UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

1 Marzo
Lezione 2

Domande

- Non possiamo usare *synchronized* per una classe, dobbiamo sincronizzare ogni metodo

~~`public synchronized class Student`~~

Domande

- Classe `Student` - non ci si può bloccare

```
public double getAverage(){
     synchronized(this.prog_lock){
        synchronized(this.math_lock){
           return (this.math_grade+this.prog_grade)/2;
        }
     }
  }
```

Domande

- Un thread NEW non può essere interrotto (non prima di richiamare `start()`)

- Compito laboratorio 2 :
  - il thread che ha trovato la stringa deve aspettare che tutti gli altri sono avviati prima di interromperli

    o

  - si deve usare un flag che diventa true quando la parola è stata trovata

Domande

- Oggetti di tipo String, Boolean, Integer, etc possono essere usati come lock pero non per controllare accessi a loro stessi (perché sono immutabili)

```
public double setName(String name){
    synchronized(this.name){
        this.name= name;
    }
}
```

```
public double setName(String name){
    synchronized(this.name_lock){
        this.name= name;
    }
}
```

Concorrenza avanzata

- Da Java 5 e stato introdotto il package `java.util.concurrent`
- Classi `Lock`, variabili di condizione dedicate
- Semafori, barriere

Oggi

- *Concurrent Collections*
- Variabili *Atomic*

Prossima settimana

- Esecuzione dei *thread* controllata e indipendente dalla logica dell'applicazione
- Possibilità di restituire un risultato per un *task* e lanciare eccezioni

Interfaccia `Lock`

- Introdotta in Java 5 - alternativa per `synchronized(o)`
- Implementata da

`ReentrantLock`

`ReentrantReadWriteLock.ReadLock`

`ReentrantReadWriteLock.WriteLock`

- Simile al *mutex* implicito - più flessibile
- Vantaggio principale - possibilità di cambiare idea quando si richiede un `Lock` - *non blocking lock*

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
```

Interfaccia Lock

```
void lock();
void unlock();
```

- Lo stesso effetto del blocco `synchronized :` il *thread* si blocca fino che il *lock* diventa disponibile, non può essere interrotto in nessun modo, i valori delle variabili modificate diventano visibili a tutti i *thread* dopo il *unlock*.

```
void lockInterruptibly() throws
InterruptedException;
```

- Il *thread* si blocca però può essere interrotto

Interfaccia Lock

```
boolean tryLock();
```

- Prova se il *lock* e disponibile
- Sì : acquisisce il *lock* e restituisce `true`
- No: restituisce `false`
- Non si blocca

```
boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
```

- Se il *lock* è disponibile lo acquisisce e restituisce `true.`
- Se il *lock* non è disponibile si blocca per un tempo definito da `timeout.`
- Se il *lock* non è ancora disponibile dopo il *timeout*, restituisce `false.`

Interfaccia `Lock`

- I `Lock` hanno associato delle `Condition` con meccanismo simile al *wait/notify*
- Vantaggio principale - possibilità di usare più `Condition` sullo stesso `Lock`
- Code di attesa (*wait sets*) specializzate
- Più flessibilità per svegliare i *thread*
- Una `Condition` nuova è associata usando il metodo

```
Condition newCondition();
```

Condition

```
void await();
boolean await(long time, TimeUnit unit);
long awaitNanos(long nanosTimeout);
void awaitUninterruptibly();
boolean awaitUntil(Date deadline);
```

- Metodi simili a `wait`
- Possibilità di mettere un *deadline* in nanosecondi e di controllare quando/come il thread è stato svegliato
- Possibilità di *wait* senza essere interrotto

Condition

```
void signal();
void signalAll();
```

- Metodi simili a `notify` e `notifyAll()`

ReentrantLock

- Metodi addizionali:

```
ReentrantLock()
ReentrantLock(boolean fair)

int getHoldCount()
int getQueueLength()
int getWaitQueueLength(Condition c)
boolean hasQueuedThread(Thread t)
boolean hasQueuedThreads()
boolean hasWaiters(Condition c)
boolean isHeldByCurrentThread()
boolean isLocked()

protected Thread getOwner()
protected Collection<Thread> getQueuedThreads()
protected Collection<Thread> getWaitingThreads()
```

Meno veloce

Approssimativi

Condition

- E.g. Produttore - consumatore con più produttori è consumatori
- Quando un produttore produce i suoi dati, vuole svegliare solo i consumatori.
- Vice versa, quando un consumatore consuma, vuole svegliare solo i produttori.
- Non si può fare con un *monitor* sullo stesso oggetto
- Usare due oggetti vuol dire sincronizzare tutti e due quando si cambia il valore prodotto/consumato
- Si può fare con due `Condition` sullo stesso `Lock`
- Mamma/bambino => tante nanny/ tanti bambini

