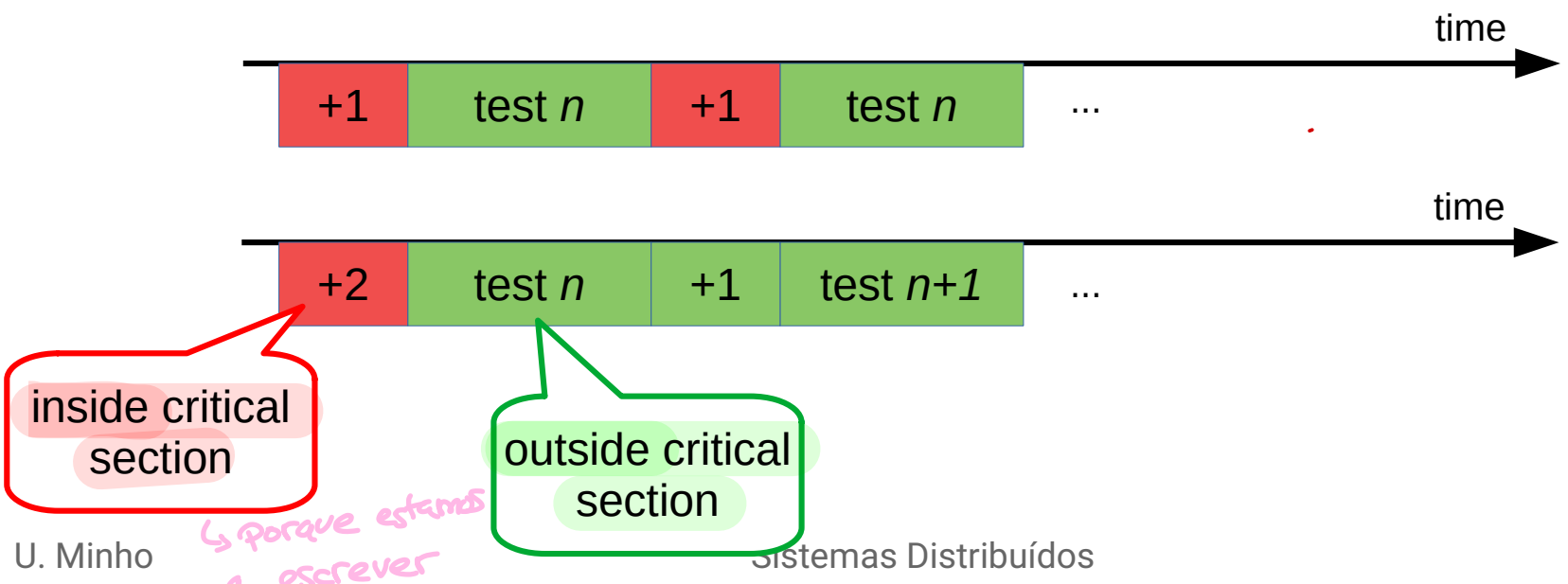
Sistemas Distribuídos

José Orlando Pereira

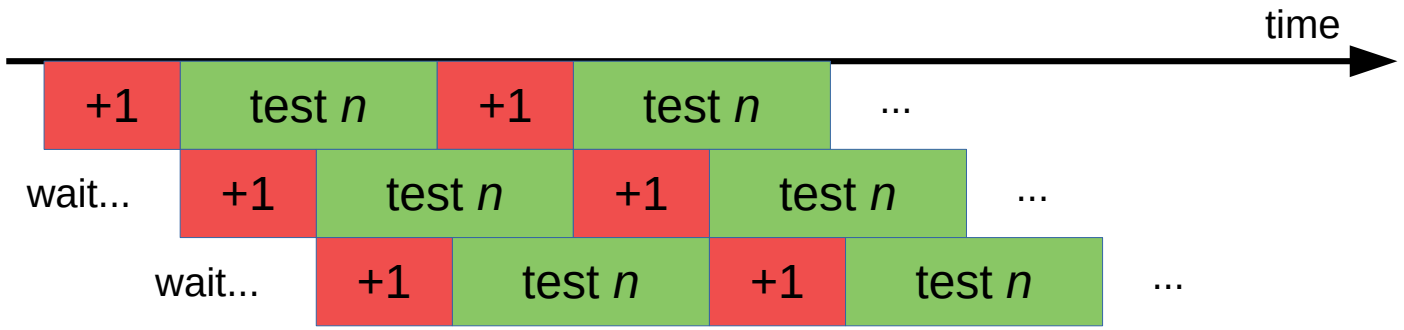
Departamento de Informática
Universidade do Minho

Motivation

- Consider two versions of the parallel primality testing code:
  - Increment +1 and get *n*, test *n*
  - Increment +2 and get *n*, test *n* and *n+1*

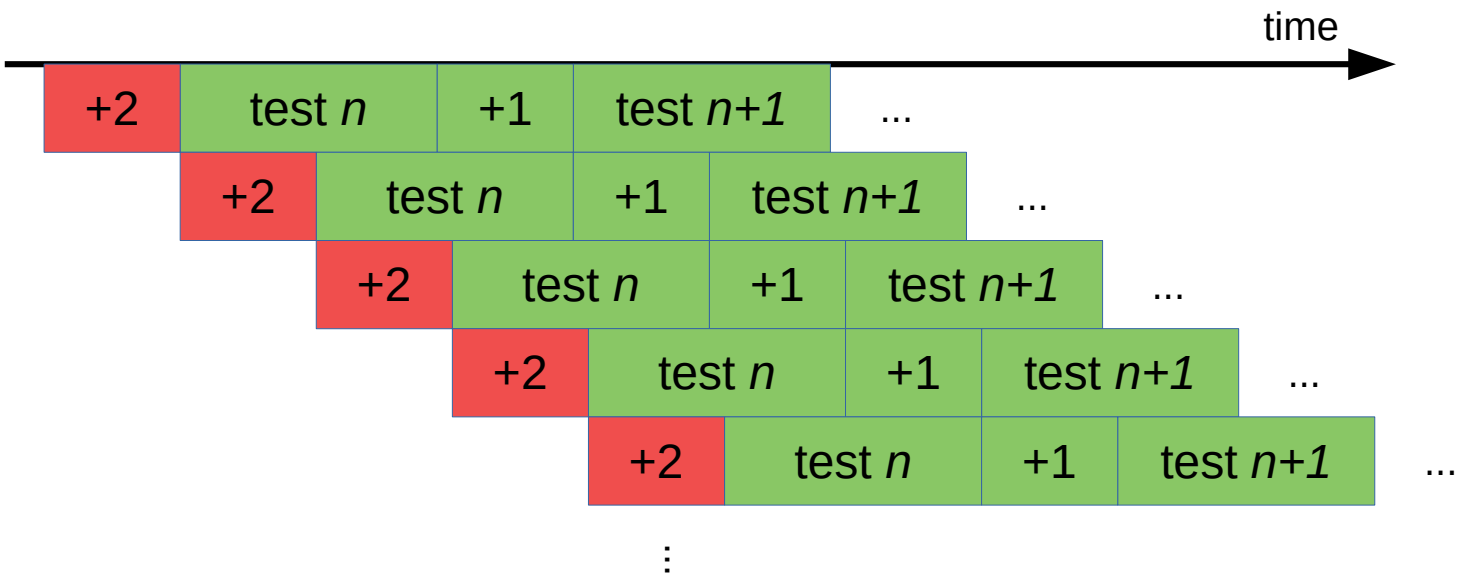


Motivation



- Eventually, at least one thread is blocked waiting for mutex...

