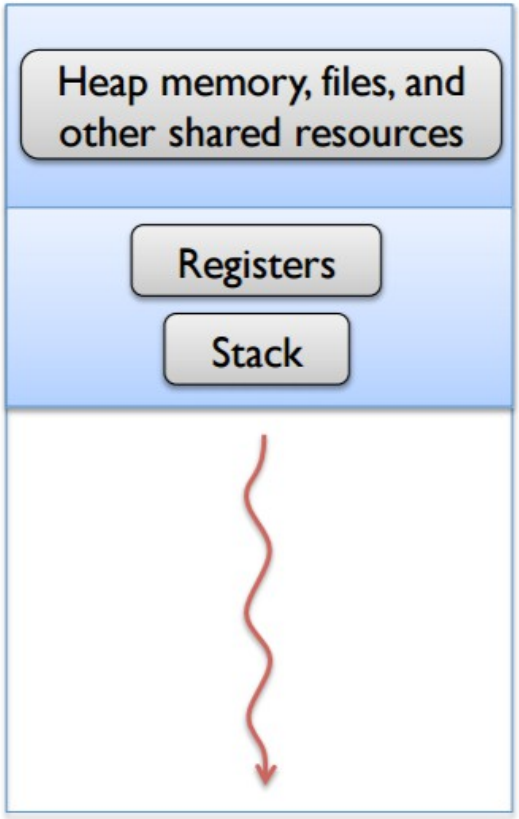
Laboratorio di Reti
Lezione 3
Sincronizzare esplicita: Condition Variables

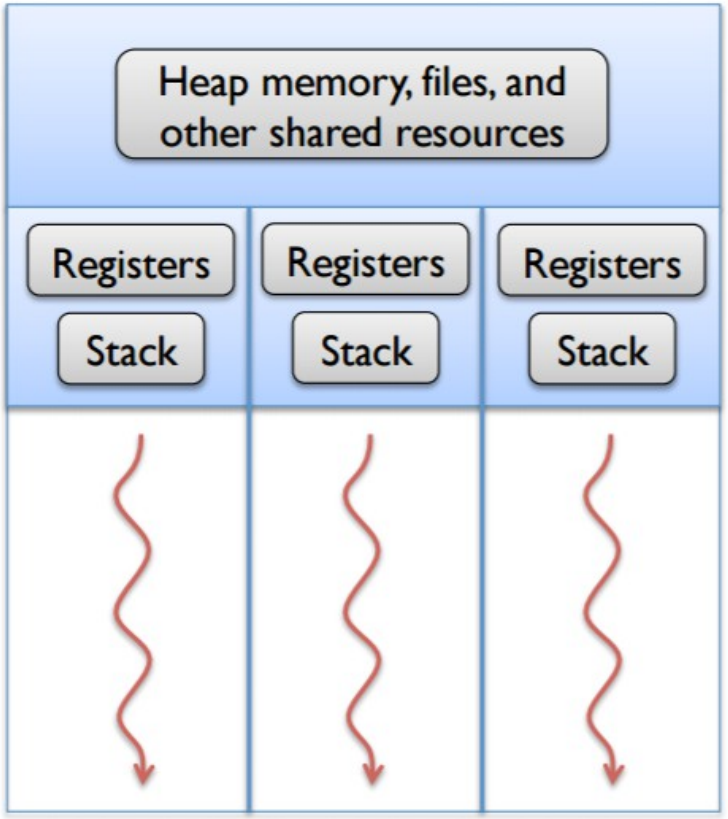
01/10/2020

Laura Ricci

THREADS: RECAP



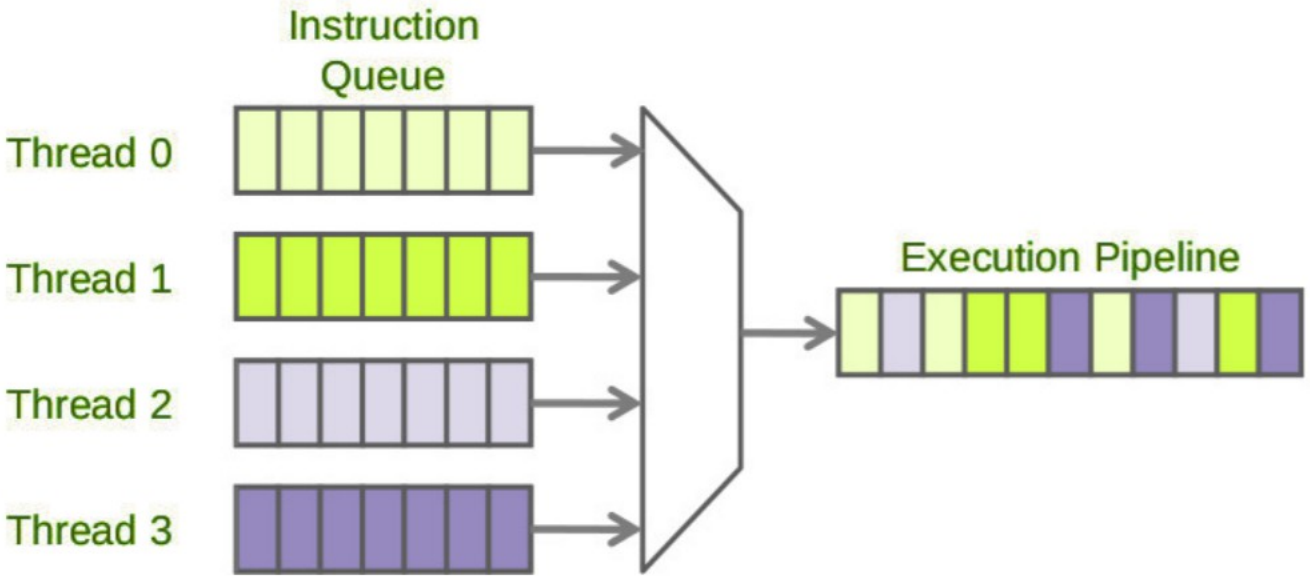
Single-threaded



Multi-threaded

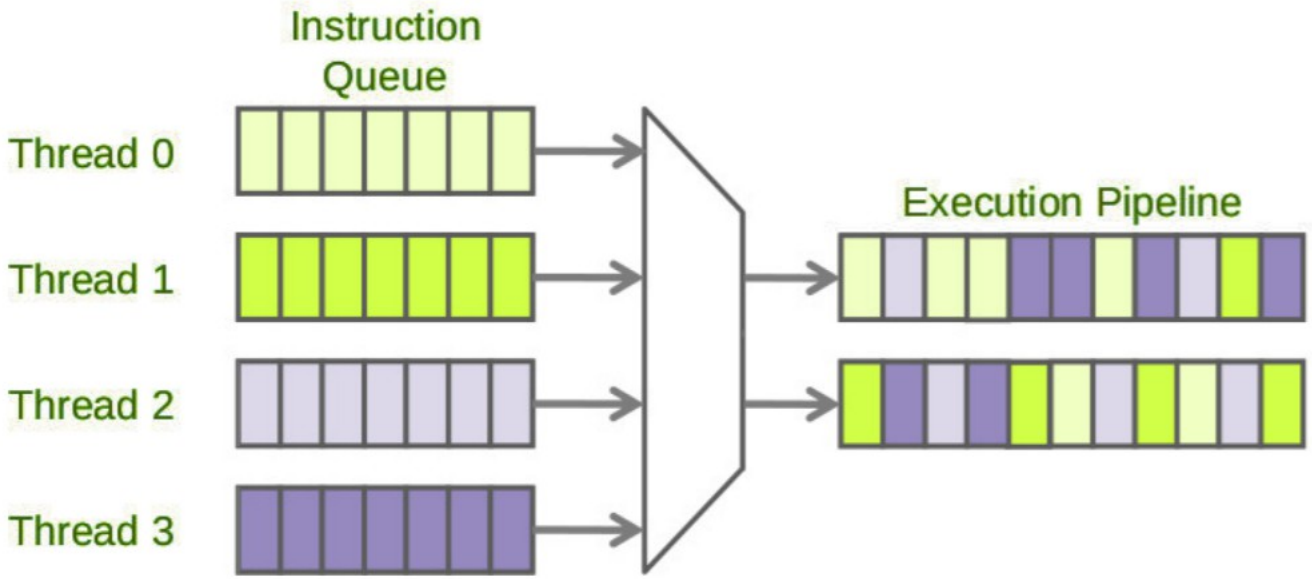
THREADS: RECAP

- diversi threads in esecuzione su single core
- ogni thread viene eseguito per un breve intervallo di tempo e poi si sospende.
- effetto complessivo: come se i thread venissiro eseguiti in parallelo

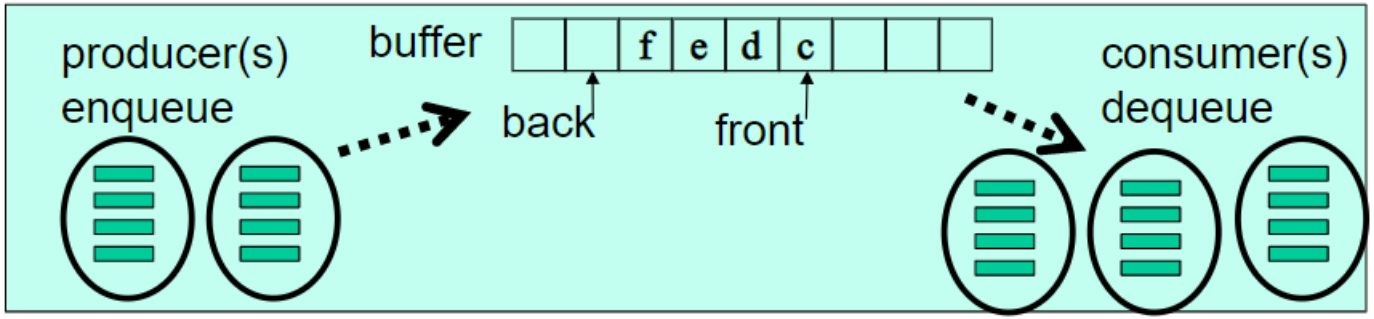


THREADS: RECAP

Se il processore è dotato di più cores (le moderne CPUs hanno spesso 2,4 ed anche 8 cores), i thread possono essere assegnati a diversi cores e vengono eseguiti in parallelo



