Múltiplas Variáveis de Condição por Monitor

Sistemas Distribuídos

Exclusão Mútua com monitor

```
synchronized void metodo(){
...
}
```

Exclusão Mútua explícita

```
import java.util.concurrent.locks.ReentrantLock;

lock = new ReentrantLock();

void metodo(){
  this.lock.lock();
    ...
  this.lock.unlock();
}
```

- Variáveis de condição:
  - suspensão/retoma de execução dentro de zona crítica
  - usadas em conjunto com o ReentrantLock():
    - java.util.concurrent.locks.ReentrantLock
      – métodos lock(), unlock(), **newCondition()**
    - java.util.concurrent.locks.Condition
      – métodos relevantes: **await(), signal(), signalAll()**

Variáveis de condição com monitor

```
synchronized void metodo(){
   ...
   while(…){
      this.wait();
   }
   this.notify();
}
```

Variáveis de condição explícitas

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

void metodo(){
  this.lock.lock();
  ...
  while(…){
     this.condition.await();
  }
  this.condition.signal();
  this.lock.unlock();
}
```

Variáveis de condição com monitor

```
synchronized void metodo(){
    ...
    while(…){
       this.wait();
    }
    this.notify();
}
```

Variáveis de condição explícitas

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();

void metodo(){
   this.lock.lock();
   ...
   while(…){
      this.condition.await();
   }
   this.condition.signal();
   this.lock.unlock();
}
```

Exercícios

1) Reimplemente a classe **BoundedBuffer**, de modo de modo a evitar acordar threads desnecessariamente, distinguindo as situações de bloqueio pelo array estar vazio e cheio.

Exercícios

2) Implemente uma classe **Warehouse** para permitir a gestão de um armazém acedido concorrentemente. Deverão ser disponibilizados os métodos:

- **supply(String item, int quantity)** – abastecer o armazém com uma dada quantidade de um item;
- **consume(String[] items)** – obter do armazém um conjunto de itens, bloqueando enquanto tal não for possível.

Warehouse

private HashMap<String, Item> stock;

Warehouse()

void supply(String item, int quantity)

void consume(String[] items)

aumenta a quantidade do item

consome 1 unidade de cada item passado no array (bloqueia caso não haja unidades disponíveis)

Item

ReentrantLock lock;
Condition isEmpty;
int quantity;

Item()

void supply(int quantity)

void consume()

Producer ← implements Runnable → Consumer

Warehouse wh;

run()

```
this.wh.supply("item1",1)
sleep(3s)
this.wh.supply("item2",1)
sleep(3s)
this.wh.supply("item3",1)
```

Warehouse wh;

run()

```
this.wh.consume(["item1","item2","item3"])
```

Main

//criar objecto Warehouse com 3 items (com 0 unidades)
//criar e iniciar Producer e Consumer

Exercícios

3) Implemente a classe **RWLock** com os métodos **readLock(), readUnlock(), writeLock()** e **writeUnlock()** de modo a permitir o acesso simultâneo de múltiplos leitores a uma dada região crítica, ou em alternativa, o acesso de um único escritor.

- Usar **ReentrantLock** e respectivas variáveis de condição.
- Evitar o fenómeno de starvation.

Exemplo (Exercício 3):

Reader 1

```
rwlock.readLock()
//secção crítica
rwlock.readUnlock()
```

Reader 2

```
rwlock.readLock()
//secção crítica
rwlock.readUnlock()
```

Writer 1

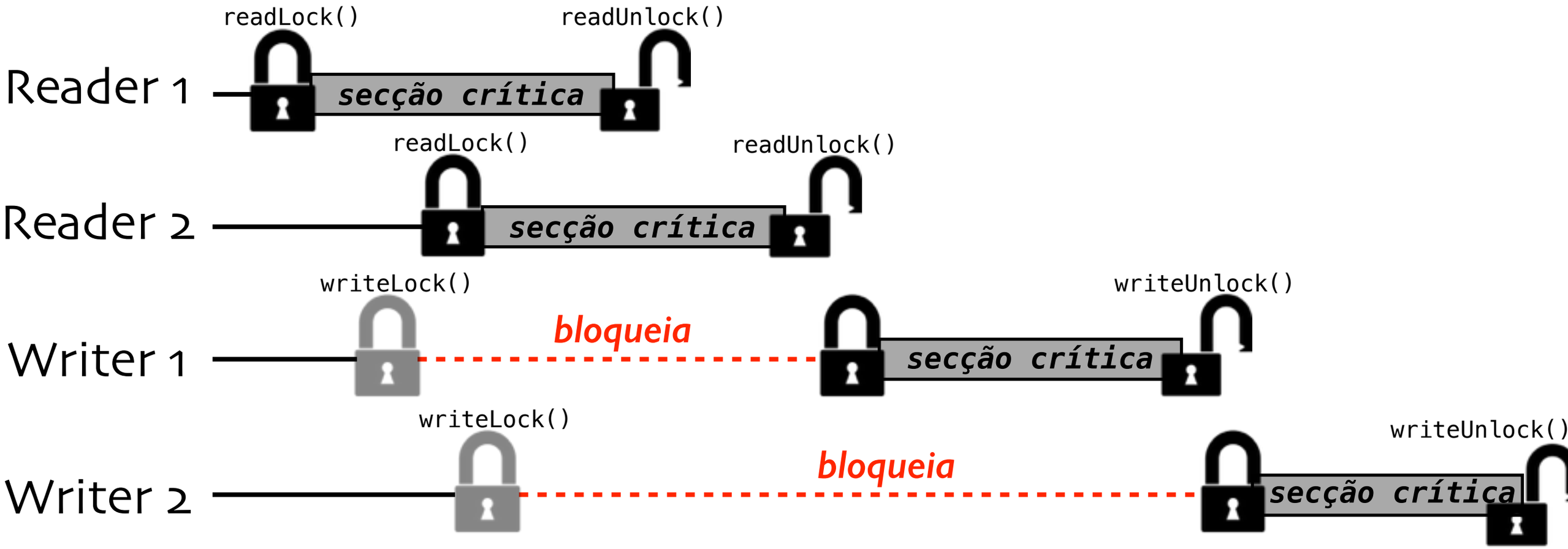
```
rwlock.writeLock()
//secção crítica
rwlock.writeUnlock()
```

