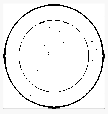
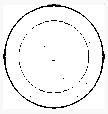


Programmazione concorrente con il linguaggio Java

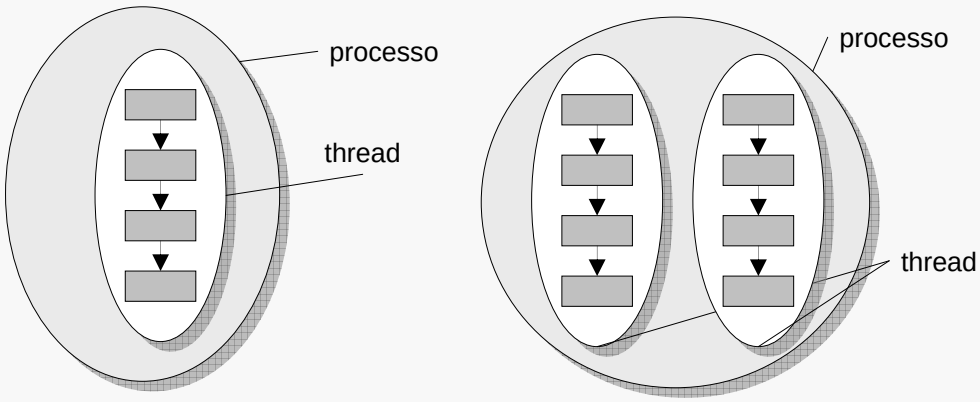
Threads, Mutua Esclusione e Sincronizzazione

Alcuni aspetti architetturali e di prestazioni



Threads

Concetto di thread



- Un **processo** è un **ambiente di esecuzione** costituito da uno o più thread
- Un **thread,** detto anche lightweight process, è un **flusso (sequenziale) di controllo indipendente**
  - Ciascun thread ha il proprio program counter e stack pointer ma condivide con gli altri thread le risorse allocate al processo (modello a memoria condivisa)

Creazione di un thread

Primo metodo: *Sottoclasse Thread*

- Dichiarare una classe (es. **HelloThread)** come sottoclasse di **Thread** (package: **java.lang**)
- Sovrascrivere il metodo **run()** della classe **Thread**
- Creare un oggetto della classe **HelloThread** ed invocare il metodo **start()** per attivarlo

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

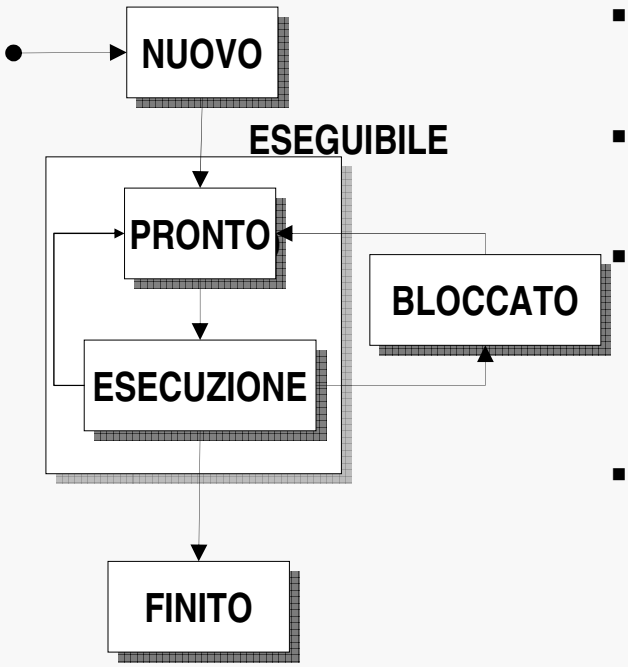
Creazione di un thread

Secondo metodo: *oggetto Runnable*

- Definire una classe (es. **HelloRun**) che implementa l'interfaccia **Runnable.** N.B. **Runnable** ha il metodo **run()**.
- Creare un oggetto della classe **HelloRun**
- Creare un oggetto di classe **Thread** passando un oggetto di classe **HelloRun** al costruttore **Thread().** Attivare l'oggetto **Thread** con il metodo **start()**

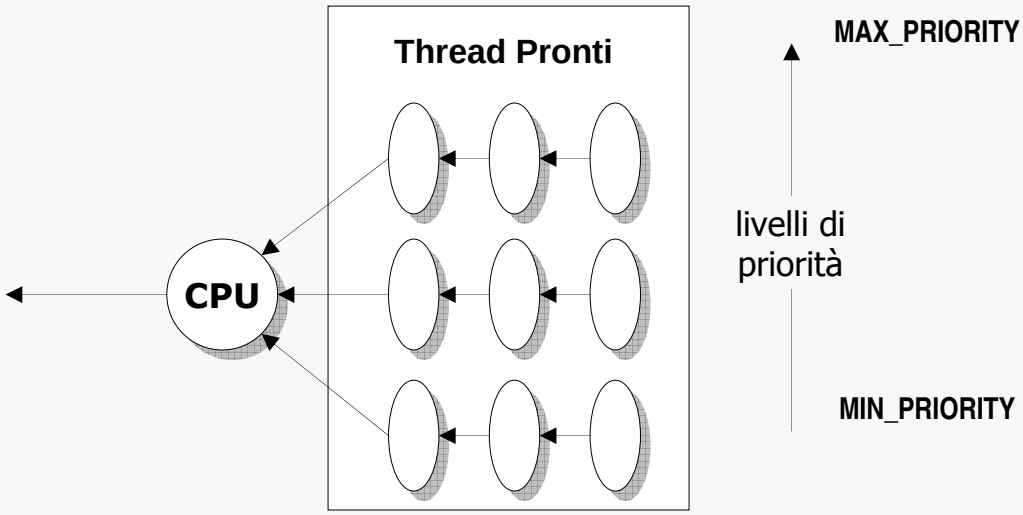
```
public class HelloRun implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRun())).start();
    }
}
```


Stati di un thread



- **In esecuzione**: sta utilizzando la CPU
- **Pronto**: in attesa di utilizzare la CPU
- Gli stati **In esecuzione** e **Pronto** sono **logicamente equivalenti**: i processi sono **Eseguibili**
- **Bloccato**: in attesa di un evento. Un processo bloccato **non è eseguibile** (anche se la CPU è libera)

Priorità nei threads



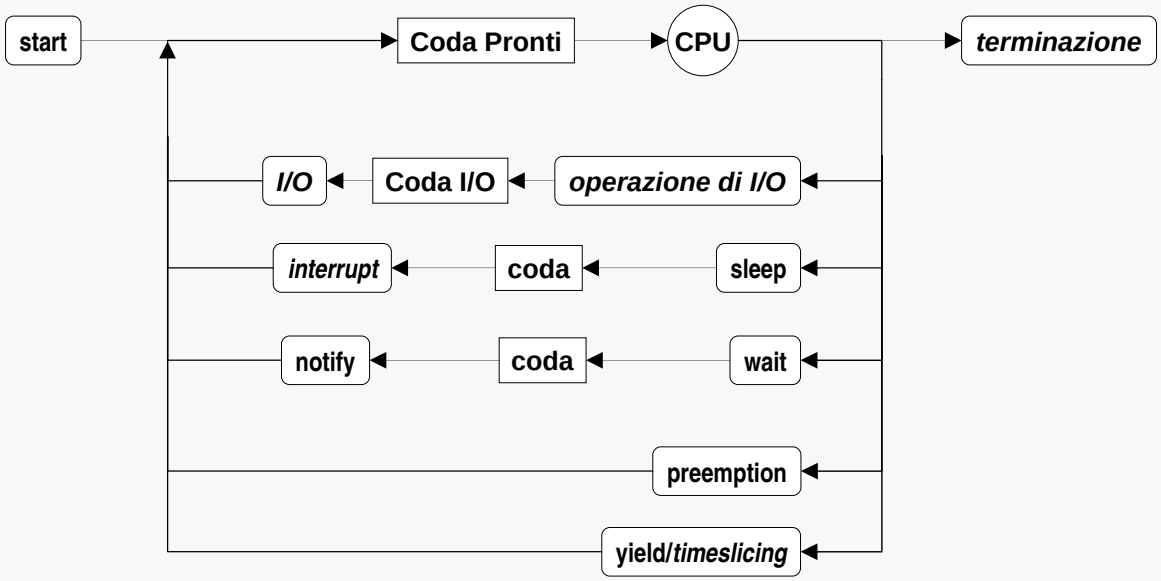
- Un thread eredita la priorità dal thread che lo ha creato, ma la sua priorità può essere modificata (**setPriority**).
- La priorità di default è **MIN_PRIORITY**

Fixed priority scheduling

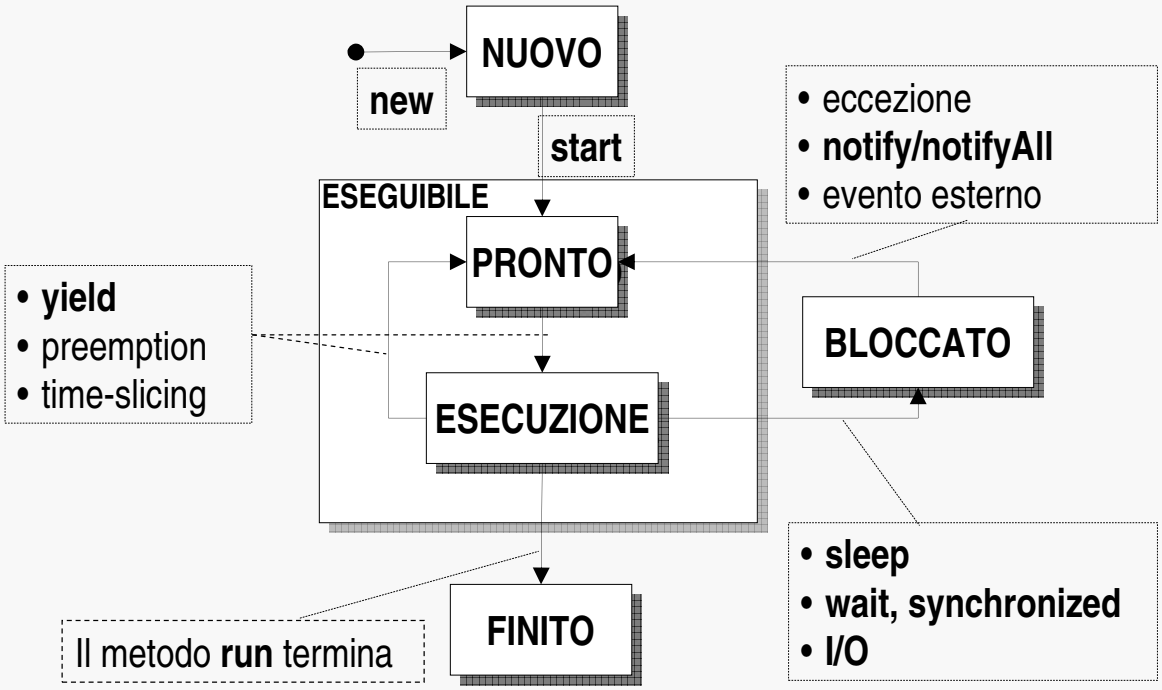
Java implementa *fixed priority scheduling*

- Ad ogni istante, quando più thread sono eseguibili, viene data preferenza ai thread a più alta priorità.
- I thread a più bassa priorità sono eseguiti solo quando quelli a più alta hanno terminato o sono bloccati.
- Lo scheduling è preemptive.
- Se ci sono più thread alla stessa priorità, ne viene scelto uno in modo arbitrario (no fairness).
- Un thread può invocare il metodo **yield** che consente l'esecuzione di un altro thread pronto a pari priorità.
- Il time-slicing non è, in generale, supportato.

