Laboratorio di Reti - A
Lezione 2
Thread Pool, Callable, sincronizzazione con Locks

24/09/2020

Laura Ricci

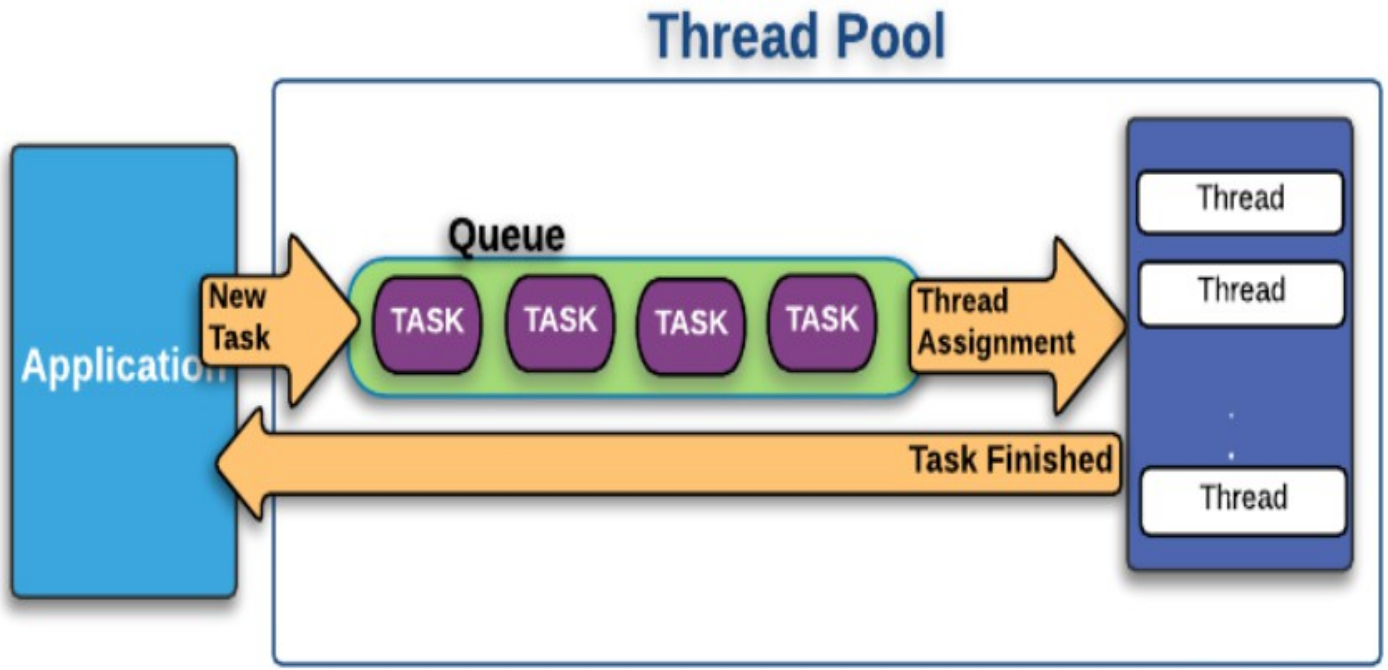
UN THREAD PER OGNI TASK: SVANTAGGI

- thread life cycle overhead
  - overhead per la creazione/distruzione dei threads: richiede un'interazione tra JVM e sistema operativo
  - varia a seconda della piattaforma, ma non è mai trascurabile
  - per richieste di servizio frequenti e 'lightweight' può impattare negativamente sulle prestazioni dell' applicazione
- resource consumption
  - molti threads idle quando il loro numero supera il numero di processori disponibili. Alta occupazione di risorse (memoria,....)
  - mette sotto stress sia il garbage collector che lo schedulatore
- stability limitazione al numero di threads imposto dalla JVM/dal SO

THREAD POOL: MOTIVAZIONI

- un thread per ogni task: una soluzione improponibile, specialmente nel caso di lightweight tasks molto frequenti.
- esiste un limite oltre il quale non risulta conveniente creare ulteriori threads
- obiettivi: definire un limite massimo per il numero di threads che possono essere attivati concorrentemente in modo da:
  - sfruttare al meglio i processori disponibili
  - evitare di avere un numero troppo alto di threads in competizione per le risorse disponibili
  - diminuire il costo per l'attivazione/terminazione dei threads

THREAD POOLING IN BREVE



THREAD POOLING IN BREVE

- l'utente struttura l'applicazione mediante un insieme di tasks.
- task segmento di codice che può essere eseguito da un esecutore
  - in JAVA corrisponde ad un oggetto di tipo Runnable
- Thread esecutore di tasks.
- Thread Pool
  - struttura dati la cui dimensione massima può essere prefissata, che contiene riferimenti ad un insieme di threads
  - i thread del pool possono essere riutilizzati per l'esecuzione di più tasks
  - la sottomissione di un task al pool viene disaccoppiata dall'esecuzione del thread.
  - l'esecuzione del task può essere ritardata se non vi sono risorse disponibili

THREAD POOL: CONCETTI GENERALI

- l'utente
  - crea il pool e stabilisce una politica per la gestione dei thread del pool
    - quando i thread vengono attivati
      - al momento della creazione del pool, on demand, all'arrivo di un nuovo task,....)
    - se e quando è opportuno terminare l'esecuzione di un thread
      - se non c'è un numero sufficiente di tasks da eseguire),...
  - sottomette i tasks per l'esecuzione al thread pool.

