

# 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

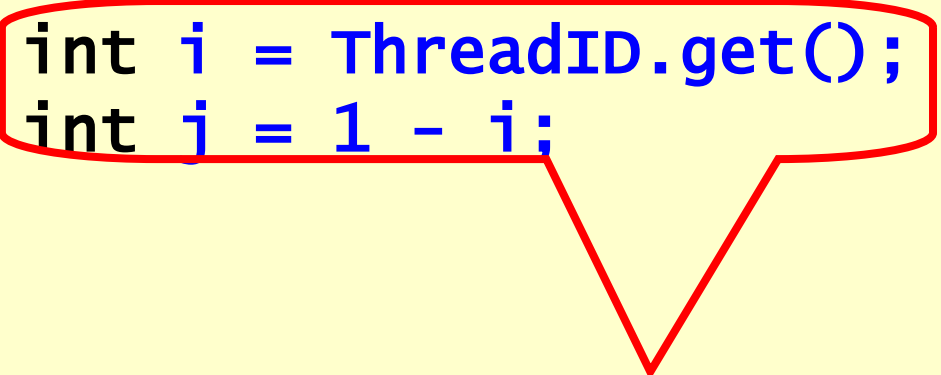


# Two-Thread Conventions

```
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() {}  
}
```

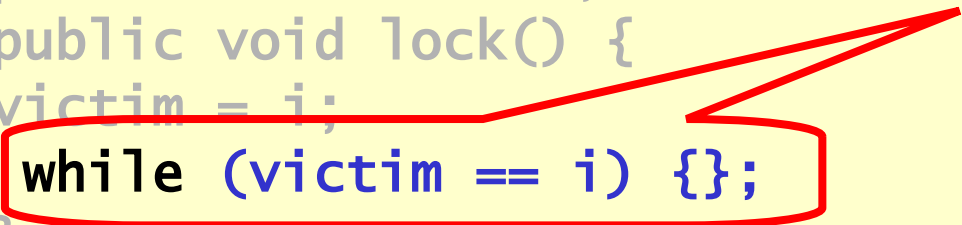
Let other go  
first



# LockTwo

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

**Wait for permission**



# 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`

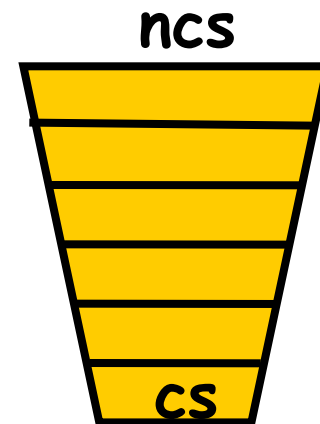
- When  $j$  re-enters
  - it sets `victim` to  $j$ .
  - So  $i$  gets in

```
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

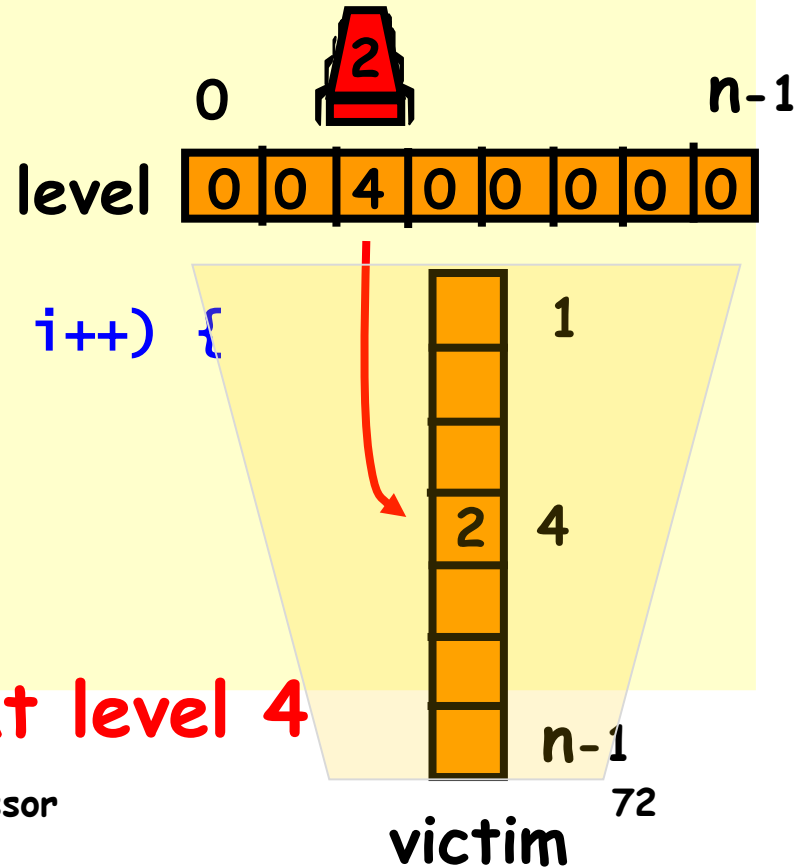
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



# 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 != 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

# Bakery Algorithm

- Provides First-Come-First-Served
- How?
  - Take a "number"
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$ 
    - If  $a > b$ , or  $a = b$  and  $i > j$

# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int i = 0; i < n; i++) {  
            flag[i] = false; label[i] = 0;  
        }  
    }  
    ...  
}
```