```
public class Food {

    private ArrayList<String> food;
    private ReentrantLock lock;
    private Condition foodFull, foodEmpty;

    public Food(){
        this.food=new ArrayList<String>();
        this.lock=new ReentrantLock(true);
        this.foodFull= this.lock.newCondition();
        this.foodEmpty= this.lock.newCondition();
    }

public ReentrantLock getLock() {
    return lock;
}

public Condition getFoodEmpty() {
    return foodEmpty;
}

public Condition getFoodFull() {
    return foodFull;
}
```

```
public  boolean isEmpty() throws InterruptedException {
    this.lock.lockInterruptibly();
    boolean result=(food.size()==0);
    this.lock.unlock();
    return result;
}

public  boolean isFull() throws InterruptedException {
    this.lock.lockInterruptibly();
    boolean result=(food.size()==2);
    this.lock.unlock();
    return result;
}
```

```
public  void cook(String food) throws InterruptedException {
    this.lock.lockInterruptibly();
    this.food.add(food);
    this.lock.unlock();
}

public  void eat() throws InterruptedException{
    this.lock.lockInterruptibly();
    this.food.remove(this.food.size()-1);
    this.lock.unlock();
}
```

```
public class Child implements Runnable {

   Food food;

   public Child(Food food){
      this.food=food;
   }

   private void print(String message){
      System.out.println(System.currentTimeMillis()/1000+ "- Child"
+Thread.currentThread().getId()+": "+message);
   }
```

```
public void run() {
      try{
         this.print("Just born");
         while(true){
            this.print("I am hungry, I want to eat.");
            this.food.getLock().lockInterruptibly();
            try{
               while (this.food.isEmpty()){
                  this.print("There is no food :(");
                  food.getFoodFull().await();
               }
               this.print("Yay, eating now :)");
               food.eat();
               food.getFoodEmpty().signal();
            } finally {
               this.food.getLock().unlock();
            }
            this.print("Going back to sleep");
            Thread.sleep(1000);
      }
   }catch (InterruptedException e){
      this.print("Interrupted");
      return;
   }}}
```

```
public class Nanny implements Runnable{
    Food food;
    public Nanny(Food food){this.food=food;}
    private void print(String message){
        System.out.println(System.currentTimeMillis()/1000+ " - Nanny “
+Thread.currentThread().getId()+": "+message);}
```

```
public void run() {
   try{
      this.print("Created");
      while(true){
         this.food.getLock().lockInterruptibly();
         try{
            while(this.food.isFull()){
               this.print("Waiting for one baby to eat");
               this.food.getFoodEmpty().await();
            }
            this.print("Plate empty, cooking");
            this.food.cook("Soup");
            this.food.getFoodFull().signal();
         } finally{
            this.food.getLock().unlock();
         }
         this.print("Now resting a bit");
         Thread.sleep(1000);
      }
   }catch (InterruptedException e){
      this.print("Interrupted");
      return;
   }
   }
}
```

```
public static void main(String[] args) {
   try{
      Food food= new Food();
      ArrayList<Thread> threads= new ArrayList<Thread>();
      for (int i=0 ;i<3;i++){
         threads.add(new Thread(new Nanny(food)));
      }
      for (int i=0 ;i<4;i++){
         threads.add(new Thread(new Child(food)));
      }
      for (Thread t : threads)
         t.start();
      Thread.sleep(20000);
      for (Thread t : threads)
         t.interrupt();
   } catch(InterruptedException e){}

}
```

Semafori

- Supponiamo di avere un set di N risorse che i *thread* possono usare
- Se tutte le risorse sono già in uso, i nuovi *thread* devono aspettare che si liberino
- Funziona con un contattore: all'inizio è N, viene decrementato ogni volta che un nuovo *thread* arriva, incrementato quando un *thread* finisce. Se 0, i *thread* aspettano.

Semaphore

- Può essere implementato con una `Condition`
- In Java esiste la classe `Semaphore`
- Costruttori:

```
Semaphore(int permits)

Semaphore(int permits,boolean fair)
```

Semaphore

- Metodi:

```
void aquire()
void aquire(int permits)
void aquireUninterruptibly()
void aquireUninterruptibly(int permits)

boolean tryAquire()
boolean tryAquire(int permits)
boolean tryAquire(long timeout, TimeUnit unit)
boolean tryAquire(int permits, long timeout,
TimeUnit unit)
```

Semaphore

- Metodi:

Non verificano se owner.

