



# ***Programmazione concorrente con il linguaggio Java***

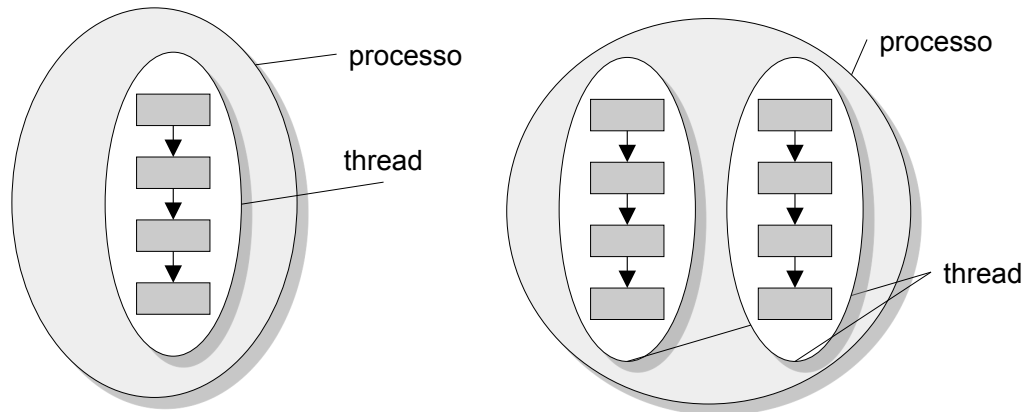
*Thread, Mutua Esclusione e Sincronizzazione*

*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, ad esempio lo spazio di indirizzamento (*modello a memoria condivisa*)

3

## I thread in Java



- Un applicazione Java è costituita da almeno un thread
- All'avvio, un'applicazione parte con il solo *main thread*
- Il *main thread* ha la possibilità di creare altri thread
- Ogni thread è associato ad un oggetto della classe Thread (java.lang)

4

# Definire, creare ed avviare un thread



## Primo metodo: *Sottoclasse Thread*

- Derivare una classe (es. `HelloThread`) da `Thread`
- 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();
    }
}
```

5

# Definire, creare ed avviare un thread



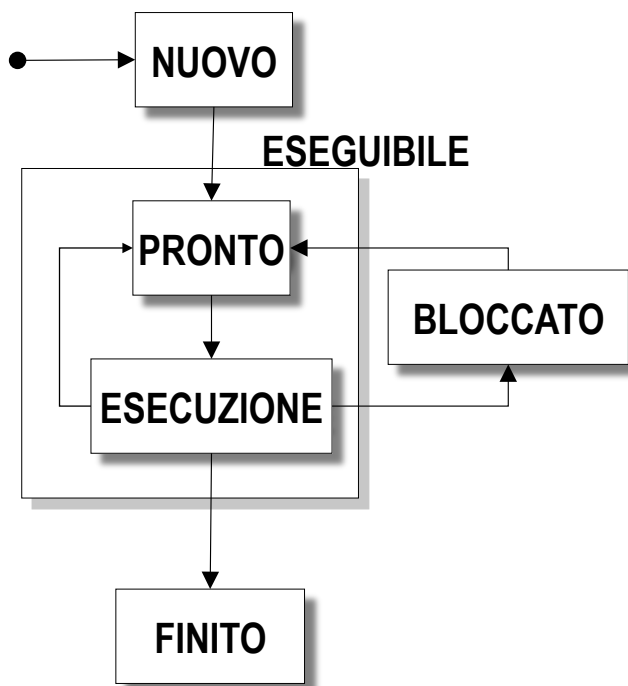
## Secondo metodo: *oggetto Runnable*

- Definire una classe (es. `HelloRun`) che implementa l'interfaccia `Runnable`. N.B. `Runnable` ha il metodo `run()`.
- Creare un oggetto della classe `HelloRun` (*runnable object*)
- Creare un oggetto di classe `Thread` passando il runnable object 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();
    }
}
```

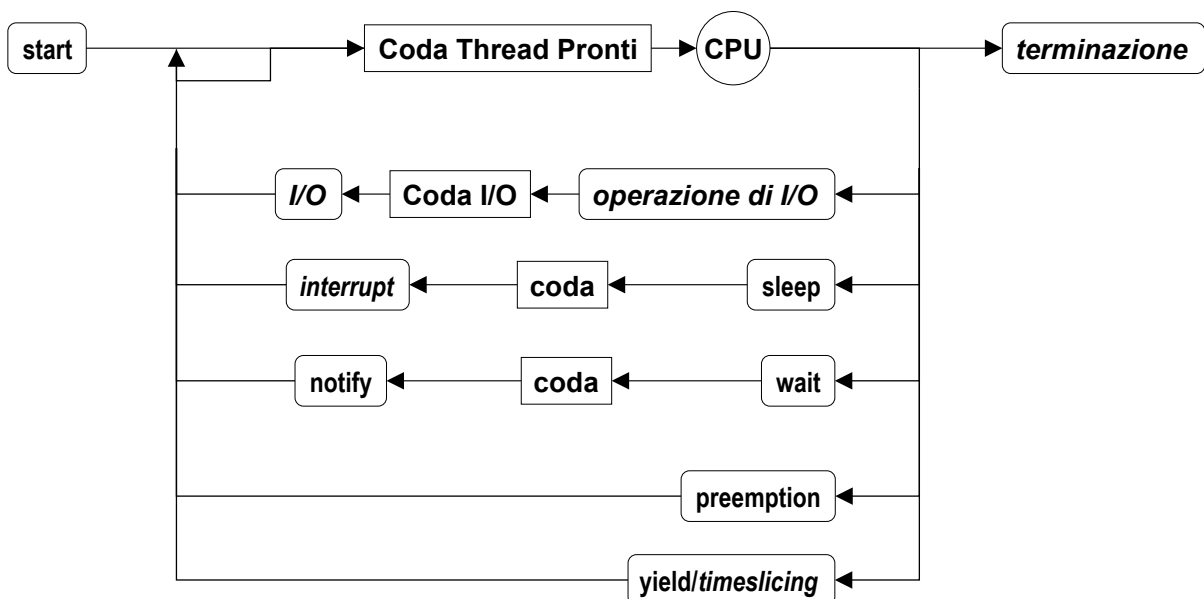
6

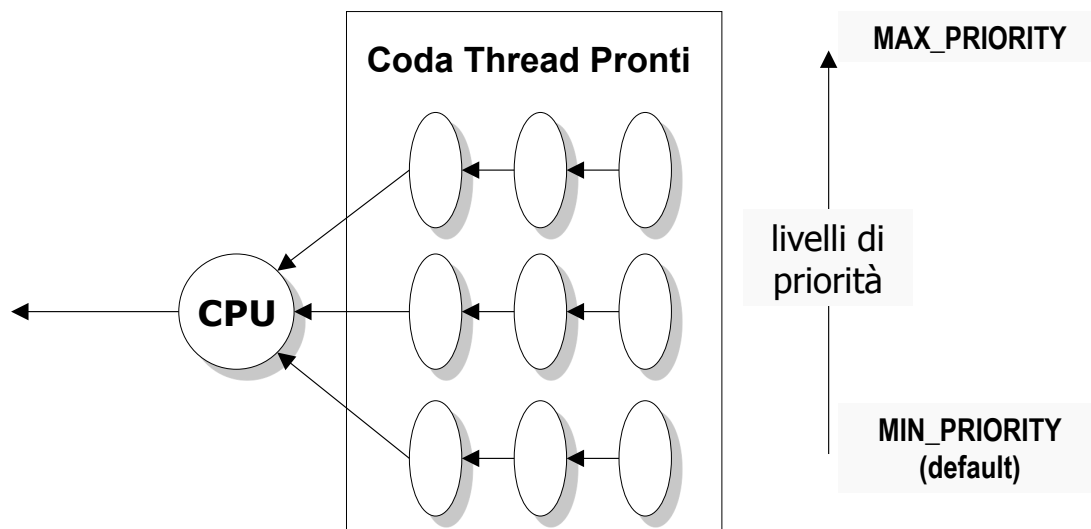
# Stati di un thread



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

# Schema dello scheduling





- Un thread eredita la priorità dal thread che lo ha creato, ma la sua priorità può essere modificata (`setPriority`).

