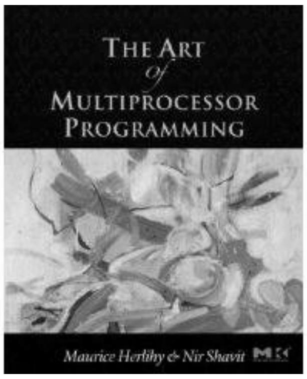
PROGRAMMAZIONE CONCORRENTE

1



The Art of Multiprocessor Programming
Maurice Herlihy & Nir Shavit

2

Mutual Exclusion

- Problema essenziale della programmazione concorrente
  - Coordinamento di azioni su risorse condivise
- Come si dimostrano proprieta' di astrazioni in presenza di thread
  - Problema essenziale nello sviluppo di applicazioni su sistemi multicore

3

Mutual Exclusion

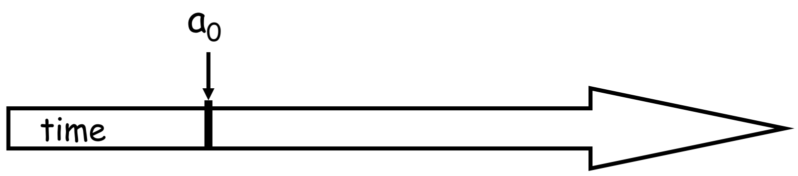
E. W. Dijkstra [1965]:

"Given in this paper is a solution to a problem which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. [...] Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved."

4

Scala temporale

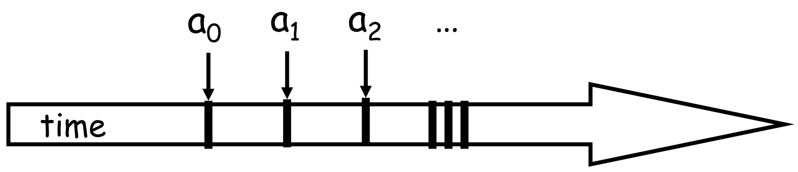
- Un *evento* a_0 di un thread A e'
  - instantaneo
  - Non si assume la simultaneita' di eventi



5

Thread (visione astratta)

- Un *thread* A una sequenza a_0, a_1, ... di eventi
  - Modello a "traccie" di esecuzione
  - $a_0 \rightarrow a_1$ indica l'ordine di occorrenza degli eeventi



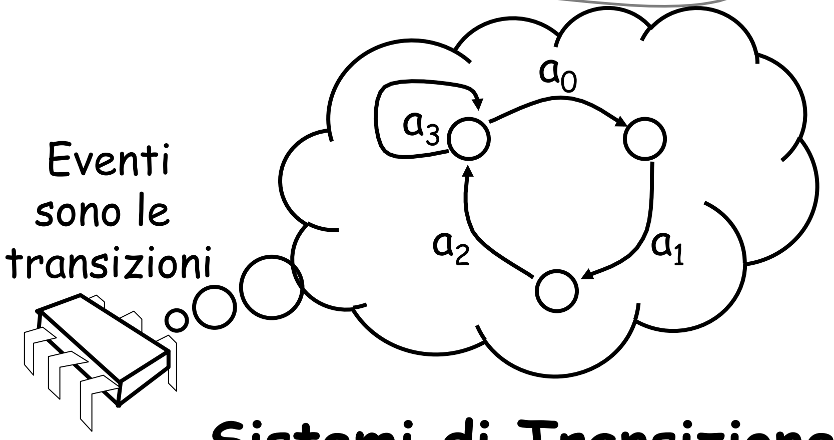
6

Cosa sono gli eventi?

- Assegna.nento aa variabili condivise (tra thread)
- Assegnamento a una variabile locale
- Invocazione di un metodo
- :
- :...

7

Threads sono sistemi di transizione



Sistemi di Transizione = State Machine

8

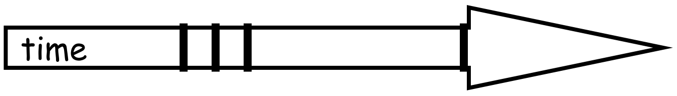
Stati

- Thread State
  - Informazioni di controllo (continuation stack)
  - Informazioni di ambiente (la struttura degli stack a run time)
- System state
  - Heap con gli oggetti condivisi
  - Stato di tutti i thread in esecuzione (ready, wait, run)

9

Concorrenza

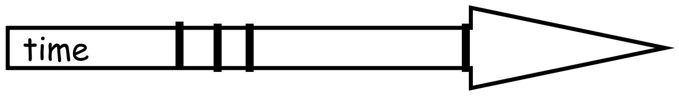
- Thread



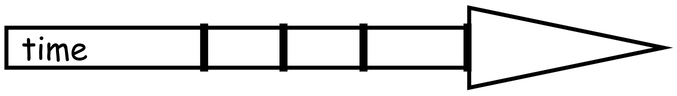
10

Concorrenza

- Thread A



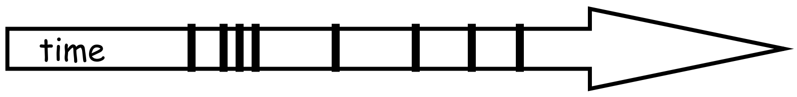
- Thread B



11

Interleaving

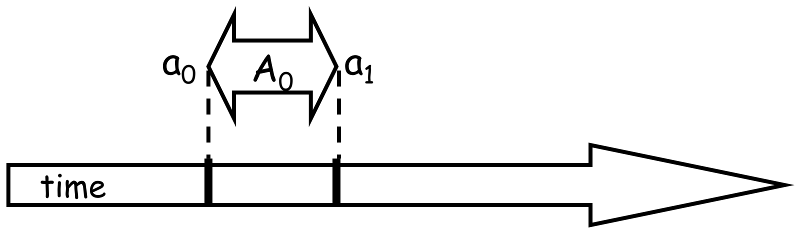
- Eventi dei thread
  - Eventi dei thread in esecuzione sono intercalati
  - Non sono sempre indipendenti



12

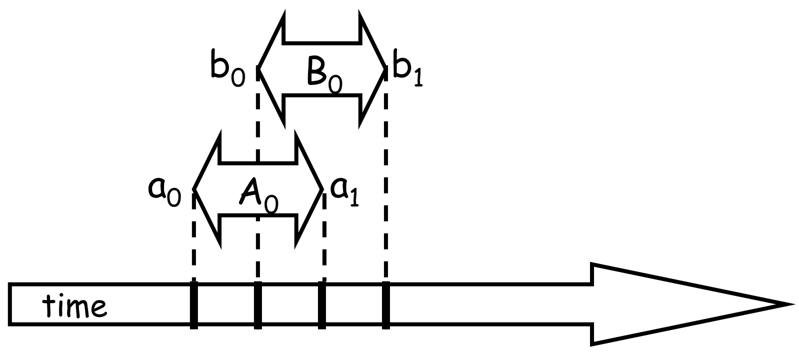
Intervallo

- Un *intervallo* $A_0 = (a_0, a_1)$
  - Periodo di tempo trascorso tra l'occorrenza dell'evento a_0 e l'evento a_1



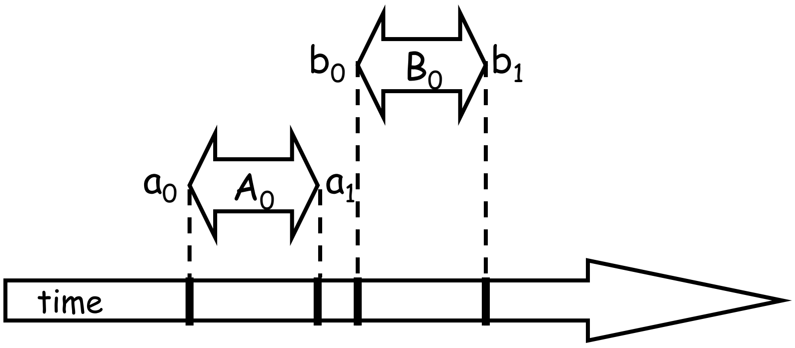
13

Intervalli si possono sovrapporre



14

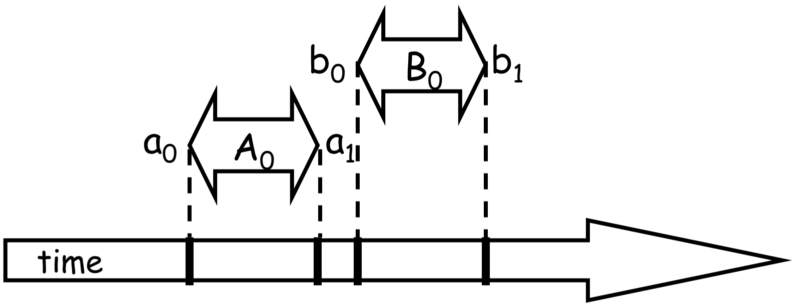
Intervalli possono essere disgiunti



15

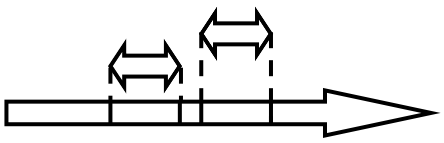
Precedenza

Intervallo A_0 precede intervallo B_0



16

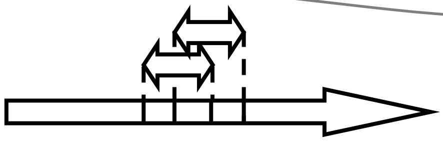
Ordinamento



- Notazione: $A_0 \rightarrow B_0$,
  - Evento terminale di A_0 occorre prima dell'evento inziale di B_0
  - "happens before" nozione introdotta da Leslie Lamport (Time, clocks and ordering of events in distributed systems)

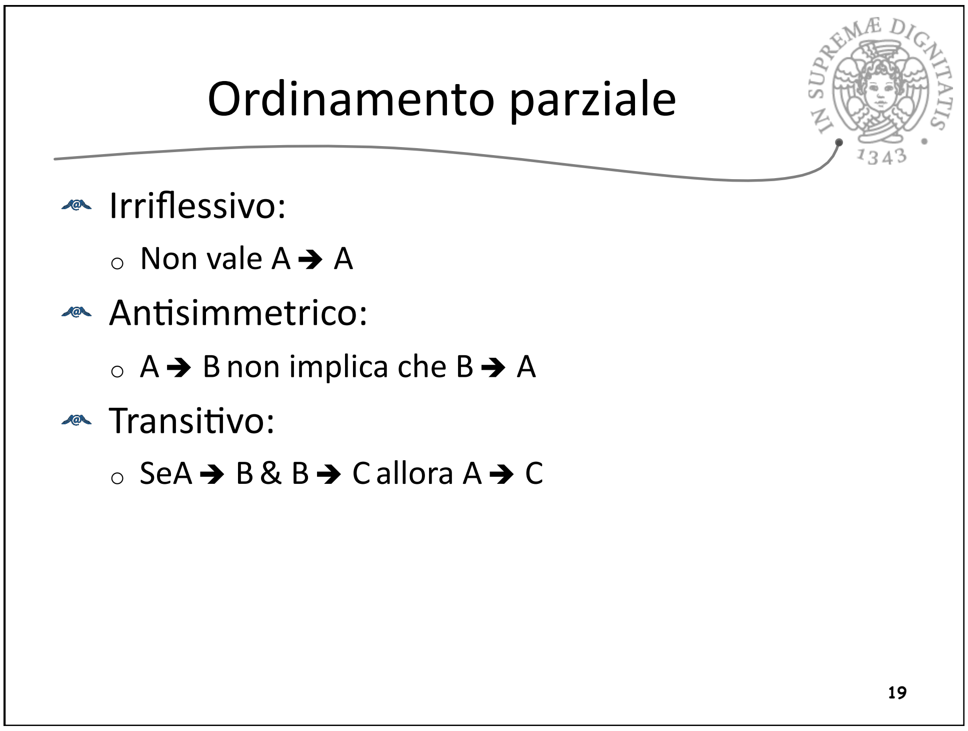
17

Ordinamento



- A ➔ A (FALSO)
- Se A ➔B allora B ➔A (FALSO)
- Se A ➔B & B ➔C A ➔C
- True Concurrency: A ➔B & B ➔A potrebbero essere entrambe false

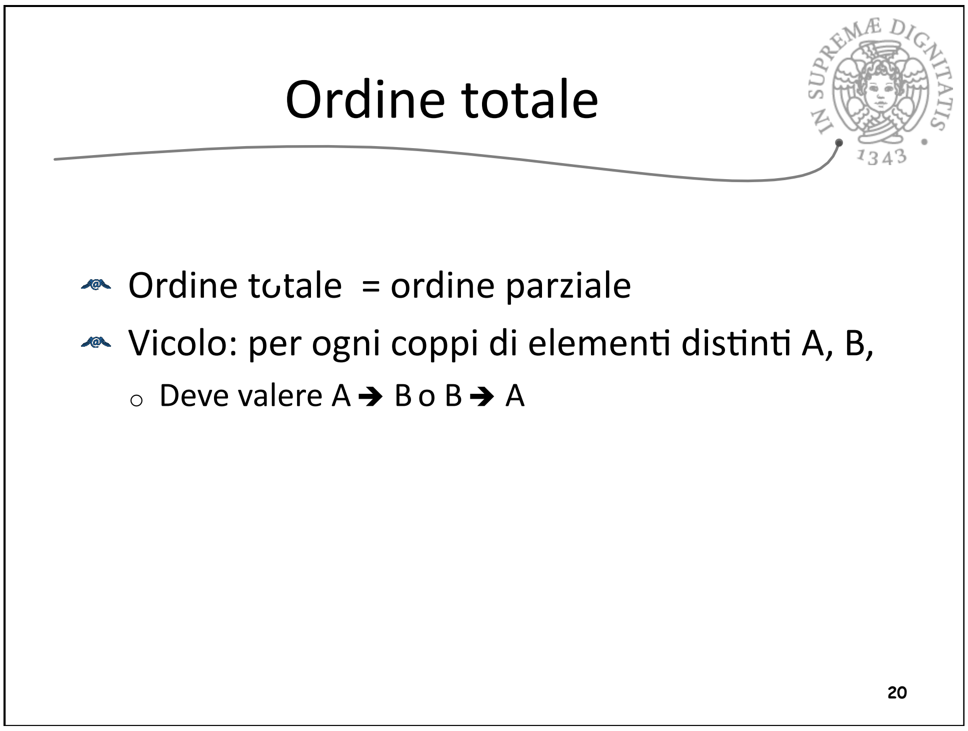
18



Ordinamento parziale

- Irriflessivo:
  - Non vale A ➔ A
- Antisimmetrico:
  - A ➔ B non implica che B ➔ A
- Transitivo:
  - Se A ➔ B & B ➔ C allora A ➔ C