```
void release()
void release(int permits)

int availablePermits()
int getQueueLength()
boolean hasQueuedThreads()

int drainPermits()

protected void reducePermits(int amount)
protected Collection<Thread> getQueuedThreads()
```

approssimativi

Esempio

- Biglietteria
- Ci sono 5 sportelli aperti
- I viaggiatori vengono a comprare biglietti
- Se nessun sportello è libero aspettano in linea

```java
import java.util.concurrent.Semaphore;

public class TrainTicketMain {

   public static void main(String[] args) {
      try{
         Semaphore ticketCounters= new Semaphore(5,true);

         for (int i=0;i<20;i++){
            (new Thread(new Traveler(ticketCounters))).start();
            Thread.sleep(10);
         }
      } catch (InterruptedException e){}

   }

}
```

```
import java.util.Random;
import java.util.concurrent.Semaphore;

public class Traveler implements Runnable{

   private Semaphore ticketCounters;

   public Traveler(Semaphore ticketCounters){

      this.ticketCounters=ticketCounters;
   }

   private void print(String message){
      System.out.println(System.currentTimeMillis()+" Traveler
"+Thread.currentThread().getId()+": "+message);
   }

   //run method goes here

}
```

```
@Override
public void run() {
   Random rand = new Random(System.currentTimeMillis());
   this.print("I am a new traveler");
   try{
      this.ticketCounters.acquire();
      this.print("I am buying my ticket");
      try{
         Thread.sleep(rand.nextInt(1000));
         this.print("Got my ticket");
      }finally{
         this.ticketCounters.release();
      }
      this.print("Done, getting on board");
   }catch (InterruptedException e){
      this.print("Something went wrong");
   }
}
```

```
1456407548224 Traveler 9: I am a new traveler
1456407548224 Traveler 9: I am buying my ticket
1456407548234 Traveler 10: I am a new traveler
1456407548234 Traveler 10: I am buying my ticket
1456407548247 Traveler 11: I am a new traveler
1456407548247 Traveler 11: I am buying my ticket
1456407548258 Traveler 12: I am a new traveler
1456407548258 Traveler 12: I am buying my ticket
1456407548269 Traveler 13: I am a new traveler
1456407548269 Traveler 13: I am buying my ticket
1456407548282 Traveler 14: I am a new traveler
1456407548293 Traveler 15: I am a new traveler
1456407548303 Traveler 16: I am a new traveler
1456407548316 Traveler 17: I am a new traveler
1456407548327 Traveler 18: I am a new traveler
1456407548337 Traveler 19: I am a new traveler
1456407548350 Traveler 20: I am a new traveler
1456407548363 Traveler 21: I am a new traveler
1456407548363 Traveler 10: Got my ticket
1456407548364 Traveler 10: Done, getting on board
1456407548364 Traveler 14: I am buying my ticket
1456407548376 Traveler 22: I am a new traveler
1456407548386 Traveler 23: I am a new traveler
1456407548398 Traveler 24: I am a new traveler
1456407548402 Traveler 12: Got my ticket
1456407548402 Traveler 12: Done, getting on boar
```

```
1456407548402 Traveler 15: I am buying my ticket
1456407548410 Traveler 25: I am a new traveler
1456407548420 Traveler 26: I am a new traveler
1456407548433 Traveler 27: I am a new traveler
1456407548444 Traveler 28: I am a new traveler
1456407548457 Traveler 9: Got my ticket
1456407548457 Traveler 9: Done, getting on board
1456407548457 Traveler 16: I am buying my ticket
1456407548493 Traveler 11: Got my ticket
1456407548493 Traveler 11: Done, getting on board
1456407548493 Traveler 17: I am buying my ticket
1456407548502 Traveler 14: Got my ticket
1456407548502 Traveler 14: Done, getting on board
…….
1456407549108 Traveler 22: Done, getting on board
1456407549108 Traveler 23: I am buying my ticket
1456407549184 Traveler 23: Got my ticket
1456407549185 Traveler 23: Done, getting on board
1456407549185 Traveler 24: I am buying my ticket
1456407549751 Traveler 28: Done, getting on board
1456407549765 Traveler 24: Got my ticket
1456407549766 Traveler 24: Done, getting on board
```

Problema *Readers-writers*

- Abbiamo un dato che può essere letto o scritto da vari *thread*.
- Ci sonno tanti scrittori e tanti lettori
- Più di un lettore possono leggere in contemporanea
- Nessuno può scrivere o leggere se c’è già un scrittore chi scrive

Problema Readers-writers

- Soluzione : *2 lock* diversi, uno per scrivere (*mutex*), uno per leggere (non esclusivo per i lettori)

- Cosa facciamo se il lock è aperto e ci sono un scrittore ed un lettore ad aspettare?

- Cosa facciamo se il *read lock* e chiuso, c'e un *writer* chi aspetta, però arriva un altro *reader*?

  - Priorità al scrittore - funziona bene quando i scrittori sono in pochi

  - Equità - ordine di arrivo

Interfaccia `ReadWriteLock`

- 2 metodi usati ad accedere al *read* e *write lock*

