Mutua Esclusione (1)

Programmazione Concorrente, Parallela e su Cloud

Vittorio Scarano

Dipartimento di Informatica
Università di Salerno

Laurea Magistrale in Informatica

Presentazione

La motivazione

- La Mutua Esclusione è forse una delle forme più importanti di coordinamento nella programmazione concorrente

La motivazione

- La Mutua Esclusione è forse una delle forme più importanti di coordinamento nella programmazione concorrente
- E' necessario presentare gli algoritmi classici di ME, in maniera costruttiva

La motivazione

- La Mutua Esclusione è forse una delle forme più importanti di coordinamento nella programmazione concorrente
- E' necessario presentare gli algoritmi classici di ME, in maniera costruttiva
- Necessario provare la correttezza ...

La motivazione

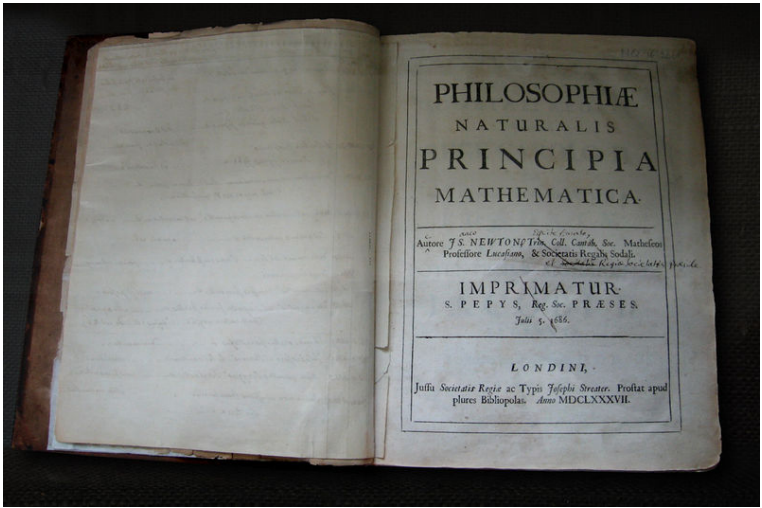
- La Mutua Esclusione è forse una delle forme più importanti di coordinamento nella programmazione concorrente
- E' necessario presentare gli algoritmi classici di ME, in maniera costruttiva
- Necessario provare la correttezza ...
- ...e in alcuni casi la impossibilità !

Warning! Proof ahead!

> rithms that appear in later chapters. This chapter is one of the few that contains proofs of algorithms. Though the reader should feel free to skip these proofs, it is very helpful to understand the kind of reasoning they present, because we can use the same approach to reason about the practical algorithms considered in later chapters.

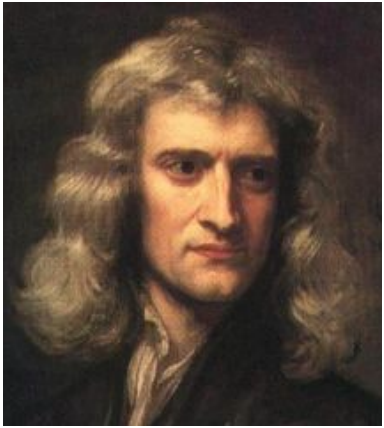
Plan

PRINCIPIA MATHEMATICA DI NEWTON



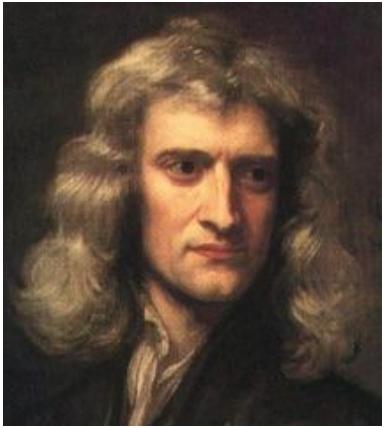
La definizione di tempo

- Newton, 1689



La definizione di tempo

- Newton, 1689
- *"absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external."*



Ordini totali e parziali

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è

ORDINI TOTALI E PARZIALI

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è
  - antisimmetrica: se $a \leq b$ e $b \leq a$ allora $a = b$

ORDINI TOTALI E PARZIALI

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è
  - antisimmetrica: se $a \leq b$ e $b \leq a$ allora $a = b$
  - transitiva: se $a \leq b$ e $b \leq c$ allora $a \leq c$

Ordini totali e parziali

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è
  - antisimmetrica: se $a \leq b$ e $b \leq a$ allora $a = b$
  - transitiva: se $a \leq b$ e $b \leq c$ allora $a \leq c$
  - totale: per ogni coppia di $a, b \in X$ vale che $a \leq b$ o $b \leq a$

Ordini totali e parziali

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è
  - antisimmetrica: se $a \leq b$ e $b \leq a$ allora $a = b$
  - transitiva: se $a \leq b$ e $b \leq c$ allora $a \leq c$
  - totale: per ogni coppia di $a, b \in X$ vale che $a \leq b$ o $b \leq a$
  - riflessiva: $a \leq a$

Ordini totali e parziali

- Un ordine totale $\leq$ su un insieme X è una relazione binaria che è
  - antisimmetrica: se $a \leq b$ e $b \leq a$ allora $a = b$
  - transitiva: se $a \leq b$ e $b \leq c$ allora $a \leq c$
  - totale: per ogni coppia di $a, b \in X$ vale che $a \leq b$ o $b \leq a$
  - riflessiva: $a \leq a$
- Un ordine parziale su un insieme X è una relazione binaria che è antisimmetrica, transitiva e riflessiva (non totale)

CARATTERISTICHE DEL TEMPO

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo

CARATTERISTICHE DEL TEMPO

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo
  - $a \to b$, ordine totale sugli eventi

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo
  - $a \to b$, ordine totale sugli eventi
- Sia $a_0 \to a_1$. Un intervallo (a_0, a_1) è la durata di tempo tra a_0 e a_1