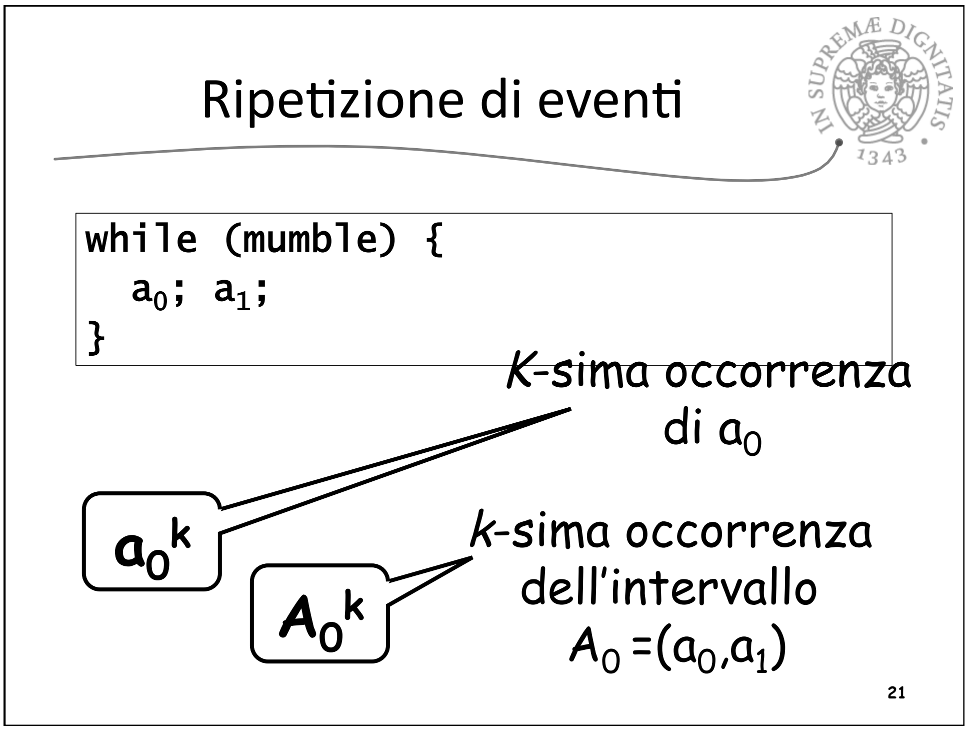
19



Ordine totale

- Ordine totale = ordine parziale
- Vicolo: per ogni coppi di elementi distinti A, B,
  - Deve valere A ➔ B o B ➔ A

20



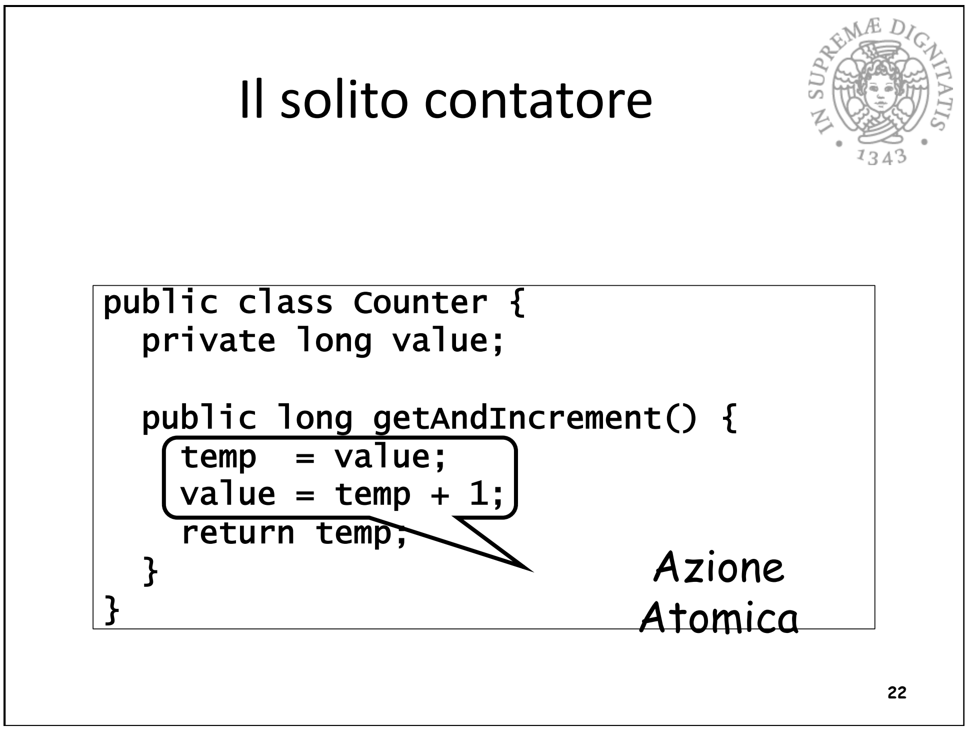
Ripetizione di eventi

```
while (mumble) {
  a0; a1;
}
```

a_0^k — *K*-sima occorrenza di a_0

A_0^k — *k*-sima occorrenza dell'intervallo $A_0 = (a_0, a_1)$

21



Il solito contatore

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

Azione Atomica

22

Locks (Mutual Exclusion)

