Concorrenza

3.1 Introduzione alla Concorrenza

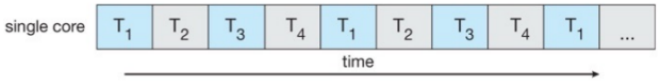
> ⓘ **Definizione di Concorrenza**
>
> Contemporaneità di esecuzione di parti diverse di uno stesso programma (processi, thread, aka agenti). È una caratteristica di primaria importanza nello sviluppo di un software. Capacità di far progredire più di un'attività (processo, thread) nel tempo.
>
> **N.B.:** due agenti possono essere in esecuzione "contemporanea" anche condividendo la stessa CPU.

> ⓘ **Definizione di Parallelismo**
>
> Capacità di eseguire più di un'attività simultaneamente (da esecutori diversi)).

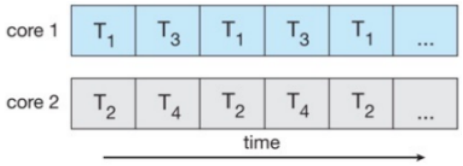
3.1.1 Tipologie di Concorrenza

Esistono due tipi di concorrenza:

1. Senza parallelismo: single cose + multiprogrammazione;



1. Con parallelismo: multicore + multiprogrammazione;



Si tendono a distinguere i seguenti due scenari:

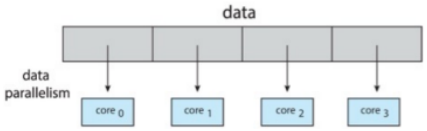
1. **Contemporaneità di esecuzione su una stessa macchina** (con una sola o con molte CPU/Core): possono condividere la stessa memoria e funzioni del SO;
2. **Il programma è in esecuzione su macchine distinte:** collegate da una rete di comunicazione (Programmazione distribuita).

Possibili approcci:

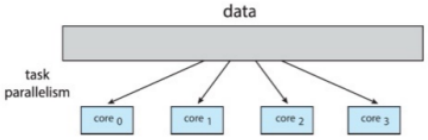
1. **Funzionalità ambiente:** Meccanismi che permettono l'esecuzione e la comunicazione tra gli agenti, soprattutto forniti dal SO.
2. **Funzionalità del linguaggi di programmazione:** Costrutti di linguaggio che espandono la programmazione dal paradigma sequenziale a quello concorrente/distribuito

3.1.2 Parallelismo

Parallelismo dei dati: diversi core eseguono la stessa operazione (eseguono in parallelo attività identiche ma distinte), operando su sottoinsiemi diversi dei dati.



Parallelismo delle attività: diversi core eseguono attività diverse in parallelo, operando su dati comuni.

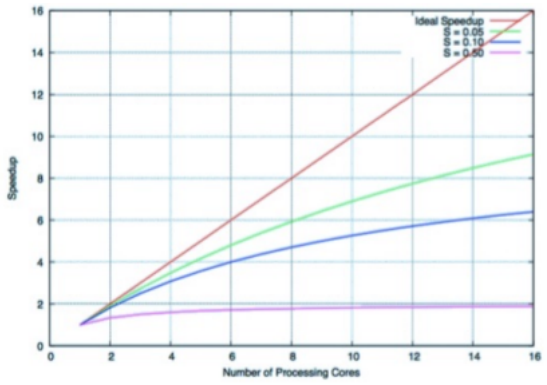


Molto spesso non viene utilizzato solo un tipo di parallelismo, ma entrambi, generando sistemi ibridi.

3.1.3 Legge di Amdahl

Questa legge fornisce il guadagno in termini di performance derivante dall'aggiunta di core ad un'applicazione che ha componenti sia sequenziali che parallele. S è la porzione di applicazione che deve essere realizzata sequenzialmente, e N è il numero di core.

$$incremento\ velocita = \frac{1}{S + \frac{1-S}{N}}$$



Esempio: se il 75% dell'applicazione è parallelizzabile (S=25%):

- *N=2 core: fino a 1,6 volte più veloce del single-core;*
- *N=4 core: fino a 2,28 volte più veloce di single-core.*

3.2 Programmi concorrenti e sequenziali

3.2.1 Programmazione concorrente

> **Definizione Programmazione concorrente**
>
> La pratica di implementare dei programmi che contengano più flussi di esecuzione (Processi o Threads).

Esempio: server web.

Utile per:

- Sfruttare gli attuali processori multi-core;
- Evitare di bloccare l'intera esecuzione di un'applicazione a causa dell'attesa del completamento di un'azione di I/O;

- Strutturare in modo più adeguato un programma (in particolare i programmi che interagiscano con l'ambiente, controllino diverse attività, gestiscano diversi tipi di eventi magari mentre forniscono funzionalità ad un utente umano).

Definizione di Programma sequenziale

Un programma sequenziale (deterministico) eseguito ripetutamente con lo stesso input produce lo stesso risultato ogni volta.

Definizione di Programma concorrente

In un programma concorrente, non è garantito lo stesso risultato ad ogni identico input. Questo perché il comportamento di un thread può dipendere dagli altri thread in maniera non deterministica.

3.2.2 Accesso alle risorse condivise

Interazioni

In un programma concorrente, le **interazioni tra agenti concorrenti** possono essere:

- **Cooperazione**
  - Interazioni prevedibili e desiderate;
  - La loro presenza è necessaria per la logica del programma;
  - Avviene tramite scambio di informazioni (anche semplici come segnali);
  - Sincronizzazione diretta o esplicita.
- **Competizione**
  - Gli agenti competono per accedere ad una risorsa condivisa;
  - Politiche di accesso alla risorsa sono necessarie;
  - Sincronizzazione può essere indiretta o implicita.
- **Interferenze**
  - Interazioni "non prevedibili e non desiderate";
  - Errori di programmazione, spesso dipendenti dalle tempistiche (time dependent) e non facilmente riproducibili.