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo
  - $a \to b$, ordine totale sugli eventi
- Sia $a_0 \to a_1$. Un intervallo (a_0, a_1) è la durata di tempo tra a_0 e a_1
- $I_A = (a_0, a_1)$ precede $I_B = (b_0, b_1)$ se $a_1 \to b_0$ (se A termina prima che inizi B)

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo
  - $a \to b$, ordine totale sugli eventi
- Sia $a_0 \to a_1$. Un intervallo (a_0, a_1) è la durata di tempo tra a_0 e a_1
- $I_A = (a_0, a_1)$ precede $I_B = (b_0, b_1)$ se $a_1 \to b_0$ (se A termina prima che inizi B)
  - $I_A \to I_B$, ordine parziale sugli intervalli

Caratteristiche del tempo

"programmazione concorrente" significa parlare del tempo

- Eventi istantanei: in un singolo istante di tempo
- Eventi mai simultanei: se necessario scegliamo un qualsiasi ordine tra eventi molto "vicini" nel tempo.
- Un thread A produce eventi $a_0, a_1, \ldots,$
- La j-ma occorrenza dell'evento a_i è indicata come a_i^j
- Un evento precede un altro se occorre prima nel tempo
  - $a \to b$, ordine totale sugli eventi
- Sia $a_0 \to a_1$. Un intervallo (a_0, a_1) è la durata di tempo tra a_0 e a_1
- $I_A = (a_0, a_1)$ precede $I_B = (b_0, b_1)$ se $a_1 \to b_0$ (se A termina prima che inizi B)
  - $I_A \to I_B$, ordine parziale sugli intervalli
- Se per due intervalli I e J vale che $I \not\to J$ e che $J \not\to I$ allora I e J sono *concorrenti*

Presentazione

Presentazione

UNA (NON-)SOLUZIONE PER IL CONTATORE

```
public class Counter {
    private long value;
    public Counter(int c) {
        value = c;
    }
    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}
```

stato del contatore (finito!)

UNA (NON-)SOLUZIONE PER IL CONTATORE

```
public class Counter {
    private long value;
    public Counter(int c) {
        value = c;
    }
    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}
```

- stato del contatore (finito!)
- costruttore che inizializza il contatore

UNA (NON-)SOLUZIONE PER IL CONTATORE

```
public class Counter {
    private long value;
    public Counter(int c) {
        value = c;
    }
    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}
```

- stato del contatore (finito!)
- costruttore che inizializza il contatore
- incrementa e restituisce il valore precedente

UNA (NON-)SOLUZIONE PER IL CONTATORE

```
public class Counter {
    private long value;
    public Counter(int c) {
        value = c;
    }
    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}
```

- stato del contatore (finito!)
- costruttore che inizializza il contatore
- incrementa e restituisce il valore precedente
- inizio della zona di "pericolo"

UNA (NON-)SOLUZIONE PER IL CONTATORE

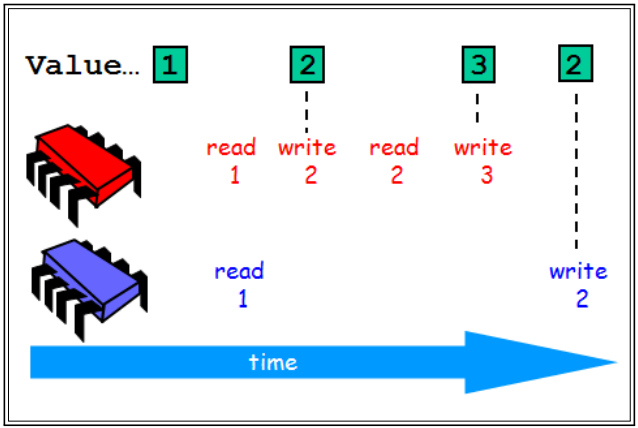
```
public class Counter {
    private long value;
    public Counter(int c) {
        value = c;
    }
    public long getAndIncrement() {
        int temp = value;
        value = temp + 1;
        return temp;
    }
}
```

- stato del contatore (finito!)
- costruttore che inizializza il contatore
- incrementa e restituisce il valore precedente
- inizio della zona di "pericolo"
- fine della zona di pericolo

Non funziona!

La lettura di `value` per l'incremento può avvenire in maniera interfogliata tra thread, causando errori

Il problema



Sezione critica

- Si trasforma la zona di "pericolo" in una *sezione critica*

Sezione critica

- Si trasforma la zona di “pericolo” in una *sezione critica*
- …eseguita solamente da un thread alla volta: *in mutua esclusione*

Sezione critica

- Si trasforma la zona di "pericolo" in una *sezione critica*
- …eseguita solamente da un thread alla volta: *in mutua esclusione*
- Metodo standard: si utilizza un *lock*: un thread acquisisce il diritto a entrare bloccando il lock, e lo rilascia quando esce

Sezione critica

- Si trasforma la zona di "pericolo" in una *sezione critica*
- …eseguita solamente da un thread alla volta: *in mutua esclusione*
- Metodo standard: si utilizza un *lock*: un thread acquisisce il diritto a entrare bloccando il lock, e lo rilascia quando esce
- Un thread che trova il lock bloccato, attende che si sblocchi

La interfaccia Lock

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

prima di entrare in sezione critica

La interfaccia Lock

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

- prima di entrare in sezione critica
- prima di lasciare la sezione critica

COME SI USA UN LOCK

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

Classe counter

COME SI USA UN LOCK

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

- Classe counter
- Usato per proteggere la sezione critica

COME SI USA UN LOCK

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

- Classe counter
- Usato per proteggere la sezione critica
- Prima di entrare in sezione critica . . .

