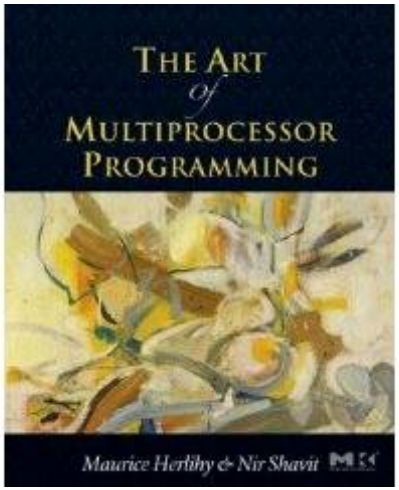
PROGRAMMAZIONE CONCORRENTE



The Art of Multiprocessor Programming

Maurice Herlihy & Nir Shavit

Mutual Exclusion

- Problema essenziale della programmazione concorrente
- Come si dimostrano proprieta' di astrazioni in presenza di thread

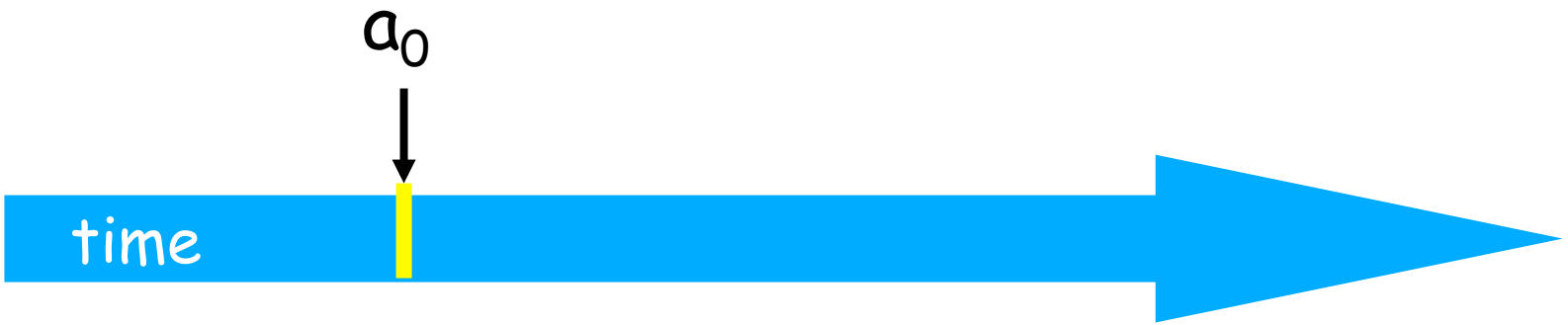
Mutual Exclusion

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."

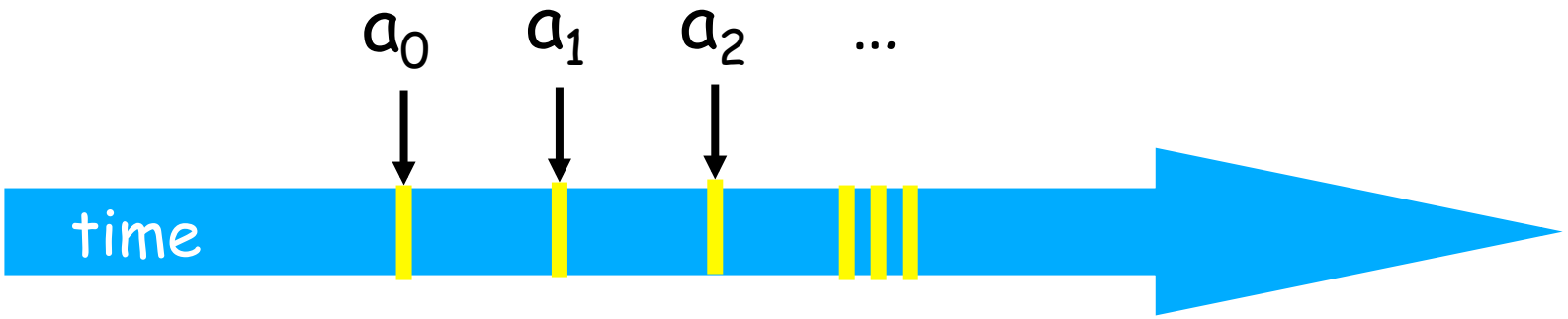
Scala temporale

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



Thread (visione astratta)