Schema dello scheduling



Transizioni di un thread



Metodi della classe Thread

run:

- **contiene il codice del thread**

start:

- **t.start() → esegue il thread t**

getName / getPriority :

- **restituisce il nome o la priorità del thread**

setName / setPriority:

- **modifica del nome o della priorità del thread**

sleep:

- **sospende l'esecuzione del thread per *m* millisecondi (valore di m passato come argomento)**

yield:

- **sospende l'esecuzione del thread corrente consentendo l'esecuzione di altri thread *pronti* e a uguale priorità**

join:

- **t.join() → attende la terminazione del thread t**

Esempio: creazione Thread (1)

```
class SimpleThread extends Thread {
  public SimpleThread(String str) {
        super(str);
  }
  public void run() {
     for (int i = 0; i < 10; i++) {
          System.out.println(i + " " + getName());
          try {
              sleep((long)(Math.random() * 1000));
          } catch (InterruptedException e) {}
        }
        System.out.println("FINITO! " + getName());
  }
}
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Pippo").start();
        new SimpleThread("Pluto").start();
    }
}
```

OUTPUT
0 Pippo
0 Pluto
1 Pluto
2 Pluto
3 Pluto
1 Pippo
4 Pluto
2 Pippo
3 Pippo
4 Pippo
5 Pippo
5 Pluto
6 Pluto
7 Pluto
6 Pippo
8 Pluto
7 Pippo
9 Pluto
FINITO! Pluto
8 Pippo
9 Pippo
FINITO! Pippo

Esempio: creazione Thread (2)

```
class SimpleThread implements Runnable {
    private Thread thr;
    public SimpleThread(String str) {
       thr = new Thread(this, str);
       thr.start();
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + thr.getName());
            try {
                Thread.sleep((long)(Math.random() * 1000));
             } catch (InterruptedException e) {}
        }
        System.out.println("FINITO! " + thr.getName());
    }
}
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Pippo");
        new SimpleThread("Pluto");
    }
}
```


Esempio: metodo join

```
1. public class JoinDemo extends Thread {
2.   public JoinDemo(String name){
3.     super(name);
4.   }

5.   public void run() {
6.     System.out.println(getName()+"start");
7.     for (int tick=0;tick<10000;tick++);
8.       System.out.println(getName()+"end");
9.   }
10.
11.  public static void main(String[] args)
                throw InterruptedException{
12.    System.out.println("main start");
13.    Thread t = new JoinDemo("pippo");
14.    t.start();
15.    t.join();
16.    System.out.println("main end");
17.  }
18.}
```

Esecuzione:

- main start
- pippo start
- pippo end
- main end

Commento linea 14:

- main start
- main end
- pippo start
- pippo end

Esempio: metodo yield

```
1. public class MyThread extends Thread {
2.   private int tick = 1, num;
3.   private final static int NUMTH=2;
4.   public MyThread(int num){this.num = num;}
5.   public void run() {
6.     while(tick < 400000){
7.       tick++;
8.       if((tick % 50000) == 0){
9.         System.out.println("Thread "+num);
10.        yield();
11.      }
12.    }
13.  }
14. public static void main(String[] args) {
15.  MyThread[] runners=new MyThread[NUMTH];
16.  for (int i=0;i<NUMTH;i++){
17.   runners[i] = new MyThread(i);
18.   runners[i].setPriority(NORM_PRIORITY+i);
19.  }
20.  for (int i=0; i<NUMTH;i++)
21.   runners[i].start();
22. }
```

Esecuzione:

Quando il thread a più alta priorità prende la CPU non la rilascia più

Commento linea 18:

Thread hanno pari priorità. Ogni 50000 tick, il thread rilascia la CPU a beneficio degli altri.

Commento linea 10 e 18:

Thread hanno pari priorità, ma il sistema non è *time-sliced*, Quando un thread prende la CPU non la rilascia più.

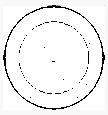
Esempio: server multithreads

```
1.  import java.io.*;
2.  import java.net.*;
3.  public class Server {
4.   static final int PORT = 8080;
5.   public static void main(String[] args) throws IOException {
6.     ServerSocket s = new ServerSocket(PORT);
7.     try {
8.       while ( true ) {
9.         Socket socket = s.accept();
10.              try {
11.              new ServerThread(socket);
12.              } catch(IOException e) {
13.              socket.close();
14.          }
15.          }
16.       } finally {
17.         s.close();
18.     }
19.  }
20. }
```

Esempio: server multithreads

```
class ServerThread extends Thread{
   private Socket socket;
   private BufferedReader in;
   private PrintWriter out;


   public ServerThread(Socket s)
     throws IOException {
     socket = s;
      in = new BufferedReader(
       new InputStreamReader(
        socket.getInputStream()));
     out =new PrintWriter(
       new BufferedWriter(
       new OutputStreamWriter(
       socket.getOutputStream())),
       true);
     start();
   }
```

```
public void run() {
   try {
     while (true) {
       String str=in.readLine();
         if(str.equals("END"))
           break;
         System.out.println
           ("Echo:" + str);
         out.println(str);
     }
   }catch(IOException e){
   }finally{
     try{
       socket.close();
     }catch(IOException e){}
   }
   }

}
```

Mutua esclusione

Mutua esclusione

- L'accesso di thread concorrenti a strutture dati condivise deve essere coordinato
- Se due thread concorrenti eseguono "contemporaneamente" l'operazione **`withdraw`** si possono avere delle inconsistenze

```
int balance;
boolean withdraw(int amt) {
  if (balance – amt >= 0) {
    balance -= amt;
    return true;
  }
  return false;
}

Invariante: balance >= 0
```

```
balance = 80
amt = 50
```

thread 1	thread 2	balance
balance – amt >= 0		80
	balance – amt >= 0	80
balance -= amt		30
	balance -= amt	-20

Sezioni critiche

Sezione critica:

- *Sequenza di istruzioni che deve essere eseguita in modo **mutuamente esclusivo** con altre sezioni critiche*

Classe di sezioni critiche:

- *Insieme di sezioni critiche le cui esecuzioni devono essere **mutuamente esclusive***

Java identifica la sezione critica per mezzo della parola chiave **synchronized**

- Metodo sincronizzato
- Blocco sincronizzato

La mutua esclusione è realizzata per mezzo di un *lock*, o semaforo binario.

Semafori

Un semaforo **s** è una variabile intera non negativa (**s** ≥ **0**) cui é possibile accedere con le operazioni primitive **p** e **v** cosí definite

p(s): **repeat until s > 0;**
s := s – 1;

v(s): **s := s + 1;**

- In caso di contemporanea richiesta di esecuzione da parte di piú thread, tali operazioni vengono eseguite sequenzialmente in ordine arbitrario
- Questa proprietà garantisce che il valore del semaforo venga modificato da un solo processo alla volta

Metodi sincronizzati

synchronized *Method_Name (args) Method_Block*

Java associa un *lock* ad ogni oggetto (non ad ogni metodo)

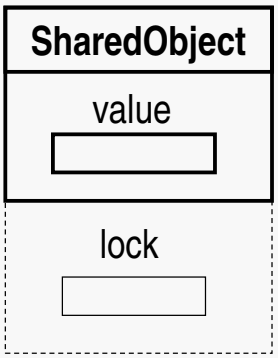
L'esecuzione di un metodo sincronizzato (es. *Method_Name*) consiste in:

- acquisire il lock per conto del thread in esecuzione (entra in sezione critica);
- eseguire il corpo del metodo *Method_Block*
- rilasciare il lock, anche se l'esecuzione del corpo è terminata da un eccezione (rilascia la sezione critica);

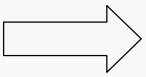
Mentre un thread detiene un lock, nessun altro thread può acquisirlo.

Implementazione semplificata

```
class SharedObject {
  private int value = 0;
  synchronized void write(int v) {
    value = v;
  }
  synchronized int read() {
    return value;
  }
}
```



implementazione concettuale



```
p(this.lock);
body of write;
v(this.lock)
```

```
p(this.lock);
body of read;
v(this.lock)
```


Esempio

```
1  package thread;
2
3  class Risorsa {
4      int[] v = {10, 10};
5      public synchronized void modifica(int val, Thread t, int it) {
6          System.out.println("Thread " + t.getName() +
7          " iterazione " + it +
8          ": Begin modifica");
9          v[0] -= val;
10         try {
11             Thread.sleep((long)(1000*(Math.random())));
12         } catch (InterruptedException e) {}
13         v[1] += val;
14         System.out.println("Thread " + t.getName() +
15         " iterazione " + it +
16         ": End modifica");
17     }
18     public synchronized void leggi(Thread t, int it) {
19         System.out.println("Thread " + t.getName() +
20         " iterazione " + it +
21         ": Begin leggi");
22         System.out.print("v[0] = " + v[0] + "; ");
23         try {
24             Thread.sleep((long)(1000*(Math.random())));
25         } catch (InterruptedException e) {}
26         System.out.println("v[1] = " + v[1]);
27         System.out.println("Thread " + t.getName() +
28         " iterazione " + it +
29         ": End leggi");
30     }
31 }
```

Esempio

```
33 class MioThread extends Thread {
34     Risorsa resource;
35
36     public MioThread(Risorsa res, String nome) {
37         super(nome);
38         resource = res;
39     }
40     public void run() {
41         int i;
42         System.out.println("Start thread " + getName());
43         for (i = 0; i < 5; i++) {
44             if (i % 2 == 0 ) resource.leggi(this, i);
45             else resource.modifica(i, this, i);
46         }
47         System.out.println("End thread " + getName());
48     }
49 }
50 public class Mutex {
51     public static void main(String[] args) {
52         Risorsa r = new Risorsa();
53
54         System.out.println("Begin thread main");
55         for (int i = 0; i < 2; i++)
56             new MioThread(r, "Thread " + i).start();
57         System.out.println("End thread main");
58     }
59
60 }
61
```

