

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;

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!**

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;
```

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

```
synchronized public void m() {
```

```
    i++;
```

```
}
```

```
}
```

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

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

```
}
```

```
}
```

Equivalent code
(approximately...)

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

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 acquire lock in FIFO order...)