## Today

- Concurrency Problems
  - Producer Consumer
    - Bounded Buffer
    - Pipes
  - Dining Philosophers
- Synchronization Mechanisms
  - Condition Variables



## Review

- What is a condition variable?
  - How do we use it?
- Can you always use a broadcast instead of a signal?



## Producer-consumer code

```
consumer () {
  sodaLock.acquire()

  while (numSodas == 0) {
    hasSoda.wait(sodaLock)
  }    CV1            Mx

  take a soda from machine

  hasRoom.signal()
    CV2
  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()

  while(numSodas==MaxSodas){
    hasRoom.wait(sodaLock)
  }    CV2            Mx

  add one soda to machine

  hasSoda.signal()
    CV1
  sodaLock.release()
}
```



## One CV: inefficient producer-consumer

```
consumer () {
  sodaLock.acquire()
  while (numSodas == 0) {
    cv.wait(sodaLock)
  }
  take a soda from machine
  cv.signal()
  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()
  while(numSodas==MaxSodas){
    cv.wait(sodaLock)
  }
  add one soda to machine
  cv.signal()
  sodaLock.release()
}
```

Consider the scenario:
- 0 sodas
- 2 consumers wait
- 1 producer adds a soda, signals

What could happen?



## One CV: inefficient producer-consumer

```
consumer () {
  sodaLock.acquire()
  while (numSodas == 0) {
    cv.wait(sodaLock)
  }
  take a soda from machine
  cv.signal()
  sodaLock.release()
}
```

```
producer () {
  sodaLock.acquire()
  while(numSodas==MaxSodas){
    cv.wait(sodaLock)
  }
  add one soda to machine
  cv.signal()
  sodaLock.release()
}
```

Consider the scenario:
- 0 sodas
- 2 consumers wait
- 1 producer adds a soda, signals

A consumer wakes up, takes a soda → 0 sodas
Signals, waking up other consumer
Consumer wakes up but no sodas! (OK because we have the while loop)



## Condition Variable Design Pattern

```
methodThatWaits() {
   lock.acquire();
   // Read/write shared
   // state

   while (
     testSharedState() ) {
      cv.wait(lock);
   }

   // Read/write shared
   // state
   lock.release();
}
```

```
methodThatSignals() {
   lock.acquire();
   // Read/write shared
   // state

   // If testSharedState is
   // now true
   cv.signal(lock);


   // Read/write shared
   // state
   lock.release();
}
```

## Summary: Condition Variables

- Condition variable is memoryless
  - If signal when no one is waiting, no op
- Wait ***atomically*** releases lock
  - What if wait, then release?
  - What if release, then wait?

```
wait (lock){                 Atomic
  release lock
  put thread on wait queue
  go to sleep
  // after wake up
  acquire lock
}
```



## Summary: Condition Variables

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast puts thread on *ready* (not running) list
  - When lock is released, anyone might acquire it
- Benefit: simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks



## Using Condition Variables

- Document the condition(s) associated with each CV.
  - What are the waiters waiting for?
  - When can a waiter expect a signal?
- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state



## Using Condition Variables

- Wait MUST be in a loop -- "Loop before you leap!"

```
while (needToWait()) {
     condition.wait(lock);
}
```

  - Another thread may beat you to the mutex.
  - The signaler may be careless.
    - Some thread packages have "spurious wakeups": 2 threads woken up, though a single signal has taken place
  - A single CV may have multiple conditions
  - Signals on CVs do not stack!
    - A signal will be lost if nobody is waiting: always check the wait condition before calling wait.



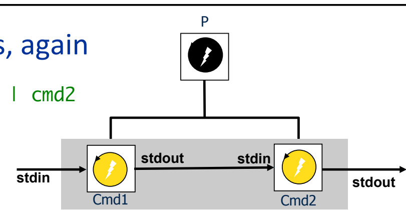
Soda machine in a computer

## BOUNDED BUFFER



## Pipes, again

cmd1 | cmd2




Read/write sys call parameters:
- The file code (file descriptor)
- The pointer to a buffer where the data is stored (buf).
- The number of bytes to be read from the buffer (nbytes).

**C1/C2 user pseudocode**

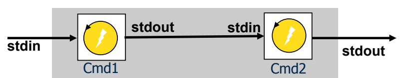
```
while(! EOF) {
      read(0, buf, count);
      compute/transform data in buf;
      write(1, buf, count);
}
```

## Pipes

What is shared?
What are the ordering constraints?

**Kernel-space pseudocode**
System call internals to read/write N bytes from pipe into buffer size B.

```
read(pipe, buf, N) {
   for (i = 0; i<N; i++) {
      move one byte from pipe into buf[i];
   }
}
```

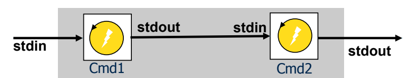





## Pipes

```
read(pipe, buf, N) {
   pipeMx.lock();
   for (int i = 0; i<N; i++) {
      while (no bytes in pipe)
         dataCv.wait();
      move one byte from pipe
      into buf[i];
      spaceCV.signal();
   }
   pipeMx.unlock();
}
```

- Read N bytes from the pipe into the user buffer named by **buf**.
- Think of this code as deep inside the **read** system call implementation on a pipe.
- The **write** implementation is similar.






## Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set
    - do not perform any updates
  - Writers – can both read and write
    - Updates
- Race conditions?



## DINING PHILOSOPHERS



## Dining Philosophers Problem


- N processes share N resources
- Resource requests occur in pairs w/ random think times
- Hungry philosopher grabs fork
  - ... and doesn't let go
  - ... until the other fork is free
  - ... and the rice is eaten

```
while(true) {
   Think();
   Eat();
}
```

What is shared?
What are the ordering constraints?

What happens in the case of 5 philosophers?
What if fewer or more philosophers?
What are my goals for a solution?



## Looking Ahead

- Project 3 due Wednesday