Sistemas Distribuídos

José Orlando Pereira

Departamento de Informática
Universidade do Minho

Mutex with Peterson’s/...

- Does it really work?

```
try {
    l.lock();
    c=c+1;
} finally {
    l.unlock();
}
```

while(flag[...]) { }

flag[i] = false;

Quiz

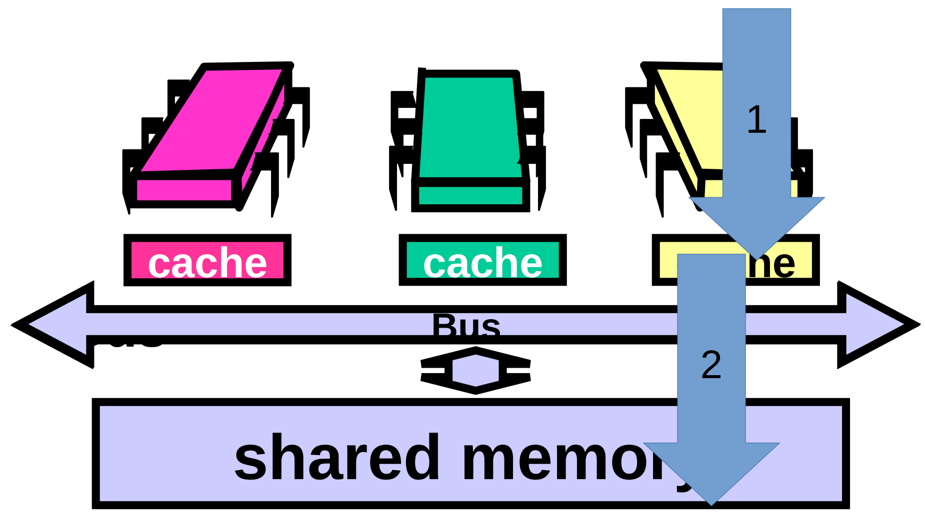
- Two variables:
  - int i=0, j=0;
- Writer code:
  - i=1; j=1;
- Reader code:
  - rj=j; ri=i; System.out.println(rj+", "+ri);
- Possible results:
  a) 0, 0
  b) 1, 1
  c) 0, 1
  d) 1, 0

Why!?!?

running concurrently!

Memory order

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

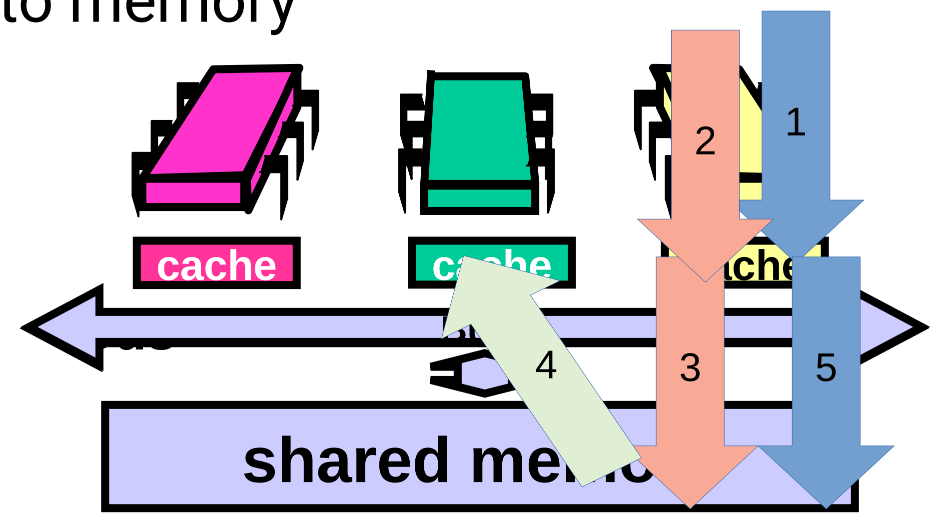


(Image from http://booksite.elsevier.com/9780123705914/?ISBN=9780123705914 . CC By-SA-3.0.)

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();
    c=c+1;
} finally {
    l.unlock();
}
```

```
while(flag[...]) { }
```

```
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: Memory barriers

- Declare: volatile int j; or AtomicInteger 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 visible

Should I use it?

- Synchronization operations also act as memory barriers:
  - java.util.concurrent.* primitives
  - synchronized keyword
- Volatile variables also impact performance:
  - A volatile access takes as much time as using a lock
  - Therefore… **use the lock**!

Corollary

```
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?

Corollary

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

The real world

- Is j.u.c.ReentrantLock implemented with Bakery or something similar?
  - **No!**
  - Needs *O(n)* space for each lock (with *n* threads)
  - Consumes CPU time / power when busy
- These algorithms are a good for:
  - Understanding concurrent programming and races
  - Preparing for distributed algorithms

The real world

- j.u.c.ReentrantLock is implemented with atomic operations and scheduling:
  - Atomic operations (testAndSet, compareAndSwap, …) are used to quickly check if mutex is available
    - For more on atomic operations, see Chapter 7 of TAOMP (not part of the program!)
  - The thread is suspended by the OS when waiting for longer periods
    - Remember scheduling from Operating Systems
    - man futex on Linux

j.u.c Locks vs Monitors

```
class C {
    private int i;

    synchronized public void m() {

        i++;

    }
}
```

There is a hidden “lock” in each object used by “synchronized”

```
class C {
    private int i;
    private Lock l =
        new ReentrantLock();

    public void m() {
        try { l.lock();
        i++;
        } finally { l.unlock(); }
    }
}
```

Equivalent code
(aproximately...)

j.u.c. Locks vs Monitors

- Main differences, for now:
  - Synchronized blocks are nested in LIFO order
    vs.
    j.u.c. Locks can be unlocked in any order
    - To take advantage of two phase locking
  - Threads waiting for a synchronized block enter in any order
    vs.
    j.u.c. Lock can be configured for threads waiting to enter in a fair (not FIFO!) order
- More later...

j.u.c Conditions vs Monitors

- class C {

There is a hidden “condition” in each object used by “wait()/notifyAll()”

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

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

- class C {

```
    private Lock l =
            new ReentrantLock();
    private Condition c =
            l.newCondition();
    public void m1() {
            try { l.lock();
                while(…) c.await()
            } finally { l.unlock(); }
    }
    public void m2() {
            try { l.lock();
                c.signalAll();
            } 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 obtained from a ReentrantLock wakeup in FIFO order (but may not aquire lock in FIFO order...)