Motivation

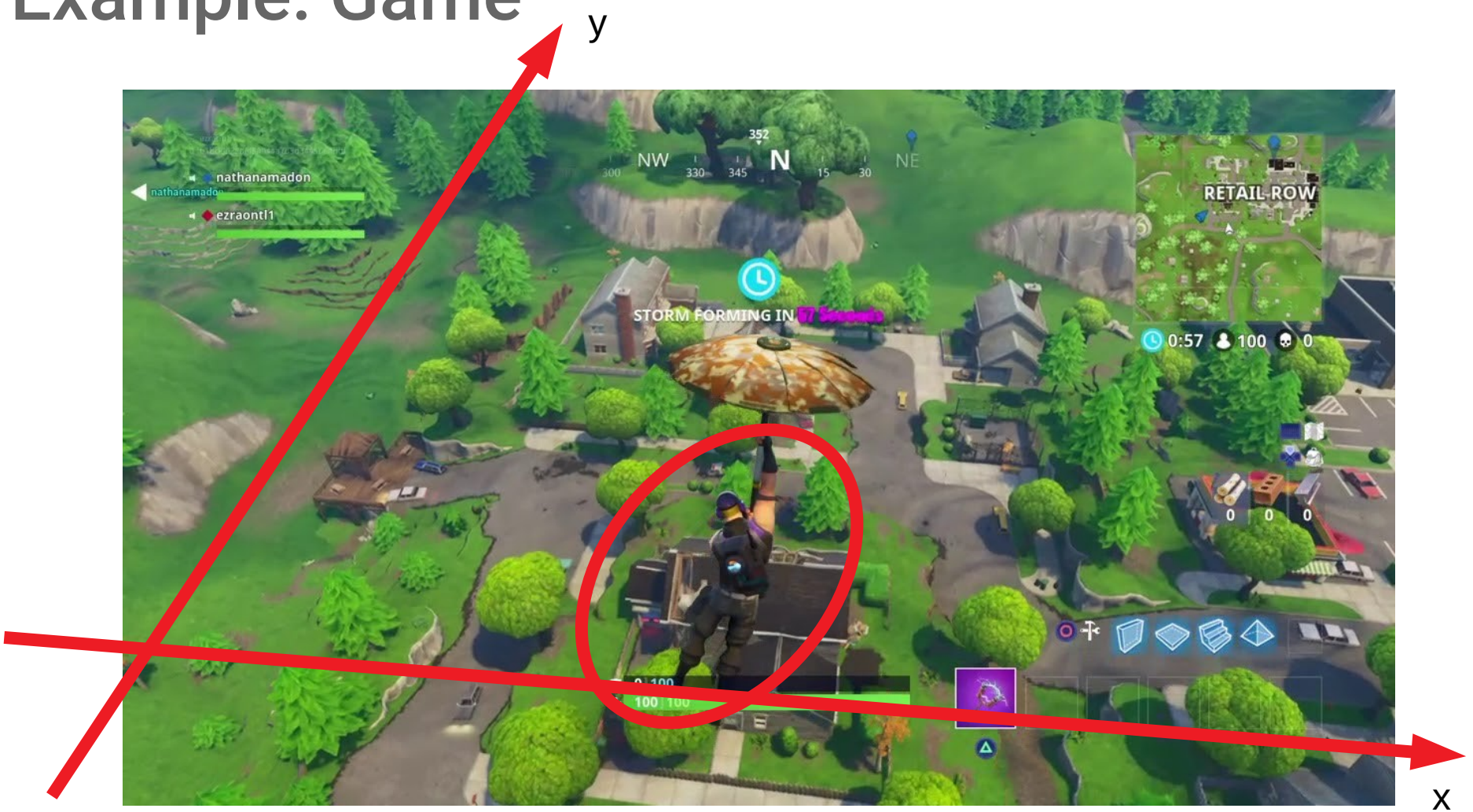


- Reducing the contention on critical sections lessens the performance impact of synchronization

Roadmap

- Use synchronization primitives to write correct concurrent code and avoid busy waiting
- Need to minimize **time** in critical sections
- Need to minimize **contention** in critical sections

Example: Game



Game state and operations

- State:
  - Map<String,Player> players;
  - class Player {
    int x,y;
    int life, score;
    }
- Operations:
  - drop in the game, move, and shoot
  - draw the game

First approach

- 1 thread for each player(*)
- 1 lock for the shared game state

(*) Later we make it distributed…

First approach

- void write(Output o) {
  try { l.lock();
  players.values().forEach(p→o.write(p.x, p.y));
  } finally { l.unlock(); }
  }
- Problems:
  - Either sending or moving
  - Writing may take a long time (blocking)
  - *“Lag”...*

Try/finally make it work with exceptions

Lengthy computation inside critical section

Immutable objects

- class Coord { final int x, y; }
- class Player {
  Coord xy;
  int life, score;
  }
- void write(Output o) {
  try { l.lock();
  c=players.values().stream()
  .map(p→p.xy).collect(toList());
  } finally { l.unlock(); }
  c.forEach(c→o.write(c.x, c.y));
  }

All fields final

logo quando inicializados não podem ser modificados

Lengthy computation outside critical section

Multiple locks

- Can’t move two players concurrently
- Forget “drop in the game” for now…
- Use one lock for each player:
- class Player {

```
    Lock l;
    Coord xy;
    int life, score;
}
```

Multiple locks

- void move(...) {
  try { l.lock();
  xy = new Coord(…);
  } finally { l.unlock(); }
  }
- Coord getLocation() {
  try { l.lock();
  return xy;
  } finally { l.unlock(); }
  }

modifica as variáveis

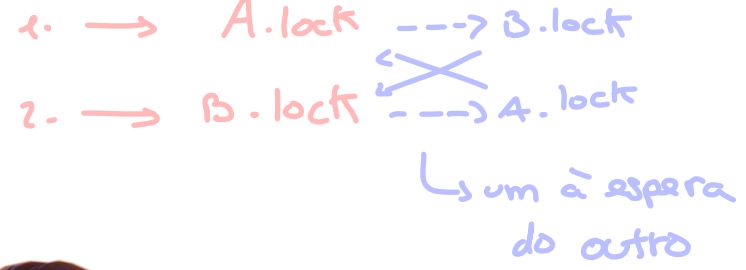
pega nas variáveis

Multiple locks

- void shoot(String sn, String tn) {
  Player s = players.get(sn);
  Player t = players.get(tn);
  try { s.l.lock(); t.l.lock();
  t.life--;
  s.score++;
  } finally { t.l.unlock(); s.l.unlock(); }
  }

Deadlock

- What if two players shoot at each other simultaneously (A → B and B → A) ?
- What if A → B, B → C and C → A?
- What if ...



Lock ordering

- What if two players A, B shoot at each other simultaneously?
  - A acquires A, B
  - B acquires A, B
- What if A → B, B → C and C → A?
  - A acquires A, B
  - B acquires B, C
  - C acquires A, C

↳ nenhum deles bloqueia o 1º ace der o lock termina e o 2º espera o 1º dar unlock

Lock ordering

```
void shoot(String sn, String tn) {
    Player s = players.get(sn);
    Player t = players.get(tn);
    try { Stream.of(sn,tn).sorted()
            .forEach(n→players.get(n).l.lock());
        t.life--;
        s.score++;
    } finally { t.l.unlock(); s.l.unlock(); }
}
```

ordenar para sempre dar lock na mesma ordem

Acquire locks in a fixed order

Release in any order

Fairness

- “Doesn’t lock ordering mean that player A has an advantage?”
- No. It means that:
  - When A shoots some X and X shoots A, at the same time, the winner will be decided by lock of A
  - Any j.u.c.ReentrantLock is fair or, optionally, FIFO
- So they have the same chances regardless of the lock used

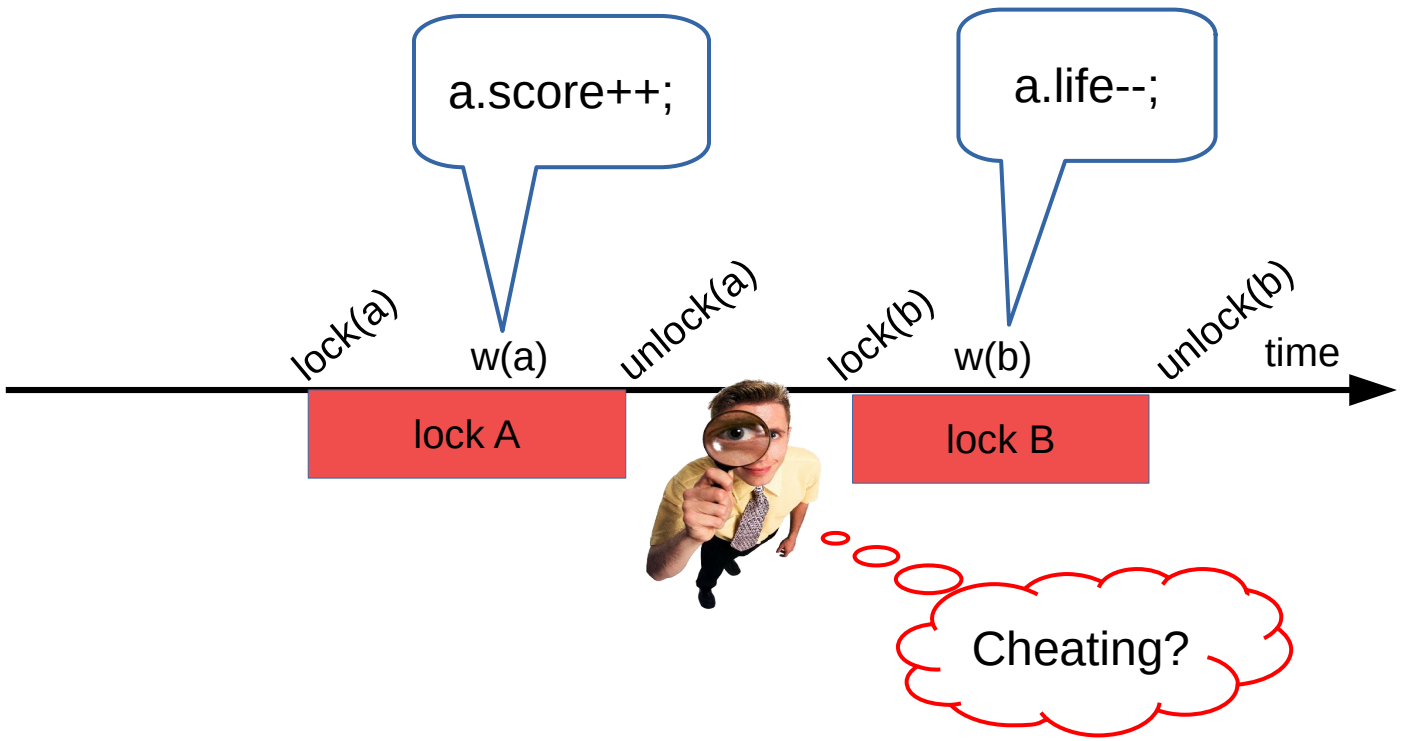
Multiple locks

- Acquiring all locks needed at the start and releasing them at the them of an operation works as well as single global lock