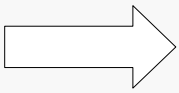

Esempio: output

```
Begin thread main
End thread main
Start thread Thread 0
Thread Thread 0 iterazione 0: Begin leggi
v[0] = 10; Start thread Thread 1
v[1] = 10
Thread Thread 0 iterazione 0: End leggi
Thread Thread 1 iterazione 0: Begin leggi
v[0] = 10; v[1] = 10
Thread Thread 1 iterazione 0: End leggi
Thread Thread 0 iterazione 1: Begin modifica
Thread Thread 0 iterazione 1: End modifica
Thread Thread 1 iterazione 1: Begin modifica
Thread Thread 1 iterazione 1: End modifica
Thread Thread 0 iterazione 2: Begin leggi
v[0] = 8; v[1] = 12
Thread Thread 0 iterazione 2: End leggi
Thread Thread 1 iterazione 2: Begin leggi
v[0] = 8; v[1] = 12
Thread Thread 1 iterazione 2: End leggi
Thread Thread 0 iterazione 3: Begin modifica
Thread Thread 0 iterazione 3: End modifica
Thread Thread 1 iterazione 3: Begin modifica
Thread Thread 1 iterazione 3: End modifica
Thread Thread 0 iterazione 4: Begin leggi
v[0] = 2; v[1] = 18
Thread Thread 0 iterazione 4: End leggi
End thread Thread 0
Thread Thread 1 iterazione 4: Begin leggi
v[0] = 2; v[1] = 18
Thread Thread 1 iterazione 4: End leggi
End thread Thread 1
```

Lock rientranti

- In Java i lock sono **rientranti**, cioè un thread può riacquisire un lock già acquisito (evitano l'**auto-deadlock).**

```
public class Rientrante {
  public synchronized void a() {
    b();
    System.out.println("Sono in a()");
  }
  public synchronized void b() {
    System.out.println("Sono in b()");
  }
  public static void main(String[] args) {
    Rientrante r = new Rientrante();
    r.a();
  }
}
```



```
// output
Sono in b()
Sono in a()
```


Blocchi sincronizzati

synchronized (*Expression*) *Block*

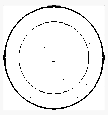
- *Expression* deve produrre un valore **V** di tipo riferimento non-**null**.
  - Se l'espressione termina *abruptly*, l'istruzione termina *abruptly*.
- Il thread acquisisce il lock associato con **V**, esegue *Block* e rilascia il lock associato a **V**.
  - Se il blocco termina *abruptly*, il lock viene rilasciato e l'istruzione termina *abruptly*.

Blocchi e metodi sincronizzati

```
synchronized  void m(args) {
  /* sezione critica   */
}
```

è equivalente a

```
void m(args) {
  synchronized (this)
  { /* sezione critica */       }
}
```

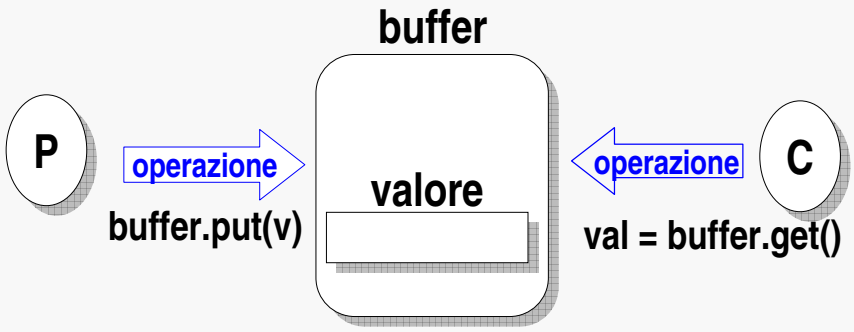
Sincronizzazione fra threads

Problema della sincronizzazione

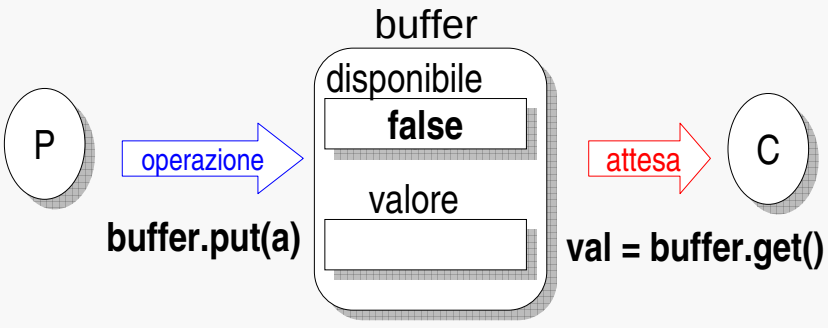
Poiché l'esecuzione di un thread è indipendente, ci possono essere errori di inconsistenza nonostante sia garantita la mutua esclusione sugli oggetti condivisi. Occorre garantire *sincronizzazione* fra i threads.

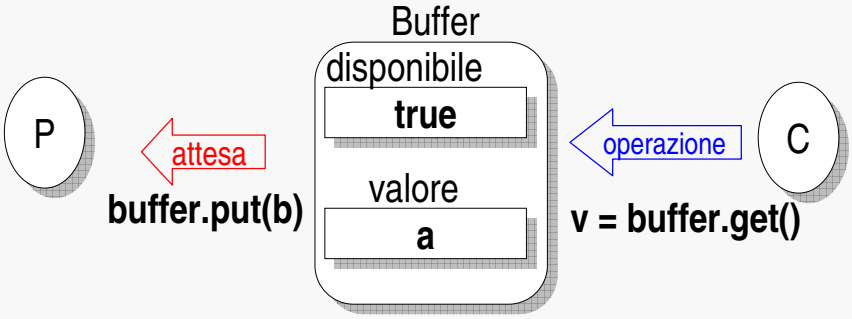
Esempio: Problema del Produttore-Consumatore.

- Requisito: Il consumatore deve prendere *ciascun valore* inserito dal produttore *esattamente una volta*



Produttore-Consumatore





Soluzione: Produttore

```
1.  public class Produttore extends Thread {
2.   private Buffer buffer;
3.   private int numero;

4.   public Produttore(Buffer b, int numero) {
5.     buffer = b;
6.     this.numero = numero;
7.   }

8.   public void run() {
9.     for (int i = 0; i < 10; i++) {
10.      buffer.put(i);
11.      System.out.println("Produttore #"+this.numero+"put:"+i);
12.        }
13.     }
14. }
```

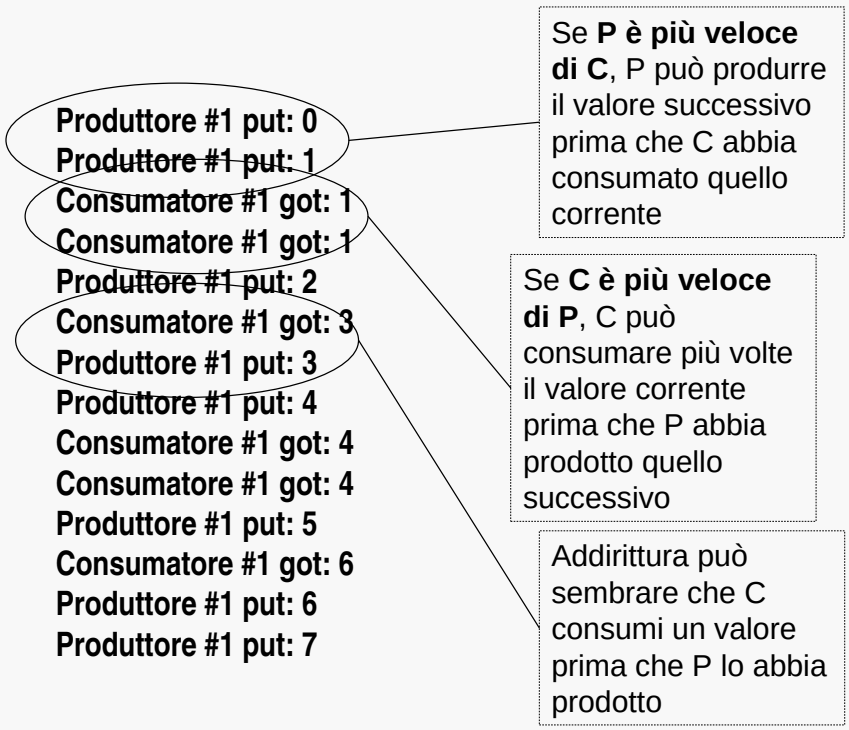

Soluzione: Consumatore

```
1.  public class Consumatore extends Thread {
2.   private Buffer buffer;
3.   private int number;

4.   public Consumatore(Buffer b, int numero) {
5.     buffer = b;
6.     this.number = numero;
7.   }
8.   public void run() {
9.     int valore = 0;
10.    for (int i = 0; i < 10; i++) {
11.      valore = buffer.get();
12.      System.out.println("Consumatore #"+this.number+"get:"+
                        valore);
13.        }
14.     }
15. }
```

Soluzione (errata): il buffer

```
1.  public class Buffer {
2.      private int valore;
3.      private boolean disponibile = false;

4.      public int get() {
5.          while (disponibile == false);
6.          disponibile = false;
7.          return valore;
8.      }

9.      public void put(int value) {
10.         while (disponibile == true);
11.         valore = value;
12.         disponibile = true;
13.     }
14. }
```


Soluzione: Esecuzione



Sincronizzazione: uso del Monitor

Il **monitor** permette di aggiungere alla definizione di tipo di dati astratto una specifica della ***sincronizzazione*** fra i threads per mezzo dell'invocazione dei seguenti metodi:

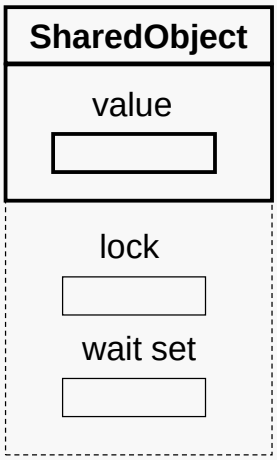
- **wait():** tale metodo rilascia il lock (mutua esclusione) sull'oggetto e sospende il thread che lo invoca in attesa di una notifica.
- **notifyAll():** tale metodo risveglia *tutti* i thread sospesi sull'oggetto in attesa di notifica. I thread risvegliati competono per acquisire il lock (mutua esclusione) sull'oggetto.
- **notify(): tale metodo** risveglia *un* thread scelto casualmente tra quelli sospesi sull'oggetto in attesa di notifica. Il thread risvegliato compete per acquisire il lock (mutua esclusione) sull'oggetto.