- Il supporto, al momento della sottomissione del task, può
  - utilizzare un thread attivato in precedenza, inattivo in quel momento
  - creare un nuovo thread
  - memorizzare il task in una struttura dati (coda), in attesa dell'esecuzione
  - respingere la richiesta di esecuzione del task

- il numero di threads attivi nel pool può variare dinamicamente

JAVA THREADPOOL: IMPLEMENTAZIONE

- fino a JAVA 4 a carico del programmatore
- JAVA 5.0 definisce la libreria java.util.concurrent che contiene metodi per
  - creare un thread pool ed il gestore associato
  - definire la struttura dati utilizzata per la memorizzazione dei tasks in attesa
  - definire specifiche politiche per la gestione del pool
- il meccanismo introdotto permette una migliore strutturazione del codice poichè tutta la gestione dei threads può essere delegata al supporto

JAVA THREADPOOL: IMPLEMENTAZIONE

- alcune interfacce definiscono servizi generici di esecuzione:

```
public interface Executor  {
        public void execute (Runnable task) }
public interface ExecutorService extends Executor
                                    {.. }
```

- diversi servizi che implementano il generico ExecutorService

  (ThreadPoolExecutor, ScheduledThreadPoolExecutor,..)

- la classe Executors che opera come una Factory in grado di generare oggetti di tipo ExecutorService con comportamenti predefiniti.

  - i tasks devono essere incapsulati in oggetti di tipo Runnable e passati a questi esecutori, mediante invocazione del metodo execute()

SOTTOMISSIONE DI TASK AD UN SERVIZIO

```
public class Main {
  public static void main(String[] args) throws Exception
    {
     Server server=new Server();
     for (int i=0; i<10; i++){
          Task task=new Task("Task "+i);
          server.executeTask(task);
          }
     server.endServer();}}
```

- simulazione del comportamento di un servizio (nelle lezioni seguenti vedremo come sviluppare un vero server che riceve richieste dalla rete).
- il servizio esegue i task in modo concorrente utilizzando un thread pool.
- Il servizio viene infine terminato

DEFINIZIONE DI UN SERVER CONCORRENTE

```
import java.util.concurrent.*;
public class Server {
 private ThreadPoolExecutor executor;
 public Server( )
      {executor=(ThreadPoolExecutor) Executors.newCachedThreadPool();}

public void executeTask(Task task){
   System.out.printf("Server: A new task has arrived\n");
   executor.execute(task);
   System.out.printf("Server:Pool Size:%d\n",executor.getPoolSize());
   System.out.printf("Server:Active Count:%d\n",executor.getActiveCount());
   System.out.printf("Server:Completed Tasks:%d\n",
                                    executor.getCompletedTaskCount());

public void endServer() {
        executor.shutdown();}}
```

NewCachedThreadPool

crea un pool con un comportamento predefinito:

- se tutti i thread del pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un nuovo thread.

nessun limite alla dimensione del pool

- se disponibile, viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente.
- se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina

- elasticità: *"un pool che può espandersi all'infinito, ma si contrae quando la domanda di esecuzione di task diminuisce"*

UN TASK CHE SIMULA UN SERVIZIO...

- in questo e nei successivi esempi, il servizio che deve essere eseguito, sarà simulato, per semplicità, inserendo delle attese casuali (`Thread.sleep()`),

```
import java.util.*;
public class Task implements Runnable {
    private String name;
    public Task(String name){
       this.name=name;
                          }
```

UN TASK CHE SIMULA UN SERVIZIO...

```
import java.util.*;
public class Task implements Runnable {
   private String name;
   public Task(String name) {this.name=name;}
   public void run() {
      System.out.printf("%s: Task %s \n",
                        Thread.currentThread().getName(),name);
      try{
        Long duration=(long)(Math.random()*10);
        System.out.printf("%s: Task %s: Doing a task during %d seconds\n",
        Thread.currentThread().getName(),name,duration);
        Thread.sleep(duration);
         }
      catch (InterruptedException e) {e.printStackTrace();}
      System.out.printf("%s: Task Finished %s \n",
                        Thread.currentThread().getName(),name);}}}
```

OSSERVARE L'OUTPUT: IL RIUSO DEI THREAD

```
Server: A new task has arrived
Server: Pool Size: 1
pool-1-thread-1: Task Task 0
Server: Active Count: 1
Server: Completed Tasks: 0
pool-1-thread-1: Task Task 0: Doing a task during 1 seconds
Server: A new task has arrived
Server: Pool Size: 2
Server: Active Count: 1
pool-1-thread-1: Task Finished Task 0
pool-1-thread-2: Task Task 1
pool-1-thread-2: Task Task 1: Doing a task during 7 seconds
Server: Completed Tasks: 0
Server: A new task has arrived
Server: Pool Size: 2
pool-1-thread-1: Task Task 2
```

AUMENTARE IL RIUSO

```
import java.util.*;
public class Main {
   public static void main(String[] args) throws Exception{
     Server server=new Server();
     for (int i=0; i<10; i++){
          Task task=new Task("Task "+i);
           server.executeTask(task);
           Thread.sleep(5000);
                                }
     server.endServer();}}
```

La sottomissione di tasks al pool viene distanziata di 5 secondi. In questo modo l'esecuzione precedente è terminata ed il programma riutilizza dempre lo stesso thread.