COME SI USA UN LOCK

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

- Classe counter
- Usato per proteggere la sezione critica
- Prima di entrare in sezione critica . . .
- Sezione critica

COME SI USA UN LOCK

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

- Classe counter
- Usato per proteggere la sezione critica
- Prima di entrare in sezione critica . . .
- Sezione critica
- Prima di uscire dalla sezione critica

Presentazione

La correttezza della mutua esclusione

Un thread che usa un lock è *well-formed* se:

- ogni sezione critica è associata ad un unico oggetto `Lock`

La correttezza della mutua esclusione

Un thread che usa un lock è *well-formed* se:

- ogni sezione critica è associata ad un unico oggetto `Lock`
- il thread chiama il metodo `lock()` quando entra in sezione critica...

La correttezza della mutua esclusione

Un thread che usa un lock è *well-formed* se:

- ogni sezione critica è associata ad un unico oggetto `Lock`
- il thread chiama il metodo `lock()` quando entra in sezione critica...
- ...e chiama il metodo `unlock()` quando esce dalla sezione critica

La correttezza della mutua esclusione

Un thread che usa un lock è *well-formed* se:

- ogni sezione critica è associata ad un unico oggetto `Lock`
- il thread chiama il metodo `lock()` quando entra in sezione critica...
- ...e chiama il metodo `unlock()` quando esce dalla sezione critica

La correttezza del `Lock`

E' cruciale per il corretto funzionamento della sezione critica: necessario formalizzare il comportamento per dimostrarne il corretto funzionamento!

PROPRIETÀ DI UN LOCK - 1

- Sia CS_A^j l'intervallo in cui A esegue la sezione critica per la j-ma volta

PROPRIETÀ DI UN LOCK - 1

- Sia CS_A^j l'intervallo in cui A esegue la sezione critica per la j-ma volta
- **Mutua esclusione:** Sezioni critiche di thread diversi non si sovrappongono: per A e B, e j, k interi, vale $CS_A^k \rightarrow CS_B^j$ oppure vale $CS_B^j \rightarrow CS_A^k$

PROPRIETÀ DI UN LOCK - 1

- Sia CS_A^j l'intervallo in cui A esegue la sezione critica per la j-ma volta
- **Mutua esclusione:** Sezioni critiche di thread diversi non si sovrappongono: per A e B, e j, k interi, vale $CS_A^k \rightarrow CS_B^j$ oppure vale $CS_B^j \rightarrow CS_A^k$
- **Deadlock-free:** se qualche thread cerca di acquisire il lock, qualcuno ha successo. Se un thread chiama `lock()` ma non ottiene mai il lock, allora altri thread devono riuscire a completare un numero infinito di volte la sezione critica

PROPRIETÀ DI UN LOCK - 1

- Sia CS_A^j l'intervallo in cui A esegue la sezione critica per la j-ma volta
- **Mutua esclusione:** Sezioni critiche di thread diversi non si sovrappongono: per A e B, e j, k interi, vale $CS_A^k \rightarrow CS_B^j$ oppure vale $CS_B^j \rightarrow CS_A^k$
- **Deadlock-free:** se qualche thread cerca di acquisire il lock, qualcuno ha successo. Se un thread chiama `lock()` ma non ottiene mai il lock, allora altri thread devono riuscire a completare un numero infinito di volte la sezione critica
- **Starvation-free:** Ogni thread che cerca di acquisire il lock, ha successo alla fine (ogni chiamata a `lock()` ritorna al metodo chiamante).

PROPRIETÀ : *rephrasing*

- **Deadlock-free:** se un processo cerca di entrare nella sezione critica, qualcuno (non necessariamente lo stesso processo) entrerà

PROPRIETÀ : *rephrasing*

- **Deadlock-free:** se un processo cerca di entrare nella sezione critica, qualcuno (non necessariamente lo stesso processo) entrerà
- **Starvation-free:** se un processo cerca di entrare nella sezione critica, alla fine ci riuscirà

PROPRIETÀ : *rephrasing*

- **Deadlock-free:** se un processo cerca di entrare nella sezione critica, qualcuno (non necessariamente lo stesso processo) entrerà
- **Starvation-free:** se un processo cerca di entrare nella sezione critica, alla fine ci riuscirà
- Quindi la *starvation-freedom* implica la *deadlock-freedom*.

PROPRIETÀ : *rephrasing*

- **Deadlock-free:** se un processo cerca di entrare nella sezione critica, qualcuno (non necessariamente lo stesso processo) entrerà
- **Starvation-free:** se un processo cerca di entrare nella sezione critica, alla fine ci riuscirà
- Quindi la *starvation-freedom* implica la *deadlock-freedom*.
- Non c'é garanzia, però, che la *starvation-freedom* garantisca quanto tempo debba aspettare un thread in attesa

PROPRIETÀ DI UN LOCK - 2

- Separiamo concettualmente la esecuzione di A del metodo `lock()` in due componenti:

PROPRIETÀ DI UN LOCK - 2

- Separiamo concettualmente la esecuzione di A del metodo `lock()` in due componenti:
  - *doorway* la cui esecuzione da parte di A prende un intervallo D_A di durata limitata superiormente

PROPRIETÀ DI UN LOCK - 2

- Separiamo concettualmente la esecuzione di A del metodo `lock()` in due componenti:
  - *doorway* la cui esecuzione da parte di A prende un intervallo D_A di durata limitata superiormente
  - *waiting* la cui esecuzione da parte di A prende un intervallo W_A di durata non limitata

PROPRIETÀ DI UN LOCK - 2

- Separiamo concettualmente la esecuzione di A del metodo `lock()` in due componenti:
  - *doorway* la cui esecuzione da parte di A prende un intervallo D_A di durata limitata superiormente
  - *waiting* la cui esecuzione da parte di A prende un intervallo W_A di durata non limitata
- **First-come-first-served:** se il thread A finisce la sua doorway prima che B inizi la sua doorway, allora A non può essere sorpassato da B:

$$\text{se } D_A^j \rightarrow D_B^k \text{ allora } CS_A^j \rightarrow CS_B^k$$

Presentazione

IL FLUSSO DELLA PRESENTAZIONE

- Presentiamo due algoritmi di lock: `LockOne` e `LockTwo`

Il flusso della presentazione

- Presentiamo due algoritmi di lock: `LockOne` e `LockTwo`
- Viene mostrato come ciascuno dei due gode della proprietà di *mutua esclusione*

