization primitive (namely condition variables and semaphores), and explain why.

Semaphores:
1. Linearizability
2. Non-negative: semaphore value must be zero or positive
3. Starvation-free: FIFO down()ing
4. Deadlock-free: down() must not prevent up()
Consider the following code:

```c
1     static double sum_stat_a = 0;
2     static double sum_stat_b = 0;
3     static double sum_stat_c = 1000;
4     int aggregateStats(double stat_a, double stat_b, double stat_c) {
5         sum_stat_a += stat_a;
6         sum_stat_b -= stat_b;
7         sum_stat_c -= stat_c;
8         return sum_stat_a + sum_stat_b + sum_stat_c;
9     }
10    void init(void) { }
```

Use a single pthread mutex or std::mutex to make this function thread-safe. Add global variables, and content to the init() function, iff necessary.
Homework 2 Q3

```c++
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std::mutex stats_lock;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
    double rval = 0;
    stats_lock.lock();
    rval = sum_stat_a += stat_a;
    rval += sum_stat_b -= stat_b;
    rval += sum_stat_c -= stat_c;
    stats_lock.unlock();
    return rval;
}

void init(void) {} 
```

Invariant: `rval` is a linearized snapshot of the three combined `sum_stat` at one instant.

Alternative: scoped_lock or unique_lock.
Let’s make this more parallelizable. We always want to reduce critical sections as much as possible to minimize the time threads need to wait for a resource protected by a lock. Modify the original code from question 3 to make it thread-safe, but use three mutexes this time, one each for `sum_stat_a`, `sum_stat_b`, and `sum_stat_c`. Hint: explain what guarantee(s) you provide about the return value of `aggregateStats()`.
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std::mutex stats_lock_a, stats_lock_b, stats_lock_c;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
    double rval = 0;
    stats_lock_a.lock();
    rval = sum_stat_a += stat_a;
    stats_lock_a.unlock();
    stats_lock_b.lock();
    rval += sum_stat_b -= stat_b;
    stats_lock_b.unlock();
    stats_lock_c.lock();
    rval += sum_stat_c -= stat_c;
    stats_lock_c.unlock();
    return rval;
}

void init(void) {
}

Invariant: very weak. Total sum is from some point after the beginning of the aggregateStats() function call, but no upper bound (arbitrarily large wait before locking stats_lock_b and stats_lock_c.
```cpp
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std::mutex stats_lock_a, stats_lock_b, stats_lock_c;

int aggregateStats(double stat_a, double stat_b, double stat_c) {
    double rval = 0;
    stats_lock_a.lock();
    rval = sum_stat_a += stat_a;
    stats_lock_b.lock();
    stats_lock_a.unlock();
    rval += sum_stat_b -= stat_b;
    stats_lock_c.lock();
    stats_lock_b.unlock();
    rval += sum_stat_c -= stat_c;
    stats_lock_c.unlock();
    return rval;
}

void init(void) {}
```

Invariant: Stronger, matches Q3.
In your own words, describe one of the possible causes of the Lost Wakeup Problem, including a scenario that triggers this cause, and how to fix it. Use C++ or pseudocode in your explanation iff you find it necessary. If you do research and find other causes beyond what we discussed in class, please note your source(s).
Lost Wakeup Cause #1

• “Consumer”-type thread does not hold mutex while checking condition or “Producer”-type thread does not hold mutex while updating condition, and

• “Consumer”-type thread is about to wait(), and

• “Producer”-type thread calls signal() or broadcast()
Homework 2 Q5

```c
void element_dequeue() {
    pthread_lock(queue_lock)
    while (empty) {
        pthread_cond_wait(cond, queue_lock)
    }
    // Inv: queue is not empty
    element = head.data
    head = head.next
    // Inv: either head=tail=NULL or !empty
    if (head is NULL) {
        tail = NULL
        empty = true
    }
    pthread_unlock(queue_lock)
    return element
}

void enqueue(element) {
    pthread_lock(queue_lock)
    n = new node(element)
    if (tail != NULL) {
        tail.next = n
        tail = n
    }
    if (head == NULL) {
        head = tail
    }
    // Inv: either !empty or head=tail=NULL
    pthread_unlock(queue_lock)
    empty = false
    pthread_cond_signal(cond)
}
```
Lost Wakeup Cause #2