```
Lock readLock()
Lock writeLock()
```

- Implementata da `ReentrantReadWriteLock`
  - Equo : nessuna priorità a *reader* o *writer*
  - *Downgrading*: se un *thread* ha il *write lock*, può prendere il *read lock* e lasciare il *write*

ReentrantReadWriteLock

`readLock().lock()`

Se *write lock* e disponibile, prende *read lock*

Se *write lock* e preso, aspetta

`writeLock().lock()`

Prende *write lock* solo se entrambi (*read* e *write*) *lock* sono disponibili

Esempio: counter di prima

```
public class Counter {
    private int count;
    private ReentrantReadWriteLock lock;

    public Counter(){
        this.count=0;
        this.lock=new ReentrantReadWriteLock();
    }
    public void increment(){
        this.lock.writeLock().lock();
        this.count++;
        try {
            Thread.sleep(100);// simulate longer write
        } catch (InterruptedException e) {}
        this.lock.writeLock().unlock();
    }
    public int getCount() {
        this.lock.readLock().lock();
        int count=this.count;
        try {
            Thread.sleep(200);// simulate longer read
        } catch (InterruptedException e) {}
        this.lock.readLock().unlock();
        return count;
}}
```

```
public class Reader implements Runnable{

   Counter c;

   public Reader(Counter c){
      this.c=c;
   }
   public void print(String message){
      System.out.format("%n%d Reader %d:%s",System.currentTimeMillis(),
            Thread.currentThread().getId(),message);
   }
   @Override
   public void run() {
         this.print("created");
         this.print("reading c="+this.c.getCount());
         this.print("finished reading");
   }

}
```

```java
public class Writer implements Runnable{
   Counter c;

   public Writer(Counter c){
      this.c=c;
   }
   public void print(String message){
      System.out.format("%n%d Writer %d:%s",System.currentTimeMillis()
            Thread.currentThread().getId(),message);
   }
   @Override
   public void run() {
         this.print("created");
         this.c.increment();
         this.print("finished incrementing c");
   }

}
```

```
import java.util.Random;

public class ReaderWriterMain {

   public static void main(String[] args) {
      Counter c= new Counter();
      Random rand = new Random(System.currentTimeMillis());
      double writerProb=0.2;

      for (int i=0;i<20;i++){
         if (rand.nextDouble()<writerProb){
            (new Thread(new Writer(c))).start();
         }
         else{
            (new Thread(new Reader(c))).start();
         }
      }
   }
}
```

Le barriere

- Punto di sincronizzazione per un gruppo di *thread*
- Nessun *thread* può continuare prima che tutti hanno raggiunto la barriera
- Classe `CyclicBarrier` - riutilizzabile

CyclicBarrier

```
CyclicBarrier(int parties)
CyclicBarrier(int parties, Runnable
barrierAction)
```

- parties - Numero di *thread* da aspettare
- barrierAction - *Task* da eseguire quando tutti i *thread* hanno raggiunto la barriera (nel ultimo *thread*)

CyclicBarrier

```
int await()
int await(long timeout, TimeUnit unit)

int getNumberWaiting()
int getParties()

void reset()
boolean isBroken() -
```
se si fa *reset* quando ci sono *thread* ad aspettare

Liveness

- Problemi di velocità del programma *multithreaded*
  - *Deadlock*
  - *Livelock*
  - *Starvation*

Deadlock

- esempio più semplice: due lock interscambiati