- What if we need to read some data before acquiring further locks?
- How to further reduce the time holding locks?

Multiple locks

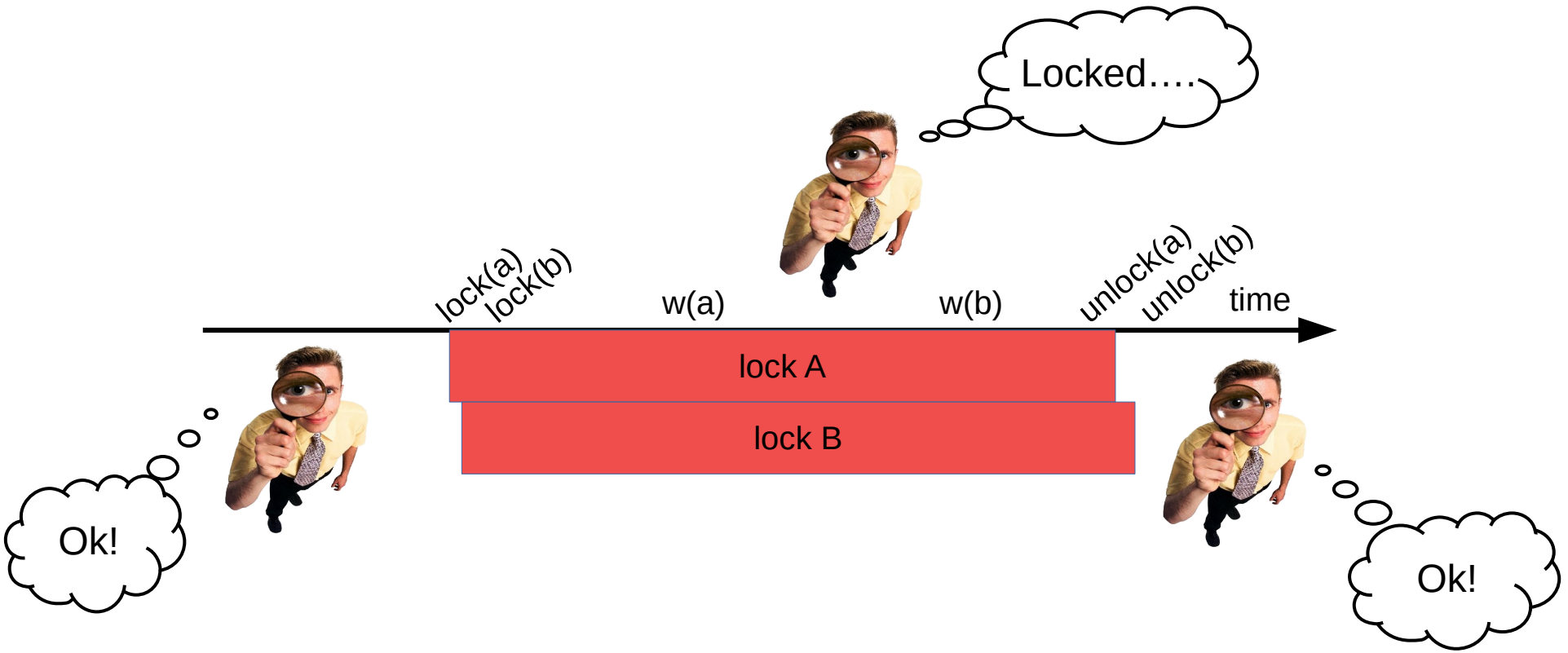
- Why acquire both locks simultaneously?
  - If we don’t….



Multiple locks

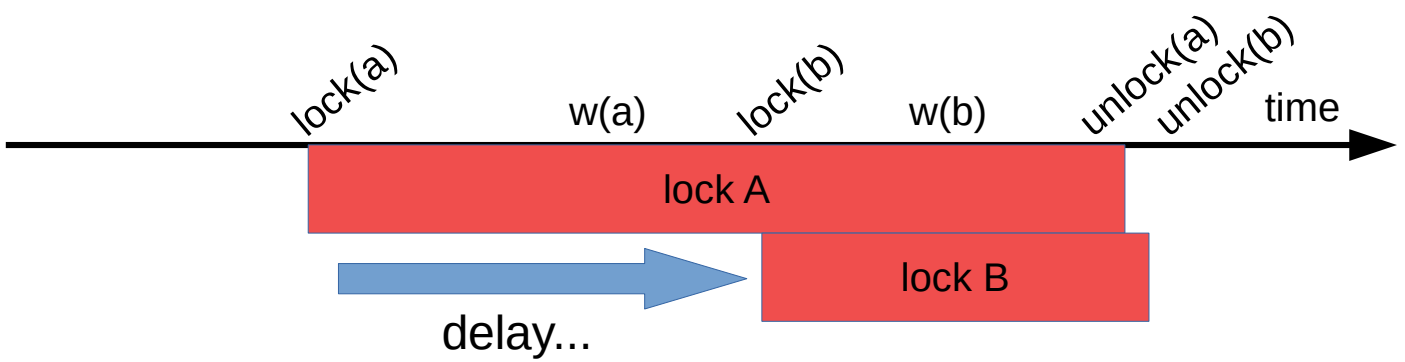
- Why acquire both locks simultaneously?

  (The observer will also lock A and B.)



