

# **Laboratorio di Reti – A** **(matricole pari)**

**Autunno 2021,**  
**instructor: Laura Ricci**





**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Lezione 3**

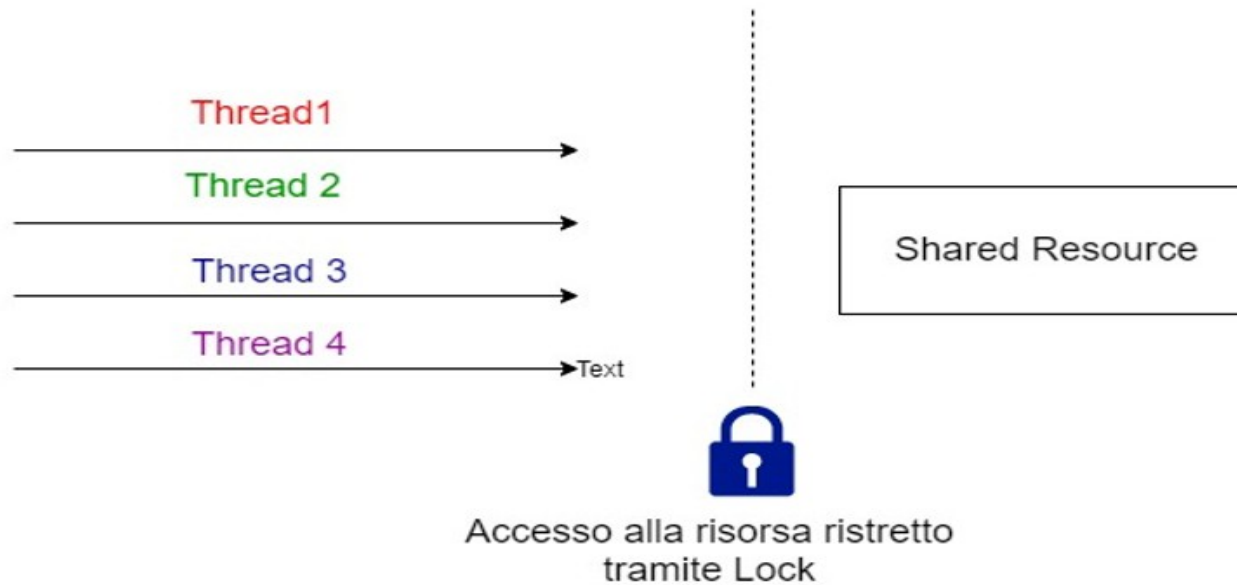
### **ReentrantLock, Condition Variables**

**28/9/2021**

# JAVA.UTIL.CONCURRENT IN JAVA 5

- esecuzione dei thread controllata e indipendente dalla logica dell'applicazione – Executor 
- possibilità di restituire un risultato per un task e lanciare eccezioni 
- classi Lock, variabili di condizione dedicate (questa lezione)
- Concurrent Collections (prossima lezione)
- Semafori, barriere 
- Variabili Atomic 

# MECCANISMI DI SINCRONIZZAZIONE: LOCK



una metafora per la lock: “come la chiave del bagno”

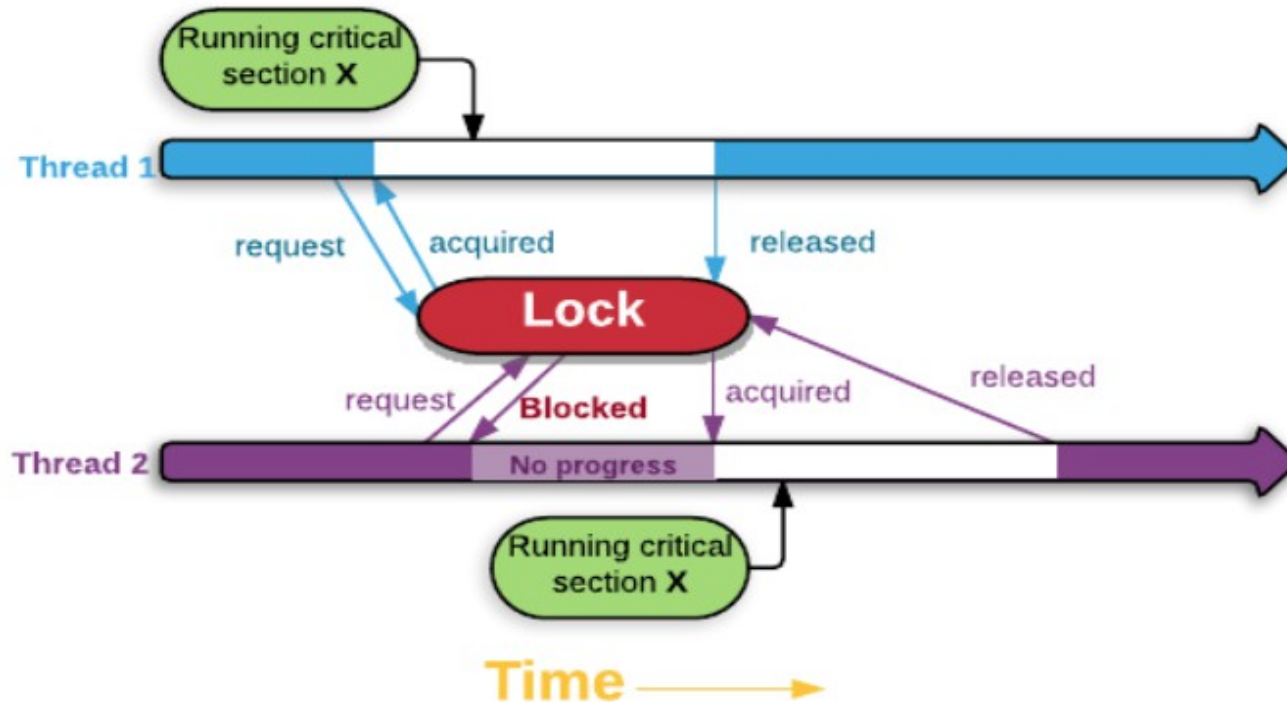
- `chiave.lock()`, prova ad aprire la porta,
  - se non è chiusa, entra e blocca la porta.
  - se è chiusa, aspetta che l'altro esca.
- `chiave.unlock()`, uscita dal bagno
  - rilascia la chiave della porta