9

## Fixed priority scheduling

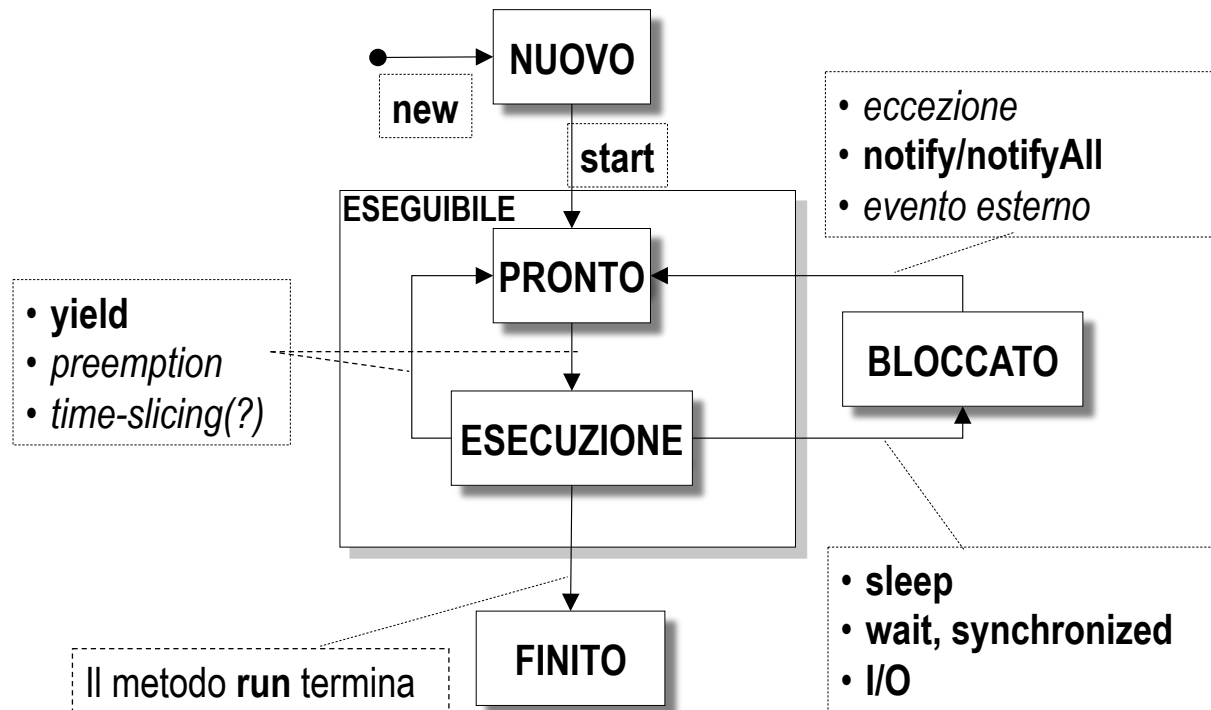


Java implementa **fixed priority scheduling**

- 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.

10

# Transizioni di un thread



11

# Metodi della classe Thread



## run:

- contiene il corpo 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); si può arrivare alla precisione dei `ns`

## 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`

12

## 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
```

13

## 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");
    }
}
```

14

## 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 15:

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

15

## 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.

16





- Un *interrupt* è un segnale inviato ad un thread per fargli interrompere ciò che sta facendo
  - Se un thread è *bloccato* (*wait*, *sleep*, *join*) il suo *interrupt status* viene “pulito” ed il thread riceve **InterruptedException**
  - Se un thread non è bloccato il suo interrupt flag viene “settato”
- Si può lanciare un’interruzione a questo thread per mezzo di *interrupt*.
- Il thread *corrente* testa se è stato interrotto per mezzo di *interrupted (static)*. L’interrupt status viene pulito
- Si può testare se un’interruzione è stata inviata a questo thread per mezzo di *isInterrupted*. L’interrupt status non viene modificato.

## Interrupt – Esempi di gestione



Il thread è bloccato, viene lanciata l’eccezione **InterruptedException**

```
for (int i = 0; i < importantInfo.length; i++) {  
    // Pause for 4 seconds  
    try {  
        Thread.sleep(4000);  
    } catch (InterruptedException e) {  
        // We've been interrupted: no more messages.  
        return;  
    }  
    // Print a message  
    System.out.println(importantInfo[i]);  
}
```

# Interrupts – Esempi di gestione



Una computazione particolarmente lunga periodicamente testa se è stata interrotta

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

In alternativa

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
}
```

19

# Esempio uso interrupt-sleep-join



## SimpleThreads

(da Java™ Tutorials)

20



# Mutua esclusione

## Il problema della mutua esclusione



```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public void value() {
        return c;
    }
}
```

### Traduzione di `increment/decrement`

1. Retrieve the current value of `c`
2. Increment/decrement `c` by 1
3. Store the new value back in `c`

### Possibile *interleaving*

- Thread A invokes `increment`
  - Thread B invokes `decrement`
1. Thread A: retrieve `c` (`c = 0`)
  2. Thread B: retrieve `c` (`c = 0`)
  3. Thread A: increment retrieved value (result 1)
  4. Thread B: decrement retrieved value (result -1)
  5. Thread A: store result in `c` (`c = 1`)
  6. Thread B: store result in `c` (`c = -1`)
- Il risultato di A è andato perduto, sovrascritto da B
  - Questo interleaving è una possibilità
  - Interleaving dipende dalla velocità relativa tra A e B
  - Interleaving è time-variant

# 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 A	Thread B	balance
balance - amt >= 0		80
	balance - amt >= 0	80
balance -= amt		30
	balance -= amt	-20

23

# 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 tra loro**

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*.

24

# Il problema della mutua esclusione



```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized void value() {
        return c;
    }
}
```

25

## Semafori



Un semaforo  $s$  è una variabile intera non negativa ( $s \geq 0$ ) cui è possibile accedere con le operazioni primitive  $p$  e  $v$  così 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 più 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

26



## synchronized Method\_Name (args) Method\_Block

- Java associa un *lock* ad ogni oggetto
- L'esecuzione del metodo sincronizzato **Method\_Name** consiste in:
  1. acquisire **automaticamente** il lock per conto del thread in esecuzione (acquisisce la sezione critica);
  2. eseguire il corpo **Method\_Block** del metodo
  3. rilasciare il lock, anche se l'esecuzione del corpo è terminata da un'eccezione (rilascia la sezione critica);
- Mentre un thread detiene un lock<sup>(\*)</sup>, nessun altro thread può acquisirlo

(\*) *intrinsic lock* o *monitor lock*

# Implementazione concettuale



```
public class SynchronizedCounter {
    private int c = 0;

    public SynchronizedCounter(int aVal) {
        c = aVal;
    }

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized void value() {
        return c;
    }
}
```

- I costruttori sincronizzati non hanno senso
- Solo il thread che crea l'oggetto (*new*) deve avere accesso ad esso mentre lo sta costruendo
- Il riferimento all'oggetto non deve essere rilasciato ad un altro thread prima che la costruzione dell'oggetto sia terminata.



## **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 */ }  
}
```

# Lock rientranti



- In Java i lock sono **rientranti**, cioè un thread può (ri)acquisire un lock che già detiene (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()
```

# Static synchronized method



```
class SynchronizedBankAccount {
    public SynchronizedBankAccount(double aBalance) {
        balance = aBalance;
    }
    public SynchronizedBankAccount() {
        SynchronizedBankAccount(0)
    }
    public synchronized void deposit(double amount) {
        balance += amount;
    }
    public synchronized void withdraw(double amount) {
        balance -= amount;
    }
    public synchronized double getBalance() {
        return balance;
    }
    private double balance = 0;
}
```



# Static synchronized methods



