Sistemas Distribuídos

José Orlando Pereira

Departamento de Informática
Universidade do Minho

2018/2019

Mutex with Peterson's/...

- Does it really work?

```
try {
    l.lock();        while(flag[...]) { }
    c=c+1;
} finally {
    l.unlock();      flag[i] = false;
}
```

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:

running concurrently!

a) 0, 0 ✓

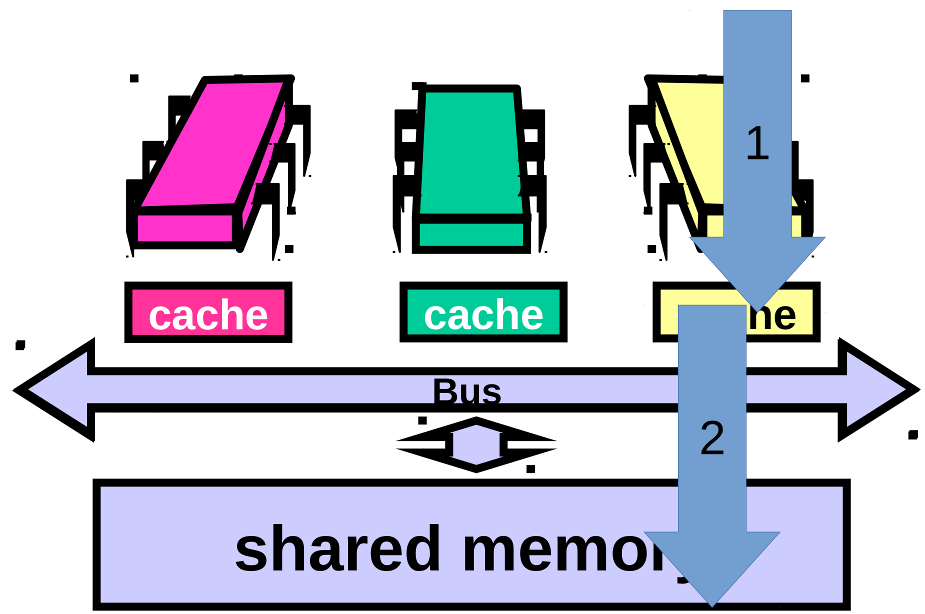
b) 1, 1 ✓

c) 0, 1 ✓

d) 1, 0 ✓ → Why!?!?

Memory order

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

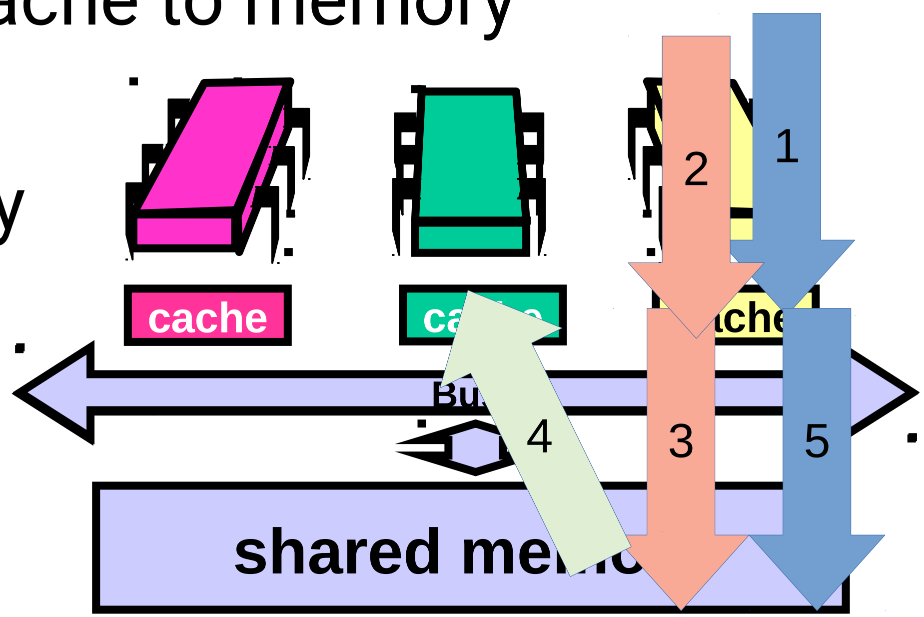


(Image from http://booksite.elsevier.com/9780123705914/?ISBN=9780123705914 . CC By-SA-3.0.)

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!!