# SEZIONI CRITICHE E RACE CONDITIONS

- lock usate per definire “sezioni critiche”
  - mutual exclusion lock (mutex)
  - assicurano che solo un thread per volta possa entrare in una sezione critica
- evitare “race conditions”
  - operazioni concorrenti incontrollate possono corrompere lo stato della risorsa
  - si verificano quando la correttezza del risultato di un programma dipende dall'ordine o dal tempo con cui le operazioni sulla risorsa condivisa sono state eseguite dai thread
    - per un certo ordine, il risultato è corretto, per un altro no

# LOCK E SEZIONI CRITICHE



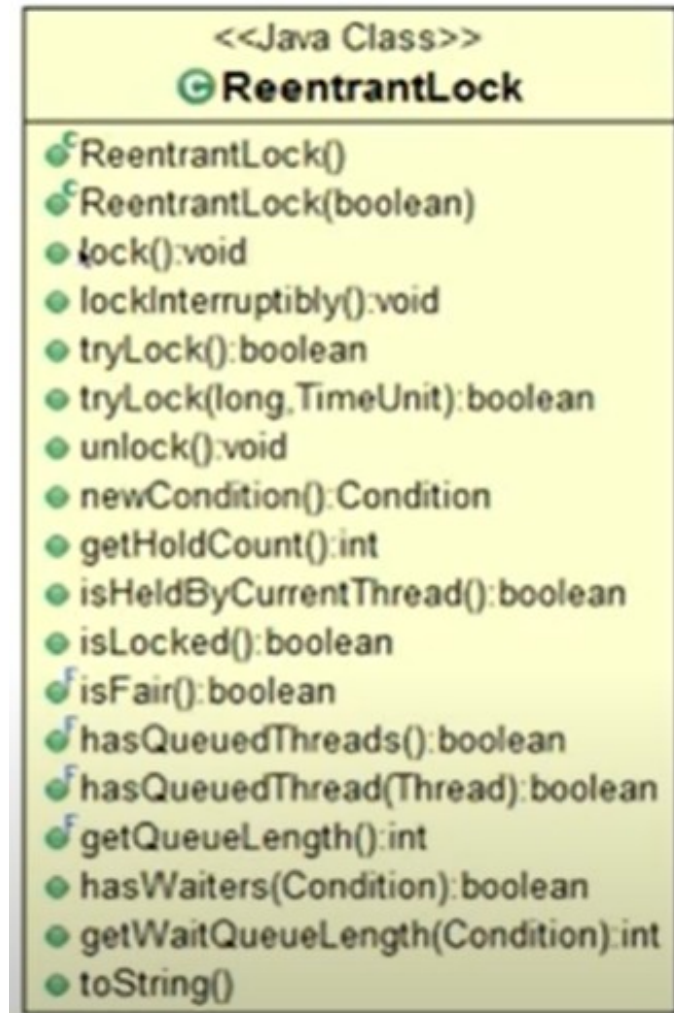
- la lock è un oggetto che può trovarsi in due stati diversi “locked”/”unlocked”,
  - lo stato è impostato con i metodi: `lock()` ed `unlock()`
- la gestione dei thread bloccati dipende dalla politica di fairness
  - **fair lock**: thread bloccati serviti secondo una politica FIFO
  - **non fair lock**: spin lock, attesa attiva, generalmente ottimizzazione hw

# L'INTERFACCIA LOCK

```
interface Lock {  
    void lock();  
    void unlock();  
    void lockInterruptibly()  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit)  
    Condition newCondition() }
```

- è implementata da
  - ReentrantLock
  - ReentrantReadWriteLock.ReadLock
  - ReentrantReadWriteLock.WriteLock
- Alternativa alle Lock: usare lock implicite (prossima lezione): synchronized(o)

# LA CLASSE REENTRANTLOCK

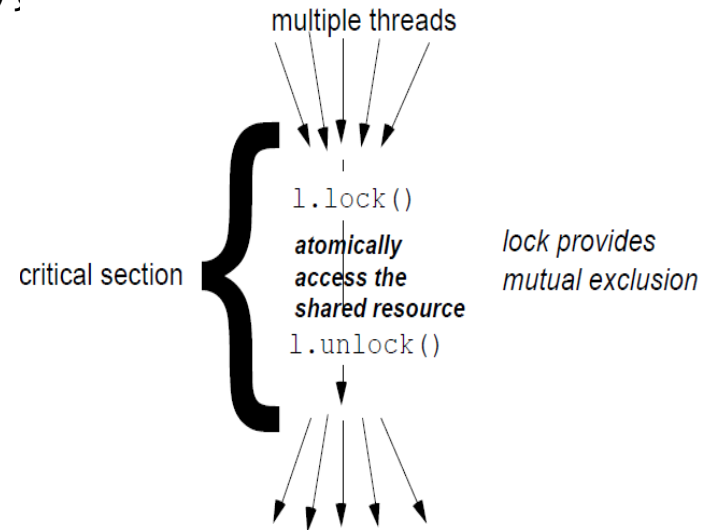


# MUTUAL EXCLUSION PATTERN

```
private static Lock lock = new ReentrantLock();
```

```
private static void accessResource() {  
    try{  
        lock.lock();  
        //access the resource  
    } finally {lock.unlock();}  
};
```

```
public static void main (String args[]) {  
    Thread t1 = new Thread() {public void run() {accessResource();}};  
    t1.start();  
    Thread t2 = new Thread() {public void run() {accessResource();}};  
    t2.start();  
    Thread t3 = new Thread() {public void run() {accessResource();}};  
    t3.start();  
    Thread t4 = new Thread() {public void run() {accessResource();}};  
    t4.start(); }  
}
```