Tali metodi **wait()**, **notify()** e **notifyAll()** devono essere invocati dall'interno di un metodo o blocco sincronizzato

Soluzione (corretta): il buffer

```
public class Buffer {
  private int valore;
  private boolean disponibile = false;
  public syncronized int get() {
    while (disponibile == false) {
      try { wait(); } catch(InterruptedException e){}
    }
    disponibile = false;
    notifyAll();
    return valore;
  }
  public synchronized void put(int value) {
    while (disponibile == true) {
      try { wait(); } catch(InterruptedException e){}
    }
    valore = value;
    disponibile = true;
    notifyAll();
  }
}
```

Implementazione concettuale



- Ad ogni oggetto è associato un set di thread detto *wait set*
- il wait set contiene tutti i thread che hanno eseguito il metodo **wait** e che non sono stati ancora notificati con **notify/notifyAll**
- In pratica, wait set può essere realizzato per mezzo di
  - un semaforo **condsem** inizializzato a zero su cui i thread che eseguono **wait** vanno a bloccarsi
  - un contatore **condcount**, inizializzato a zero, che registra il numero dei thread sospesi su **condsem**

Implementazione concettuale *(semplificata)*

Implementazione di un metodo sincronizzato *m*:

```
p(lock)
⟨corpo di m⟩
v(lock)
```

Implementazione della notify:

```
if (condcount > 0) {
    v(condsem);
    condcount--;
}
```

Implementazione della wait:

```
condcount++;
v(lock);
p(condsem);
p(lock);
```

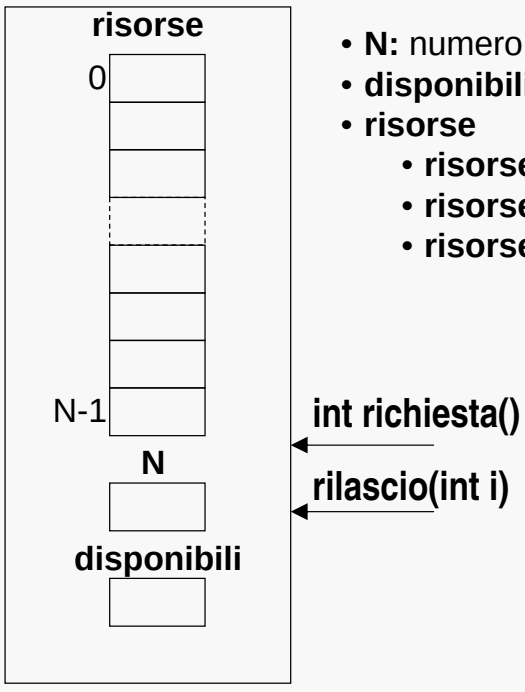
Implementazione della notifyAll:

```
while (condcount > 0) {
    v(condsem);
    condcount--;
}
```

Problemi notevoli

- Gestore di un pool di risorse equivalenti
- Produttori e consumatori nel caso di buffer limitato
- Lettori e scrittori

Gestore di un pool di risorse equivalenti



- **N:** numero totale di risorse
- **disponibili:** risorse disponibili **(initial N)**
- **risorse**
  - **risorse[i] = true** => la risorsa i è disponibile
  - **risorse[i] = false** => la risorsa i è occupata
  - **risorse[i] initial true**

- **richiesta** alloca una risorsa al thread in esecuzione ritornandone il nome; se nessuna risorsa è disponibile, richiesta si blocca fintanto che una risorsa non è disponibile
- **rilascio** rende disponibile la risorsa *i* allocata al thread in esecuzione

Implementazione del Gestore

Classi:

- *DriverGestore* (main):

  crea NUMTHREADS thread che usano il pool di NUMRISORSE risorse equivalenti

- *MioThread:*

  thread che effettua l'accesso alle risorse:

  – richiesta()

  – uso della risorsa (*sleep*)

  – rilascio()

- *Gestore:*

  contiene e gestisce il pool di risorse equivalenti per mezzo delle funzioni **richiesta** e **rilascio**

Implementazione : classe DriverGestore (main)

```
public class DriverGestore {

   public static void main(String[] args) {
      final  int NUMTHREADS = 3;
      final  int NUMRISORSE = 2;

      Gestore g = new Gestore(NUMRISORSE);

      for (int i = 0; i < NUMTHREADS; i++)
         new MioThread("thread[" + i + "]", g).start();
   }
}
```

Implementazione : classe mioThread

```
class MioThread extends Thread {
    final int TIMES = 2;
    Gestore g;
    public MioThread(String name, Gestore gest) {
        super(name);
        g = gest;
    }
    public void run() {
        int r;
        for (int i = 0; i < TIMES; i++) {
            r = g.richiesta();
            try{
                sleep((long)(Math.random()*1000));
            } catch(InterruptedException e){}
            g.rilascio(r);
        }
    }
}
```

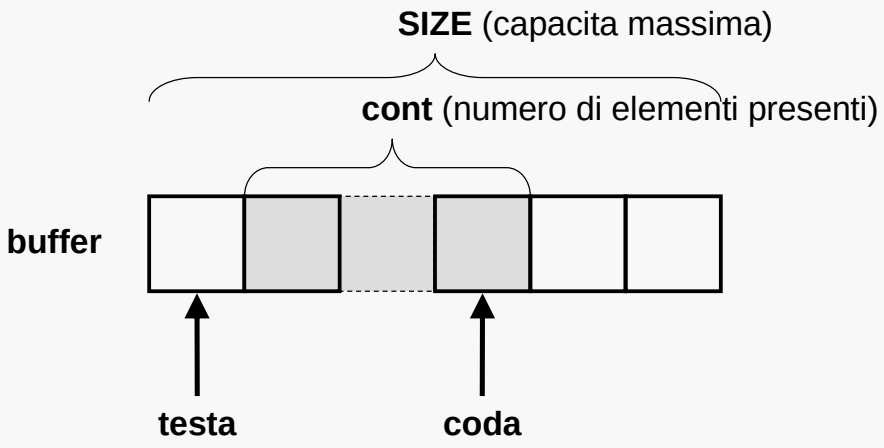

Implementazione: classe Gestore

```
1.  public class Gestore {
2.      // numero totale di risorse
3.      private final int N;
4.      // se risorse[i] = true, la risorsa i è disponibile
5.      private boolean[] risorse;
6.       // numero delle risorse disponibili
7.      private int disponibili;
8.
9.  public Gestore(int numRisorse) {
10.             N = numRisorse;
11.             disponibili = N;
12.             risorse = new boolean[N];
13.             for (int i = 0; i < N; i++) risorse[i] = true;
14.     }
15.
16.     public Gestore() {this(2);}
17.     // continua
```

Implementazione: classe Gestore

```
synchronized int richiesta() {
   if (disponibili <= 0)
    try { wait();  } catch(InterruptedException e){}
   int i = 0;
   while ((i < N) && !risorse[i]) i++;
   risorse[i] = false;
   disponibili--;
   System.out.println("Alloco risorsa " + i);
   return i;
  }

synchronized void rilascio(int i) {
   System.out.println(“Rilascio risorsa " + i);
   risorse[i] = true;
   disponibili++;
   notify();
  }
} // class
```


Buffer di lunghezza limitata



se (**testa == coda**) and (**cont == 0**) allora **buffer vuoto**;
se (**testa == coda**) and (**cont == SIZE**) allora **buffer pieno**;

Implementazione del Buffer

Classi:

- *DriverBuffer* (main):
  - crea NUMPROD Produttori e NUMCONS Consumatori.
  - Produttori e Consumatori sono Threads e devono essere eseguiti (funzione **start()**)
- *Produttore*:
  - inserisce i dati nel buffer (**put**)
  - si deve bloccare se il buffer è PIENO
- *Consumatore*:
  - estrae i dati dal buffer (**get**)
  - si deve bloccare se il buffer è VUOTO
- *Buffer*:
  - contiene e gestisce il buffer per mezzo delle operazioni **put** e **get** secondo la politica FIFO

Implementazione: classe Produttore

```
class Produttore extends Thread {
  final int TIMES = 2;
  Buffer buf;
  public Produttore(String name, Buffer b) {
    super(name);
    buf = b;
  }
  public void run() {
    int r;
    for (int i = 0; i < TIMES; i++) {
      String s = getName() + ": string " + i;
      System.out.println(s);
      buf.put(s);
      try{  sleep((long)(Math.random()*1000));}
         catch(InterruptedException e){}
    }
  }
}
```

Implementazione: classe Consumatore

```
class Consumatore extends Thread {
   final int TIMES = 2;
   Buffer buf;

   public Consumatore(String name, Buffer b) {
      super(name);
      buf = b;
   }
   public void run() {
      int r;
      for (int i = 0; i < TIMES; i++) {
         String s;
         s = (String)buf.get();
         System.out.println(getName() +
                                 " consuma " + s);
         try{
            sleep((long)(Math.random()*1000));
         } catch(InterruptedException e){}
      }
   }
}
```


Implementazione: classe DriverBuffer (main)

```
public class DriverBuffer {
       public static void main(String[] args) {

       final int NUMPROD = 3;
       final int NUMCONS = 3;

       Buffer buf = new Buffer();

       for (int i = 0; i < NUMCONS; i++)
              new Consumatore("cons[" + i + "]",
       buf).start();

       for (int i = 0; i < NUMPROD; i++)
              new Produttore("prod[" + i + "]",
       buf).start();
       }
}
```

Implementazione: classe Buffer

```
public class Buffer {
  private Object[] buffer;  //il buffer
  private final int SIZE;  //capacità del buffer
  private int testa;        //punto di inserimento
  private int coda;         //punto di estrazione
  private int cont;         //num oggetti presenti nel buffer

  public Buffer(int sz) {
    SIZE = sz;
    buffer = new Object[SIZE];
    cont = 0; testa = 0; coda = testa;
  }

  public Buffer() {
    this(10);
  }
```


Implementazione: classe Buffer

```
synchronized void put(Object elem) {
    while (cont >= SIZE)
        try { wait(); } catch(InterruptedException e){}
    buffer[testa] = elem;
    testa = (testa + 1) % SIZE;
    cont++;
    notifyAll();
    }

synchronized Object get() {
        Object elem;
        while (cont <= 0)
            try { wait(); } catch(InterruptedException e){}
        elem = buffer[coda];
        coda = (coda + 1) % SIZE;
        cont--;
        notifyAll();
        return elem;
    }
}
```

Produttori e Consumatori (socket, buffer illimitato)

```
1.  class MyMsgQueue {
2.    private LinkedList<Message> msgqueue;

3.    public MyMsgQueue() {
4.      msgqueue = new LinkedList<Message>();
5.    }
6.    synchronized void add(Message m){
7.      msgqueue.addLast(m);
8.      notify();
9.    }
10.   synchronized Message remove(){
11.     if (msgqueue.size() == 0) {
12.       try {
13.         wait();
14.       } catch(InterruptedException e){}
15.     }
16.     return msgqueue.removeFirst();
17.   }
18. }
```

- L'implementazione di **LinkedList** non è thread-safe
- **LinkedList** deve essere sincronizzato esternamente

1. Sincronizzazione su di un oggetto che incapsula **LinkedList**
2. **Collections.synchronizedList** che ritorna una lista thread basate su quella specificata come argomento

Produttori e Consumatori (socket, buffer illimitato)

```
1.public class Buffer {
2.  public static final int PORT = 8080;
3.  public static void main(String[] args)
4.                       throws IOException {
5.    MyMsgQueue msgqueue = new MyMsgQueue();
6.    ServerSocket servsock =
7.                    new ServerSocket(PORT);
8.    try {
9.      while(true) {
10.       Socket sock = servsock.accept();
11.       new MyThread(sock, msgqueue);
12.     }
13.   } catch(IOException e) {
14.   } finally {
15.       servsock.close();
16.   }
17. }
18.}
```

Linea 11. Il thread riceve come parametro la connessione da gestire (**socket**) e la coda dei messaggi (**msgqueue**), una struttura dati condivisa

Produttori e Consumatori (socket, buffer illimitato)