```
class SynchronizedBankAccount {
    public SynchronizedBankAccount(double aBalance) {
        balance = aBalance;
    }
    public SynchronizedBankAccount() { SynchronizedBankAccount(0) }

    private static synchronized int getAccountNumber() {
        return accountCounter++;
    }

    // deposit, withdraw, balance

    private static int accountCounter = 0;
    private final int accountNumber = getAccountNumber();
    private double balance = 0;
}
```

33

# Static synchronized methods



```
class A {
    // other methods
    public static synchronized void p() {
        // body of p()
    }
}
```

```
...
A pippo = new A();
```

```
...
pippo.p();
...
```

- Viene acquisito il lock dell'oggetto **Class** che rappresenta a runtime la classe **A**
  - **Class c = pippo.getClass();**

34



# Sincronizzazione fra threads

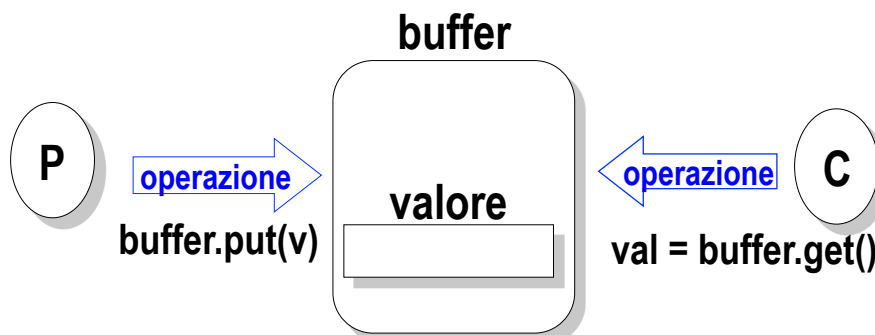
## Problema della sincronizzazione



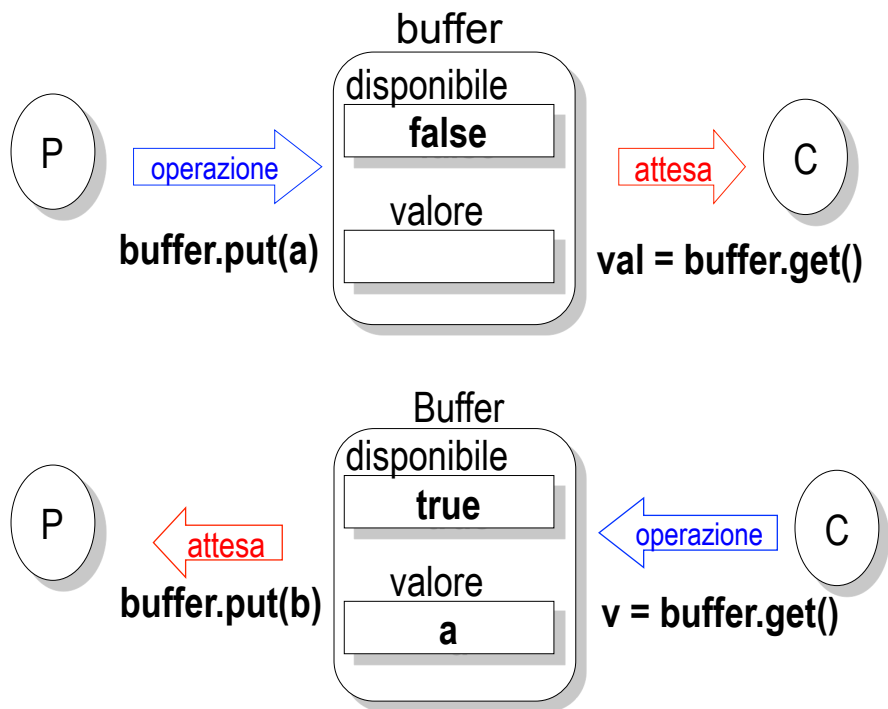
Poiché l' esecuzione di un thread è indipendente, ci possono essere errori di inconsistenza nonostante sia garantita la mutua esclusione sugli oggetti condivisi. Occorre garantire *sincronizzazione* fra i threads.

Esempio: Problema del Produttore-Consumatore.

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



# Produttore-Consumatore



37

## 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. }
```

38

# 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

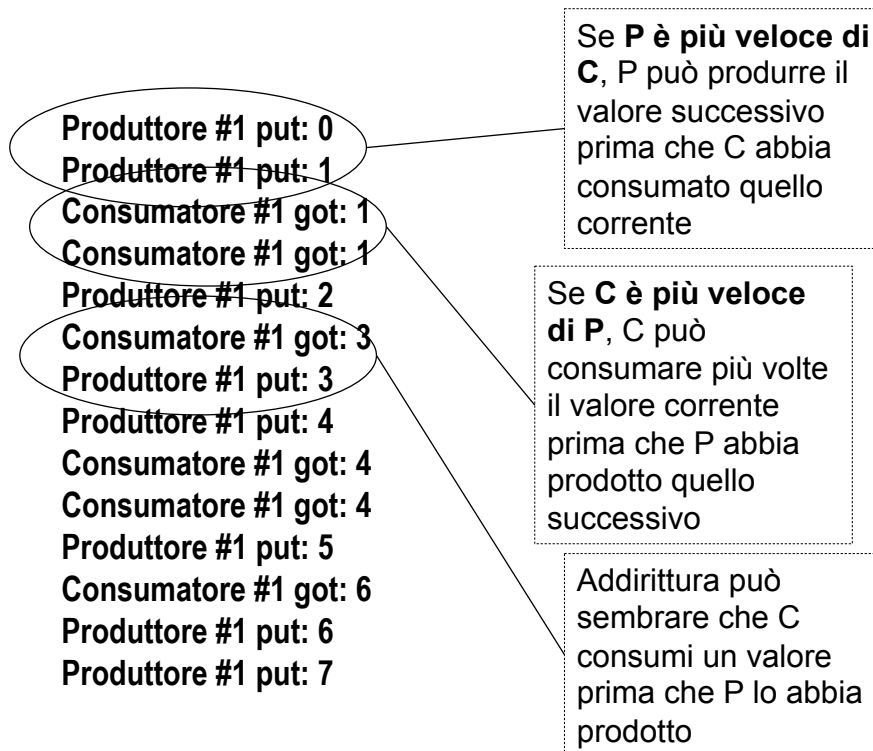


```
1. public class Consumatore extends Thread {
2.     private Buffer buffer;
3.     private int number;

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: Esecuzione



41

# 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

42

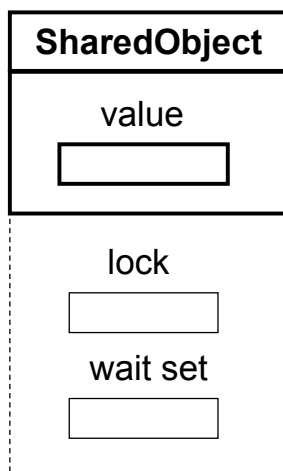
## Soluzione (corretta): il buffer



```
public class Buffer {
    private int valore;
    private boolean disponibile = false;
    public synchronized 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();
    }
}
```

43

## 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**

44



### 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:

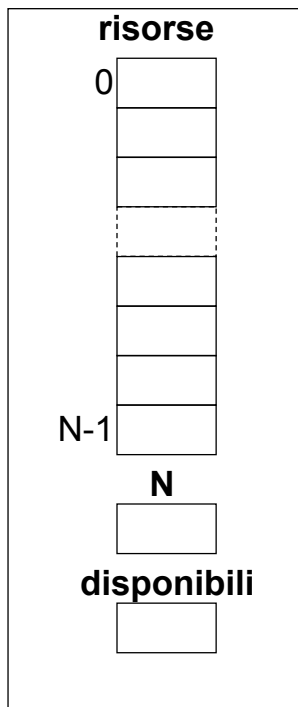
```
while (condcount > 0) {
    v(condsem);
    condcount--;
}
```

## 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

47

## 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**

48



## 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();
    }
}
```

49

## 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);
        }
    }
}
```

50

## 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
```