IL PROBLEMA DEL PRODUTTORE CONSUMATORE



- un classico problema che descrive due (o più thread) che condividono un buffer, di dimenzione fissata, usato come una coda
  - il produttore P produce un nuovo valore, lo inserisce nel buffer e torna a produrre valori
  - il consumatore C consuma il valore (lo rimuove dal buffer) e torna a richiedere valori
  - garantire che il produttore non provi ad aggiungere un dato nelle coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota
- generalizzazione per più produttori e più consumatori

PRODUTTORE CONSUMATORE: SINCRONIZZAZIONE

- l'interazione esplicita tra threads avviene in JAVA mediante l'utilizzo di oggetti condivisi
  - la coda che memorizza i messaggi scambiati tra P e C è un oggetto condiviso tra di essi

- necessari costrutti per sospendere un thread T quando una condizione non è verificata e riattivare T quando diventa vera
  - il produttore si sospende se la coda è piena
  - si riattiva quando c'è una posizione libera

- due tipi di sincronizzazione:
  - implicita: la mutua esclusione sull'oggetto condiviso è garantita dall'uso di lock (implicite o esplicite)
  - esplicita: occorrono altri meccanismi

PRODUTTORE CONSUMATORE: SINCRONIZZAZIONE

una ipotesi importante:

- si utilizzano buffer con dimensione finita
  - una `ArrayList` la cui dimensione massima è prefissata
  - oppure un vettore di dimensione limitata
- non si utilizzano strutture dati sincronizzate di JAVA
- anche se si utilizzasse una coda di dimensione illimitata (ad esempio una `LinkedBlockingQueue`) sarebbe comunque necessaria la sincronizzazione per coda vuota

PRODUTTORE/CONSUMATORE IN JAVA

```
import java.util.*;
import java.util.concurrent.locks.*;
public class MessageQueue {
    private int bufferSize;
    private List<String> buffer = new ArrayList<String>();
    private ReentrantLock l = new ReentrantLock();
    public  MessageQueue(int bufferSize){
       if(bufferSize<=0)
           throw new IllegalArgumentException("Size is illegal.");
       this.bufferSize = bufferSize; }
    public  boolean isFull() {
       return buffer.size() == bufferSize; }
    public boolean isEmpty() {
       return buffer.isEmpty(); }
```

PRODUTTORE_ CONSUMATORE: STARVATION

```
public void put(String message)
    { l.lock();
     while (isFull()) { }
     buffer.add(message);
     l.unlock(); }
public String get()
   { l.lock();
     while (isEmpty()) { }
     String message = buffer.remove(0);
     l.unlock();
    return message;}}}
```

ATTENZIONE: QUESTA SOLUZIONE NON E' CORRETTA!!

- il thread che acquisisce la lock e non può effettuare l'operazione non rilascia la lock
  - altri thread non possono rendere la condizione verificata
  - accesso bloccato per altri thread

PRODUTTORE CONSUMATORE: SPIN LOCK - I

```
public void put (String message)
   {l.lock();
    while (isFull()) {
       l.unlock();
       l.lock(); }
     buffer.add(message);
     l.unlock(); }
  public String get()
     {l.lock();
      while (isEmpty()) {
         l.unlock();
         l.lock(); }
      String message = buffer.remove(0);
      l.unlock();
      return message; }}
```

- spin-lock
  - attesa attiva
  - spreco di risorse computazionali
- la correttezza della soluzione dipende dallo schedulatore

PRODUTTORE CONSUMATORE: SPIN LOCK - 2

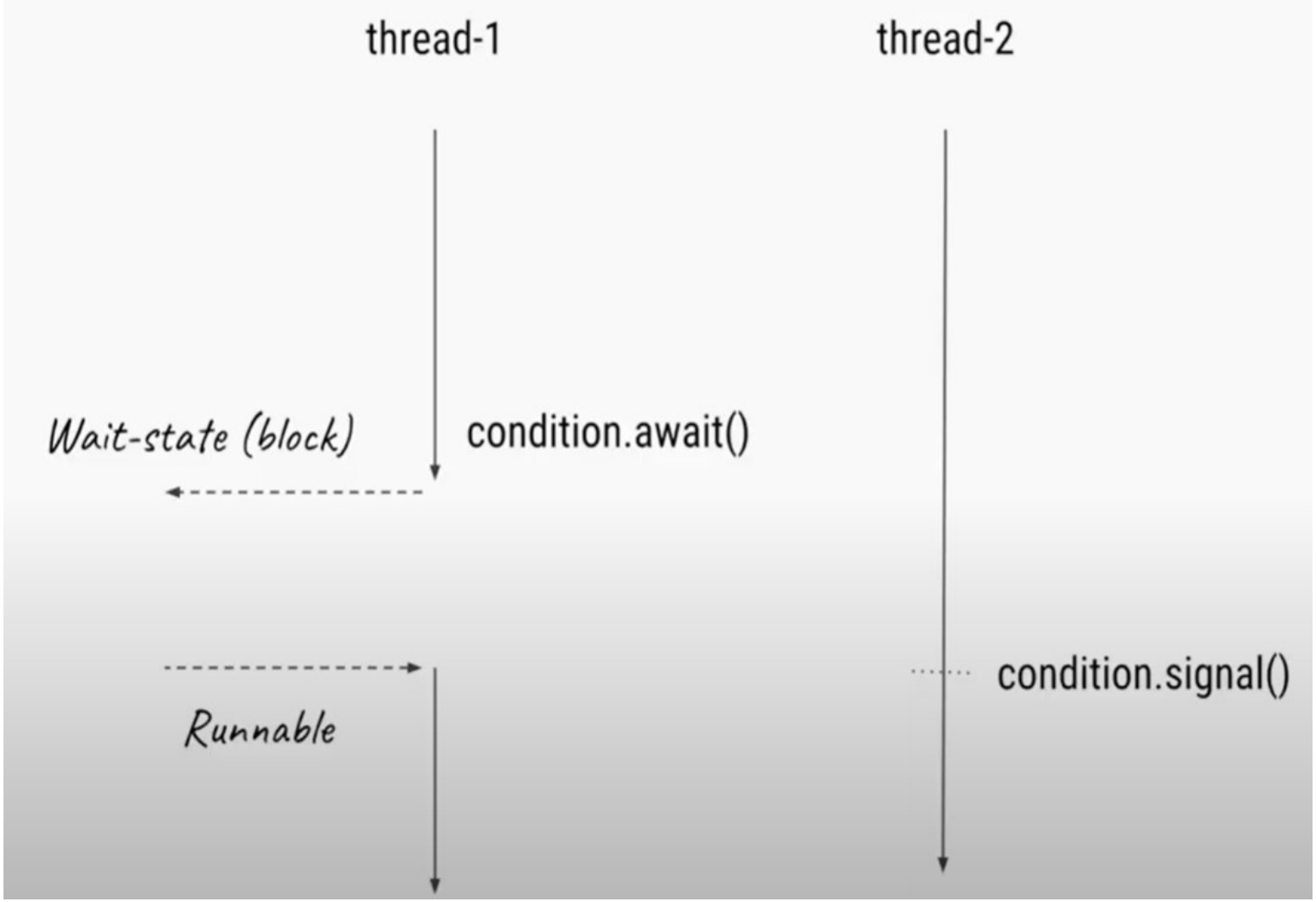
```
public void put (String message)
  { l.lock();
    while (isFull()) {
      l.unlock();
      Thread.yield();
      l.lock(); }
     buffer.add(message);
     l.unlock(); }
public String get()
   { l.lock();
     while (isEmpty()) {
       l.unlock();
       Thread.yield();
       l.lock(); }
      String message = buffer.remove(0);
      l.unlock();
      return message; }}
```

- spin-lock con rilascio volontario del processore
- ancora attesa attiva
- correttezza dipende dalla implementazione dell `yield()`

THREAD COOPERATION: MECCANISMI

- necessari meccanismi
  - che consentano di definire un insieme di condizioni sullo stato dell'oggetto condiviso
  - che consentano la sospensione/riattivazione dei threads sulla base del valore di queste condizioni
- una possibile implementazione:
  - definizione di variabili di condizione
  - metodi per la sospensione su queste variabili
  - definizione di code associate alle variabili in cui memorizzare i threads sospesi
- soluzione alternativa: meccanismi di monitoring ad alto livello (in una lezione successiva)

VARIABILI DI CONDIZIONE: L'IDEA



VARIABILI DI CONDIZIONE IN JAVA

- sono associate ad una lock
- permettono ai thread di controllare se una condizione sullo stato della risorsa è verificata o meno e
  - se la condizione è falsa, rilasciano la lock(), sospendono ed inseriscono il thread in una coda di attesa per quella condizione
  - risvegliano un thread in attesa quando la condizione risulta verificata
- solo dopo aver acquisito la lock su un oggetto è possibile sospendersi su una variabile di condizione, altrimenti viene generata una `IllegalMonitorException`
- quindi, la JVM mantiene più code
  - una per i threads in attesa di acquisire la lock
  - una per ogni variabile di condizione

