SERVER PROGRAMMING

- **Server Web, Mail Server, File Server, Database Server** sono tutti caratterizzati dalla seguente struttura:

```
ServerSocket socket = new ServerSocket(80); // HTTP
while (true){
    Socket connection = socket.accept( );
    handleRequest(connection); }
```

- La gestione di ogni servizio (**handleRequest**) è indipendente dalla gestione degli altri
- Esempio: la gestione di un messaggio inviato a un mail server non dipende da quella degli altri messaggi elaborati contemporaneamente
- E' naturale associare un thread diverso a ogni richiesta di servizio.

SERVER PROGRAMMING

```
import java.net.*; import java.util.concurrent.*; import java.io.*;

public class ServerPool {

private static final int NTHREADS = 100;

private static final Executor exec=Executors.newFixedThreadPool(NTHREADS);

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

    ServerSocket socket = new ServerSocket(80);

    while (true) {

        final Socket connection = socket.accept();

        Runnable task = new Runnable( ){  // classe anonima
            public void run( )   {  handleRequest(connection); }  };

        exec.execute(task);  }   }   }
```

THREAD POOL: POLITICHE DI SATURAZIONE

- Politica di saturazione:
  - serve se si usano thread pool con code di dimensione limitata
  - indica cosa si deve fare se il pool è saturo, cioè la coda è piena ed i threads del pool sono tutti attivi
- Strategia di default: **abort**, il task viene rifiutato e viene sollevata una **RejectedExecutionException**
- E' possibile definire una politica *ad hoc* mediante un **Rejected Execution Handler**
- JAVA mette a disposizione diversi **Rejected Execution Handler**, ognuno dei quali implementa una diversa politica di saturazione
- Selezione della politica: setRejectedExecutionHandler

THREAD POOL:POLITICHE DI SATURAZIONE

Politiche di saturazione: quando viene sottomesso un task T e la coda è piena, possono essere adottate le seguenti politiche

- **abort**: T viene scartato e viene sollevata un'eccezione
- **discard policy**: rifiuta T, ma non solleva alcun tipo di eccezione
- **discard oldest** scarta il primo task della coda (quello che avrebbe dovuto essere eseguito successivamente) e inserisce T in coda
- **caller-runs**
  - non scarta il task e non solleva eccezioni.
  - cerca di rallentare (throttling, to throttle = strozzare) il flusso dei tasks restituendo alcuni task al chiamante per l'esecuzione. Il task non viene eseguito in un thread del pool, ma nel thread che ha invocato la execute

THREAD POOL: POLITICHE DI SATURAZIONE

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

public class prova {

public static void main (String args[ ]) {
      ThreadPoolExecutor executor =
                  new ThreadPoolExecutor(10,  // corePoolSize
                                         11,  // maxPoolSize
                                         0L,  // keepAliveTime
                                TimeUnit.MILLISECONDS,  // timeUnit
                               new LinkedBlockingQueue<Runnable>(100));
      executor.setRejectedExecutionHandler  (
                                new ThreadPoolExecutor.CallerRunsPolicy());
      }
}
```

THREAD POOL:POLITICHE DI SATURAZIONE

- Consideriamo un **Web Server**
  - riceve richieste di servizio da parte dei clients
  - ogni richiesta corrisponde a un diverso task
- Supponiamo che il Web Server usi un thread pool con una coda a dimensione limitata e una politica di saturazione caller-runs.
- Se tutti i threads sono occupati e la coda è piena, la successiva richiesta di servizio viene eseguita nel thread che ha invocato la execute (il main thread del server)
- Poiché l'esecuzione del servizio richiesto richiederà un intervallo di tempo ΔT, il main thread non sottometterà ulteriori taks per l'esecuzione al pool, durante ΔT
- Alcuni thread del pool possono terminare l'esecuzione del task assegnato durante ΔT, rendendosi disponibili a nuove sottomissioni

THREAD POOL: POLITICHE DI SATURAZIONE

- Inoltre, il main thread, impiegato nell'esecuzione del task rifiutato dal pool, non accetta ulteriori connessioni (non esegue l'accept sul ServerSocket) durante l'intervallo di tempo ΔT
- Le richieste di servizio vengono quindi accodate a livello TCP
- Se l'overload persiste, sarà il livello TCP che eventualmente inizierà a scartare richieste, quando la sua coda sarà, a sua volta, satura
- Conclusione: se il server è sovraccarico, si può implementare una graceful degradation, spostando gradualmente il sovraccarico
  - dai thread del pool alla coda associata al pool,
  - dalla coda all'applicazione,
  - dall'applicazione al livello TCP
  - Il livello TCP è in grado di bloccare il client, mediante il meccanismo di controllo del flusso

BLOCCHI SINCRONIZZATI

- Sincronizzazione di un blocco di codice