Writer 2

```
rwlock.writeLock()
//secção crítica
rwlock.writeUnlock()
```

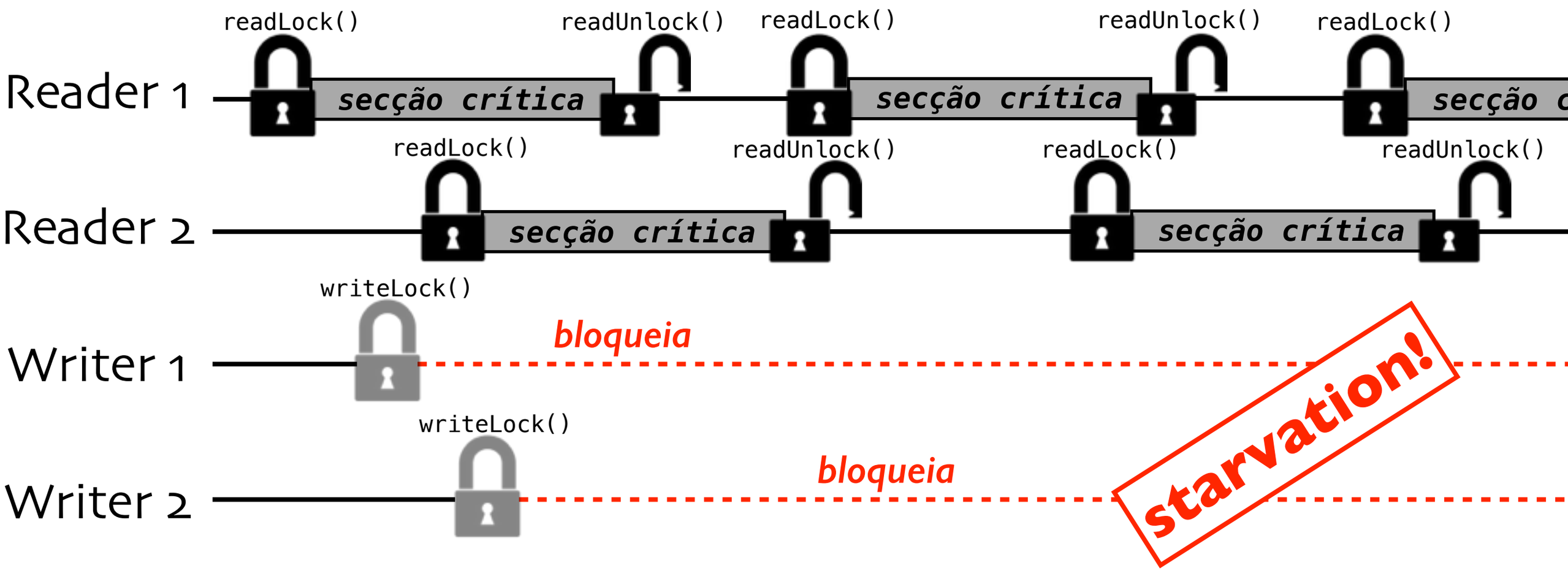
Exemplo (Exercício 3):

Reader 1	Reader 2	Writer 1	Writer 2
`rwlock.readLock()`	`rwlock.readLock()`	`rwlock.writeLock()`	`rwlock.writeLock()`
`//secção crítica`	`//secção crítica`	`//secção crítica`	`//secção crítica`
`rwlock.readUnlock()`	`rwlock.readUnlock()`	`rwlock.writeUnlock()`	`rwlock.writeUnlock()`



Exemplo (Exercício 3):

Reader 1	Reader 2	Writer 1	Writer 2
`rwlock.readLock()`	`rwlock.readLock()`	`rwlock.writeLock()`	`rwlock.writeLock()`
`//secção crítica`	`//secção crítica`	`//secção crítica`	`//secção crítica`
`rwlock.readUnlock()`	`rwlock.readUnlock()`	`rwlock.writeUnlock()`	`rwlock.writeUnlock()`



3) Implemente a classe **RWLock** com os métodos **readLock(), readUnlock(), writeLock()** e **writeUnlock()** de modo a permitir o acesso simultâneo de múltiplos leitores a uma dada região crítica, ou em alternativa, o acesso de um único escritor.

- Usar **ReentrantLock** e respectivas variáveis de condição.
- Evitar o fenómeno de starvation.

implements Runnable

Main

```
//criar objecto RWLock
//criar e iniciar 15 Readers
e 15 Writers
```

RWLock

```
(…)
void readLock()
void readUnlock()
void writeLock()
void writeUnlock()
```

Reader

```
RWlock rwlock;
run()
    rwlock.readLock();
    sleep(1s);
    rwlock.readUnlock();
```

Writer

```
RWlock rwlock;
run()
    rwlock.writeLock();
    sleep(1s);
    rwlock.writeUnlock();
```