```
ReentrantLock lock1, lock2;
```

t=1
```
lock1.lock();
lock2.lock();
doSomething();
lock2.lock();
lock1.lock();
```

t=2
```
lock2.lock();
lock1.lock();
doSomethingElse();
lock1.lock();
lock2.lock();
```

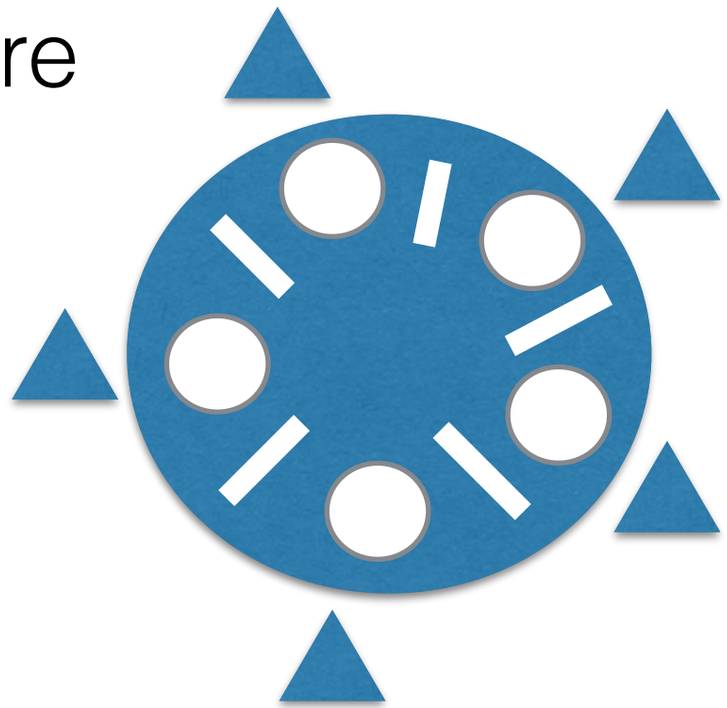
- programma si ferma

Deadlock

- Soluzioni
- *Reentrant lock* - non lascia il *thread* entrare in *deadlock* con se stesso
- sempre fare *lock* nella stessa successione - gerarchia dei *lock*
- evitare di richiamare metodi di un altro oggetto quando il *lock* associato con un oggetto e acquisito
- Evitare di fare `sleep()` quando si tiene un *lock*
- Esempio avanzato: *Dining philosophers*

Dining philosophers

- 5 filosofi sono seduti in un tavolo rotondo
- Hanno di fronte un piatto di riso ogni uno più 5 *chopstick* (uno a destra e uno a sinistra di ogni piato)
- I filosofi sanno fare 2 cose:
  - meditare
  - mangiare - hanno bisogno di due *chopstick* - li devono condividere



Dining philosophers

- Strategia più semplice:
    - Quando il filosofo finisce di meditare
        - Prende il *chopstick* a sinistra
        - Prende il *chopstick* a destra
        - Mangia
        - Lascia il *chopstick* a destra
        - Lascia il *chopstick* a sinistra

Dining philosophers

- Modello con *thread*
- Ogni filosofo e un *thread*
- Meditare con `sleep()`
- Mangiare con `sleep()`
- Ogni *chopstick* è un `Lock`

```
public class Philosopher implements Runnable{

   private Lock leftChopstick, rightChopstick;

   public Philosopher(Lock left, Lock right){
      this.leftChopstick=left;
      this.rightChopstick=right;
   }

   public void print(String message){
      System.out.println(System.currentTimeMillis()+"-Philosopher
"+Thread.currentThread().getId()+": "+message);
   }

   private void meditate() throws InterruptedException{
      this.print("Starting to meditate");
      Thread.sleep(5000);
      this.print("Finished meditating");
   }
```

```
private void eat() throws InterruptedException{
   this.print("Picking up chopsticks");
   //first pick up the chopsticks, first left then right
   this.leftChopstick.lockInterruptibly();
   try{
      //Thread.sleep(50);
      this.rightChopstick.lockInterruptibly();
      try{
         this.print("Starting to eat");
         Thread.sleep(5000);
         this.print("Finished eating");
      } finally {
         //release the chopsticks
         this.rightChopstick.unlock();
      }
   } finally {
      this.leftChopstick.unlock();
   }
   this.print("Left chopsticks");
}
```

```
   @Override
   public void run() {
      try{
         while (true){
            //alternate between meditate and eat until
interrupted
            this.eat();
            this.meditate();
         }
      } catch (InterruptedException e){
         this.print("Interrupted, finishing"); return;}

   }
//chiude la classe Philosopher
}
```

```
public class PhilosophersMain {

   public static void main(String[] args) {
      int N=5;