VARIABILI DI CONDIZIONE IN SINTESI

```
private Lock lock = new ReentrantLock();
private Condition conditionMet = lock.newCondition();
public void method1 () throws InterruptedException{
        lock.lock();
        try { conditionMet.await(); //sospensione
             //l'esecuzione riprende da questo punto
             //operazioni che dipendevano dalla verifica della condizione
             } finally { lock.unlock(); }
         }
public void method2() {
        lock.lock();
        try {
             //operazioni che rendono valida la condizione
             conditionMet.signal();}
         finally {lock.unlock(); }
        }
```

L'INTERFACCIA CONDITION

- definisce i metodi per sospendere un thread e per risvegliarlo
- le condizioni sono istanze di una classe che implementa questa interfaccia

```
interface Condition {
    void await()
    boolean await(long time, TimeUnit unit )
    long awaitNanos(long nanosTimeout)
    void awaitUninterruptibly()
    boolean awaitUntil(Date deadline)
    void signal();
    void signalAll();}
```

PRODUTTORE/CONSUMATORE CON CONDIZIONI

```
public class Messagesystem {
  public static void main(String[] args) {
        MessageQueue queue = new MessageQueue(10);
        new Producer(queue).start();
        new Producer(queue).start();
        new Producer(queue).start();
        new Consumer(queue).start();
        new Consumer(queue).start();
        new Consumer(queue).start();
         }
   }
```

- Nota: la coda ad ogni thread, quando viene invocato il costruttore

IL PRODUTTORE

```
import java.util.concurrent.locks.*;
public class Producer extends Thread {
    private int count = 0;
    private MessageQueue queue = null;
    public Producer(MessageQueue queue){
        this.queue = queue;
        }
    public void run(){
        for(int i=0;i<10;i++){
            queue.produce ("MSG#"+count+Thread.currentThread());
            count++;
        }}}
```

IL CONSUMATORE

```
public class Consumer extends Thread {
   private MessageQueue queue = null;

   public Consumer(MessageQueue queue){
       this.queue = queue;
                                       }
   public void run(){
      for(int i=0;i<10;i++){
         Object o=queue.consume();
         int x = (int)(Math.random() * 10000);
         try{
            Thread.sleep(x);
            }catch (Exception e){};  } } }
```

LA CODA

```
import java.util.concurrent.locks.*;
public class MessageQueue {
    final Lock lockcoda;
    final Condition notFull;
    final Condition notEmpty;
    int putptr, takeptr, count;
    final Object[] items;

    public MessageQueue(int size){
        lockcoda = new ReentrantLock();
        notFull  = lockcoda.newCondition();
        notEmpty = lockcoda.newCondition();
        items = new Object[size];
        count=0;putptr=0;takeptr=0;}
```

LA CODA

```
public void produce(Object x)  throws InterruptedException {
    lockcoda.lock();
    try{
       while (count == items.length)
              notFull.await();
        // gestione puntatori coda
        items[putptr] = x; putptr++;++count;
        if (putptr == items.length) putptr = 0;
        System.out.println("Message Produced"+x);
        notEmpty.signal();
    }
    finally {lockcoda.unlock();
   }
  }
```

LA CODA

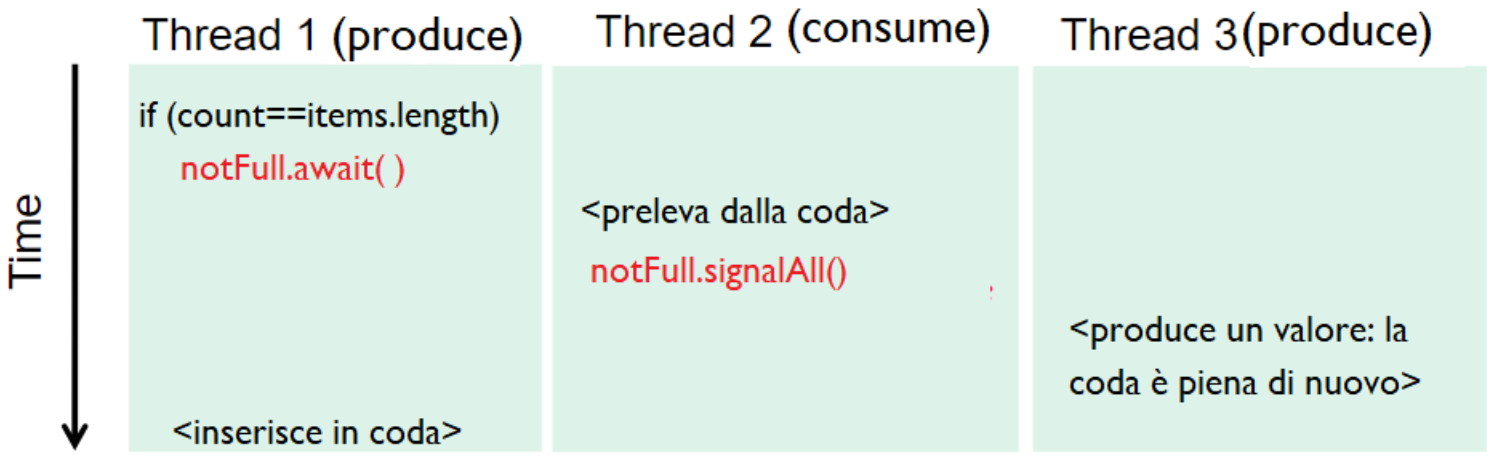
```
public Object consume() throws InterrupedExeceptions {
    lockcoda.lock();
    try{
       while (count == 0)
             notEmpty.await();}
       \\ gestione puntatori coda
       Object data  = items[takeptr]; takeptr=takeptr+1; --count;
       if (takeptr == items.length) {takeptr = 0};
       notFull.signal();
       System.out.println("Message Consumed"+x);
       return x;}
    finally
        {lockcoda.unlock(); }}}
```

PRODUTTORE/CONSUMATORE CON CONDIZIONI

```
Message ProducedMSG#0Thread[Thread-2,5,main]
Message ProducedMSG#0Thread[Thread-0,5,main]
Message ProducedMSG#0Thread[Thread-1,5,main]
Message ProducedMSG#1Thread[Thread-2,5,main]
Message ProducedMSG#1Thread[Thread-0,5,main]
Message ProducedMSG#1Thread[Thread-1,5,main]
Message ProducedMSG#2Thread[Thread-2,5,main]
Message ConsumedMSG#0Thread[Thread-2,5,main]
Message ProducedMSG#2Thread[Thread-0,5,main]
Message ProducedMSG#2Thread[Thread-1,5,main]
Message ConsumedMSG#0Thread[Thread-0,5,main]
Message ConsumedMSG#0Thread[Thread-1,5,main]
Message ProducedMSG#3Thread[Thread-2,5,main]
Message ProducedMSG#3Thread[Thread-0,5,main]
Message ProducedMSG#3Thread[Thread-1,5,main]
Message ProducedMSG#4Thread[Thread-2,5,main]
Message ConsumedMSG#1Thread[Thread-2,5,main]
……..
```

IL PROBLEMA DELLE SIGNAL SPURIE

```
if (count == items. length)
    notfull.await();
// inserisci elemento
```



cosa accade se sostituisco il while con un if nella guardia che controlla se la coda è piena?

IL PROBLEMA DELLE SIGNAL SPURIE

- il problema delle notifiche "spurie"
- tra il momento in cui ad un thread arriva una notifica ed il momento in cui riacquisisce la lock, la condizione può diventare di nuovo falsa
- regola "d'oro"
  - ricontrollare sempre la condizione dopo aver acquisito la lock
  - inserire la wait in un ciclo while
  - possibile evitarlo solo in casi particolari

OTTIMIZZARE LA GRANULARITA' DELLE LOCK

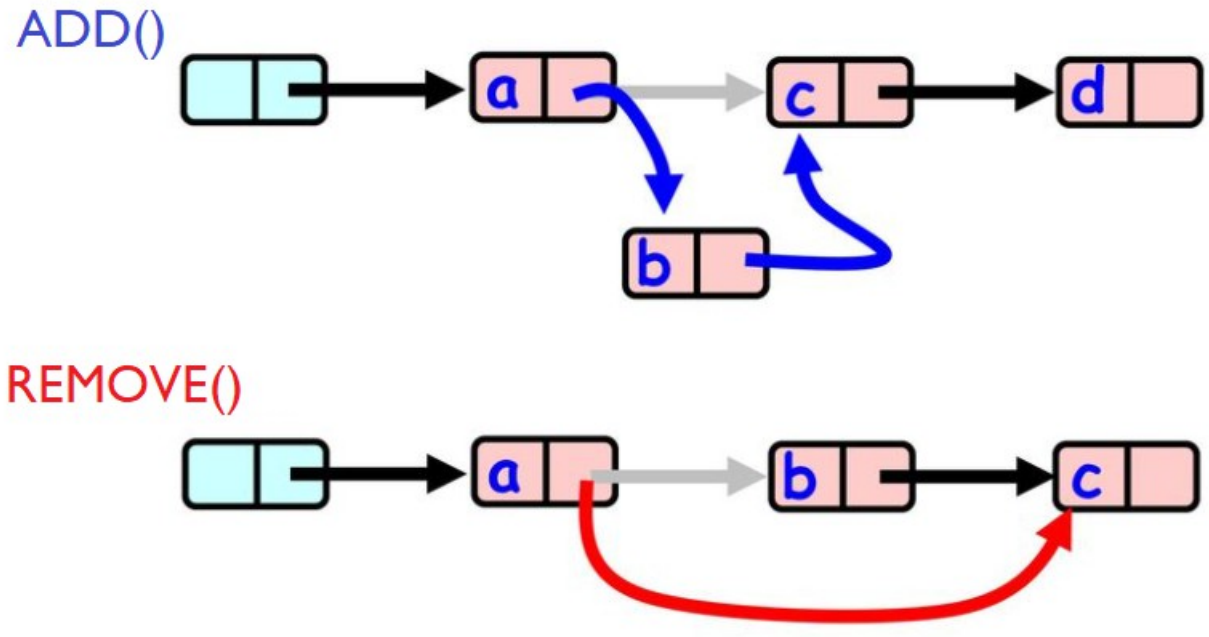
- accesso concorrente a strutture dati
  - thread safeness la struttura rimanere consistente quando più thread accedono concorrentemente

- linked list
  - per ogni elemento puntatori all'elemento successivo ed al precedente
  - inserzione ed eliminazione di elementi dalla lista
  - come garantire la thread safeness?

