UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

27 Settembre
Lezione 2

Shared memory

- *Shared memory* molto efficiente per scambiare informazioni
- Suscettibile agli errori
  - interferenza dei thread
    - errori di calcolo quando thread multipli usano la stessa variabile shared

Interferenza (*race condition*)

- E.g. Applicazione bancaria
  - Thread 1 : legge il saldo del conto corrente e aggiunge gli interessi (1%)
  - Thread 2: legge il saldo e aggiunge lo stipendio
  - Sfortuna:
    - Thread 1 legge il saldo ($100)
    - Thread 2 legge il saldo ($100)
    - Thread 2 aggiunge stipendio ($1100)
    - Thread 1 aggiunge interessi ($101)
    - !!!!!!!!!

Shared memory

- Suscettibile agli errori
  - memoria incoerente (*memory consistency errors*)
    - variabili non volatili - il nuovo valore potrebbe non essere visibile agli *reader thread* subito dopo il *write*
    - volatile fa il nuovo valore visibile subito

```java
public class B implements Runnable{
    private Counter c;

    public B(Counter c){
        this.c=c;
    }
    public void run() {
        System.out.println(Thread.currentThread()+": "+c.getCount());
    }
}
```

Questo programma non è ancora corretto

```java
public class A  implements Runnable{

    private Counter c;

    public A(Counter c){
        this.c=c;
    }
    public void run() {
        this.c.increment();
    }
}
```

```java
public class Counter {
    private int count;

    public Counter(){
        this.count=0;
    }

    public void increment(){
        this.count++;
    }

    public int getCount() {
        return count;
    }
}
```

```
public static void main(String[] args) {
      Counter c= new Counter();
      ExecutorService e= Executors.newFixedThreadPool(4);
      e.execute(new A(c));
      e.execute(new B(c));
      e.execute(new A(c));
      e.execute(new B(c));
      e.shutdown();
}
```

```
Output:
Thread[pool-1-thread-4,5,main]: 2
Thread[pool-1-thread-2,5,main]: 2
```

Sincronizzazione

- Soluzione per tutti i problemi
- *Operazioni atomici* : operazioni che succedono in un unico passo, non e possibile interrompere il thread nel mezzo
  - read/write riferimenti e variabili primitivi (no long e double) `a=1;`
  - read/write tutti variabili *volatile* (long, double)
  - a++ non e atomica!!! (*read + write*)

Sincronizzazione

- Evitare errori di calcolo
  - Definire sezioni critici : set di istruzioni che possono essere eseguite da un solo thread alla volta
- Evitare memoria incoerente
  - Proteggere dati condivisi usando i *lock*
  - Ogni accesso a variabili condivisi deve essere protetto (**anche read**), se c'è almeno un thread che modifica l'oggetto!
- Fermare i thread quando non hanno niente da fare - usando condizioni, semafori, join, etc.
- **Tutti i programmi della lezione precedente dovevano usare la sincronizzazione!**

Mutex

- *Mutual exclusion lock* : *lock* che può essere acquisito da un solo *thread* alla volta

- ogni oggetto java (derivato da Object, no int or double) ha un *mutex* incluso, chiamato *monitor* o *lock* “intrinseco”

- Il *monitor* può essere acquisito usando la parola chiave `synchronized`

Metodi synchronized

- Il più semplice modo di proteggere i dati

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

- Prende il monitor dell'oggetto **this**
- Solo un thread alla volta può richiamare un metodo synchronized di un oggetto
- Gli oggetti diversi sono protetti da *monitor* diversi, quindi lo stesso metodo può essere richiamato in contemporaneo per due oggetti diversi

```
public class Student {
    private String name;
    private int math_grade, prog_grade;

    public Student(String name){
        this.name=name;
        this.math_grade=0;
        this.prog_grade=0;}

    public synchronized String getName() {return name;}
    public synchronized void setName(String name) {this.name = name;}

    public synchronized int getMath_grade() {return math_grade;}
    public synchronized void setMath_grade(int math_grade) {
        this.math_grade = math_grade;}

    public synchronized int getProg_grade(){return progr_grade;}
    public synchronized void setProg_grade(int prog_grade){
        this.prog_grade = prog_grade;}

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

Student class

- *Synchronized object*
- Siamo sicuri che nessun *read* o *write* sarà fato in contemporaneo con un altro
- Non ci saranno problemi di memoria invalida