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

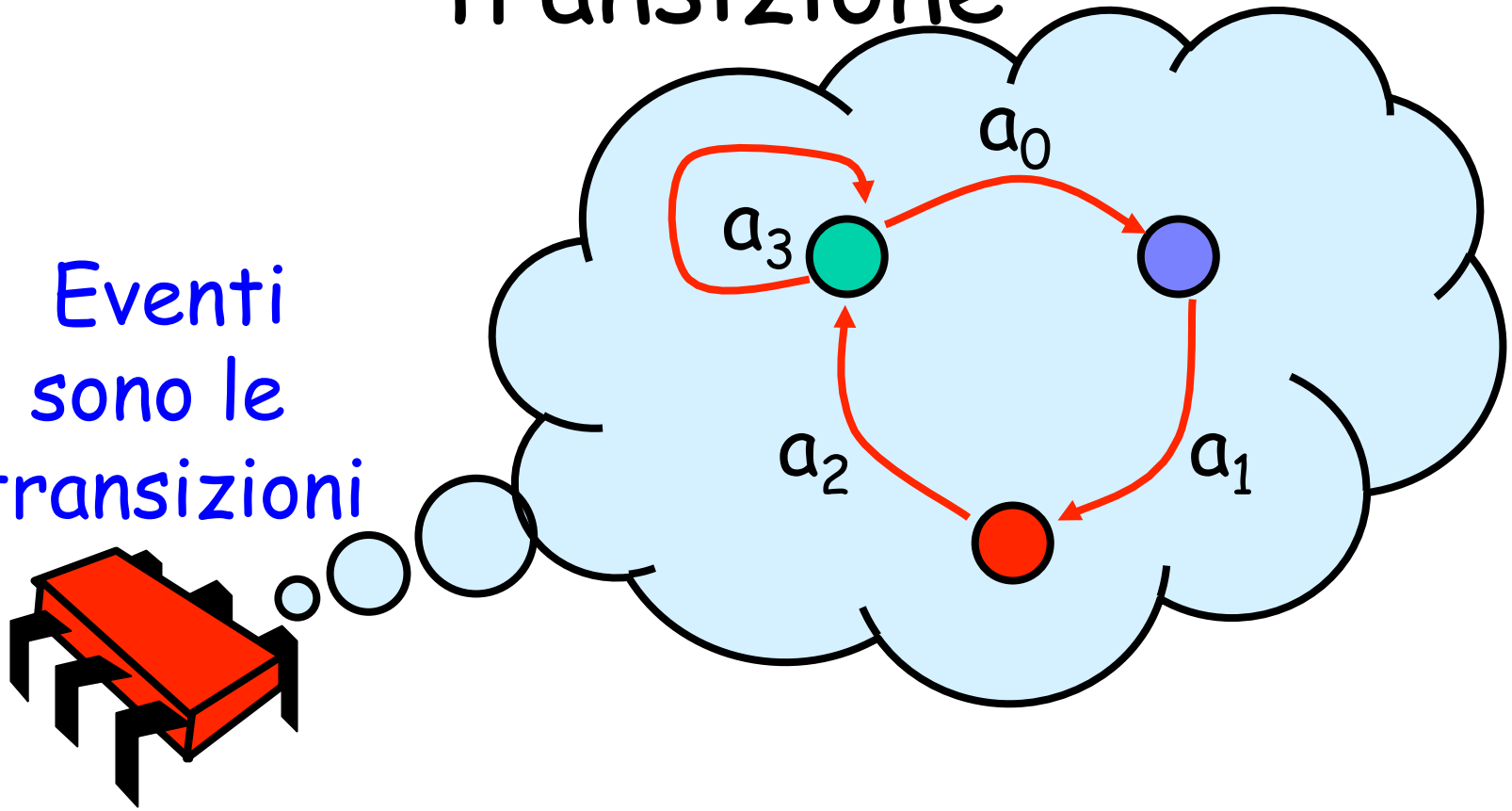


Cosa sono gli eventi?

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

Threads sono sistemi di transizione

Eventi sono le transizioni



Stati

- Thread State
  - Informazioni di controllo
  - Informazioni di ambiente
- System state
  - Heap con gli oggetti condivisi
  - Stato di tutti i thread in esecuzione