Il flusso della presentazione

- Presentiamo due algoritmi di lock: `LockOne` e `LockTwo`
- Viene mostrato come ciascuno dei due gode della proprietà di *mutua esclusione*
- ma che ciascuno di loro **non è** *deadlock-free*

Il flusso della presentazione

- Presentiamo due algoritmi di lock: `LockOne` e `LockTwo`
- Viene mostrato come ciascuno dei due gode della proprietà di *mutua esclusione*
- ma che ciascuno di loro **non è** *deadlock-free*
- ciascuno in due situazioni diverse (quando i task sono ointerfogliati o quando vengono eseguiti uno dopo l'altro)

Il flusso della presentazione

- Presentiamo due algoritmi di lock: `LockOne` e `LockTwo`
- Viene mostrato come ciascuno dei due gode della proprietà di *mutua esclusione*
- ma che ciascuno di loro **non è** *deadlock-free*
- ciascuno in due situazioni diverse (quando i task sono ointerfogliati o quando vengono eseguiti uno dopo l'altro)
- Il lock di Peterson, che unisce le tecniche dei due algoritmi, soddisfa sia la mutua esclusione che la condizione di deadlock-free.

Presentazione

L'ALGORITMO LockOne

implementa Lock

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```


L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)

L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)
- l'indice del thread è 0 o 1

L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)
- l'indice del thread è 0 o 1
- . . . e si calcola l'indice dell'altro

L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)
- l'indice del thread è 0 o 1
- . . . e si calcola l'indice dell'altro
- *"sono interessato al lock"*

L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)
- l'indice del thread è 0 o 1
- . . . e si calcola l'indice dell'altro
- *"sono interessato al lock"*
- attesa finquando l'altro è interessato

L'ALGORITMO LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }

}
```

- implementa `Lock`
- devono essere `volatile` (errore sul libro)
- l'indice del thread è 0 o 1
- . . . e si calcola l'indice dell'altro
- *"sono interessato al lock"*
- attesa finquando l'altro è interessato
- *"non sono più interessato"*

NOTAZIONE

- Indichiamo con $w_A(x = v)$ l'evento in cui A assegna il valore v ad x

NOTAZIONE

- Indichiamo con $w_A(x = v)$ l'evento in cui A assegna il valore v ad x
- Indichiamo con $r_A(v == x)$ l'evento in cui A legge dalla variabile x un valore v

THM: LockOne SODDISFA LA ME - 1

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per contraddizione, esistono interi j, k tali che:
  $CS^j_A \not\to CS^k_B$ e $CS^k_B \not\to CS^j_A$

THM: LockOne SODDISFA LA ME - 1

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per contraddizione, esistono interi j, k tali che: $CS_A^j \not\to CS_B^k$ e $CS_B^k \not\to CS_A^j$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto

THM: LockOne SODDISFA LA ME - 1

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per contraddizione, esistono interi j, k tali che: $CS^j_A \not\to CS^k_B$ e $CS^k_B \not\to CS^j_A$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto

ha scritto che era interessato

$w_A(flag[A] = true) \to$

THM: LockOne SODDISFA LA ME - 1

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per contraddizione, esistono interi j, k tali che: $CS_A^j \not\to CS_B^k$ e $CS_B^k \not\to CS_A^j$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto
- ha scritto che era interessato
- ha verificato che B non era interessato

$$w_A(flag[A] = true) \to r_A(flag[B] == false) \to$$

THM: LockOne SODDISFA LA ME - 1

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per contraddizione, esistono interi j, k tali che: $CS_A^j \not\to CS_B^k$ e $CS_B^k \not\to CS_A^j$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto
- ha scritto che era interessato
- ha verificato che B non era interessato
- e poi è entrato in sezione critica

$$w_A(flag[A] = true) \to r_A(flag[B] == false) \to CS_A$$

THM: LockOne SODDISFA LA ME - 2

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto

THM: LockOne SODDISFA LA ME - 2

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto

ha scritto che era interessato

$w_B(flag[B] = true) \rightarrow$

THM: LockOne SODDISFA LA ME - 2

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto
- ha scritto che era interessato

ha verificato che A non era interessato

$$w_B(flag[B] = true) \rightarrow r_B(flag[A] == false) \rightarrow$$

THM: LockOne SODDISFA LA ME - 2

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto
- ha scritto che era interessato
- ha verificato che A non era interessato
- e poi è entrato in sezione critica

$$w_B(flag[B] = true) \rightarrow r_B(flag[A] == false) \rightarrow CS_B$$

THM: LockOne SODDISFA LA ME - 3

$$w_A(flag[A] = true) \rightarrow r_A(flag[B] == false) \rightarrow CS_A$$

$$w_B(flag[B] = true) \rightarrow r_B(flag[A] == false) \rightarrow CS_B$$

- Poiché quando $flag[B]$ è messo a true da B, nessuno lo cambia (siamo nel conflitto della CS), allora

$$r_A(flag[B] == false) \rightarrow w_B(flag[B] = true)$$

THM: LockOne SODDISFA LA ME - 3

$$w_A(flag[A] = true) \rightarrow r_A(flag[B] == false) \rightarrow CS_A$$

$$w_B(flag[B] = true) \rightarrow r_B(flag[A] == false) \rightarrow CS_B$$

- Poiché quando $flag[B]$ è messo a true da B, nessuno lo cambia (siamo nel conflitto della CS), allora

$$r_A(flag[B] == false) \rightarrow w_B(flag[B] = true)$$

THM: LockOne SODDISFA LA ME - 4

Cosa abbiamo finora

$$w_A(flag[A] = true) \rightarrow r_A(flag[B] == false) \rightarrow CS_A \quad (1)$$

$$w_B(flag[B] = true) \rightarrow r_B(flag[A] == false) \rightarrow CS_B \quad (2)$$

$$r_A(flag[B] == false) \rightarrow w_B(flag[B] = true) \quad (3)$$

- Dalla (1)

$w_A(flag[A] = true) \rightarrow r_A(flag[B] == false)$

THM: LockOne SODDISFA LA ME - 4

Cosa abbiamo finora

$$\mathrm{w}_A(flag[A] = true) \rightarrow \mathrm{r}_A(flag[B] == false) \rightarrow CS_A \quad (1)$$

$$\mathrm{w}_B(flag[B] = true) \rightarrow \mathrm{r}_B(flag[A] == false) \rightarrow CS_B \quad (2)$$

$$\mathrm{r}_A(flag[B] == false) \rightarrow \mathrm{w}_B(flag[B] = true) \quad (3)$$

- Dalla (1)
- Dalla (3)

$$\mathrm{w}_A(flag[A] = true) \rightarrow \mathrm{r}_A(flag[B] == false) \rightarrow \mathrm{w}_B(flag[B] = true)$$

THM: LockOne SODDISFA LA ME - 4

Cosa abbiamo finora

$$\mathrm{w}_A(flag[A] = true) \rightarrow \mathrm{r}_A(flag[B] == false) \rightarrow CS_A \quad (1)$$

$$\mathrm{w}_B(flag[B] = true) \rightarrow \mathrm{r}_B(flag[A] == false) \rightarrow CS_B \quad (2)$$

$$\mathrm{r}_A(flag[B] == false) \rightarrow \mathrm{w}_B(flag[B] = true) \quad (3)$$

- Dalla (1)
- Dalla (3)
- Dalla (2)

$$\mathrm{w}_A(flag[A] = true) \rightarrow \mathrm{r}_A(flag[B] == false) \rightarrow \mathrm{w}_B(flag[B] = true) \rightarrow \mathrm{r}_B(flag[A] == false)$$

THM: LockOne SODDISFA LA ME - 5

- $w_A(flag[A] = true) \rightarrow r_B(flag[A] == false)$

THM: LockOne SODDISFA LA ME - 5

- $w_A(flag[A] = true) \rightarrow r_B(flag[A] == false)$
- è un assurdo perché nessun altro modifica $flag[A]$.

THM: `LockOne` SODDISFA LA ME - 5

- $w_A(flag[A] = true) \rightarrow r_B(flag[A] == false)$
- è un assurdo perché nessun altro modifica $flag[A]$.
- q.e.d.

INADEGUATEZZA DI LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Se la esecuzione dei thread si “interfogliano”…

INADEGUATEZZA DI LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Se la esecuzione dei thread si "interfogliano"...

...allora se $w_A(flag[A] = true)$ e $w_B(flag[B] = true)$ capitano prima di

INADEGUATEZZA DI LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Se la esecuzione dei thread si "interfogliano"...
- ...allora se $w_A(flag[A] = true)$ e $w_B(flag[B] = true)$ capitano prima di

...$r_A(flag[B])$ e di $r_B(flag[A])$

INADEGUATEZZA DI LockOne