In caso di accesso concorrente ad una risorsa condivisa è necessario garantire due condizioni per un comportamento coerente:

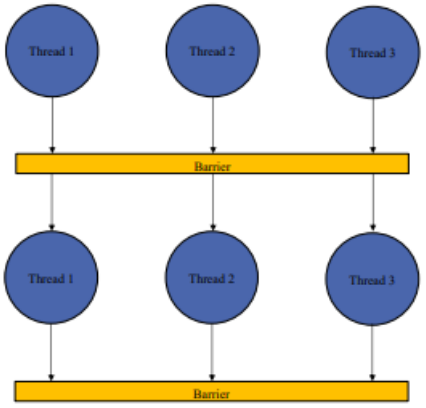
- **Mutua esclusione**: solo un thread esegue una sezione critica alla volta.
- **Visibilità**: le modifiche apportate da un thread ai dati condivisi sono visibili agli altri thread per mantenere la coerenza dei dati.
  Il problema della visibilità viene ovviata dalle variabili *volatile*.

3.2.2.1 Meccanismi di sincronizzazione

Sono i meccanismi che permettono di controllare l'ordine relativo delle varie attività dei processi/thread.

- **Memoria condivisa**
  - **Mutua esclusione**: dati, attraverso la creazione di regioni critiche del codice, non sono accessibili contemporaneamente a più thread;
  - **Sincronizzazione su condizione**: si sospende l'esecuzione di un thread fino al verificarsi di una opportuna condizione sulle risorse condivise.

Barriere: i thread vengono messi progressivamente in attesa che una condizione globale non venga soddisfatta.



Esempio:

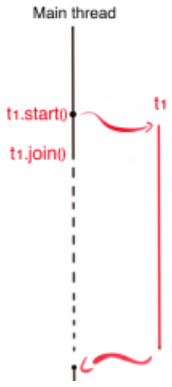
- *Un compito comune suddiviso in più fasi e ogni fase è assegnata a più thread;*
- *Ogni thread esegue il suo task ed attende che tutti gli altri abbiano terminato prima di proseguire alla fase successiva.*

Possibile implementazione:

- **Contatore condiviso** conta il numero di processi che devono terminare:
- Il contatore decrementa ogni volta che un thread termina il suo task;
- Aggiornamento del contatore “atomico”.

- **Scambio di messaggi**
  - Di solito impliciti nelle primitive di send e receive == un thread può ricevere un messaggio solo dopo il suo invio;
  - **Sincronizzazione su eventi**: si sospende l’esecuzione di un thread fino al verificarsi di un evento.
    join(): quando si invoca il metodo join()su un thread, il thread chiamante entra in uno stato di attesa (wait). Rimane in tale stato finché il thread chiamato (t1) non termina.

    Può anche ritornare se il thread chiamato viene interrotto. In questo caso, il metodo lancia una ***InterruptedException****. Infine, se il thread chiamato è già terminato o non è stato avviato, la chiamata al metodo join() ritorna immediatamente.*



Problemi di accesso a risorse condivise: Race Condition

> ⚠ **Race Condition**
>
> «Tutte quelle situazioni in cui thread diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall’ordine in cui essi effettuano le loro operazioni».

È causato dalla competizione nell'accesso ad una risorsa condivisa, come una variabile in memoria. Può causare problemi seri.

Il problema è dovuto al fatto che alcune azioni possono essere non atomiche, ovvero interrompibili dallo scheduler. Lo scheduler può inframmezzare l'esecuzione di thread differenti (thread interleaving) portando la risorsa condivisa in uno stato inconsistente.

⚠ **Interferenza**

Un aggiornamento incorretto di una risorsa dovuto a una combinazione aleatoria di letture e scritture viene detto interferenza

Soluzione: fornire accesso mutuamente esclusivo alla risorsa (logicamente non interrompibile).

3.2.2.2 Implementazione della sincronizzazione

Due possibili approcci:

1. **Attesa attiva (busy waiting o spinning)**:
   - Ha senso solo in sistemi multiprocessore;
   - Non richiede cambio di contesto;
   - Spreca tempo di calcolo sulla CPU;
   - La tecnica consiste nel verificare periodicamente che una certa condizione sia valida.
2. **Sincronizzazione basata sullo scheduler**:
   - (auto-)sospensione del thread;
   - Il thread viene risvegliato quando si verifica l'evento atteso.

   Sincronizzazione basata su Sezione Critica
   La sezione critica è un blocco di codice che può essere eseguito da un solo thread alla volta.

   Obiettivo: realizzare un protocollo di cooperazione tra i processi che faccia sì che, quando un thread è nella sua sezione critica, gli altri non lo siano (Mutua esclusione);

   Protocollo: ogni thread accede ad una risorsa condivisa (accessibile tramite sezione critica) mediante le seguenti operazioni:

   1. **Sezione di ingresso** (richiesta entrata nella sezione critica)
   2. **Sezione critica**
   3. **Eventuale sezione di uscita** (ad esempio, per notificare gli altri processi)

   Per realizzare un protocollo di questo tipo serve un operazione non interrompibile (atomica), il quale verrà utilizzato per realizzare la sezione critica (per mezzo di un lock). Supponendo che B sia una variabile booleana in memoria, l'operazione ritorna il valore originario della cella di memoria puntata e imposta il valore della cella di memoria puntata a **true**.

