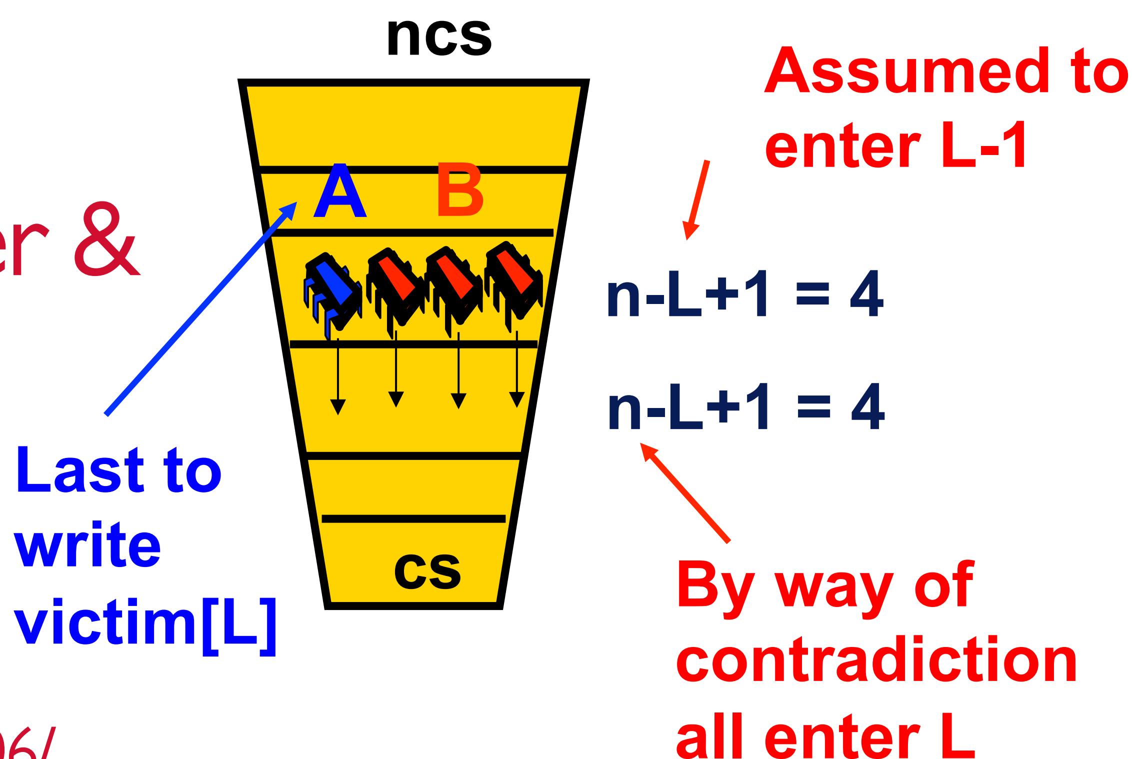
CS-206 Concurrency

Lecture 6

Peterson's, Filter & Bakery



Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

Where are We?

M	T	W (Lecture & Lab)	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	[illegible]	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Peterson's algorithm
  - Two threads
  - Deadlock free
  - Starvation free
- From 2 to n threads
  - Filter lock
  - Lamport's Bakery algo
- Next week
  - Synchronization

Recall: 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]) {}
 }
```

Wait for other flag to become false

Recall: LockTwo

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

 public void unlock() {}
}
```

LockTwo

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

 public void unlock() {}
}
```

Let other go first

LockTwo

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

 public void unlock() {}
}
```

Wait for permission

LockTwo

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

 public void unlock() {}
}
```

Nothing to do

LockOne & LockTwo

- LockOne
  - Guarantees mutual exclusion
    - Using while(flag[])
  - But might deadlock while entering

- LockTwo
  - Guarantees waiting until another thread wants to enter
    - Using while (victim == i)
  - But might deadlock if one thread finishes
    - Other might continue waiting

Peterson's Algorithm (Gary L. Peterson)

```
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

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

(1) $write_B(Flag[B]=true) \rightarrow write_B(victim=B)$

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

From the Code

Also from the Code

(2) $\text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B])$
$\rightarrow \text{read}_A(\text{victim})$

