

Mutual Exclusion

Companion slides for
The Art of Multiprocessor
Programming
by Maurice Herlihy & Nir Shavit

(Abridged version. Original at <http://booksite.elsevier.com/9780123705914/?ISBN=9780123705914>)

Warning

- You will never use these protocols
 - Get over it
- You are advised to understand them
 - The same issues show up everywhere
 - Except hidden and more complex

Locks (Mutual Exclusion)

```
public interface Lock {  
    public void lock();  
    public void unlock();  
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

```
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

release lock

```
}
```

Deadlock-Free



- If some thread calls `lock()`
 - And never returns
 - Then other threads must complete `lock()` and `unlock()` calls infinitely often
- System as a whole makes progress
 - Even if individuals starve

Starvation-Free



- If some thread calls `lock()`
 - It will eventually return
- Individual threads make progress

Two-Thread vs n -Thread Solutions

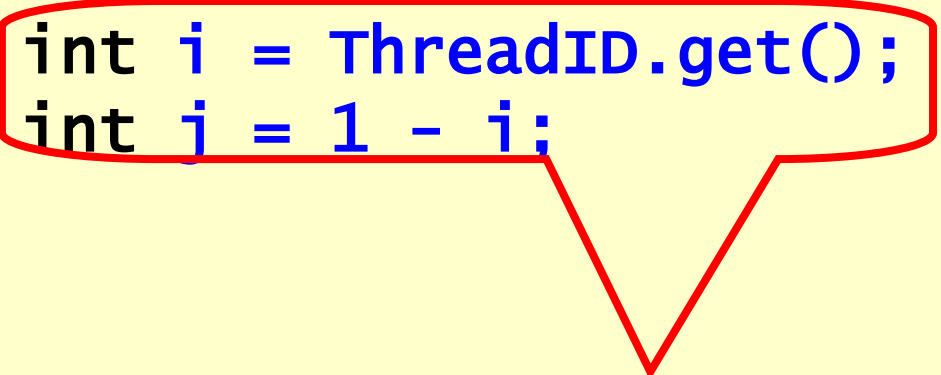
- Two-thread solutions first
 - Illustrate most basic ideas
 - Fits on one slide
- Then n -Thread solutions

Two-Thread Conventions

```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```

Two-Thread Conventions

```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```



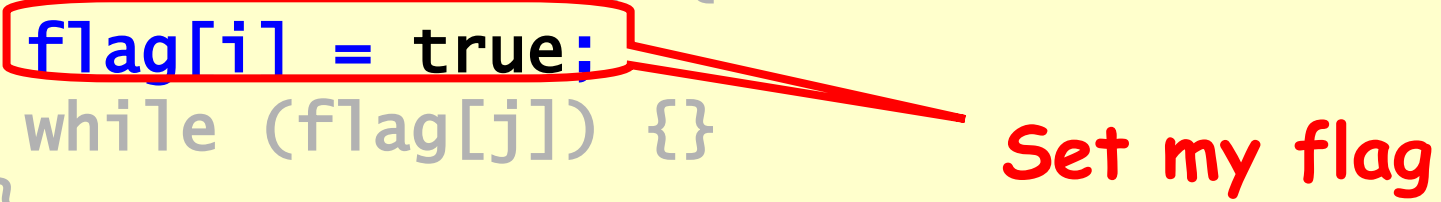
Henceforth: **i** is current thread, **j** is other thread

LockOne

```
class LockOne implements Lock {  
    private boolean[] flag =  
                                new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

LockOne

```
class LockOne implements Lock {  
    private boolean[] flag =  
                                new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```



Set my flag

LockOne

```
class LockOne implements Lock {  
    private boolean[] flag =  
        new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

Set my flag

Wait for other
flag to go false

Deadlock Freedom

- LockOne Fails deadlock-freedom
 - Concurrent execution can deadlock