```
class LockOne implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {}
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Se la esecuzione dei thread si "interfogliano"...
- ...allora se $w_A(flag[A] = true)$ e $w_B(flag[B] = true)$ capitano prima di
- ... $r_A(flag[B])$ e di $r_B(flag[A])$
- allora sia A che B attendono bloccati

Presentazione

L'ALGORITMO LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

Implementazione interfaccia di Lock

L'ALGORITMO LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Implementazione interfaccia di Lock

"prego, prima lei!"

L'ALGORITMO LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Implementazione interfaccia di Lock
- *"prego, prima lei!"*
- attendi

THM: LockTwo SODDISFA LA ME - 1

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Per contraddizione, esistono interi j, k tali che:
  $CS^j_A \not\to CS^k_B$ e $CS^k_B \not\to CS^j_A$

THM: LockTwo SODDISFA LA ME - 1

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Per contraddizione, esistono interi j, k tali che:
  $CS_A^j \not\to CS_B^k$ e $CS_B^k \not\to CS_A^j$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto

THM: LockTwo SODDISFA LA ME - 1

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Per contraddizione, esistono interi j, k tali che: $CS_A^j \not\to CS_B^k$ e $CS_B^k \not\to CS_A^j$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto

ha dato precedenza

$w_A(victim = A) \to$

THM: LockTwo SODDISFA LA ME - 1

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Per contraddizione, esistono interi j, k tali che: $CS^j_A \not\to CS^k_B$ e $CS^k_B \not\to CS^j_A$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto
- ha dato precedenza
- ha verificato che B dava precedenza

$$w_A(victim = A) \to r_A(victim == B) \to$$

THM: LockTwo SODDISFA LA ME - 1

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Per contraddizione, esistono interi j, k tali che: $CS^j_A \not\to CS^k_B$ e $CS^k_B \not\to CS^j_A$
- L'ultima volta che il thread A ha eseguito `lock()` prima del conflitto
- ha dato precedenza
- ha verificato che B dava precedenza
- e poi è entrato in CS

$$w_A(victim = A) \to r_A(victim == B) \to CS_A$$

THM: LockTwo SODDISFA LA ME - 2

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto

THM: LockTwo SODDISFA LA ME - 2

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto

ha dato precedenza

$w_B(victim = B) \rightarrow$

THM: LockTwo SODDISFA LA ME - 2

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto
- ha dato precedenza
- ha verificato che A dava precedenza

$$w_B(victim = B) \rightarrow r_B(victim == A) \rightarrow$$

THM: LockTwo SODDISFA LA ME - 2

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- L'ultima volta che il thread B ha eseguito `lock()` prima del conflitto
- ha dato precedenza
- ha verificato che A dava precedenza
- e poi è entrato in CS

$$w_B(victim = B) \rightarrow r_B(victim == A) \rightarrow CS_B$$

THM: LockTwo SODDISFA LA ME - 3

Cosa abbiamo finora

$$w_A(victim = A) \rightarrow r_A(victim == B) \rightarrow CS_A \quad (4)$$

$$w_B(victim = B) \rightarrow r_B(victim == A) \rightarrow CS_B \quad (5)$$

- B deve assegnare B a `victim` tra gli eventi $w_A(victim = A)$ e $r_A(victim == B)$ dalla (4)