AUMENTARE IL RIUSO

```
Server: A new task has arrived
Server: Pool Size: 1
pool-1-thread-1: Task Task 0
Server: Active Count: 1
Server: Completed Tasks: 0
pool-1-thread-1: Task Task 0: Doing a task during 6 seconds
pool-1-thread-1: Task Finished Task 0
Server: A new task has arrived
Server: Pool Size: 1
pool-1-thread-1: Task Task 1
Server: Active Count: 1
pool-1-thread-1: Task Task 1: Doing a task during 2 seconds
Server: Completed Tasks: 1
pool-1-thread-1: Task Finished Task 1
```

AUMENTARE IL RIUSO

```
Server: A new task has arrived
Server: A new task has arrived
Server: Pool Size: 1
pool-1-thread-1: Task Task 2
Server: Active Count: 1
pool-1-thread-1: Task Task 2: Doing a task during 5 seconds
Server: Completed Tasks: 2
pool-1-thread-1: Task Finished Task 2
Server: A new task has arrived
Server: Pool Size: 1
Server: Active Count: 1
pool-1-thread-1: Task Task 3
```

newFixedThreadPool( )

```
import java.util.concurrent.*;
   public class Server {
    private ThreadPoolExecutor executor;
    public Server(){
    executor=(ThreadPoolExecutor) Executors.newFixedThreadPool(2);
      }   …...
```

newFixedThreadPool(int N) crea un pool in cui:

- vengono creati N thread, al momento della inizializzazione del pool, riutilizzati per l'esecuzione di più tasks
- quando viene sottomesso un task T
  - se tutti i threads sono occupati nell'esecuzione di altri tasks,T viene inserito in una coda, gestita automaticamente dall'ExecutorService
  - la coda è illimitata
  - se almeno un thread è inattivo, viene utilizzato quel thread

IL COSTRUTTORE THREAD POOL EXECUTOR

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements ExecutorService
    {public ThreadPoolExecutor
                (int CorePoolSize,
                 int MaximumPoolSize,
                 long keepAliveTime,
                 TimeUnit unit,
                  BlockingQueue <Runnable> workqueue)......}
```

- il costruttore più generale: consente di personalizzare la politica di gestione del pool
- CorePoolSize, MaximumPoolSize, keepAliveTime controllano la gestione dei thread del pool
- workqueue è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione

THREAD POOL EXECUTOR

- CorePoolSize: dimensione minima del pool, definisce il core del pool.

- I thread del core possono venire creati secondo le seguente modalità:
  - PrestartAllCoreThreads( ) al momento della creazione del pool:
  - "on demand" alla sottomissione di un task, si crea un nuovo thread, anche se qualche thread già creato del corecè inattivo.

    obiettivo: riempire il pool prima possibile.
    - quando sono stati creati tutti i threads del core, la politica varia (vedi pagina successiva)

- MaxPoolSize: dimensione massima del pool.
  - non più di MaxpoolSize threads nel pool, anche se vi sono task da eseguire e tutti i threads sono occupati nell'elaborazione di altri tasks.

THREAD POOL EXECUTOR

se tutti i thread del core sono già stati creati e viene sottomesso un nuovo task:

- se un thread del core è inattivo, il task viene assegnato ad esso
- se la coda passata come ultimo parametro del costruttore, non è piena, il task viene inserito nella coda
  - i task vengono poi prelevati dalla coda ed inviati ai thread disponibili
- se coda piena e tutti i thread del core stanno eseguendo un task
  - si crea un nuovo thread attivando così k thread,

$$corePoolSize \leq k \leq MaxPoolSize$$

- se coda piena e sono attivi MaxPoolSize threads
  - il task viene respinto

- È possibile scegliere diversi tipi di coda (tipi derivati da BlockingQueue). Il tipo di coda scelto influisce sullo scheduling.

ELIMINAZIONE DI THREAD INUTILI

Supponiamo che un thread termini l' esecuzione di un task, e che il pool contenga k threads:

- se k ≤ core: il thread si mette in attesa di nuovi tasks da eseguire.
  - attesa è indefinita.
- se k > core, ed il thread non appartiene al core
  - si considera il timeout T definito al momento della costruzione del thread pool
  - se nessun task viene sottomesso entro T, il thread termina la sua esecuzione, riducendo così il numero di threads del pool
- per definire il timeout: occorre specificare
  - un valore (es: 50000) e
  - l'unità di misura utilizzata (es: `TimeUnit. MILLISECONDS`)

THREAD POOL EXECUTOR: TIPI DI CODA

- SynchronousQueue: dimensione uguale a 0. Ogni nuovo task T
  - viene eseguito immediatamente oppure respinto.
  - eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di threads ≤ MaxPoolSize)
- LinkedBlockingQueue: dimensione illimitata
  - E' sempre possibile accodare un nuovo task, nel caso in cui tutti i threads siano attivi nell'esecuzione di altri tasks
  - la dimensione del pool di non può superare core
- ArrayBlockingQueue: dimensione limitata, stabilita dal programmatore

THREAD POOL EXECUTOR: ISTANZE

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
  return new ThreadPoolExecutor(nThreads, nThreads, 0L,
          TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
```