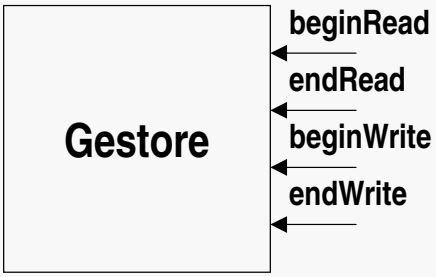
```
1.class MyThread extends Thread {
2.  private MyMsgQueue mq;
3.  private Socket socket;
4.  private ObjectInputStream in;
5.  private ObjectOutputStream out;
6.
7.  MyThread(Socket s, MyMsgQueue amq)
8.                     throws IOException {
9.    socket = s;
10.   mq = amq;
11.   in = new ObjectInputStream(
12.              socket.getInputStream());
13.   out = new ObjectOutputStream(
14.             socket.getOutputStream());
15.   start();
16. }
17. // continua
```


Produttori e Consumatori (socket, buffer illimitato)

```
18.  public void run() {
19.    try {
20.     Message m = (Message)in.readObject();
21.     if (m.getType() == 0)  // produttore
22.       mq.add(m);
23.     else { // consumatore
24.       Message app = mq.remove();
25.       out.writeObject(app);
26.     }
27.   } catch(IOException e){
28.   } catch(ClassNotFoundException e){}
29.   finally {
30.     try {
31.       socket.close();
32.     } catch(IOException e){}
33.   }
34.  }
35. }
```

Linea 24. Il thread sospende la propria esecuzione se non ci sono messaggi disponibili

Lettori e scrittori



Il gestore **ReadWrite** realizza la *politica di sincronizzazione*

Protocollo

```
// Lettore
...
rw.beginRead()
lettura
rw.endRead()
...
```

```
// Scrittore
...
rw.beginWrite()
scrittura
rw.endWrite()
...
```


Lettori e scrittori

Politica che garantisce mutua esclusione ed assenza di starvation

A. i lettori possono accedere contemporaneamente alla risorsa
B. gli scrittori hanno accesso esclusivo alla risorsa
C. lettori e scrittori si escludono mutuamente nell'uso della risorsa
D. un nuovo lettore non può acquisire la risorsa se c'è uno scrittore in attesa
E. tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore

Implementazione dei Lettori e scrittori

Classi:

- Gestore:
- Scrittore:
  accesso esclusivo alla risorsa (**write()**)
- Lettore:
  accesso concorrente alla risorsa con altri lettori (**read()**)

Implementazione: classe Scrittore

```
class Scrittore extends Thread {
  final int TIMES = 5;
  ReadWrite rw;
  public Scrittore(String name, ReadWrite rw) {
    super(name);
    this.rw = rw;
  }
  public void run() {
    for (int i = 0; i < TIMES; i++) {
      String name = getName();
      rw.beginWrite();
      System.out.println(name + " begin write");
      try{
        sleep((long)(Math.random()*1000));
      } catch(InterruptedException e){}
      System.out.println(name + " end write");
      rw.endWrite();
    }
  }
}
```

Implementazione: classe Lettore

```
class Lettore extends Thread {
  final int TIMES = 3;
  ReadWrite rw;
  public Lettore(String name, ReadWrite rw) {
     super(name);
     this.rw = rw;
  }
  public void run() {
     String name = getName();
     for (int i = 0; i < TIMES; i++) {
       rw.beginRead();
       System.out.println(name + " begin read");
       try{sleep((long)(Math.random()*1000));
       }catch(InterruptedException e){}
       System.out.println(name + " end read");
       rw.endRead();
     }
  }
}
```


Implementazione: classe DriverReadWrite (main)

```
public class DriverReadWrite {
  public static void main(String[] args) throws InterruptedException {
    final  int NUMREADERS = 2, NUMWRITERS = 2;
    Thread[] t = new Thread[NUMREADERS + NUMWRITERS];
    Gestore g = new Gestore();
    System.out.println("MAIN: BEGIN");
    for (int i = 0; i < NUMREADERS; i++) { // creaz. & attiv. lettori
      t[i] = new Lettore("lettore[" + i + "]", g);
      t[i].start();
    }
    for (int i = 0; i < NUMWRITERS; i++) { // creaz. & attivaz. scrittori
      t[i + NUMREADERS] = new Scrittore("scrittore[" + i + "]",g);
      t[i + NUMREADERS].start();
    }
    for (int i = 0; i < NUMREADERS+NUMWRITERS;i++) // attendo tutti i thr.
      t[i].join();
    System.out.println("MAIN: END");
  }
}
```

Implementazione: classe Gestore (errata)

```
public class Gestore {

 private int aw = 0; // num active writers
 private int rr = 0; // num running readers
 private boolean busy_writing = false; // a writer is in

 public Gestore() {}

 synchronized void beginRead() {
  while (aw > 0)
   try {
    wait();
   } catch(InterruptedException e){}
  rr++;
 }

 synchronized void endRead() {
  rr--;
  notifyAll();
 }
 // continua
```


Implementazione: classe ReadWrite (errata)

```
 synchronized void beginWrite() {
  aw++;
  while (busy_writing || (rr > 0))
   try {
    wait();
   } catch(InterruptedException e){}
  busy_writing = true;
 }

 synchronized void endWrite() {
  busy_writing = false;
  aw--;
  notifyAll();
 }
}
```

Questa soluzione causa la starvation dei lettori.

Implementazione: classe Gestore (corretta)

```
public class Gestore {
    private boolean okReader = true; // readers can proceed
    private boolean okWriter = true; // a writer can proceed
    private int aw = 0; // active writers
    private int rr = 0; // running readers
    private int ar = 0; // active readers
    private boolean busy = false; // a writer is in

    public Gestore() {}
```


Implementazione: classe ReadWrite (corretta)

```
synchronized void  beginRead() {
  ar++;
  while ((aw > 0) && okWriter)
    try {wait();
    } catch(InterruptedException e){}
  ar--;
  rr++;
}
synchronized void endRead() {
  rr--;
  if (aw > 0) {
    okWriter = true;
    okReader = false;
  }
  if (rr == 0) notifyAll();
}
```

Implementazione: classe ReadWrite (corretta)

```
synchronized void beginWrite() {
 aw++;
 while (busy || (rr > 0) || ((ar > 0) && okReader))
  try {
       wait();
  }catch(InterruptedException e){}
  busy = true;
}
synchronized void endWrite(){
 busy = false;
 aw--;
 if (ar > 0) {
  okReader = true;
  okWriter = false;
 }
 notifyAll();
 }
} // Gestore
```


Implementazione: commento

Questa soluzione è “complicata” per vari motivi:

- ad un certo istante le condizioni di sincronizzazione per i lettori e gli scrittori possono essere verificate contemporaneamente perciò *bisogna programmare esplicitamente la politica di priorità* degli uni sugli altri;
- *la politica di priorità dipende dallo stato/storia della risorsa*
  (es. dopo un treno di lettori uno scrittore; dopo uno scrittore un treno di lettori)
- lettori e scrittori sospendono l’esecuzione nello stesso wait set e quando bisogna risvegliarli non c’è modo di specificare l’uno o l’altro;

  Vediamo adesso una soluzione basata sui *semafori privati* che costituiscono una risposta ai problemi 1–3.

Semafori privati

In seguito alla modifica della risorsa, le condizioni di sincronizzazione di più thread sospesi possono essere simultaneamente verificate. Quale thread sospeso riattivare?

Nel caso che si voglia realizzare un particolare ***politica di gestione delle risorse****,*

- la scelta deve avvenire sulla base di un algoritmo specifico per la risorsa
- Non ci si può basare sul meccanismo **wait-notify** ma si utilizzano I ***semafori privati***: solo un (gruppo di) thread può fare **p()**; chiunque può fare **v()**.

Semaphore

```
public class Semaphore {
   private int s;
   public Semaphore(int v) {
        s = v;
   }
   // a mutex by default
   public Semaphore() { this(1); }

   synchronized public void p() {
     while (s <= 0)
       try{
         wait();
       } catch(InterruptedException e){}
     s--;
   }

   synchronized public void v() {
     s++;
     notifyAll();
   }
}
```

Implementazione: Classe Gestore (semafori privati)

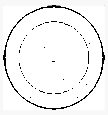
```
public class Gestore {
    private int ar = 0; // active readers
    private boolean aw = false; // active writer
    private int br = 0; // blocked readers
    private int bw = 0; // blocked writers
    private Semaphore rs = new Semaphore(0);  // priv. sem.
    private Semaphore ws = new Semaphore(0); // priv. sem.

    // continua
```


Implementazione: Classe Gestore (semafori privati)

```
 public void beginRead() {
    synchronized(this) {
       if (!aw && (bw == 0)) {
          rs.v();
          ar++;
       }
       else br++;
    }
    rs.p(); // potentially blocking; outside of mutex
 }
 synchronized public void endRead() {
    ar--;
    if ( ar == 0 && bw > 0) {
       aw = true;
       bw--;
       ws.v(); // awake writers
    }
}
```

Implementazione: Classe Gestore (semafori privati)

```
public void beginWrite() {
   synchronized(this) {
      if (ar == 0 && !aw) {
         ws.v();
         aw = true;
      } else bw++;
   }
   ws.p(); // potentially blocking; outside of mutex
}
synchronized public void endWrite() {
   aw = false;
   if (br > 0) while (br >0) {
         br--; ar++;
         rs.v(); // awake readers
   } else if (bw > 0) {
      aw = true; bw--;
      ws.v(); // awake writers
   }
}
} // class ReadWrite
```

Classi di Threads

Thread temporizzati, user thread e daemon, avvio e terminazione di una JVM

Timer thread (thread temporizzati)

- La classe **Timer** permette di schedulare l'esecuzione di istanze della classe **TimerTask** (*timer thread*) ad istanti o ad intervalli regolari prefissati
- Implementazione e scheduling di un timer thread
  1. definire una sottoclasse di **TimerTask**, sovrascrivendo il metodo **run**;(*)
  2. Creare un *timer* istanziando la classe **Timer**
  3. Creare un *timer thread* istanziando la sottoclasse;
  4. Schedulare il thread per mezzo del metodo **schedule** del timer.