51

## Implementazione: classe Gestore

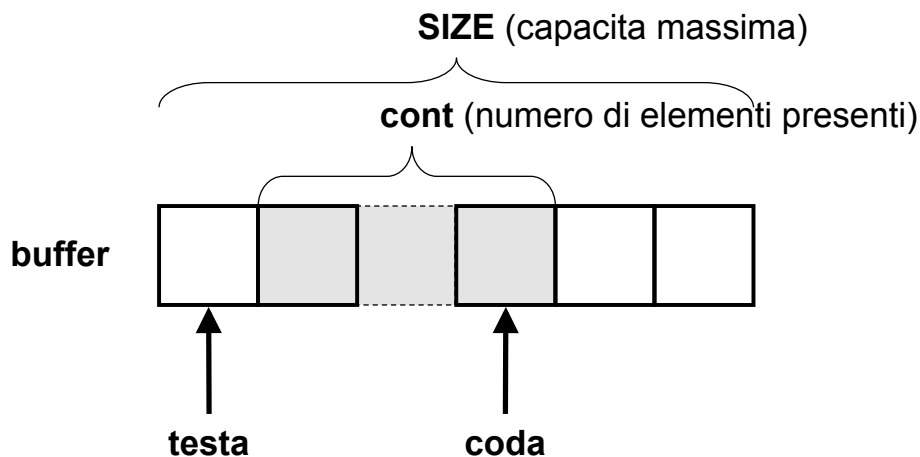


```
synchronized int richiesta() {
    if (disponibili <= 0)
        try { wait(); } catch(InterruptedException e){}
    int i = 0;
    while ((i < N) && !risorse[i]) i++;
    risorse[i] = false;
    disponibili--;
    System.out.println("Alloco risorsa " + i);
    return i;
}

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

52

# Buffer di lunghezza limitata



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

53

## 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

54

## 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){}
        }
    }
}
```

55

## 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){}
        }
    }
}
```

56

## 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();
    }
}
```

57

## 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 = 0;
    }

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

58

## 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;
}
}
```

59

## 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-safe basata su quella specificata come argomento
- **Linea 11.** Notare l'if: sono bloccati solo i thread serventi i consumatori e se ne deve sbloccare solo uno alla volta.

60

## 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

61

## 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
```

62

## Produttori e Consumatori (socket, buffer illimitato)

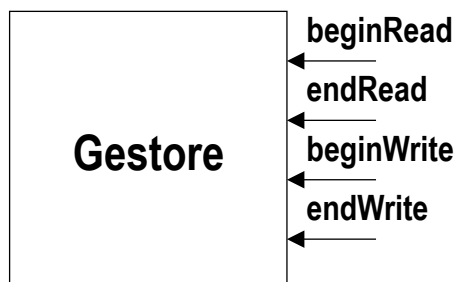


```
18. public void run() {
19.   try {
20.     Message m = (Message)in.readObject();
21.     if (m.getType() == 0) // produttore
22.       mq.add(m);
23.     else { // consumatore
24.       Message app = mq.remove();
25.       out.writeObject(app);
26.     }
27.   } catch(IOException e){
28.   } catch(ClassNotFoundException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

63

## Lettori e scrittori



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

### Protocollo

// Lettore

...

rw.beginRead()

*lettura*

rw.endRead()

...

// Scrittore

...

rw.beginWrite()

*scrittura*

rw.endWrite()

...

64



## 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

---

65

## Implementazione dei Lettori e scrittori

---



Classi:

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

---

66

## 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();
        }
    }
}
```

67

## 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();
        }
    }
}
```

68

## 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");
    }
}
```

69

## Implementazione: classe *Gestore* (starvation lettori)

```
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
```

70

## Implementazione: classe Gestore (starvation lettori)

```
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**

71

## Implementazione: classe Gestore (senza starvation)

```
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() {}
}
```

- Un lettore che ha iniziato ma non terminato **beginRead** è **attivo**
- Un lettore che ha terminato **beginRead** ma che non ha ancora eseguito **endRead** è **running**.
- Uno scrittore che ha eseguito **beginWrite** ma non **endWrite** è **attivo**
- La variabile **okWriter** (**okReader**) permette di far procedere uno scrittore (un treno di lettori) anche in presenza di lettori (scrittori) attivi

72

## Implementazione: classe `ReadWrite` (corretta)



```
1. synchronized void beginRead() {
2.     ar++;
3.     while ((aw > 0) && okWriter)
4.         try {wait();
5.         } catch(InterruptedException e){}
6.     ar--;
7.     rr++;
8. }
9. synchronized void endRead() {
10.    rr--;
11.    if (aw > 0) {
12.        okWriter = true;
13.        okReader = false;
14.    }
15.    if (rr == 0) notifyAll();
16. }
```

- **Linea 3-5.** Un nuovo lettore deve attendere se ci sono degli scrittori attivi in attesa (`aw > 0`)
- **Linea 11-14.** Un lettore passa il turno di accesso agli scrittori attivi in attesa (`aw > 0`).

73

## Implementazione: classe `ReadWrite` (corretta)



```
1. synchronized void beginWrite() {
2.     aw++;
3.     while (busy || (rr > 0) || ((ar > 0) && okReader))
4.         try {
5.             wait();
6.         } catch(InterruptedException e){}
7.     busy = true;
8. }
9. synchronized void endWrite(){
10.    busy = false;
11.    aw--;
12.    if (ar > 0) {
13.        okReader = true;
14.        okWriter = false;
15.    }
16.    notifyAll();
17. }
18. } // Gestore
```

- **Linea 3-6.** Un nuovo scrittore deve attendere se ci sono dei lettori attivi in attesa (`ar > 0`)
- **Linea 11-14.** Uno scrittore passa il turno di accesso ai lettori attivi in attesa (`ar > 0`).

74



Questa soluzione è “complicata” per vari motivi:

1. 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;
2. *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)
3. 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



```
1. public class Semaphore {
2.     private int s;
3.     public Semaphore(int v) {
4.         s = v;
5.     }
6.     // a mutex by default
7.     public Semaphore() { this(1); }
8.
9.     synchronized public void p() {
10.        while (s <= 0)
11.            try{
12.                wait();
13.            } catch(InterruptedException e){}
14.        s--;
15.    }
16.
17.    synchronized public void v() {
18.        s++;
19.        notifyAll();
20.    }
21. }
```

77

## Implementazione: Classe Gestore (semafori privati)



```
1. public class Gestore {
2.     private int ar = 0; // active readers
3.     private boolean aw = false; // active writer
4.     private int br = 0; // blocked readers
5.     private int bw = 0; // blocked writers
6.     private Semaphore rs = new Semaphore(0); // priv. sem.
7.     private Semaphore ws = new Semaphore(0); // priv. sem.
8.
9.     // continua
```

- **Linea 6-7.** I Semaphore rs e ws permettono di definire due diversi punti di accodamento: uno per i lettori ed uno per gli scrittori

78

## Implementazione: Classe Gestore (semafori privati)



```
1. public void beginRead() {
2.     synchronized(this) {
3.         if (!aw && (bw == 0)) {
4.             rs.v(); // this avoids to block at line 9
5.             ar++;
6.         }
7.         else br++;
8.     }
9.     rs.p(); // potentially blocking; outside of mutex
10. }
11. synchronized public void endRead() {
12.     ar--;
13.     if ( ar == 0 && bw > 0) {
14.         aw = true;
15.         bw--;
16.         ws.v(); // awake writers
17.     }
18. }
```

79

## Implementazione: Classe Gestore (semafori privati)



```
public void beginWrite() {
    synchronized(this) {
        if (ar == 0 && !aw) {
            ws.v(); // idem
            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
```

80





# 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 un timer thread estendendo una sottoclasse di **TimerTask** e 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
```

83

## 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()**

84

# 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 **System.exit()**
  - tutti gli *user-thread* terminano la loro esecuzione (ritornano dal metodo **run()**, oppure lanciano un'eccezione)