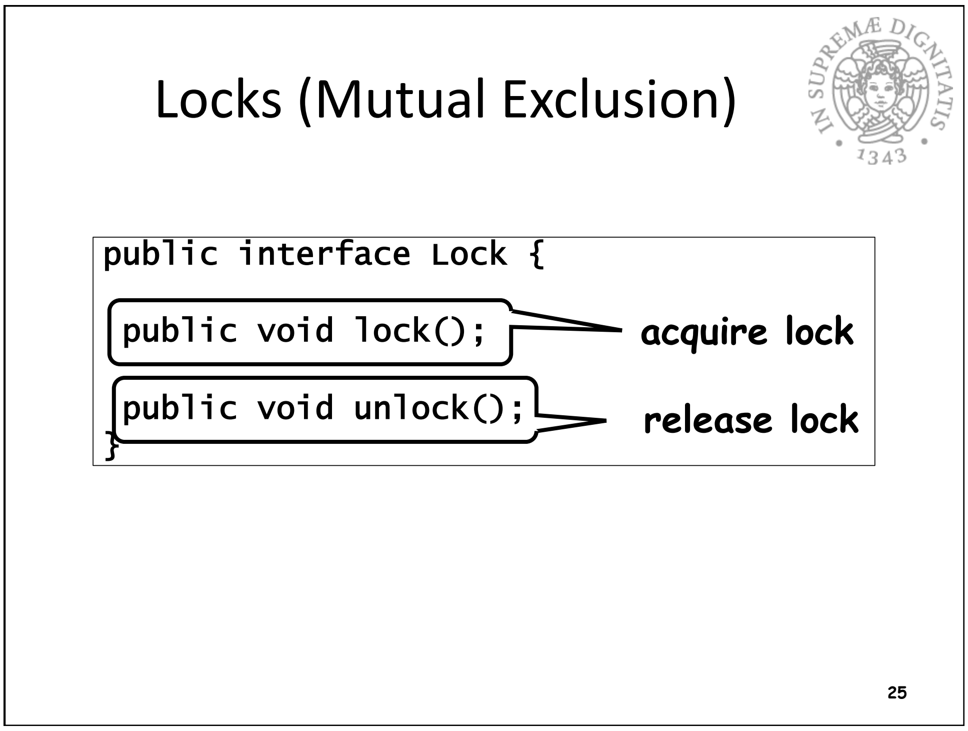
```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Definizione dei lock usata in Java

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock



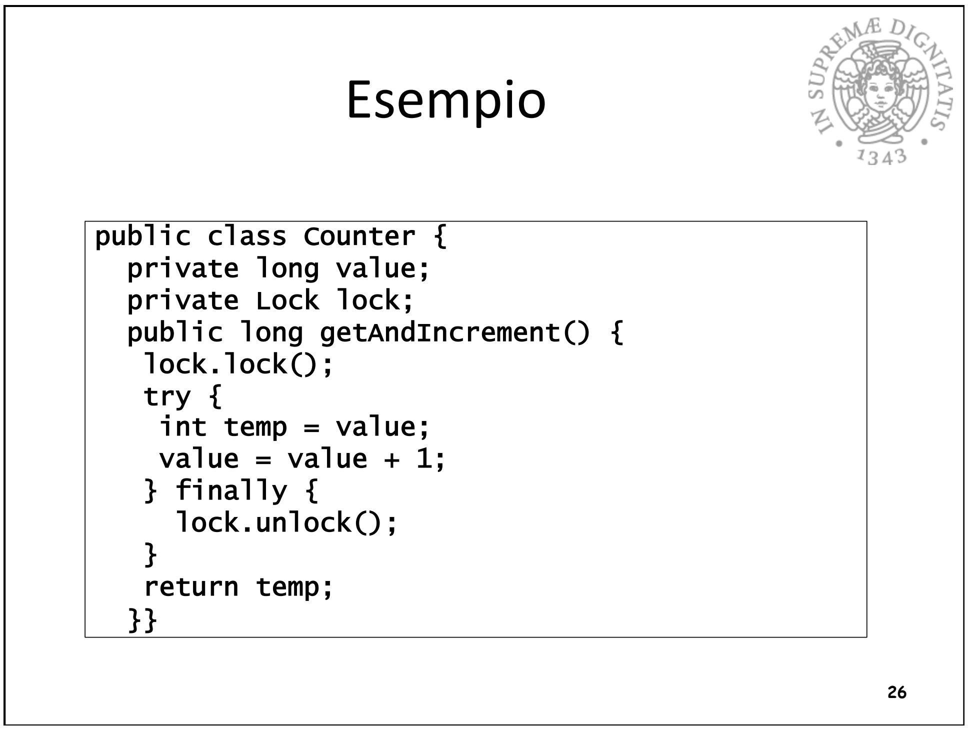
Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

release lock

25



Esempio

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

26

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

acquire Lock

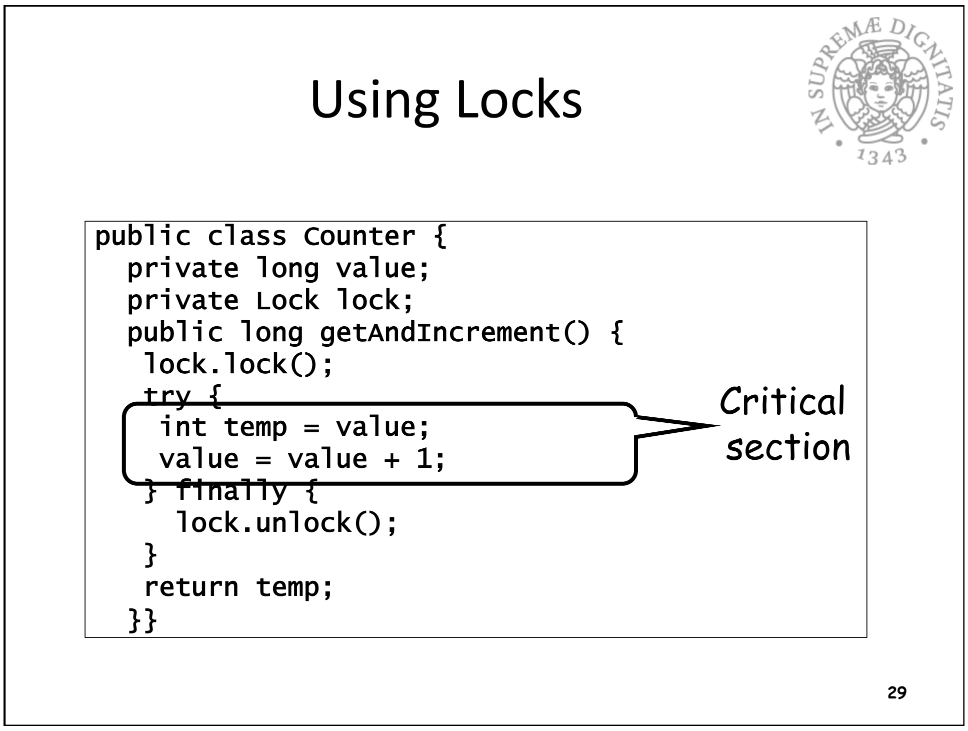
27

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Release lock

28

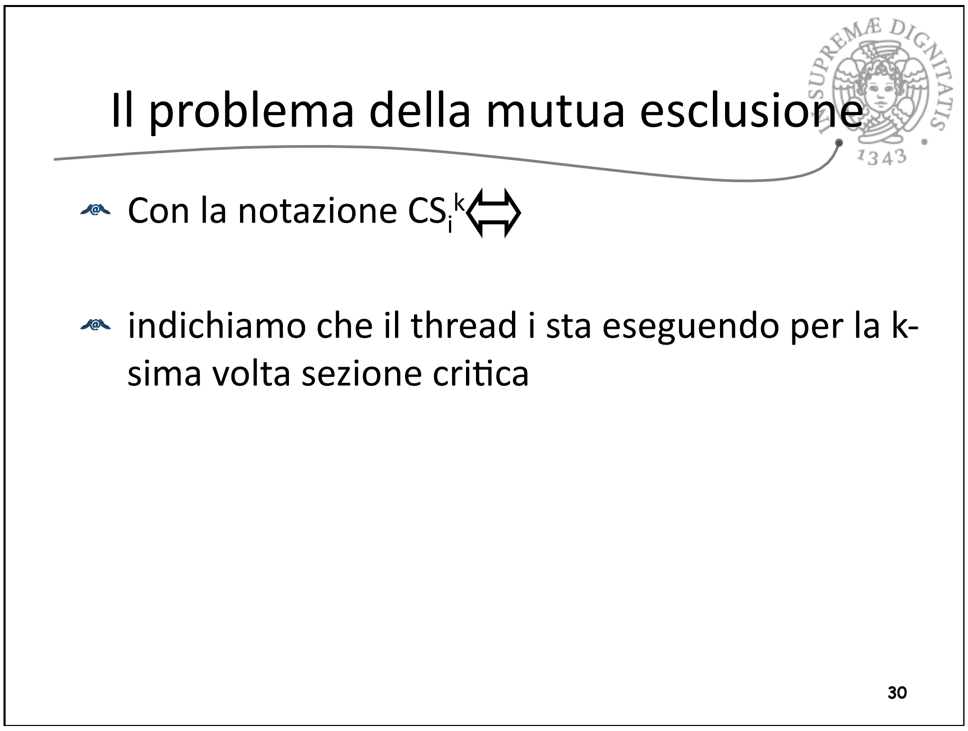


Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Critical section

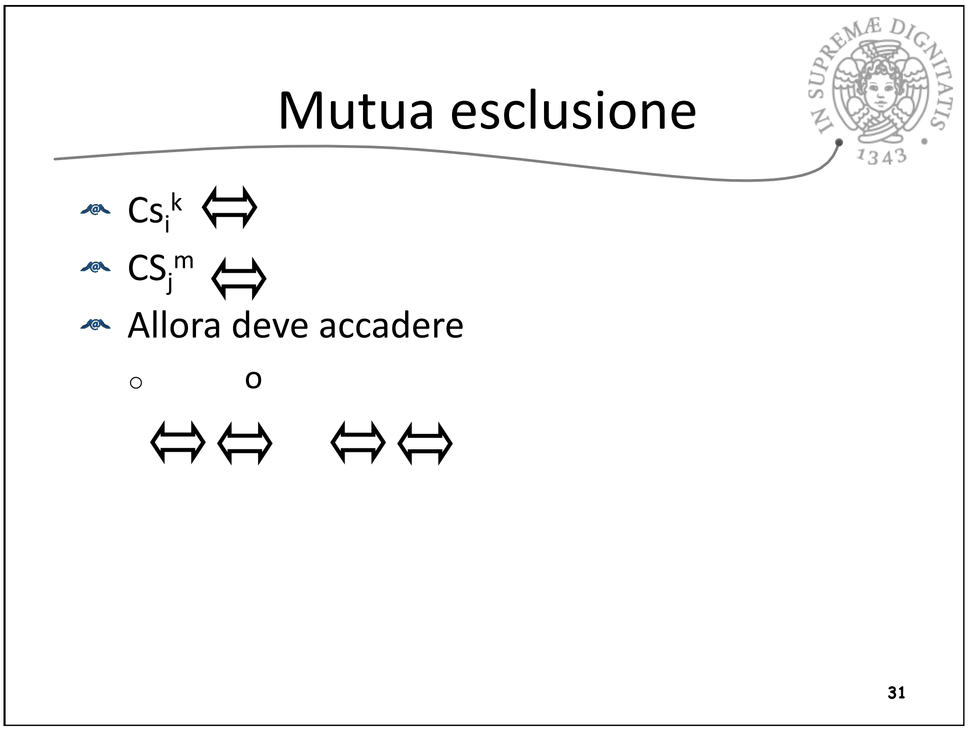
29



Il problema della mutua esclusione

- Con la notazione $CS_i^k \Leftrightarrow$
- indichiamo che il thread i sta eseguendo per la k-sima volta sezione critica

30

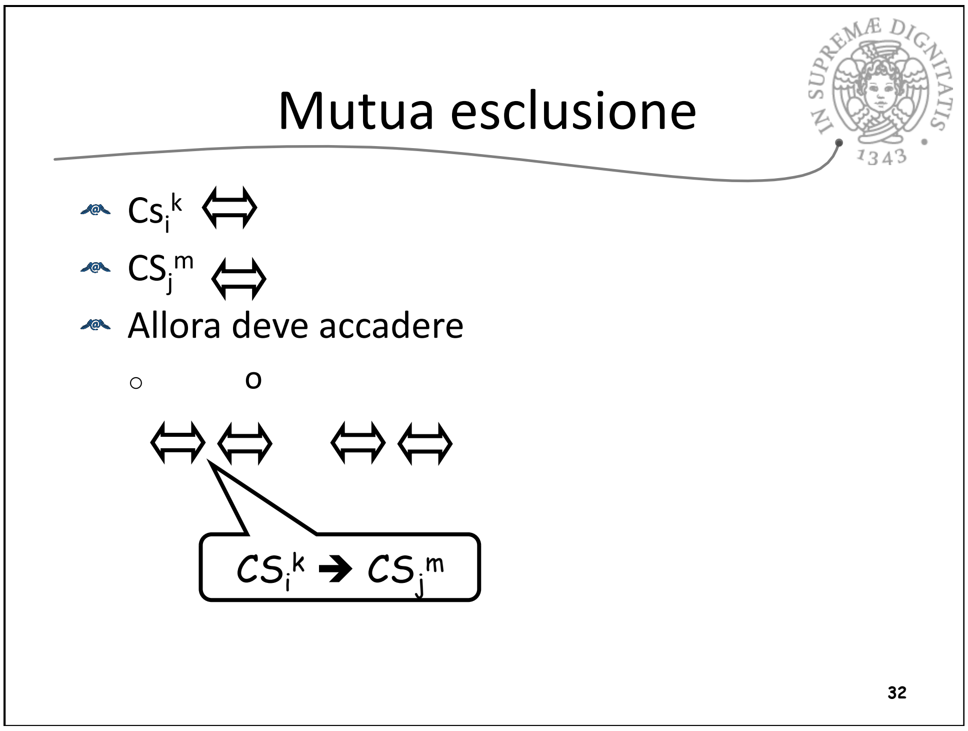


Mutua esclusione

- Cs_i^k ⇔
- CS_j^m ⇔
- Allora deve accadere
  - o

  ⇔⇔ ⇔⇔

31



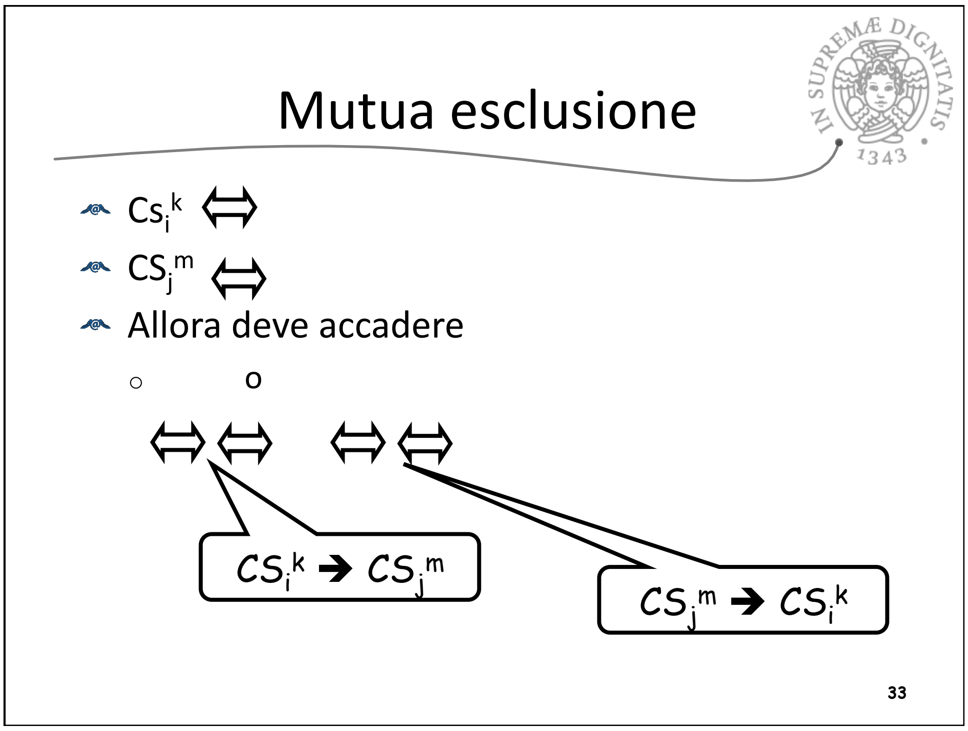
Mutua esclusione

- Cs_i^k ⇔
- CS_j^m ⇔
- Allora deve accadere
  - o

  ⇔⇔ ⇔⇔

$CS_i^k \rightarrow CS_j^m$

32



Mutua esclusione

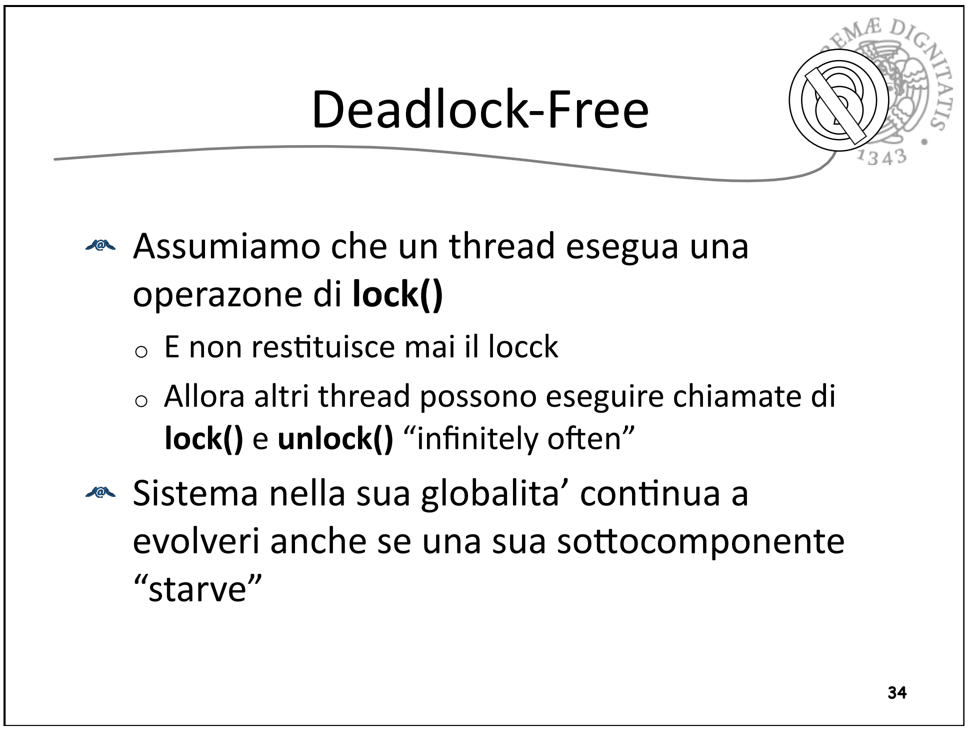
- Cs_i^k $\Leftrightarrow$
- CS_j^m $\Leftrightarrow$
- Allora deve accadere
  - o o

$\Leftrightarrow\Leftrightarrow$ $\Leftrightarrow\Leftrightarrow$

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

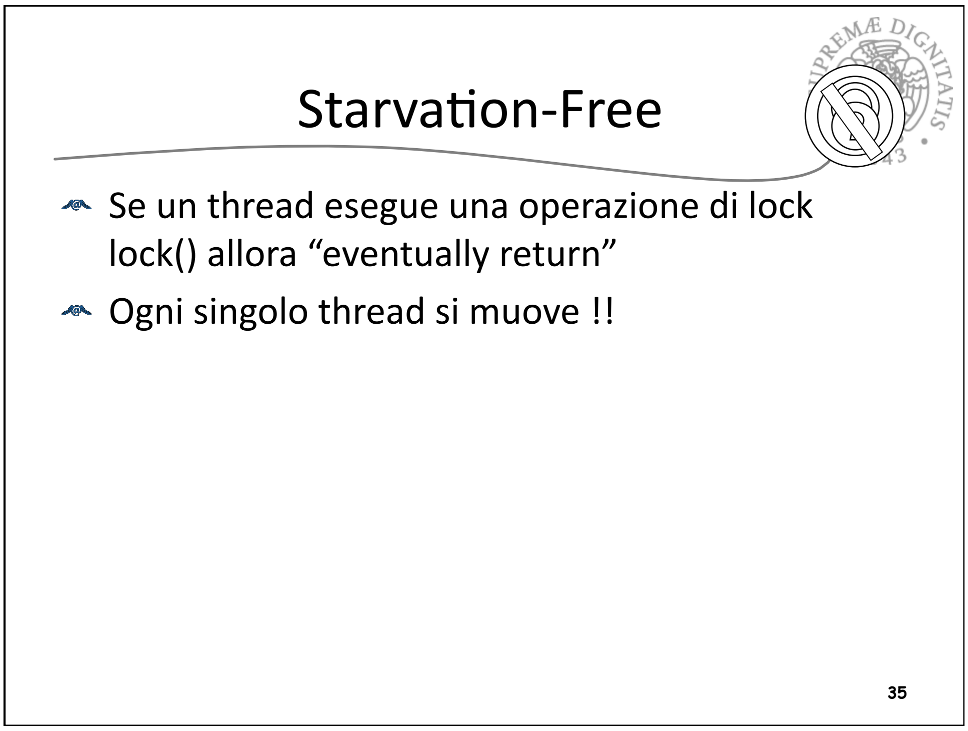
33



Deadlock-Free

- Assumiamo che un thread esegua una operazone di **lock()**
  - E non restituisce mai il locck
  - Allora altri thread possono eseguire chiamate di **lock()** e **unlock()** “infinitely often”
- Sistema nella sua globalita’ continua a evolveri anche se una sua sottocomponente “starve”

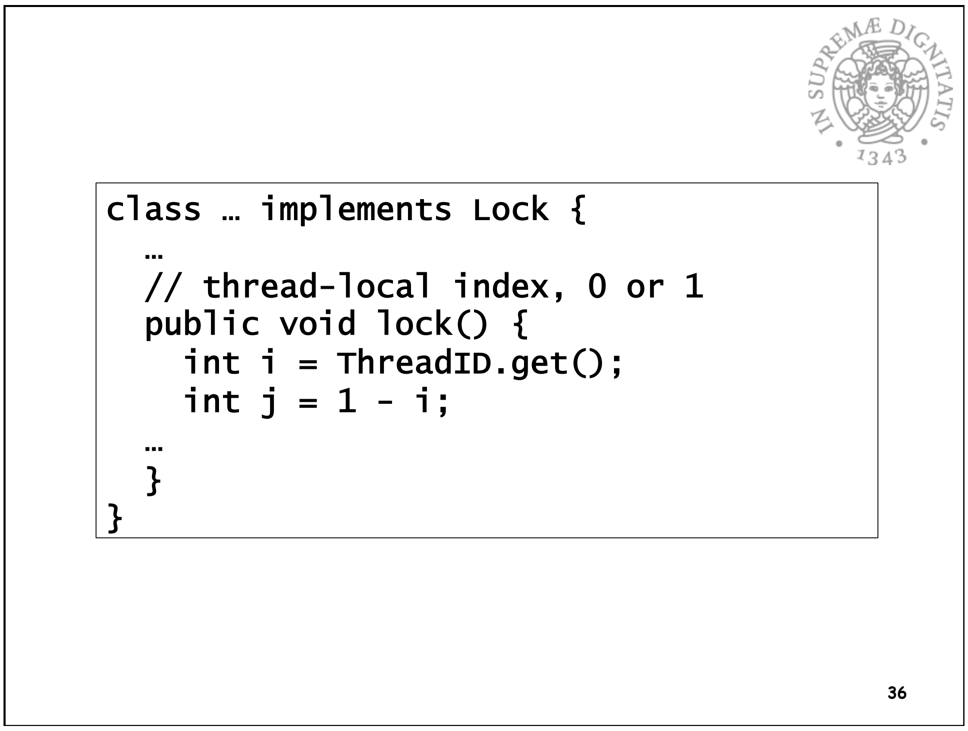
34



Starvation-Free

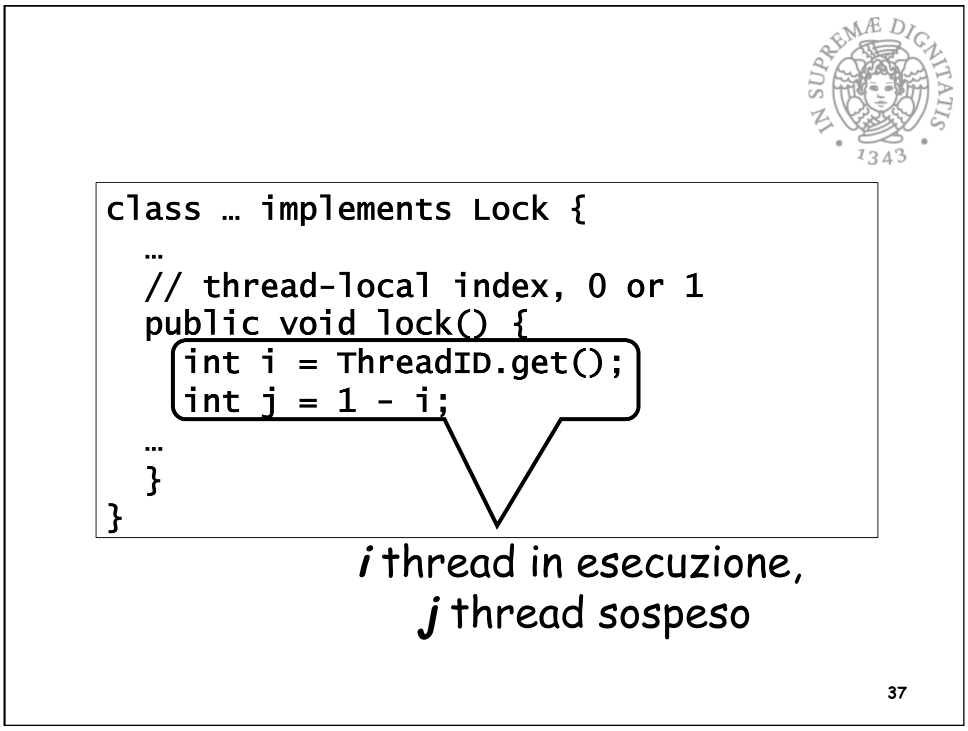
- Se un thread esegue una operazione di lock lock() allora “eventually return”
- Ogni singolo thread si muove !!

35



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

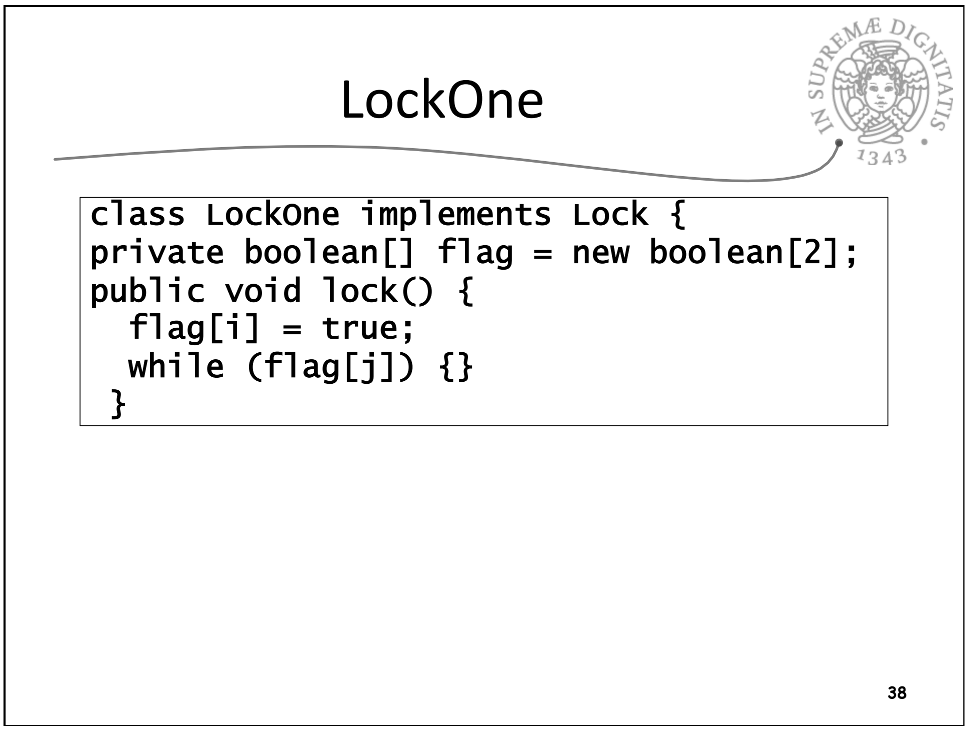
36



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

i thread in esecuzione,
j thread sospeso

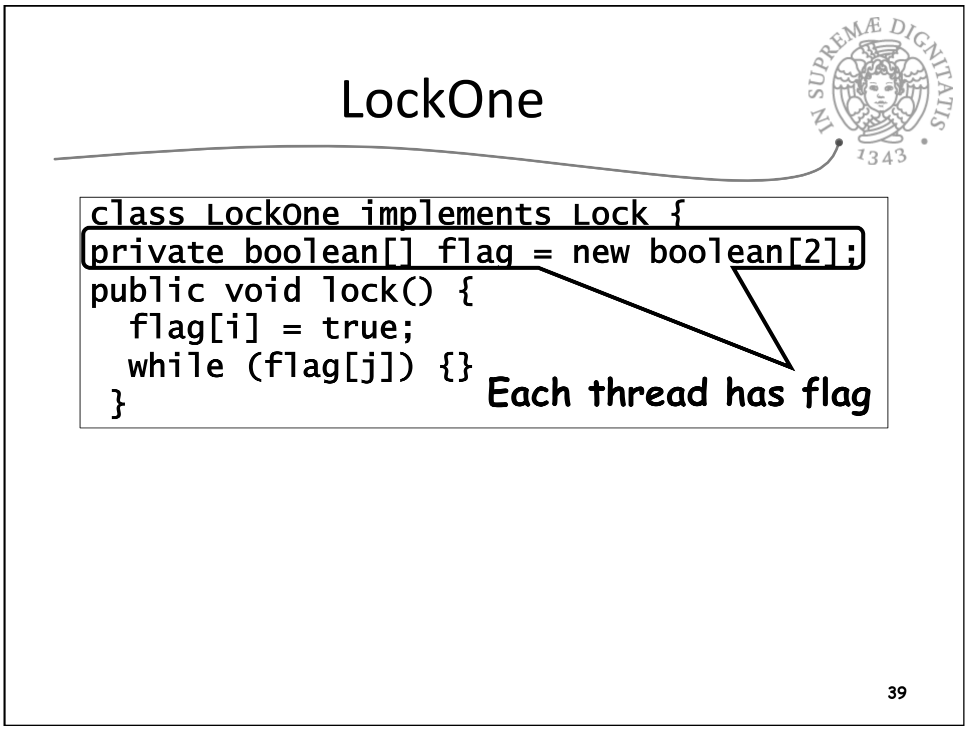
37



LockOne

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

38

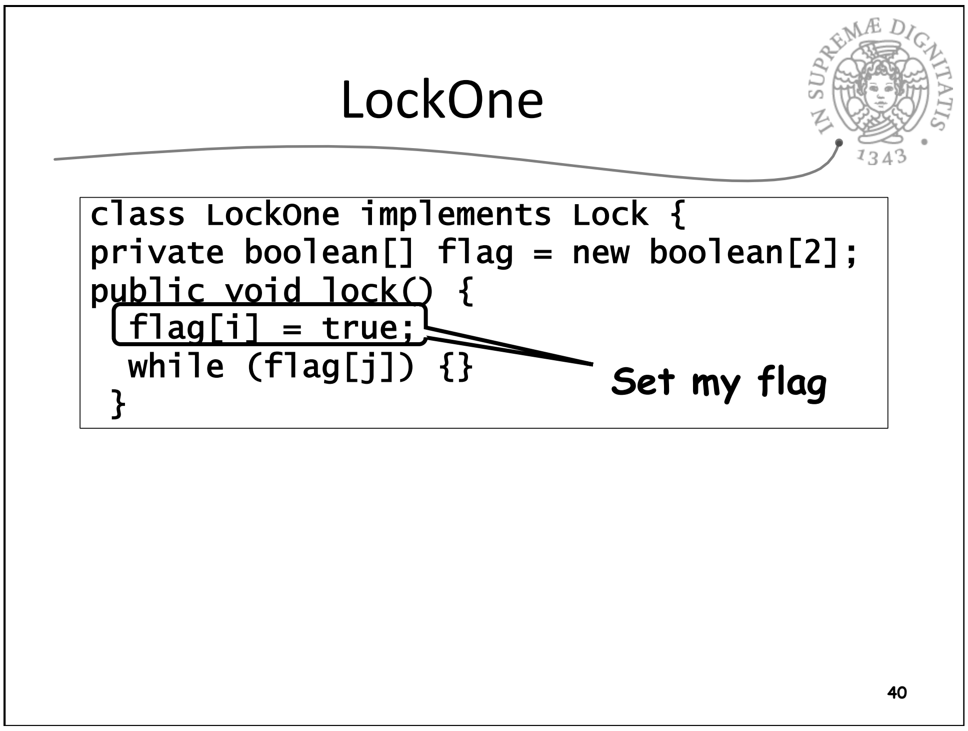


LockOne

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

Each thread has flag

39



LockOne

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

Set my flag

40