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

Assumption

(3) $write_B(victim=B) \rightarrow write_A(victim=A)$

W.L.O.G. assume A is the last thread to write **victim**

Combining Observations

(1) $write_B(flag[B]=true) \rightarrow write_B(victim=B)$

(3) $write_B(victim=B) \rightarrow write_A(victim=A)$

(2) $write_A(victim=A) \rightarrow read_A(flag[B])$
$\rightarrow read_A(victim)$

Combining Observations

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

(2) $\text{write}_A(\text{victim=A}) \rightarrow \text{read}_A(\text{flag[B]})$
$\rightarrow \text{read}_A(\text{victim})$

Combining Observations

(1) $write_B(flag[B]=true) \rightarrow$

(3) $write_B(victim=B) \rightarrow$

(2) $write_A(victim=A) \rightarrow read_A(flag[B])$

$\rightarrow read_A(victim)$

A read flag[B] == true and victim == A, so it could not have entered the CS **(QED)**

Deadlock Free

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

- Thread blocked
  - only at **`while`** loop
  - only if other's flag is true
  - only if it is the victim
- Solo: other's flag is false
- Both: one or the other not the victim

Starvation Free

- Thread `i` would be blocked only if `j` re-enters so that
  `flag[j] == true` and `victim == i`
- But, when `j` re-enters
  - it sets `victim` to `j`
  - So `i` gets in

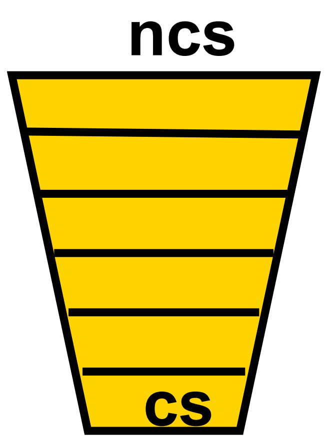
```
public void lock() {
   …
   flag[i] = true;
   victim     = i;
   while (flag[j] && victim == i) {};
}

public void unlock() {
   flag[i] = false;
}
```

The Filter Algorithm for *n* Threads

There are **n-1** "waiting rooms" called levels

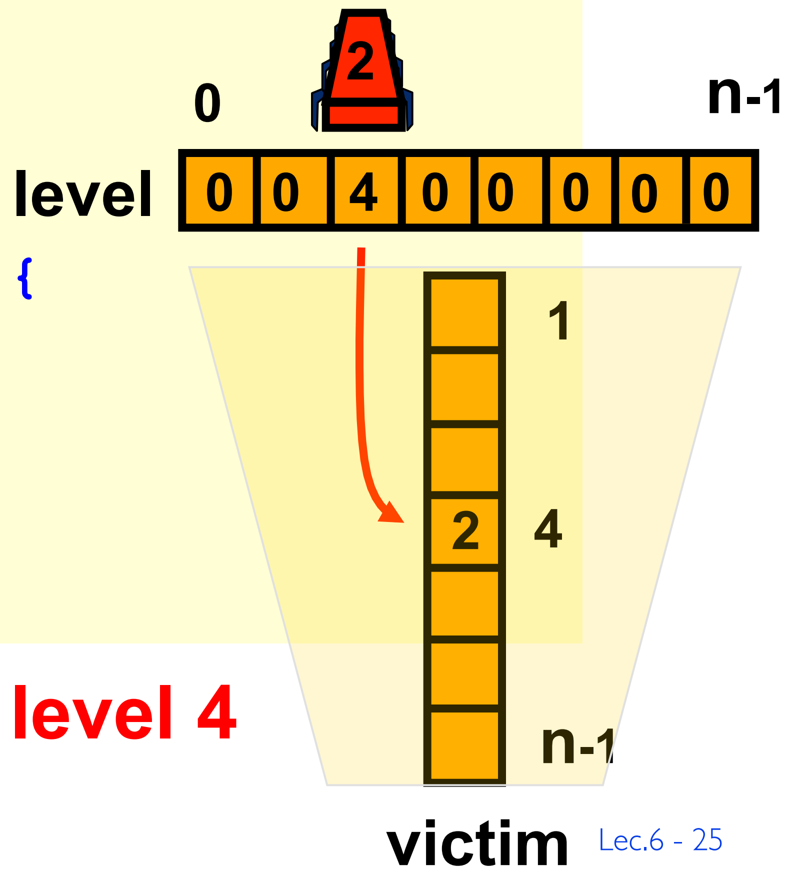
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