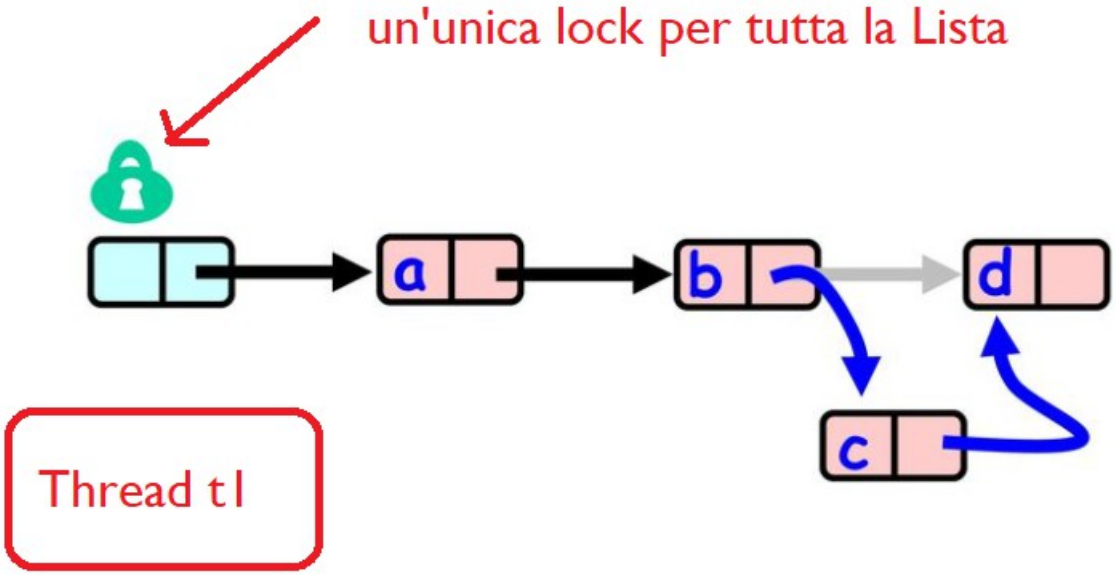
- coarse grain lock
  - una singola lock per tutta la struttura
  - inefficiente: nessun thread può accedere alla struttura mentre un altro la sta modificando

- hand-over-hand locking
  - mutua esclusion solo su piccole porzioni della lista, permettendo ad altri thread l'accesso ad elementi diversi della struttura

CONCURRENT DATA STRUCTURES: LISTE



COARSE GRAIN LOCK



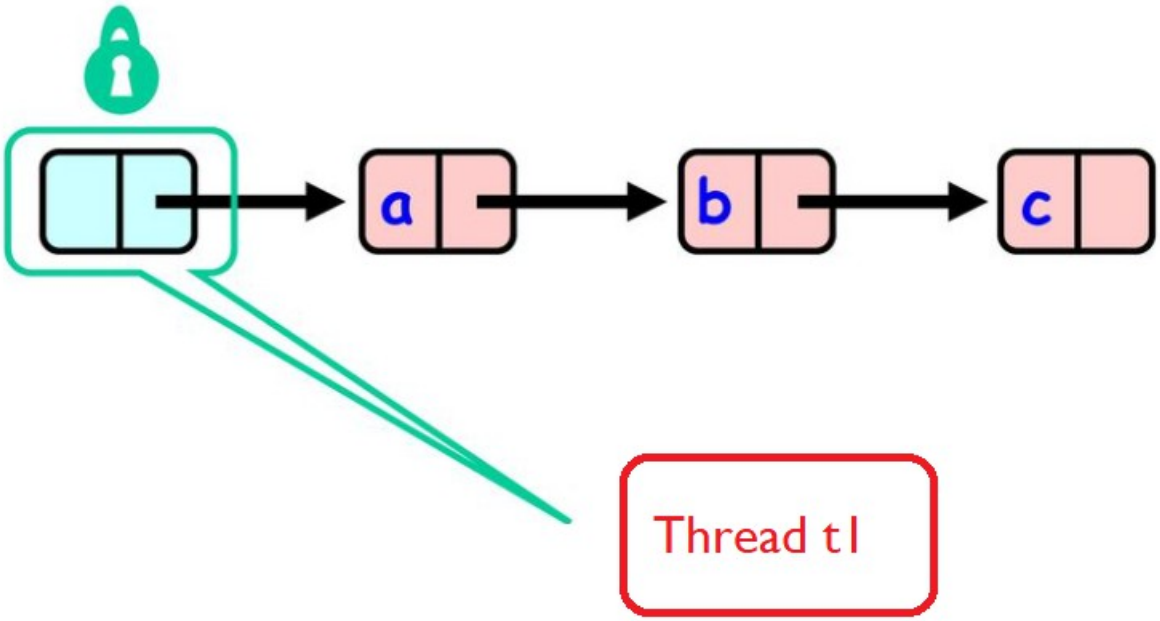
Vantaggi: semplicità, facile dimostrare correttezza

Svantaggi: contention, la struttura diventa un bottleneck

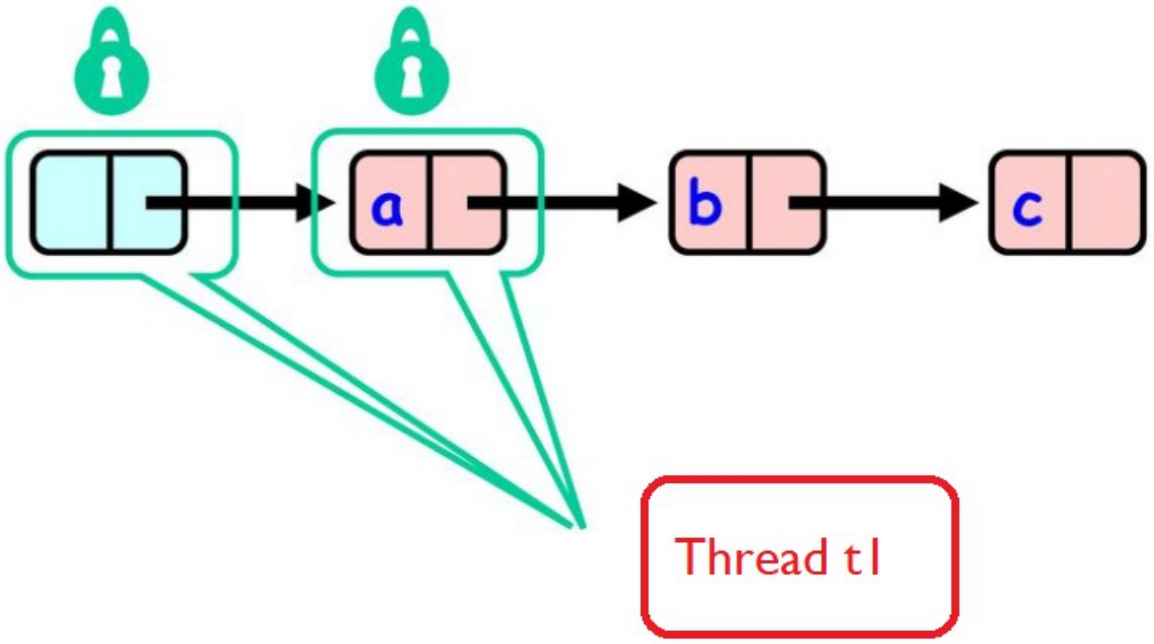
FINE GRAIN LOCKS

- usare lock multiple per "proteggere" parti diverse di una struttura dati, invece di una singola lock()
- suddividere un oggetto in più componenti che possono essere accedute in modo concorrente
  - associare una lock diversa ad ogni parte di quell'oggetto
- "hand-over-hand locking"
  - permettere operazioni concorrenti su parti diverse della struttura
  - acquisire le lock sulle diverse parti della struttura dinamicamente
- semplice per alcune strutture, più complesso per altre
- idea alla base delle concurrent collections di JAVA