```
flag[i] = true;    flag[j] = true;
while (flag[j]){   while (flag[i]){
```

- Sequential executions OK

LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

LockTwo

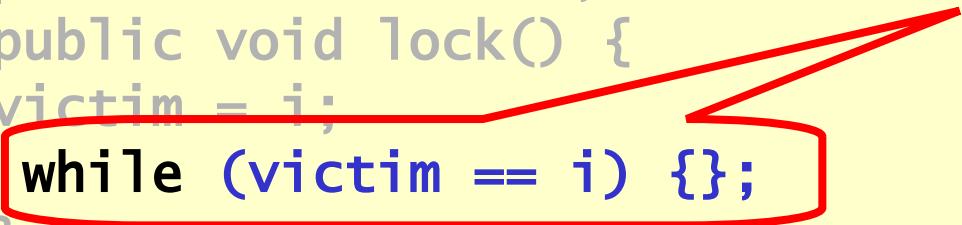
```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

Let other go
first

LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

Wait for permission

A red callout bubble with a tail pointing to the `while (victim == i) {};` line in the code. The bubble contains the text "Wait for permission" in red.

LockTwo

```
public class Lock2 implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
    public void unlock() {}  
}
```

Nothing to do



LockTwo Claims

- Satisfies mutual exclusion
 - If thread i in CS
 - Then $victim == j$
 - Cannot be both 0 and 1
- Not deadlock free
 - Sequential execution deadlocks
 - Concurrent execution does not

```
public void LockTwo() {  
    victim = i;  
    while (victim == i) {}  
}
```

Peterson's Algorithm

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
public void unlock() {  
    flag[i] = false;  
}
```

Peterson's Algorithm

Announce I'm
interested

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {}  
}  
public void unlock() {  
    flag[i] = false;  
}
```

Peterson's Algorithm

```
public void lock() {
```

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {};
```

```
}
```

```
public void unlock() {
```

```
    flag[i] = false;
```

```
}
```

Announce I'm
interested

Defer to other

Peterson's Algorithm

```
public void lock() {
```

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {}:
```

```
}
```

```
public void unlock() {
```

```
    flag[i] = false;
```

```
}
```

Announce I'm
interested

Defer to
other

Wait while other
interested & I'm
the victim

Peterson's Algorithm

```
public void lock() {
```

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {}:
```

```
}
```

```
public void unlock() {
```

```
    flag[i] = false;
```

```
}
```

Announce I'm
interested

Defer to
other

Wait while other
interested & I'm
the victim

No longer
interested

Mutual Exclusion

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};
```

- If thread 0 in critical section,
 - flag[0]=true,
 - victim = 1
- If thread 1 in critical section,
 - flag[1]=true,
 - victim = 0

Cannot both be true

Deadlock Free

```
public void lock() {  
    ...  
    while (flag[j] && victim == i) {};
```

- Thread blocked
 - only at while loop
 - only if it is the victim
- One or the other must not be the victim

Starvation Free

- Thread i blocked only if j repeatedly re-enters so that

`flag[j] == true` and
`victim == i`

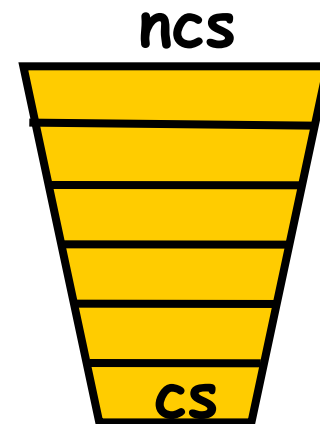
- When j re-enters
 - it sets `victim` to j .
 - So i gets in

```
public void lock() {  
    flag[i] = true;  
    victim  = i;  
    while (flag[j] && victim == i) {};  
}  
  
public void unlock() {  
    flag[i] = false;  
}
```

