Sistemas Distribuídos

José Orlando Pereira

Departamento de Informática
Universidade do Minho

2018/2019

Example: Game

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

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

Race: ready = ready + 1

Busy waiting

How many problems with this code?

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++;          /*leaving
}
```

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
}
```

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… forever...

- Player 2:
  - lock…

  … acquired
  - ready++;
  - enough ready, skip “if”
  - wake suspended
  - unlock

Deadlock!

Condition Variables

- Atomically suspends the thread and releases the lock:

```
Lock l = new ReentrantLock();

Condition c = l.newCondition();

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

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

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 < Min:
  - lock
  - ready++;
  - not enough ready, enter “if”
  - unlock and suspend
    ...
    ...
  - wakes up!
    (relock)
  - continues!

- Player i = Min:
  - lock…
    … acquired
  - ready++;
  - enough ready, skip “if”
  - wake suspended
  - unlock

Waiting for an event: 3rd attempt

- Player 1:
  - lock
  - ready++;
  - not enough ready, enter “if”
  - unlock and suspend

    ...
  - wakes up!
  - finally gets lock!
  - playing++

Playing > Max!

- Players 2 to Max:
  - player 2 joins
  - ...
  - player Max joins

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”)

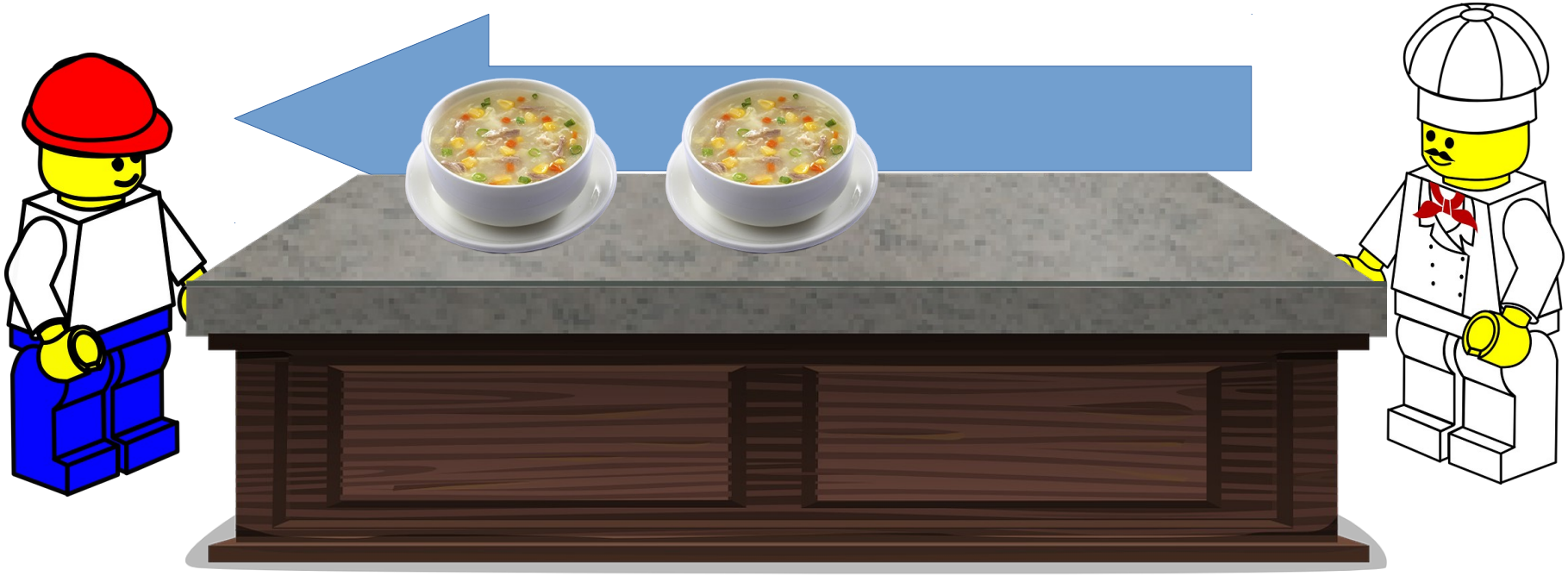
Waiting for an event: General case