HAND-OVER-HAND LOCKING

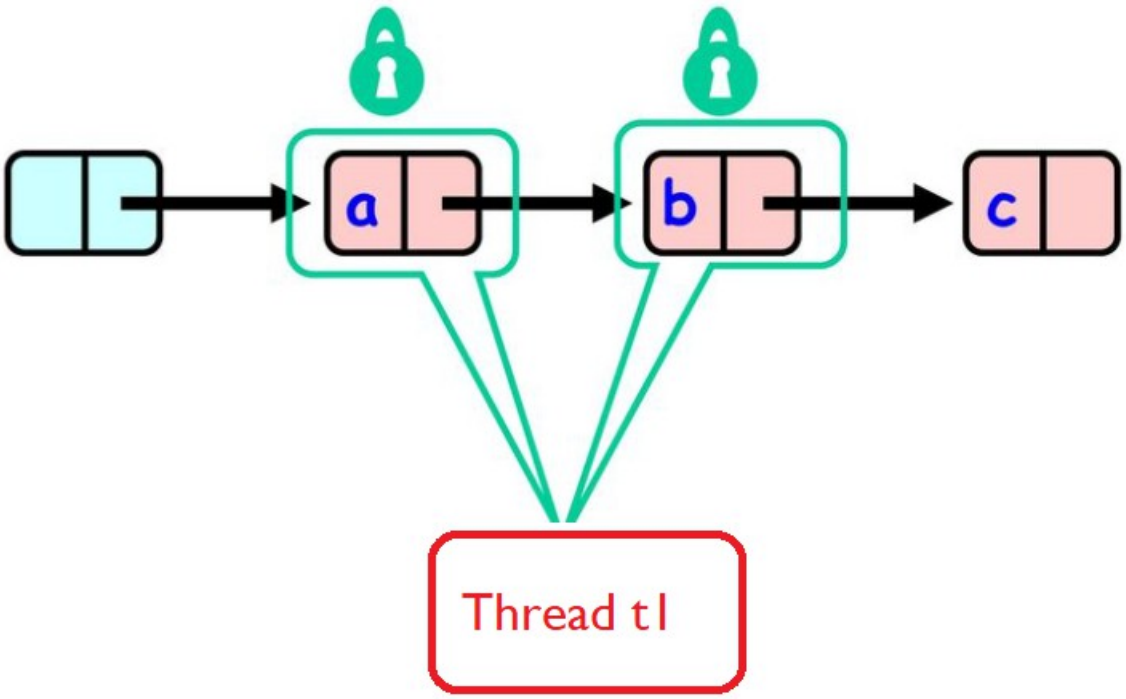


HAND-OVER-HAND LOCKING



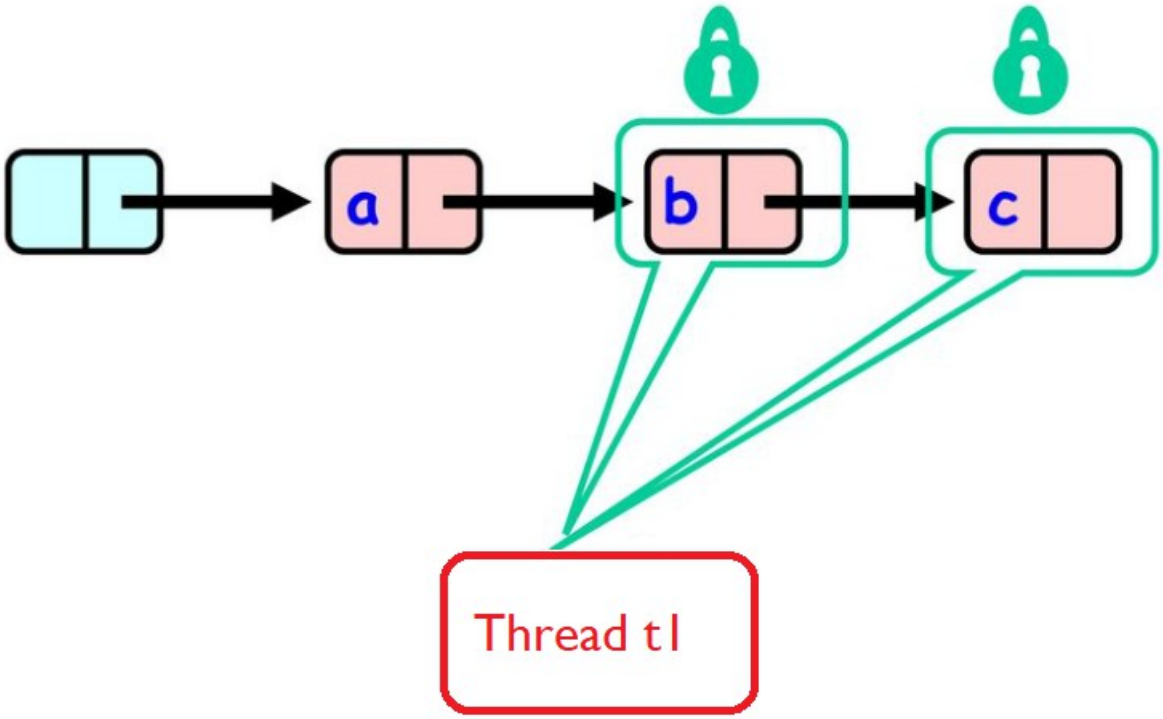
- il thread acquisisce la lock su due elementi successivi della lista
- vedreemo in seguito che questo garantirà la thread safeness

HAND-OVER-HAND LOCKING



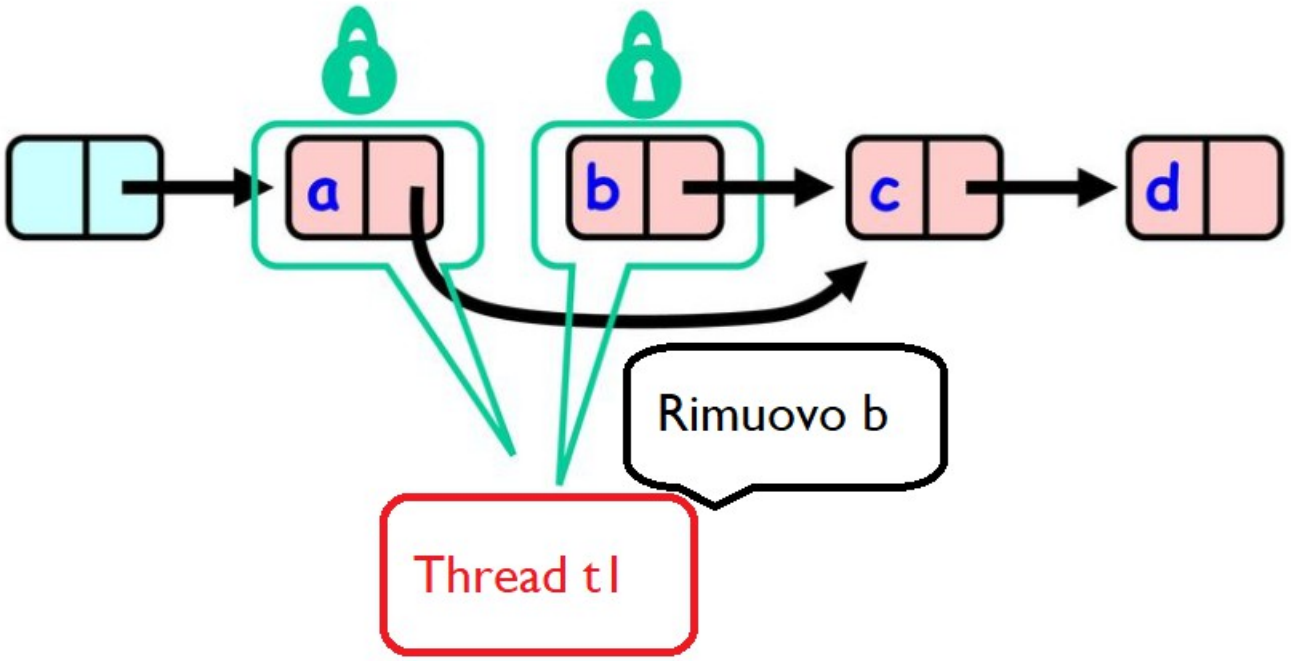
- il thread acquisisce la lock su due elementi successivi della lista
- vedreemo in seguito che questo garantirà la thread safeness

HAND-OVER-HAND LOCKING

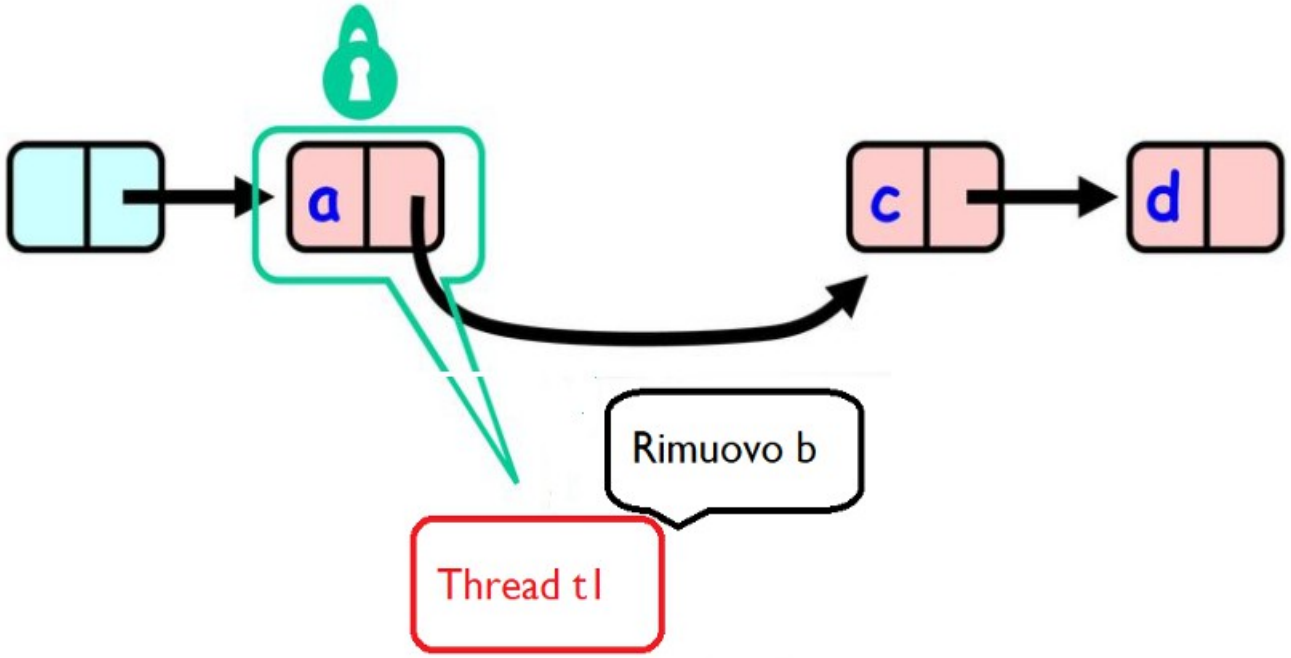


- il thread acquisisce la lock su due elementi successivi della lista
- vedreemo in seguito che questo garantirà la thread safeness

