Recap

- Two types of synchronization
  - Mutual exclusion → Locks
  - Ordering constraints → Condition variables

- Condition variables: Enable a thread to sleep inside a critical section by
  - Releasing lock
  - Putting thread onto waiting list
  - Going to sleep
  - After being woken, call lock()

Thread-safe queue with condition variables

```java
atomic

Thread-safe queue with condition variables

```
Monitors

- Combine two types of synchronization
  - Locks for mutual exclusion
  - Condition variables for ordering constraints
- A monitor = a lock + the condition variables associated with that lock

Mesa vs. Hoare monitors

- Mesa monitors
  - When waiter is woken, it must contend for the lock
  - So it must re-check the condition it was waiting for
- Hoare monitors
  - Special priority to woken-up waiter
  - Signalling thread immediately gives up lock
  - Signalling thread reacquires lock after waiter unlocks

Producer-consumer (bounded buffer)

- Producers put things into a shared buffer; consumers take them out
- Need to synchronize actions of producers and consumers

Producer-consumer with monitors

- Shared variables
  - State of coke machine buffers
  - numCokes (assume coke machine can hold at most MAX cokes)
- One lock (cokeLock) to protect this data

Producer-consumer with monitors

- Producer
  - cokeLock.lock()
  - while (numCokes == MAX) {
    - waitingProducers.wait()
  - }
  - add coke to machine
  - numCokes++
  - waitingProducers.signal()
  - cokeLock.unlock()

- Consumer
  - cokeLock.lock()
  - while (numCokes == 0) {
    - waitingConsumers.wait()
  - }
  - take coke out of machine
  - numCokes--
  - waitingConsumers.signal()
  - cokeLock.unlock()
Producer-consumer with monitors

Consumer

cokeLock.lock()

while (numCokes == 0) {
    waitingCons Prod.wait()
}

take coke out of machine

numCokes--

cokeLock.unlock()

waitingCons Prod.signal()

Producer

cokeLock.lock()

while (numCokes == MAX) {
    waitingCons Prod.wait()
}

add coke to machine

numCokes++

cokeLock.unlock()

waitingCons Prod.signal()
Implementing reader-writer locks with monitors

```java
readerStart() {
    while (numWriters > 0) {
        waitingReaders.wait()
    }
    numReaders++
}

readerFinish() {
    if (numReaders == 0) {
        waitingWriters.broadcast()
    }
}
```

```java
writerStart() {
    while (numReaders > 0 || numWriters > 0) {
        waitingWriters.wait()
    }
    numWriters++
}

writerFinish() {
    if (numReaders == 0) {
        waitingWriters.signal()
    }
}
```

How to program with monitors

- List shared data needed for the problem
- Assign locks to each group of shared data
- Assign condition variables for every condition that a thread holding a lock may have to wait for
  - Use while(!condition) wait
- A queue of threads waiting associated with every lock and every condition variable
- Call signal or broadcast when a thread changes something that another thread may be waiting for

Typical way to program with monitors

```java
lock
while (!condition) {
    wait
}
do stuff
signal about the stuff you did
unlock
```
Monitor Queues

Monitor bounded_buffer {
    Condition not_full;
    ...other variables...
    Condition not_empty;
    void put_resource () {
        ...wait(not_full)...  
        ...signal(not_empty)...  
    }
    Resource get_resource () {
        ...
    }
}

Waiting to enter
Waiting on condition variables
Executing inside the monitor