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
  return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,
          TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
```

EXECUTOR LIFECYCLE

- la JVM termina la sua esecuzione quando tutti i thread (non demoni) terminano la loro esecuzione
- è necessario analizzare il concetto di terminazione, nel caso si utilizzi un `Executor Service` poichè
  - i tasks vengono eseguito in modo asincrono rispetto alla loro sottomissione.
  - in un certo istante, alcuni task sottomessi precedentemente possono essere completati, alcuni in esecuzione, alcuni in coda.
  - un thread del pool può rimanere attivo anche quando ha terminato l'esecuzione di un task
- poichè alcuni threads possono essere sempre attivi, JAVA mette a disposizione dell'utente alcuni metodi che permettono di terminare l'esecuzione del pool

EXECUTORS: TERMINAZIONE

- La terminazione del pool può avvenire
  - in modo graduale: "finisci ciò che hai iniziato, ma non iniziare nuovi tasks".
  - in modo istantaneo. "stacca la spina immediatamente"

- `shutdown( )` graceful termination.
  - nessun task viene accettato dopo che la `shutdown( )` è stata invocata.
  - tutti i tasks sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli accodati, la cui esecuzione non è ancora iniziata
  - successivamente tutti i threads del pool terminano la loro esecuzione

- `shutdowNow( )` immediate termination
  - non accetta ulteriori tasks, ed elimina i tasks non ancora iniziati
  - restituisce una lista dei tasks che sono stati eliminati dalla coda
  - tenta di terminare l'esecuzione dei thread che stanno eseguendo i tasks (come?)

EXECUTORS: TERMINAZIONE

ShutdownNow( )

- implementazione best effort
- non garantisce la terminazione immediata dei threads del pool
- implementazione: invio di una interruzione ai thread in esecuzione nel pool
- se un thread non risponde all'interruzione non termina
- infatti, se sottometto il seguente task al pool

```
public class ThreadLoop implements Runnable {
    public ThreadLoop(){};
    public void run( ){while (true)  { }  }  }
```

e poi invoco la shutdownNow( ) ed osservate che il programma non termina

EXECUTORS: TERMINAZIONE

- life-cycle di un pool (execution service)
  - running
  - shutting down
  - terminated

- Un pool viene creato nello stato running, quando viene invocata una `ShutDown()` o una `ShutDownNow()` passa allo stato shutting down, quando tutti i thread sono terminati passa nello stato terminated

- I task sottomessi per l'esecuzione ad un pool in stato Shutting Down o Terminated possono essere gestiti da un rejected execution handler che
  - può semplicemente scartarli
  - può sollevare una eccezione
  - può adottare politiche più complesse (lo vedremo in seguito)

EXECUTORS: TERMINAZIONE

Alcuni metodi definiti dalla interfaccia ExecutorService per gestire la terminazione del pool

- `void shutdown()`
- `List<Runnable> shutdownNow()`
  `restituisce la lista di threads eliminati dalla coda`
- `boolean isShutdown()`
- `boolean isTerminated()`
- `boolean awaitTermination(long timeout, TimeUnit unit)`
  attende che il pool passi in stato Terminated

Per capire se l'esecuizione del pool è terminata:

- attesa passiva: invoco la `awaitTermination( )`
- attesa attiva: invoco ripetuttamente la `isTerminated ( )`

CALLABLE E FUTURE

- un oggetto di tipo `Runnable`
  - incapsula un'attività che viene eseguita in modo asincrono
  - la `Runnable` è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno
- per definire un task che restituisca un valore di ritorno
  - `Interface Callable`: per definire un task che può restituire un risultato e sollevare eccezioni
  - `Future`: per rappresentare il risultato di una computazione asincrona. e definisce metodi
    - per controllare se la computazione è terminata
    - per attendere la terminazione di una computazione (eventualmente per un tempo limitato)
    - per cancellare una computazione, ...
- la classe `FutureTask` fornisce una implementazione della interfaccia Future.

L'INTERFACCIA CALLABLE

```
public interface Callable <V>
    { V call() throws Exception;}
```

- contiene il solo metodo `call()`, analogo al metodo `run( )` dell'interfaccia Runnable
- il codice del task, è implementato nel il metodo call
- a differenza del metodo `run( )`, il metodo `call( )` può restituire un valore e sollevare eccezioni
- il parametro di tipo <V> indica il tipo del valore restituito
- ad esempio: `Callable <Integer>` rappresenta una elaborazione asincrona che restituisce un valore di tipo Integer

CALLABLE: UN ESEMPIO

Definire un task T che calcoli una approssimazione di π, mediante la serie di Gregory-Leibniz (vedi lezione precedente). T restituisce il valore calcolato quando la differenza tra l'approssimazione ottenuta ed il valore di Math.PI risulta inferiore ad una soglia precision. T deve essere eseguito in un thread.