HAND OVER LOCK: RIMOZIONE

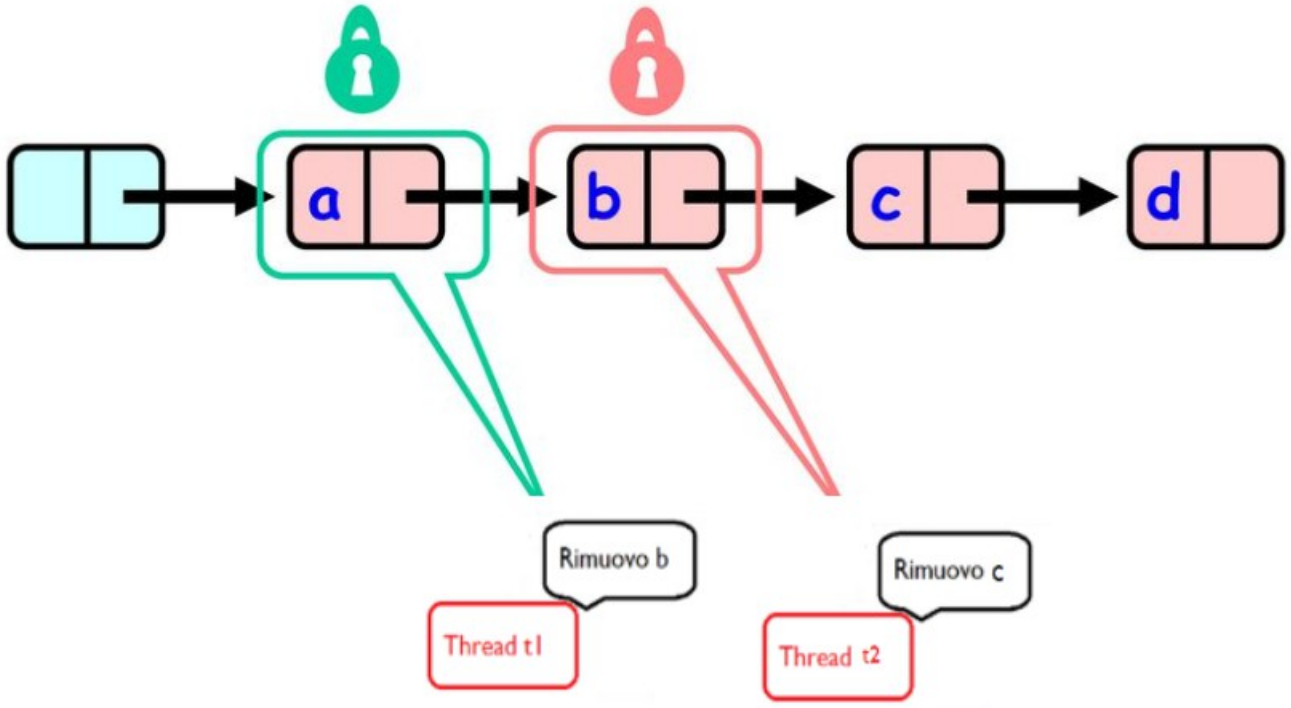


HAND OVER LOCK: RIMOZIONE



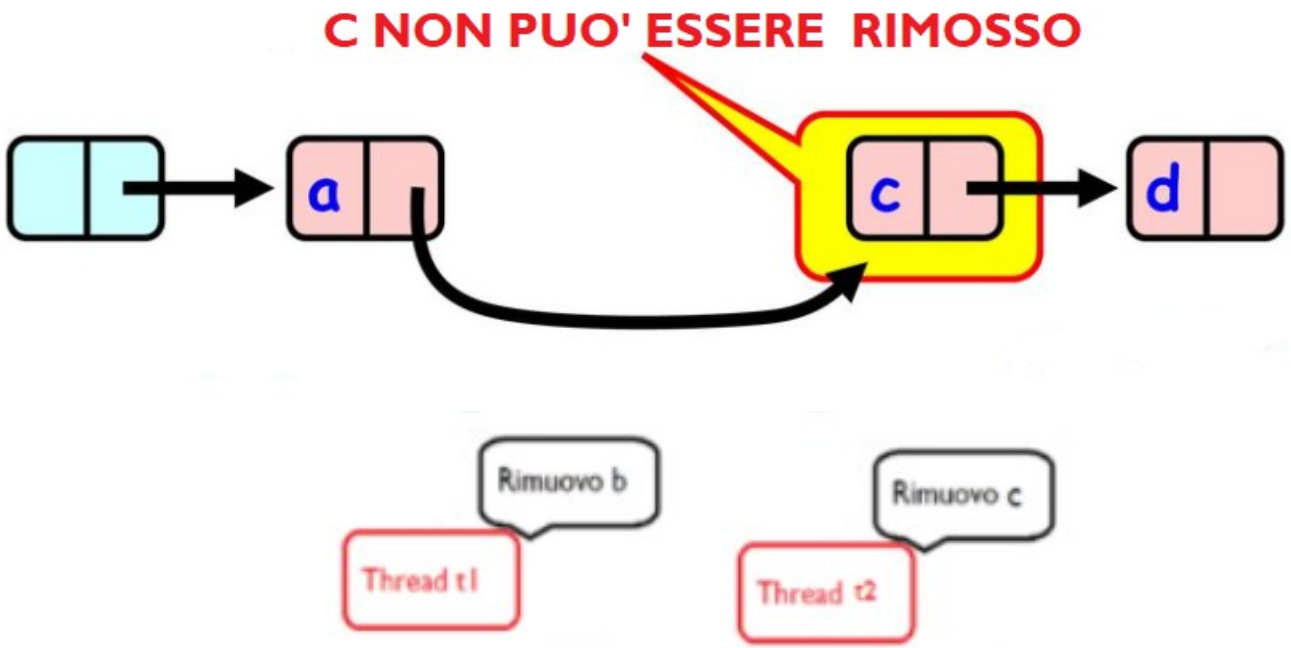
- cosa accadrebbe se t1 non avesse acquisito la lock sull'elemento da rimuovere (oltre che sull'elemento precedente)?
- possibile inconsistenza (vedi slide successiva)

HAND OVER LOCK: RIMOZIONI CONCORRENTI



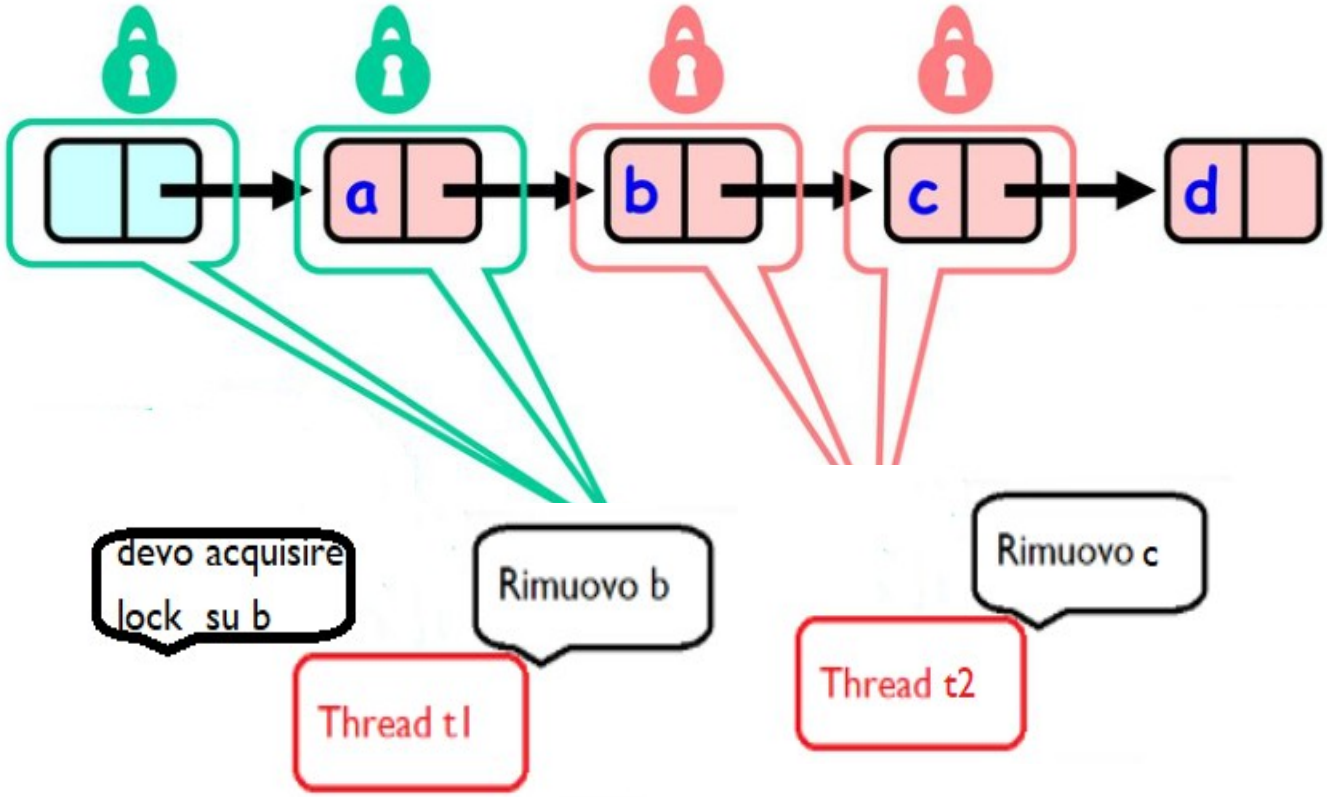
- t1 e t2 accedono concorrentemente alla lista
- ipotesi: ogni thread acquisisce la lock solo sull'elemento che sta visitando
- i due thread rimuovono in parallelo due elementi consecutivi nella lista (b e c)