L'implementazione in Java dipende dalla particolare JVM e dall'ottimizzatore.

Lock

🔥 **Lock**

Variabile logica manipolabile atomicamente.
Quando un thread ha acquisito un lock, gli altri thread che richiedono il lock() si bloccano finchè il thread che lo detiene non lo rilascia.

Caratteristiche

- È una variabile binaria;
- Due metodi lock() e unlock() permettono di acquisire il lock in maniera atomica;
- Per ogni lock deve essere gestita una coda di thread in attesa.

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

public class SharedRes
{
private Lock l = new ReentrantLock();
private int counter = 0;

public void increment(){
        l.lock();
        try { // doing some action
                try {
                        counter++;
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        }
        } finally {
                // Release the permit.
                l.unlock();
                }
        }
}
```

Problemi associati al meccanismo di Locking

Se un thread non riesce ad acquisire un lock viene sospeso.

- **Context switch**, risvegliare un thread presenta un costo;
- Un thread in attesa non può eseguire nessuna operazione;
- Un thread a bassa priorità può bloccare thread che ne hanno una più alta (**priority inversion**), infatti quando un thread acquisisce un lock nessun altro thread che ha bisogno di quel lock può proseguire.

Il processo di locking viene detto ***pessimistico***: infatti se la contesa è non è frequente, nella maggior parte dei casi la richiesta e l'esecuzione di un lock non è necessaria ed aggiunge overhead.

Alternativa al Locking: Optimistic retrying

> **Idea** : è più efficiente riprovare che chiedere il permesso

- Nessuna sincronizzazione in lettura;
- Per eseguire una scrittura
    1. Lettura della variabile (creazione di una copia locale)
    2. Aggiornamento della copia
    3. Scrittura della variabile se non c'è collisione, altrimenti riprovare

Quest'approccio è particolarmente adatto all'aggiornamento delle strutture accessibili per mezzo di un unico indirizzo.

```
tmp = readMem(pos);
tmp = update(tmp)
if Collision(pos) goto 1
else writeMem(pos,tmp)
```

Lock in breve

1. Acquisizione – accesso ai diritti di sincronizzazione
2. Algoritmo di attesa – se il lock non è stato acquisito - spin o sleep in attesa che la sincronizzazione sia disponibile
3. Rilascio – i diritti di accesso vengono rilasciati ed altri thread possono richiederli

Semaforo

Semaforo

Variabile intera che può essere manipolata solo attraverso due operazioni atomiche (non interrompibili): acquire() e release().

acquire() aspetta che la variabile sia maggiore di zero, e quindi la decrementa;
release() incrementa incondizionatamente la variabile;
Per ogni semaforo deve essere gestita una coda di thread in attesa.

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

public class MaxAccessRes {
        private Semaphore sem;
        MaxAccessRes(int size) {
                sem = new Semaphore(size);
        }

        public void access() {
                try {
                        sem.acquire();
                        // doing some action
                        Thread.sleep(1000);
                } catch (InterruptedException exc) {}
                // Release the permit.
                sem.release();
        }
}
```

Metodi Synchronized

Mutua esclusione implicita in oggetti java. Java associa un lock intrinseco/implicito (sistema di controllo dell'accesso automatico – semaforo binario) ad ogni oggetto che abbia almeno un metodo *synchronized*.

Poiché ogni oggetto in Java estende la classe Object, ogni oggetto ha un suo lock:

- 1 oggetto = 1 lock (attivo se c'è almeno un metodo *synchronized*);
- L'oggetto con il metodo *synchronized* è la risorsa condivisa tra i thread;
- Non si può usare *synchronized* su un costruttore né su singoli campi.

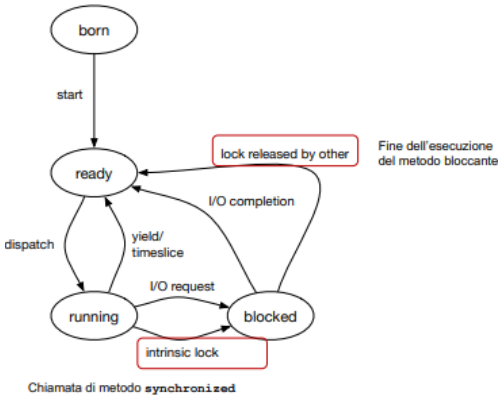
Il fatto che un thread T1 sia in esecuzione all'interno di un metodo (o blocco) *synchronized* fa sì che altri thread che richiedano l'esecuzione dello stesso o un altro metodo (o blocco) *synchronized* vengano messi in attesa che T1 completi l'esecuzione del metodo.

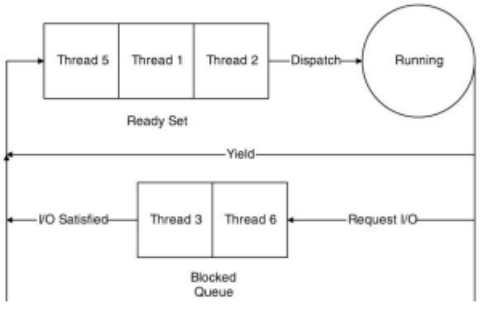
Quando un metodo *synchronized* viene invocato:

- Se l'oggetto non è bloccato (locked) (ossia se nessun thread sta attualmente eseguendo un metodo *synchronized* sull'oggetto), l'oggetto viene bloccato e quindi il metodo è eseguito;
- Se l'oggetto è bloccato, il thread chiamante viene sospeso fino a quando quello "bloccante" non rilascia il lock.
  NB: un metodo non synchronized non viene mai bloccato.

```
[…]
// ESEMPIO
public synchronized void deposito(float cifra){
        saldo += cifra;
}

public synchronized void prelievo(float cifra){
        if(saldo > cifra){
        saldo -= cifra;
        return cifra;
        }
        float prelevati = saldo;
        saldo = 0.0f;
}
[…]
```

Diagramma degli stati di un thread





Lo **scheduler** inserisce nella coda dei thread bloccati tutti i thread in attesa del completamento di un'operazione di I/O e quelli in attesa che si liberi l'accesso ad una risorsa condivisa (lock).