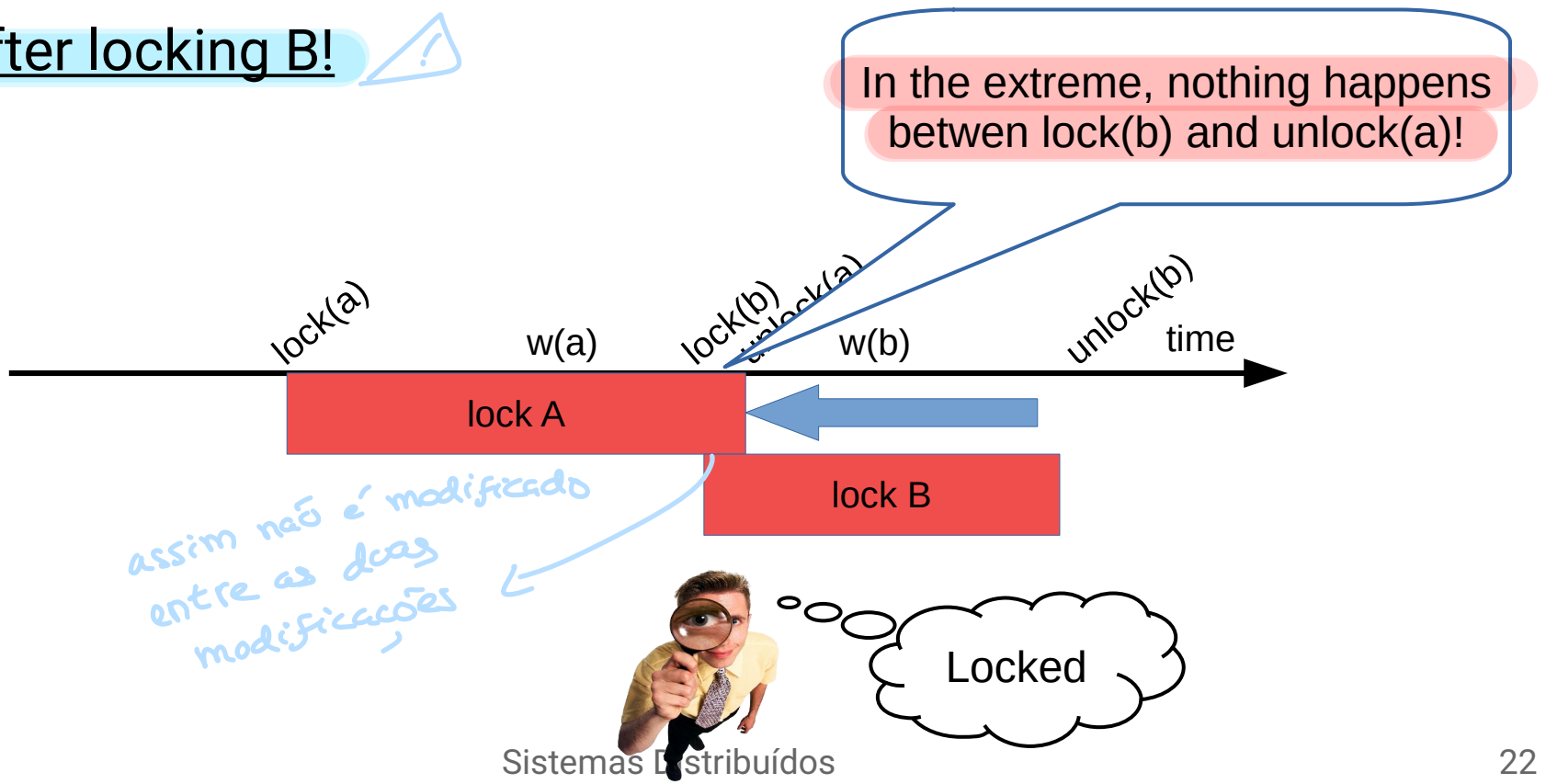
Lock later

- How much can we delay acquiring lock for B?
  - Until needed for modifying item b



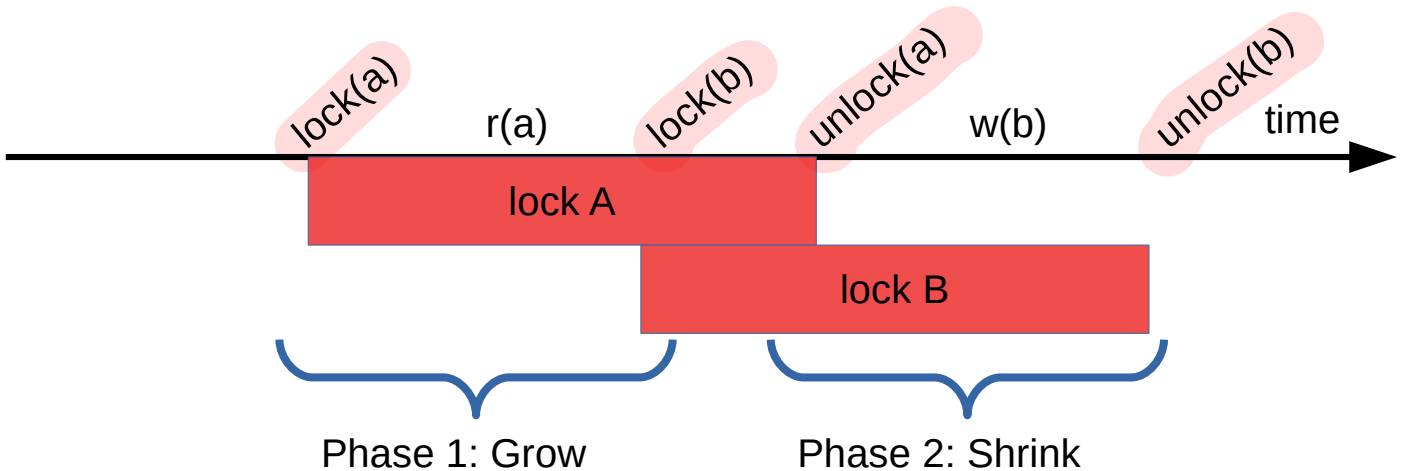
Unlock earlier

- How much can we anticipate releasing lock for A?
  - After modifying item a and…
  - after locking B!



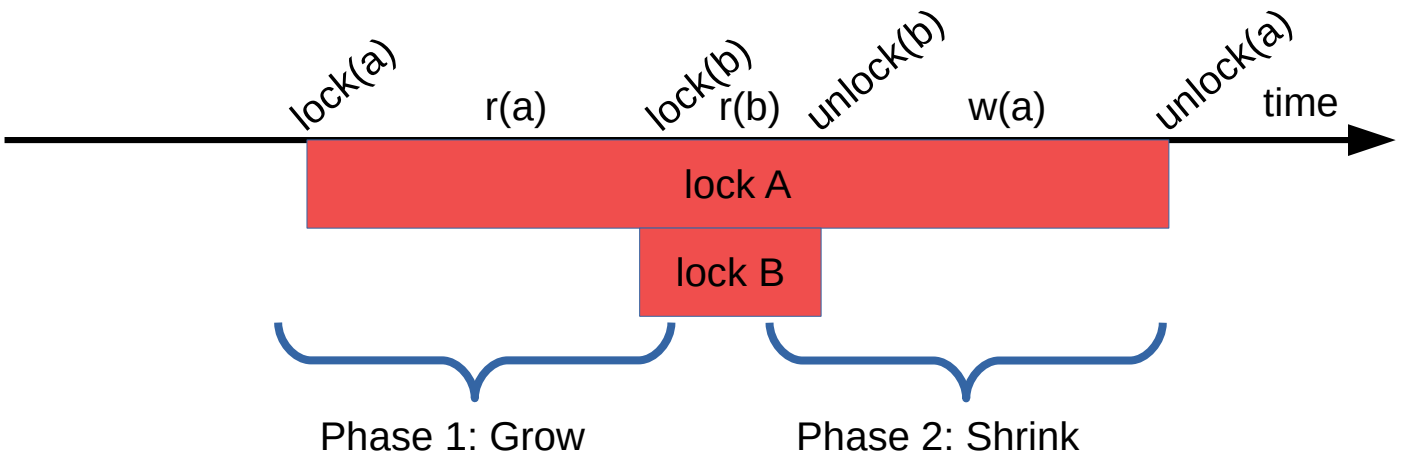
Two phase locking (2PL)

- **Rule 1**: All lock() precede all unlock()
- **Rule 2**: Each data item is read/written within the corresponding lock
  - Equivalent to holding all relevant locks, all the time



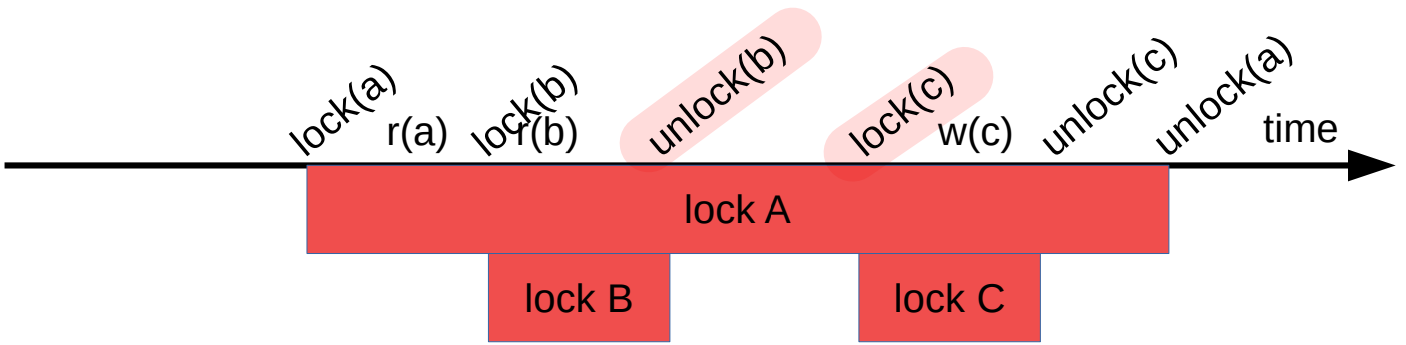
Two phase locking (2PL)

- Another example:

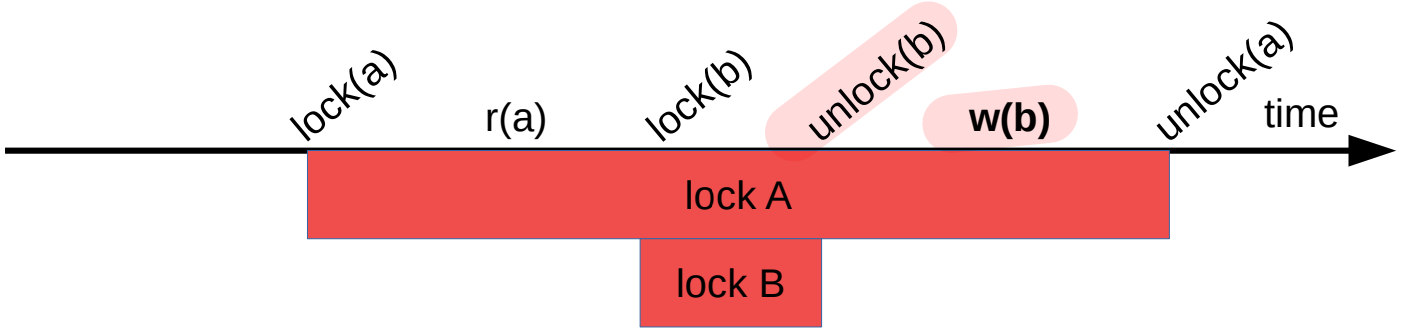


Two phase locking (2PL)

- Fails Rule 1:



- Fails Rule 2:



Two phase locking (2PL)

- void shoot(String sn, String tn) {

```
    Player s = players.get(sn);
    Player t = players.get(tn);
    Stream.of(sn,tn).sorted()
      .forEach(n→players.get(n).l.lock());
    t.life--;
    t.l.unlock();
    s.score++;
    s.l.unlock();
}
```

Phase 1: Grow

Phase 2: Shrink

Collection locking

- What if the collection is not immutable?
  - “drop in the game”
- Add back a global lock to game state...

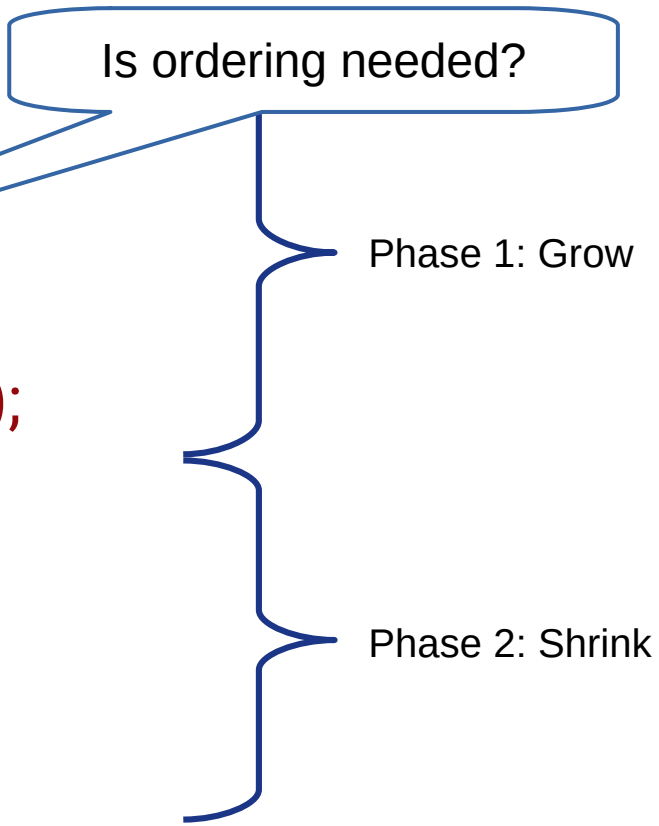
Collection locking

- void shoot(String sn, String tn) {
  l.lock();
  Player s = players.get(sn);
  Player t = players.get(tn);
  Stream.of(sn,tn).sorted()
  .forEach(n→players.get(n).l.lock());
  t.life--;
  s.score++;
  t.l.unlock(); s.l.unlock();
  l.unlock();
  }

Collections with 2PL

- void shoot(String sn, String tn) {

```
    l.lock();
    Player s = players.get(sn);
    Player t = players.get(tn);
    Stream.of(sn,tn).sorted()
      .forEach(n→players.get(n).l.lock());
    l.unlock();
    t.life--;
    t.l.unlock();
    s.score++;
    s.l.unlock();
}
```



Collections with 2PL

- void shoot(String sn, String tn) {

```
    l.lock();
    Player s = players.get(sn);
    Player t = players.get(tn);
    s.l.lock();
    t.l.lock();
    l.unlock();
    t.life--;
    t.l.unlock();
    s.score++;
    s.l.unlock();
}
```

No, if these locks are always acquired in the context of the collection lock!

↳ porque há um lock global

Conclusions

- Minimizing critical sections is key to performance and scale
- Strategies to reduce impact of critical sections:
  - Immutable objects
  - Granular locking
  - Two phase locking
    - Collections
- Avoid deadlocks by using a fixed locking order

Locks vs Variables

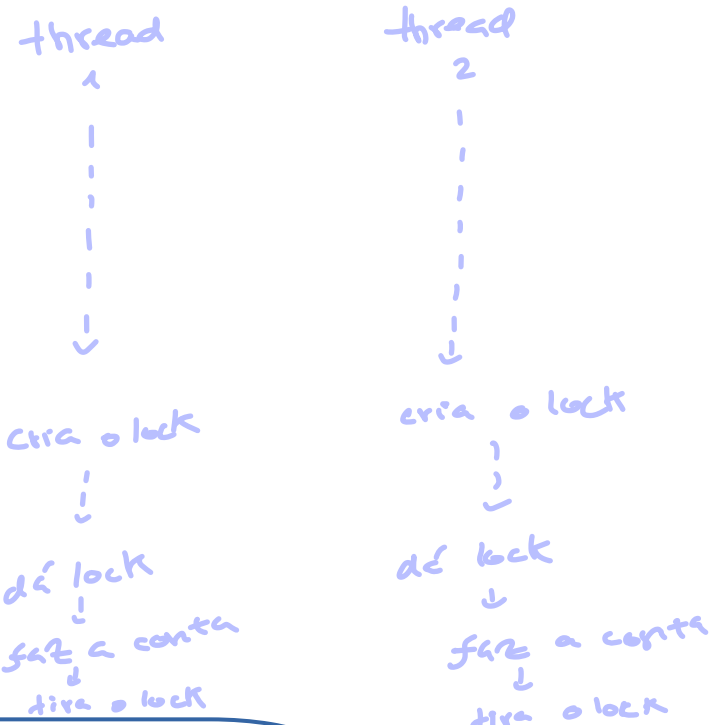
- “Which lock corresponds to each data item?”
- Multiple threads accessing some data item concurrently must have acquired the same lock
- Not automatic / not checked
- It is up to the developer to ensure this!

Pitfall: Automatic variables

- Variables in methods are created on every invocation
- The method might still access shared state:
  - Instance variables
  - Class (static) variables

Wrong

```
class SomeClass {
    int s;
    void doSomething() {
        Lock l = new ReentrantLock();
        l.lock();
        s = s+1;
        l.unlock();
    }
}
```

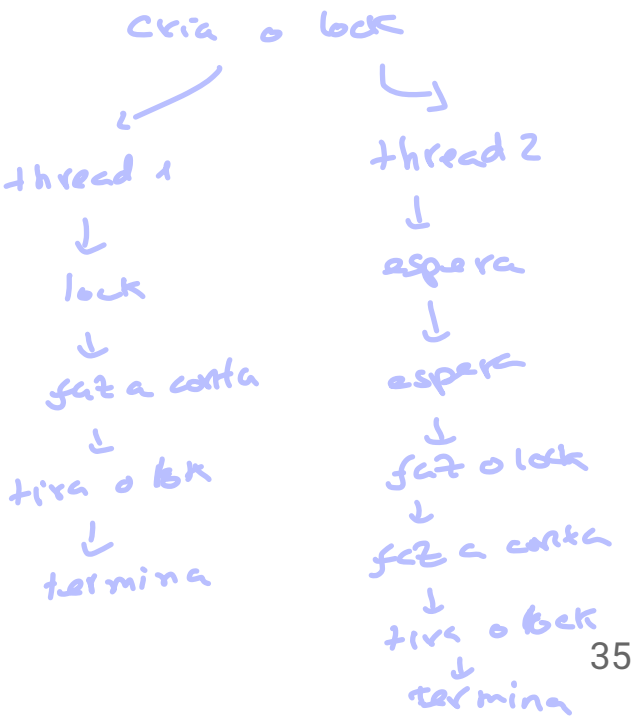


Race: Each thread locks its own lock!