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 ((∃ 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 ((∃ k != 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 ((∃ k != 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 ((∃ k != 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

```
public void lock() {
  for (int L = 1; L < n; L++) {
    level[i]  = L;
    victim[L] = i;
    while ((∃ k != i) level[k] >= L) &&
           victim[L] == i) {};
  }}
public void release(int i) {
  level[i] = 0;
}}
```

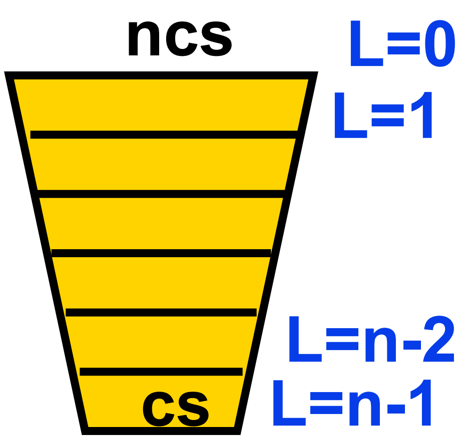
Filter

```
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 ((∃ k != i) level[k] >= L) &&
             victim[L] == i) {};
    }}
```

Thread *enters* level L when it completes the loop

Claim

- Start at level L=0
- At most n-L threads enter level L
- Mutual exclusion at level L=**n-1**