Concorrenza

- Thread A

Concorrenza

- Thread A

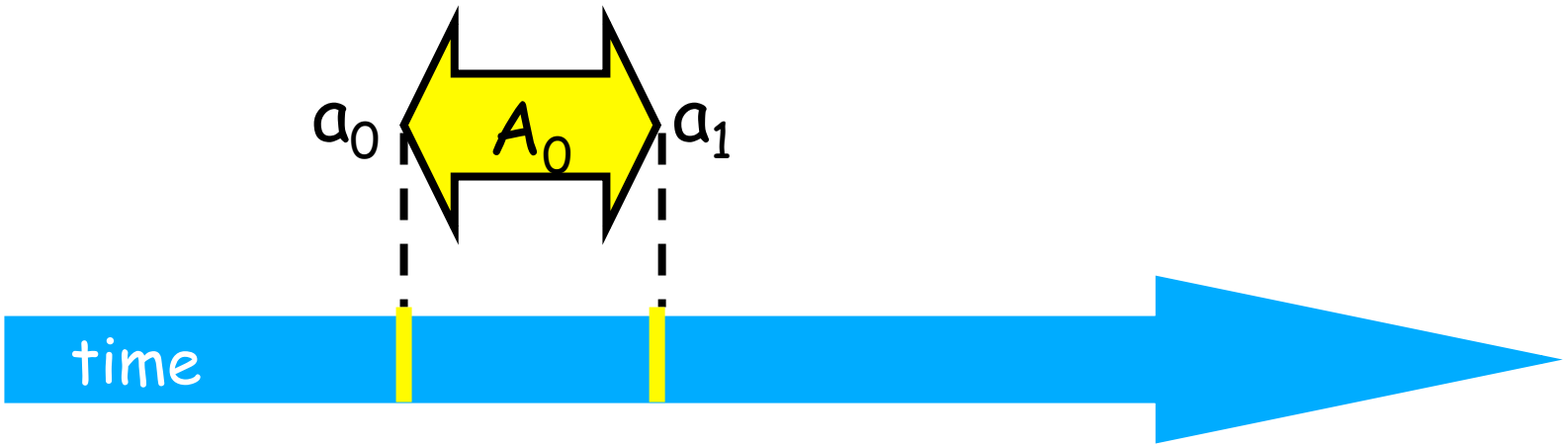
- Thread B

Interleaving

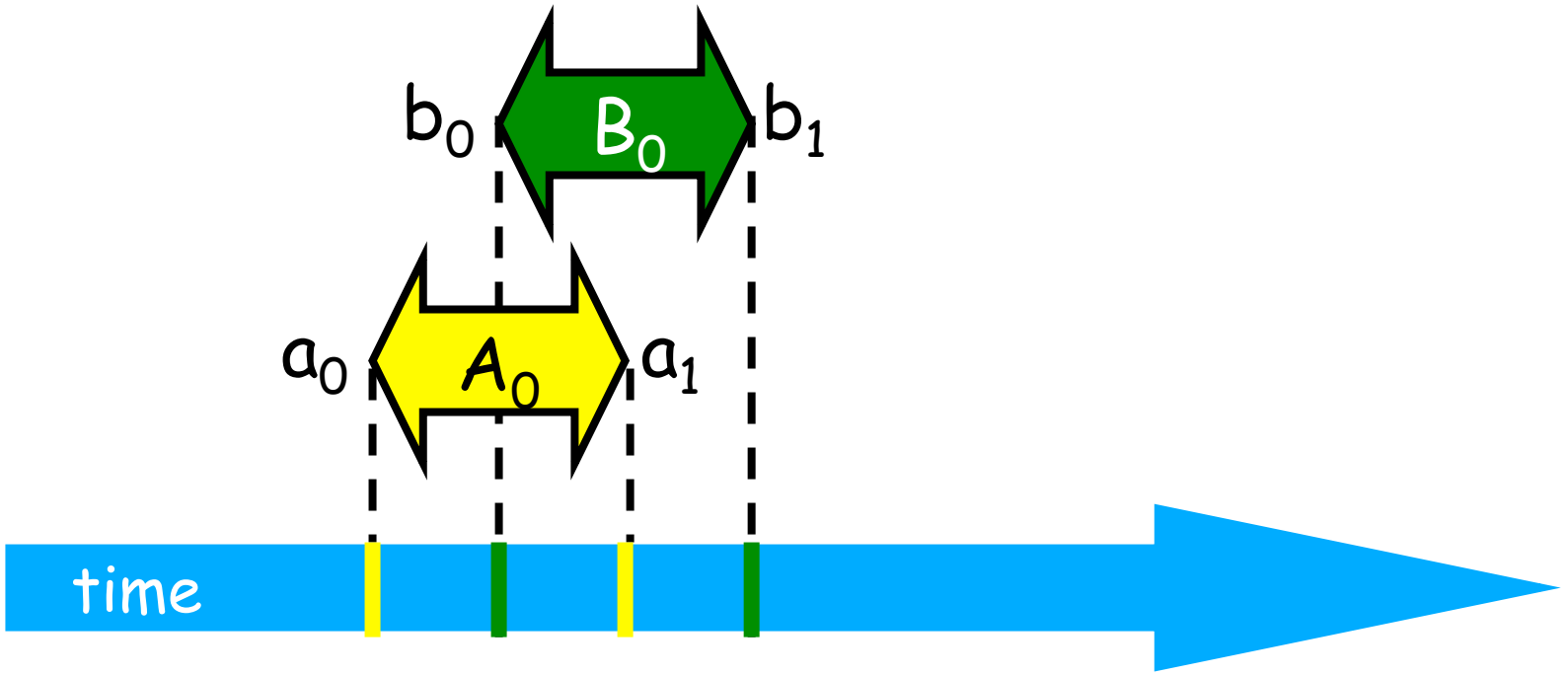
- Eventi dei thread
  - Intercanalo
  - Non sono sempre indipendenti

Intervallo

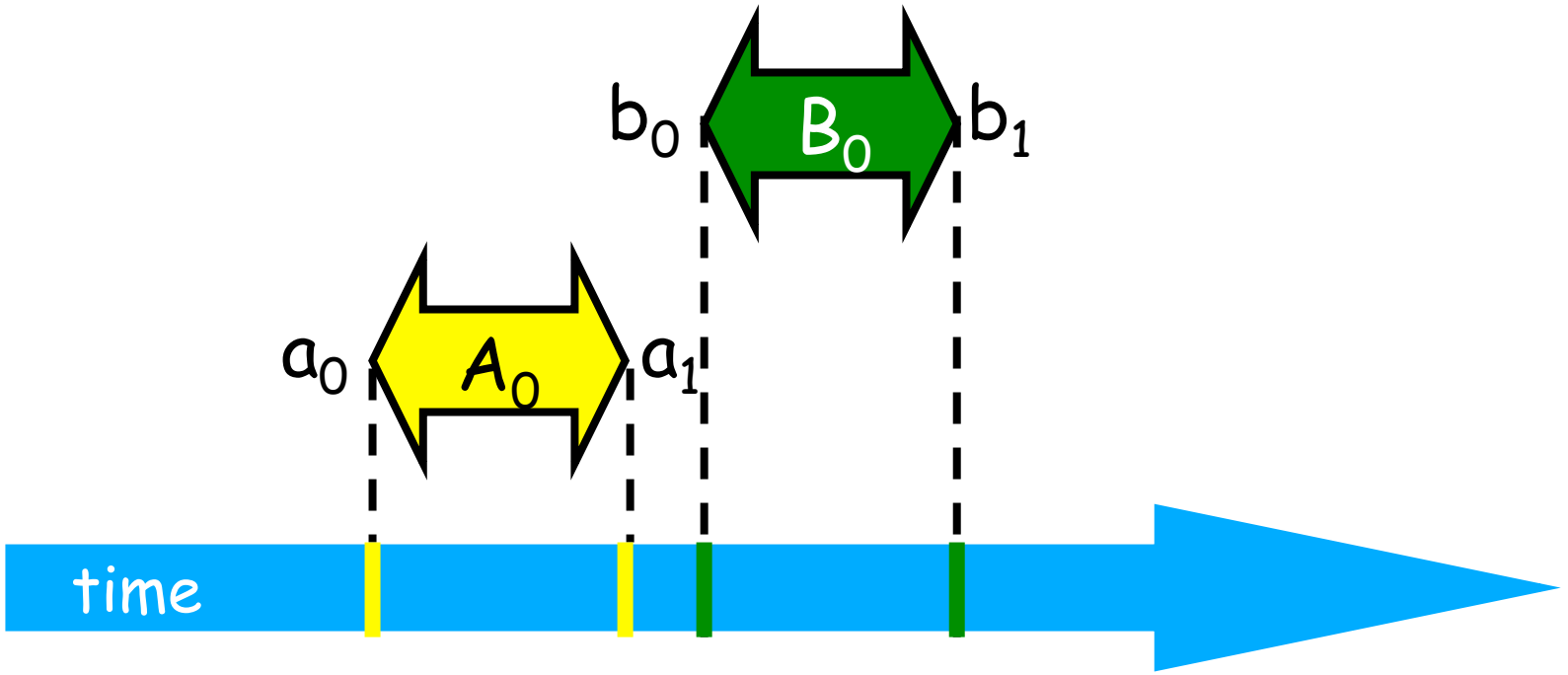
- Un *intervallo* $A_0 = (a_0, a_1)$ e'
  - Periodo di tempo trascorso tra l'occorrenza dell'evento a_0 e l'evento a_1



