

Sistemas Distribuídos

José Orlando Pereira

Departamento de Informática
Universidade do Minho



Example: Game

- Generic problem: Waiting for an event in a different thread
- Wait for at least Min players before joining the game:

```
void joinTheGame() {  
    ready++;  
    while (ready < Min)  
        ;  
}
```

Example: Game

- Wait for at least Min and at most Max players before joining the game:

```
void joinTheGame() {  
    ready++;  
    while (ready < Min || playing >= Max)  
        ;  
    playing++;  
}
```

Race: ready = ready + 1

Busy waiting

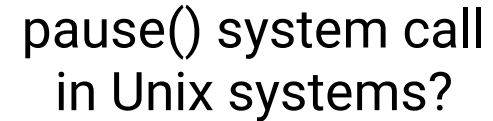
Race: playing = playing + 1

How many problems with this code?

Waiting for an event

- Assuming we can ask the OS to suspend and wakeup threads, we solve busy waiting

```
void joinTheGame() {  
    ready++;  
    if (ready < Min || playing >= Max)  
        suspendMe();  
    playing++;    // playing-- when leaving  
    wakeAllOthers(); // also on leaving the game  
}
```



pause() system call
in Unix systems?

Waiting for an event: 1st attempt

- Add locking:

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    if (ready < Min || playing >= Max)  
        suspendMe();  
    playing++;  
    wakeAllOthers();  
    l.unlock();  
}
```



Suspended with
lock acquired:
Deadlock!

Waiting for an event: 2nd attempt


- Unlock while suspended:

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    if (ready < Min || playing >= Max) {  
        l.unlock();  
        suspendMe();  
        l.lock();  
    }  
    playing++;  
    wakeAllOthers();  
    l.unlock();  
}
```

Unlock to sleep

Relock to update other variables

Waiting for an event: 2nd attempt

- Player 1:
 - lock
 - ready++;
 - not enough ready, enter “if”
 - unlock 
 - (between unlock and suspend)*
 - suspended...
possibly forever...
- Player 2:
 - lock...
 - ... acquired
 - ready++;
 - enough ready, skip “if”
 - wake suspended
 - unlock

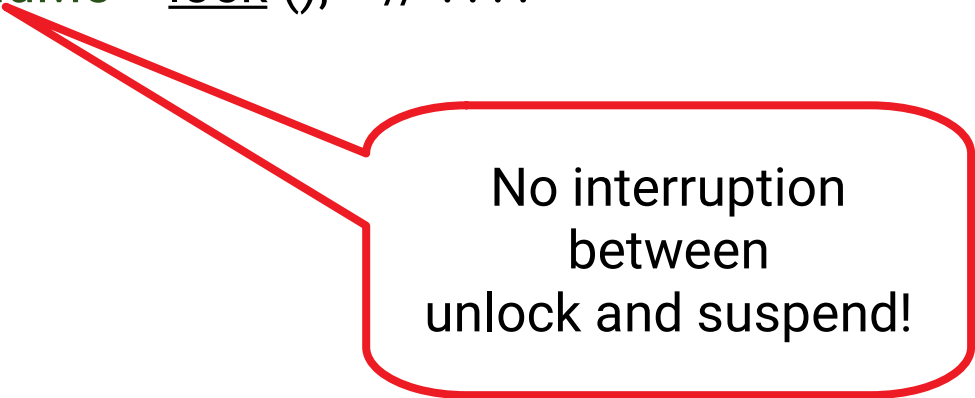
Waiting for events

- Inefficient if the waiting thread is busy polling the condition
- Leads to race conditions / deadlocks if the thread is suspended by the operating system

Waiting for an event: 3rd attempt

- Would need something like this:

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    if (ready < Min || playing >= Max) {  
        l.unlock + _suspendMe + lock (); // ????  
    }  
    playing++;  
    wakeAllOthers();  
    l.unlock();  
}
```



No interruption
between
unlock and suspend!

Condition Variables

- Atomically suspends the thread and releases the lock:

```
Lock l = new ReentrantLock();
```

```
Condition c = l.newCondition();
```

```
    c.await();    // atomically unlocks l and suspends,  
                  // relocks l on wakeup
```

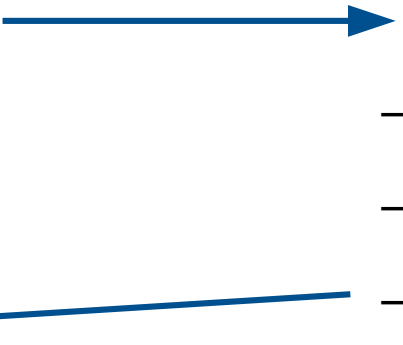
- Waking up suspended threads:
 - c.signalAll(); // all threads
 - c.signal(); // one thread

Waiting for an event: 3rd attempt



- Atomically unlock and suspend
- Relock when waking up

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    if (ready < Min || playing >= Max)  
        c.await();  
    playing++;  
    c.signalAll();  
    l.unlock();  
}
```

Waiting for an event: 3rd attempt

- Player $i < \text{Min}$:
 - lock
 - `ready++`;
 - not enough ready, enter “if”
 - unlock and suspend
 - ...
 - ...
 - wakes up! (relock)
 - continues!
 - Player $i = \text{Min}$:
 - lock...
 - ... acquired
 - `ready++`;
 - enough ready, skip “if”
 - wake suspended
 - unlock
- 

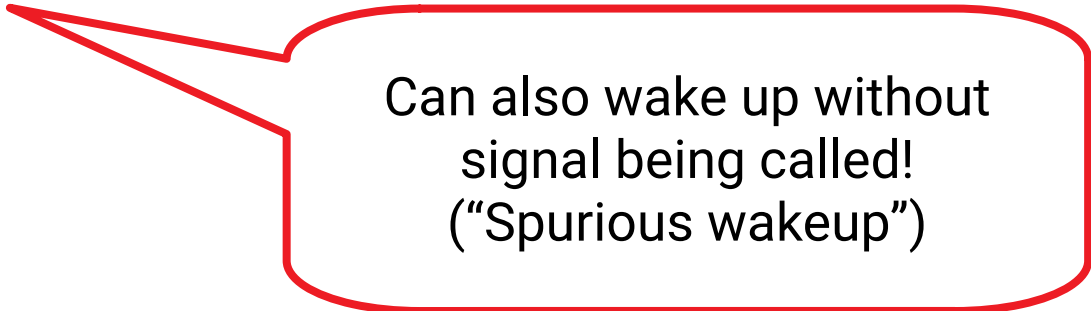
Waiting for an event: 3rd attempt

- Players $i < \text{Min}$:
 - lock
 - ready++;
 - not enough ready, enter “if”
 - unlock and suspend
 - ...
 - waking up... 
- Players $\text{Min} \leq j < \text{Min} + \text{Max}$:
 - player joins ($\text{playing}++ = 1$)
 - ...
 - player joins ($\text{playing}++ = \text{Max}$)
- finally wakes up and gets lock!
- playing++ 

Waiting for an event: 4th version

- Must always use “while” loop:

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    while (ready < Min || playing >= Max) {  
        c.await();  
    }  
    playing++;  
    c.signalAll();  
    l.unlock();  
}
```



Can also wake up without
signal being called!
("Spurious wakeup")

Signaling an event

```
void joinTheGame() {  
    l.lock();  
    ready++;  
    while (ready < Min ||  
           playing >= Max) {  
        c.await();  
    }  
    ...  
    l.unlock();  
}
```

```
void joinTheGame() {  
    l.lock();  
    ...  
    playing++;  
    c.signalAll();  
    l.unlock();  
}
```

>1 or not all
interested

```
void leaveTheGame() {  
    l.lock();  
    playing--;  
    c.signal();  
    l.unlock();  
}
```

At most any 1
interested

Waiting for an event: General case

```
Lock l = new ReentrantLock();
```

```
Condition c = l.newCondition();
```

```
void waitForEvent() {
```

```
    l.lock();
```

```
    ...
```

```
    while(!happened)
```

```
        c.await();
```

```
    ...
```

```
    l.unlock();
```

```
}
```

```
void event() {
```

```
    l.lock();
```

```
    ... // change state
```

```
    c.signal() or c.signalAll();
```

```
    l.unlock();
```

```
}
```

changes some value that
makes the condition true

wakes up
waiting threads

Conclusions

- Condition variables for efficiently / correctly waiting for events in concurrent programs
- Use `await()` always within a “while” loop due to:
 - Races
 - Spurious wakeups

Example: Level crossing



(a.k.a. Readers-Writers Lock)

RW lock: Identify conditions

- For a reader:

```
lock() {  
    while(there is a writer)  
        wait...  
}
```

- For a writer:

```
lock() {  
    while(there is anyone)  
        wait...  
}
```

RW lock: Identify conditions

- For a reader:

```
lock() {  
    while(there is a writer)  
        wait...  
}
```

- For a writer:

```
lock() {  
    while(there is anyone)  
        wait...  
}
```

- How to implement conditions? Add sufficient state:
 - int nr_readers
 - ~~int nr_writers~~ → boolean a_writer

RW lock: Add state

- For a reader:

```
lock() {  
    while(there is a writer)  
        wait...  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
}
```

- For a writer:

```
lock() {  
    while(there is anyone)  
        wait...  
    a_writer = true  
}  
unlock() {  
    a_writer = false  
}
```

RW lock: Write conditions

- For a reader:

```
lock() {  
    while(a_writer)  
        wait...  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
}
```

- For a writer:

```
lock() {  
    while(a_writer ||  
        nr_readers > 0)  
        wait...  
    a_writer = true  
}  
unlock() {  
    a_writer = false  
}
```

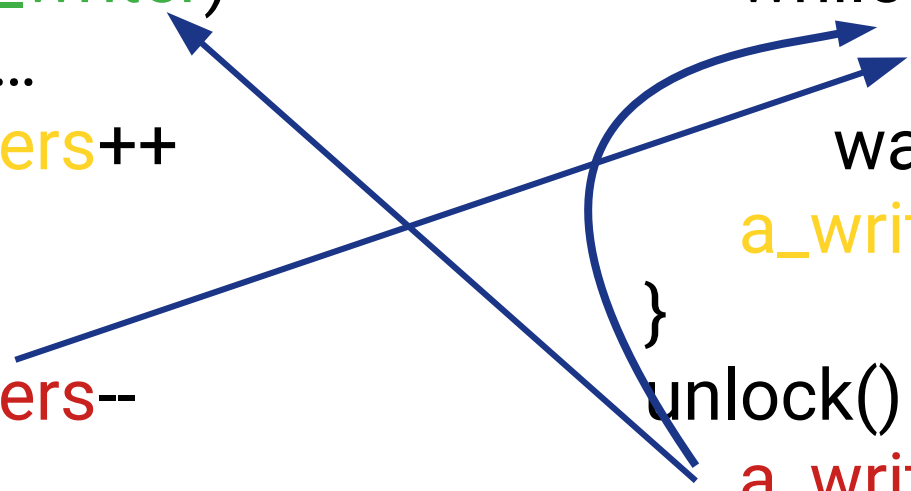
RW lock: Identify events

- For a reader:

```
lock() {  
    while(a_writer)  
        wait...  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
}
```

- For a writer:

```
lock() {  
    while(a_writer ||  
          nr_readers > 0)  
        wait...  
    a_writer = true  
}  
unlock() {  
    a_writer = false  
}
```



RW lock: Implement events

- For a reader:

```
lock() {  
    while(a_writer)  
        c.await()  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
    c.signalAll()  
}
```

- For a writer:

```
lock() {  
    while(a_writer ||  
        nr_readers > 0)  
        c.await()  
    a_writer = true  
}  
unlock() {  
    a_writer = false  
    c.signalAll()  
}
```


RW lock: Add locks

```
void readLock() {  
    l.lock();  
    while(a_writer)  
        c.await();  
    nr_readers++;  
    l.unlock();  
}  
void readUnlock() {  
    l.lock();  
    nr_readers--;  
    c.signalAll();  
    l.unlock();  
}
```

```
void writeLock() {  
    l.lock();  
    while(a_writer ||  
        nr_readers>0)  
        c.await();  
    a_writer = true;  
    l.unlock();  
}  
void writeUnlock() {  
    l.lock();  
    a_writer = false;  
    c.signalAll();  
    l.unlock();  
}
```

RW lock: Reduce wakeups

```
void readLock() {  
    l.lock();  
    while(a_writer)  
        c.await();  
    nr_readers++;  
    l.unlock();  
}  
void readUnlock() {  
    l.lock();  
    nr_readers--;  
    if (nr_readers==0) c.signal();  
    l.unlock();  
}
```