THM: LockTwo SODDISFA LA ME - 3

Cosa abbiamo finora

$$\mathrm{w}_A(victim = A) \rightarrow \mathrm{r}_A(victim == B) \rightarrow CS_A \quad (4)$$

$$\mathrm{w}_B(victim = B) \rightarrow \mathrm{r}_B(victim == A) \rightarrow CS_B \quad (5)$$

- B deve assegnare B a `victim` tra gli eventi $\mathrm{w}_A(victim = A)$ e $\mathrm{r}_A(victim == B)$ dalla (4)
- Quindi $\mathrm{w}_A(victim = A) \rightarrow \mathrm{w}_B(victim = B) \rightarrow \mathrm{r}_A(victim == B)$

THM: `LockTwo` SODDISFA LA ME - 3

Cosa abbiamo finora

$$w_A(victim = A) \rightarrow r_A(victim == B) \rightarrow CS_A \quad (4)$$

$$w_B(victim = B) \rightarrow r_B(victim == A) \rightarrow CS_B \quad (5)$$

- B deve assegnare B a `victim` tra gli eventi $w_A(victim = A)$ e $r_A(victim == B)$ dalla (4)
- Quindi $w_A(victim = A) \rightarrow w_B(victim = B) \rightarrow r_A(victim == B)$
- Ma a questo punto, `victim` non cambia e ogni lettura restituisce B contraddicendo (5)

THM: LockTwo SODDISFA LA ME - 3

Cosa abbiamo finora

$$\mathrm{w}_A(victim = A) \rightarrow \mathrm{r}_A(victim == B) \rightarrow CS_A \quad (4)$$

$$\mathrm{w}_B(victim = B) \rightarrow \mathrm{r}_B(victim == A) \rightarrow CS_B \quad (5)$$

- B deve assegnare B a `victim` tra gli eventi $\mathrm{w}_A(victim = A)$ e $\mathrm{r}_A(victim == B)$ dalla (4)
- Quindi $\mathrm{w}_A(victim = A) \rightarrow \mathrm{w}_B(victim = B) \rightarrow \mathrm{r}_A(victim == B)$
- Ma a questo punto, `victim` non cambia e ogni lettura restituisce B contraddicendo (5)
- q.e.d.

INADEGUATEZZA DI LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Se un thread viene eseguito prima dell'altro . . .

INADEGUATEZZA DI LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Se un thread viene eseguito prima dell'altro ...
- ...c'é deadlock

INADEGUATEZZA DI LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Se un thread viene eseguito prima dell'altro ...
- ...c'é deadlock
- Ma se i due thread si interfogliano, il metodo `lock()` ha successo

INADEGUATEZZA DI LockTwo

```
class LockTwo implements Lock {

    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;
        while (victim == i) {}
    }

    public void unlock() {
    }

}
```

- Se un thread viene eseguito prima dell'altro ...
- ...c'é deadlock
- Ma se i due thread si interfogliano, il metodo `lock()` ha successo
- `LockTwo` complementa `LockOne` !

Presentazione

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

per manifestare interesse

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- per manifestare interesse
- per dare la precedenza

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- per manifestare interesse
- per dare la precedenza
- segnala interesse

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- per manifestare interesse
- per dare la precedenza
- segnala interesse
- ma dai la precedenza all'altro

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- per manifestare interesse
- per dare la precedenza
- segnala interesse
- ma dai la precedenza all'altro
- attendi . . .

LOCK DI PETERSON

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- per manifestare interesse
- per dare la precedenza
- segnala interesse
- ma dai la precedenza all'altro
- attendi ...
- non più interessato

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no
- L'ultima esecuzione del `lock()` da A e B

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no
- L'ultima esecuzione del `lock()` da A e B

A segnala interesse

$w_A(flag[A] = true) \rightarrow$

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no
- L'ultima esecuzione del `lock()` da A e B
- A segnala interesse
- ma da la precedenza a B

$$w_A(flag[A] = true) \rightarrow w_A(victim = A) \rightarrow$$

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no
- L'ultima esecuzione del `lock()` da A e B
- A segnala interesse
- ma da la precedenza a B
- attendi ...

$$w_A(flag[A] = true) \rightarrow w_A(victim = A) \rightarrow r_A(flag[B]) \rightarrow r_A(victim) \rightarrow$$

THM: IL LOCK DI PETERSON SODDISFA ME - 1

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- Per assurdo, si supponga di no
- L'ultima esecuzione del `lock()` da A e B
- A segnala interesse
- ma da la precedenza a B
- attendi ...
- entra in CS

$$w_A(flag[A] = true) \rightarrow w_A(victim = A) \rightarrow r_A(flag[B]) \rightarrow r_A(victim) \rightarrow CS_A$$

THM: IL LOCK DI PETERSON SODDISFA ME - 2

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima esecuzione del `lock()` da A e B

THM: IL LOCK DI PETERSON SODDISFA ME - 2

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima esecuzione del `lock()` da A e B

B segnala interesse

$w_B(flag[B] = true) \rightarrow$

THM: IL LOCK DI PETERSON SODDISFA ME - 2

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima esecuzione del `lock()` da A e B
- B segnala interesse
- ma da la precedenza a A