```
public class Professor implements Runnable{

    private String subject;
    private ArrayList<Student> students;

    public Professor(String subject,ArrayList<Student> students){
        this.subject=subject;
        this.students=students;
    }

    public void run() {
        try{
            for (Student s: this.students){
                long sleeptime=System.nanoTime()%10;
                Thread.sleep(sleeptime*1000);
                this.grade(s);
                System.out.println(Thread.currentThread()+" graded "
                +s.getName()+ " average is " + s.getAverage());
            }
        }catch (InterruptedException e){}
     }
   //grade method goes here
}
```

```
private void grade(Student s){
    switch(subject)
    {
    case "math":
        s.setMath_grade(25);
        break;
    case "prog":
        s.setProg_grade(29);
        break;
    default: break;
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
      ArrayList<Student> students= new ArrayList<Student>();
      students.add(new Student("Alina Sirbu"));
      students.add(new Student("Andrea DeSalve"));

      Professor prof1= new Professor("math",students);
      Professor prof2= new Professor("prog",students);

      ExecutorService executor= Executors.newFixedThreadPool(2);
      executor.execute(prof1);
      executor.execute(prof2);
      executor.shutdown();

      executor.awaitTermination(1, TimeUnit.HOURS);

      for (Student s: students){
         System.out.println(s.getName()+ " : " + s.getAverage());
      }

}
```

Output:

```
Thread[pool-1-thread-1,5,main] graded Alina Sirbu average is 12.0
Thread[pool-1-thread-2,5,main] graded Alina Sirbu average is 27.0
Thread[pool-1-thread-2,5,main] graded Andrea DeSalve average is 27.0
Thread[pool-1-thread-1,5,main] graded Andrea DeSalve average is 27.0
Alina Sirbu : 27.0
Andrea DeSalve : 27.0
```

Metodi synchronized

- La classe di un oggetto è un oggetto
- `Student.class` ha il suo *monitor*
- Un metodo *static* sincronizzato
  - acquisisce il *monitor* della classe (non del oggetto)
  - protegge tutti i membri static della classe

Blocchi synchronized

- Stessa parola chiave:

```
synchronized(o)
```

- Metodo synchronized equivalente con
  `synchronized(this):`

```
public void setMath_grade(int math_grade) {
     synchronized(this){
       this.math_grade = math_grade;
     }
}
```

Blocchi *synchronized*

- Sincronizzazione dettagliata
- I due prof lavorano su attributi diversi
- Possiamo usare un monitor di un oggetto diverso per ogni attributo primitivo di Student
- Per gli attributi non primitivi possiamo usare i *monitor* degli attributi stessi
- Il nome non è primitivo però è **immutabile** - non possiamo usarlo per se stesso (this.name=“new” crea un nuovo riferimento)
- Dobbiamo acquisire i due *monitor* dei voti insieme quando calcoliamo la media.

```
public class Student {
    private String name;
    private int math_grade, prog_grade;
    private Integer math_lock, prog_lock, name_lock;

    public Student(String name){
        this.name=name;
        this.math_grade=0;
        this.prog_grade=0;
        this.math_lock=new Integer(0);
        this.prog_lock=new Integer(0);
     this.name_lock=new Integer(0);
}
public String getName() {
    synchronized(this.name_lock){
        return name;
}}
public void setName(String name) {
    synchronized(this.name_lock){
        this.name = name;
}}
```

```
public int getMath_grade() {
    synchronized(this.math_lock){
        return math_grade;
}}
public void setMath_grade(int math_grade) {
    synchronized(this.math_lock){
        this.math_grade = math_grade;
}}

public int getProg_grade() {
    synchronized(this.prog_lock){
        return prog_grade;
 }}
public void setProg_grade(int programming_grade) {
     synchronized(this.prog_lock){
         this.prog_grade = programming_grade;
 }}

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

Variabili di condizione

- Secondo meccanismo di sincronizzazione
- Quando un blocco di codice deve procedere solo se una condizione è soddisfatta (*guarded blocks)*
- Non dobbiamo controllare la condizione in un *active loop* - attesa attiva consuma risorse

```
while(!condition){}
```