85

# 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**)

86

## Timer task: *ReminderBeep*



```
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

87

## 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

88

## 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)

---

89

## 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")

---

90

## 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' 1'ora");
                numVolte--;
            } else {
                System.out.println("E' 1'ora");
                timer.cancel();
            }
        }
    } // run
} // RemindTask
// main
```

91

## Metodo *Timer.schedule*



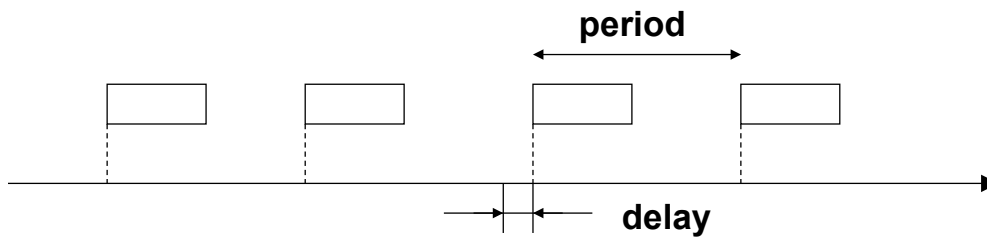
### Esecuzione periodica a frequenza fissa (fixed-rate execution)

- **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**

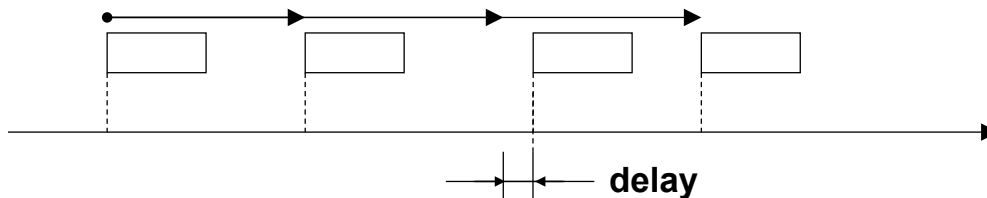
## Fixed-delay vs fixed-rate execution



**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



93

## Fixed-delay vs fixed-rate execution



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

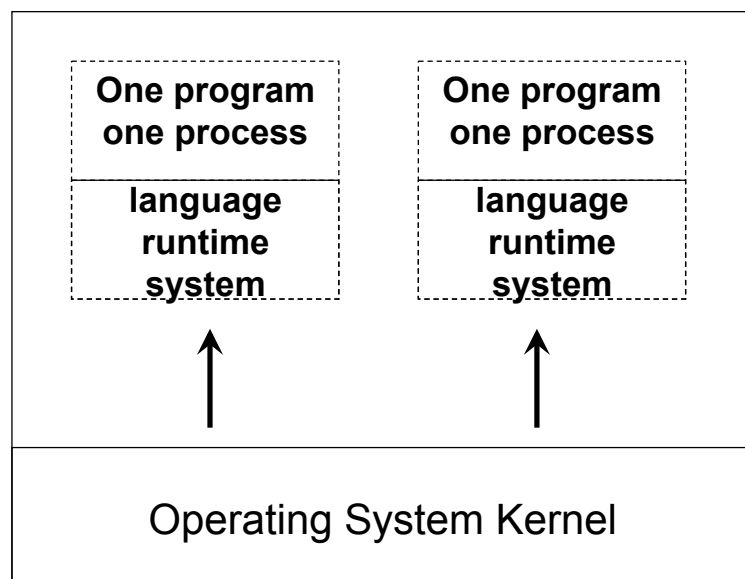


# ***Thread: Alcuni aspetti architetturali***

## ***Schemi realizzativi***



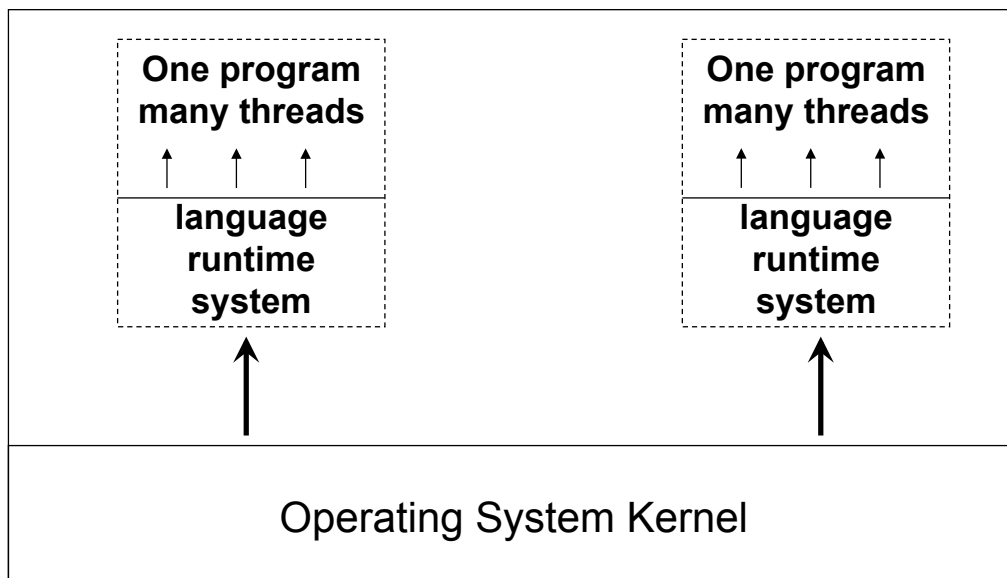
Modello sequenziale



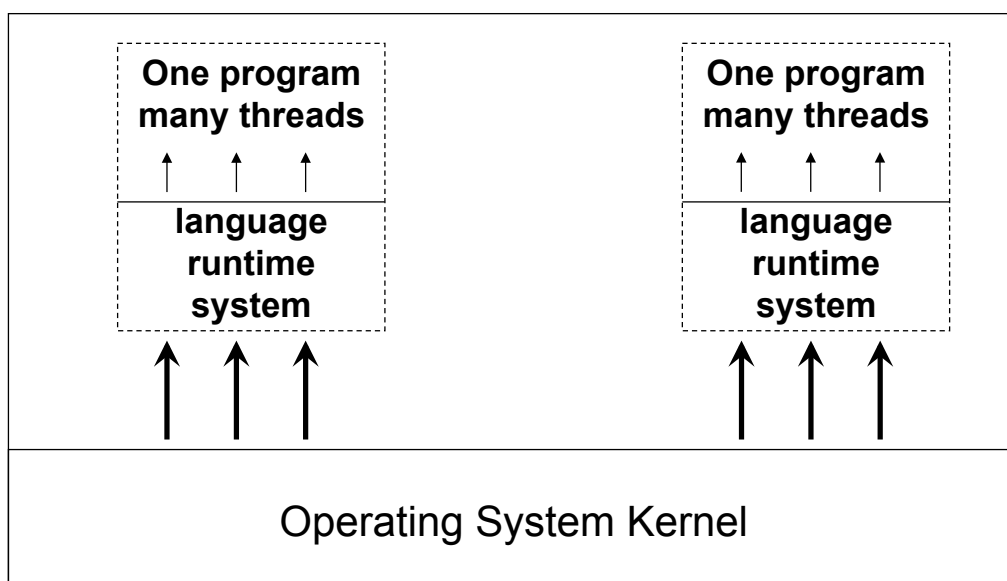




