Alexander Nguyen

Atn290

Christopher Mitchell

Multicore Programming

Homework 2

1. Differences:
   a. Semaphores have memory whereas conditional variables do not. Conditional variables rely on a signal whereas a semaphore must keep track of a binary or counting semaphore.
   b. Conditional variables are able to broadcast to all waiting threads as well as signal to individual threads whereas a semaphore is unable to do so. Semaphores are not susceptible to the lost wake up issue that conditional variables are however.

2. Mutexes must provide mutual exclusion through lock and unlock that enable the user to define what the critical section is within code. As a result, mutexes must prevent consumers from accessing resources simultaneously. In a sense, a mutex is considered a locking mechanism.

   Semaphore: A synchronization primitive that enables waiting without busy wait. This can be done with either a counting semaphore, where a set number of consumers may access a resource at any given time, and a binary semaphore, where only one consumer may access a resource at any time. Semaphores are both starvation free and deadlock free. Because of the changes in counting and binary semaphores, a semaphore can be considered a signaling mechanism. As such, semaphores require mutual exclusion with locking and unlocking properties.

   Condition Variables: Enables a thread to wait until a signal is given to change. Has operations such as wait, signal, and broadcast. Conditional variables also require locking and unlocking mechanisms to ensure thread synchronization by unlocking when the thread sleeps and locking again once a signal is sent to wake up.

3. Consider the following code:

```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
size_t NUM_THREADS = 10;
std:: mutex mu;
    int aggregateStats(double stat_a, double stat_b, double stat_c) {
            std::unique_lock<mutex> locker(mu);
```

```
                sum_stat_a += stat_a;
                sum_stat_b -= stat_b;
                sum_stat_c -= stat_c;
                int total = sum_stat_a + sum_stat_b + sum_stat_c
                locker.unlock();
                return total;
       }
void init(void) {
    std::unique_ptr threads[NUM_THREADS];
    for(size_t t = 0; t < NUM_THREADS; t++) {
        threads[t].reset(new std::thread(aggregateStats, stat_a, stat_b, stat_c));
    }
    //for(size_t t = 0; t < NUM_THREADS; t++) {
        // threads[t]->join(); //} return 0;
     }
```

This code is able to prevent other threads from accessing aggregateStats and changing the variables through that method. As a result, the user can be assured that returning total as a local variable will produce the results a user can expect as long as the static variables are not changed elsewhere.

4. Consider the following code:

```
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 1000;
std:: mutex mu;
    int aggregateStats(double stat_a, double stat_b, double stat_c) {
            std::unique_lock<mutex> locker(mu);
            sum_stat_a += stat_a;
            int a =  sum_stat_a
            locker.unlock();

            std::unique_lock<mutex> locker(mu);
            sum_stat_b -= stat_b;
            int b = sum_stat_b;
            locker.unlock();

            std::unique_lock<mutex> locker(mu);
            sum_stat_c -= stat_c;
            int c = sum_stat_c;
            locker.unlock();

            int total = a + b + c;
            return total;
 }
void init(void) { }
```

The issue with doing this is that the sum_stat's can be changed in between stats by another function or elsewhere in the code such that the return result may not be what the user desired when summing all three stats. Although variation can be limited by using more local variables, another thread can change the value of sum_stat_c through the function while the current thread is processing sum_stat_a. We can only guarantee that the values stored locally will not be changed once it has been processed and returned as a sum.

5. In our Lab1, consider if instead of signaling after every insert that we only signal if the queue was previously empty. The issue becomes that the signal can be lost if the queue changes from empty to not empty after the if condition in the pop function but before the thread goes to sleep at cond.wait. Because of this, the thread has to wait until another signal is sent by insert to wake up.

```
Insert(x){
    If(queue.isEmpty()) signal();
    Queue.insert(x)
    Signal()
}

pop(){
    while(queue.isEmpty())
        Cond.wait(lock)
    Return queue.pop()
}
```

   This can be an issue if pop missed the signal from insert and is stuck waiting. This can be fixed using a semaphore such that pop waits until a semaphore suggests that the critical section is accessible or signaling after every insert instead of it was only empty.

6.

```
Push(item) {
    Lock(mu)
    q.push()
    unlock(mu)
    signal()
}
Pop(){
    Lock(mu)
    If(queue is not empty) q.pop()
    unlock(mu)
}
Listen(){
    Lock(mu)
    while(empty) cond.wait()
    q.pop()
    unlock(mu)
}

s, m, sem_lock

lock(){
    m.down()
    s.down()
    locked = true
    holder = self
    s.up()
}

unlock(){
    if(!s.isLocked || holder != self()) s.up();
    return;
    locked = false;
    s.up();
    m.up();
}

Push(item) {
    Lock()
    q.push()
    unlock()
    sem_post(sem_lock)
}
Pop(){
    Lock()
    q.pop()
    unlock()
}
Listen(){
    Lock()
    if(empty)sem_wait(sem_lock)
    s.pop()
    unlock()
}
```