```
import java.util.concurrent.*;
public class pigreco implements Callable <Double>
        {private Double precision;
         public pigreco (Double precision)
              {this.precision=precision;};
         public Double call( )
                    {Double result = <calcolo approssimazione di π>
                     return result }  }
```

L'INTERFACCIA FUTURE

- Il valore restituito dalla Callable, acceduto mediante un oggetto di tipo `<Future>`, rappresenta il risultato della computazione
- Se si usano i thread pools
  - sottomette direttamente l'oggetto di tipo Callable al pool mediante il metodo submit
  - la sottomissione restituisce un oggetto di tipo <Future>
- è possibile applicare all'oggetto Future restituito diversi metodi
  - consentono di individuare se il thread ha terminato la computazione del valore richiesto

L'INTERFACCIA FUTURE

```
public interface Future <V>
    { V get( ) throws...;
      V get (long timeout, TimeUnit) throws...;
      void cancel (boolean mayInterrupt);
      boolean isCancelled(  );
      boolean isDone(  ); }
```

- metodo get()
  - si blocca fino a che il thread non ha prodotto il valore richiesto e restituisce il valore calcolato
- metodo `get (long timeout, TimeUnit)`
  - definisce un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una TimeoutException
- è possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

THREAD POOLING CON CALLABLE

```
import java.util.*;
import java.util.concurrent.*;
public class futurepools {
public static void main(String args[])
    ExecutorService pool = Executors.newCachedThreadPool ( );
    double precision = .........;
    pigreco pg = new pigreco(precision);
    Future <Double> result = pool.submit(pg);
    try{double ris = result.get(1000L,TimeUnit.MILLISECONDS);
       System.out.println(ris+"valore di pigreco");}
    catch(...........){       }
     ..............}}
```

CONDIVIDERE RISORSE

- Threads-and-lock programming: "come guidare una Ford Model T"
  - ti porta da A a B, ma è una macchina difficile da guidare, non affidabile e pericolosa in confronto a nuove tecnologie

- tuttavia:
  - le locks sono ancora utilizzate per scrivere programmi concorrenti
  - sono alla base di altri costrutti ad alto livello
  - anche se non le utilizzerete direttamente, è importante capire come funzionano

- locks: spesso non molto di più che una formalizzazione dei meccanismi hardware sottostanti
  - vantaggio: semplici da utilizzare, quasi tutti i linguaggi le utilizzano, pochi vincoli sul loro utilizzo
  - svantaggio: programmi difficili da mantenere e poco leggibili
  - definizione di meccanismi a più alto livello basati su lock

CONDIVIDERE RISORSE

- scenario tipico di un programma concorrente: un insieme di thread condividono una risorsa.

  più thread accedono concorrentemente allo stesso file, alla stessa parte di un database o di una struttura di memoria

- L'accesso non controllato a risorse condivise può provocare situazioni di errore ed inconsistenze.

  race conditions

- sezione critica: blocco di codice in cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta

- Meccanismi di sincronizzazione per l'implementazione di sezioni critiche
  - interfaccia `Lock` e le sue diverse implementazioni
  - concetto di `monitor`

UN ESEMPIO DI RACE CONDITION

- si considera un conto bancario e due thread che vi accedono in modo concorrente
  - il thread Company versa denaro sul conto corrente
  - il thread BancoMat preleva denaro dal conto corrente
- mostreremo come si possa verificare una race condition, nel caso in cui l'accesso al conto sia incontrollato
- proprietà

  lo stesso numero di versamenti e prelievi dello stesso valore dovrebbe lasciare invariato l'ammontare inizialmente presente sul conto corrente

UN ESEMPIO DI RACE CONDITION

```
public class Account {
    private double balance;
    public  double getBalance() { return balance; }
    public  void   setBalance(double balance) { this.balance =
                                                    balance;}
    public void addAmount(double amount) {
        double tmp=balance;
        try
                { Thread.sleep(10);}
        catch (InterruptedException e) { e.printStackTrace();}
        tmp=tmp+amount;
        balance=tmp;
     }
```

UN ESEMPIO DI RACE CONDITION

```
public void subtractAmount(double amount) {
     double tmp=balance;
     try {
              Thread.sleep(10);
         } catch (InterruptedException e){e.printStackTrace(); }
     tmp=tmp-amount;
     balance=tmp;   } }
```

- un oggetto istanza della classe Account rappresenta un oggetto condiviso tra thread che effettuano versamenti e altri che effettuano prelievi
- l'accesso non sincronizzato alla risorsa condivisa può generare situazioni di inconsistenza.

UN ESEMPIO DI RACE CONDITIONS