## Linguaggio concorrente senza supporto del SO



## Linguaggio concorrente con supporto del SO



# Thread vs Processi



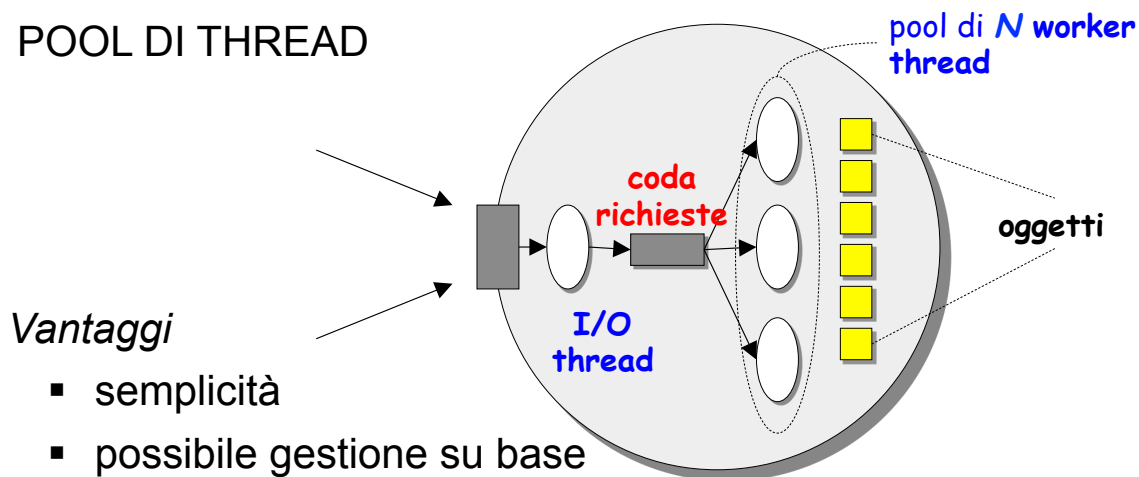
- Sia i thread sia i processi permettono di incrementare il grado di multiprogrammazione
  - permettono di sovrapporre l'elaborazione all'I/O
  - permettono l'esecuzione concorrente sui multiprocessori
- Tuttavia, rispetto ai processi, i thread
  - *facilitano la condivisione delle risorse* (memoria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

99

## 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*

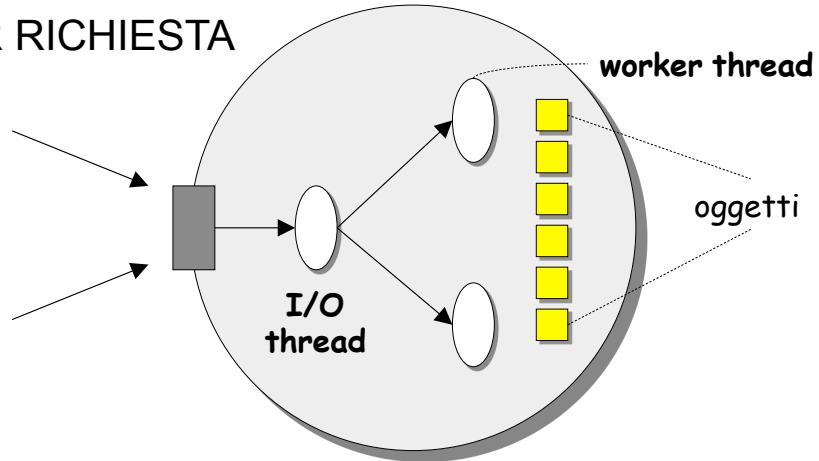
100

# 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

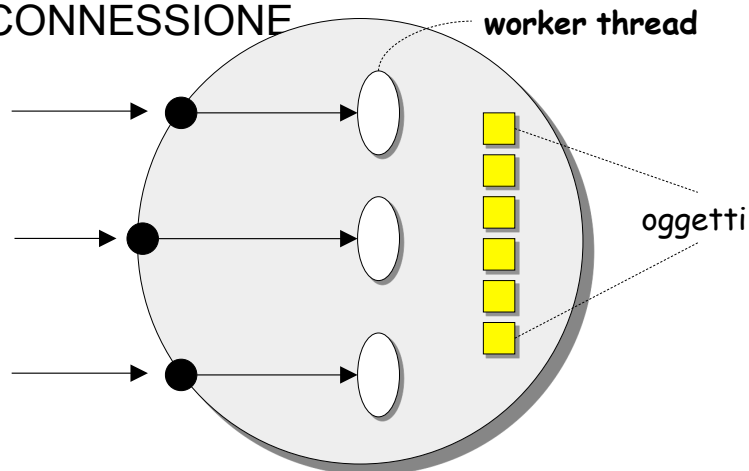
101

# 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

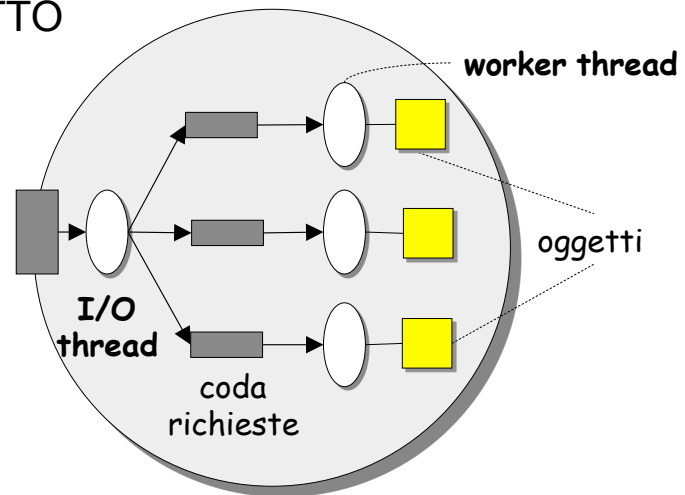
102

# 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



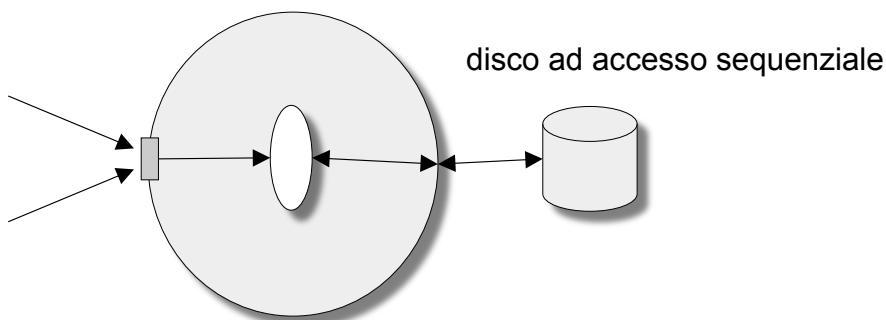
- *Vantaggi*: elimina l' overhead della creazione/distruzione dinamica dei thread
- *Svantaggi*: un thread può avere richieste pendenti mentre altri thread sono inattivi

103

## Prestazioni: thread



### server monoelaboratore



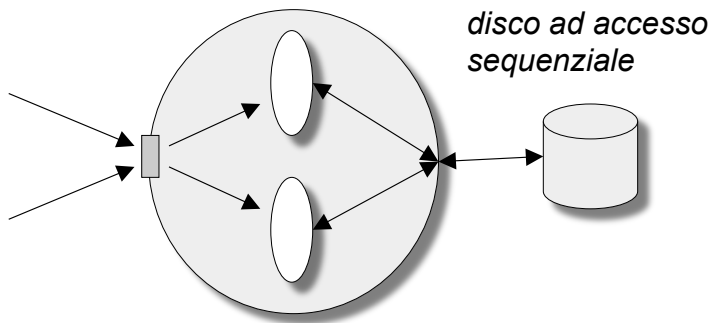
Tempo di calcolo  $T_c$  (es., 2 ms);  
Tempo di accesso al disco  $T_a$  (es., 8 ms)  
Elapsed Time  $T_e = T_c + T_a = 2 + 8 = 10$  ms  
Throughput  $\mathcal{T}_S = 1/T_e = 100$  richieste/s