(*) **TimerTask** e **run()** sono **abstract** **(vedi esempio)**

Timer thread: example

```
import java.util.Timer;
import java.util.TimerTask;

public class Reminder {
  class RemindTask extends TimerTask { // step 1: subclass of
     public void run() {               //    TimerTask
     System.out.println("E' l'ora!");
     timer.cancel();
    }
  Timer timer;
  public Reminder(int seconds) {
    timer = new Timer(); // step 2 (creazione timer)
    // steps 3 and 4: creazione e scheduling di un timer thread
    timer.schedule(new RemindTask(), seconds*1000);
  }
  public static void main(String args[]) {
    new Reminder(5);
  }
} // class Reminder
```

Tipi di thread: daemon e user-thread

- Ogni thread può essere marcato *daemon* o un *user-thread*
- Un thread eredita la marcatura del thread che lo crea.
- Un thread può essere marcato daemon con l'operazione **Thread.setDaemon()**

Avvio e terminazione di una JVM

- La JVM viene avviata con un *user-thread* che chiama il metodo **main**
- La JVM rimane in esecuzione finché non si verifica almeno uno dei seguenti eventi
  - viene eseguito il metodo **Runtime.exit()**
  - tutti gli *user-thread* terminano la loro esecuzione (ritornano dal metodo **run(),** oppure lanciano un eccezione)

Terminazione di un timer thread

Un timer thread può essere terminato

- invocando il metodo **TimerTask.cancel** che cancella questo timer thread
- rendendo il timer thread un daemon (**new Timer(true)**)
  se gli unici thread attivi sono demoni, la JVM termina
- quando il timer thread ha terminato la sua esecuzione, rimuovendo tutti i riferimenti all'oggetto **Timer**
- Invocando **cancel** dell'oggetto **Timer**.
- invocando il metodo **System.exit**

Commento

i timer thread non sono gli unici thread che impediscono al programma di terminare: ad esempio AWT (vedi esercizio **ReminderBeep**)

Timer task: RemainderBeep

```
import java.util.Timer;
import java.util.TimerTask;
import java.awt.Toolkit;

public class ReminderBeep {
 Toolkit toolkit;
 Timer timer;
 public ReminderBeep(int seconds) {
  toolkit = Toolkit.getDefaultToolkit();
  timer = new Timer();
  timer.schedule(new RemindTask(), seconds*1000);
 }
 class RemindTask extends TimerTask {
  public void run() {
   System.out.println("E' l'ora");
   toolkit.beep();
   //timer.cancel(); (2)
   System.exit(0);   (1)
  }
 }
}
```

- (1) commentata; (2) no: la JVM non termina
- viceversa: la JVM termina

Metodo **TimerTask.cancel**

- L'operazione ***TimerTask.cancel*** *cancella la prossima esecuzione di questo timer task*
  - Se il timer task è stato schedulato per un'esecuzione singola e non è ancora andato in esecuzione oppure se il timer task non è stato ancora schedulato, allora il timer task non andrà mai in esecuzione
  - Se il task è in esecuzione quando viene chiamato questo metodo, allora il task completa la sua esecuzione e non andrà più in esecuzione un'altra volta
- Il metodo può essere invocato più volte: le invocazioni successive alla prima non hanno effetto

Metodo **Timer.schedule**: esecuzione ritardata

Esecuzione ritardata

- **void schedule(TimerTask task, Date time)** schedula **task** per l'esecuzione all'istante **time**
- **void schedule(TimerTask task, long delay)** schedula **task** per l'esecuzione dopo il ritardo **delay** (in ms)

Metodo **Timer.schedule**: esecuzione periodica

Esecuzione periodica a periodo fisso (fixed-delay execution)

- **void schedule(TimerTask task, Date firstTime, long period)** schedula **task** per l'esecuzione periodica, con periodo **period** (in ms), a partire dall'istante **firstTime**
- **void schedule(TimerTask task, long delay, long period)** schedula **task** per l'esecuzione periodica, con periodo **period** (in ms), dopo il ritardo **delay**

(esempio "Esecuzione periodica di un thread")

Esecuzione periodica: esempio

```
import java.util.Timer;
import java.util.TimerTask;

public class PeriodicReminder {
  Timer timer;
  public PeriodicReminder() {
    timer = new Timer();
    timer.schedule(new RemindTask(), 0, 1000);
  }
  class RemindTask extends TimerTask {
    int numVolte = 10;
    public void run() {
      if (numVolte > 0) {
        System.out.println("E' l'ora");
        numVolte--;
      } else {
        System.out.println("E' l'ora");
        timer.cancel();
      }
    } // run
  } // RemindTask
  // main
```

- ritardo iniziale: 0
- periodo: 1*1000

Metodo Timer.schedule

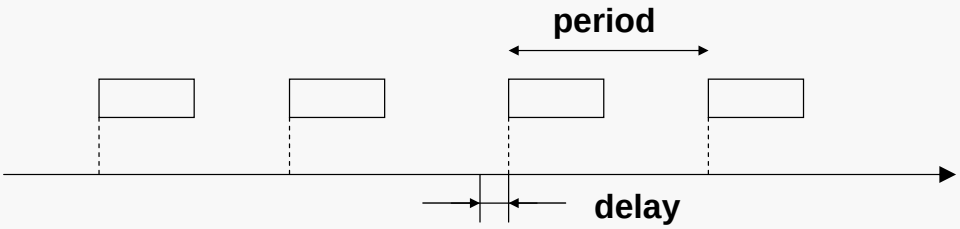
Esecuzione periodica a frequenza fissa (fixed-rate execution)

- **void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)** schedula **task** per l'esecuzione periodica, con periodo **period** (in ms), a partire dall'istante **firstTime**

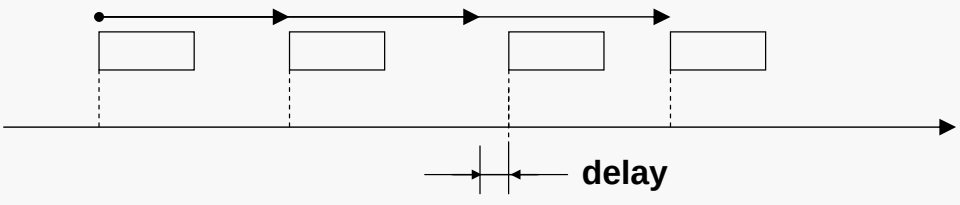
- **void scheduleAtFixedRate(TimerTask task, long delay, long period)** schedula **task** per l'esecuzione periodica, con periodo **period** (in ms), dopo il ritardo **delay**

Fixed-delay vs fixed-rate execution

Fixed-delay execution: ogni esecuzione è schedulata rispetto all'istante effettivo di schedulazione della **precedente** esecuzione

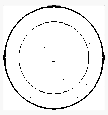


Fixed-rate execution: ogni esecuzione è schedulata rispetto all'istante effettivo di schedulazione della **prima** esecuzione



Fixed-delay vs fixed-rate execution

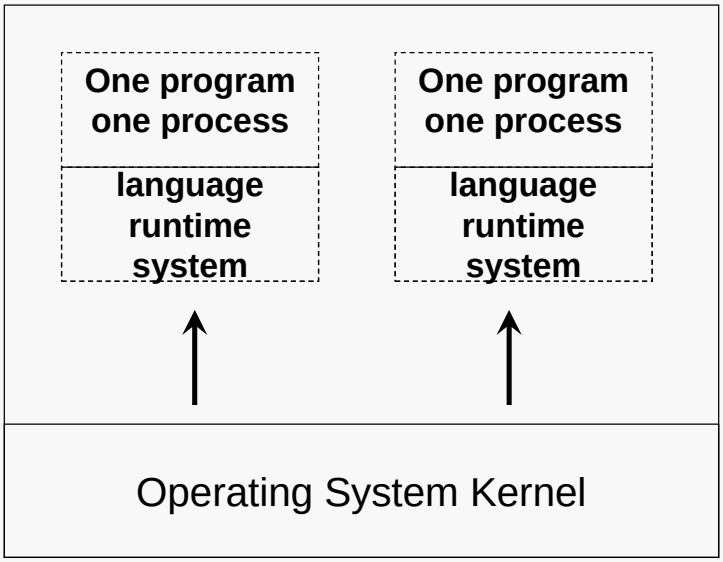
- *fixed-delay execution*
  - a lungo termine la frequenza di scheduling è minore di 1/delay
  - garantisce la frequenza di scheduling a breve termine
  - adatto per animazione
- *fixed-rate execution*
  - a lungo termine la frequenza di scheduling è uguale a 1/delay
  - è adatta per tutte quelle attività ricorrenti che sono sensibili al tempo assoluto (inviare un segnale periodicamente, eseguire manutenzione periodicamente,…)