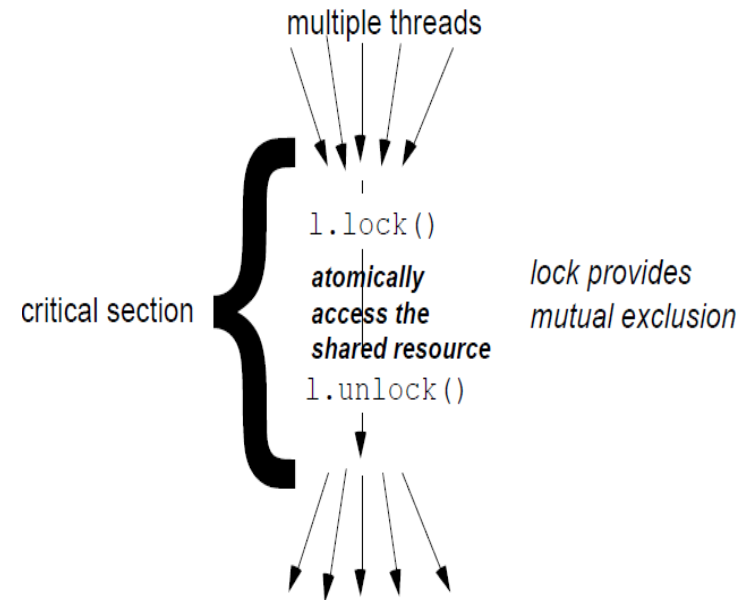


# SINCRONIZZIAMO IL CONTATORE

```
import java.util.concurrent.locks.*;

public class Counter {
    private int count = 0;
    private Lock lock= new ReentrantLock();
    public void increment()
        {try {
            lock.lock();
            this.count++;
        } finally {lock.unlock();}
        }

    public int getCount()
        {try {
            lock.lock();
            return this.count;
        } finally {lock.unlock();
        }}}}
```



- Se la lock sospende il thread, perchè InterruptedException non viene intercettata?
- la risposte nelle slide successive

# MUTUAL EXCLUSION PATTERN

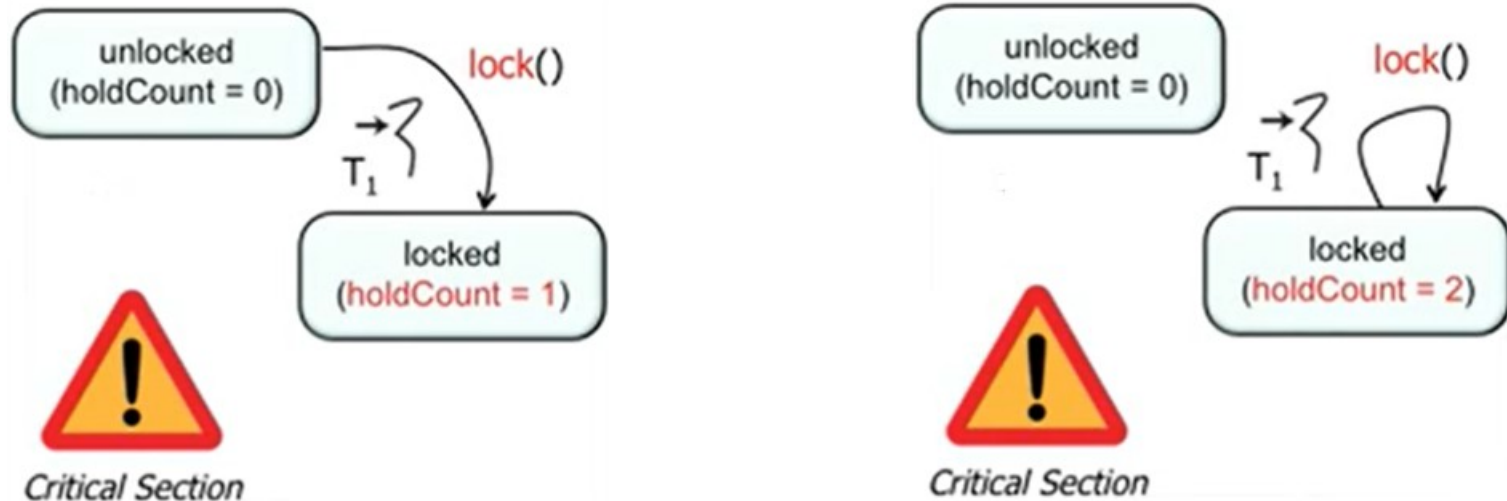
```
private static Lock lock = new ReentrantLock();
```

```
private static void accessResource() {  
    try{  
        lock.lock();  
        //accesso alla risorsa  
    } finally {lock.unlock();}  
};
```

notare l'uso della finally

- cosa accade se viene sollevata un'eccezione non intercettata nel frammento di codice per l'accesso alla risorsa e quella eccezione non è stata intercettata?
- senza la finally, il metodo unlock() non verrebbe mai invocato e la lock mai rilasciata. Nessun thread potrà più accedere alla sezione critica