Intervalli si possono sovrapporre

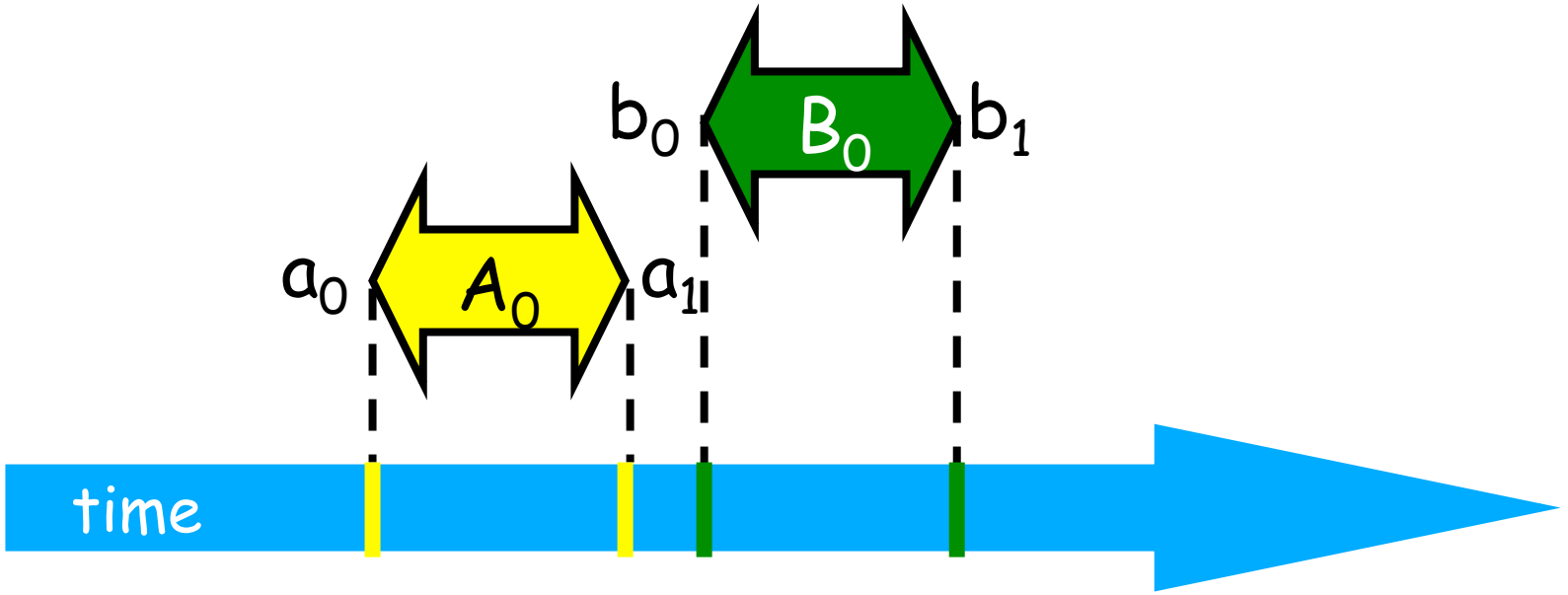


Intervalli possono essere disgiunti

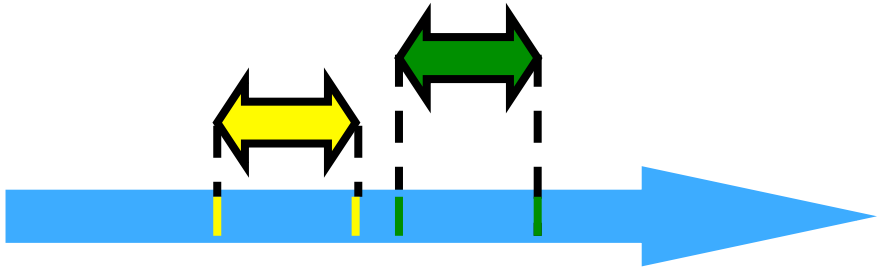


Precedenza

Intervallo A_0 precede intervallo B_0

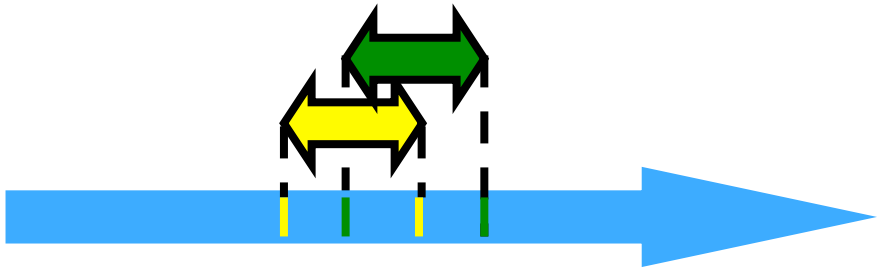


Ordinamento



- 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

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

Ordinamento parziale

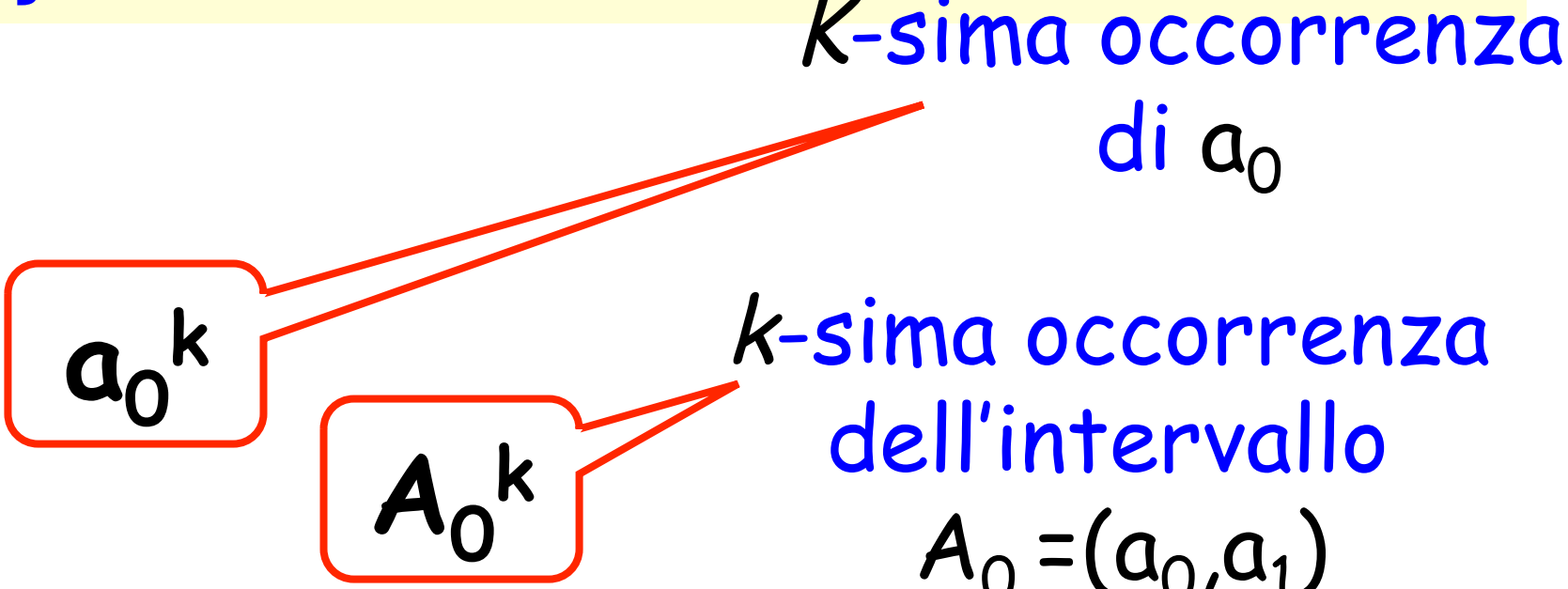
- Irriflessivo:
  - Non vale $A \rightarrow A$
- Antisimmetrico:
  - $A \rightarrow B$ non implica che $B \rightarrow A$
- Transitivo:
  - Se $A \rightarrow B$ & $B \rightarrow C$ allora $A \rightarrow C$