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

41

LockOne verifica la proprieta' di mutua esclusione?

- Ipotesi (per assurdo): CS_A^j si soprappone con CS_B^k
- Consideriamo l'occorrenza (j-sima e k-sima)
- Si deriva una contraddizione

42

Analizziamo il codice

- **write_A(flag[A]=true) → read_A(flag[B]==false) →CS_A**
- **write_B(flag[B]=true) → read_B(flag[A]==false) → CS_B**

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

43

Ipotesi

- **read_A(flag[B]==false) → write_B(flag[B]=true)**
- **read_B(flag[A]==false) → write_A(flag[A]=true)**

44

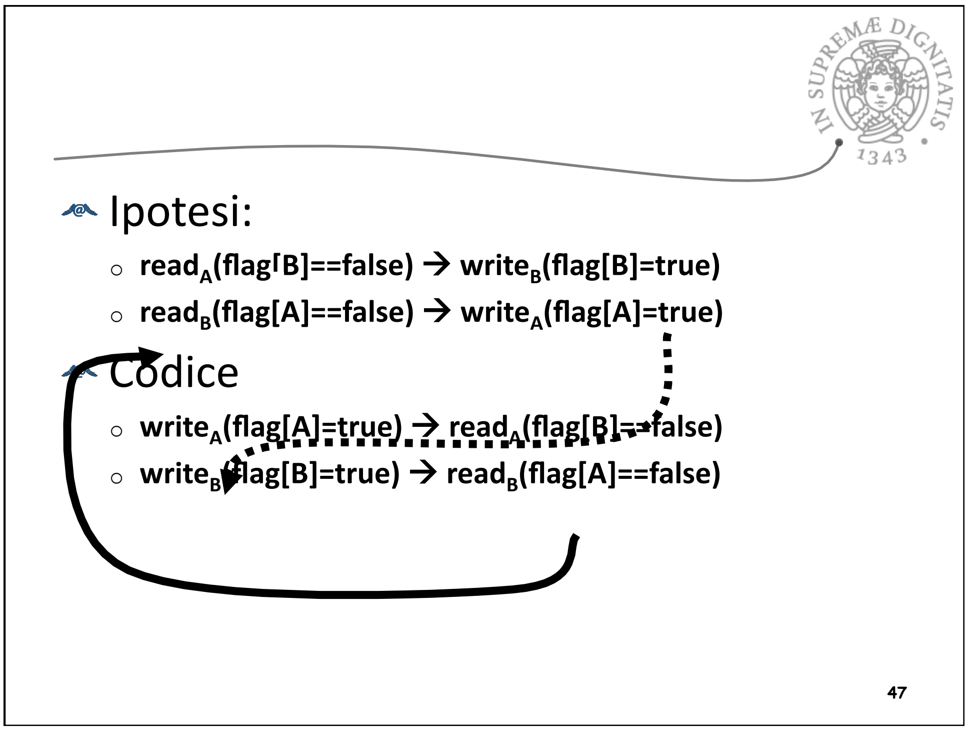
Mettiamo tutto assieme

- Ipotesi:
  - read_A(flag[B]==false) → write_B(flag[B]=true)
  - read_B(flag[A]==false) → write_A(flag[A]=true)
- Il codice
  - write_A(flag[A]=true) → read_A(flag[B]==false)
  - write_B(flag[B]=true) → read_B(flag[A]==false)

45

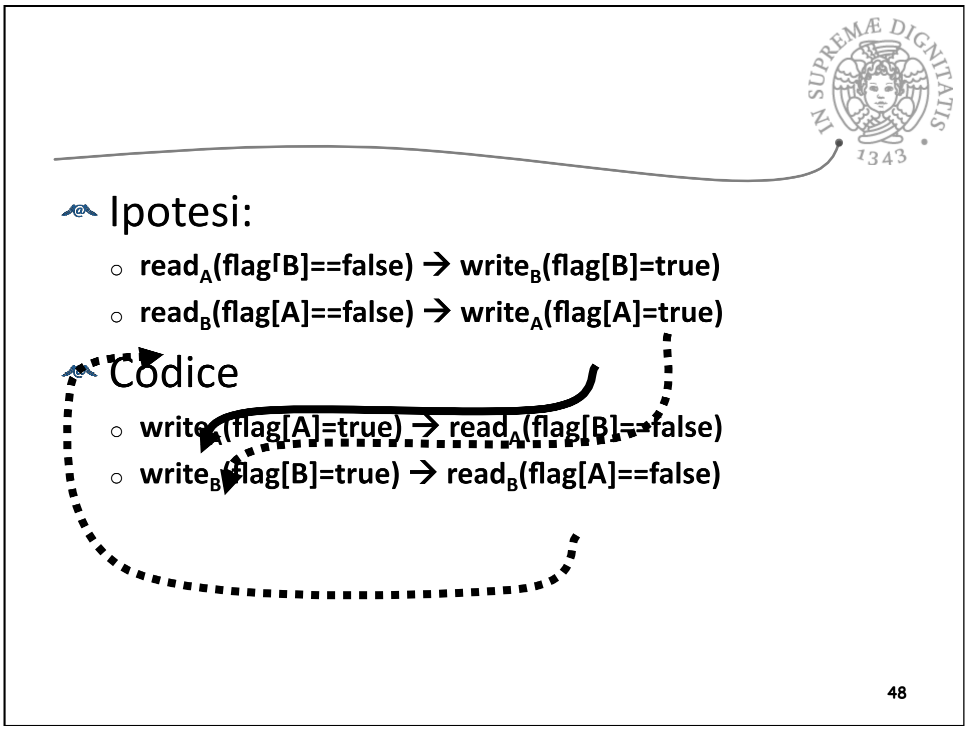
- Ipotesi:
  - read_A(flag[B]==false) → write_B(flag[B]=true)
  - read_B(flag[A]==false) → write_A(flag[A]=true)
- codice
  - write_A(flag[A]=true) → read_A(flag[B]==false)
  - write_B(flag[B]=true) → read_B(flag[A]==false)

46



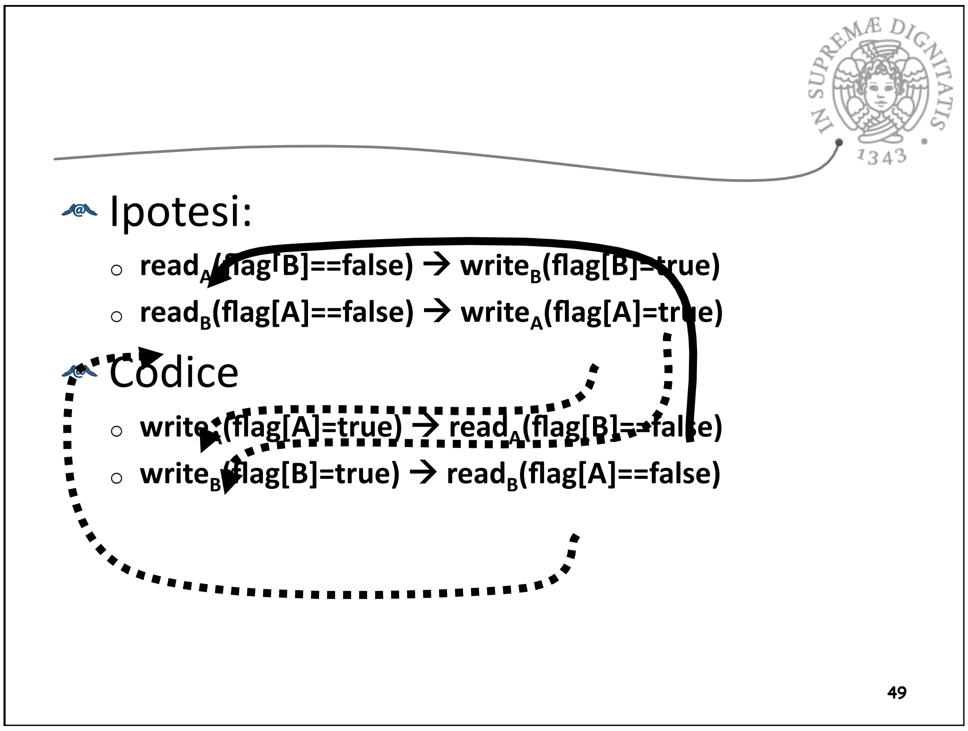
- Ipotesi:
  - read$_A$(flag[B]==false) → write$_B$(flag[B]=true)
  - read$_B$(flag[A]==false) → write$_A$(flag[A]=true)
- Codice
  - write$_A$(flag[A]=true) → read$_A$(flag[B]==false)
  - write$_B$(flag[B]=true) → read$_B$(flag[A]==false)

47



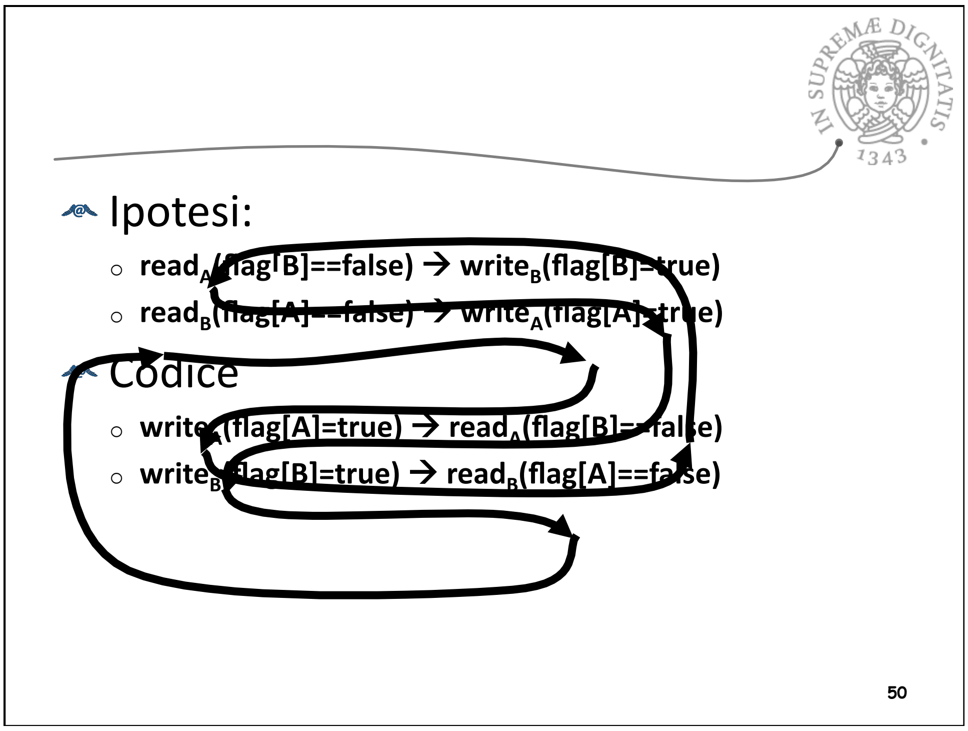
- Ipotesi:
  - read$_A$(flag[B]==false) → write$_B$(flag[B]=true)
  - read$_B$(flag[A]==false) → write$_A$(flag[A]=true)
- Codice
  - write$_A$(flag[A]=true) → read$_A$(flag[B]==false)
  - write$_B$(flag[B]=true) → read$_B$(flag[A]==false)

48



- Ipotesi:
  - **$read_A$(flag[B]==false) → $write_B$(flag[B]=true)**
  - **$read_B$(flag[A]==false) → $write_A$(flag[A]=true)**
- Codice
  - **$write_A$(flag[A]=true) → $read_A$(flag[B]==false)**
  - **$write_B$(flag[B]=true) → $read_B$(flag[A]==false)**

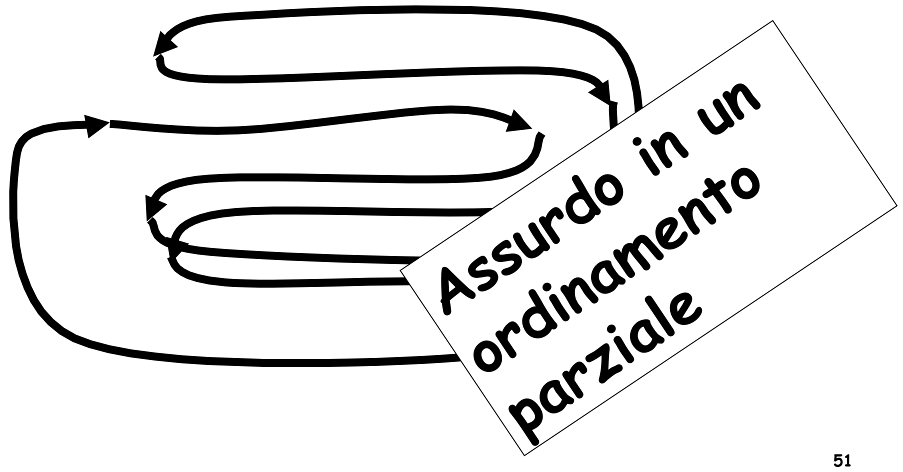
49



- Ipotesi:
  - **$read_A$(flag[B]==false) → $write_B$(flag[B]=true)**
  - **$read_B$(flag[A]==false) → $write_A$(flag[A]=true)**
- Codice
  - **$write_A$(flag[A]=true) → $read_A$(flag[B]==false)**
  - **$write_B$(flag[B]=true) → $read_B$(flag[A]==false)**

50

Otteniamo un ciclo!



Deadlock Freedom

- LockOne non soddisfa la proprieta' di deadlock-freedom
  - Concurrent execution can deadlock

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

LockTwo

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

 public void unlock() {}
}
```