• “Producer” only signals for empty→non-empty, and only signals one thread on empty→non-empty, and

• >1 “Consumer”-type threads are waiting

Demonstrated by:

• “Producer”-type thread calls signal()
element dequeue():
    pthread_lock(queue_lock)
    while (empty):
        pthread_cond_wait(cond, queue_lock)

    // Inv: queue is not empty
    element = head.data
    head = head.next

    // Inv: either head=tail=NULL or !empty
    if head is NULL:
        tail = NULL
        empty = true

    pthread_unlock(queue_lock)
    return element

enqueue(element)
    pthread_lock(queue_lock)
    n = new node(element)
    if tail != NULL:
        tail.next = n
        tail = n
    if head == NULL:
        head = tail

    // Inv: either !empty or head=tail=NULL
    empty = false
    if head == tail:  // first element
        pthread_cond_signal(cond)
    pthread_unlock(queue_lock)
Using any pseudocode style you prefer, outline the functionality of a thread-safe:

- **push(item)**: Add item to the end of the queue
- **pop()**: Remove item from the beginning of the queue and return it
- **listen()**: wait for and then return an item when available

Built using mutexes, semaphores, and/or condition variables.
mutex mut;
cond_var cond;
queue<Item> q;

SafeQueue::push(Item item):
    mut.lock()
    q.push(item);
    cond.broadcast();
    mut.unlock();

Item SafeQueue::pop():
    Item rval = None;
    mut.lock();
    if q.size():
        rval = q.front();
        rval.pop();
    mut.unlock();
    return rval;

Item SafeQueue::listen():
    mut.lock();
    while !q.size():
        cond.wait(mut)
    Item rval = q.front();
    q.pop();
    mut.unlock();
    return rval;
Using any pseudocode style you prefer, outline the functionality of a thread-safe:

• push(item): Add item to the end of the queue
• pop(): Remove item from the beginning of the queue and return it
• listen(): wait for and then return an item when available

Built using **only** semaphores.
sem mut = 1;
sem cond;
queue<Thread> waiters;
queue<Item> q;

SafeQueue::push(Item item):
    mut.down()
    q.push(item);
    cond.sem.down();
    if waiters.size():
        waiters[0].sem.up();
        waiters.pop();
    cond.sem.up();
    mut.up();

Item SafeQueue::pop():
    Item rval = None;
    mut.down()
    if q.size():
        rval = q.front();
        rval.pop();
        items.down();
        mut.up();
    return rval;

Item SafeQueue::listen():
    mut.down();
    while !q.size():
        Thread self; // has a sem
        cond.sem.down();
        waiters.push(self);
        cond.sem.up();
        mut.up();
        self.sem.down();
        mut.down();
    Item rval = q.front();
    q.pop();
    mut.unlock();
    return rval;
Barriers
while (true) {
    frame.prepare();
    frame.display();
}
Simple Video Game

while (true) {
    frame.prepare();
    frame.display();
}

• Optimization: What about overlapping work?
  • 1st thread displays frame
  • 2nd prepares next frame
Two-Phase Rendering

```c
int phase = 0;
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

```c
int phase = 0;
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Two-Phase Rendering

```java
int phase = 0;
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

```java
int phase = 0;
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Two-Phase Rendering

```java
int phase = 0;
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

```java
int phase = 0;
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Synchronization Problems

• How do threads stay in sync?
• Too early?
  • “we render no frame before its time”
• Too late?
  • Reuse memory containing prepared frame before frame is displayed
Barrier

• Synchronize group of threads at single point
  • Each thread waits until all threads arrive
  • Each thread continues

• Solution
  • Mutex or semaphore to count arrivals and hold threads
semaphore sem = 0;
int counter = 0, int n = num_threads;

void await(void) {
    counter += 1;
    if (counter < n) {
        sem.down();
    } else {
        for(size_t i = 0; i < n - 1; i++) {
            sem.up();
        }
    }
}
semaphore sem = 0;
int counter = 0, int n = num_threads;

void await(void) {
    counter += 1;
    if (counter < n) {
        sem.down();
    } else {
        for(size_t i = 0; i < n - 1; i++) {
            sem.up();
        }
        counter = 0;  // Make it reentrant
    }
}