Solution

```
class SomeClass {
    int s;
    Lock l = new ReentrantLock();
    void doSomething() {
        l.lock();
        s = s+1;
        l.unlock();
    }
}
```

Solution: Same scope for shared state and lock

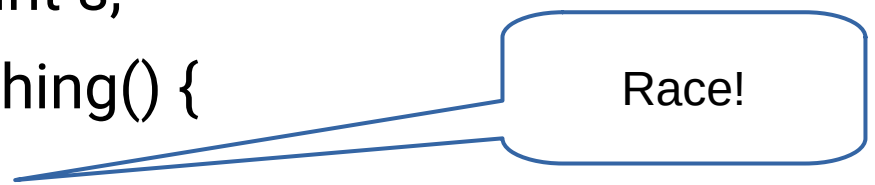


Pitfall: Class/global variables

- Variables marked with “static” in Java are global and (probably) need concurrency control
  - Not if marked “final”
  - Not if the class is used by a single thread

Wrong

```
class SomeClass {
    private static int s;
    void doSomething() {
        s = s+1;
    }
}
```



Still wrong

```
class SomeClass {
    private Lock l = new ReentrantLock();
    private static int s;
    void doSomething() {
        l.lock();
        s = s+1;
        l.unlock();
    }
}
```

There is one lock for each object, but s is shared!

Solution

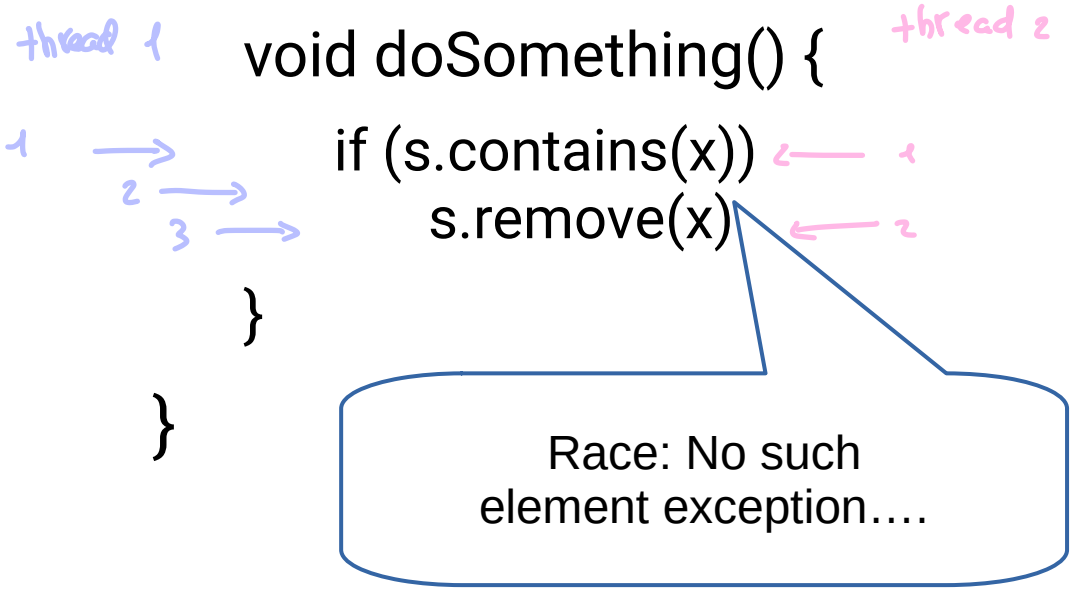
```
class SomeClass {
    private static Lock l = new ReentrantLock();
    private static int s;
    void doSomething() {
        l.lock();
        s = s+1;
        l.unlock();
    }
}
```

Pitfall: Encapsulated locks

- Keep variables and the corresponding lock encapsulated within the same object
- (The default using old-style “synchronized” in Java.)

Wrong

```
class SomeClass {
    SomeState s;
    void doSomething() {
        if (s.contains(x))
            s.remove(x)
    }
}
```




```
class SomeState {
    private Lock l;
    boolean contains(...) {
        l.lock(); … l.unlock();
    }
    void remove(...) {
        l.lock(); … l.unlock();
    }
}
```

Solution

```
class SomeClass {
    private Lock l;
    SomeState s;
    void doSomething() {
        l.lock();
        if (s.contains(x))
            s.remove(x)
        l.unlock();
    }
}
```

```
class SomeState {
    private Lock l;
    boolean contains(...) {
        l.lock(); … l.unlock();
    }
    void remove(...) {
        l.lock(); … l.unlock();
    }
}
```

Now useless...

Better solution

```
class SomeClass {
    private Lock l;
    SomeState s;
    void doSomething() {
        l.lock();
        if (s.contains(x))
            s.remove(x)
        l.unlock();
    }
}
```

```
class SomeState {
    private Lock l;
    boolean contains(...) {
        l.lock(); … l.unlock();
    }
    void remov     .) {
        l.lock()       unlock();
    }
}
```

Rely on locking by the callers. This is done by Java Collections (Lists, ...)

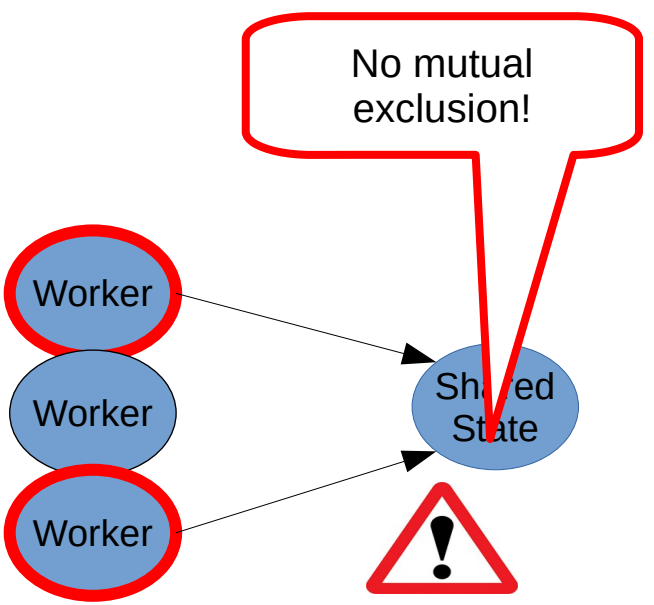
Pitfall: Shared vs thread-local state

- Program state often contains:
  - Local thread state in workers
  - Shared state, used by all threads
- Both are objects, with instance variables

Wrong

```
class Worker
      extends Thread {
    Lock l;
    SharedState s;
    void doSomething() {
        l.lock(); s.doit(); l.unlock();
    }
    public void run() {
        doSomething();
    }
}
```

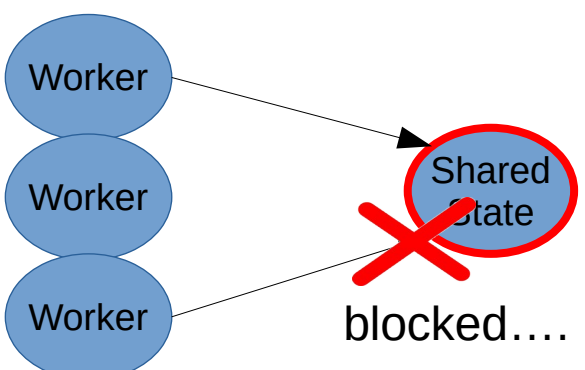
```
class SharedState {
    public void doit() {
        ...
    }
}
```



Solution

```
class Worker
    extends  Thread {
    SharedState s;
    void doit() {
        s.doit();
    }
    public void run() {
        doit();
    }
}
```

```
class SharedState {
    Lock l;
    public void doit() {
        l.lock(); … l.unlock();
    }
}
```

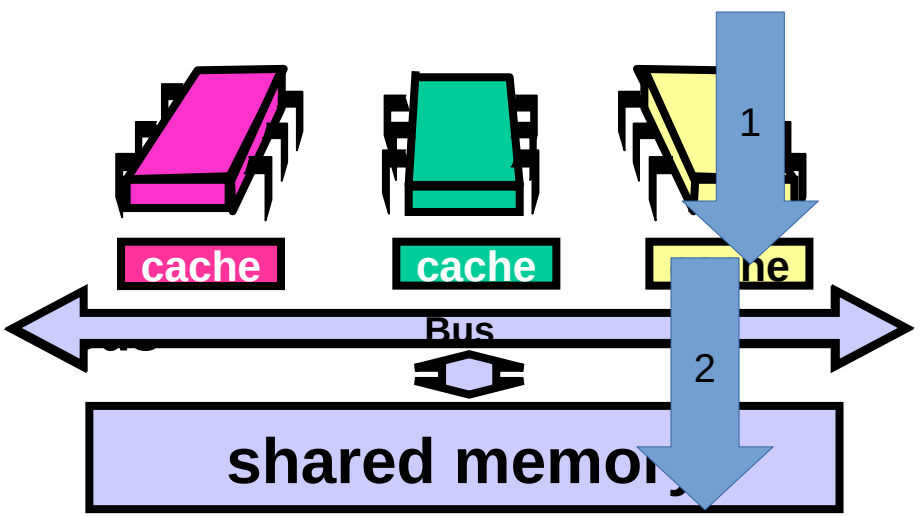


Quiz

- Two variables:
  - int i=0, j=0;
- Writer code:
  - i=1; j=1;
- Reader code:
  - rj=j; ri=i; System.out.println(rj+", "+ri);
- Possible results:
  - a) 0, 0 ✓
  - b) 1, 1 ✓
  - c) 0, 1 ✓
  - d) 1, 0 ✓ → Why!?!?