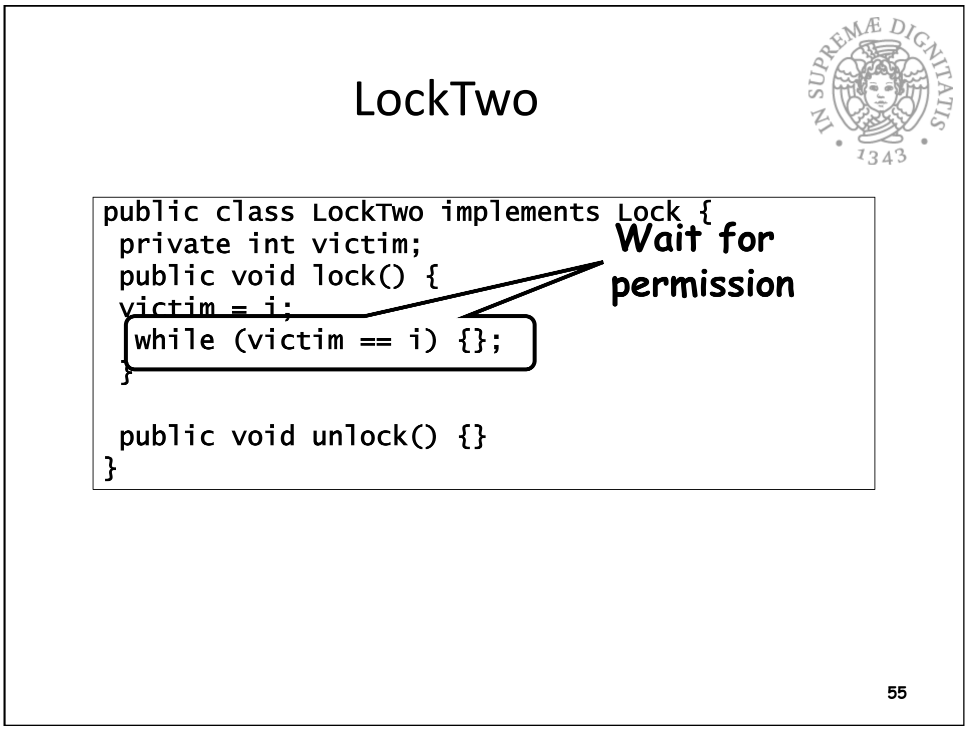
53

LockTwo

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

 public void unlock() {}
}
```

Let other go first

54

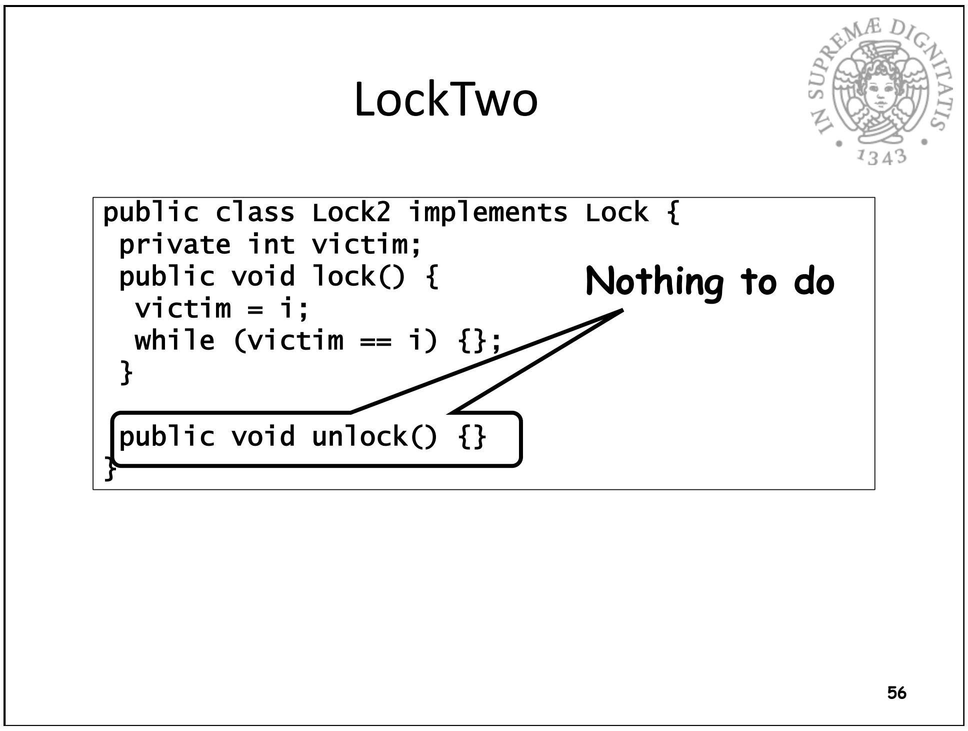


LockTwo

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

 public void unlock() {}
}
```