# PERCHE' "REENTRANT LOCK"?



- permettono ad un thread di invocare il metodo `lock()` più volte sullo stesso oggetto, senza provocare un “self deadlock”
- il numero di `unlock()` deve corrispondere a quello delle `lock()`
- l'implementazione utilizza un contatore
  - incrementato ogni volta che il thread acquisisce la lock e decrementato ogni volta che il thread rilascia la lock
  - la lock viene definitivamente rilasciata quando il contatore diventa 0

# PERCHE' IL NOME REENTRANT?

```
private ReentrantLock = lock new ReentrantLock();  
private static void accessResource() {  
    lock.lock();  
    // aggiorna risorsa  
    if (some condition()) {  
        accessResource();  
    }  
    lock.unlock();}}
```

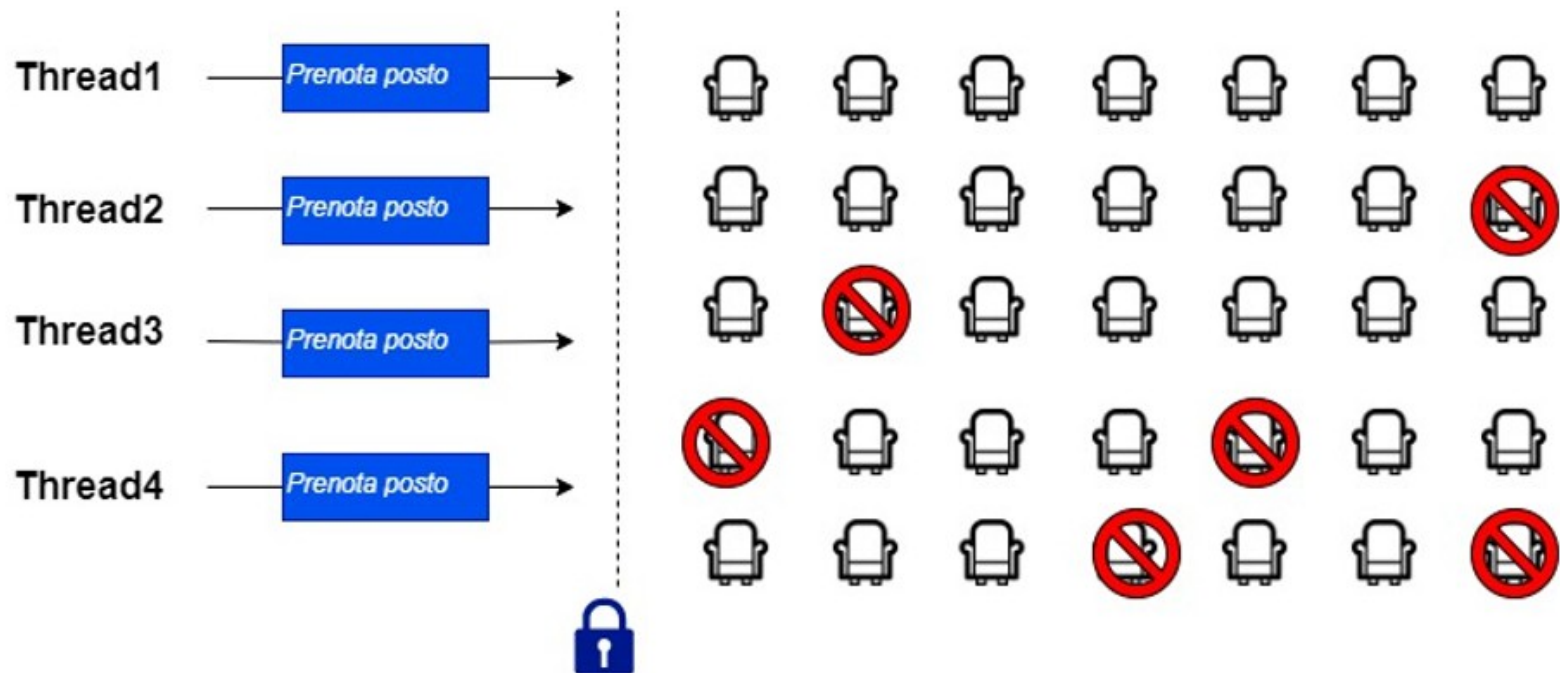
- invocazione ricorsive per l'aggiornamento della risorsa, fino a che non è verificata una certa condizione sulla risorsa
- Reentrant lock perchè il codice “rientra” nel blocco, cercando di riacquisire la lock()
- recursive semantics
  - leggermente meno efficienti
  - POSIX locks non sono di default ricorsive

# LOCK INTERRUPTIBLY

- se un thread è bloccato in attesa di una lock intrinseca, non è possibile “interagirci” in alcun modo, solo se acquisirà la lock e proseguirà l'esecuzione, sarà possibile inviargli una interruzione
- LockInterruptibly()
  - consente di “rispondere” ad una interruzione, mentre si è in attesa di lock()
  - solleva una InterruptedException quando un altro metodo invoca il metodo interrupt