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

Consequence

```
try {
    l.lock();        while(flag[...]) { }
    c=c+1;
} finally {
    l.unlock();      flag[i] = false;
}
```

- Initially c=10
- One thread:
  - read c
  - write c = 11
  - write flag[i] = false
- Other thread:
  - read flag[...] = false
  - read c = 10!!!!

Solution

- Declare: volatile int j;
- Reading from a volatile j waits for all writes preceding the observed value on j to be also visible
  - Writer code:
    - i=1; j=1;
  - Reader code:
    - rj=j; ri=i; System.out.println(rj+", "+ri);

waits for write i=1
to be flushed

Volatile

- Volatile variables impact performance:
  - Use only when needed!
- Also have the same effect:
  - using monitors (e.g. synchronized)
  - using java.util.concurrent.*

Corollary

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

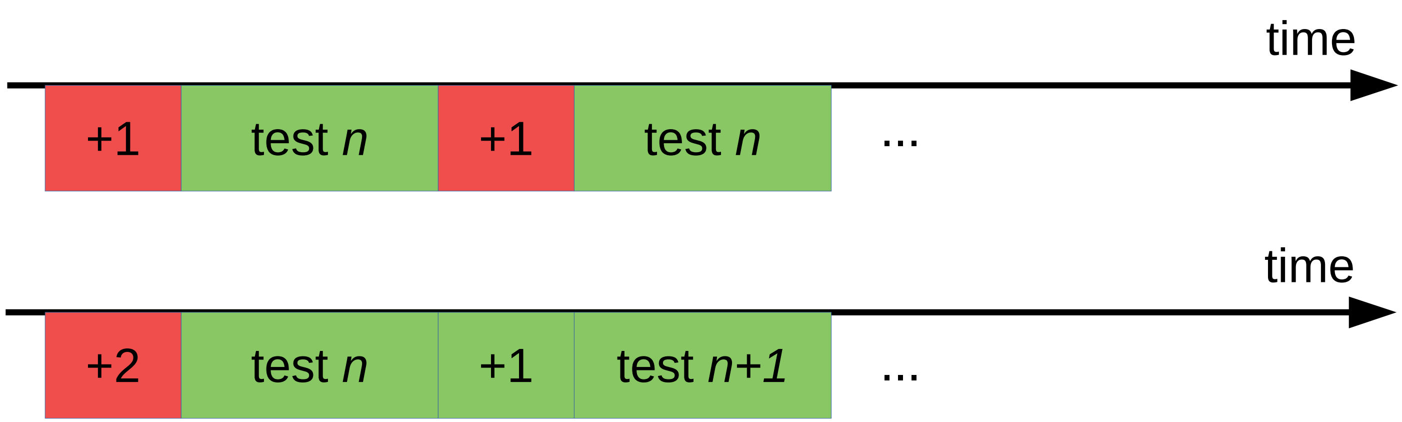
- Can we omit synchronized on getters?
- Can read inconsistent Y fields!
- In this case:
  - reader might not see y.i == 1!!!!

Mutual exclusion

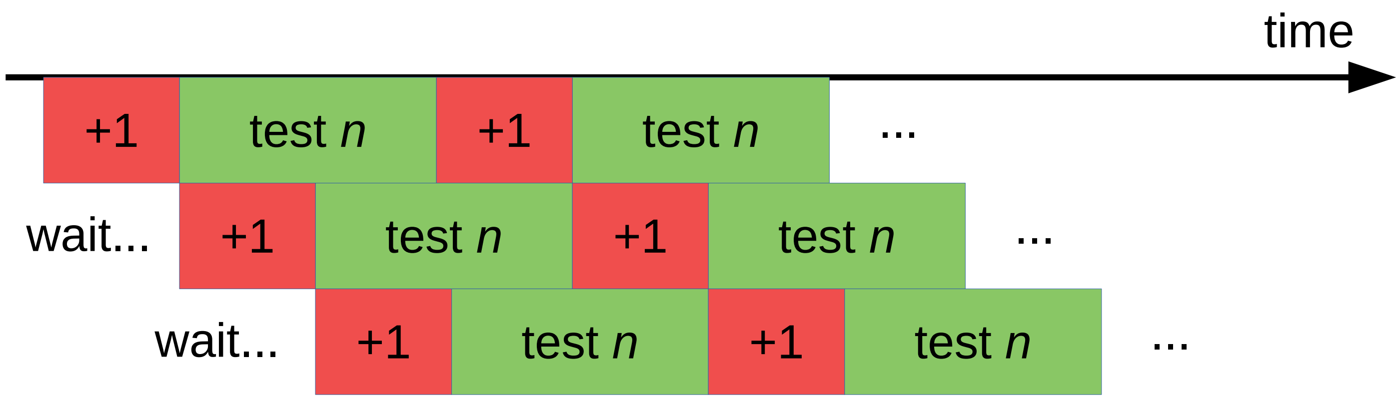
- Mutual exclusion with busy waiting / spinning:
  - Low latency when lock becomes available
  - Consumes CPU time / power when busy
  - Good for parallel programming
    - threads ~= cores, small critical sections
- Mutual exclusion with blocking:
  - High latency when lock becomes available (waits for kernel to schedule it)
  - No CPU time / power when blocked
  - Good for distributed programming
    - threads >> cores, large critical sections