Ordine totale

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

Ripetizione di eventi

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



Il solito contatore

```
public class Counter {
  private long value;

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

Azione Atomica

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

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

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;
  }}
```

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

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
(no matter what)

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

Il problema della mutua esclusione

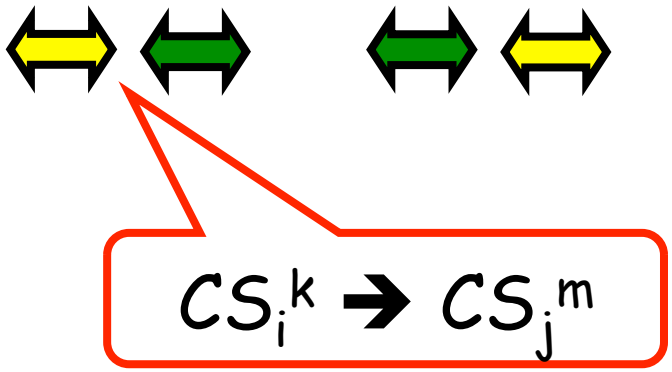
- Con la notazione $CS_i^k \Leftrightarrow$ indichiamo che il thread i sta eseguendo per la k-sima volta sezione critica

Mutua esclusione

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

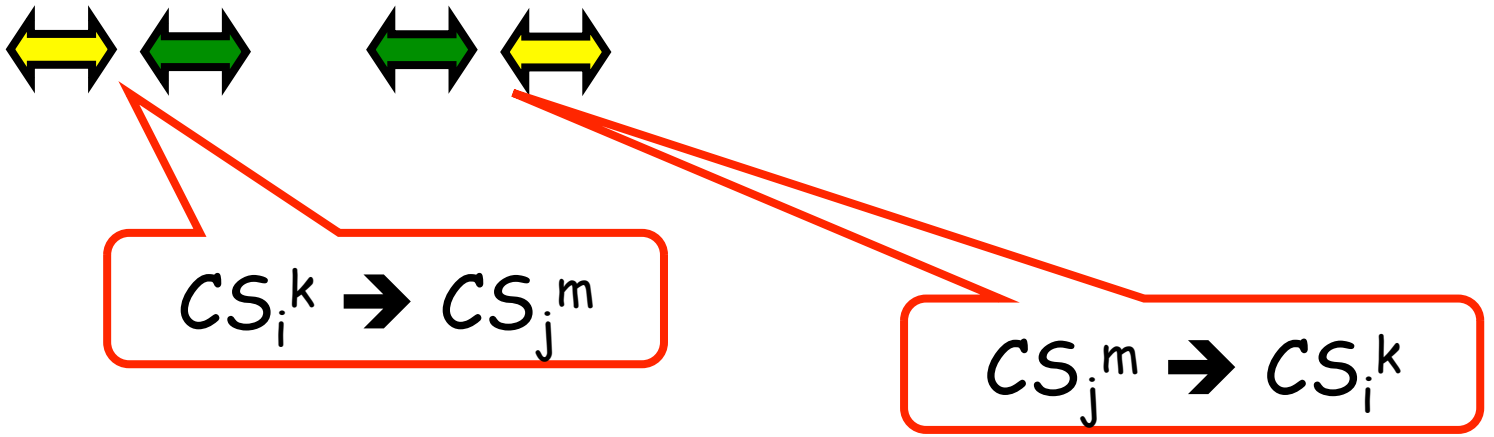
Mutua esclusione

- Cs_i^k ⟺
- CS_j^m ⟺
- *Allora deve accadere*
  - o



Mutua esclusione

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



Deadlock-Free

- Assumiamo che un thread esegue 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"

Starvation-Free

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

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

```
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

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]) {}
 }
```

Each thread has flag

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

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

Analizziamo il codice

- $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false) \rightarrow CS_A$
- $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false) \rightarrow CS_B$