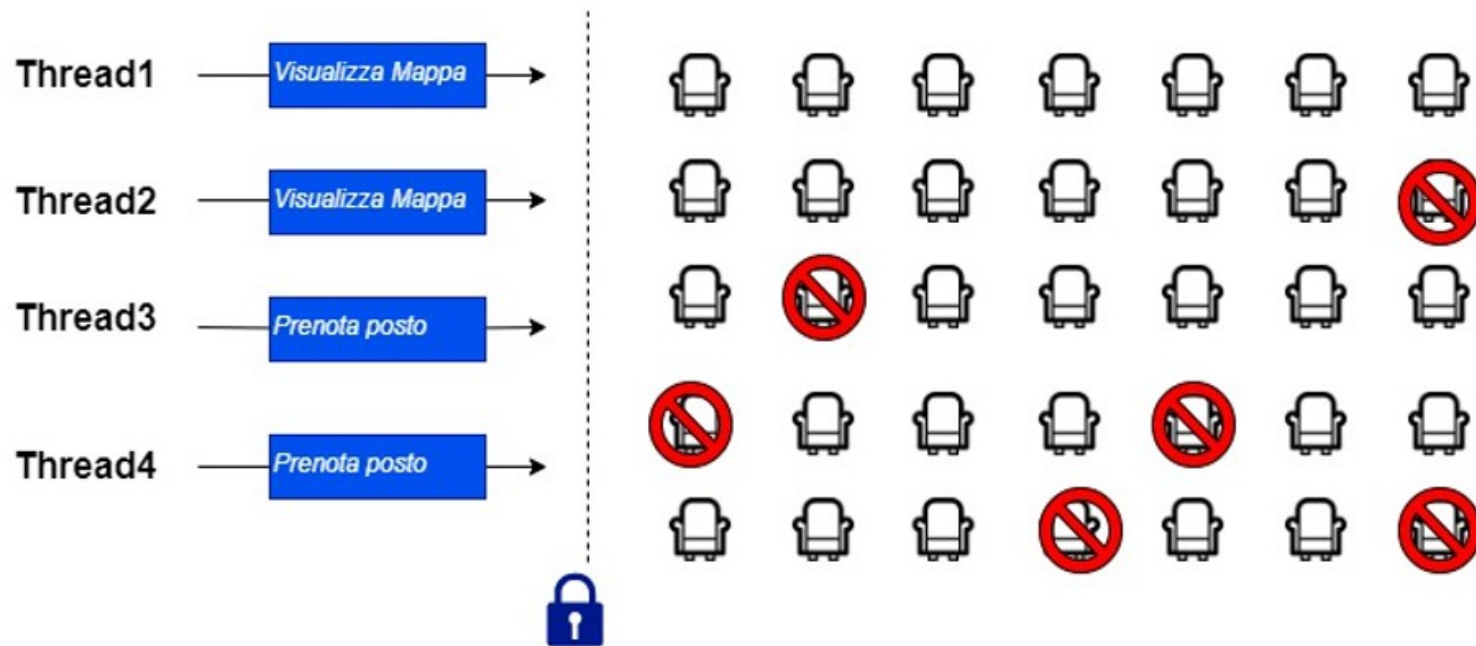
```
private Lock lock= new ReentrantLock();  
public void increment()  
    {try  
        { lock.lockInterruptibly();  
          this.count++;  
        }catch(InterruptedException e) {}  
        finally {lock.unlock();}  
    }
```

# READ/WRITE LOCKS: MOTIVAZIONE



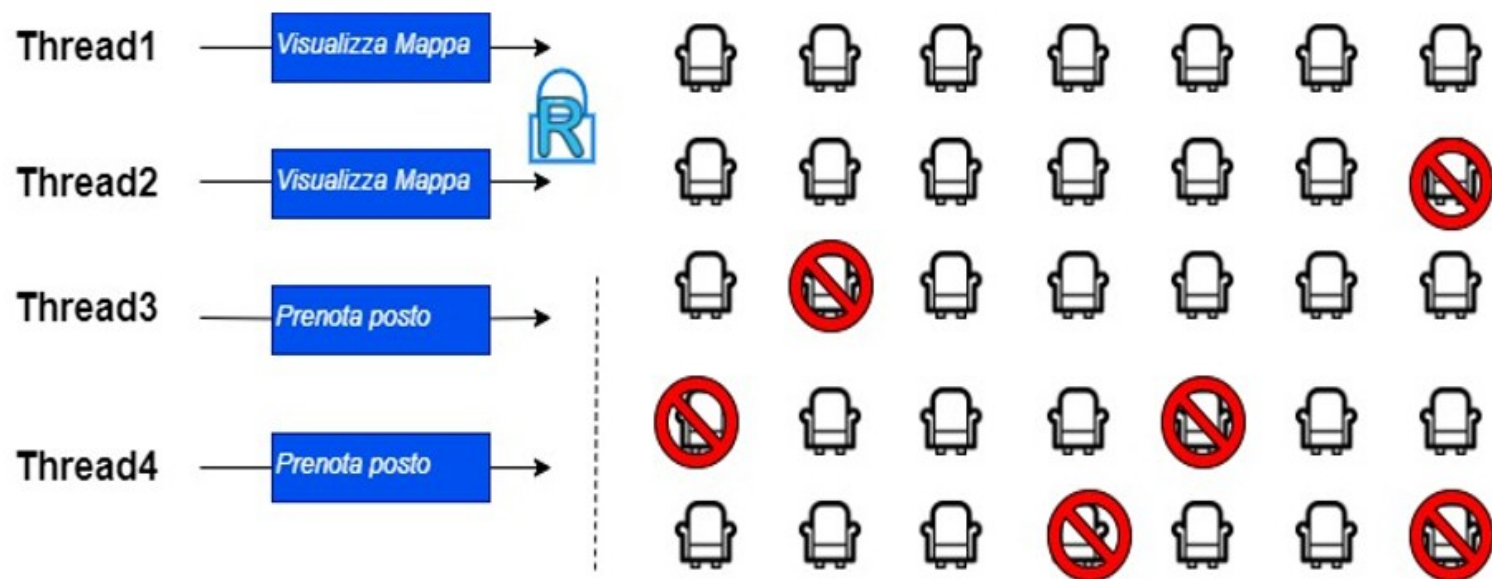
- un insieme di thread tenta di prenotare, simultaneamente, i posti di un teatro
- occorre garantire che due thread non prenotino simultaneamente lo stesso posto: potrebbe risultare in un posto prenotato da due diversi spettatori
- una lock per tutta la struttura garantisce mutua esclusione