Memory order

- Steps to write a variable:
  1. Write to cache
  2. Flush cache to memory

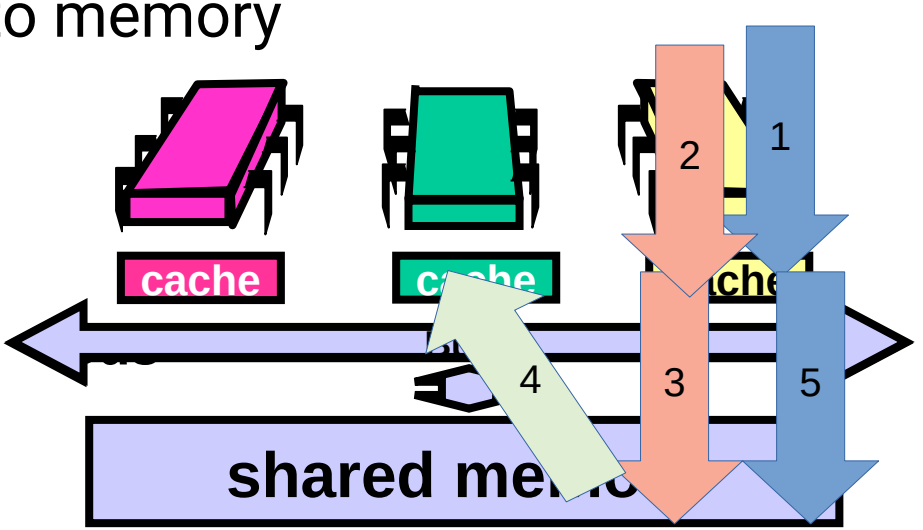


(Image from http://booksite.elsevier.com/9780123705914/?ISBN=9780123705914 . CC By-SA-3.0.)

Memory order

- Possible outcome with two variables:
  1. Write i to cache
  2. Write j to cache
  3. Flush j from cache to memory
  5. Flush i from cache to memory

4. Paradox observed if i,j read here!!



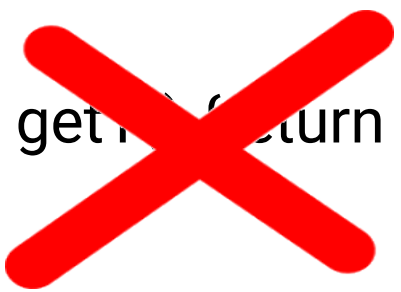
Pitfall: Read races

```
class X {
    private Y y;
    void changeY() {
        l.lock();
        tmp.i = 1;
        y = tmp;
        l.unlock();
    }
    int getY() {return y.i;}
}
```

- Can we omit synchronization in getters?

Pitfall: Read races

```
class X {
    private Y y;
    void changeY() {
        l.lock();
        tmp.i = 1;
        y = tmp;
        l.unlock();
    }
    int get() {return y.i;}
}
```



- Can we omit synchronization in getters?
  - **NO!**
- Can read inconsistent Y fields!
- In this case:
  - reader might not see y.i == 1!!!!

Pitfall: Collections and getters

- Getter methods may return references to shared collections (or other mutable objects)
  - Iterators include references to the original object!

Wrong

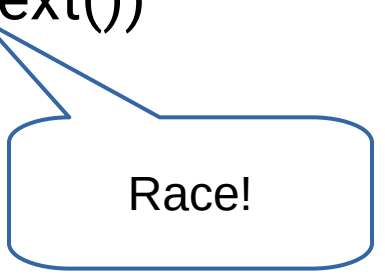
```
class SomeClass {
    private Lock l = new ReentrantLock();
    private List l;
    List getElements() {
        try { l.lock();
            return l;
        } finally { l.unlock(); }
    }
}
```

```
SomeClass s = …;
    List l = l.getElements();
    l.add(…);
```

Wrong

```
class SomeClass {
    private Lock l = new ReentrantLock();
    private List l;
    Iterator getElements() {
        try { l.lock();
            return l.iterator();
        } finally { l.unlock(); }
    }
}
```

```
SomeClass s = …;
    Iterator i = l.getElements();
    while(i.hasNext())
        …
```



Still wrong

```
class SomeClass {
    private Lock l = new ReentrantLock();
    private List l;
    List getElements() {
        try { l.lock();
            return l.clone();
        } finally { l.unlock(); }
    }
}
```

```
SomeClass s = …;
    List l = l.getElements();
    l.add(…);
```

Not adding to the list...

...reconsider encapsulated lock!

Summary

- There is no simple rule to match locks with variables
- Some thinking needed… :-)

Scaling up

- Example:
  - In a distributed database table with millions of records
  - Executing “select sum(x) from … where …” queries
  - Updating records
- Do we use a single lock?
  - Cannot run more than one query at the same time
- Do we use a lock for each line?
  - Way too many individual locks!

Readers-Writers locks

- Strict mutual exclusion with locks is too conservative:
  - More than one reader would not be a problem
  - A writer must exclude all others (readers and writers)
- Different methods for readers and writers:

```
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- More costly than a simple lock

Image from: https://www.deccanherald.com/content/354607/dangerous-rail-crossings.html

Readers-Writers locks

- R/W locks also known as shared locks in database management systems:
  - Readers lock <=> Shared mode
  - Writers lock <=> eXclusive mode
- Behavior described by a compatibility matrix:

Mode	R/S	W/X
R/S	Yes	No
W/X	No	No

-> quando um está a escrever
↓
ninguém pode escrever também
+
ninguém pode ler

Readers-Writers example

```
int v;
Lock l = new ReentrantLock();
void doSomething() {
    l.lock();
    v++;
    l.unlock();
}
int getV() {
    try { l.lock();
    return v;
    } finally { l.unlock(); }
}
```

```
int v;
ReadWriteLock l = new ReentrantReadWriteLock();
void doSomething() {
    l.writeLock().lock();
    v++;
    l.writeLock().unlock();
}
int getV() {
    try { l.readLock().lock();
    return v;
    } finally { l.readLock().unlock(); }
}
```

… not worth it for such simple operations!

Revisiting collections with 2PL+RW lock

- void shoot(String sn, String tn) {
  l.**readLock()**.lock();
  Player s = players.get(sn);
  Player t = players.get(tn);
  Stream.of(sn,tn).sorted()
  .forEach(n→players.get(n).l.lock());
  l.**readLock()**.unlock();
  t.life--;
  t.l.unlock();
  s.score++;
  s.l.unlock();
  }

Allow multiple threads to acquire locks concurrently

Sorting is needed again

Readers-Writers fairness

- Priority to readers
  - Allow more readers in, even if a writer is waiting
  - The writer may starve…
- Priority to writers
  - Do not allow more readers in, if a writer is waiting
  - Less concurrency among readers

Lock managers

- Individual locks inefficient for huge collections of objects
  - A lock object uses memory even when not in use
- A lock manager provides locks on demand:

```
interface LockManager {
    void lock(Object name);