- Possiamo fare aspettare il *thread* - *wait*
- Il *thread* verra notificato automaticamente quando c’è stato un cambiamento di stato

Variabili di condizione

- Struttura del codice:

```
synchronized(o){
        while(!condition){
            o.wait();
        }
        //make changes to o and other data
        o.notify();
    }
```

Variabili di condizione

- Ogni oggetto Java ha una variabile di condizione associata
- Verifica della condizione in ambiente protetto - blocco `synchronized`
- `o.wait()` : *Thread* va in *sleep*
- `o.notify()`: Un *thread* in aspettativa è svegliato - deve verificare di nuovo se la condizione è vera (*spurious wakeup*)
- `o.notifyall()`: Tutti i *thread* in aspettativa sono svegliati
- il monitor e la variabile di condizione dello stesso oggetto sono legati
  - non si può fare `o.wait()` o `o.notify()` prima di fare `synchronized(o)`

Produttore - consumatore

- Paradigma molto comune in *multithreading*
- Produttore: crea dati, risultati.
- Consumatore: usa i dati o risultati.
- Esempio semplice:
  - Produttore produce un String
  - Consumatore consuma il String (facendolo sparire)
  - Il consumatore aspetta se il String e vuoto
  - Il produttore aspetta se il String non è stato consumato
  - Mamma , bambino e il cibo

```
public class Food {
    String food;

    public synchronized boolean isEmpty() {
        return food.equals("");
    }

    public synchronized void cook(String food) {
        this.food = food;
    }

    public synchronized void eat(){
        this.food="";
    }

    public Food(){
        this.food="";
    }
}
```

```
public class Mother implements Runnable{
    Food food;
    public Mother(Food food){this.food=food;}
    private void print(String m){
        System.out.println(System.currentTimeMillis()/1000+ " - Mother: "+m);
    }
    public void run() {
        try{
            this.print("Created");
            while(true){
                synchronized(this.food){
                    while(!this.food.isEmpty()){
                        this.print("Waiting for baby to eat");
                        this.food.wait();
                    }
                    this.print("No food left, cooking");
                    this.food.cook("Soup");
                    this.food.notify();
                }
                this.print("Now resting a bit");
                Thread.sleep(10000);
            }
        }catch (InterruptedException e){this.print("Interrupted"); return;}
}}
```

Acquisisce monitor

Acquisisce monitor

while no if

Lascia monitor

Lascia monitor

Reentrant lock

- Il monitor di un oggetto può essere acquisito varie volte dallo stesso *thread* - rientrante
- Nel nostro esempio, nel metodo **`run`** sono richiamati metodi *synchronized* del oggetto **`Food`** in un blocco *synchronized* sullo stesso oggetto
- Questo è possibile a causa del meccanismo dei *lock* rientranti
- Altrimenti il programma blocca se stesso
- Tutti i *lock* in Java sono rientranti

```
synchronized(this.food){
    while(!this.food.isEmpty()){
        this.print("Waiting for baby to eat");
        this.food.wait();
    }
    this.print("No food left, cooking");
    this.food.cook("Soup");
    this.food.notify();
}
```

```
public synchronized void cook(String food) {
        this.food = food;
    }
```