# READ/WRITE LOCKS: MOTIVAZIONE



- aggiungiamo una nuova funzionalità: visualizzazione della mappa del teatro
- una sola lock: soluzione inefficiente
  - Thread1 e Thread2 possono visualizzare concorrentemente la mappa, ma con una sola lock questo è impossibile
  - ReentrantLock tratta tutti i thread “alla stessa maniera” occorre una lock più “granulare”

# READ/WRITE LOCKS: MOTIVAZIONE



- idea di base: introdurre una **read lock**  che consenta a molteplici “thread lettori” di procedere alla visualizzazione, ma che escluda l'accesso ai thread che prenotano
- i thread che vogliono aggiornare lo stato si mettono in coda in attesa che non vi siano più thread lettori



# READ/WRITE LOCKS

- interfaccia `ReadWriteLock`
- classe che la implementa: `ReentrantReadWriteLock()`  

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
```
- la classe mantiene due lock separate, una per le operazioni di lettura e una per le scritture.
- la read lock può essere acquisita da più thread lettori, purchè non vi siano scrittori.  

```
Lock readLock = rwLock.readLock();
```
- la write lock è esclusiva  

```
Lock writeLock = rwLock.writeLock();
```

# PRENOTAZIONE POSTI TEATRO

```
import java.util.concurrent.locks.*;

public class Theatre

{private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
  private Lock readLock  = lock.readLock();
  private Lock writeLock = lock.writeLock();
  String [][] theatreMap = new String[4][6];
  public Theatre() {
    for (int i=0; i<4; i++)
      for (int j=0; j<6; j++) {theaterMap[i][j]="";}
  }
  public void readMap() {
    readLock.lock();
    //Stampa la mappa
    readLock.unlock();  }
  public void Book(int x, int y) {
    writeLock.lock();
    theatreMap[x][y]="X";
    writeLock.unlock(); }}
```

# PRENOTAZIONE POSTI TEATRO

```
public class TheatreBooking {  
    public static void main(String [] args)  
    { Theatre th= new Theatre();  
      Thread t3 = new Thread()  
          {public void run() {th.Book(2,3);}}; t3.start();  
      Thread t4 = new Thread()  
          {public void run() {th.Book(3,1);}}; t4.start();  
      Thread t1 = new Thread()  
          {public void run() {th.readMap();}}; t1.start();  
      Thread t2 = new Thread()  
          {public void run() {th.readMap();}}; t2.start();  
    }  
}
```

# READ WRITE LOCK: UTILIZZI



The screenshot shows the 'Area didattica' (Didactic Area) of the University of Pisa. The breadcrumb trail is: Area didattica > Triennale in informatica > Presentazione > Insegnamenti > Informazioni utili. The page title is 'Lista dei corsi' (List of courses). The subtitle is 'Elenco corsi, a.a. 2021/2022'. Below this is a table with 5 columns: Codice, Insegnamento, Docente/i, CFU, and Periodo. The table lists 6 courses.

Codice	Insegnamento	Docente/i	CFU	Periodo
725AA	ARCHITETTURE E SISTEMI OPERATIVI (A)	DANELUTTO MARCO	15	Annuale
725AA	ARCHITETTURE E SISTEMI OPERATIVI (B)	MENCAGLI GABRIELE, TORQUATI MASSIMO	15	Annuale
724AA	ANALISI MATEMATICA (B)	CHIODAROLI ELISABETTA	12	Annuale
724AA	ANALISI MATEMATICA (A)	GRISANTI CARLO ROMANO	12	Annuale
724AA	ANALISI MATEMATICA (C)	TALPO MATTIA	12	Annuale

- gestire accessi concorrenti ad un web site il cui contenuto cambia poco di frequente
- l'elenco dei corsi di informatica è acceduto continuamente, ma cambia una volta all'anno (circa)
- più thread lettori lo possono accedere concorrentemente
- un solo thread lettore lo accede quando si effettua l'aggiornamento annuo

# IL METODO TRY LOCK

- “prova ad acquisire” la lock(), se ci riesce accede alla risorsa, altrimenti non si blocca
  - versione con o senza time-out

```
private ReentrantLock lock = new ReentrantLock();  
private static void accessResource() {  
    boolean lockacquired = lock.tryLock();  
    // in alternativa boolean acquired = lock.tryLock(5, TimeUnit.SECONDS);  
    if (lockacquired) {  
        try {  
            // accedi alla risorsa  
        } finally {  
            lock.unlock();  
        }  
    } else { // fai qualcos'altro }  
}
```

# IL METODO TRY LOCK: UN ESEMPIO

```
import java.util.concurrent.locks.*;

public class TryLockMain

{ public static void main (String args[]) {

    Lock lock = new ReentrantLock();

    new Thread(new ThreadTryLock(lock), "thread_1").start();

    new Thread(new ThreadTryLock(lock), "thread_2").start();

    }

}
```

# IL METODO TRY LOCK: UN ESEMPIO

```
import java.util.concurrent.TimeUnit; import java.util.concurrent.locks.*;

public class ThreadTryLock implements Runnable {
    Lock lock;

    public ThreadTryLock(Lock lock) {
        this.lock = lock; }

    public void run() {
        while (true) {
            try { if (lock.tryLock(1, TimeUnit.SECONDS)) {
                try {
                    System.out.println("The lock is taken by " +
                        Thread.currentThread().getName());
                    TimeUnit.SECONDS.sleep(2);
                } finally {
                    lock.unlock();
                    System.out.println("The lock is released by" +
                        Thread.currentThread().getName());
                }
            } break;
        }
    }
}
```

# IL METODO TRY LOCK: UN ESEMPIO