Thread: Alcuni aspetti architetturali

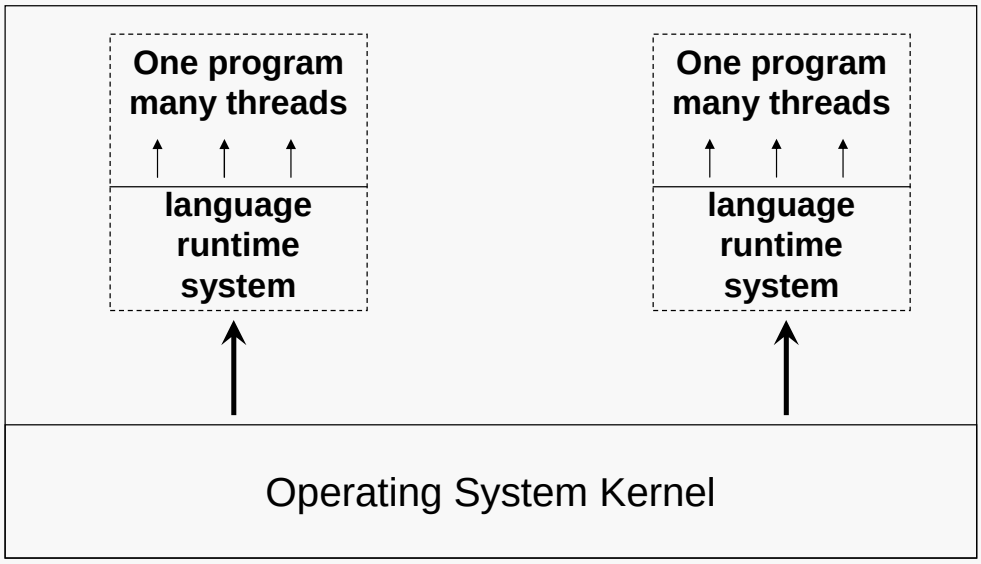
Schemi realizzativi

Modello sequenziale



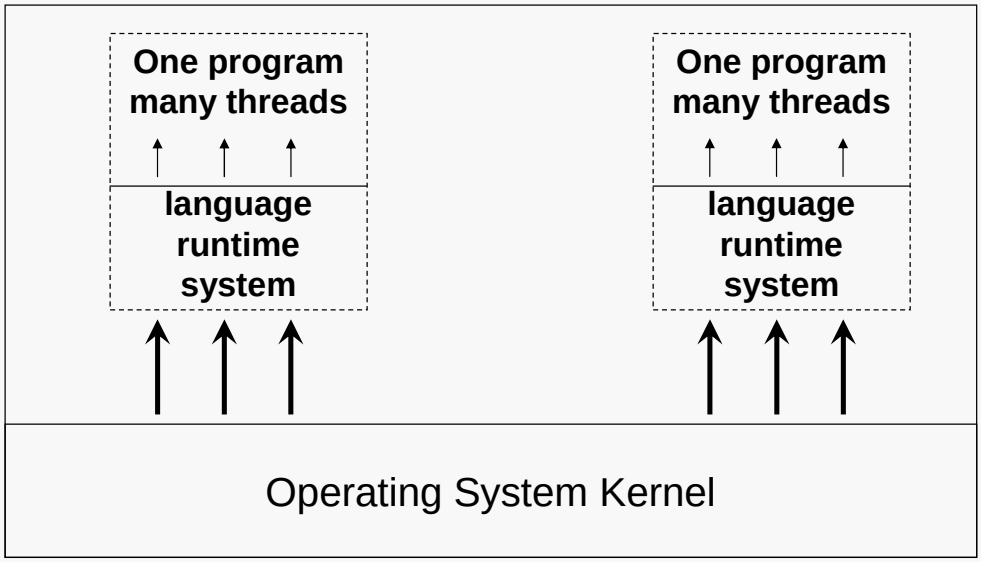
Schemi realizzativi

Linguaggio concorrente senza supporto del SO



Schemi realizzativi

Linguaggio concorrente con supporto del SO



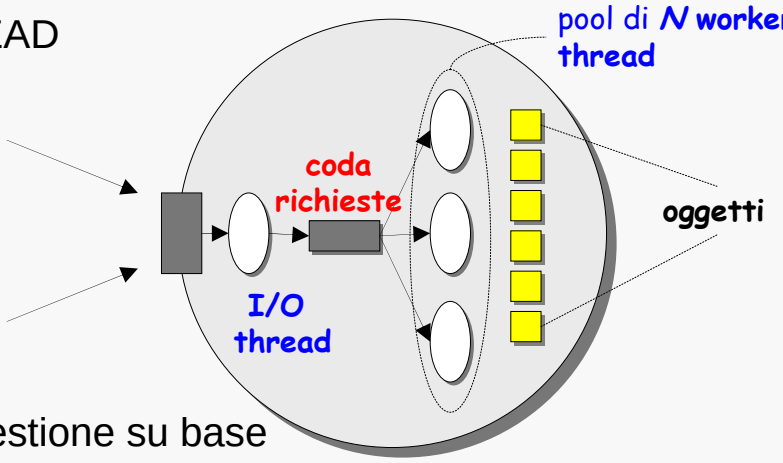
Thread vs Processi

- Sia i thread sia i processi permettono di incrementare il grado di multiprogrammazione
  - permettono di sovrapporre l'elaborazione all'I/O
  - permettono l'esecuzione concorrente sui multiprocessori
- Tuttavia, rispetto ai processi, i thread
  - *facilitano la condivisione delle risorse* (memoria condivisa)
  - *sono più efficienti da gestire*

    la creazione ed il context switch sono più efficienti con i thread che con i processi (≅ rapporto 1:10)

Architetture di server multi-thread

POOL DI THREAD



Vantaggi

- semplicità
- possibile gestione su base prioritaria delle richieste

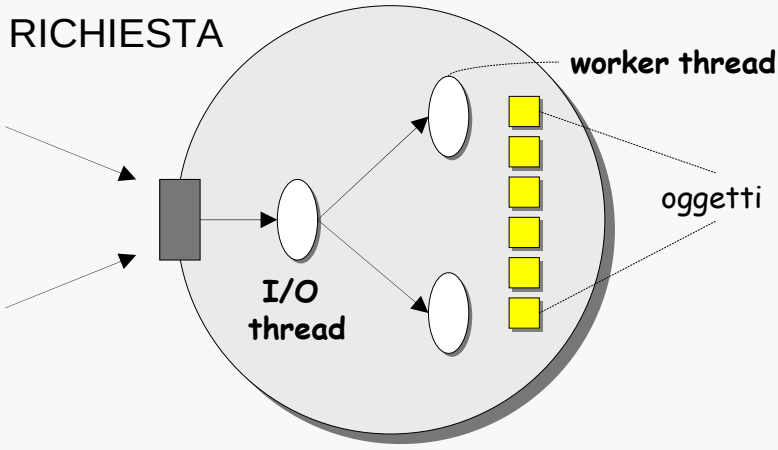
Svantaggi

- limitata flessibilità dovuto al numero *N* fisso dei thread
- overhead dovuto alla *coda richieste*

Architetture di server multi-thread

UN THREAD PER RICHIESTA

- **Si crea un thread per ogni richiesta.**
- **Si distrugge il thread non appena ha servito la richiesta**



Vantaggi

- non c'è l'overhead della coda delle richieste
- il numero di thread non è limitato ed è auto-regolato

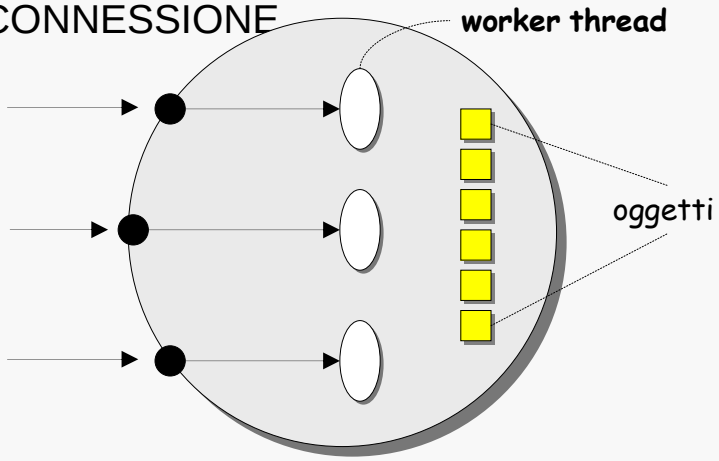
Svantaggi

- overhead dovuto alla creazione ed alla distruzione dinamica dei thread

Architetture di server multi-thread

UN THREAD PER CONNESSIONE

Si crea un worker thread quando il cliente stabilisce una connessione e lo si distrugge quando la connessione viene chiusa



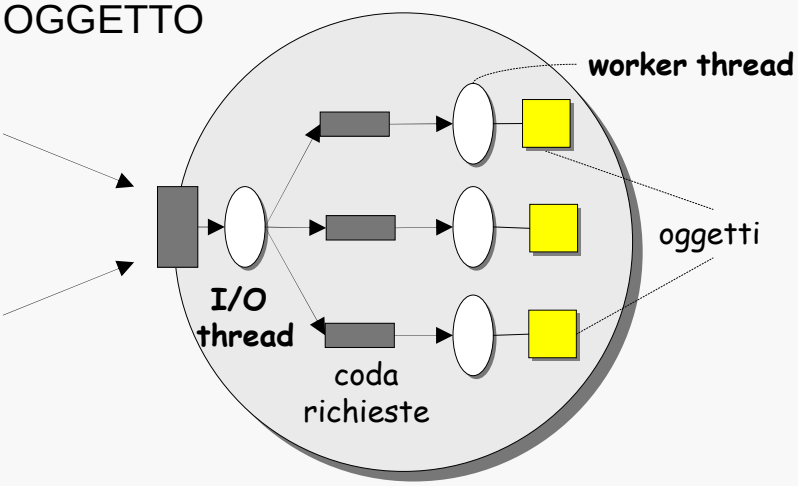
Su una connessione possono arrivare richieste per qualunque oggetto

- *Vantaggi*: si riduce l'overhead della creazione/distruzione dinamica dei thread;
- *Svantaggi*: un thread può avere richieste pendenti mentre altri thread sono inattivi

Architetture di server multi-thread

UN THREAD PER OGGETTO

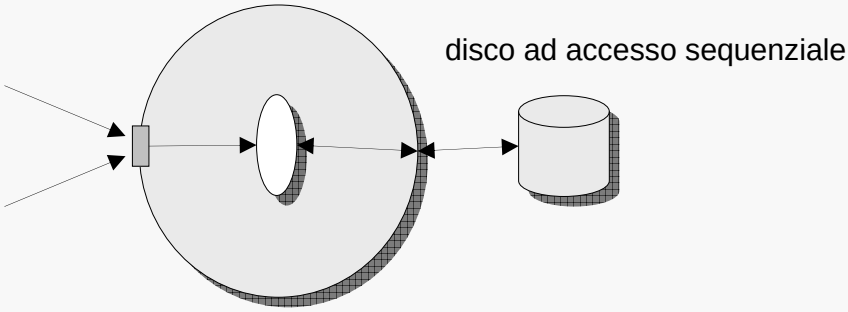
- Si crea un worker thread per ciascun oggetto.
- L'I/O thread inoltra le richieste agli worker thread



- *Vantaggi*: elimina l'overhead della creazione/distruzione dinamica dei thread
- *Svantaggi*: un thread può avere richieste pendenti mentre altri thread sono inattivi

Prestazioni: thread

server monoelaboratore



Tempo di calcolo T_c (es., 2 ms);

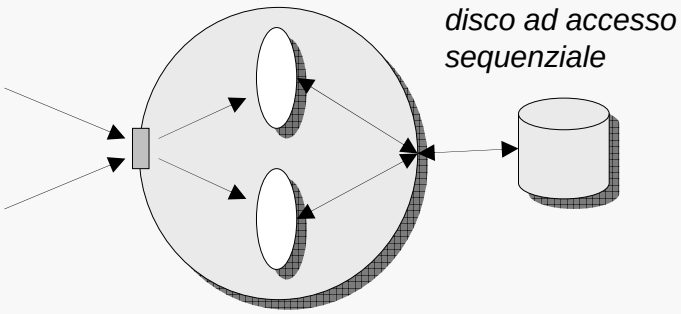
Tempo di accesso al disco T_a (es., 8 ms)

Elapsed Time $T_e = T_c + T_a = 2 + 8 = 10$ ms

Throughput $\mathcal{T}_S = 1/T_e = 100$ richieste/s

Prestazioni: thread

server monoelaboratore

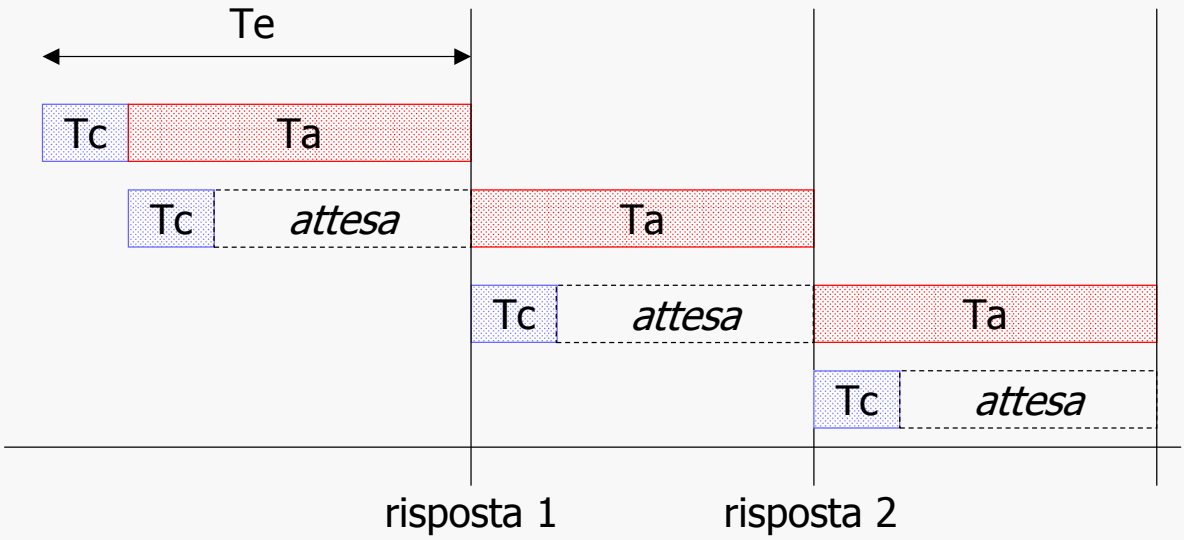


Ipotesi: si trascura ogni overhead legato alla gestione dei thread (creazione, distruzione, scheduling,…)