      Lock[] chopsticks = new ReentrantLock[N];
      Philosopher[] phils= new Philosopher[N];
      Thread[] threads = new Thread[N];
      try{
         //first create the chopsticks
         for (int i=0;i<N;i++)
            chopsticks[i]= new ReentrantLock();
         //then create the philosophers with 2 chopsticks each
      for (int i=0;i<N;i++){
         phils[i]= new Philosopher(chopsticks[i], chopsticks[(i+1)%N]);
         threads[i]= new Thread(phils[i]);
      }
      for (Thread t: threads){
         t.start();
      }
      //After a while stop all philosophers
      Thread.sleep(2000);
      for (Thread t: threads){
         t.interrupt();
      }
   }catch (InterruptedException e){}}}
```

```
1456227802466-Philosopher 9: Picking up chopsticks
1456227802467-Philosopher 13: Picking up chopsticks
1456227802466-Philosopher 10: Picking up chopsticks
1456227802467-Philosopher 11: Picking up chopsticks
1456227802467-Philosopher 11: Starting to eat
1456227802466-Philosopher 12: Picking up chopsticks
1456227802467-Philosopher 9: Starting to eat
1456227807472-Philosopher 9: Finished eating
1456227807472-Philosopher 11: Finished eating
1456227807473-Philosopher 11: Left chopsticks
1456227807473-Philosopher 11: Starting to meditate
1456227807473-Philosopher 9: Left chopsticks
1456227807473-Philosopher 9: Starting to meditate
1456227812478-Philosopher 11: Finished meditating
1456227812478-Philosopher 9: Finished meditating
1456227812478-Philosopher 9: Picking up chopsticks
1456227812479-Philosopher 9: Starting to eat
1456227812478-Philosopher 11: Picking up chopsticks
1456227812479-Philosopher 11: Starting to eat
1456227817483-Philosopher 9: Finished eating
1456227817483-Philosopher 11: Finished eating
1456227817483-Philosopher 9: Left chopsticks
1456227817483-Philosopher 11: Left chopsticks
1456227817483-Philosopher 9: Starting to meditate
1456227817483-Philosopher 11: Starting to meditate
1456227822472-Philosopher 9: Interrupted, finishing
1456227822472-Philosopher 13: Interrupted, finishing
1456227822472-Philosopher 11: Interrupted, finishing
1456227822472-Philosopher 10: Interrupted, finishing
1456227822472-Philosopher 12: Interrupted, finishing
```

```
1456227664626-Philosopher 12: Picking up chopsticks
1456227664627-Philosopher 13: Picking up chopsticks
1456227664626-Philosopher 9: Picking up chopsticks
1456227664626-Philosopher 10: Picking up chopsticks
1456227664627-P                Picking up chopsticks
1456227684629-Philosopher 12: Interrupted, finishing
1456227684629-Philosopher 10: Interrupted, finishing
1456227684629-Philosopher 13: Interrupted, finishing
1456227684629-Philosopher 9: Interrupted, finishing
1456227684629-Philosopher 11: Interrupted, finishing
```

DEADLOCK

Dining philosophers

- Evitare il *deadlock*:
  - Usa `tryLock()`
  - Se il primo `lock()` funziona però il secondo no, libera anche il primo *lock*
  - Questa soluzione funziona solo con i `Lock`, non con `synchronized`

```
private void eat() throws InterruptedException{
        boolean ate=false;
        while (!ate){
            //first pick up the chopsticks, first left then right
            if(this.leftChopstick.tryLock()){
                try{
                    this.print("Picked up left chopstick");
                    if (this.rightChopstick.tryLock()){
                        try{
                            this.print("Picked up right chopstick");
                            ate=true;
                            this.print("Eating");
                            Thread.sleep(5000);
                        } finally{
                            this.rightChopstick.unlock();
                            this.print("Dropped right chopstick");
                        }
                    }
                    else{
                        this.print("Right chopstick not available.");}
                } finally{
                    this.leftChopstick.unlock();
                    this.print("Dropped left chopstick");
                }
            }
            else{
                this.print("Left chopstick not available");}
            Thread.sleep(100);
}}
```