```
public class Bancomat implements Runnable {
  private Account account;
  public Bancomat(Account account)
      {
      this.account=account;
      }
  public void run() {
        for (int i=0; i<100; i++)
              {
              account.subtractAmount(1000);
              }
                    }
                                        }
```

UN ESEMPIO DI RACE CONDITION

```
public class Company implements Runnable {
    private Account account;
    public Company(Account account) {
           this.account=account;
                                    }
    public void run() {
          for (int i=0; i<100; i++){
                 account.addAmount(1000);
                                    }}}
```

- un riferimento all'oggetto condiviso Account viene passato esplicitamente ai thread Company e Bancomat
- tutti i thread mantengono un riferimento alla struttura dati condivisa

UN ESEMPIO DI RACE CONDITION

```
public class Main {
public static void main(String[] args) {
   Account account=new Account();
   account.setBalance(1000);
   Company company=new Company(account);
   Thread companyThread=new Thread(company);
   Bancomat bank=new Bancomat(account);
   Thread bankThread=new Thread(bank);
   System.out.printf("Initial Balance:%f\n",account.getBalance());
   companyThread.start();
   bankThread.start();
   try { companyThread.join();
         bankThread.join();
         System.out.printf("Final Balance:%f\n",account.getBalance());
       } catch (InterruptedException e) {e.printStackTrace();}}}
```

UN ESEMPIO DI RACE CONDITION

- output di alcune esecuzioni del programma:

```
Account : Initial Balance: 1000,000000
Account : Final Balance: 17000,000000
Account : Initial Balance: 1000,000000
Account : Final Balance: 89000,000000
                    …...
```

- se avviene una commutazione di contesto prima che l'esecuzione di uno dei metodi di Account termini, lo stato della risorsa può risultare inconsistente

  race condition, codice non rientrante

- non necessariamente l' inconsistenza si presenta ad ogni esecuzione e, se si presenta, non vengono prodotti sempre i medesimi risultati
  - non determinismo
  - comportamento dipendente dal tempo

UN ESEMPIO DI RACE CONDITION

- Race condition
  - un thread invoca i metodi `addAmount` o `subtractAmount` e viene deschedulato prima di avere completato l'esecuzione del metodo
  - la risorsa viene lasciata in uno stato inconsistente
  - un esempio:
    - primo thread esegue subtractAccount: tmp=1000, poi deschedulato prima di completare il metodo
    - secondo thread: completa il metodo addAccount, balance=2000
    - ritorna in esecuzione primo thread: balance=0
- Classe Thread Safe: l'esecuzione concorrente dei metodi definiti nella classe non provoca comportamenti scorretti, ad esempio race conditions
  - `Account` non è una classe thread safe !
  - per renderla thread safe: garantire che le istruzioni contenute all'interno dei metodi addAmount e subtractAmount vengano eseguite in modo atomico / indivisibile / in mutua esclusione

OPERAZIONI "PSEUDO ATOMICHE"

```
public class Counter {
    private int count = 0;
    public void increment()
            {++count; }
    public int getCount()
            {return count; }
}
public class CountingThread extends Thread {
  Counter c;
  public CountingThread (Counter c)
       {this.c=c;}
  public void run() {
     for(int x = 0; x < 10000; ++x)
         c.increment();
         } }
```

OPERAZIONI "PSEUDO ATOMICHE"