```
void writeLock() {  
    l.lock();  
    while(a_writer ||  
        nr_readers>0)  
        c.await();  
    a_writer = true;  
    l.unlock();  
}  
void writeUnlock() {  
    l.lock();  
    a_writer = false;  
    c.signalAll();  
    l.unlock();  
}
```

RW lock: Fairness

- Will readers and writers eventually access the critical section?



RW lock: Identify conditions

- For a reader:

```
lock() {  
    while(there is a writer  
         inside or waiting)  
        wait...  
}
```

- For a writer:

```
lock() {  
    while(there is anyone)  
        wait...  
}
```

- How to implement conditions? Add sufficient state:
 - int nr_readers
 - ~~int nr_writers~~ → boolean a_writer
 - int waiting

RW lock: Add state and conditions

- For a reader:

```
lock() {  
    while(writer inside  
           || waiting>0)  
        wait...  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
}
```

- For a writer:

```
lock() {  
    waiting++  
    while(there is anyone)  
        wait...  
    a_writer = true  
    waiting--  
}  
unlock() {  
    a_writer = false  
}
```

RW lock: Implement events

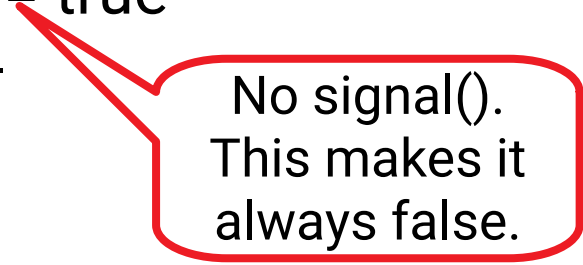
```
lock() {  
    while(a_writer ||  
          waiting > 0)  
        c.await()  
    nr_readers++  
}  
unlock() {  
    nr_readers--  
    c.signalAll()  
}
```

```
lock() {  
    waiting++  
    while(a_writer ||  
          nr_readers > 0)  
        c.await()  
    a_writer = true  
    waiting--  
}  
unlock() {  
    a_writer = false  
    c.signalAll()  
}
```

signal?



No signal().
This makes it
always false.



RW lock: Reduce wakeups

```
void readLock() {  
    l.lock();  
    while(a_writer ||  
          waiting>0)  
        c.await();  
    nr_readers++;  
    l.unlock();  
}  
void readUnlock() {  
    l.lock();  
    nr_readers--;  
    if (nr_readers==0) c.signalAll();  
    l.unlock();  
}
```

```
void writeLock() {  
    l.lock();  
    waiting++;  
    while(a_writer ||  
          nr_readers>0)  
        c.await();  
    a_writer = true;  
    waiting--;  
    l.unlock();  
}  
void writeUnlock() {  
    l.lock();  
    a_writer = false;  
    c.signalAll();  
    l.unlock();  
}
```

Cannot be
optimized to
c.signal()!

Conclusions

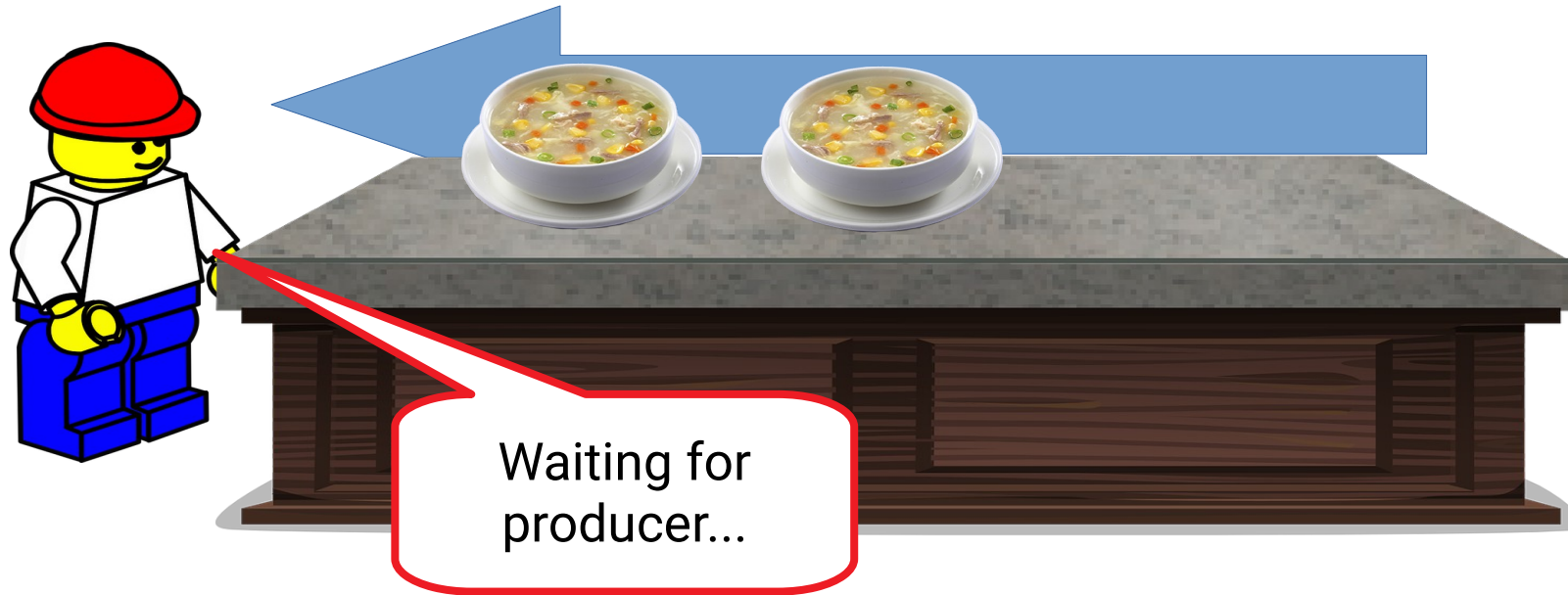
- `signalAll()`:
 - Needed to avoid deadlocks... but wasteful / inefficient
 - How to avoid it?

Example: Soup counter



(a.k.a. Bounded Buffer)

Blocking consumer



Bounded buffer: blocking consumer