```
    } else {  
        // non sono riuscito a prendere la lock, riprovo  
        System.out.println("Thread"+Thread.currentThread().getName() +  
                           "unable to acquire the lock");  
    }  
} catch (InterruptedException ignore) {  
}  
}  
}  
}
```



Le lock introducono una performance penalty dovuta a più fattori

- contention
- bookkeeping
- scheduling
- blocking
- unblocking

Performance penalty caratterizza tutti i costrutti a più alto livello introdotti da JAVA, basati su lock (synchronized, monitors, semaphores,...)

# LOCK E PERFORMANCE

- l'uso delle lock introduce overhead, per cui vanno usate con oculatezza
- inserire l'istruzione

```
long time1=System.currentTimeMillis();
```

prima dell'attivazione dei threads

e le istruzioni

```
long time2=System.currentTimeMillis();
```

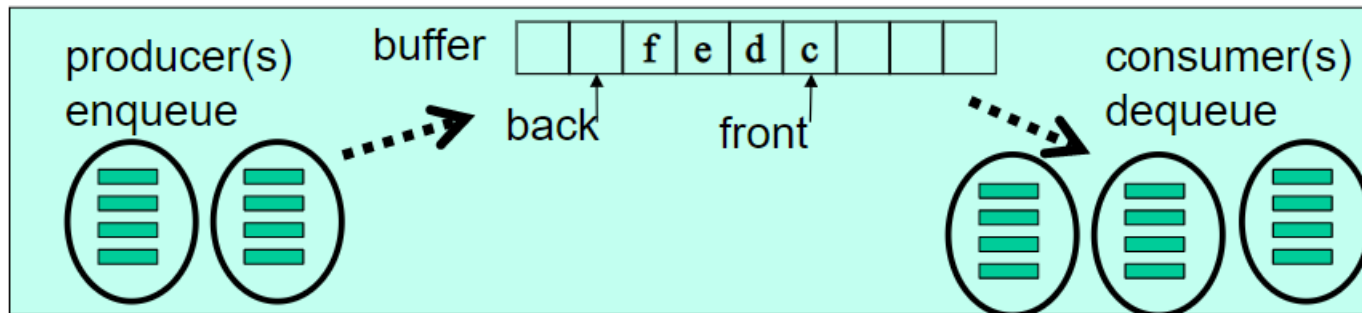
```
System.out.println(time2-time1);
```

```
System.out.println(count);}}
```

alla fine del programma

Il tempo di esecuzione del programma senza uso di lock è circa la metà di quello con uso di lock !

# IL PROBLEMA DEL PRODUTTORE CONSUMATORE



- un classico problema che descrive due (o più thread) che condividono un buffer, di dimensione 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 condivisa
- 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

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

# PRODUTTORE CONSUMATORE: SINCRONIZZAZIONE

```
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()) { }          ATTENZIONE: QUESTA SOLUZIONE
    buffer.add(message);          NON E' CORRETTA!!
    l.unlock();
}

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

- 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 schedatore



# 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(); }

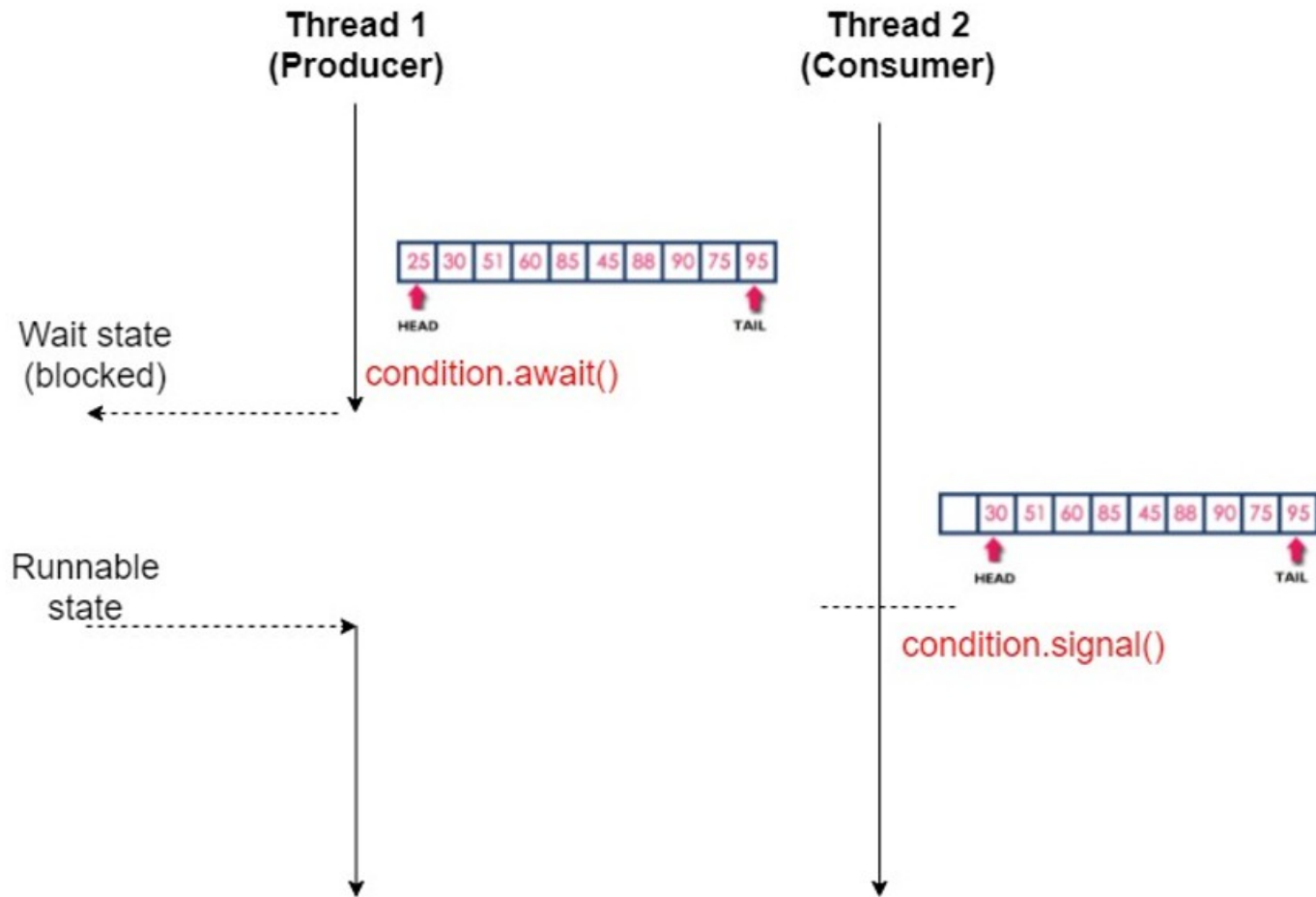
public String get()
{
    l.lock();
    while (isEmpty()) {
        l.unlock();
        Thread.yield();
        l.lock(); }
    String message = buffer.remove(0);
    l.unlock();
    return message; }}
```

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

# SINCRONIZZAZIONE ESPLICITA

- sono necessari meccanismi che consentano
  - di definire un insieme di **condizioni sullo stato** dell'oggetto condiviso
  - 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 (nella lezione successiva)

# VARIABILI DI CONDIZIONE: L'IDEA



# 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(); }
}
```