Commenti

- L'introduzione della parola chiave **synchronized** nella dichiarazione dei metodi di una risorsa condivisa introduce (forza) un ordinamento nell'esecuzione di questi metodi da parte di thread concorrenti.
- Quando un metodo synchronized termina, si stabilisce automaticamente una relazione "happens-before" con ogni invocazione successiva di altri metodi synchronized sullo stesso oggetto.
- Ai fini delle problematiche di interferenza, è come se questi metodi fossero stati resi atomici.
- L'acquisizione e il rilascio dell'intrinsic lock sono completamente automatiche, gestite dalla JVM, "trasparenti" al programmatore.
- Si possono definire sezioni critiche più piccole di un metodo intero.

```
[…]
public int read(){
        synchronized(this){
                while(writers>0){
                        try {
                                wait();
                        } catch(InterruptedException e){}
                }
                readers++;
        }
        // Do the reading...
        try{
                Thread.sleep(50);
        } catch(InterruptedException e){}
        int contSnapshot = content;
        synchronized(this){
                readers--;
                if(readers==0) notifyAll();
        }
        return contSnapshot;
}
```

Proprietà

I metodi *synchronized* danno accesso **esclusivo** ai dati incapsulati nell'oggetto solo se a tali dati si può accedere **esclusivamente** con metodi *synchronized*.

- Non ha senso pensare di sincronizzare le variabili locali di un metodo.
- Non ha senso creare metodi sincronizzati che non accedano a proprietà dell'oggetto.

Sincronizzare i metodi può essere vanificato se i dati sono accessibili via *dot notation*.

- Sia il main che i thread potrebbero modificare i dati dell'oggetto senza passare per le **barriere della sincronizzazione**;
- Tutte le proprietà che rappresentano risorse condivise dovrebbero essere accessibili solo da un metodo sincronizzato all'interno dell'oggetto.

Se di norma è consigliabile dichiarare **private** le property di un oggetto, in caso di risorse condivise in programmi concorrenti questo diventa davvero cruciale.

Variabili statiche

Metodi e blocchi synchronized non assicurano l'accesso mutuamente esclusivo ai dati "statici".
I dati statici sono condivisi da tutti gli oggetti della stessa classe.

In Java ad ogni classe è associato un oggetto di classe Class (es. contocorrente.class).
Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto di tipo Class:

- Si può dichiarare un metodo come static synchronized;
- Si può dichiarare un blocco come synchronized sull'oggetto di tipo Class.

Attenzione: Il lock a livello classe non si ottiene quando ci si sincronizza su un oggetto di tale classe e viceversa.

```
class StaticSharedVariable {
        // due modi per ottenere lock a livello di classe

        private static int shared;
        public int read() {
                synchronized(StaticSharedVariable.class) {
                return shared;
        }

        public synchronized static void write(int i) {
                shared=i;
        }
}
```

Ereditarietà

La specifica synchronized non fa propriamente parte della segnatura di un metodo.
Quindi una classe derivata può ridefinire un metodo synchronized come non synchronized e viceversa.

Il fatto che synchronized non faccia parte della segnatura è molto comodo, perché ci consente di

1. Definire classi adatte all'uso sequenziale senza preoccuparci dei problemi della concorrenza;
2. Poi modificare queste classi derivando sottoclassi che vengono rese adatte all'uso concorrente mediante synchronized.

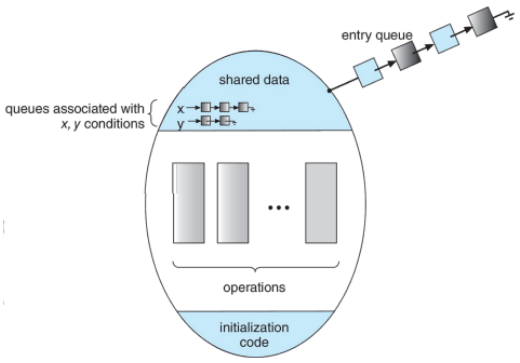
```
public class AssegnatoreConcorrente extends AssegnatoreSequenziale {
                public synchronized boolean assegnaPosti(String cliente, int numPosti){
                System.out.println("Assegnatore derivato");
                return super.assegnaPosti(cliente,numPosti);
        }
}
```

Si può quindi ereditare da una classe "non sincronizzata" e ridefinire un metodo come synchronized, che richiama semplicemente l'implementazione della superclasse. Questo assicura che gli accessi ai metodi nella sottoclasse avvengano in modo sincronizzato.

3.2.2.3 Accesso condizionato alle risorse condivise

Monitor

> **Monitor**
>
> Tipo di dati astratto, le cui variabili interne son accessibili da un insieme di procedure esposte dal monitor. Solo un processo/thread alla volta può essere attivo nel monitor (mutua esclusione).



