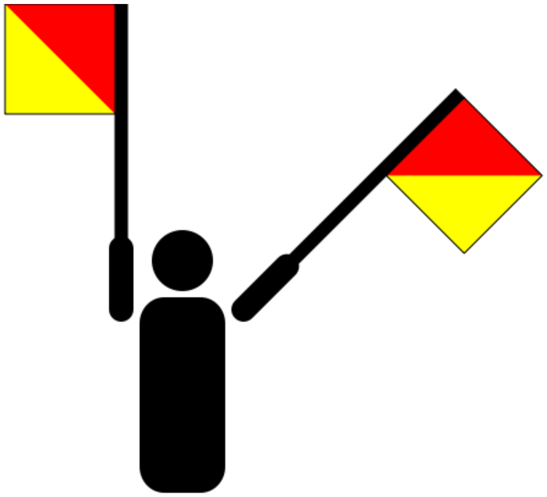
Too Few Chairs

Threads A, B, C, ...

```
if (chairAvailable()) {
    takeChair();
} else {
    waitForChair();
}
print(“I’m working!”);
releaseChair();
```

Semaphores



Too Much Milk with Semaphores

```
sem = new Semaphore(1);
```

Threads A, B

```
sem.wait();
if (noMilk) {
    buy milk;
}
sem.signal();
```

Too Few Chairs with Semaphores

```
sem = new Semaphore(NUM_CHAIRS);
```

Threads A, B, C, ...

```
sem.wait();
print(“I’m working!”);
sem.signal();
```

Implementing Semaphores

```
class Semaphore {

  public:
    void Wait(Thread T);
    void Signal();

  private:
   int value;
   Queue Q;

}

Semaphore(int val) {
   value = val;
   Q = empty;
}
```

```
Wait(Thread T) {
   value--;
   if (value < 0) {
      add T to Q;
      T->block();
   }
}

Signal() {
   value++;
   if (value <= 0){
      remove T from Q;
      wakeup(T);
   }
}
```

Must be atomic (interrupts or Test&Set)!

Producer/Consumer with Semaphores

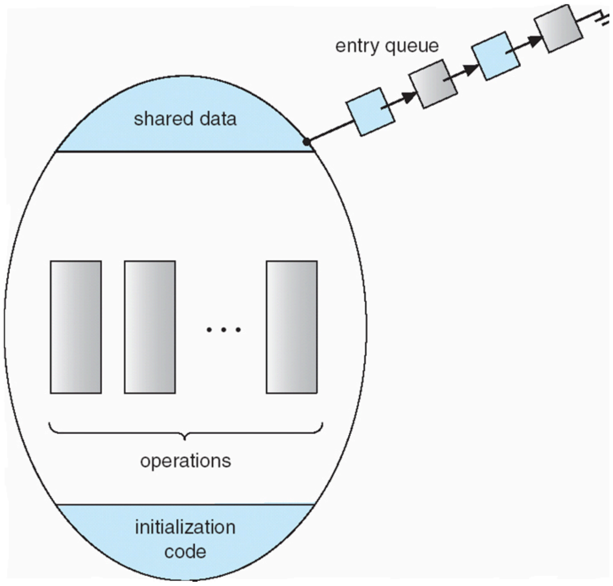
```
class ProducerConsumer {
   public:
     void Producer();
     void Consumer();
   private:
     Items buffer;
     // control buffer access
     Semaphore mutex;
     // count of free slots
     Semaphore empty;
     // count of used slots
     Semaphore full;
}

ProducerConsumer(int N) {
     mutex.value = 1;
     empty.value = N;
     full.value  = 0;
     buffer = new buffer[N];
}
```

```
void Producer() {
   <produce item>
   empty.Wait(); // one fewer slot, or wait
   mutex.Wait(); // get access to buffers
   <add item to buffer>
   mutex.Signal(); // release buffers
   full.Signal(); // one more used slot
}


void Consumer() {
   full.Wait(); //wait until there's an item
   mutex.Wait(); // get access to buffers
   <remove item from buffer>
   mutex.Signal(); // release buffers
   empty.Signal(); // one more free slot
   <use item>
}
```

Monitors



Producer/Consumer with Java Monitors

```
class Queue {

  private ...; // queue data

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

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

}
```

Producer/Consumer with Java Monitors

```
class Queue {

  private ...; // queue data

  public synchronized void add(Object item) {
    put item on queue;
    this.notify(); // wake up waiting thread, aka Signal
  }

  public synchronized Object remove() {
    while queue is empty {
      this.wait(); // give up lock and sleep
    }
    remove and return item;
  }

}
```

Synchronization Summary

- Cooperation between processes and/or threads

- Synchronization primitives provided by OS
  - Locks: acquire/release
  - Semaphores: wait/signal
  - Monitors: methods with mutex + condition variables

- All require hardware support
  - Disabling interrupts or test&set