```
public class Main  {
    public static void main (String args[])
    {
      final Counter counter = new Counter();
      CountingThread t1 = new CountingThread(counter);
      CountingThread t2 = new CountingThread(counter);
      t1.start(); t2.start();
      try
         { t1.join(); t2.join();
          } catch (InterruptedException e){};
      System.out.println(counter.getCount());
    }
  }
```

OPERAZIONI "PSEUDO ATOMICHE"

- 2 threads, ognuno invoca 10,000 volte il metodo `increment()`: valore finale di counter dovrebbe essere 20,000, invece, ottengo i seguenti valori per 3 esecuzioni distinte del programma

  12349

  12639

  12170

- read-modify-write pattern: JAVA bytecodes generati per l'istruzione ++count

  getfield #2

  iconst_1

  iadd

  putfield #2

- valore di count= 42, entrambi i threads lo leggono, quindi entrambi memorizzano il valore modificato: un aggiornamento viene perduto

MECCANISMI DI SINCRONIZZAZIONE

- JAVA offre diversi meccanismi per la sincronizzazione di threads
- meccanismi a basso livello
  - lock()
  - variabili di condizione associate a lock()
- meccanismi ad alto livello
  - parola chiave `synchronized()`
  - `wait()`, `notify()`, `notifyAll()`
  - `monitors`
- il nostro approccio:
  - iniziamo con i meccanismi a basso livello, con l'obiettivo di capire meglio quelli ad alto livello
  - introduciamo poi quelli ad alto livello motivando le ragione per cui sono stati introdotti.

MECCANISMI DI SINCRONIZZAZIONE: LOCK

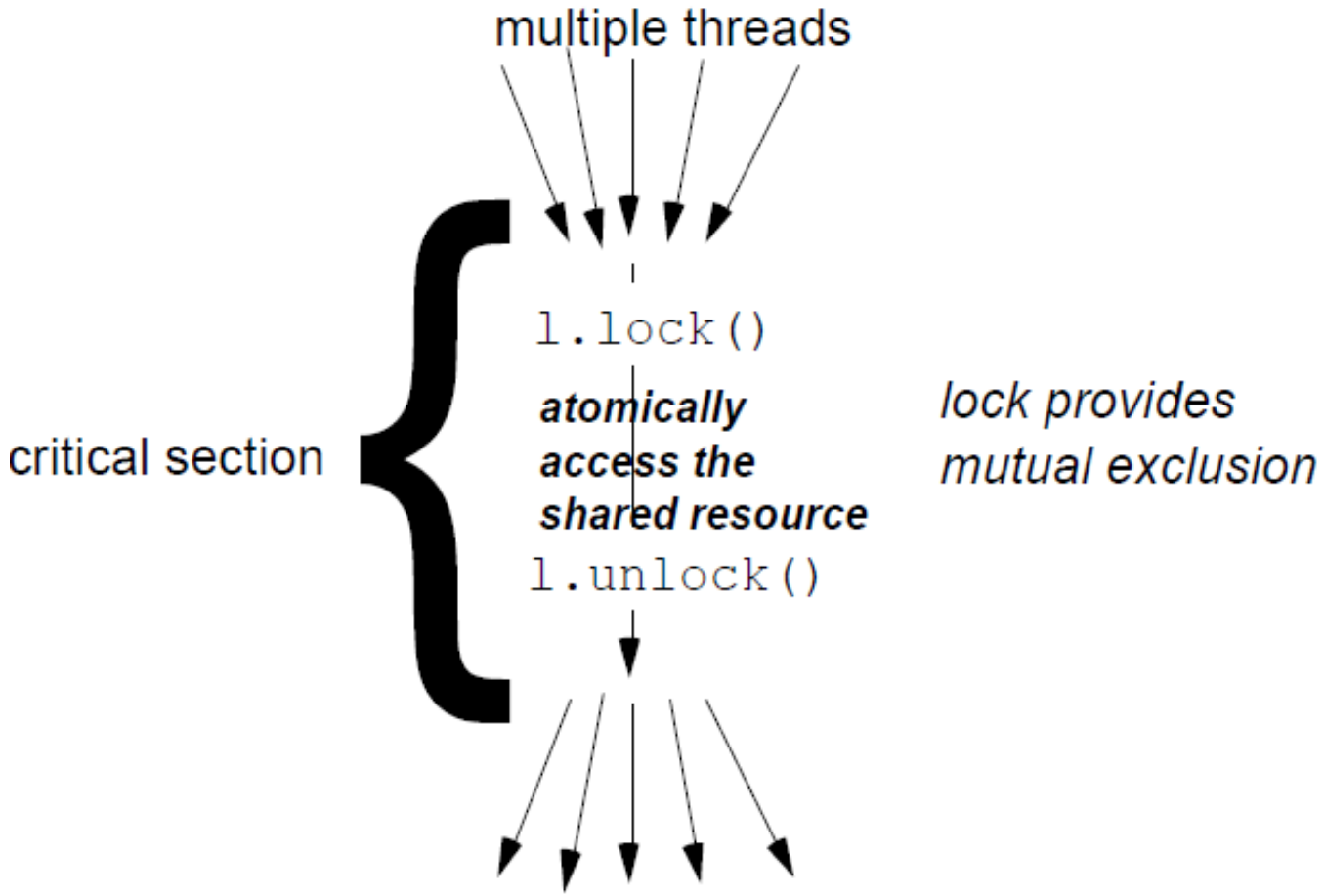
Cosa è una lock in JAVA?

- un oggetto che può trovarsi in due stati diversi
  - "locked"/"unlocked"
  - stato impostato con i metodi: `lock( )` ed `unlock( )`
- un solo thread alla volta può impostare lo stato a "locked", cioè ottenere la lock
  - gli altri thread che tentano di ottenere la lock si bloccano
- quando un thread tenta di acquisire una lock
  - rimane bloccato fintanto che la lock è detenuta da un altro thread,
  - rilascio della lock: uno dei thread in attesa la acquisisce

Metafora: "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

MECCANISMI DI SINCRONIZZAZIONE: LOCK



MECCANISMI DI SINCRONIZZAZIONE: LOCK

Interfaccia:

java.util.concurrent.locks.Lock

Implementazione:

java.util.concurrent.locks.ReentrantLock

Metodi:

- lock ed unlock + altre varianti
- altri metodi (vedere le API): `tryLock(..), lockInterruptibly()`

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

MECCANISMI DI SINCRONIZZAZIONE: LOCK

```
import java.util.concurrent.locks.*;
public class Account {
    private double balance;
    private final Lock accountLock=new ReentrantLock();
    public double getBalance() { return balance; }
    public void setBalance(double balance) { this.balance = balance;}
    public  void addAmount(double amount) {
        accountLock.lock();
        double tmp=balance;
        try {
            Thread.sleep(10);
            } catch (InterruptedException e) { e.printStackTrace();}
        tmp+=amount;
        balance=tmp;
        accountLock.unlock(); }
```

MECCANISMI DI SINCRONIZZAZIONE: LOCK