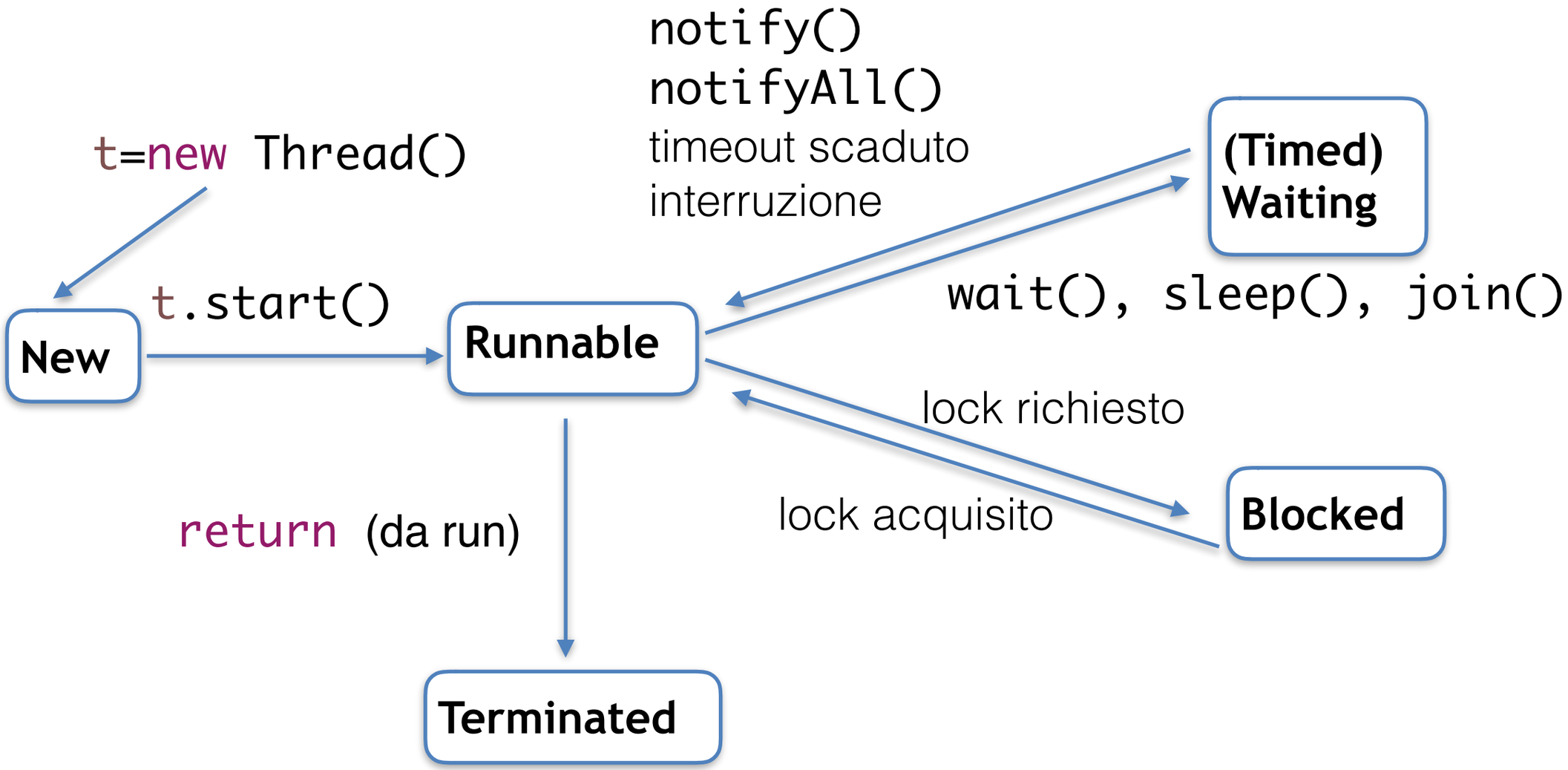
```
public class Child implements Runnable {
     Food food;
     public Child(Food food){this.food=food;}
     private void print(String m){
          System.out.println(System.currentTimeMillis()/1000+ "- Child: "+m);
     }
     public void run() {
          try{
               this.print("Just born");
               while(true){
                    this.print("I am hungry, I want to eat.");
                    synchronized(food){
                         while (this.food.isEmpty()){
                              this.print("There is no food :(");
                              food.wait();
                         }
                         this.print("Yay, eating now :)");
                         food.eat();
                         food.notify();
                    }
                    this.print("Going back to sleep");
                    Thread.sleep(5000);
               }
          }catch (InterruptedException e){ this.print(“Interrupted");
               return;}
}}
```

```
public static void main(String[] args) {
      try{
         Food food= new Food();
         Mother mother = new Mother(food);
         Child child= new Child(food);
         ExecutorService executor=Executors.newFixedThreadPool(2);
         executor.execute(mother);
         executor.execute(child);
         Thread.sleep(20000);
         executor.shutdownNow();
      } catch(InterruptedException e){}

}
```

```
1474724028- Child: Just born
1474724028- Child: I am hungry, I want to eat.
1474724028- Child: There is no food :(
1474724028 - Mother: Created
1474724028 - Mother: No food left, cooking
1474724028 - Mother: Now resting a bit
1474724028- Child: Yay, eating now :)
1474724028- Child: Going back to sleep
1474724033- Child: I am hungry, I want to eat.
1474724033- Child: There is no food :(
1474724038 - Mother: No food left, cooking
1474724038 - Mother: Now resting a bit
1474724038- Child: Yay, eating now :)
1474724038- Child: Going back to sleep
1474724043- Child: I am hungry, I want to eat.
1474724043- Child: There is no food :(
1474724048 - Mother: No food left, cooking
1474724048 - Mother: Now resting a bit
1474724048- Child: Yay, eating now :)
1474724048- Child: Going back to sleep
1474724048- Child: Interrupted
1474724048 - Mother: Interrupted
```