Throughput $\mathcal{T}_M = 1/T_a = 125$ richieste/s

Prestazioni: thread

server monoelaboratore



Prestazioni: thread

server monoelaboratore

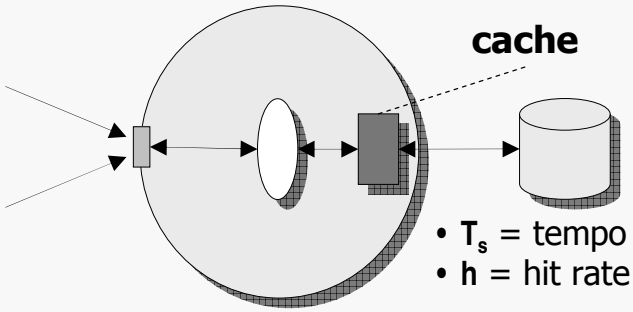
Applicazione della legge di Amdhal

$$\begin{cases} F_e = \dfrac{T_c}{T_e} = \dfrac{2}{10} = 0.2 \\ S_e \to \infty \end{cases} \Rightarrow \boxed{S_o = \dfrac{1}{1-F_e} = \dfrac{1}{0.8} = 1.25}$$

- È la parte sequenziale del programma ($1-F_e$) che fissa le prestazioni
- Non c'è vantaggio (teorico) ad aumentare il grado di parallelismo ($S_e \to \infty$)

Prestazioni: cache

server monoelaboratore



- **T_s** = tempo di ricerca in cache (es., 0.5 ms);
- **h** = hit rate in cache (es., 75%)

$T_e' = T_c + T_s + (1–h)T_a$

$T_{ah} = (1 – h)T_a = 2$ ms

$T_a' = T_s + T_{ah} = 2.5$ ms (tempo medio di accesso al disco)

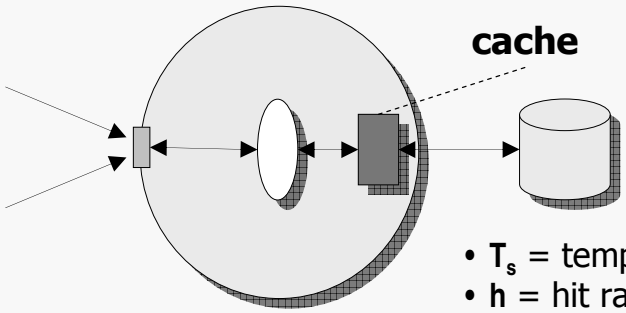
$T_e' = 4.5$ ms

$\mathcal{T} = 1/\ T_e' = 222$ richieste/secondo

$S_0 = T_e' / T_e = 10/4.5 = 2.22$

Prestazioni: cache

server monoelaboratore

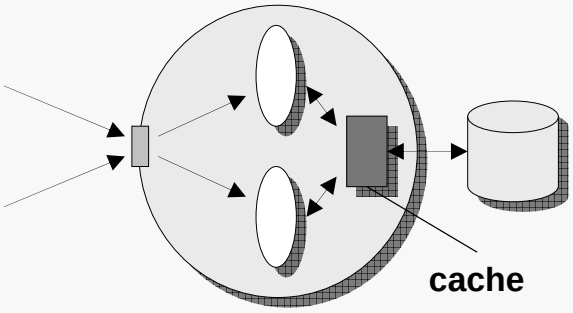


- **T_s** = tempo di ricerca in cache (es., 0.5 ms)
- **h** = hit rate in cache (es., 75%)

$$\begin{cases} F_e = \dfrac{h \times T_a}{T_e} = \dfrac{0.75 \times 8}{10} = 0.6 \\ S_e = \dfrac{h \times T_a}{T_s} = \dfrac{0.75 \times 8}{0.5} = 12 \end{cases} \Rightarrow S_0 = \frac{1}{(1 - F_e) + \dfrac{F_e}{S_e}} = \frac{1}{0.4 + \dfrac{0.6}{12}} = \frac{1}{0.45} = 2.22$$

Prestazioni: cache e thread

server monoelaboratore



h = hit rate in cache (75%)

T_s = tempo di ricerca in cache (0.5 ms)

Throughput

$$\mathcal{T}_{\mathcal{M}} = \frac{1}{T_c + T_s} = \frac{10^3}{2 + 0.5} = 400 \text{ richieste/s}$$

Prestazioni: cache e thread

server monoelaboratore

ipotesi semplificativa: cache e multi-thread sono indipendenti

Speedup$_{cache}$

$$\begin{cases} F_e' = \dfrac{h \times T_a}{T_e} = \dfrac{0.75 \times 8}{10} = 0.6 \\ S_e' = \dfrac{h \times T_a}{T_s} = \dfrac{0.75 \times 8}{0.5} = 12 \end{cases} \Rightarrow S_o' = 2.22$$

Speedup$_{multi}$

$$\begin{cases} F_e'' = \dfrac{T_c}{T_e'} = \dfrac{2}{4.5} \Rightarrow S_0'' = \dfrac{1}{1 - F_e''} = \dfrac{4.5}{2.5} = 1.8 \\ S_e' \to \infty \end{cases}$$

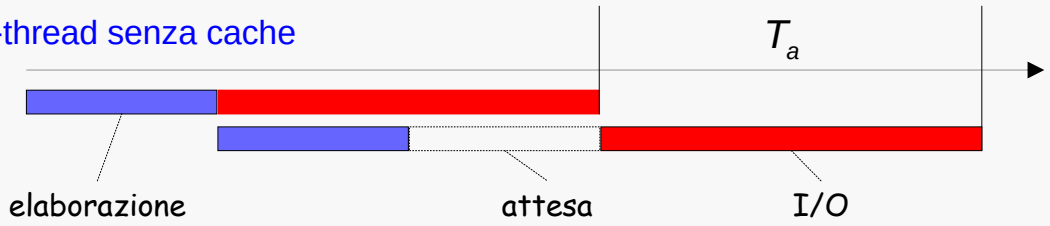
Speedup$_{totale}$

$$S_o = S_o' \times S_0'' = \frac{1}{0.45} \times \frac{4.5}{2.5} = 4$$

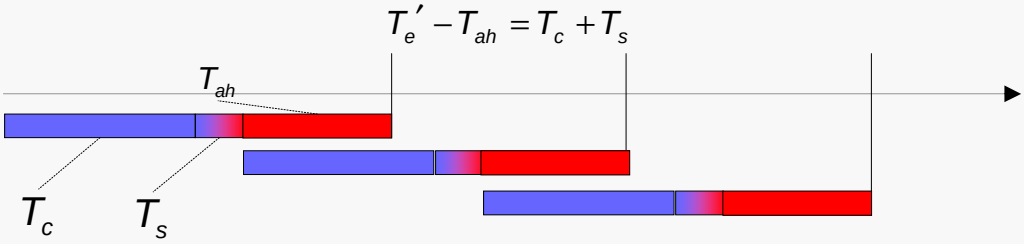
Prestazioni: cache e thread

server monoelaboratore

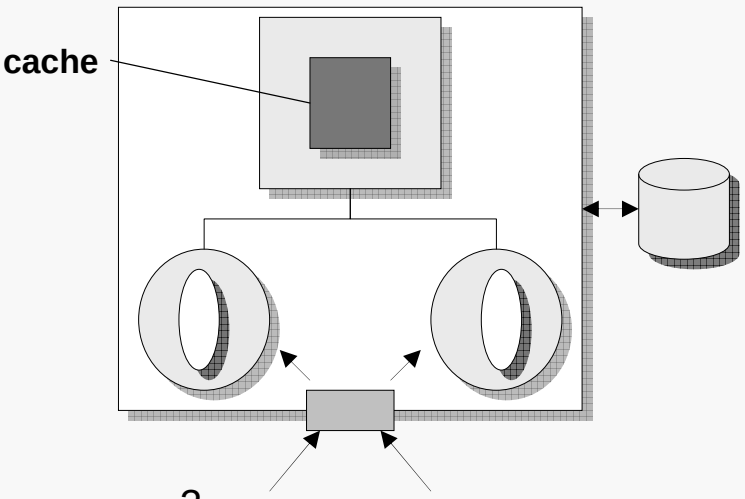
multi-thread senza cache



multi-thread con cache



Cache e thread



server biprocessore

disco ad accesso sequenziale

$$\mathcal{T}_S = \frac{2}{(2T_{ah} + T_s)} = 444 \text{ richieste/s}$$

(due thread, un thread per processore)

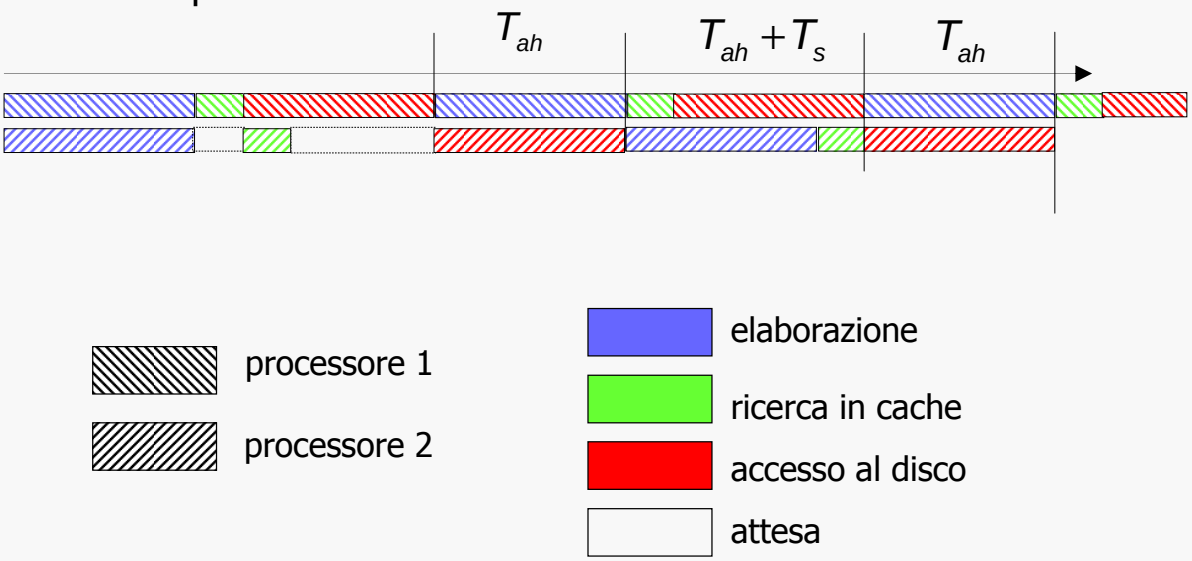
$$\mathcal{T}_M = \frac{1}{T_{ah}} = \frac{10^3}{2} = 500 \text{ richieste/s}$$

(due thread per processore)

Cache e thread

server biprocessore

un thread/processore



Cache e thread

server biprocessore

due thread/processore