HAND OVER LOCK: RIMOZIONI CONCORRENTI



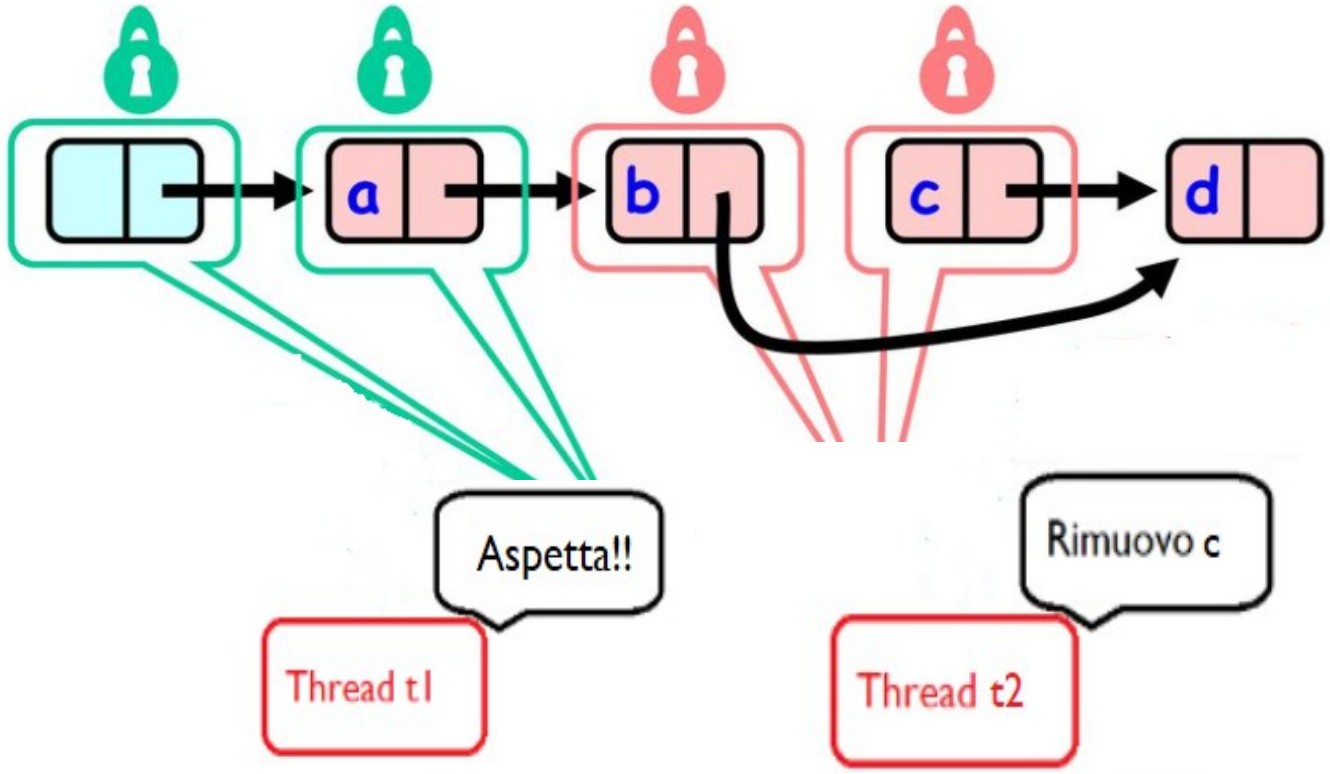
- per rimuovere il nodo c, t2 assegna al predecessore di c (b) il puntatore a d
- contemporaneamente t1 elimina b facendo puntare a direttamente a c "sganciando" così b dalla lista
- la rimozione effettuata da t2 non ha alcun effetto sulla lista

RIMOZIONI CONCORRENTI: SOLUZIONE CORRETTA



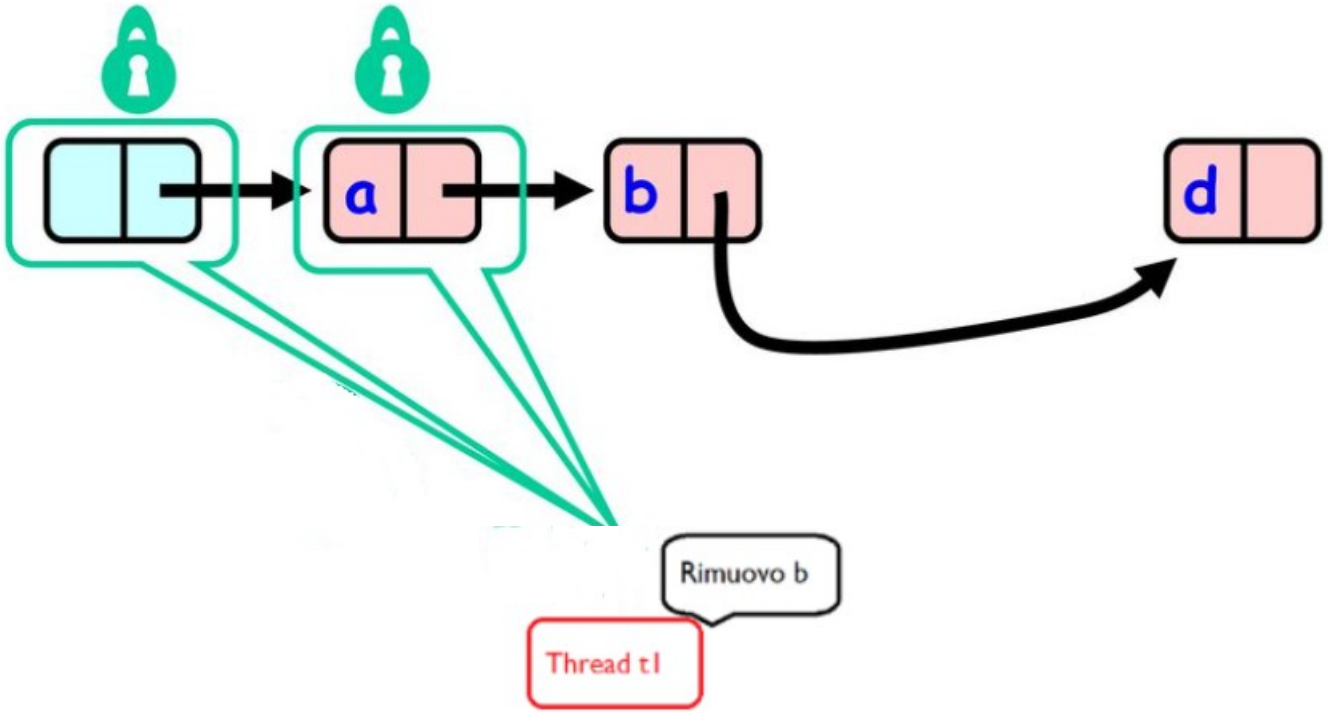
- in questa soluzione ogni thread acquisisce lock su due elementi consecutivi
- t1 vuole rimuovere b, ma non può acquisirne la lock, perchè è stata acquisita da c