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

THM: IL LOCK DI PETERSON SODDISFA ME - 2

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                  new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima esecuzione del `lock()` da A e B
- B segnala interesse
- ma da la precedenza a A
- attendi ...

$$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow$$

THM: IL LOCK DI PETERSON SODDISFA ME - 2

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- L'ultima esecuzione del `lock()` da A e B
- B segnala interesse
- ma da la precedenza a A
- attendi ...
- entra in CS

$$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$$

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$

Se A entra in CS, allora $flag[B] = false$

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$
- Se A entra in CS, allora $flag[B] = false$
- $w_A(victim = A) \rightarrow r_A(flag[B] == false)$

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$
- Se A entra in CS, allora $flag[B] = false$
- $w_A(victim = A) \rightarrow r_A(flag[B] == false)$

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

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$
- Se A entra in CS, allora $flag[B] = false$
- $w_A(victim = A) \rightarrow r_A(flag[B] == false)$

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow w_A(victim = A) \rightarrow r_A(flag[B] == false)$

THM: IL LOCK DI PETERSON SODDISFA ME - 3

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow r_B(flag[A]) \rightarrow r_B(victim) \rightarrow CS_B$

```
class Peterson implements Lock {

    private volatile boolean[] flag =
                                new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

- W.l.o.g. sia A l'ultimo a scrivere victim
- $w_B(victim = B) \rightarrow w_A(victim = A)$
- Se A entra in CS, allora $flag[B] = false$
- $w_A(victim = A) \rightarrow r_A(flag[B] == false)$

$w_B(flag[B] = true) \rightarrow w_B(victim = B) \rightarrow w_A(victim = A) \rightarrow r_A(flag[B] == false)$

Contraddizione!

ALTRI RISULTATI SUL LOCK DI PETERSON

- Il Lock di Peterson è starvation free (dimostrazione sul libro)

ALTRI RISULTATI SUL LOCK DI PETERSON

- Il Lock di Peterson è starvation free (dimostrazione sul libro)
- Il Lock di Peterson è deadlock free (dimostrazione sul libro)

Presentazione

L'ALGORITMO DEL "PANETTIERE" (BAKERY)

- Algoritmo sviluppato da Leslie Lamport

L'ALGORITMO DEL "PANETTIERE" (BAKERY)

- Algoritmo sviluppato da Leslie Lamport
- *First-come-first-served*: una versione distribuita di una macchinetta che distribuisce i numeri dal "panettiere"

L'ALGORITMO DEL "PANETTIERE" (BAKERY)

- Algoritmo sviluppato da Leslie Lamport
- *First-come-first-served*: una versione distribuita di una macchinetta che distribuisce i numeri dal "panettiere"
- Ogni thread prende un numero durante la *doorway* ...

L'ALGORITMO DEL "PANETTIERE" (BAKERY)

- Algoritmo sviluppato da Leslie Lamport
- *First-come-first-served*: una versione distribuita di una macchinetta che distribuisce i numeri dal "panettiere"
- Ogni thread prende un numero durante la *doorway* ...
- ...ed aspetta fino a quando non c'é più nessun thread in attesa con un numero più "basso" del proprio

L'ALGORITMO DEL "PANETTIERE"

Implementa `Lock`

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```


L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)
- *Doorway:* intenzione di entrare in CS

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)
- *Doorway:* intenzione di entrare in CS
- prende il biglietto: end *Doorway*

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)
- *Doorway:* intenzione di entrare in CS
- prende il biglietto: end *Doorway*
- *Waiting:* finquando esiste un thread con una etichetta più piccola della propria

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)
- *Doorway:* intenzione di entrare in CS
- prende il biglietto: end *Doorway*
- *Waiting:* finquando esiste un thread con una etichetta più piccola della propria
- cicla

L'ALGORITMO DEL "PANETTIERE"

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Implementa `Lock`
- Array di label
- Costruttore (azzera)
- *Doorway:* intenzione di entrare in CS
- prende il biglietto: end *Doorway*
- *Waiting:* finquando esiste un thread con una etichetta più piccola della propria
- cicla
- non interessato alla CS

Il confronto tra etichette

- Due o più thread possono eseguire la doorway contemporaneamente

IL CONFRONTO TRA ETICHETTE

- Due o più thread possono eseguire la doorway contemporaneamente
- Potrebbero prendere lo stesso valore di etichetta:

```
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```


Il confronto tra etichette

- Due o più thread possono eseguire la doorway contemporaneamente
- Potrebbero prendere lo stesso valore di etichetta:

```
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Soluzione: il confronto viene effettuato "lessicograficamente" concatenando la propria ID

IL CONFRONTO TRA ETICHETTE

- Due o più thread possono eseguire la doorway contemporaneamente
- Potrebbero prendere lo stesso valore di etichetta:

```
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Soluzione: il confronto viene effettuato "lessicograficamente" concatenando la propria ID
- $(label[i], i) << (label[j], j)$ sse

Il confronto tra etichette

- Due o più thread possono eseguire la doorway contemporaneamente
- Potrebbero prendere lo stesso valore di etichetta:

```
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Soluzione: il confronto viene effettuato "lessicograficamente" concatenando la propria ID
- $(label[i], i) << (label[j], j)$ sse
- $label[i] < label[j]$ o $(label[i] = label[j]$ e $i < j)$

Il confronto tra etichette

- Due o più thread possono eseguire la doorway contemporaneamente
- Potrebbero prendere lo stesso valore di etichetta:

```
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Soluzione: il confronto viene effettuato "lessicograficamente" concatenando la propria ID
- $(label[i], i) << (label[j], j)$ sse
- $label[i] < label[j]$ o $(label[i] = label[j]$ e $i < j)$
- label che crescono sempre (problema?)