# Bakery Algorithm

```
class Bakery implements Lock {
```

```
    boolean[] flag;
```

```
    Label[] label;
```

```
    public Bakery (int n) {
```

```
        flag = new boolean[n];
```

```
        label = new Label[n];
```

```
        for (int i = 0; i < n; i++) {
```

```
            flag[i] = false; label[i] = 0;
```

```
        }
```

```
    }
```

```
    ...
```



CS

# Bakery Algorithm

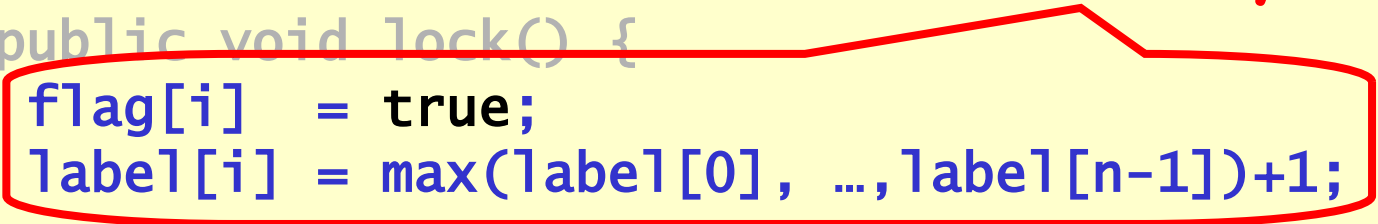
```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i],i) > (label[k],k));  
    }
```



# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

**Doorway**



# Bakery Algorithm

```
class Bakery implements Lock {
```

```
...
```

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

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

```
    label[i] = max(label[0], ..., label[n-1])+1;
```

```
    while ( $\exists k$  flag[k]  
           && (label[i], i) > (label[k], k));
```

```
}
```

I'm interested

# Bakery Algorithm

Take increasing  
label (read labels  
in some arbitrary  
order)

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }
```

Someone is  
interested



# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean flag[n];  
    int label[n];  
  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }
```

Someone is  
interested

With lower (label,i) in  
lexicographic order

# Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
  
    public void unlock() {  
        flag[i] = false;  
    }  
}
```

# Bakery Algorithm

```
class Bakery implements Lock {
```

```
...
```

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

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

```
}
```

```
}
```

No longer  
interested



labels are always increasing

# No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)



# Mutual Exclusion

- Suppose A and B in CS together
- Suppose A has earlier label
- When B entered, it must have seen
  - flag[A] is *false*, or
  - label[A] > label[B]

```
class Bakery implements Lock {  
  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0],  
                        ..., label[n-1])+1;  
  
        while (∃k flag[k]  
                && (label[i], i) >  
                (label[k], k));  
    }  
}
```

# Mutual Exclusion

- Labels are strictly increasing so
- B must have seen `flag[A] == false`

# Mutual Exclusion

- Labels are strictly increasing so
- B must have seen  $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$

# Mutual Exclusion

- Labels are strictly increasing so
- B must have seen  $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$
- Which contradicts the assumption that A has an earlier label

# **This work is licensed under a**

## **Creative Commons Attribution-ShareAlike 2.5 License.**

- **You are free:**
  - **to Share — to copy, distribute and transmit the work**
  - **to Remix — to adapt the work**
- **Under the following conditions:**
  - **Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).**
  - **Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.**
- **For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to**
  - **<http://creativecommons.org/licenses/by-sa/3.0/>.**
- **Any of the above conditions can be waived if you get permission from the copyright holder.**
- **Nothing in this license impairs or restricts the author's moral rights.**