Thread state machine



Concorrenza avanzata

- Da Java 5 e stato introdotto il package `java.util.concurrent`
  - Classi `Lock`, variabili di condizione dedicate
  - Semafori, barriere
  - ———————————————————————-
  - *Concurrent Collections*
  - Variabili *Atomic*
  - Esecuzione dei *thread* controllata e indipendente dalla logica dell'applicazione - Executor
  - 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 a *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();
    try{
        return food.size()==2;
    } finally {
        this.lock.unlock();
    }
}

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<Runnable> people= new ArrayList<Runnable>();
         for (int i=0 ;i<3;i++){
            people.add(new Nanny(food));
         }
         for (int i=0 ;i<4;i++){
            people.add(new Child(food));
         }
         ExecutorService s= Executors.newFixedThreadPool(people.size());
         for (Runnable task : people)
            s.execute(task);
         Thread.sleep(20000);
         s.shutdownNow();
      } catch(InterruptedException e){}

   }
```

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 contemporaneo
- Nessuno può scrivere o leggere se c’è già un scrittore che 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 - implementato in Java

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();
        try {
            this.count++;
            Thread.sleep(100);// simulate longer write
        } catch (InterruptedException e) {
        } finally{
            this.lock.writeLock().unlock();
        }
    }
    public int getCount() {
        this.lock.readLock().lock();
        try {
            Thread.sleep(200);// simulate longer read
            return this.count;
        } catch (InterruptedException e) {
            return Integer.MIN_VALUE;
        } finally{
            this.lock.readLock().unlock();
        }
    }}}
```

```
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");
   }

}
```

```
public class ReaderWriterMain {

   public static void main(String[] args) {
      Counter c= new Counter();
      Random rand = new Random(System.currentTimeMillis());
      double writerProb=0.2;
      ExecutorService es= Executors.newFixedThreadPool(20);
      for (int i=0;i<20;i++){
         if (rand.nextDouble()<writerProb){
            es.execute(new Writer(c));
         }
         else{
            es.execute(new Reader(c));
         }
      }
      es.shutdown();

   }
}
```

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 - usando un Semaforo

- 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++){
            ExecutorService es = Executors.newSingleThreadExecutor();
            es.execute(new Traveler(ticketCounters));
            es.shutdown();
            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
```

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 - programma si ferma*
  - *Livelock - programma è bloccato in un loop*
  - *Starvation - qualche thread non riesce mai a continuare*

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*

Conclusioni sincronizzazione

- Creare thread molto facile
- Sincronizzazione importante!
- Sempre proteggere dati condivisi!
  - scrittura
  - **lettura!!!!**
- 2 meccanismi molto semplici di sincronizzazione
  - molto potenti
  - serve attenzione
  - serve creatività

Esercizio

Gestione laboratorio informatica

- Tre tipi di utenti: studenti, tesisti e professori, ogni utente deve fare una richiesta al tutor per accedere al laboratorio.
- I computer 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
  b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice s
  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.
- Programma JAVA:
  - riceve in ingresso il numero di studenti, tesisti e professori
  - attiva un thread per ogni utente
  - ogni utente accede k volte al laboratorio, con k generato casualmente. sleep() per simulare l'intervallo di tempo tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio
  - il tutor deve coordinare gli accessi al laboratorio.
  - terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.