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()
Condition variables eliminate busy waiting

lock
...
while (queue is empty) {
  unlock
  lock
}
...
unlock

lock
...
while (queue is empty) {
  cv.wait
}
...
unlock
Thread-safe queue with condition variables

```java
    cv queueCV;
    enqueue()
        queueMutex.lock()
        find tail of queue
        add new element to tail of queue
        queueCV.signal()
        queueMutex.unlock()
    }
    dequeue()
        queueMutex.lock()
        if (queue is empty) { }
            queueCV.wait();
        }
        remove item from queue
        queueMutex.unlock()
        return removed item
    }
```
Operations on condition variables

- **wait()**
  - Atomically release lock, add thread to waiting list, go to sleep

- **signal()**
  - Wake up one thread waiting on this condition variable

- **broadcast()**
  - Wake up all threads waiting on this condition variable
  - When is this useful?
Thread-safe queue with condition variables

```c

    cv queueCV;
    enqueue(set of elements)
        queueMutex.lock()
        find tail of queue
        add new elements to tail of queue
        queueCV.broadcast()
        queueMutex.unlock()
    }

decqueue()
    queueMutex.lock()
    while (queue is empty) {
        queueCV.wait();
    }
    remove item from queue
    queueMutex.unlock()
    return removed item
    }
```
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
How to program with monitors

- List the shared data needed for the problem
- Assign locks to each group of shared data
- Each thread tries to go as fast as possible, without worrying about other threads, except for two reasons
  - **Mutual exclusion**: Enforce with lock/unlock
  - **Ordering conditions**
    - Can’t proceed because condition of shared state isn’t satisfactory
    - Some other thread must do something
    - Assign a condition variable for each situation
      - Belongs to lock that protects the shared data used to evaluate the condition
    - Use “while(!condition) { wait }”
    - Call signal() or broadcast() when a thread changes something that another thread might be waiting for
Typical way to program with monitors

lock
while (!condition) {
    wait
}
do stuff

signal about the stuff you did
unlock
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.
  - What would be required to ensure condition is met when waiter starts running again?

- Hoare monitors
  - Special priority to woken-up waiter
  - Signaling thread immediately gives up lock
  - Signaling thread reacquires lock after waiter unlocks

We (and most OSes) use Mesa monitors

Waiter is solely responsible for ensuring condition is met.
Producer-consumer (bounded buffer)

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

Why use a shared buffer?
- Lets producers and consumers operate somewhat independently

Used in many situations
- Unix pipes
- Project 1!
- Coke vending machine
Producer-consumer with monitors

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

- **When must a thread wait?**
  - Mutual exclusion (when acquiring a lock)
  - Consumer must wait if all slots are empty
    - Use condition variable `waitingConsumers`
  - Producer must wait if all slots are full
    - Use condition variable `waitingProducers`
Producer-consumer with monitors
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()

while (numCokes == 0) {
    waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()

while (1) {
    sleep(1 hour)
    while (numCokes == MAX) {
        waitingProducers.wait()
    }
    add coke to machine
    numCokes++
    waitingConsumers.signal()
}

cokeLock.unlock()
```
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()

while (numCokes == 0) {
    waitingConsumers.wait()
}

take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait()
}

add coke to machine
numCokes++

if (numCokes == 1) {
    waitingConsumers.signal()
}

cokeLock.unlock()
```
Producer-consumer with monitors

**Consumer**

```java
cokeLock.lock()

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

take coke out of machine
numCokes--

waitingCons&Prod.signal()

cokeLock.unlock()
```

**Producer**

```java
cokeLock.lock()

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

add coke to machine
numCokes++

waitingCons&Prod.signal()

cokeLock.unlock()
```
Producer-consumer with monitors

**Consumer**

cokeLock.lock()

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

take coke out of machine
numCokes--

waitingCons&Prod.broadcast()

cokeLock.unlock()

**Producer**

cokeLock.lock()

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

add coke to machine
numCokes++

waitingCons&Prod.broadcast()

cokeLock.unlock()
Producer-consumer with monitors

<table>
<thead>
<tr>
<th>Consumer</th>
<th>Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>cokeLock.lock()</td>
<td>cokeLock.lock()</td>
</tr>
<tr>
<td>while (numCokes == 0) {</td>
<td>while (numCokes == MAX) {</td>
</tr>
<tr>
<td>waitingConsumers.wait()</td>
<td>waitingProducers.wait()</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>take coke out of machine</td>
<td>add coke to machine</td>
</tr>
<tr>
<td>numCokes--</td>
<td>numCokes++</td>
</tr>
<tr>
<td>waitingProducers.signal()</td>
<td>waitingConsumers.signal()</td>
</tr>
<tr>
<td>cokeLock.unlock()</td>
<td>cokeLock.unlock()</td>
</tr>
</tbody>
</table>
Announcements

- Started with Project 1?
  - Due in a week (4\textsuperscript{th} of Feb)

- Group declaration due Wed
  - Project 2 will be posted on the Mon 4\textsuperscript{th} of Feb
Reader-writer locks

- Recall: Threads need to acquire lock even to read shared data
  - This prevents other threads from accessing the data
- Can we allow more concurrency without risking reading unstable data?

- Problem definition:
  - Shared data will be read and written by multiple threads
  - Allow multiple readers, if no threads are writing data
  - A thread can write only when no other thread is reading or writing
Need for reader-writer locks

- Use of normal mutex locks limits concurrency

**Reader:**
- lock()
- print catalog
- unlock()

**Writer:**
- lock()
- change catalog
- unlock()
Reader-writer locks

- Implement set of functions that a program can use to follow “multiple-reader, single-writer” paradigm
  - readerStart()
  - readerFinish()
  - writerStart()
  - writerFinish()

- Pros and cons compared to normal mutex locks?
Another level of abstraction

Atomic operations
(load/store, interrupt enable/disable, test&set)

Concurrent programs
(higher-level synchronization primitives
(lock, monitor, semaphore)
(even higher-level synchronization primitives
(readerStart, readerFinish, writerStart, writerFinish))
Implementing reader-writer locks with monitors

- Shared data needed to implement readerStart, readerFinish, writerStart, writerFinish?

- Condition variables?
Implementing reader-writer locks with monitors

readerStart()          writerStart()

readerFinish()         writerFinish()