Wait for permission

55



LockTwo

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

 public void unlock() {}
}
```

Nothing to do

56

LockTwo

- Verifica la mutual exclusion
  - Se il thread `I` e in CS
  - Allora `victim == j`
  - Non puo' avere il valore 0 e 1 allo stesso tempo!!
- Non vala le deadlock free
  - Sequential deadlock
  - Concurrent no

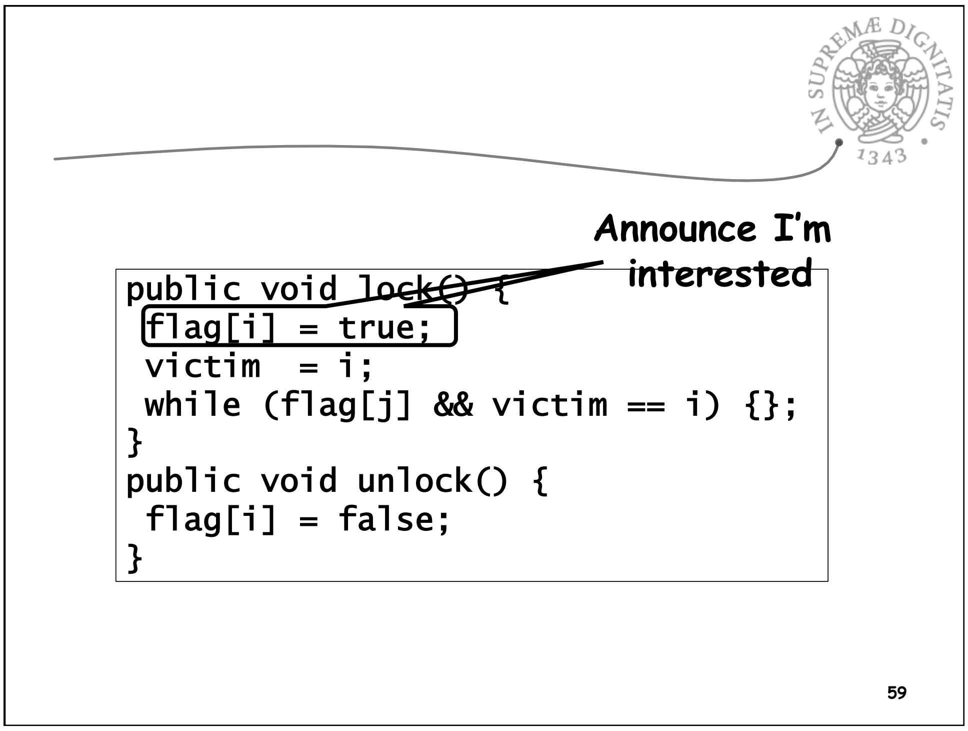
```
public void LockTwo() {
  victim = i;
  while (victim == i) {};
 }
```

57

Algoritmo di Peterson

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

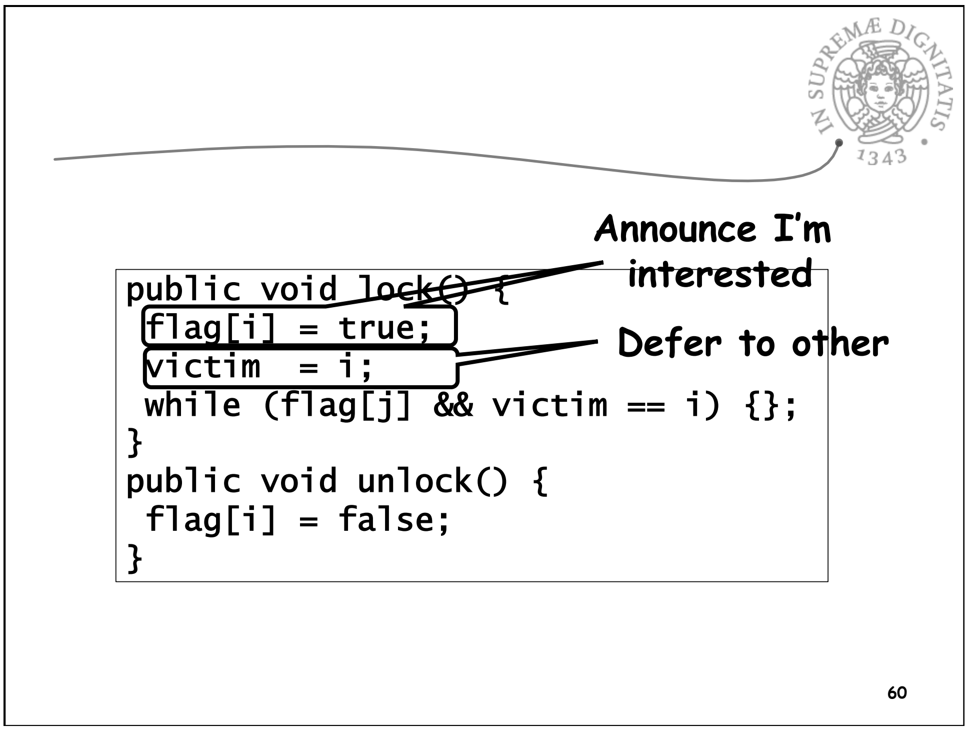
58



Announce I'm interested

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

59

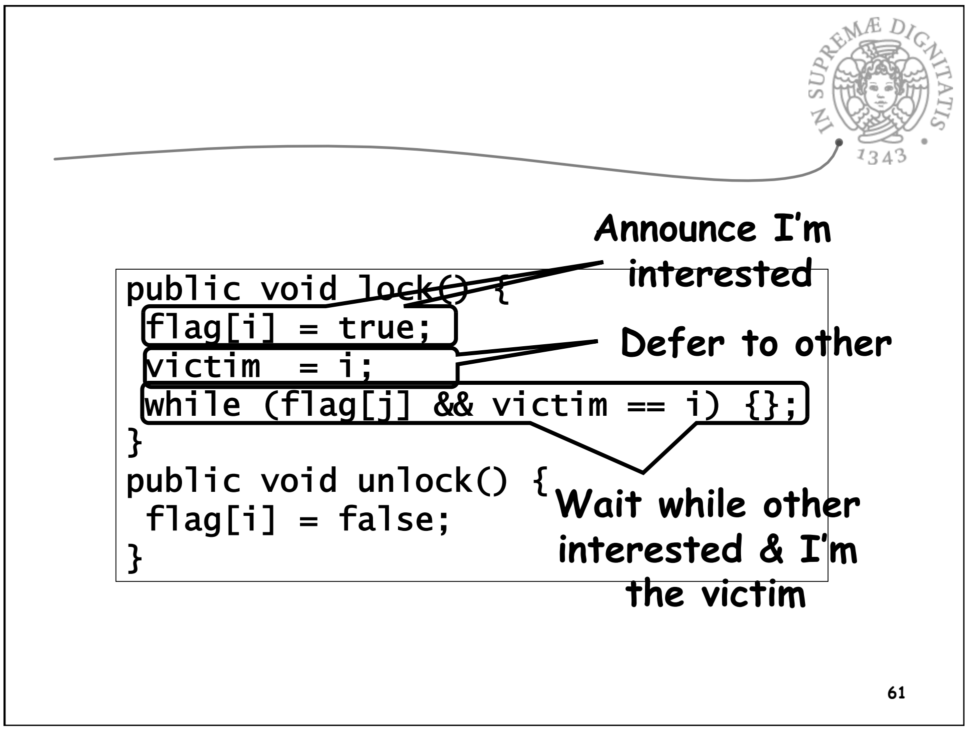


Announce I'm interested

Defer to other

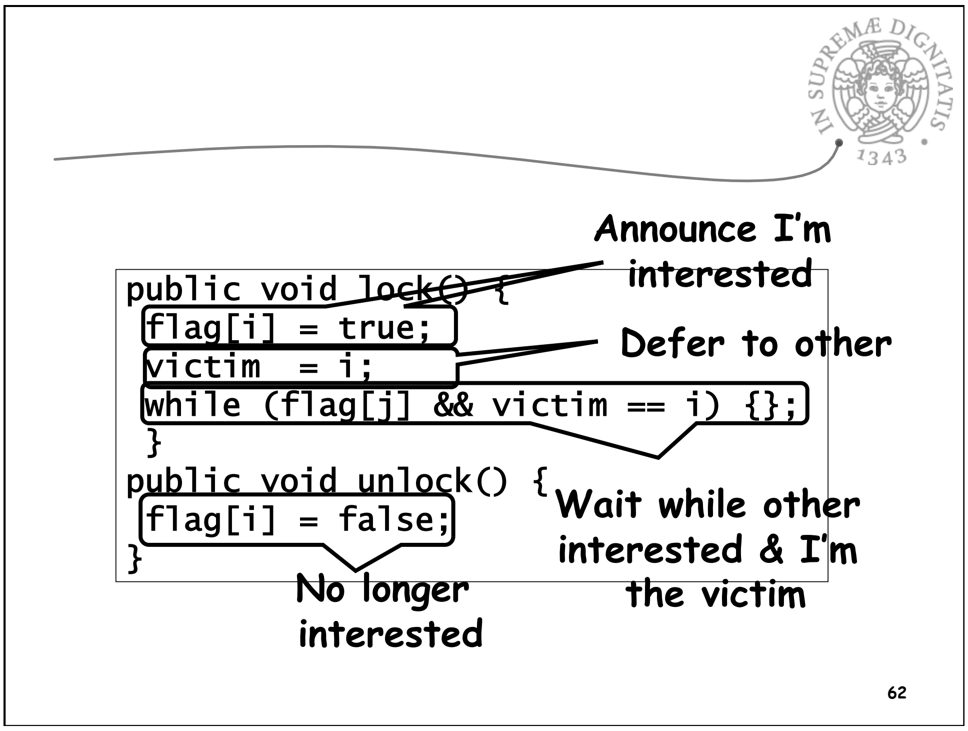
```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

60



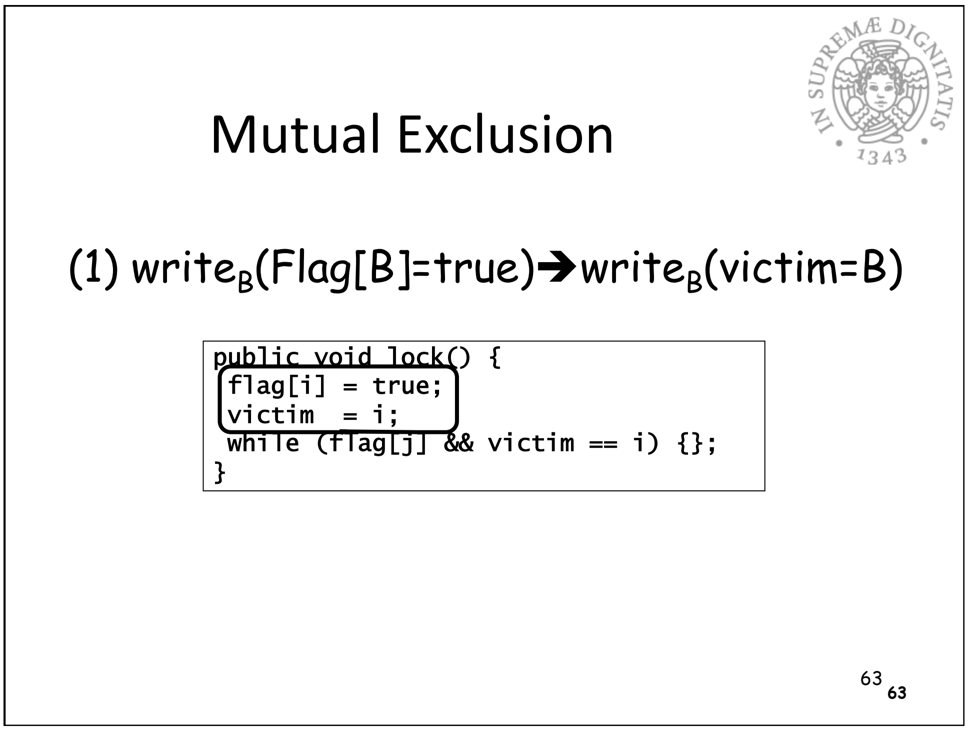
```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested
Defer to other
Wait while other interested & I'm the victim
61



```
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
62

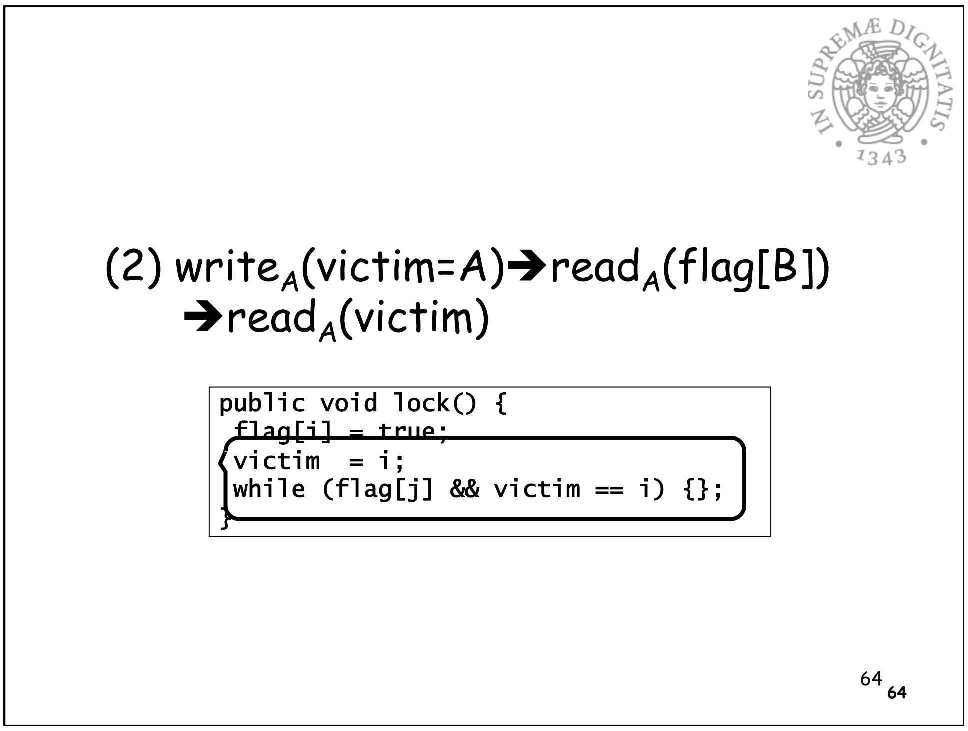


Mutual Exclusion

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

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

63 63



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

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

64 64

Ipotesi

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

65 65

Mescoliamo ben bene

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

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

(2) $\boxed{}$ $\boxed{\rightarrow \text{read}_A(\text{flag[B]})}$

$\boxed{\rightarrow \text{read}_A(\text{victim})}$

Pertanto, A read flag[B] == true
e
victim == A, non intra in CS
QED

66 66

Deadlock Free

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

- Il thread viene bloccato
  - `while` loop
  - Se il flag dell'atro thread e' true
  - Solo se e' la "vittima" **victim**
- Una sola flag ha valore false

67 67

Starvation Free

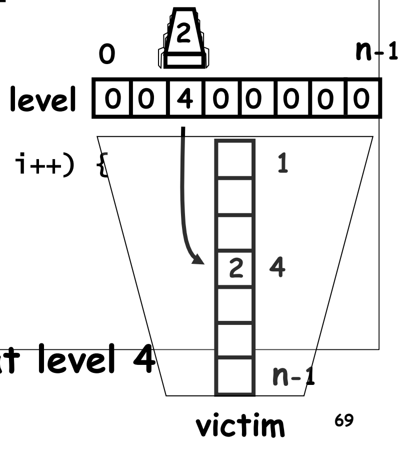
- Thread `i` e' bloccato solo se `j` esegue sempre
- `flag[j] == true` and `victim == i`
- Ma se `j` entra nuovamente
  - `victim` diventa `j`.
  - Pertanto entra `i`

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

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

68 68

Filter

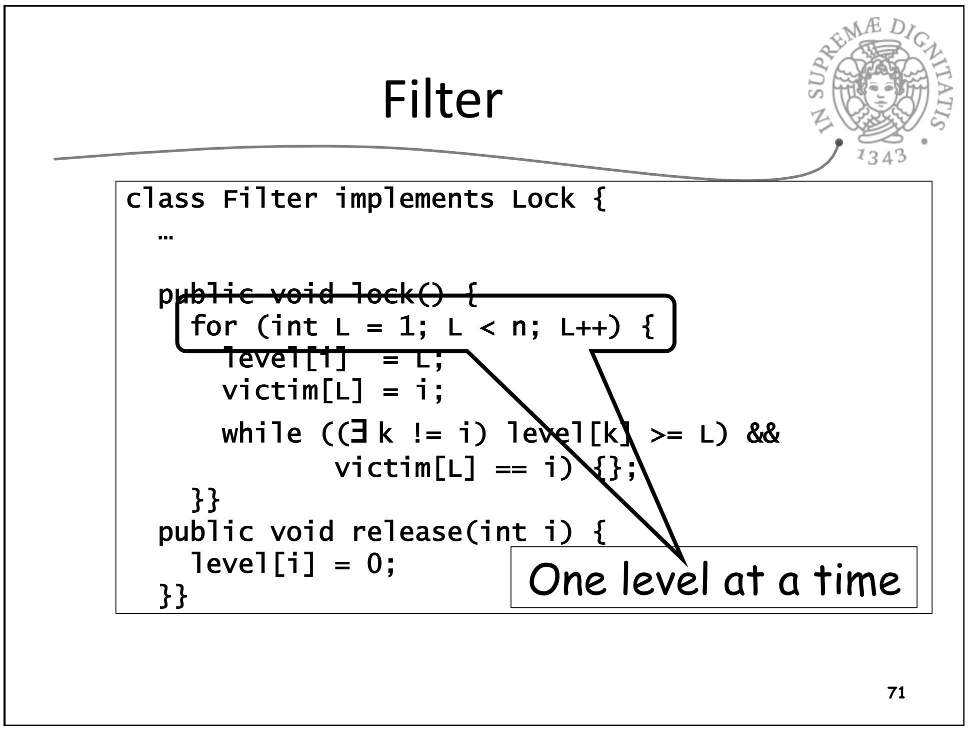
```
class Filter implements Lock {
   int[] level;  // level[i] for thread i
   int[] victim; // victim[L] for level L

  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;
  }}
```