Caratteristiche

- Permette di definire in essa delle **variabili di tipo condizione**, sulle quali è possibile esegue due operazioni:
  - *x.wait()*: mette in stato di attesa il processo/thread corrente, e lo forza a lasciare il monitor;
  - *x.notify()*: rende ready un processo/thread che aveva invocato x.wait(), se esiste.
    Possono essere eseguiti solo all'interno di un metodo synchronized.
- Consente al programmatore di:
  - Avere **mutua esclusione** nell'accesso a determinate risorse (le proprietà dell'istanza)
    - *wait* e *notify/notifyAll*
  - **Poter bloccare** (far attendere) un thread in attesa del verificarsi di una certa condizione
  - **Poter segnalare** ai thread in attesa che le condizioni in merito al costrutto sono cambiate e va rivalutata la necessità di attendere ancora
- Necessario avere delle **primitive** (metodi) per:
  - **Attendere** il verificarsi di una condizione su una risorsa condivisa
  - **Notificare chi è in attesa** del verificarsi di una certa condizione

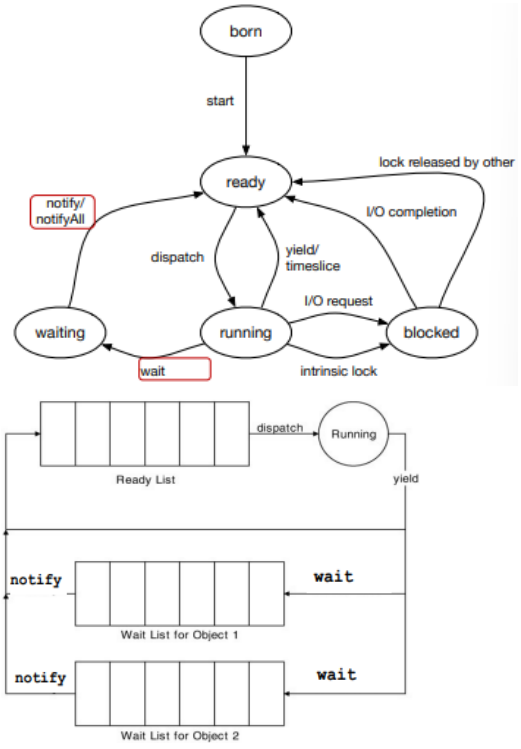
Esempio Produttore-Consumatore

```
public synchronized void insert(char ch) {
        try { while(BufferSize == MaxBuffSize) {
                //Finchè il buffer non è pieno, attendi
                wait();
        }
        // Inserisci nel buffer
        BufferEnd = (BufferEnd + 1) % MaxBuffSize;
        store[BufferEnd] = ch;
        BufferSize++;
        // Avvisa chi è in attesa sul buffer
        notifyAll();
        } catch (InterruptedException e){
                System.out.println("Thread interrupted.");
        }
}

public synchronized char delete() {
        try { while (BufferSize == 0) {
                //Finchè il buffer è vuoto, attendi
                wait();
        }
        // Cancella dal buffer
        char ch = store[BufferStart];
        BufferStart = (BufferStart + 1) % MaxBuffSize;
        BufferSize--;
        // Avvisa chi è in attesa sul buffer
        notifyAll();
        return ch;
```

```
            } catch (InterruptedException e) {
                    System.out.println("Thread interrupted.");
                    return '%';
            }
    }
```

N.B.: l'oggetto Buffer costruito può essere considerato anche come una coda bloccante (BlockingQueue). Questa interfaccia presenta due metodi bloccanti put() e take().

Diagramma degli stati di un thread



Quando un thread va in attesa (**wait**) su un oggetto, lo scheduler lo sposta in una “Wait List” associata a quell’oggetto, e rilascia il lock sull’oggetto. Quando un thread in attesa viene svegliato (mediante **notify**), lo scheduler lo sposta nella Ready List.

Il monitor ha bisogno di **due code di processi in attesa**:

1. **Una per per i thread bloccati** dal meccanismo automatico/trasparente di gestione della mutua esclusione
2. **Una per i thread** che si sono messi **in attesa** del verificarsi di una condizione

---

3.3 Variabili atomiche

In ambienti multi-threaded, dove diversi thread operano su una singola variabile, un thread per volta può accedere e leggere o modificare la variabile (mutua esclusione). Se così non fosse si creerebbero stati di inconsistenza dei dati (**race condition**).

> **Operazione atomica**
>
> Un'operazione è atomica quando tutte le sub-operazioni che la compongono verranno eseguite senza possibilità di interruzione da parte di un altro thread.

Nel package *java.util.concurrent.atomic* sono definite una serie di classi che supportano le operazioni atomiche. L'atomicità rende la mutua esclusione ridondante.

Tipi di atomicità

- **Reale**: una sola istruzione di CPU viene impiegata per eseguire l'operazione.
- **Virtuale**: il thread "crede" di avere accesso atomico alla variabile. Concettualmente simile a monitor (oggetti con metodi synchronized).

Come abbiamo già detto, in caso di accesso concorrente ad una risorsa condivisa è necessario garantire due condizioni per un comportamento coerente: **Mutua esclusione** e **Visibilità.**
Problemi di visibilità si creano quando due thread sono in esecuzione su core diversi, perché le variabili condivise vengono copiate e mantenute in cache per ragioni di performance.
Si risolve utilizzando variabili *volatile* (anche i metodi e i blocchi sincronizzati forniscono entrambe le proprietà, a scapito delle performance).

→ Variabili atomiche: *java.util.concurrent.atomic*

- "Variabili volatile migliorate";
- Queste classi incapsulano lo stato di una variabile e forniscono operazioni atomiche per manipolarne i valori;
- Operazioni di aggiornamento che non richiedono dovrebbero richiedere lock, sono basate su operazioni atomiche (se disponibili).