```
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.signalAll();
    l.unlock();
}
```

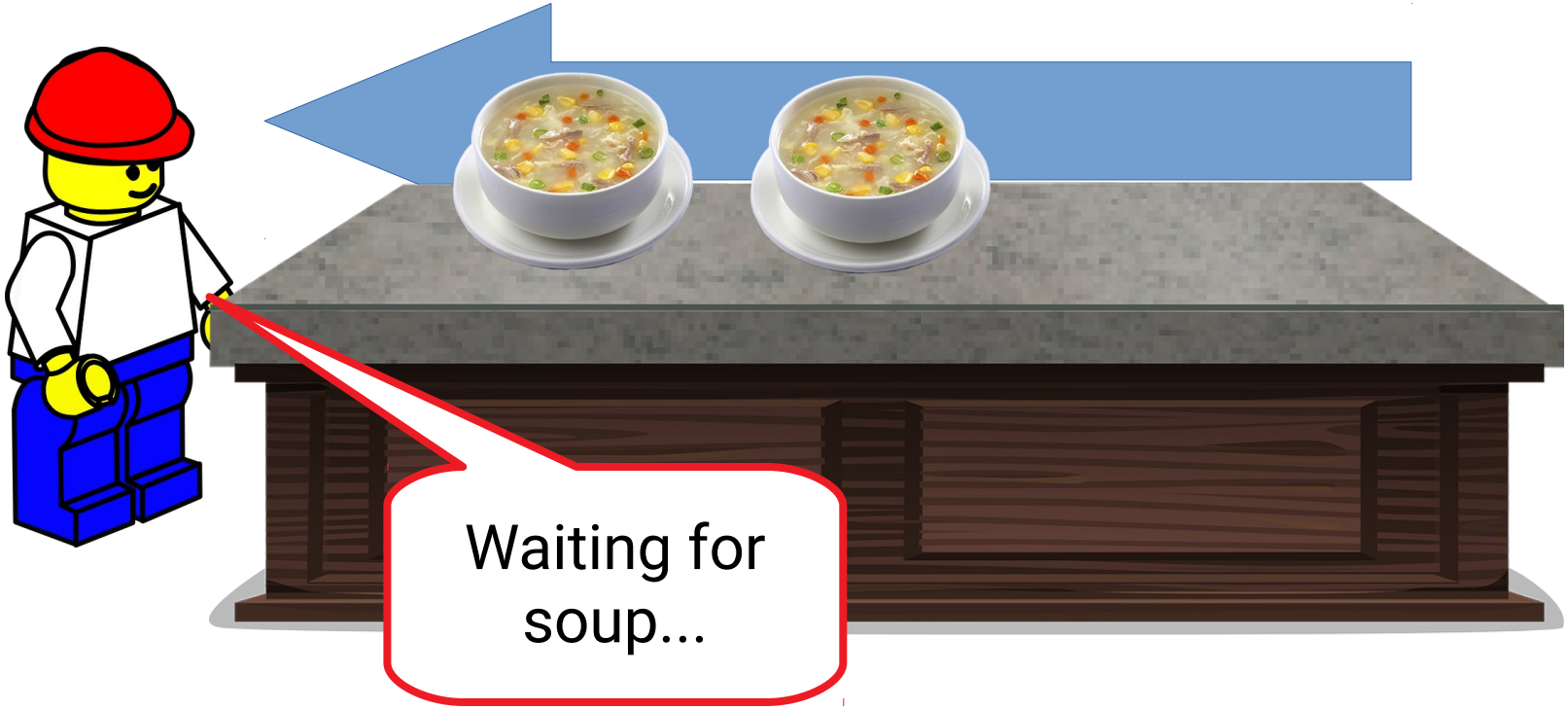
changes some value that makes the condition true

wakes up waiting threads

Example: Soup counter



Blocking reader



Bounded buffer: blocking reader

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

Object get() {
    l.lock();

    while(q.isEmpty())
        c.await();
    q.remove();
    l.unlock();
}
```

changes some value that makes the condition true

wakes up waiting threads

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

Why signal ALL if only one continues?