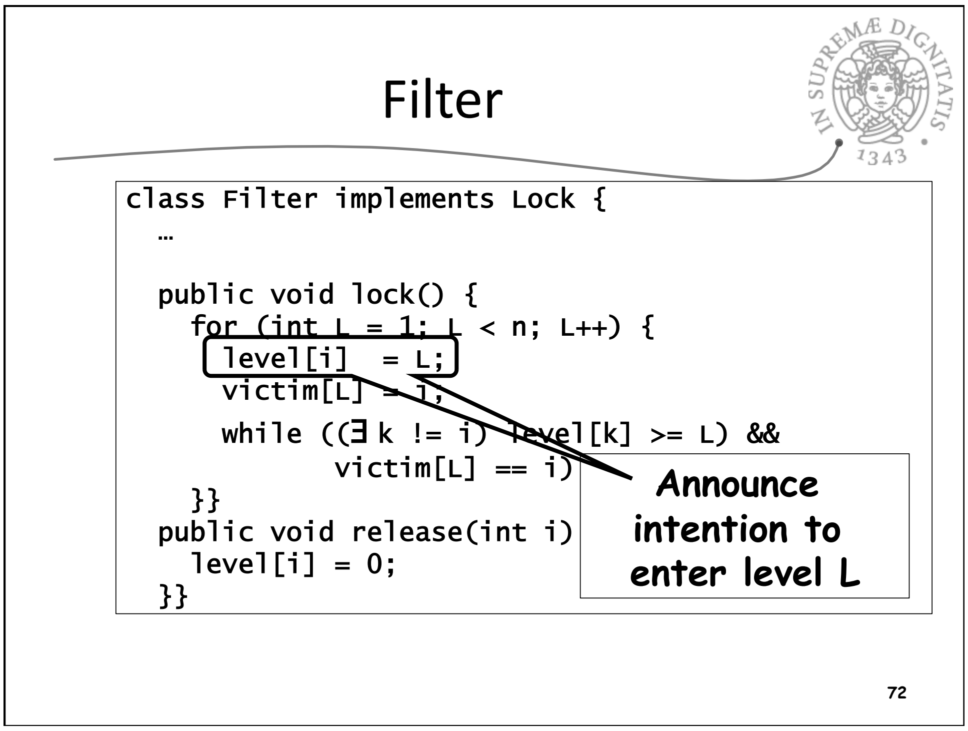


Filter

```
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

71

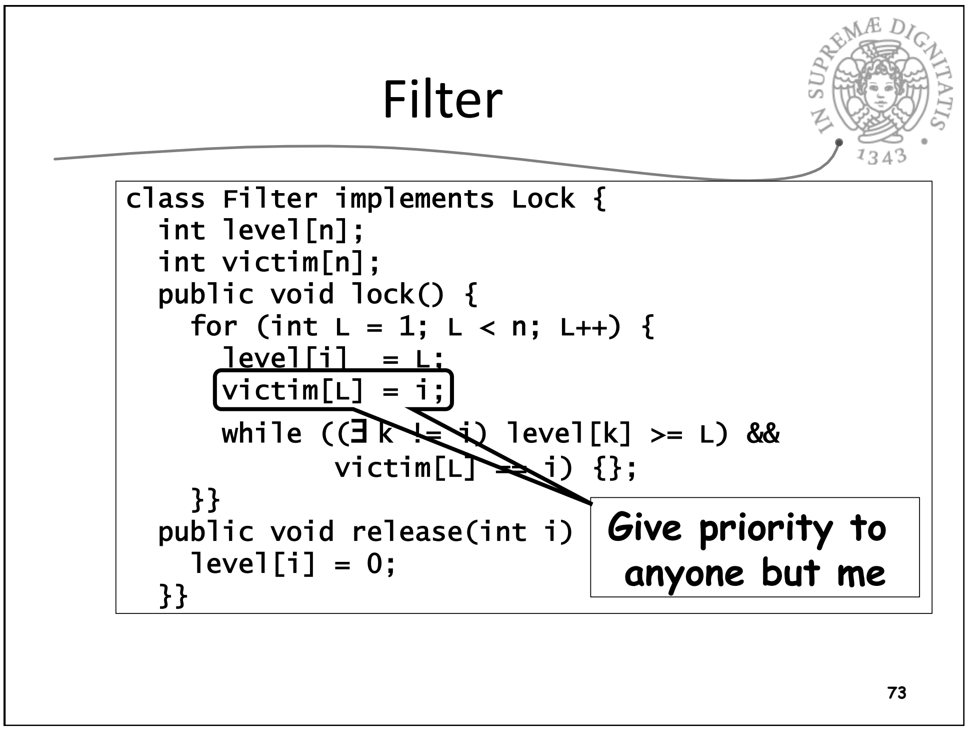


Filter

```
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

72

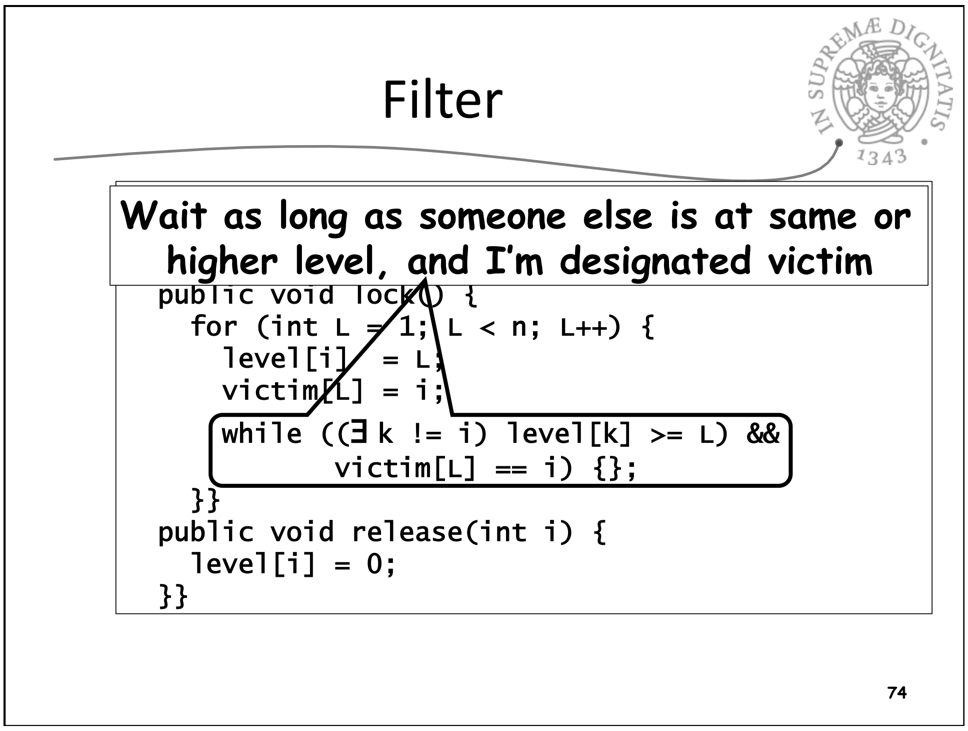


Filter

```
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

73

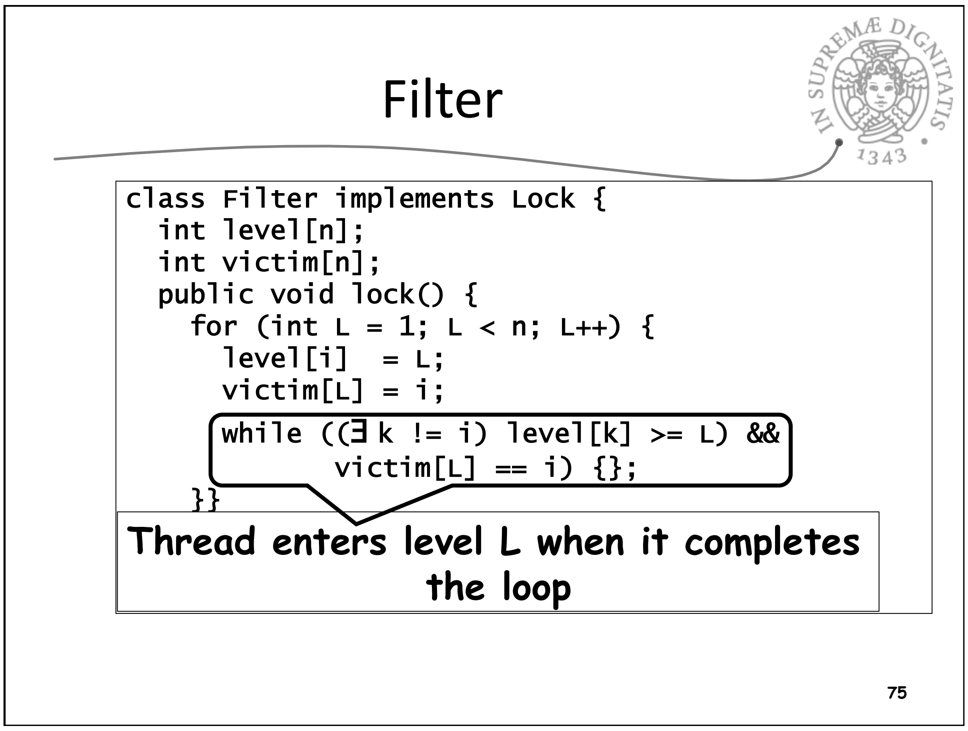


Filter

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;
  }}
```

74

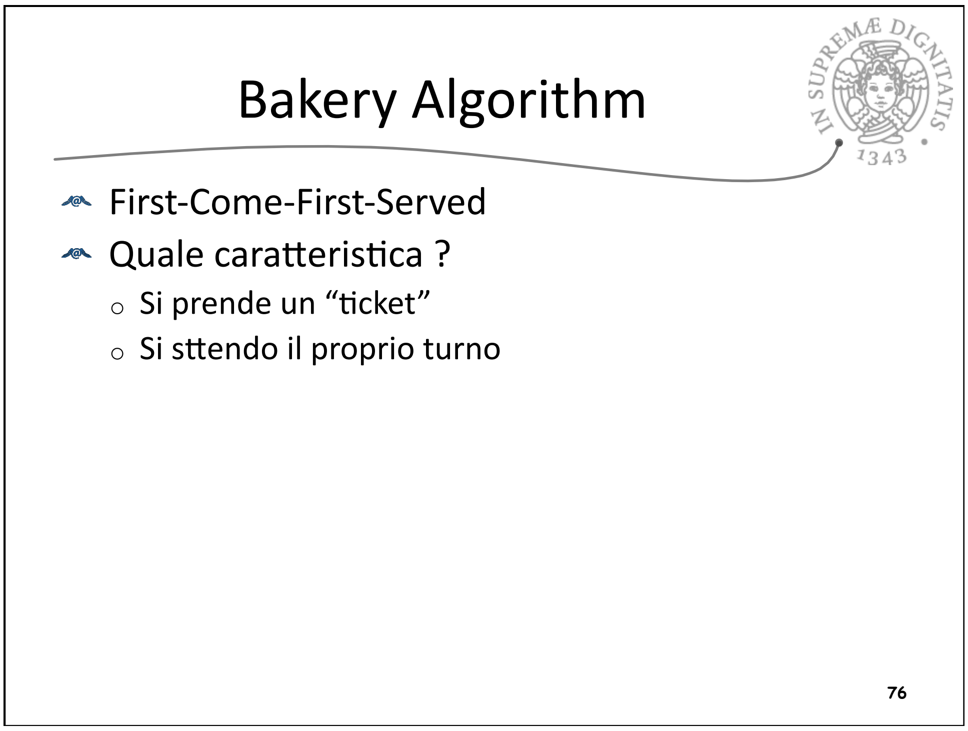


Filter

```
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

75



Bakery Algorithm

- First-Come-First-Served
- Quale caratteristica ?
  - Si prende un “ticket”
  - Si sttendo il proprio turno

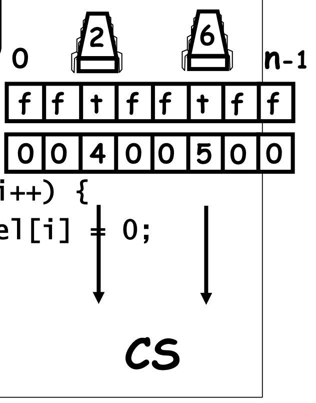
76

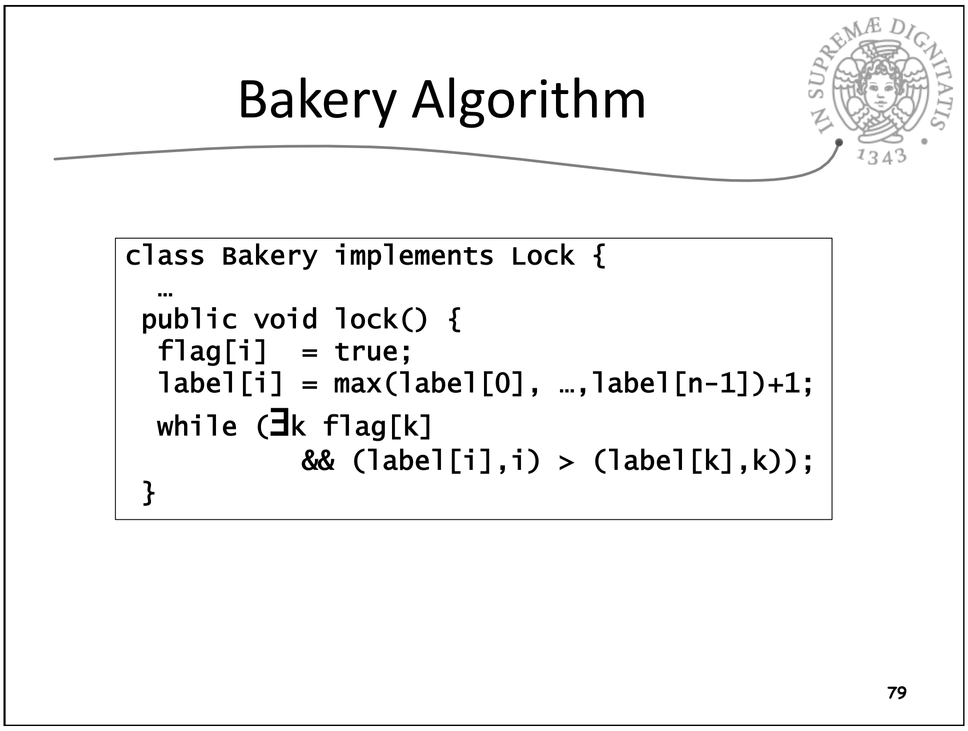
Bakery Algorithm

```
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;
    }
  }
 …
```

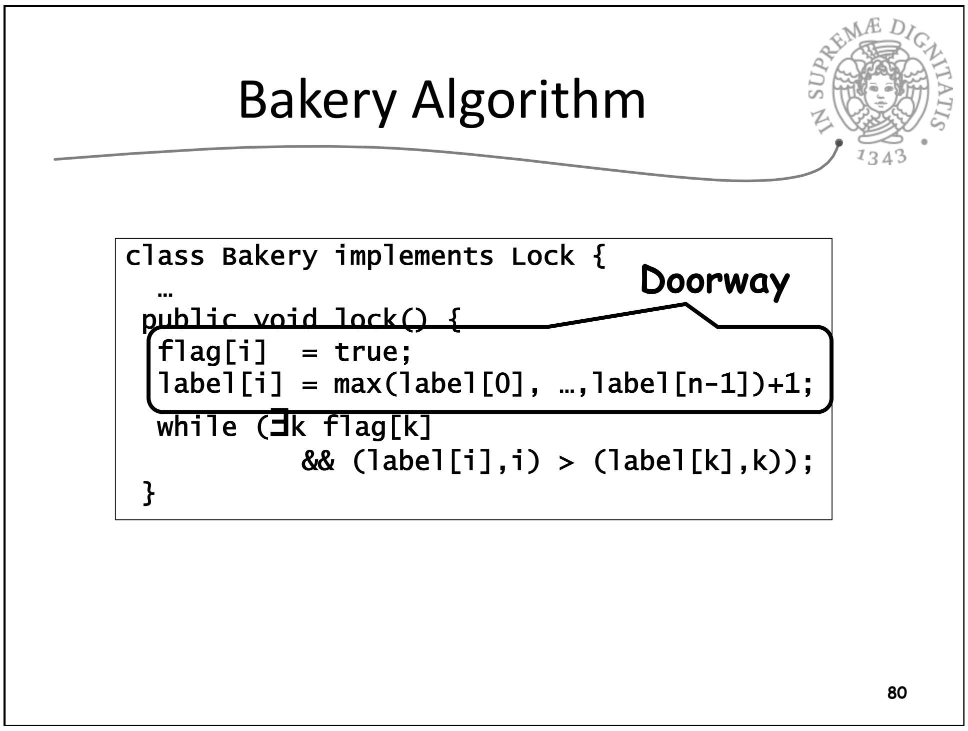




Bakery Algorithm