```
Lock l = new ReentrantLock();  
Condition c = l.newCondition();  
Queue<Object> q = ...;
```

```
Object get() {  
    l.lock();  
  
    while(q.isEmpty())  
        c.await();  
    s = q.remove();  
    l.unlock();  
}
```

```
void put(Object s) {  
    l.lock();  
    q.add(s);  
    c.signalAll();  
    l.unlock();  
}
```

changes some value that
makes the condition true

wakes up
waiting threads

Why signal ALL
if only one
continues?


Bounded buffer: blocking consumer

```
Lock l = new ReentrantLock();  
Condition c = l.newCondition();  
Queue<Object> q = ...;
```

```
Object get() {  
    l.lock();  
  
    while(q.isEmpty())  
        c.await();  
    s = q.remove();  
    l.unlock();  
}
```

```
void put(Object s) {  
    l.lock();  
    q.add(s);  
    c.signal();  
    l.unlock();  
}
```

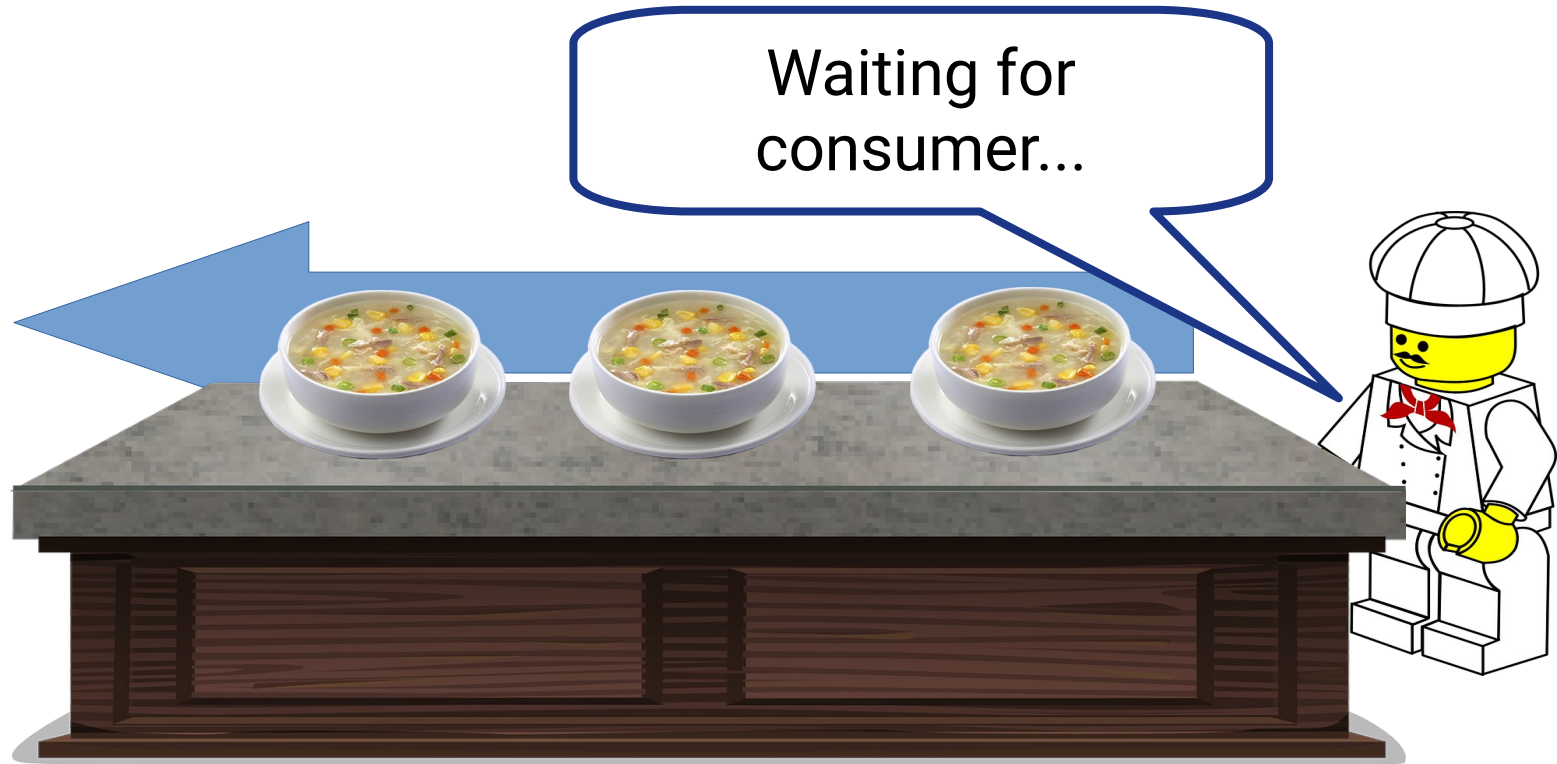
changes some value that
makes the condition true



wakes up ONE
waiting thread



Blocking producer



Bounded buffer: blocking producer

```
Lock l = new ReentrantLock();  
Condition c = l.newCondition();  
Queue<Object> q = ...;
```

```
Object get() {  
    l.lock();  
    s = q.remove();  
    c.signal();  
    l.unlock();  
}
```

```
void put(Object s) {  
    l.lock();  
    while(q.size() >= Max)  
        c.await();  
    q.add(s);  
    l.unlock();  
}
```

Bounded buffer: blocking both

```
Lock l = new ReentrantLock();  
Condition c = l.newCondition();  
Queue<Object> q = ...;
```

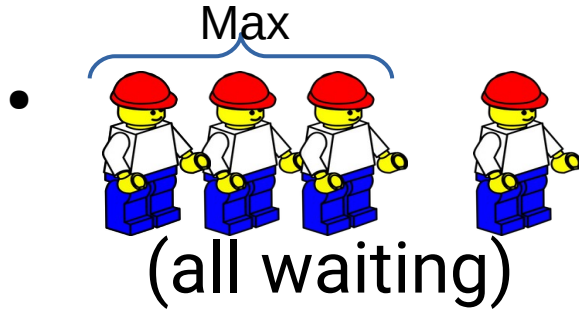
```
Object get() {  
    l.lock();  
    while(q.isEmpty())  
        c.await();  
    s = q.remove();  
    c.signal();  
    l.unlock();  
}
```

