Announcements/Reminders

- HW 3 will be posted today.
- Lab 3 proposal due Friday.
Today: Monitors and Condition Variables

- What is wrong with semaphores?

- Monitors
  - What are they?
  - How do we implement monitors?
  - Two types of monitors: Mesa and Hoare

- Compare semaphore and monitors
What’s wrong with Semaphores?

- Semaphores are a huge step up from the equivalent load/store implementation, but have the following drawbacks.
  - They are essentially shared global variables.
  - There is no semantic connection between the semaphore and the data to which the semaphore controls access.
  - Access to semaphores can come from anywhere in a program.
  - They serve two purposes, mutual exclusion and scheduling constraints.
  - There is no control or guarantee of proper usage.

- **Solution:** use a higher level primitive called *monitors*
What is a Monitor?

- A monitor is similar to a class that ties the data, operations, and in particular, the synchronization operations all together,

- Unlike classes,
  - monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.
  - monitors require all data to be private.
Monitors: A Formal Definition

- A Monitor defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data.
  - The monitor uses the *lock* to insure that only a single thread is active in the monitor at any instance.
  - The *lock* also provides mutual exclusion for shared data.
  - *Condition variables* enable threads to go to sleep inside of critical sections – they release the lock as the thread goes to sleep.
Monitors: A Formal Definition (cont)

- Monitor operations:
  - Encapsulates the shared data you want to protect.
  - Acquires the mutex at the start.
  - Operates on the shared data.
  - Temporarily releases the mutex if it can’t complete.
  - Reacquires the mutex when it can continue.
  - Releases the mutex at the end.
Implementing Monitors in Java

- It is simple to turn a Java class into a monitor:
  - Make all the data private
  - Make all methods synchronized (or at least the non-private ones)

```java
class Queue{
    private ...; // queue data

    public void synchronized Add( Object item ) {
        put item on queue;
    }

    public Object synchronized Remove() {
        if queue not empty {
            remove item;
            return item;
        }
    }
}
Condition Variables

- How can we change `remove()` to wait until something is on the queue?
  - Logically, we want to go to sleep inside of the critical section
  - But if we hold on to the lock and sleep, then other threads cannot access the shared queue, add an item to it, and wake up the sleeping thread
  ⇒ The thread could sleep forever

- **Solution:** use condition variables
  - Condition variables enable a thread to sleep inside a critical section
  - Any lock held by the thread is atomically released when the thread is put to sleep
Operations on Condition Variables

- **Condition variable**: is a queue of threads waiting for something inside a critical section.

- Condition variables support three operations:
  1. \textit{Wait}(\textit{Lock} \ lock): atomic (release lock, go to sleep), when the process wakes up it re-acquires lock.
  2. \textit{Signal}(): wake up waiting thread, if one exists. Otherwise, it does nothing.
  3. \textit{Broadcast}(): wake up all waiting threads

- **Rule**: thread must hold the lock when doing condition variable operations.
Condition Variables in Java

- Use `wait()` to give up the lock
- Use `notify()` to signal that the condition a thread is waiting on is satisfied.
- Use `notifyAll()` to wake up all waiting threads.
- Effectively one condition variable per object.

```java
class Queue {
    private ...; // queue data
    public void synchronized Add( Object item ) {
        put item on queue;
        notify ();
    }
    public Object synchronized Remove() {
        while queue is empty
            wait (); // give up lock and go to sleep
        remove and return item;
    }
}
```
Mesa versus Hoare Monitors

What should happen when signal() is called?

- No waiting threads $\Rightarrow$ the signaler continues and the signal is effectively lost (unlike what happens with semaphores).
- If there is a waiting thread, one of the threads starts executing, others must wait.
Mesa versus Hoare Monitors (cont)

**Mesa-style:** (Nachos, Java, and most real operating systems)

- The thread that signals keeps the lock (and thus the processor).
- The waiting thread waits for the lock.

**Hoare-style:** (most textbooks)

- The thread that signals gives up the lock and the waiting thread gets the lock.
- When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread.
Mesa versus Hoare Monitors (cont)

The synchronized queuing example above works for either style of monitor, but we can simplify it for Hoare-style semantics:

- **Mesa-style:** the waiting thread may need to wait again after it is awakened, because some other thread could grab the lock and remove the item before it gets to run.

- **Hoare-style:** we can change the ‘while’ in Remove to an ‘if’ because the waiting thread runs immediately after an item is added to the queue.
Mesa versus Hoare Monitors (cont)

class Queue {
    private ...; // queue data
    public void synchronized add( Object item ){
        put item on queue; notify();
    }
    public Object synchronized remove() {
        if queue is empty // while becomes if
            wait();
        remove and return item;
    }
}
class ReaderWriter {
    private int numReaders = 0;
    private int numWriters = 0;

    private synchronized void prepareToRead () {
        while ( numWriters > 0 ) wait ();
        numReaders++;
    }
    private synchronized void doneReading () {
        numReaders--;
        if ( numReaders == 0 ) notifyAll ();
    }
    public ... someReadMethod () {
        // reads NOT synchronized: multiple readers
        prepareToRead ();
        <do the reading>
        doneReading ();
    }
}
private void prepareToWrite () {
    numWriters++;
    while ( numReaders != 0 ) wait ();
}

private void doneWriting () {
    numWriters--;
    notifyAll ();
}

public synchronized void someWriteMethod (...) {
    // synchronized => only one writer
    prepareToWrite ();
    <do the writing>
    doneWriting ();
}

Readers/Writers using Monitors (Java)
Bounded Buffer using Hoare-style condition variables