104

# Prestazioni: thread



## server monoelaboratore



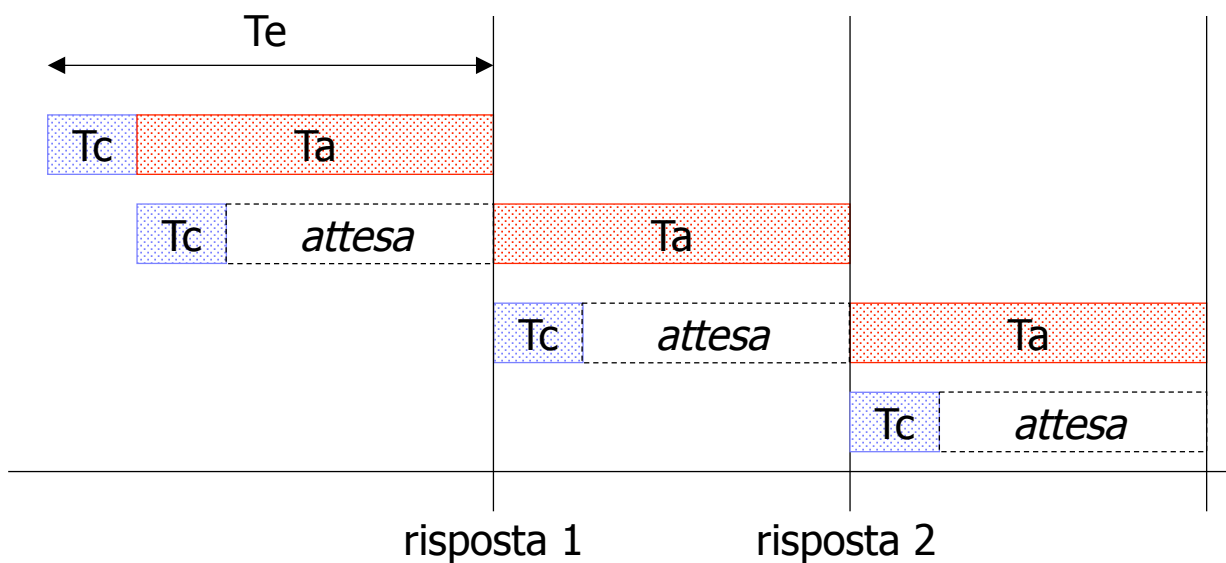
Ipotesi: si trascura ogni overhead legato alla gestione dei thread (creazione, distruzione, scheduling,...)

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

# Prestazioni: thread



## server monoelaboratore



# Prestazioni: thread



## server monoelaboratore

Applicazione della legge di Amdhal

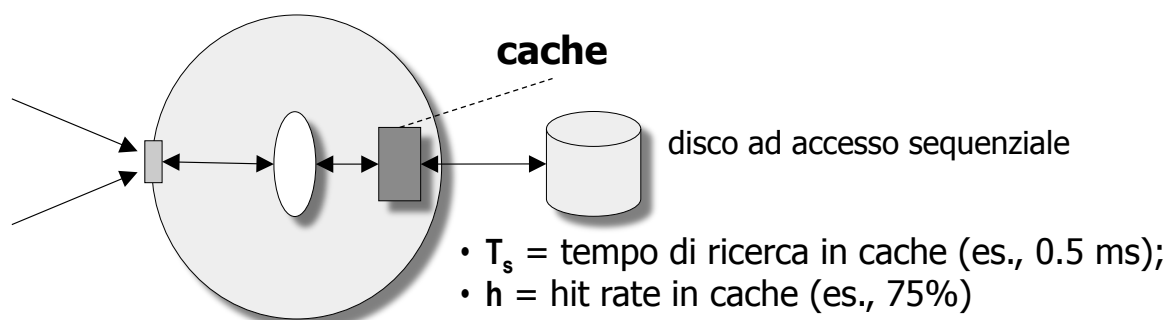
$$\left\{ \begin{array}{l} F_e = \frac{T_c}{T_e} = \frac{2}{10} = 0.2 \\ S_e \rightarrow \infty \end{array} \right. \Rightarrow S_o = \frac{1}{1 - F_e} = \frac{1}{0.8} = 1.25$$

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

# Prestazioni: cache



## server monoelaboratore



$$T_{e[M]} = T_c + T_s + (1-h)T_a$$

$$T_{ah} = (1-h)T_a = 2 \text{ ms}$$

$$T_{a[V]} = T_s + T_{ah} = 2.5 \text{ ms (tempo medio di accesso al disco)}$$

$$T_e[V] = 4.5 \text{ ms}$$

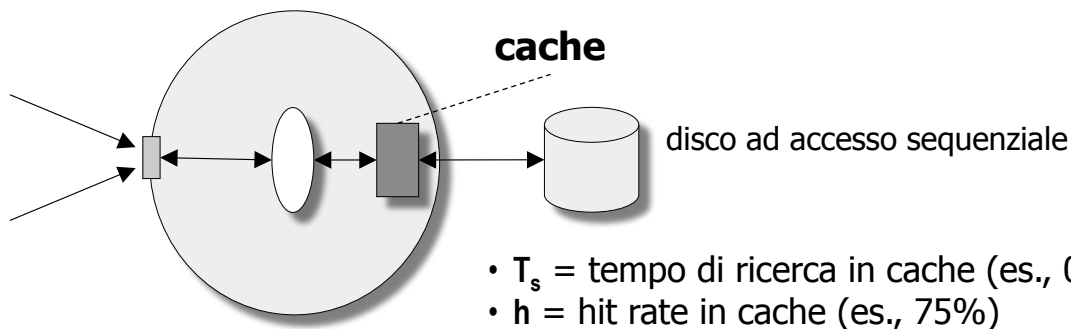
$$T = 1 / T_{e[V]} = 222 \text{ richieste/secondo}$$

$$S_o = T_{e[V]} / T_e = 10 / 4.5 = 2.22$$

# Prestazioni: cache



## server monoelaboratore

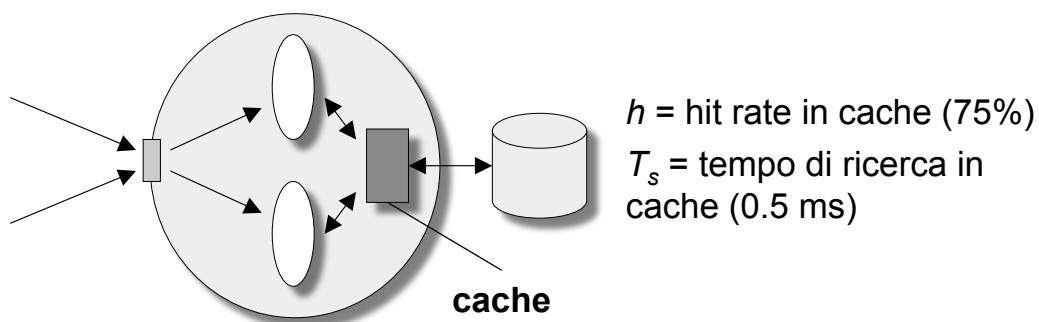


$$\begin{cases} F_e = \frac{h \times T_a}{T_e} = \frac{0.75 \times 8}{10} = 0.6 \\ S_e = \frac{h \times T_a}{T_s} = \frac{0.75 \times 8}{0.5} = 12 \end{cases} \Rightarrow S_0 = \frac{1}{(1 - F_e) + \frac{F_e}{S_e}} = \frac{1}{0.4 + \frac{0.6}{12}} = \frac{1}{0.45} = 2.22$$

# Prestazioni: cache e thread



## server monoelaboratore



Throughput

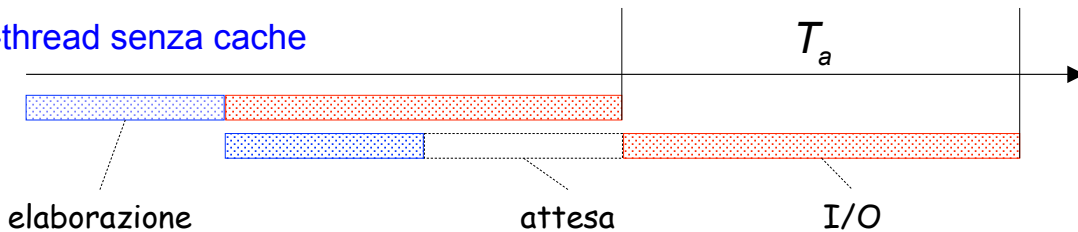
$$T_M = \frac{1}{T_c + T_s} = \frac{10^3}{2 + 0.5} = 400 \text{ richieste/s}$$