Speedup: Example

- 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*

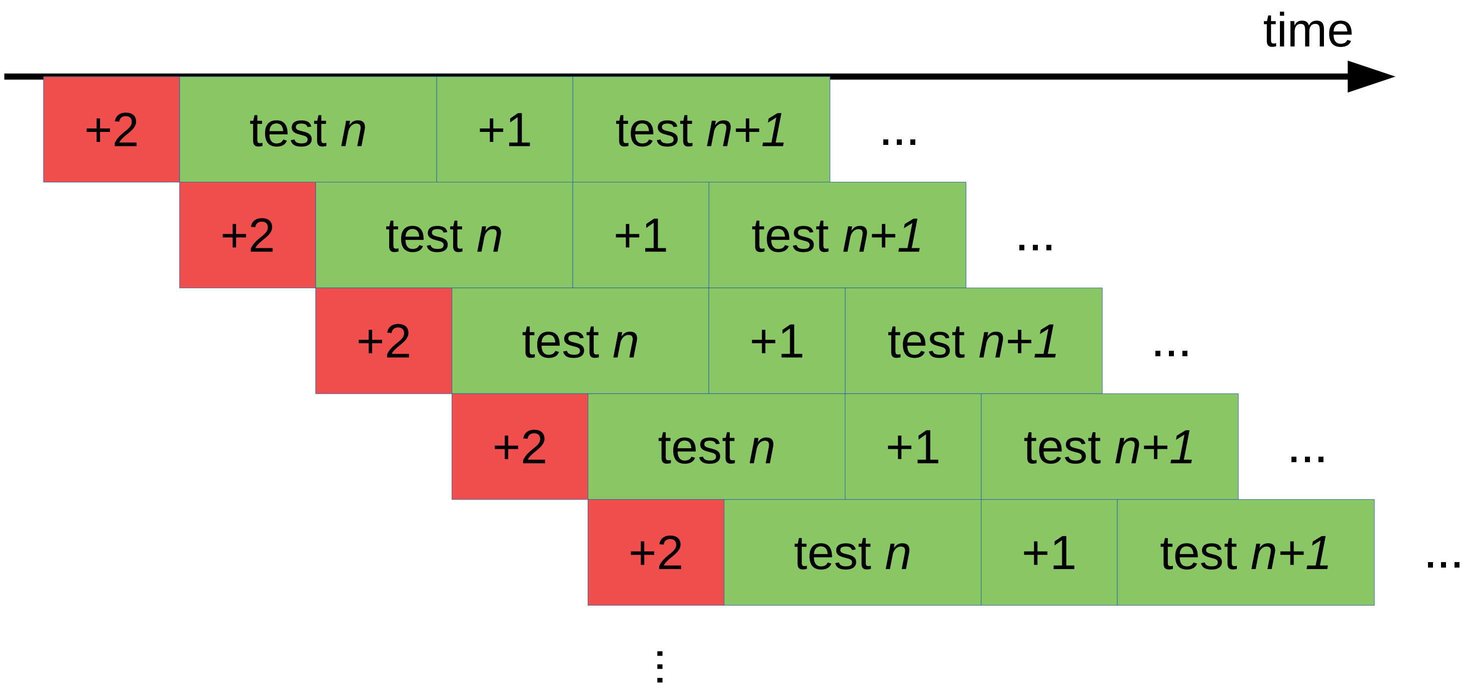


Speedup: Example



- With more than 3 threads, one is always blocked waiting for mutex

Speedup: Example



- Can now achieve greater speedup from concurrent processing!

Conclusions

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