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

Ipotesi

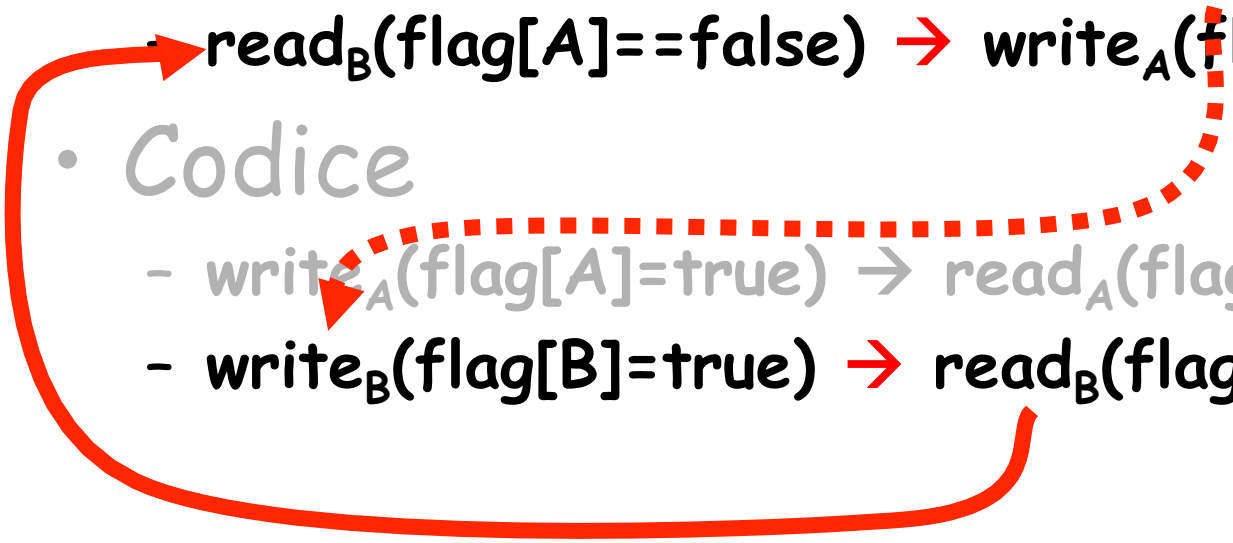
- $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
- $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$

Mettiamo tutto assieme

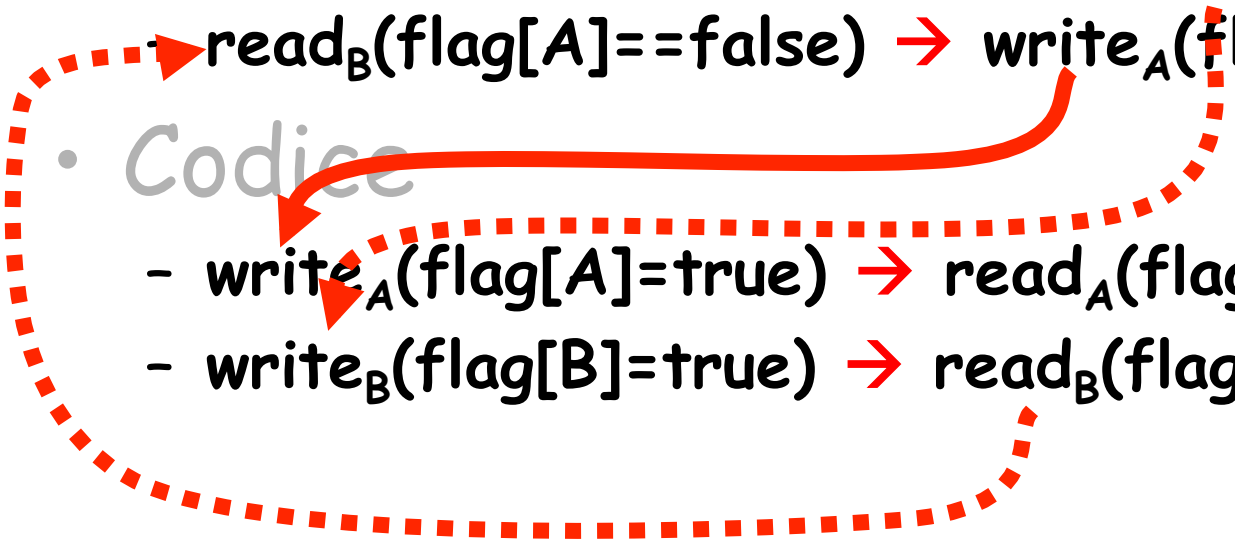
- Ipotesi:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- Il codice
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

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

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

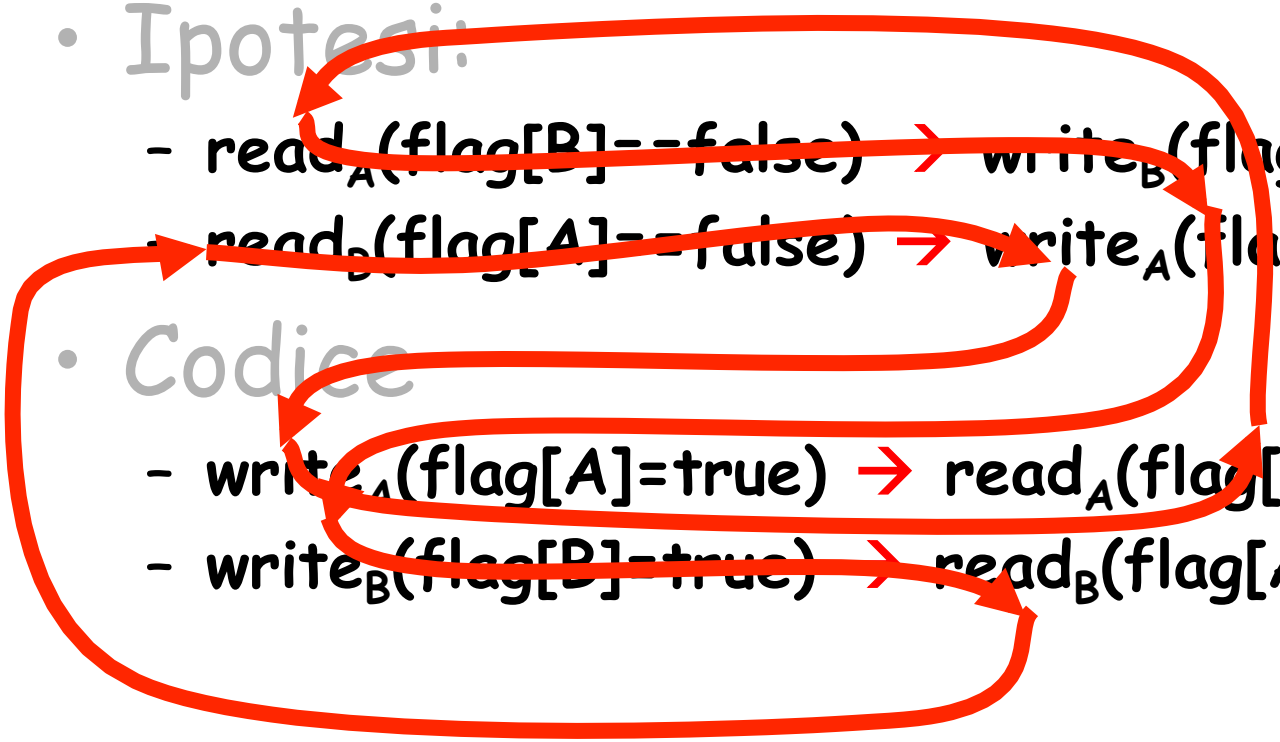


- Ipotesi:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- Codice
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$



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

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



Otteniamo un ciclo!