→ Algoritmi non bloccanti

- Sono meccanismi per accesso ad una risorsa condivisa thread-safe senza ricorrere a lock (se la contesa non è frequente, la richiesta e l'esecuzione di un lock non è necessaria e aggiunge overhead);
- Più complessi degli equivalenti basati su lock bloccanti;
- Basati sulle funzionalità delle variabili atomiche.

3.3.1 Compare-and-Swap (CAS)

Compare-and-Swap (CAS)

Operazione atomica messa a disposizione da alcuni processori che prende in ingresso una posizione di memoria V, un valore atteso E e un nuovo valore N ed aggiorna atomicamente la posizione V al nuovo valore N, soltanto se il valore presente in V corrisponde al valore atteso E.

L'aggiornamento ha avuto successo

- Se il valore restituito è uguale al valore atteso E
- Altrimenti non c'è stato aggiornamento

```
function cas(v: pointer to int, e: int, n: int) returns int {
        o ← &v
        if o == e
                *v ← n
        return o }
```

Compare-and-Set

Simile a CAS, ma restituisce true se l'operazione si è conclusa con successo, false altrimenti.

```
public class IntSimulatedCAS {
        private int value;

        public int compareAndSwap (int exp, int newValue) {
                int oldValue = value;
                synchronized(this) {
                        if (oldValue == exp) value= new;
                }
                return oldValue;
        }
        public boolean compareAndSet(int exp,int newValue) {
                return (exp == compareAndSwap(exp, newValue));
        }
        …
}
```

In questo modo, in caso si debba effettuare l'incremento di una variabile tra più thread, si può utilizzare CAS.

Problemi senza CAS: incrementare una variabile intera condivisa c, evitando la race condition ma con meno overhead del blocco synchronized.

> Idea 💡 : Più thread provano ad aggiornare c, ma solo uno ha successo.
> I thread perdenti non vengono sospesi e possono riprovarci.

Questo perchè:

- CAS è una operazione atomica (non può essere interrotta - 10 a 150 cicli di CPU);
- Dato che il risultato di CAS è visibile (come gli aggiornamenti delle variabili volatile), se non fallisce la variabile risulterà aggiornata per tutti i thread;
- Altrimenti l'operazione viene ritentata fino a che l'aggiornamento non ha successo.

Il CAS quindi garantisce una **sincronizzazione lock-free**, ma c'è il rischio (remoto) di **starvation** (il thread potrebbe non riuscire mai ad incrementare la variabile).

Interfaccia delle classi atomiche in Java

- set(), get()
- getAndSet() cambia atomicamente il riferimento ad un oggetto restituendo il riferimento precedente
- boolean compareAndSet(T expect, T new)
- Aritmetica (incrementa, decrementa) se supportata dal tipo
- Gli aggiornamenti degli elementi nelle varie posizioni sono atomici (thread-safe)

3.3.2 Aggiornamento di oggetti complessi

Immaginiamo di voler mantenere una caratteristica invariante tra due campi. Possiamo fare un lock e controllare per ogni aggiornamento che la caratteristica sia mantenuta, ma è possibile farlo in maniera non bloccante usando CAS?

> Idea 💡 : Trasformare l'aggiornamento multiplo in un aggiornamento unico e utilizziamo la strategia del **Optimistic retrying**.

Esempio: se vogliamo aggiornare un oggetto che indica un intervallo di numeri tra lower e upper, dobbiamo ricorrerre a delle classi di aiuto (helper classes) in cui salvare entrambi i campi.
AtomicReferences può essere usato per verificare l'aggiornamento dell'oggetto in questione usando compare-and-set (CAS).
Per realizzare una versione lock-free e threadsafe della classe NumberRange, usiamo una classe Helper immutabile che rappresenta una coppia di interi.

```
// Classe helper immutabile
public class IntPair {
        private final int l;
        private final int u;

        IntPair(int l, int u) {
                this.l = l;
                this.u = u;
        }

        int getL() {return l;}
        int getU() {return u;}
}
```

```
public class NumberRange {
        private final AtomicReference pair;

        public NumberRange (int lower, int upper) {
                if (lower > upper) {
                        throw new IllegalArgumentException();
                } else {
                        pair = new AtomicReference(new IntPair(lower,upper));
                }
        }

        public void setLower (int newLower) {
                IntPair oldPair, newPair;
                do {
                        oldPair = pair.get();
                        if (newLower > oldPair.getU())
                                throw new IllegalArgumentException();
                        newPair = new IntPair(newLower, oldPair.getU());
                }
                while (!pair.compareAndSet(oldPair,newPair));
        }
}
```

Riassunto variabili atomiche

- Sono un sistema di sincronizzazione “leggero” e performante (no context switch, no thread scheduling);
- Sono una generalizzazione delle variabili volatile (garanzie sulla visibilità);
- Permettono operazioni read-modify-write atomiche senza usare lock;
- La gestione della contesa è limitata ad una singola variabile;
- In generale, le variabili atomiche non offrono il supporto per sequenze atomiche di tipo check-then-act.

⚠ Check-then-act race condition:

1. Check: Molti thread verificano che una certa condizione sia vera.
2. Act: In base al risultato ottenuto, loro provano ad eseguire delle operazioni.

Nel tempo che intercorre tra la verifica della condizione e l’inizio dell’azione, la condizione può essere stata invalidata da altri thread.