```
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));
 }
```

79



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));
 }
```

80

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

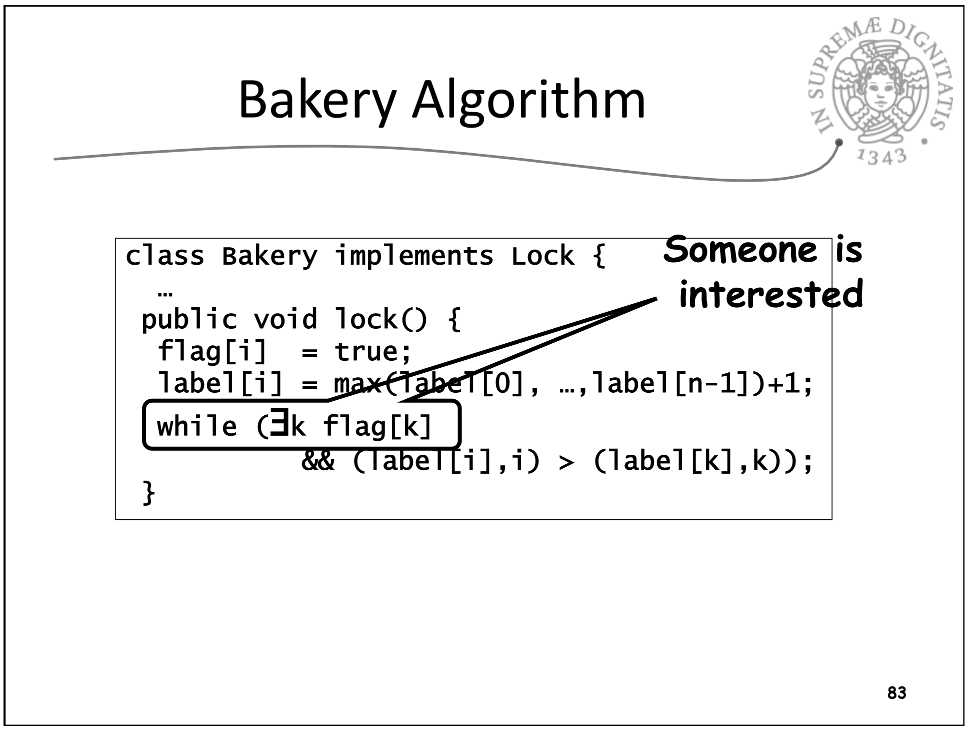
81

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)

82

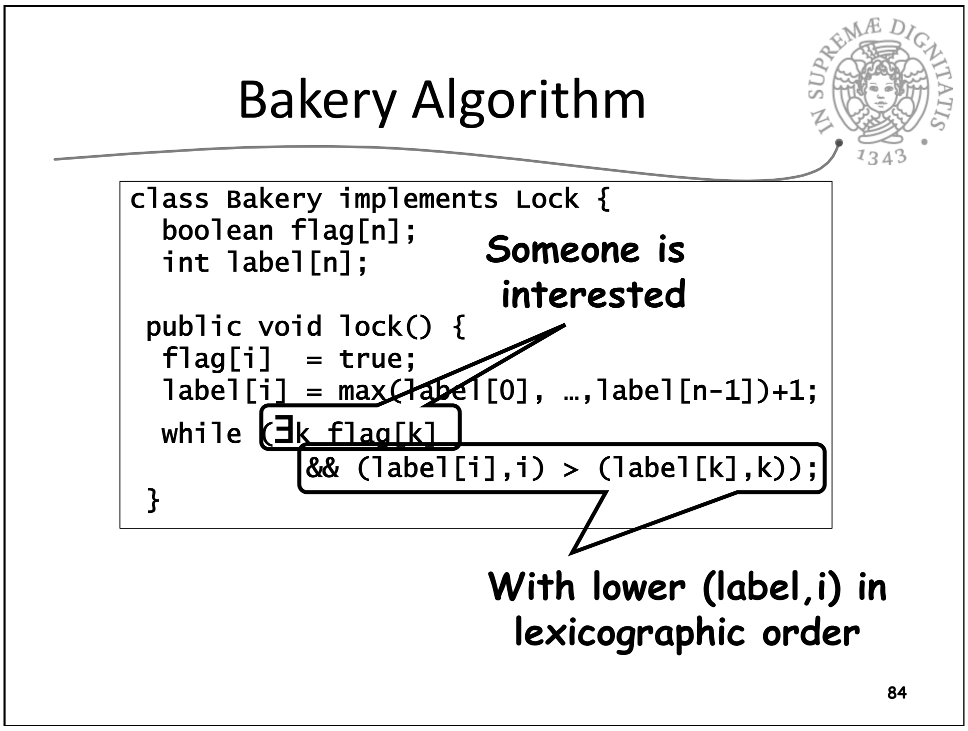


Bakery Algorithm

```
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

83



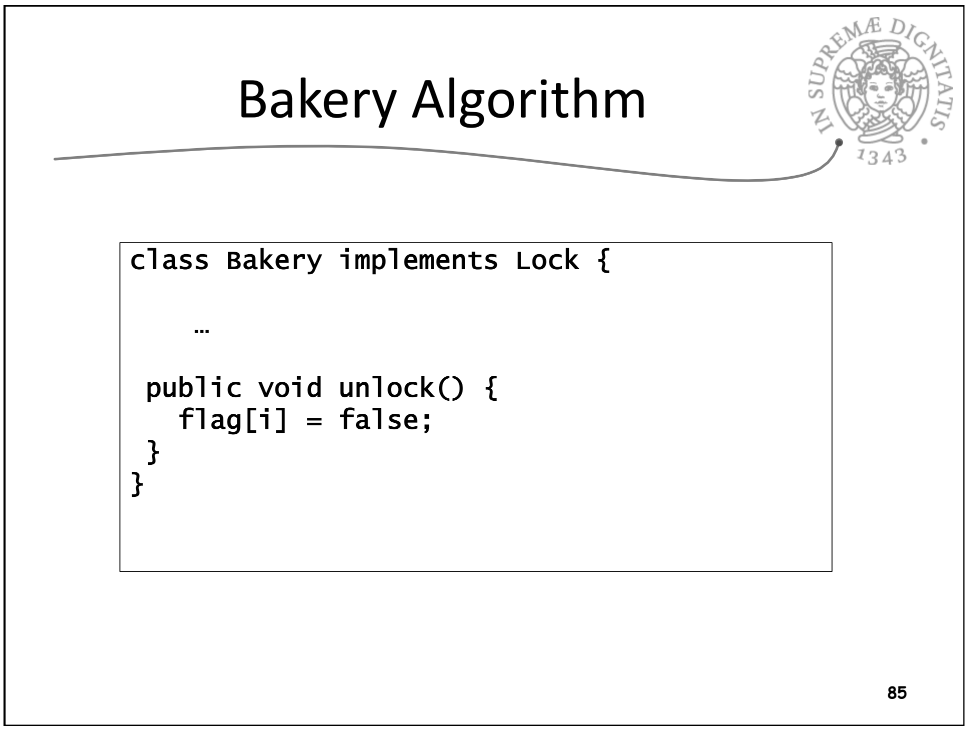
Bakery Algorithm

```
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

With lower (label,i) in lexicographic order

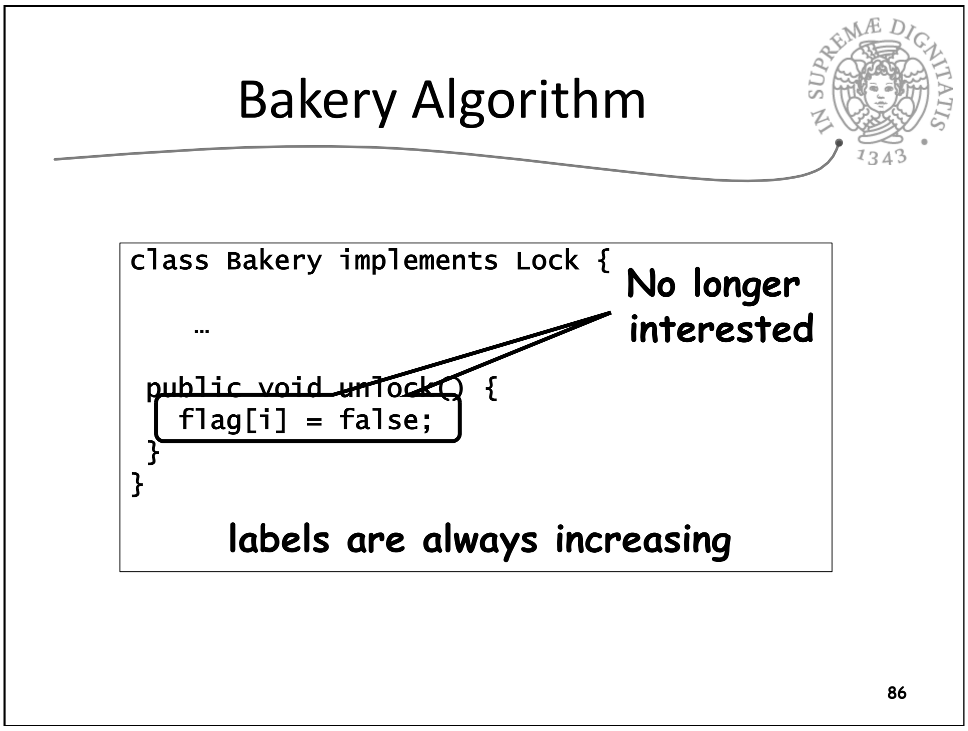
84



Bakery Algorithm

```
class Bakery implements Lock {

    …

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

85



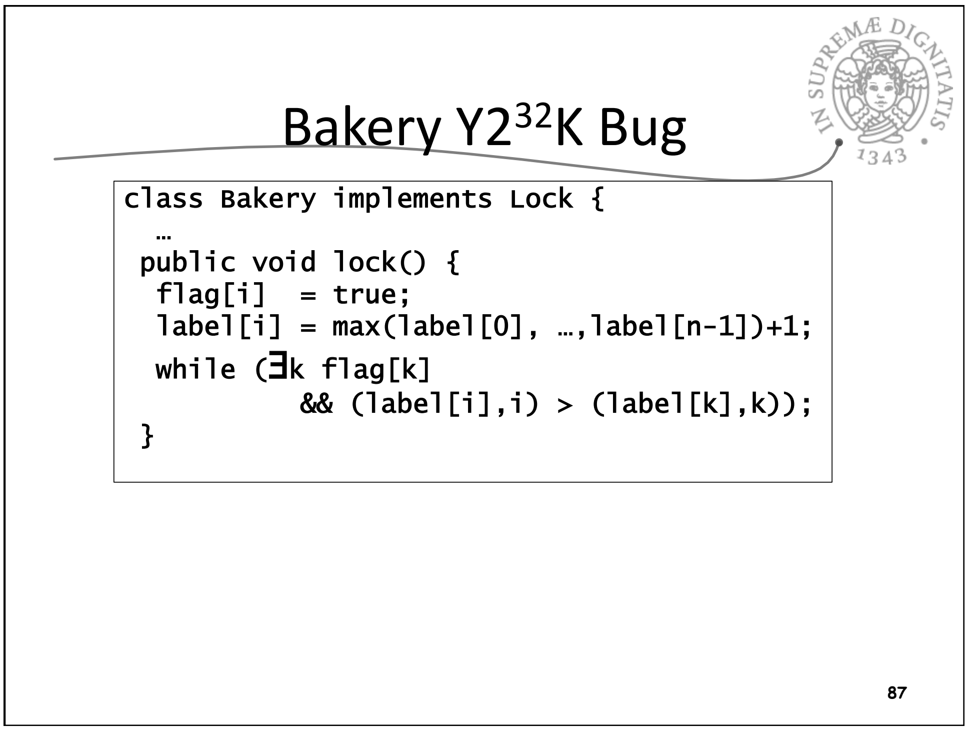
Bakery Algorithm

```
class Bakery implements Lock {

    …

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

No longer interested

labels are always increasing

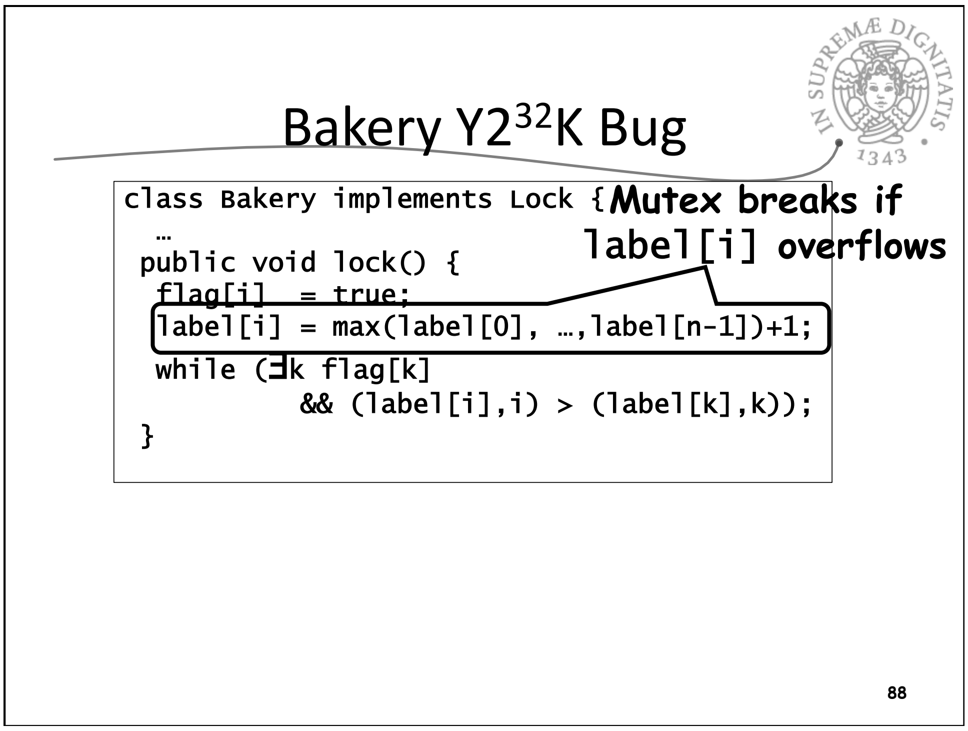
86



Bakery Y2^{32}K Bug

```
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));
 }
```

87



Bakery Y2^{32}K Bug

```
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));
 }
```

Mutex breaks if `label[i]` overflows

88