Deadlock Freedom

- LockOne non soddisfa la proprieta' di deadlock-freedom

```
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() {}
}
```

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

- 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

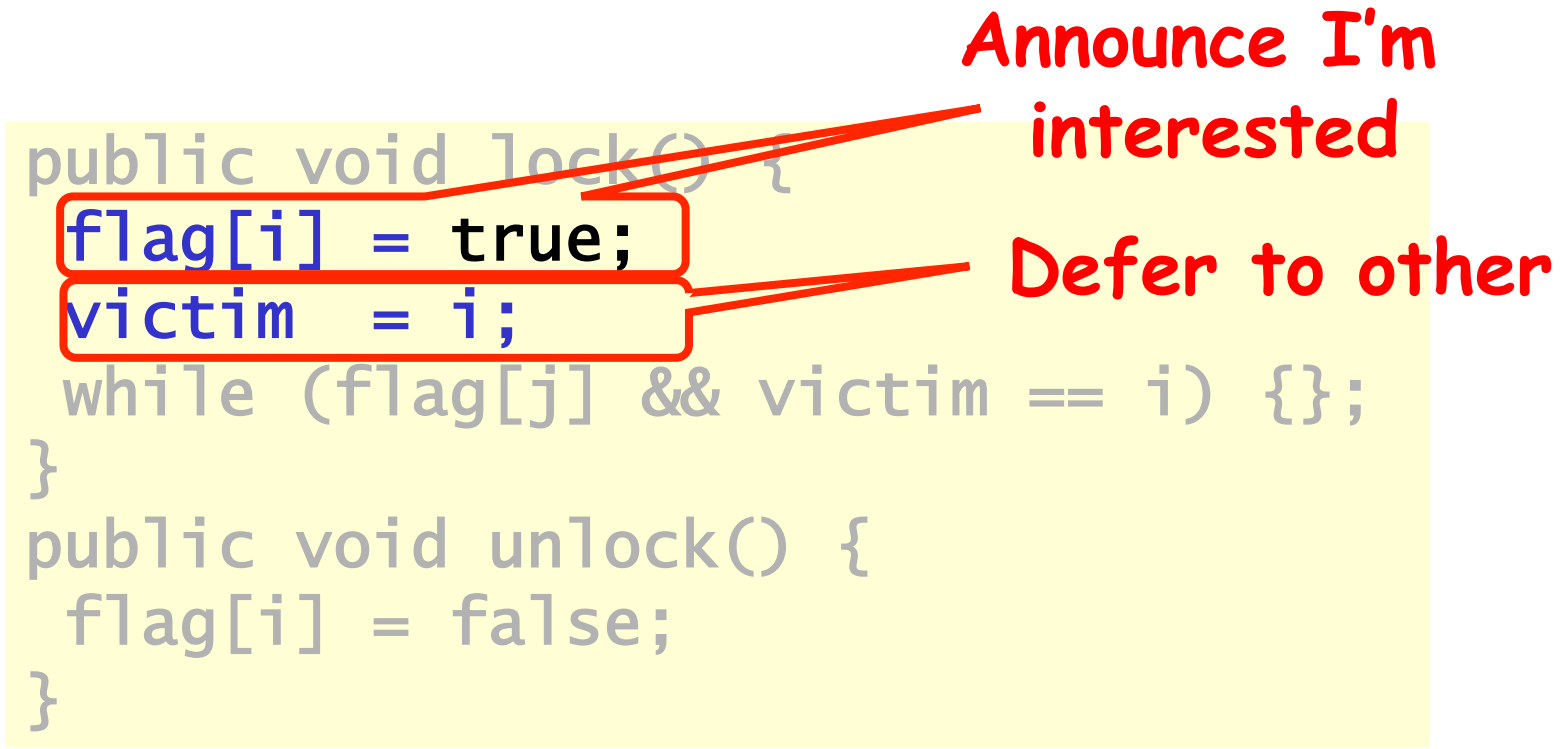
```
public void LockTwo() {
  victim = i;
  while (victim == i) {};
 }
```