Bounded buffer: blocking reader

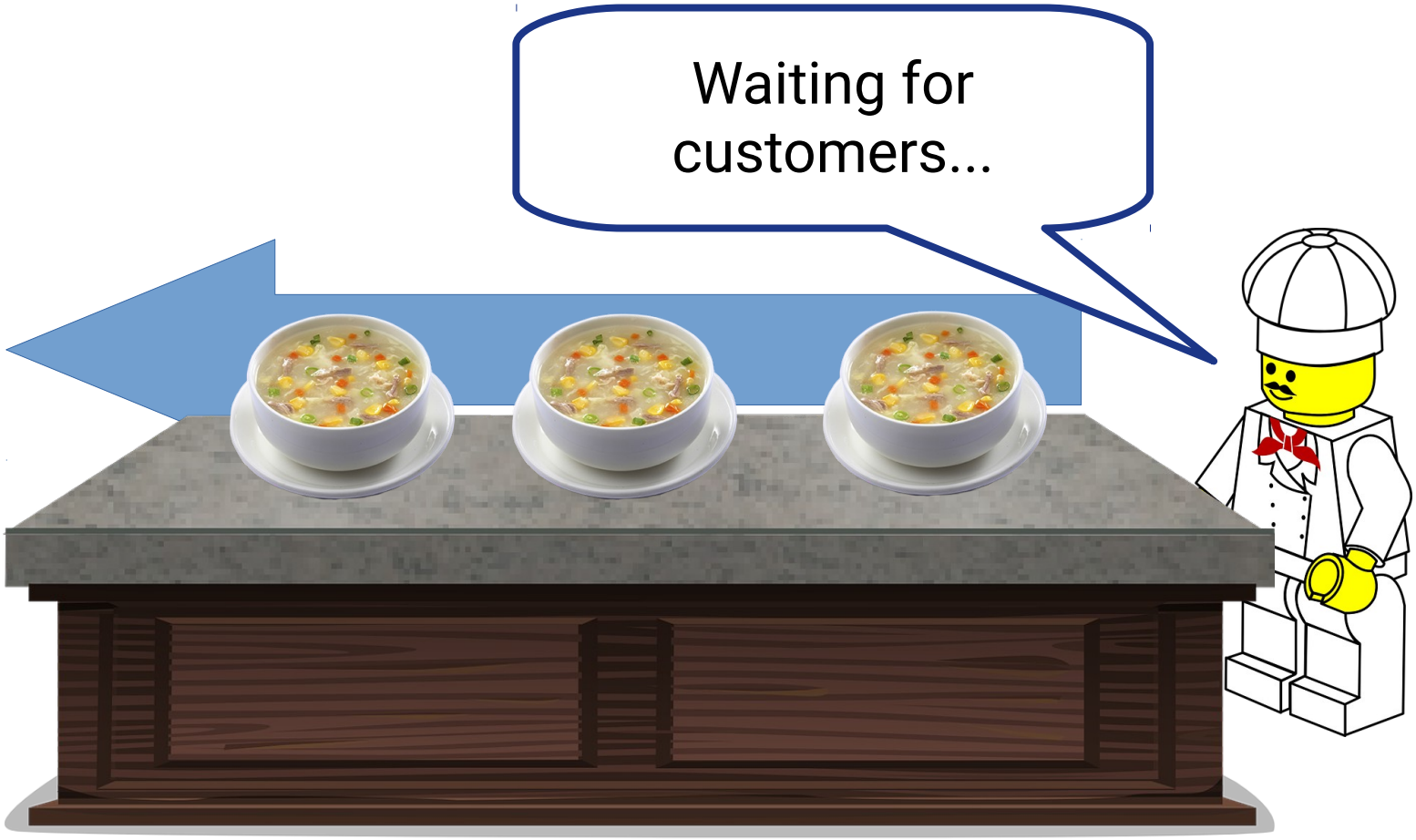
```
Lock l = new ReentrantLock();
Condition c = l.newCondition();
Queue<Object> q = ...;

Object get() {
    l.lock();

    while(q.isEmpty())
        c.await();
    q.remove();
    l.unlock();
}
```

changes some value that makes the condition true

wakes up ONE waiting thread

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

Blocking writer



Bounded buffer: blocking writer

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

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

```
void put(Object s) {
    l.lock();

    while(q.size()>=Max)
        c.await()
    q.add(s);
    l.unlock();
}
```