```
synchronized (obj)
    { // blocco di codice
    }
```

- L'oggetto obj può essere quello su cui è stato invocato il metodo che contiene il codice (this) oppure un altro oggetto
- Il thread che esegue il blocco sincronizzato deve acquisire la lock sull'oggetto obj
- La lock viene rilasciata nel momento in cui il thread termina l'esecuzione del blocco (es: return, throw, esecuzione dell'ultima istruzione del blocco)
- La lock non viene rilasciata nel caso in cui si invochi un altro metodo all'interno del blocco di codice (lock rientrante)

BLOCCHI SINCRONIZZATI

- Utilizzo:
  - ridurre la lunghezza di una sezione critica
  - rendere indivisibile un frammento di codice che invoca due o più metodi synchronized
- L'esempio presentato nei lucidi successivi mostra in modo operativo

  il vantaggio di diminuire la lunghezza di una sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
double A,B.....
public void setValues (int x, double r){
    double tempA= // questa elaborazione richiede molto tempo...
    double tempB= // questa elaborazione richiede molto tempo...
    synchronized ( this ) {
        A = tempA;
        B = tempB; } }
```

- L'elaborazione che richiede molto tempo viene eseguita fuori dalla sezione critica ed i risultati vengono assegnati a variabili locali
- la sezione critica contiene solo l'aggiornamento delle variabili di istanza del metodo
- si riduce il tempo in cui un thread che ha acquisito la lock rimane all'interno della sezione critica

OTTIMIZZAZIONE DI SEZIONI CRITICHE

La classe **Pair** incapsula

- una coppia di valori interi (x,y)
- un valore intero accesscount.

La coppia di valori deve essere aggiornata in modo atomico e in mutua esclusione rispetto all'aggiornamento della variabile **accesscount**

```
public class Pair {
    private int x, y;
    private int accesscount = 0;
    public Pair (int x, int y)
        {  this.x = x;   this.y = y;  };
    public synchronized void accessincrement( ) { accesscount++;};
    public synchronized int accessreturn( ){ return accesscount;};
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

Per l'aggiornamento della coppia di valori, si definiscono due diversi metodi

- **increment_syn_method**( ): utilizza un metodo sincronizzato
- **increment_syn_block**( ): utilizza un blocco sincronizzato
- **Nota Bene**: la sleep simula un'operazione computazionalmente costosa

```
public synchronized void increment_syn_method(  ){
      x++;  y++;
      try {Thread.sleep(100);} catch(Exception e)  {  };  }
public void increment_syn_block( ){
      synchronized(this){
            x++;  y++;  }
      try {Thread.sleep(100);} catch(Exception e)  {  };  }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class Thread_Syn_Method  extends Thread {
      Pair p;
      public Thread_Syn_Method(Pair p) {this.p=p;};
      public void run(   ) {
            while (true)  {
                  p.increment_syn_method(  );   }  }  }
public class Thread_Syn_Block extends Thread{
      Pair p;
      public Thread_Syn_Block(Pair p) {this.p=p;};
      public void run(   ){
            while (true)  {
                  p.increment_syn_block(); }   }    }
```

OTTIMIZZAZIONE DI SEZIONI CRITICHE

```
public class CountAccess extends Thread {

        Pair p;

        public CountAccess(Pair p){

        this.p = p;

        }

        public void run(  ) {
              while (true) p.accessincrement( );
        }
}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

```
public class MainPair {
public static void main(String args[]){
    Pair syn_method_pair = new Pair(1,2);
    Pair syn_block_pair = new Pair(3,4);
    new Thread_Syn_Method(syn_method_pair).start( );
    new CountAccess(syn_method_pair).start( );
    new Thread_Syn_Block(syn_block_pair).start( );
    new CountAccess(syn_block_pair).start( );
    try      {Thread.sleep(200);}      catch(Exception e)  {  }
    System.out.println("Accessi con blocco sincronizzato " +
        syn_block_pair.accessreturn( ) +
        "\nAccessi con metodo sincronizzato " +
        syn_method_pair.accessreturn( ));  }}
```

OTTIMIZZAZIONE SEZIONI CRITICHE

Risultati di alcune esecuzioni:

Accessi con blocco sincronizzato 2258106 Accessi con metodo sincronizzato 2843

Accessi con blocco sincronizzato 2242580 Accessi con metodosincronizzato 2

Accessi con blocco sincronizzato 3749759 Accessi con metodo sincronizzato 1252

Conclusioni:

- il thread che accede all'oggetto coppia con **blocco** sincronizzato (invece che con un metodo sincronizzato) consente un maggior numero di accessi al thread **CountAccess**, che viene eseguito concorrentemente
- La versione che utilizza il **blocco** sincronizzato aumenta il numero di accessi concorrenti effettuati sull'oggetto coppia
- NOTA BENE: La differenza nel numero di accessi dipende ovviamente dal fatto che il programma simula la computazione costosa con una sleep( ). Se si sincronizza tutto il metodo, quando il thread esegue la sleep rilascia il processore, ma non la lock().