```
1456243531451-Philosopher 12: Picked up left chopstick
1456243531451-Philosopher 12: Picked up right chopstick
1456243531451-Philosopher 10: Picked up left chopstick
1456243531451-Philosopher 9: Picked up left chopstick
1456243531451-Philosopher 11: Picked up left chopstick
1456243531452-Philosopher 9: Right chopstick not available.
1456243531452-Philosopher 10: Right chopstick not available.
1456243531451-Philosopher 12: Eating
1456243531451-Philosopher 13: Left chopstick not available
1456243531452-Philosopher 10: Dropped left chopstick
1456243531452-Philosopher 9: Dropped left chopstick
1456243531452-Philosopher 11: Right chopstick not available.
1456243531452-Philosopher 11: Dropped left chopstick
1456243531556-Philosopher 13: Left chopstick not available
1456243531557-Philosopher 9: Picked up left chopstick
1456243531556-Philosopher 10: Picked up left chopstick
1456243531557-Philosopher 10: Right chopstick not available.
1456243531557-Philosopher 10: Dropped left chopstick
1456243531557-Philosopher 9: Right chopstick not available.
1456243531557-Philosopher 9: Dropped left chopstick
1456243531557-Philosopher 11: Picked up left chopstick
1456243531557-Philosopher 11: Right chopstick not available.
1456243531557-Philosopher 11: Dropped left chopstick
```

```
1456243531658-Philosopher 9: Picked up left chopstick
1456243531658-Philosopher 13: Left chopstick not available
1456243531658-Philosopher 10: Picked up left chopstick
1456243531658-Philosopher 11: Picked up left chopstick
1456243531658-Philosopher 10: Right chopstick not available.
1456243531658-Philosopher 9: Right chopstick not available.
1456243531658-Philo                          left chopstick
1456243531658-Philo                          hopstick not available.
1456243531658-Philosopher 9: Dropped left chopstick
1456243531659-Philosopher 11: Dropped left chopstick
1456243531763-Philosopher 9: Picked up left chopstick
1456243531763-Philosopher 10: Picked up left chopstick
1456243531763-Philosopher 11: Picked up left chopstick
1456243531763-Philosopher 13: Left chopstick not available
1456243531764-Philosopher 11: Right chopstick not available.
1456243531763-Philosopher 10: Right chopstick not available.
1456243531764-Philosopher 10: Dropped left chopstick
1456243531763-Philosopher 9: Right chopstick not available.
1456243531764-Philosopher 11: Dropped left chopstick
1456243531764-Philosopher 9: Dropped left chopstick
1456243531866-Philosopher 13: Left chopstick not available
```

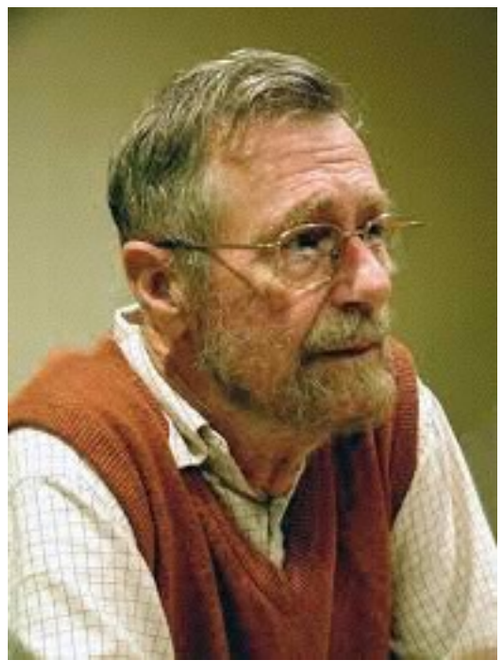
LIVELOCK

Livelock

- Simile al *deadlock* - il programma non va avanti
- *Thread* non sono sospesi, ma ripetono le stesse azioni alternate all'infinito
- I filosofi prendono è lasciano i *chopstick* in continuazione senza arrivare mai a mangiare:
  - Primo filosofo prende il *chopstick* a sinistra
  - Secondo, terzo, etc prendono il *chopstick* a sinistra
  - Primo prova di prendere *chopstick* a destra - non funziona
  - secondo, terzo, etc lo stesso
  - Primo lascia il *chopstick* a sinistra
  - secondo, terzo etc lo stesso
  - Primo cominciano da capo

Dining philosophers

- Evitare il *livelock*:
    - Definiamo gerarchia dei *chopstick* - una priorità
    - Ogni filosofo prende prima il *chopstick* con priorità più alta
    - non si blocca mai
    - Soluzione proposta da Dijkstra



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

public class Chopstick extends ReentrantLock{
   int priority;

   public Chopstick(int priority){
      super();
      this.priority=priority;
   }

   public int getPriority() {
      return priority;
   }
}
```

```
public class SmartPhilosopher implements Runnable{

   private Chopstick firstChopstick, secondChopstick;

   public SmartPhilosopher(Chopstick c1, Chopstick c2){
      if (c1.getPriority()>=c2.getPriority()){
         this.firstChopstick=c1;
         this.secondChopstick=c2;
      }
      else{
         this.firstChopstick=c2;
         this.secondChopstick=c1;
      }