```
void put(Object s) {  
    l.lock();  
    while(q.size() >= Max)  
        c.await();  
    q.add(s);  
    c.signal();  
    l.unlock();  
}
```

?

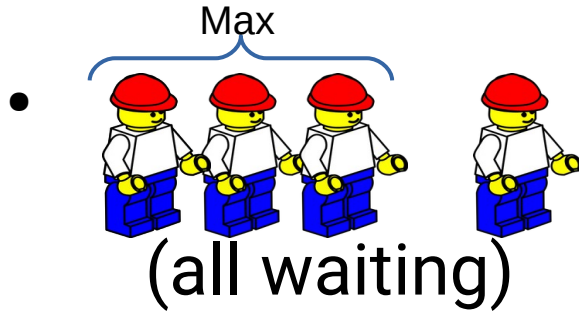
Is it possible to have
producers and consumers
blocked at the same time?

Bounded buffer

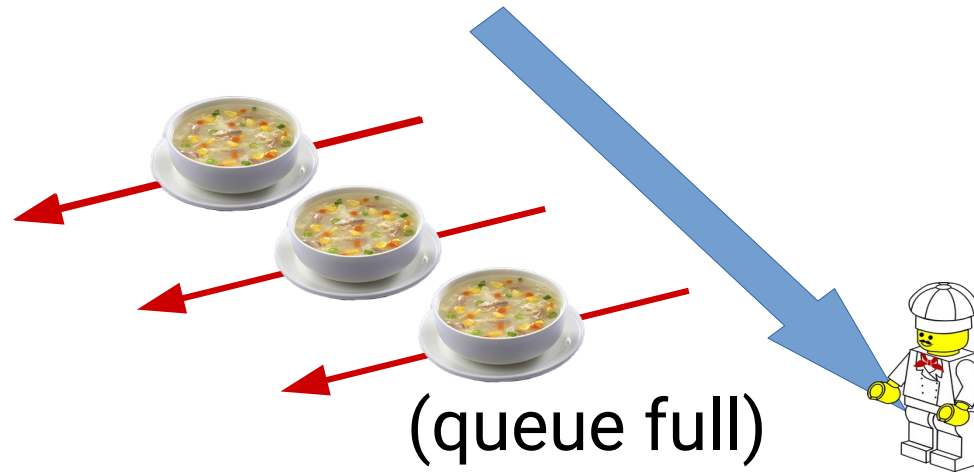


- Buffer of size Max
- More than Max consumers waiting...
- All producers busy elsewhere....

Bounded buffer

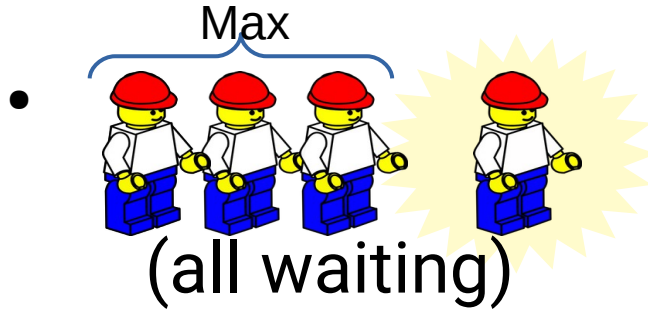


(3 waiting)
(2 waiting)
(1 waiting)

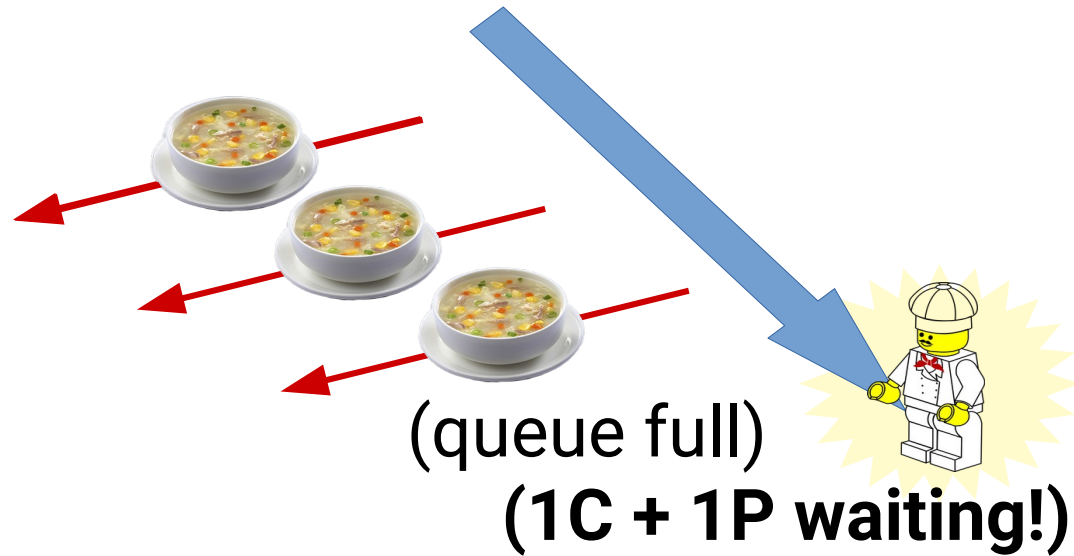


- Producer tries to put more than Max and blocks

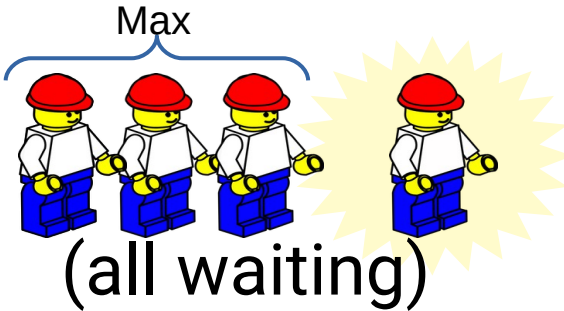
Bounded buffer



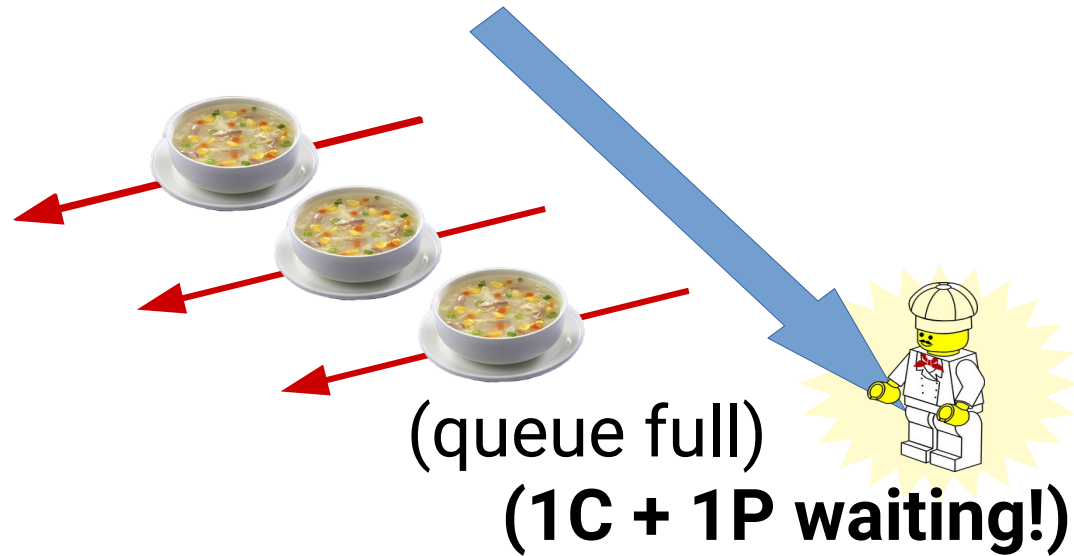
(3 waiting)
(2 waiting)
(1 waiting)



Bounded buffer

- 
(all waiting)

(3 waiting)
(2 waiting)
(1 waiting)



-  signal() wakes **Consumer** or Producer?

Deadlock

Bounded buffer

- This is a general problem with different conditions and the same condition variable
- Workaround: Use `signalAll()`
 - Inefficient (“thundering herd”)
 - This was the only solution with Java “synchronized” monitors (with `notifyAll()`)



Bounded buffer: 2 condition variables

```
Lock l = new ReentrantLock();  
Condition notEmpty = l.newCondition();  
Condition notFull = l.newCondition();  
Queue<Object> q = ...;
```

```
Object get() {  
    l.lock();  
    while(q.isEmpty())  
        notEmpty.await();  
    q.remove();  
    notFull.signal();  
    l.unlock();  
}
```

```
void put(Object s) {  
    l.lock();  
    while(q.size() >= Max)  
        notFull.await();  
    q.add(s);  
    notEmpty.signal();  
    l.unlock();  
}
```

Summary

- Using multiple condition variables for the same lock reduces the need to use `signalAll()`
- But... not easy to be sure!!!
- “Bounded buffer” is an important building block for concurrent programs:
 - Unix *pipes*
 - *Sockets* in distributed programs