The Filter Algorithm for n Threads

There are $n-1$ “waiting rooms” called levels

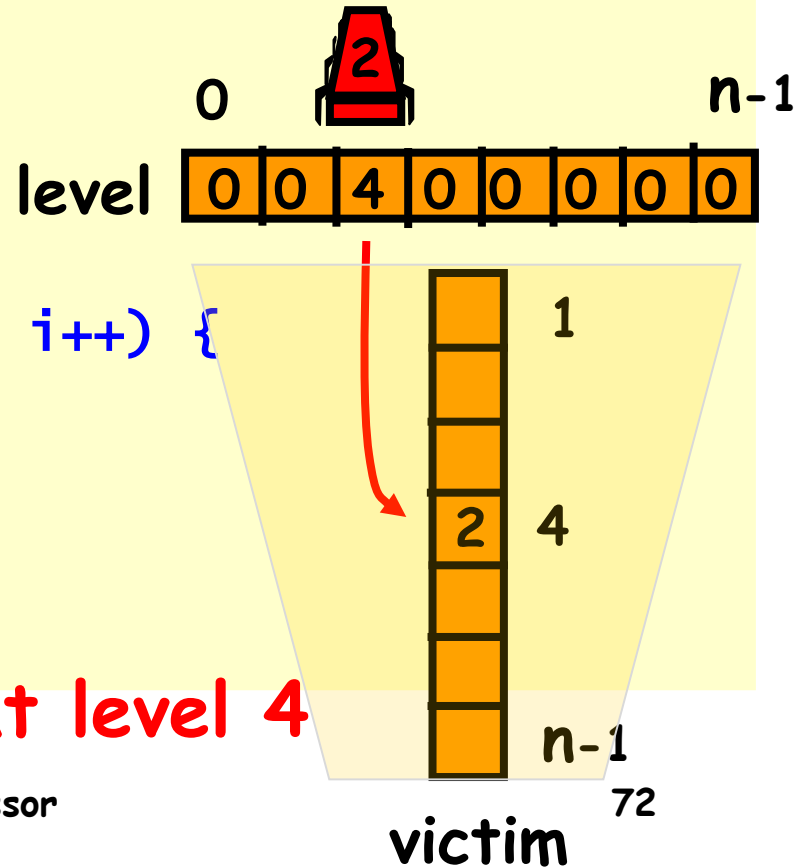
- At each level
 - At least one enters level
 - At least one blocked if many try
- Only one thread makes it through



Filter

```
class Filter implements Lock {  
    int[] level; // level[i] for thread i  
    int[] victim; // victim[L] for level L
```

```
    public Filter(int n) {  
        level = new int[n];  
        victim = new int[n];  
        for (int i = 1; i < n; i++) {  
            level[i] = 0;  
        }  
    }  
    ...  
}
```



Thread 2 at level 4

Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock(){  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$  level[k] >= L) &&  
                victim[L] == i );  
        }  
    }  
    public void unlock() {  
        level[i] = 0;  
    }  
}
```

Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }  
    public void release(int i) {  
        level[i] = 0;  
    }  
}
```

One level at a
time

Filter

```
class Filter implements Lock {
```

```
...
```

```
public void lock() {
```

```
    for (int L = 1; L < n; L++) {
```

```
        level[i] = L;
```

```
        victim[L] = i;
```

```
        while (( $\exists k \neq i$ ) level[k] >= L) &&  
            victim[L] == i)
```

```
    }
```

```
public void release(int i)
```

```
    level[i] = 0;
```

```
}
```

**Announce
intention to
enter level L**

Filter

```
class Filter implements Lock {  
    int level[n];  
    int victim[n];  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }  
    public void release(int i)  
        level[i] = 0;  
}
```

**Give priority to
anyone but me**

Filter

Wait as long as someone else is at same or higher level, and I'm designated victim

```
for (int L = 1; L < N; L++) {  
    level[i] = L;  
    victim[L] = i;  
    while (( $\exists k \neq i$ ) level[k] >= L) &&  
        victim[L] == i);  
}  
public void release(int i) {  
    level[i] = 0;  
}
```

Filter

```
class Filter implements Lock {  
    int level[n];  
    int victim[n];  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }
```

Thread enters level L when it completes the loop

No Starvation

- Filter Lock satisfies properties:
 - Just like Peterson Alg at any level
 - So no one starves
- But what about fairness?
 - Threads can be overtaken by others