# VARIABILI DI CONDIZIONE IN SINTESI

- 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

# 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 viene passata ad ogni thread, quando viene invocato il costruttore
- si realizza così la condivisione della risorsa

# 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){}; } } }
```

```
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;}
}
```

```
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();  
}  
}
```

```
public Object consume() throws InterruptedException {
    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"+data);
    return data;}
finally
    {lockcoda.unlock(); }}
```

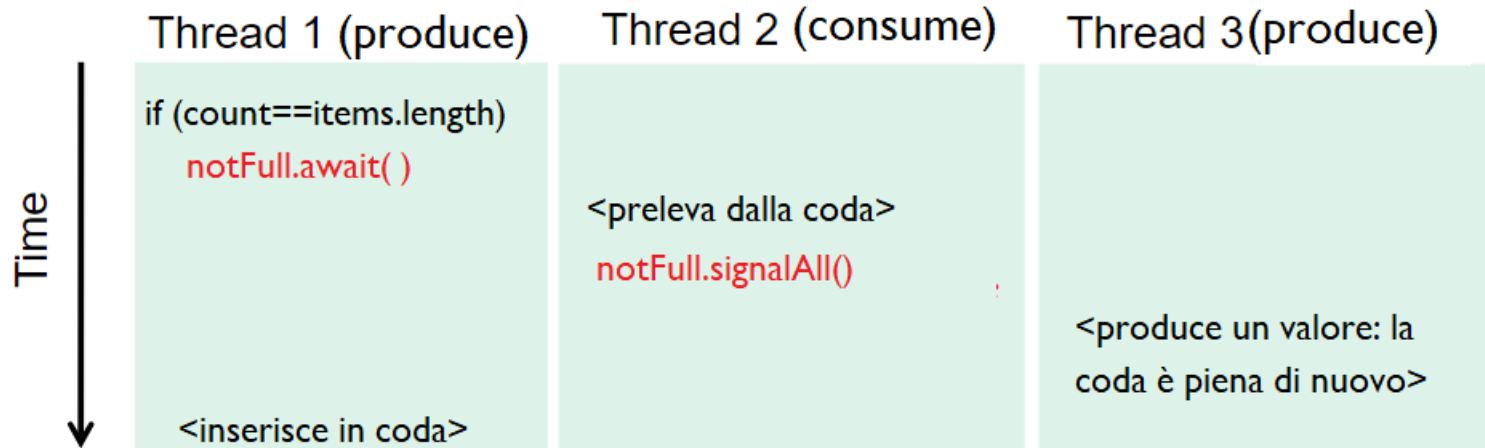
# PRODUTTORE/CONSUMATORE CON CONDIZIONI

```
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

- **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 esclusione solo su piccole porzioni della lista, permettendo ad altri thread l'accesso ad elementi diversi della struttura



# LOCK E DEADLOCK

- Attenzione ai deadlock:
  - **Thread(A)** acquisisce **Lock (X)** e **Thread(B)** acquisisce **Lock(Y)**
  - **Thread(A)** tenta di acquisire **Lock(Y)** e simultaneamente **Thread(B)** tenta di acquisire **Lock(X)**
  - entrambe i threads bloccati all'infinito, in attesa della lock detenuta dall'altro thread!
- **tryLock()**
  - tenta di acquisire la lock() e se essa è già posseduta da un altro thread, il metodo termina immediatamente e restituisce il controllo al chiamante.
  - restituisce un valore booleano, vero se è riuscito ad acquisire la lock(), falso altrimenti
  - può aiutare nella prevenzione del deadlock

# 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 installato 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.