THM: Bakery È DEADLOCK-FREE

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

Tra i thread in attesa ($flag[A]$ settato) . . .

THM: Bakery È DEADLOCK-FREE

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Tra i thread in attesa ($flag[A]$ settato) ...
- ce ne sarà uno, sia A, con una etichetta $(label[A], A)$

THM: Bakery È DEADLOCK-FREE

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Tra i thread in attesa ($flag[A]$ settato) ...
- ce ne sarà uno, sia A, con una etichetta $(label[A], A)$
- ...che risulta essere la più piccola

THM: Bakery È DEADLOCK-FREE

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Tra i thread in attesa ($flag[A]$ settato) ...
- ce ne sarà uno, sia A, con una etichetta $(label[A], A)$
- ...che risulta essere la più piccola
- Questo thread non attenderà nessun altro thread

THM: Bakery È DEADLOCK-FREE

```
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;
        }
    }

    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ...,label[n-1]) + 1;
        while ((∃k != i)(flag[k] &&
                (label[k],k) << (label[i],i))) {};
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

- Tra i thread in attesa ($flag[A]$ settato) ...
- ce ne sarà uno, sia A, con una etichetta $(label[A], A)$
- ...che risulta essere la più piccola
- Questo thread non attenderà nessun altro thread
- q.e.d.

THM: `Bakery` È FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$

THM: `Bakery` È FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$
- $\ldots$ allora la label di A precede quella di B

THM: `Bakery` È FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$
- $\ldots$ allora la label di A precede quella di B
- Infatti: $\mathrm{w}_A(label[A]) \rightarrow \mathrm{r}_B(label[A]) \rightarrow \mathrm{w}_B(label[B]) \rightarrow \mathrm{r}_B(flag[A])$

THM: `Bakery` È FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$
- $\ldots$ allora la label di A precede quella di B
- Infatti: $\mathrm{w}_A(label[A]) \rightarrow \mathrm{r}_B(label[A]) \rightarrow \mathrm{w}_B(label[B]) \rightarrow \mathrm{r}_B(flag[A])$
- Quindi B non entra in quanto $flag[A]$ è settato

THM: `Bakery` È FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$
- $\ldots$ allora la label di A precede quella di B
- Infatti: $\mathrm{w}_A(label[A]) \rightarrow \mathrm{r}_B(label[A]) \rightarrow \mathrm{w}_B(label[B]) \rightarrow \mathrm{r}_B(flag[A])$
- Quindi B non entra in quanto $flag[A]$ è settato
- q.e.d.

THM: `Bakery` è FCFS

- Se la doorway di A precede quella di B: $D_A \rightarrow D_B \ldots$
- $\ldots$ allora la label di A precede quella di B
- Infatti: $\mathrm{w}_A(label[A]) \rightarrow \mathrm{r}_B(label[A]) \rightarrow \mathrm{w}_B(label[B]) \rightarrow \mathrm{r}_B(flag[A])$
- Quindi B non entra in quanto $flag[A]$ è settato
- q.e.d.
- Nota: un algoritmo deadlock-free e FCFS è anche starvation-free

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS

THM: `Bakery` GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```


THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Allora deve aver letto che $flag[A]$ era falso

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Allora deve aver letto che $flag[A]$ era falso
  - id fissati e label strettamente crescenti

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Allora deve aver letto che $flag[A]$ era falso
  - id fissati e label strettamente crescenti
- Quindi $labeling_B \rightarrow \mathrm{r}_B(flag[A]) \rightarrow \mathrm{w}_A(flag[A]) \rightarrow labeling_A$

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Allora deve aver letto che $flag[A]$ era falso
  - id fissati e label strettamente crescenti
- Quindi $labeling_B \rightarrow \mathrm{r}_B(flag[A]) \rightarrow \mathrm{w}_A(flag[A]) \rightarrow labeling_A$
- ma in questa maniera si avrebbe che $(label[B], B) << (label[A], A)$ (contraddizione)

THM: Bakery GODE DELLA ME

- Per assurdo, siano A e B insieme in CS
- Sia $labeling_A$ sia l'insieme degli eventi per calcolare le label da A prima di entrare in CS
- W.l.o.g. sia $(label[A], A) << (label[B], B)$
- Quando B ha passato la parte di waiting

```
flag[i] = true;
label[i] = max(label[0], ...,label[n-1]) + 1;
while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
```

- Allora deve aver letto che $flag[A]$ era falso
  - id fissati e label strettamente crescenti
- Quindi $labeling_B \rightarrow \mathrm{r}_B(flag[A]) \rightarrow \mathrm{w}_A(flag[A]) \rightarrow labeling_A$
- ma in questa maniera si avrebbe che $(label[B], B) << (label[A], A)$ (contraddizione)
- q.e.d.

Sommario

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?
- Si. O si ha un overflow, oppure si resetta a 0

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?
- Si. O si ha un overflow, oppure si resetta a 0
  - la correttezza di Bakery non è più assicurata

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?
- Si. O si ha un overflow, oppure si resetta a 0
  - la correttezza di Bakery non è più assicurata
- Possibile usare *timestamp* limitati superiormente per evitare questo problema (costruzione sequenziale e concorrente)

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?
- Si. O si ha un overflow, oppure si resetta a 0
  - la correttezza di Bakery non è più assicurata
- Possibile usare *timestamp* limitati superiormente per evitare questo problema (costruzione sequenziale e concorrente)
- Limiti di Bakery: lettura/scrittura da n locazioni

ALCUNI COMMENTI FINALI

- La crescita dei ticket: un problema?
- Si. O si ha un overflow, oppure si resetta a 0
  - la correttezza di Bakery non è più assicurata
- Possibile usare *timestamp* limitati superiormente per evitare questo problema (costruzione sequenziale e concorrente)
- Limiti di Bakery: lettura/scrittura da n locazioni
- Lower bound di n locazioni necessarie per un algoritmo di Lock deadlock-free