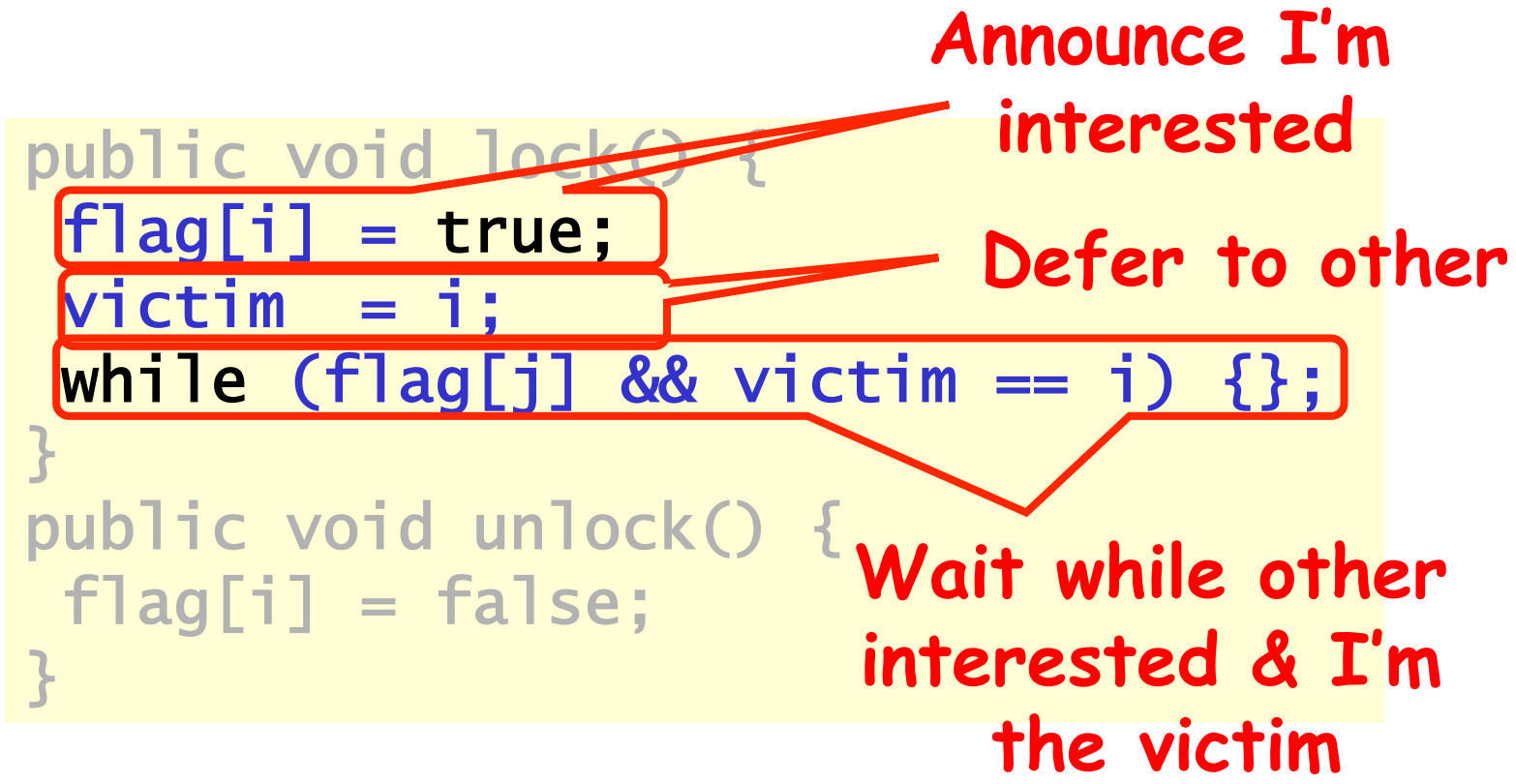
Algoritmo di Peterson

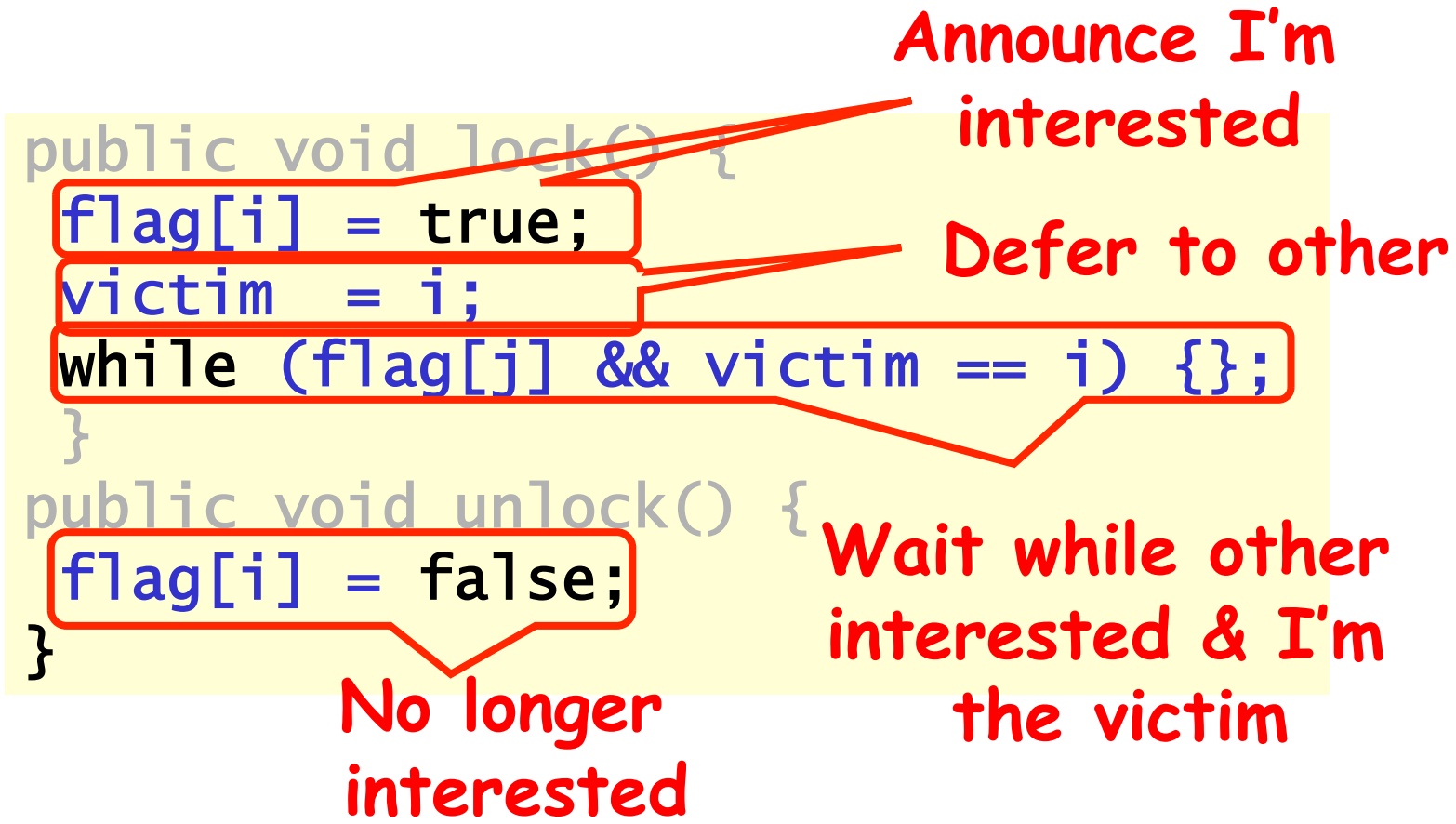
```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested

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


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


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) {};
}
```

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

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

Ipotesi

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

Mescoliamo ben bene

(1) $write_B(flag[B]=true)$ ➔

(3) $write_B(victim=B)$ ➔ $write_A(victim=A)$

(2)

➔ $read_A(victim)$

➔ $read_A(flag[B])$

Pertanto, A read flag[B] == true

e

victim == A, non intra in CS

QED

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

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;
}
```