   }

   //meditate, run and print don't change from Philosopher
```

```
private void eat() throws InterruptedException{
   //first pick up first chopstick, then second
   this.firstChopstick.lockInterruptibly();
   try{
      this.print("Got one chopstick");
      this.secondChopstick.lockInterruptibly();
      try{
         this.print("Got second chopstick");
         this.print("Starting to eat");
         Thread.sleep(5000);
         this.print("Finished eating");
      } finally{
      //release the chopsticks
         this.secondChopstick.unlock();
         this.print("Releasing second chopstick");
      }
   } finally {
      this.firstChopstick.unlock();
      this.print("Releasing first chopstick");
   }

}
```

```
public class SmartPhilosophersMain {

   public static void main(String[] args) {
      int N=5;
      Chopstick[] chopsticks = new Chopstick[N];
      SmartPhilosopher[] phils= new SmartPhilosopher[N];
      Thread[] threads = new Thread[N];
      try{
         //first create the chopsticks
         for (int i=0;i<N;i++)
            chopsticks[i]= new Chopstick(i);
         //then create the philosophers with 2 chopsticks each
         for (int i=0;i<N;i++){
            phils[i]= new SmartPhilosopher(chopsticks[i], chopsticks[(i+1)%N]);
            threads[i]= new Thread(phils[i]);
         }
         for (Thread t: threads){
            t.start();
         }
         //After a while stop all philosophers
         Thread.sleep(40000);
         for (Thread t: threads){
            t.interrupt();
         }
      }catch (InterruptedException e){}
}}
```

```
1456245017193-Philosopher 11: Got one chopstick
1456245017194-Philosopher 12: Got one chopstick
1456245017193-Philosopher 9: Got one chopstick
1456245017194-Philosopher 9: Got second chopstick
1456245017194-Philosopher 9: Starting to eat
1456245017193-Philosopher 10: Got one chopstick
1456245022195-Philosopher 9: Finished eating
1456245022195-Philosopher 9: Releasing second chopstick
1456245022196-Philosopher 9: Releasing first chopstick
1456245022196-Philosopher 10: Got second chopstick
1456245022196-Philosopher 9: Starting to meditate
1456245022196-Philosopher 10: Starting to eat
1456245027199-Philosopher 9: Finished meditating
1456245027199-Philosopher 10: Finished eating
1456245027200-Philosopher 10: Releasing second chopstick
1456245027200-Philosopher 9: Got one chopstick
1456245027200-Philosopher 11: Got second chopstick
1456245027200-Philosopher 10: Releasing first chopstick
1456245027200-Philosopher 11: Starting to eat
1456245027200-Philosopher 9: Got second chopstick
1456245027200-Philosopher 10: Starting to meditate
1456245027200-Philosopher 9: Starting to eat
1456245032200-Philosopher 11: Finished eating
1456245032201-Philosopher 11: Releasing second chopstick
1456245032201-Philosopher 9: Finished eating
1456245032201-Philosopher 12: Got second chopstick
```

Starvation

- Uno dei *thread* non ha accesso alle risorse condivise per molto tempo
- Esempio: problema *Reader/Writer* con precedenza agli *writer*
  - se all'improvviso arrivano molti *writer*, i *reader* non riescono mai a leggere

Esercizi

- 1. Prendere la parola con `Lock` e `Condition`
- 2. Simulazione rotonda (4 entrate)
  - Modellare con i thread delle auto che percorrono una rotonda.
  - Per entrare, le auto devono essere sicure di non avere nessuna machina a sinistra (nel quarto di rotonda precedente alla loro entrata).
  - Una volta entrate, le auto percorrono un quarto di rotonda ogni secondo, fin che raggiungono la loro uscita. Supponiamo che ci possono essere un numero illimitato di auto nella rotonda, in qualsiasi quarto.
  - Per ogni auto, l'entrata e l'uscita della rotonda sono generate aleatorio all'inizio (costruttore). Le auto sono generate dalla main() con intervallo da 0 a 2 secondi tra loro.
  - Tip: servono 4 Lock - attenzione: ogni auto aspetta il lock di sinistra però chiude il lock di destra. Ogni quarto di rotonda l'auto cambia lock.