# Prestazioni: cache e thread

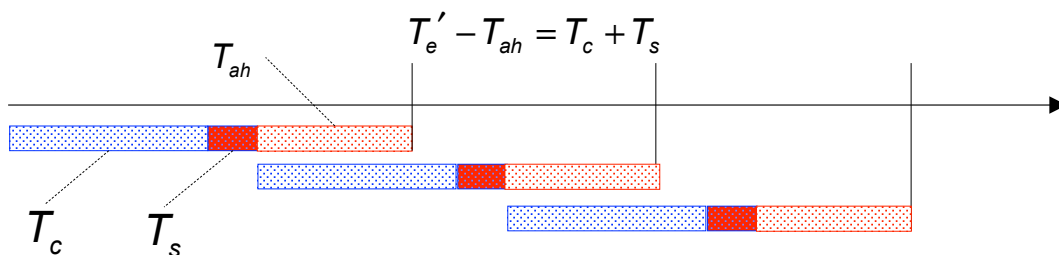


## server monoelaboratore

### multi-thread senza cache



### multi-thread con cache



$T_s$  non è "mascherato" da I/O perché deve essere eseguito in sezione critica

# Prestazioni: cache e thread



## server monoelaboratore

*ipotesi semplificativa: cache e multi-thread sono indipendenti*

$$\text{Speedup}_{\text{cache}} \begin{cases} F_e' = \frac{h \times T_a}{T_e} = \frac{0.75 \times 8}{10} = 0.6 \\ S_e' = \frac{h \times T_a}{T_s} = \frac{0.75 \times 8}{0.5} = 12 \end{cases} \Rightarrow S_0' = 2.22$$

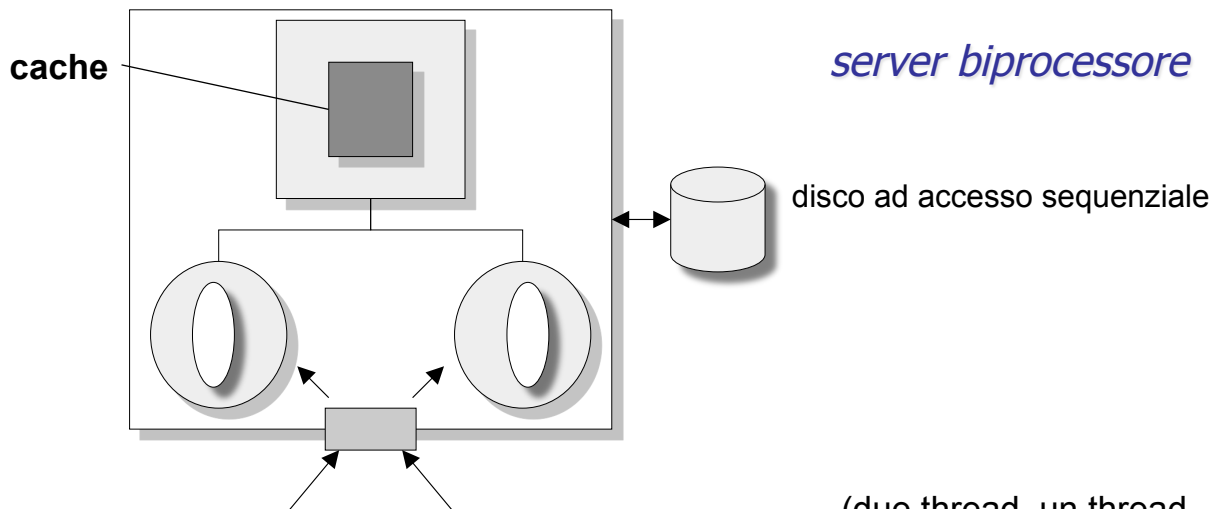
$$\text{Speedup}_{\text{multi}} \begin{cases} F_e'' = \frac{T_c}{T_e'} = \frac{2}{4.5} \Rightarrow S_0'' = \frac{1}{1 - F_e''} = \frac{4.5}{2.5} = 1.8 \\ S_e' \rightarrow \infty \end{cases}$$

$$\text{Speedup}_{\text{totale}} S_0 = S_0' \times S_0'' = \frac{1}{0.45} \times \frac{4.5}{2.5} = 4$$

In  $F_e''$ , si considera  $T_c$  e non  $T_c + T_s$  perché la ricerca in cache è eseguita in mutex



# Cache e thread



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

(due thread, un thread per processore)

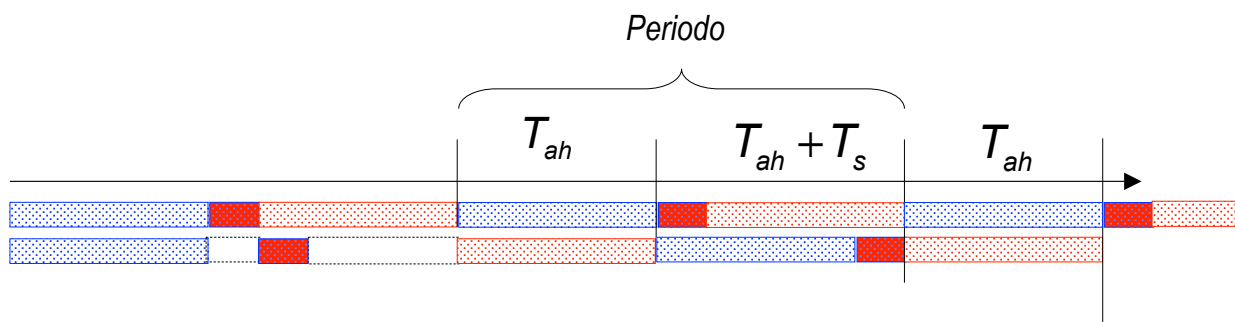
$$T_M = \frac{1}{T_{ah}} = \frac{10^3}{2} = 500 \text{ richieste/s}$$

(due thread per processore)

# Cache e thread



## server biprocessore, un thread per processore

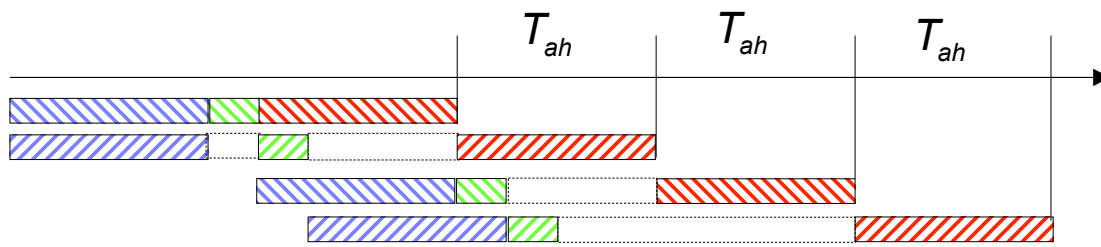


- elaborazione
- ricerca in cache
- accesso al disco
- attesa

# Cache e thread



server bi-processore, due thread per processore



processore 1



processore 2



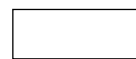
elaborazione



ricerca in cache



accesso al disco



attesa

$$T_s = \frac{1}{T_{ah}} = 500 \text{ richieste/s}$$



## 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 }
```

117

## 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 }
```

118

# 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: 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

```

# Implementazione semplificata

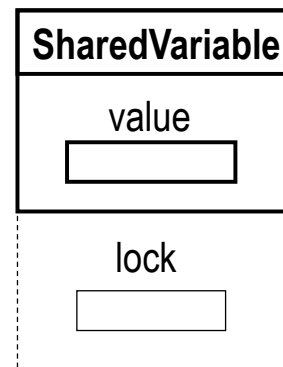


```

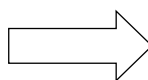
public class SynchronizedVariable {
    private int value = 0;

    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: multi-threaded server



```
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); // conn <-> thread
12.                } catch(IOException e) {
13.                    socket.close();
14.                }
15.            }
16.        } finally {
17.            s.close();
18.        }
19.    }
20. }
```

121

## 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){}
        }
    }
}
```

122