RIMOZIONI CONCORRENTI: SOLUZIONE CORRETTA



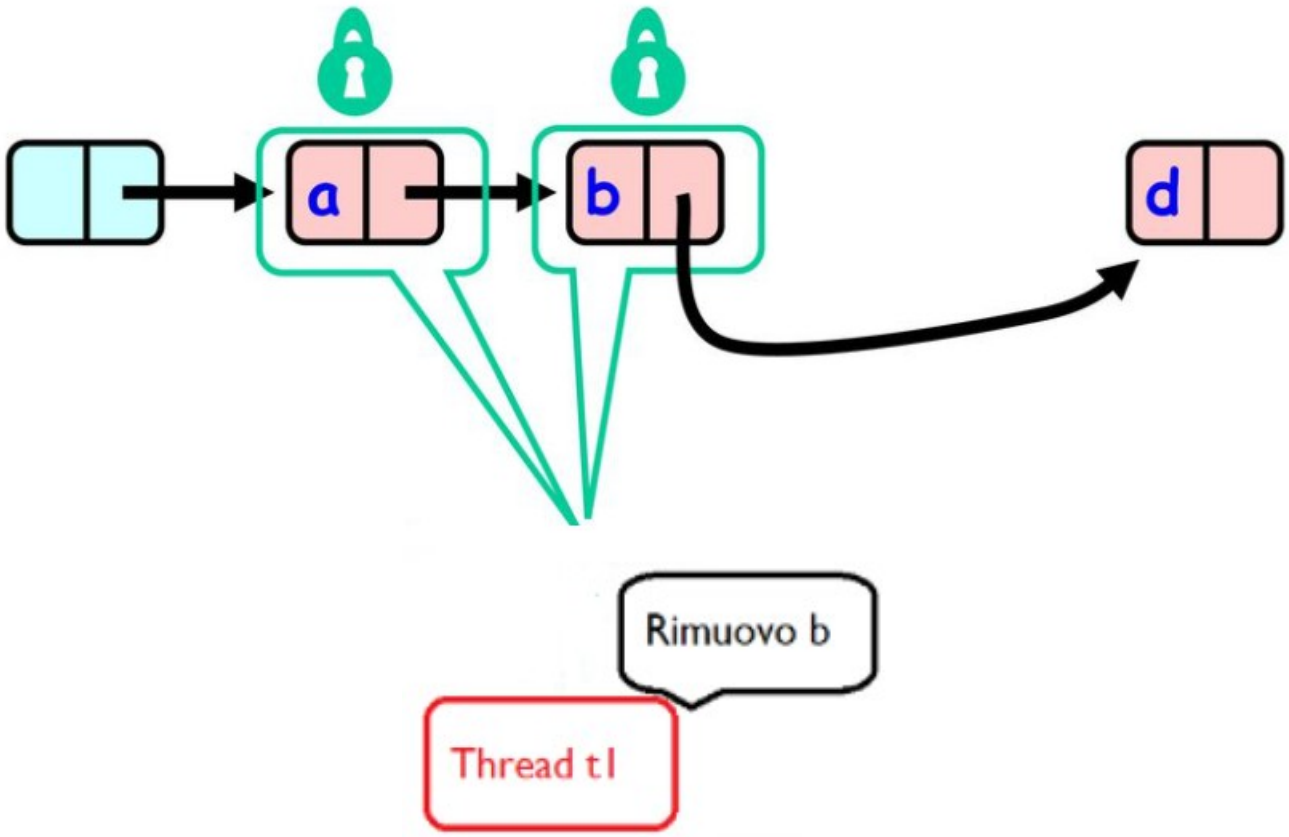
- t1 si sospendere in attesa della lock sull'elemento che vuole eliminare
- t2 può rimuovere c, perchè posside la lock su c e sull'elemento precedente

RIMOZIONI CONCORRENTI: SOLUZIONE CORRETTA



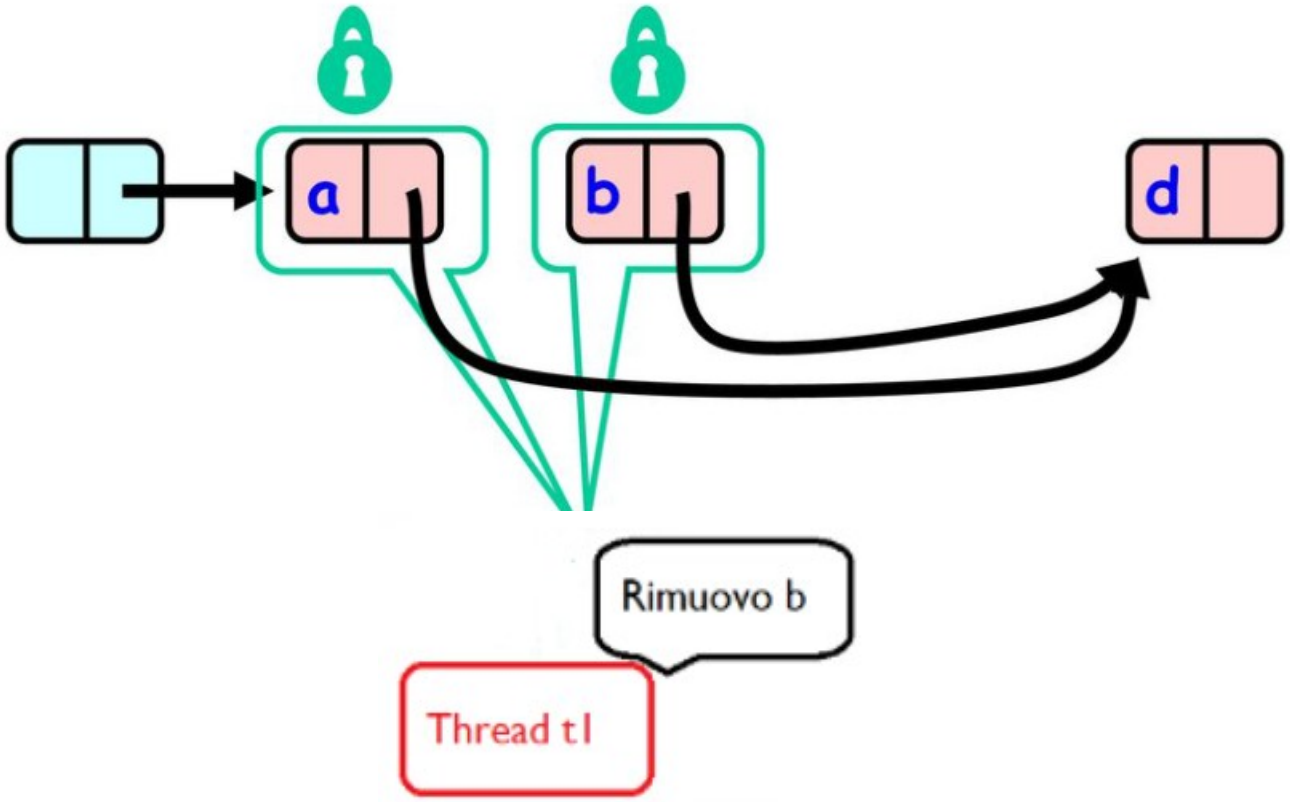
- c è stato rimosso correttamente
- t1 ora può acquisire la lock su b e rimuoverlo

RIMOZIONI CONCORRENTI: SOLUZIONE CORRETTA



- c è stato rimosso correttamente
- t1 ora può acquisire la lock su b e rimuoverlo

RIMOZIONI CONCORRENTI: SOLUZIONE CORRETTA



- c è stato rimosso correttamente
- t1 ora può acquisire la lock su b e rimuoverlo

STATO FINALE DELLA LISTA

- considerazioni analoghe nel caso della inserzione
  - Il thread che vuole inserire deve acquisire la lock sul predecessore e
  - sul successore rispetto all'elemento da inserire

HAND-OVER-HAND LOCKING IN JAVA

```
import java.util.concurrent.locks.ReentrantLock;
public class Node {
    int  value;
    Node prev;
    Node next;
    ReentrantLock lock = new ReentrantLock();
    public Node()
        {   }
    public Node(int value, Node prev, Node next)
        {this.value = value; this.prev = prev; this.next = next; }

     }
```

HAND-OVER-HAND LOCKING IN JAVA

```
public class ConcurrentSortedList {
  private final Node head;
  private final Node tail;
  public ConcurrentSortedList()
     {
        head = new Node(); tail = new Node();
        head.next = tail; tail.prev = head;
        head.value = 0; tail.value= 0;
      }
```

HAND-OVER-HAND LOCKING IN JAVA

```
public void insert(int value) {
   Node current = head; current.lock.lock();
   Node next = current.next; fine = false;
    try {
      while (!fine) {
        next.lock.lock();
        try {
          if (next == tail || next.value < value) {
            Node node = new Node(value, current, next);
            next.prev = node;
            current.next = node;
            fine = true;          }
        } finally { current.lock.unlock(); }
        current = next;
        next = current.next;}
      } finally { next.lock.unlock(); }  }
```

HAND-OVER-HAND LOCKING IN JAVA

```
public int size()
 {
     Node current = tail;
     int count = 0;
     while (current.prev != head) {
         ReentrantLock lock = current.lock;
         lock.lock(); riflettere: serve la lock o può essere
                                  omessa?
         try {
            ++count;
            current = current.prev;
             } finally { lock.unlock(); }
      }
   return count;
 }
```

HAND-OVER-HAND LOCKING IN JAVA

```
public boolean isSorted() {
   Node current = head;
   while (current.next.next != tail) {
     current = current.next;
     if (current.value < current.next.value)
       return false;
   }
   return true;
 }
```

riflettere: serve inserire la lock in qualche punto?

HAND-OVER-HAND LOCKING IN JAVA

```
import java.util.Random;

public class TestThread extends Thread {
      ConcurrentSortedList list;
      final Random random = new Random();
      public TestThread (ConcurrentSortedList l)
            {this.list=l;};
      public void run() {
         for (int i = 0; i < 10000; ++i)
             list.insert(random.nextInt());
      }
   }
```

HAND-OVER-HAND LOCKING IN JAVA

```
import java.util.Random;
public class CountingThread extends Thread {
    ConcurrentSortedList list;
    final Random random = new Random();
    public CountingThread (ConcurrentSortedList l)
        {this.list=l;};
    public void run() {
        while (!interrupted()) {
           System.out.print("\r" + list.size());
           System.out.flush();
        }
      }
     }
```

HAND-OVER-HAND LOCKING IN JAVA

```
import java.util.Random;
public class LinkedList {
public static void main(String[] args) throws InterruptedException
    {final ConcurrentSortedList list = new ConcurrentSortedList();
    Thread t1 = new TestThread(list);
    Thread t2 = new TestThread(list);
    Thread t3 = new CountingThread(list);
    t1.start(); t2.start(); t3.start();
    t1.join(); t2.join();
    t3.interrupt();
    System.out.println("\r" + list.size());
    if (list.size() != 20000)
      System.out.println("*** Wrong size!");
```

ASSIGNMENT 3: GESTIONE LABORATORIO

Il laboratorio di Informatica del Polo Marzotto è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.

b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice i, poichè su quel computer è istallato un particolare software necessario per lo sviluppo della tesi.

c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti.

Nessuno però può essere interrotto mentre sta usando un computer (prosegue nella pagina successiva)

ASSIGNMENT 3: GESTIONE LABORATORIO

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede k volte al laboratorio, con k generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo sleep. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.