```cpp
class BBMonitor {
public:
    void Append(item);
    void Remove(item);
private:
    item buffer[N];
    int last, count;
    Condition full, empty;
};

BBMonitor {
    count = 0;
    last = 0;
}

Append(item){
    lock.Acquire();
    if (count == N)
        empty.Wait(lock); // Wait, temp release lock
    buffer[last] = item;
    last = (last + 1) mod N;
    count += 1;
    full.Signal();
    lock.Release();
}

Remove(item){
    lock.Acquire();
    if (count == 0)
        full.Wait(lock);
    item = buffer[(last-count) mod N];
    count = count-1;
    empty.Signal();
    lock.Release();
}
```
Can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing. Does the following work?

```java
  condition.Wait() { semaphore.wait(); }
  condition.Signal() { semaphore.signal(); }
```

But condition variables only work inside a lock. If we use semaphores inside a lock, we have may get deadlock. Why?
Semaphores versus Monitors

How about this?

```java
condition.Wait(Lock lock) {
    lock.Release();
    semaphore.wait();
    lock.Acquire();
}
condition.Signal() {
    semaphore.signal();
}
```
Semaphores versus Condition Variables

- Condition variables do not have any history, but semaphores do.
  - On a condition variable signal, if no one is waiting, the signal is a no-op.
    \[ \Rightarrow \text{If a thread then does a condition.Wait, it waits.} \]
  - On a semaphore signal, if no one is waiting, the value of the semaphore is incremented.
    \[ \Rightarrow \text{If a thread then does a semaphore.Wait, the value is decremented and the thread continues.} \]

- semaphore.Wait and Signal are commutative, the result is the same regardless of the order of execution

- Condition variables are not, and as a result they must be in a critical section to access state variables and do their job.

- It is possible to implement monitors with semaphores
Implementing Monitors with Semaphores

class Monitor {
    public:
    void ConditionWait(); // Condition Wait
    void ConditionSignal(); // Condition Signal

    private:
    <shared data>; // data being protected by monitor
    semaphore cvar; // suspends a thread on a wait
    int waiters; // number of threads waiting on
    // a cvar (one for every condition)
    semaphore lock; // controls entry to monitor
    semaphore next; // suspends this thread when signaling another
    int nextCount; // number of threads suspended

    Monitor {
        cvar = 0; // Nobody waiting on condition variable
        lock = FREE; // Nobody in the monitor
        next = nextCount = waiters = 0;
    }
}
Implementing Monitors with Semaphores

```c
ConditionWait() {    // Condition Wait
    waiters += 1;
    if (nextCount > 0)
        next.Signal();  // resume a suspended thread
    else
        lock.Signal();   // allow a new thread in the monitor
    cvar.wait();        // wait on the condition
    waiters -= 1;
}
ConditionSignal(){   // Condition Signal
    if (waiters > 0) {   // don’t signal cvar if nobody is waiting
        nextCount += 1;
        cvar.Signal();    // Semaphore Signal
        next.Wait();      // Semaphore Wait
        nextCount -= 1;
    }
}
```
Using the Monitor Class

// Wrapper code for all methods on the shared data
someMethod () {
    lock.Wait();      // lock the monitor
    // ops on data and calls to ConditionWait() and ConditionSignal()
    if (nextCount > 0)
        next.Signal();  // resume a suspended thread
    else
        lock.Signal();  // allow a new thread into the monitor
}

- Is this Hoare semantics or Mesa semantics? What would you change to provide the other semantics?
Summary

- Monitor wraps operations with a mutex
- Condition variables release mutex temporarily
- C++ does not provide a monitor construct, but monitors can be implemented by following the monitor rules for acquiring and releasing locks
- It is possible to implement monitors with semaphores
- The Java `synchronized` construct implements a limited form of monitor
• Network Structures

Next Time