3.4 Prestazioni a confronto

Contatore atomico VS synchronized

Single-thread

	Non sincronizzato	Sincronizzato	Atomico
Collisioni	No	No	No
Tempo (in millisecondi)	6	229	97

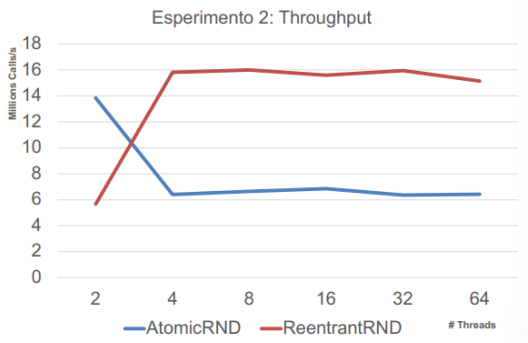
Multi-thread

	Non sincronizzato	Sincronizzato	Atomico
Collisioni	Sì	No	No
Tempo (in millisecondi)	66	399	216

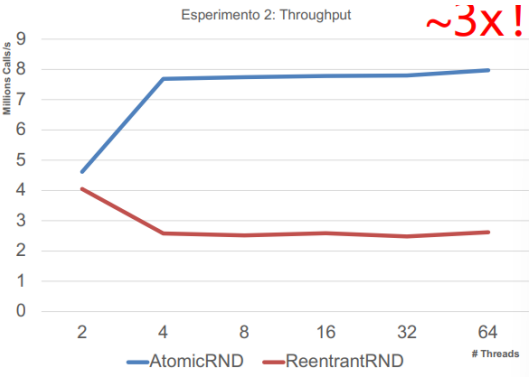
> Il contatore atomico è circa 2x più veloce dell'implementazione con synchronized!

Lock vs CAS

Contesa elevata



Contesa medio-bassa



> N.B.: il meccanismo di Locking viene detto pessimistico. Se la contesa non è frequente, nella maggior parte dei casi la richiesta e l'esecuzione di un lock non è necessaria ed aggiunge overhead.
>
> Nello schema viene rappresentato il caso in cui in presenza di un medio-basso numero di contese, il throughput del Locking è molto più basso.

Throughput

Throughput: quantità di dati che possono essere trasferiti attraverso una rete in un dato intervallo di tempo. Il throughput è una misura della performance di un sistema, indicando la sua efficienza nel completare un determinato tipo di operazione o processo.

3.5 Problemi della concorrenza

Liveness

Differenza tra Safety e Liveness

Safety: nessuno stato scorretto/incoerente è raggiungibile dall'applicazione.
Liveness: gli agenti riescono sempre a progredire nella loro elaborazione.

Nella programmazione concorrente, per liveness si intende il fatto che un programma sia sempre in grado di progredire correttamente.
Tuttavia questa proprietà non è sempre garantita dal fatto che le componenti concorrenti siano attive, e accedano in maniera mutuamente esclusiva alle cosiddette **sezioni critiche** di un oggetto (porzioni di codice ad accesso controllato e ristretto, come metodi *synchronized*).
La proprietà di liveness è desiderabile, ma per essere definita più precisamente richiede di specificare quali siano i problemi che si vogliono evitare.

Concetto di Liveness

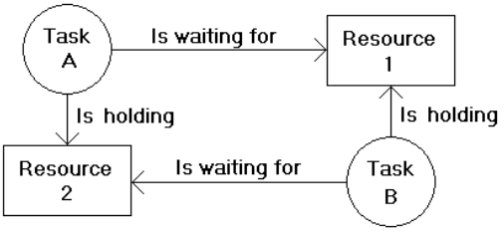
- Un sistema con diversi thread è libero da deadlock se, nonostante la competizione, almeno un processo riuscirà sempre ad accedere alla sezione critica e riuscirà a progredire nella sua esecuzione.
- Un sistema, in circostanze analoghe al precedente, è libero da starvation se garantisce che tutti i thread riescano ad accedere alla sezione critica e progredire nella loro esecuzione.

Deadlock

Definizione di deadlock

In programmazione concorrente, la situazione di deadlock si verifica quando ogni membro di un gruppo di agenti (nel nostro caso i thread) è in attesa che qualche altro membro rilasci un lock su di una risorsa per poter continuare la sua eleborazione.

Quando ad un processo viene garantito l'accesso esclusivo (ad esempio tramite una mutua esclusione) ad una risorsa, possono crearsi situazioni di stallo (deadlock).
In pratica si tratta di ***un'attesa circolare destinata a non terminare mai.*** Infatti, essendo tutti i thread in attesa, nessuno potrà mai generare l'evento di sblocco, quindi l'attesa si protrae all'infinito.



Condizioni necessarie

Ci sono quattro condizioni necessarie affinché un deadlock si verifichi. Queste sono generalmente espresse in termini di risorse assegnate a un thread.

1. **Mutua esclusione:** solo un agente concorrente (thread o processo) per volta può utilizzare una risorsa (cioè, la risorsa non è accessibile simultaneamente).
2. **Hold and wait o accumulo incrementale**: agenti concorrenti che sono in possesso di una risorsa possono richiederne altre senza rilasciare la prima. Possono mettersi quindi in attesa mantenendo il lock sulle loro risorse aspettando altre risorse da acquisire.
3. **No preemption sulle risorse condivise**: una risorsa può essere rilasciata solo volontariamente (non tolta con la forza dal SO/Scheduler/JVM) da un agente.
4. **Attesa circolare**: deve esistere una possibile catena circolare di agenti concorrenti e di richieste di accesso a risorse condivise tale che ogni agente mantiene bloccate delle risorse che contemporaneamente vengono richieste dagli agenti successivi. Dato l'insieme degli attività $A = \{A_1, A_2, \ldots, A_n\}$, A_1 è in attesa di almeno una risorsa da A_2, A_2 è in attesa di almeno una risorsa da A_3, … ed A_n è in attesa di almeno una risorsa da A_1.