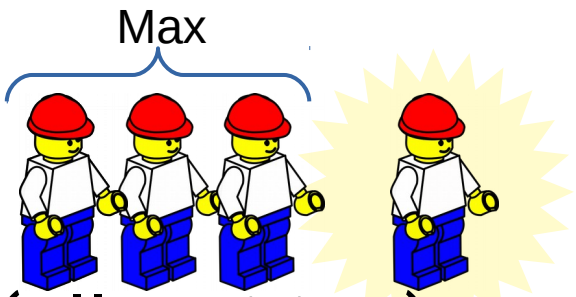
Bounded buffer: blocking both

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

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

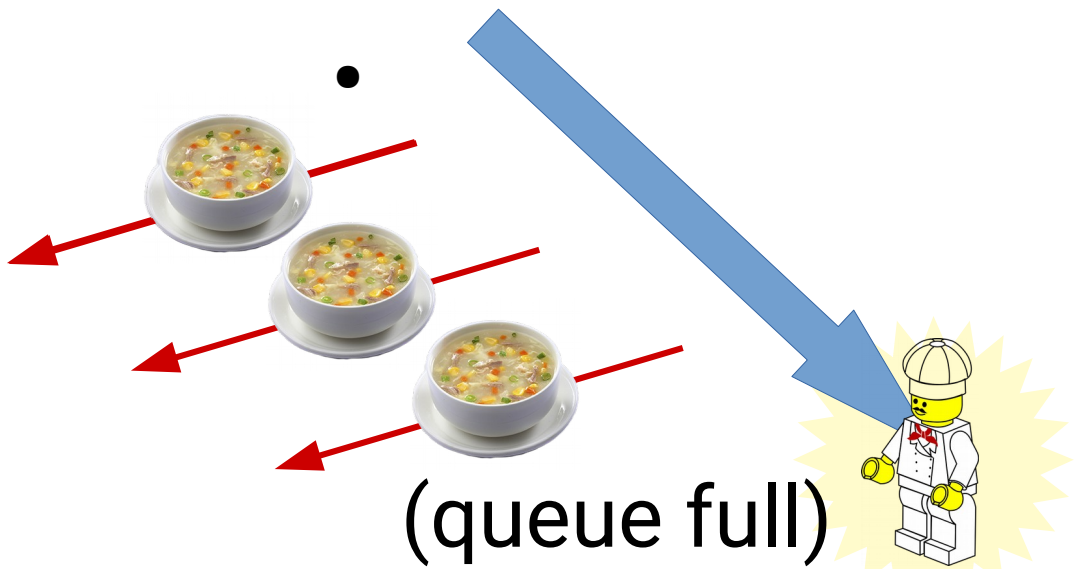
?

Is it possible to have readers and writers blocked at the same time?

Bounded buffer

- 

(all waiting)

- 

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

(queue full)
(1R + 1W waiting!)

-  signal(): R or W?

Deadlock

Bounded buffer

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

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();
}
```

Conclusions

- Use await() <u>always</u> within a “while” loop
  - Races
  - Spurious wakeups
- Minimizing wakeups improves performance and scale
  - Use one condition variable for each condition to avoid signalAll()
- Blocking bounded buffer is an important building block in concurrent programming

j.u.c Conditions vs Monitors

- class C {

There is a hidden "condition" in each object used by "wait()/notify()"

```
synchronized public void m1() {
    while(...) wait();
}

synchronized public void m2() {
    notify();
}
}
```

- class C {

```
private Lock l =
    new ReentrantLock();
private Condition c =
    l.newCondition();
public void m() {
    try { l.lock();
        while(...) c.await()
    } finally { l.unlock(); }
}
public void m() {
    try { l.lock();
        c.signal();
    } finally { l.unlock(); }
}
}
```

Equivalent code (aproximately...)

j.u.c. Conditions vs Monitors

- Main differences:
  - One implicit condition for each lock
    vs.
    Many j.u.c. conditions for the same lock
    - Avoids signalAll()
  - Threads waiting for a condition wakeup in any order
    vs.
    Threads waiting for a j.u.c. Condition wakeup in FIFO order