    void unlock(Object name);
}
```

lookup lock l for “name” in map
if it doesn’t exist:
    create it and add to map
l.lock()!

lookup lock for “name”
l.unlock()
if nobody else is using it:
    remove it from map

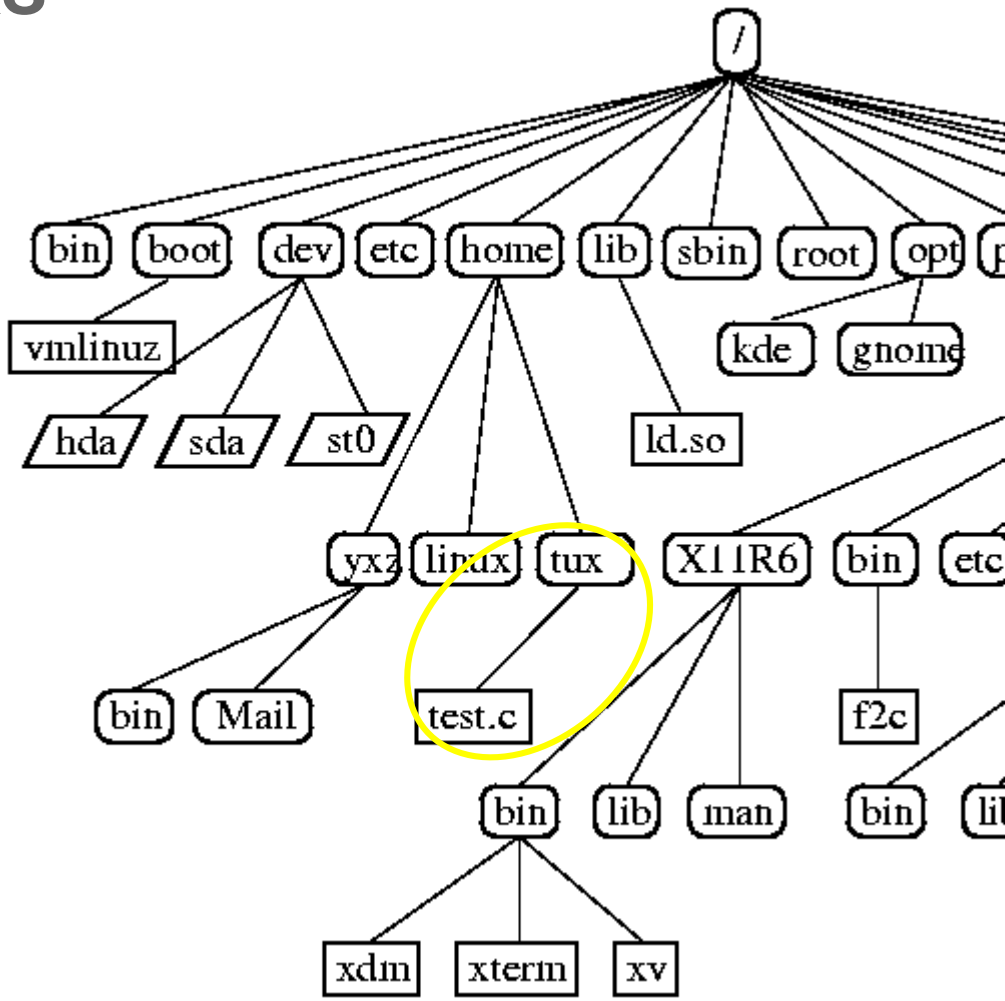
Lock managers

- Usually provides shared/exclusive semantics:

```
enum Mode { SHARED, EXCLUSIVE };
interface LockManager {
    void lock(Object name, Mode mode);
    void unlock(Object name);
}
```

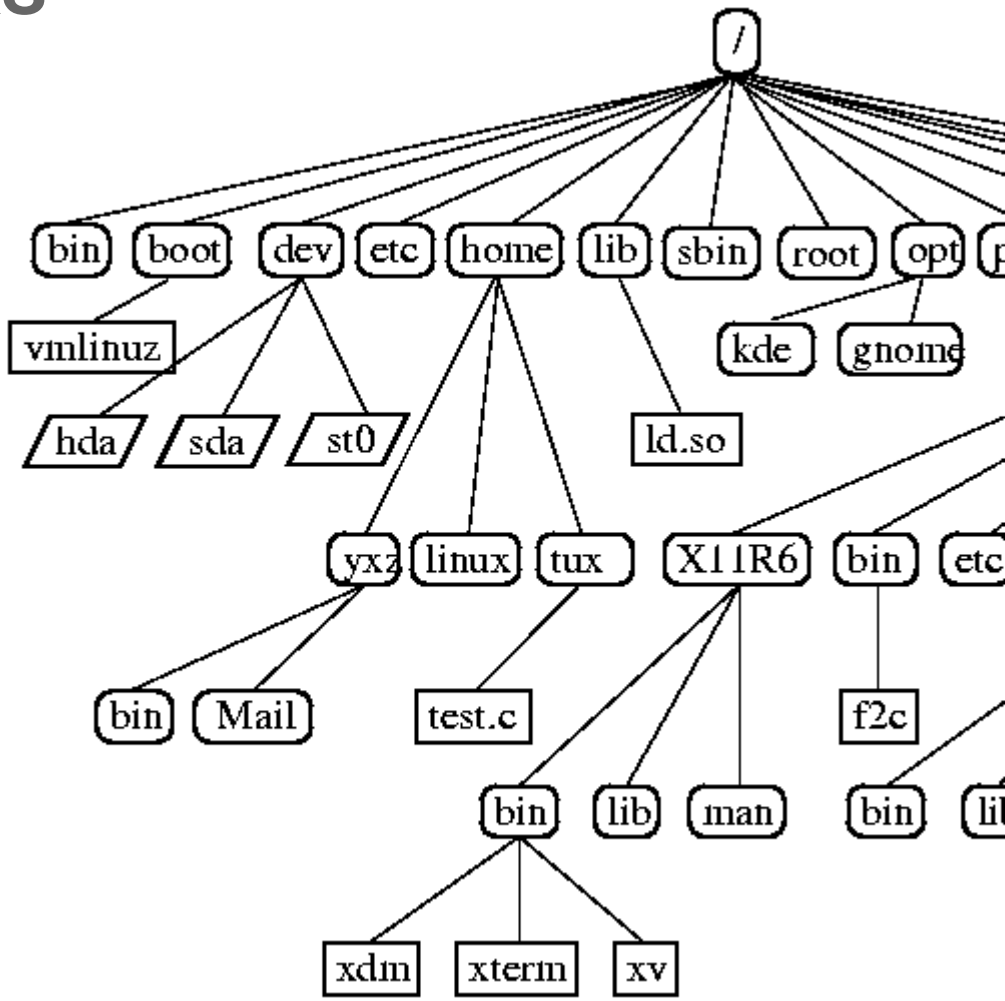
Multiple granularity locks

- Motivation:
  - Locking /home/tux/*
  - Assume large number of files
- Inefficient even with a lock manager
- Idea: Take advantage of hierarchical namespace and lock folders



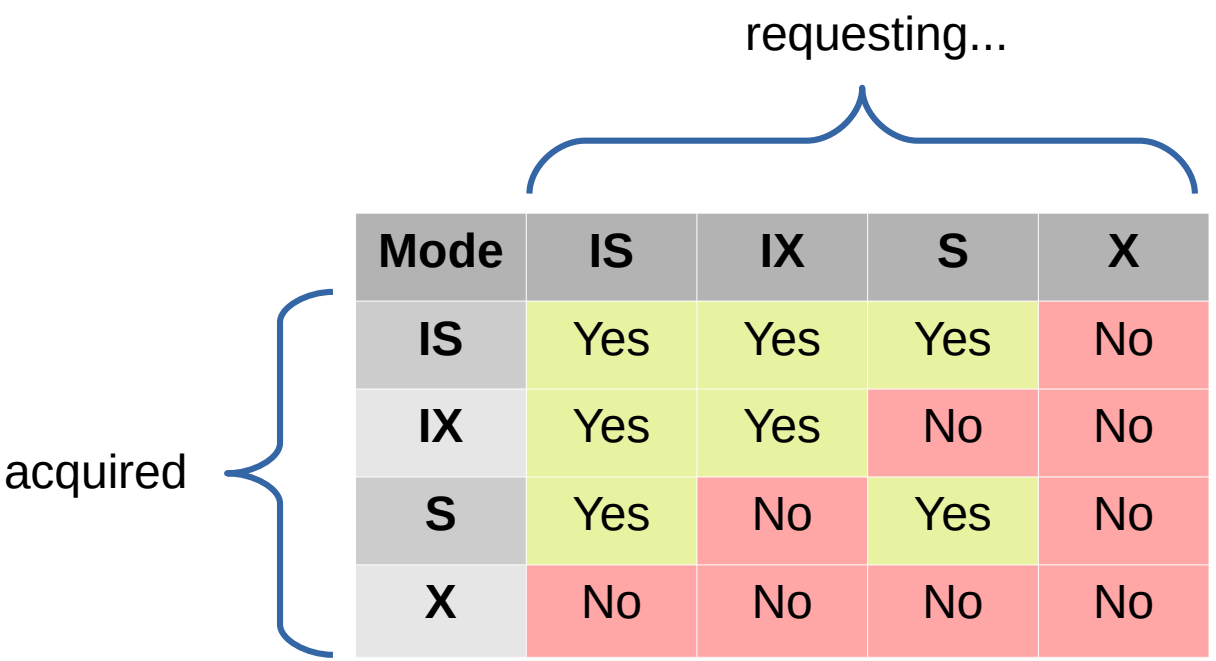
Multiple granularity locks

- Protocol:
  - “intention” locks on containers
  - “actual” locks on the target
- Intention locks conflict with actual locks, not with other intention locks
- Combine with (S)hared and e(X)clusive semantics



Compatibility matrix

- An MGL is defined by a compatibility matrix:



Mode	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

Multiple granularity locks

- Shared lock /home/tux
- Shared lock on /home/tux/test.c
- Exclusive lock on /boot
  - Shared lock on /boot/vmlinuz
    - IS on /boot/ conflicts with X
  - Exclusive lock on /home
    - X on /home conflicts with IS