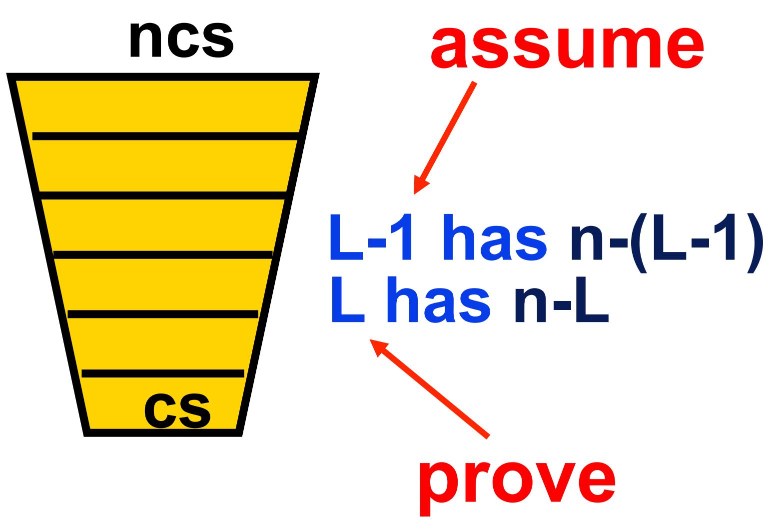
Induction Hypothesis

- No more than **n-(L-1)** at level **L-1**
- Induction step: by contradiction

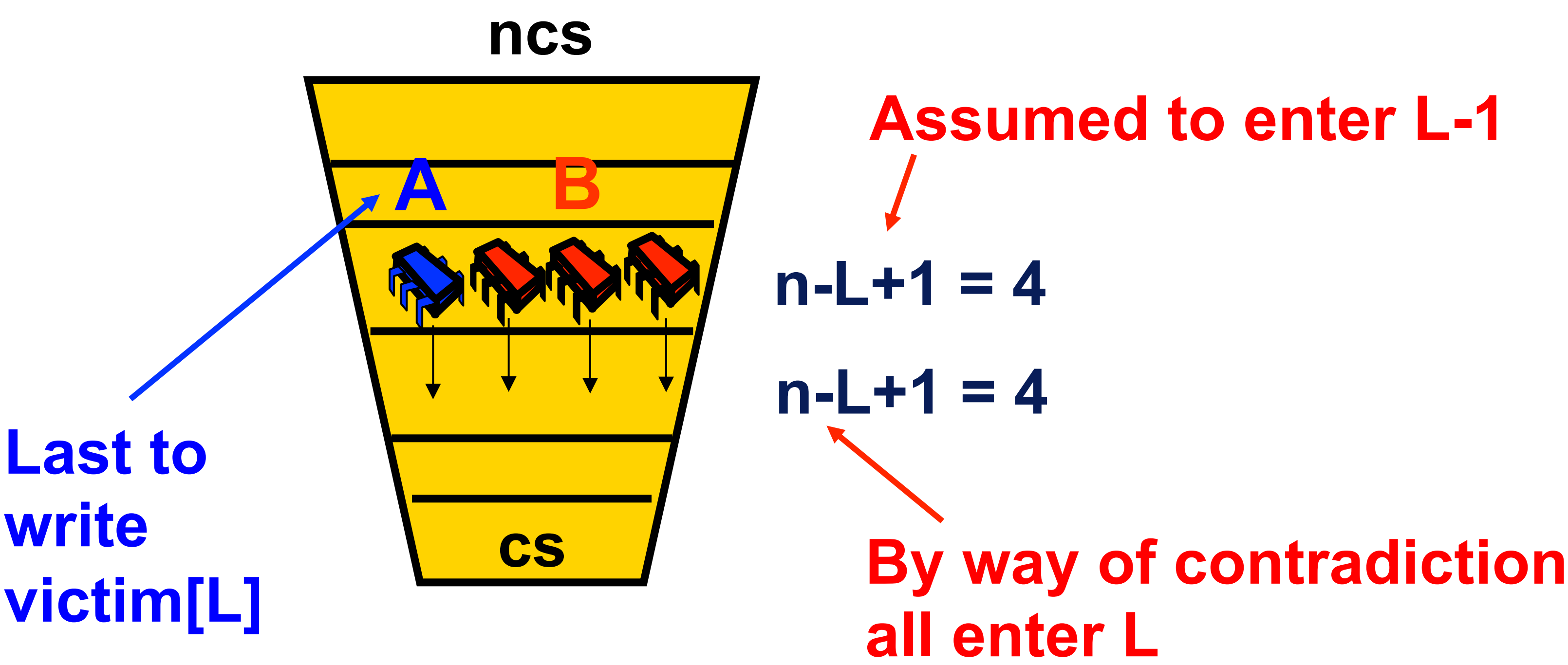
▶ Assume all at level **L-1** enter level **L**

▶ A last to write victim[**L**]

▶ B is any other thread at level **L**



Proof Structure



Show that A must have seen B in level[L] and since victim[L] == A could not have entered

Just Like Peterson

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i] = L;
   victim[L]  = i;
   while ((∃ k != i) level[k] >= L)
          && victim[L] == i) {};
   }}
```

From the Code

From the Code

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B]) \rightarrow \text{read}_A(\text{victim}[L])$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i] = L;
   victim[L]  = i;
   while ((∃ k != i) level[k] >= L)
          && victim[L] == i) {};
 }}
```

By Assumption

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

By assumption, A is the last thread to write **victim[L]**

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$
$\rightarrow \text{read}_A(\text{victim}[L])$

Combining Observations

(1) $write_B(level[B]=L) \rightarrow$

(3) $write_B(victim[L]=B) \rightarrow write_A(victim[L]=A)$

(2) $\rightarrow read_A(level[B])$

$\rightarrow read_A(victim[L])$

Combining Observations

(1) $\text{write}_B(\text{level[B]=L})$ ➔

(3) $\text{write}_B(\text{victim[L]=B})$ ➔ $\text{write}_A(\text{victim[L]=A})$

(2) ➔ $\text{read}_A(\text{level[B]})$

➔ $\text{read}_A(\text{victim[L]})$

A read level[B] ≥ L, and victim[L] = A, so it could not have entered level L!

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

Bounded Waiting

- Want stronger fairness guarantees
- Thread not "overtaken" too much
- If A starts before B, then A enters before B?
- But what does "start" mean?
- Need to adjust definitions ….