Alternate:

if (counter % num_threads) {
    sem.down();
} else {

semaphore sem = 0;
int counter = 0, int n = num_threads;

void await(void) {
    counter += 1;
    if (counter % n) {
        sem.down();
    } else {
        for(size_t i = 0; i < n - 1; i++) {
            sem.up();
        }
    }
}

1. Critical section
2. Phase 2 arrivals start before phase 1 departures leave
Barrier Strawman #1
Problem

```c
semaphore sem = 0, mut = 1;
int counter = 0, int n = num_threads;

void await(void) {
    mut.down();
    counter += 1;
    if (counter % n) {
        mut.up();
        sem.down();
    } else {
        for(size_t i = 0; i < n - 1; i++) {
            sem.up();
        }
        mut.up();
    }
}
```

1. Critical section
2. Phase 2 arrivals start before phase 1 departures leave
Barrier

• Synchronize group of threads at single point
  • Each thread waits until all threads arrive
  • Each thread continues

• **Better** Solution
  • Mutex or semaphore to count arrivals
  • Mutex or semaphore to hold threads until count is equal to number of threads
Simple Semaphore-Based Barrier

```c
semaphore arrival = 1, departure = 0;
int counter = 0, int n = num_threads;

void await(void) {
    arrival.down();  // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
}
departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```

Must be known a priori
Simple Semaphore-Based Barrier

First arrival

```c
semaphore arrival = 1, departure = 0;
int counter = 1, int n = num_threads;

void await(void) {
    arrival.down();       // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();     // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();       // Back to initial conditions
    }
}
```
Simple Semaphore-Based Barrier

n - 1 arrivals

```c
semaphore arrival = 1, departure = 0;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down(); // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down(); // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up(); // Back to initial conditions
    }
}
```
Simple Semaphore-Based Barrier

n arrivals

```c
semaphore arrival = 0, departure = 1;
int counter = n, int n = num_threads;

void await(void) {
    arrival.down(); // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        departure.up();
    } else {
        // Acts as mutex & block on departure
        departure.down();
        counter -= 1;
        if (counter > 0) {
            departure.up();
        } else {
            arrival.up(); // Back to initial conditions
        }
    }
}
```
Simple Semaphore-Based Barrier

n arrivals, 1 departure

```c
semaphore arrival = 0, departure = 1;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```
Simple Semaphore-Based Barrier

n arrivals, n - 1 departures

```c
semaphore arrival = 0, departure = 1;
int counter = 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();  // Back to initial conditions
    }
}
```
Simple Semaphore-Based Barrier

n arrivals

```
semaphore arrival = 1, departure = 0;
int counter = n - 1, int n = num_threads;

void await(void) {
    arrival.down();    // Acts as mutex & block on arrival
    counter += 1;
    if (counter < n) {
        arrival.up();
    } else {
        departure.up();
    }
    departure.down();  // Acts as mutex & block on departure
    counter -= 1;
    if (counter > 0) {
        departure.up();
    } else {
        arrival.up();    // Back to initial conditions
    }
}
```
Pthread Barrier

• Surprise! Pthread has a barrier primitive
• Type: pthread_barrier_t
• Initialization:
  int pthread_barrier_init(pthread_barrier_t* barrier, attributes, unsigned int count);
• Wait:
  int pthread_barrier_wait(pthread_barrier_t* barrier);
Outline

• Coordinating Resources
  • Reader-Writer Locks
  • Barriers

• Lab 2 Techniques
  • Thread Pools
  • Genetic Algorithms
Thread Pool

- Thread work can be small pieces
  - Creating and destroying threads is expensive
  - Reduce overhead: reuse threads

**Option A: Centralized Scheduling**

1. Create group of N threads
2. Use thread-safe queue to identify “idle” threads
3. Main thread atomically removes and invoke an idle thread when new work arrives
4. Threads atomically add themselves back to queue when work is done
Thread Pool

- Thread work can be small pieces
  - Creating and destroying threads is expensive
  - Reduce overhead: reuse threads

**Option B: Decentralized Scheduling**

1. Create group of N threads
2. Use thread-safe queue to store work ("job queue")
3. Each thread uses listen() to remove work from job queue when available.
4. Add results from completed work to second queue for main (driver) thread to consume.
Genetic Algorithms

Overview
1. Initialize Population
2. Compute fitness -> terminate if good enough
3. Select best candidates from population
4. Crossover (mate) candidates
5. Mutate candidates
6. GOTO 2