```
public void subtractAmount(double amount)
    {accountLock.lock();
     double tmp=balance;
     try {
         Thread.sleep(10);
         } catch (InterruptedException e) {e.printStackTrace();}
     tmp-=amount;
     balance=tmp;
     accountLock.unlock();
     } }
```

Output di alcune esecuzioni del programma:

```
Account : Initial Balance: 1000,000000
Account : Final Balance:   1000,000000
Account : Initial Balance: 1000,000000
Account : Final Balance:   1000,000000
```

MECCANISMI DI SINCRONIZZAZIONE: LOCK

- Attenzione ai deadlock:
  - `Thread(A)` acquisice `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!

- L'interfaccia `Lock` e la classe `ReentrantLock` che la implementa include un altro metodo utlizzato per ottenere il controllo della lock: `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

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 !

LOCKS E PERFORMANCE

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,...)

REENTRANT LOCKS

```
import java.util.concurrent.locks.*;
public class ProveReentrant extends Thread {
     static ReentrantLock printer=new ReentrantLock();
     public void foo()
          {printer.lock(); //dosomething
           printer.unlock(); }
     public void run()
          {printer.lock();
           foo();
           printer.unlock(); }
     public static void main (String args[])
          {new ProveReentrant().start();
           System.out.println("terminated");}}
```

REENTRANT LOCKS

- nel programma precedente il thread potrebbe entrare in deadlock con se stesso!
- per evitare queste situazioni: reentrant locks o recursive lock: utilizzano un contatore
  - incrementato ogni volta che un thread acquisisce la lock
  - decrementato ogni volta che un thread rilascia la lock
  - lock viene definitivamente rilasciata quando il contatore diventa 0
  - un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi
- non tutte le lock sono rientranti: POSIX locks non lo sono di default
- il meccanismo delle lock rientranti favorisce la prevenzione di situazioni di deadlock

READ/WRITE LOCKS

```
import java.util.concurrent.locks.*;
public class SharedLocks {
   int a =1000, b=0;
   ReentrantLock l = new ReentrantLock();
   public int getsum ()
     { int result;
       l.lock();
       result=a+b;
       l.unlock();
       return result;};
   public void transfer (int x)
     { l.lock();
       a = a-x;
       b = b+x;
       l.unlock(); }}}
```

READ/WRITE LOCKS

- il codice del lucido precedente:
  - garantisce che la transfer( ) non interferisca con la getSum( )
  - non consente l'esecuzione concorrente di getSum() diverse.
  - se getSum() invocata da thread degradazione di performance inutile

- soluzione: read/write locks (shared locks), implementate in JAVA come:
  - interfaccia ReadWriteLock: mantiene una coppia di lock associate, 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.
    - la write lock è esclusiva.
  - implementazione: ReentrantReadWriteLock()

READ WRITE LOCK

```
import java.util.concurrent.locks.*;
public class SharedLocks extends Thread {
  int a =1000, b=0;
  private ReentrantReadWriteLock readWriteLock = new
                                          ReentrantReadWriteLock();
  private Lock read  = readWriteLock.readLock();
  private Lock write = readWriteLock.writeLock();
  public int getsum ()
   { int result; read.lock(); result=a+b; read.unlock(); return
                                                      result;};
  public void transfer (int x)
   { write.lock(); a = a-x; b = b+x; write.unlock();}}
```

ASSIGNMENT 2: SIMULAZIONE UFFICIO POSTALE

- Simulare il flusso di clienti in un ufficio postale che ha 4 sportelli. Nell'ufficio esiste:
  - un'ampia sala d'attesa in cui ogni persona può entrare liberamente. Quando entra, ogni persona prende il numero dalla numeratrice e aspetta il proprio turno in questa sala.
  - una seconda sala, meno ampia, posta davanti agli sportelli, in cui si può entrare solo a gruppi di k persone

- Una persona si mette quindi prima in coda nella prima sala, poi passa nella seconda sala.

- Ogni persona impiega un tempo differente per la propria operazione allo sportello. Una volta terminata l'operazione, la persona esce dall'ufficio

ASSIGNMENT 2: SIMULAZIONE UFFICIO POSTALE

- Scrivere un programma in cui:
  - l'ufficio viene modellato come una classe JAVA, in cui viene attivato un `ThreadPool` di dimensione uguale al numero degli sportelli
  - la coda delle persone presenti nella sala d'attesa è gestita esplicitamente dal programma
  - la seconda coda (davanti agli sportelli) è quella gestita implicitamente dal `ThreadPool`
  - ogni persona viene modellata come un task, un task che deve essere assegnato ad uno dei thread associati agli sportelli
  - si preveda di far entrare tutte le persone nell'ufficio postale, all'inizio del programma

- Facoltativo: prevedere il caso di un flusso continuo di clienti e la possibilità che l'operatore chiuda lo sportello stesso dopo che in un certo intervallo di tempo non si presentano clienti al suo sportello.