Bounded Waiting

- Divide **`lock()`** method into 2 parts:
  - Doorway interval:
    - Written D_A
    - always finishes in finite steps
  - Waiting interval:
    - Written W_A
    - may take unbounded steps

First-Come-First-Served

- For threads A and B:
  - If $D_A^k \rightarrow D_B^j$
    - A's k-th doorway precedes B's j-th doorway
  - Then $CS_A^k \rightarrow CS_B^j$
    - A's k-th critical section precedes B's j-th critical section
    - B cannot overtake A

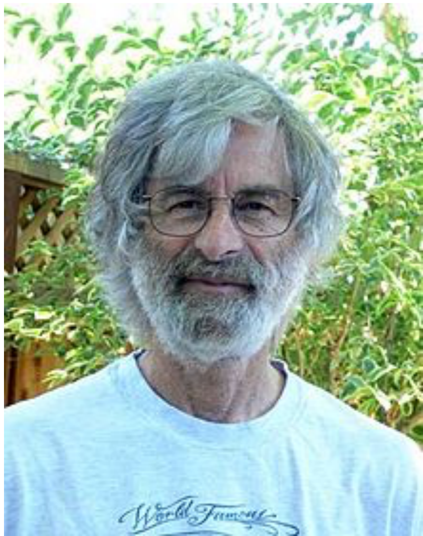
Fairness Again

- Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness
    - Can be overtaken **arbitrary** # of times
  - That's pretty lame…

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

Leslie Lamport

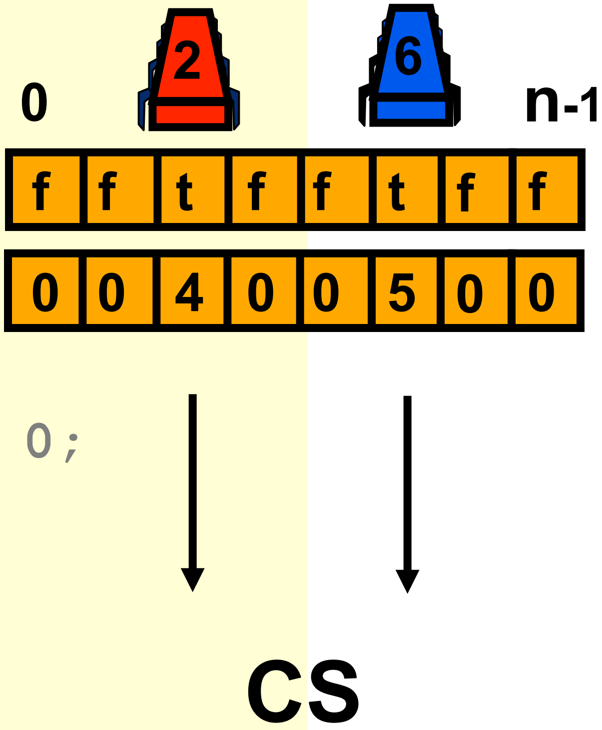


Bakery Algorithm

```
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;
    }
  }
 …
```



Bakery Algorithm

```
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));
 }
```

Bakery Algorithm

```
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));
 }
```

Doorway

Bakery Algorithm

```
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));
 }
```

I'm interested

Bakery Algorithm

```
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));
 }
```

Take increasing label (read labels in some arbitrary order)

Bakery Algorithm

```
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));
 }
```

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 (∃k flag[k]
            && (label[i],i) > (label[k],k));
 }
```

Someone is interested …

… whose (label,i) in lexicographic order is lower

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?)

First-Come-First-Served

- If D_A ➔ D_B then
  - A's label is smaller
- And:
  - $write_A$(label[A]) ➔
  - $read_B$(label[A]) ➔
  - $write_B$(label[B]) ➔ $read_B$(flag[A])
- So B sees
  - smaller label for A
  - locked out while flag[A] is true

```
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

- 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 flag[A] == false
- Labeling_B ➔ read_B(flag[A]) ➔ write_A(flag[A]) ➔ Labeling_A

Mutual Exclusion

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

Summary

- LockOne gives us mutual exclusion
- LockTwo gives us starvation freedom
  - as long as both threads are running
- Peterson's combines the two for 2 threads
- Filter lock
  - n-thread solution (with n=L levels)
  - Mutual exclusion at L=n-1
- Lamport's Bakery
  - Get a ticket
  - Order tickets with thread ID's lexographically