Come affrontare il deadlock

- Deadlock prevention - il Deadlock può essere evitato se si fa in modo che almeno una delle quattro condizioni richieste per deadlock (Mutual exclusion, Hold and wait, No preemption e Circular wait) non si verifichi mai.
- Deadlock removal – non si previene il deadlock, ma lo si risolve quando ci si accorge che è avvenuto
  Ad es. rendendo il possesso delle risorse interrompibile, le risorse cioè sono requisibili da parte dello scheduler e quindi liberabile.

Deadlock prevention

- No hold and wait:
  - Si fa in modo che nessun agente possa detenere una risorsa mentre si è in attesa di un'altra risorsa
  - Non è sempre possibile (ad es., se devo fare un'elaborazione non interrompibile che coinvolge sia la risorsa a sia la risorsa b)
- No attesa circolare:
  - Si ordinano le richieste di accesso alle risorse, e si richiede il lock seguendo l'ordine.
    Ad es., se A precede B, bisogna cercare di acquisire B solo se si detiene già A.
  - Se tutti i thread si attengono a questa disciplina, non si possono creare deadlock circolari.
  - **Problema** (un esempio fra i tanti possibili):
    - Non è detto che esista questo ordinamento assoluto fra le richieste di risorse, né che possa valere (a livello di logica del programma) per tutti i thread.

Starvation

> **Definizione di Starvation**
>
> In programmazione concorrente, la situazione di starvation si verifica quando un agente non riesce ad accedere a una risorsa che gli viene perpetuamente negata.

Questo può essere dovuto alle circostanze associate ad una particolare politica dello scheduler.
Ad esempio, con uno scheduler a code con priorità statiche, in presenza di molti processi ad alta priorità, un processo a priorità bassa potrebbe non essere mai eseguito.

Anche una gestione scorretta della mutua esclusione o una definizione inadeguata dell'algoritmo di accesso alle

risorse (esempio della rotonda con traffico sbilanciato) potrebbe causare starvation.
Questa condizione è più difficile da definire formalmente, e anche più difficile da prevenire.

Livelock

Simile al deadlock, ma più complicato da caratterizzare formalmente e da poter gestire in modo automatico.
L'idea è che in una certa situazione i membri di un gruppo di agenti possono non essere bloccati, ma ciononostante non progredire effettivamente…
Esempio: Immaginate due thread che devono "salutarsi" e abbiano due modi di farlo, inchinarsi o stringersi la mano: se l'algoritmo prevede che provino alternativamente un modo o l'altro ciclicamente l'uno o l'altro modo finché l'altro thread non saluta allo stesso modo e se partono da due punti diversi non si troveranno mai ad effettuare lo stesso saluto.

Se non gestita in modo attento, questa situazione può portare a livelock.
Diversi protocolli distribuiti hanno necessità di effettuare operazioni di sincronizzazione iniziale, anche detta **handshake**, quindi questo genere di problematica è meno caricaturale di quanto possa sembrare.

3.6 Thread in Java

In Java ogni programma in esecuzione è un thread.
Esempio: il metodo main() è associato al thread "main"

- **currentThread()**: permette di accedere alle proprietà del thread in esecuzione.
  Thread[Nome thread, Priorità, Gruppo di appartenenza del thread]

```
public class ThreadMain {
public static void main(String args []) {
        Thread t = Thread.currentThread();
        System.out.println("Thread corrente: " + t );
        t.setName("Mio Thread");
        System.out.println("Dopo cambio nome: " + t );
        }
}
```

Output:
Thread corrente: Thread[main, 5, main]
Dopo cambio nome: Thread[Mio Thread, 5, main]

3.6.1 Creazione di un thread

Estensione di Thread

1. Estendere la classe *java.lang.Thread* (che contiene un metodo run() vuoto)
2. Riscrivere (ridefinire, **override**) il metodo *run()* nella sottoclasse
   - Il codice eseguito dal thread è incluso nel metodo *run()* e nei metodi invocati direttamente o indirettamente da *run()*
   - Questo è il codice che verrà eseguito (concorrentemente a quello degli altri thread)
3. Creare un'istanza della sottoclasse
4. Richiamare il metodo **start()** su questa istanza
   NB: spesso si estende il costruttore in modo che questo invochi *start()*: in tal modo creare l'istanza della sottoclasse fa anche partire il thread.

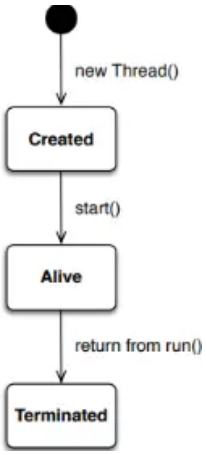
Questa chiamata rende il thread pronto (**ready**) all'esecuzione. Prima o poi (quando lo scheduler lo riterrà opportuno) il thread verrà mandato in esecuzione e verrà invocato il metodo *run()* del thread (di cui l'invocazione avviene in maniera automatica ed è implementata internamente alla classe Thread).

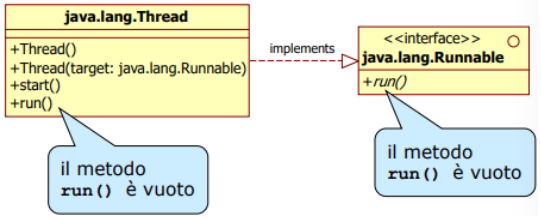
- Un thread è considerato alive finché il metodo *run()* non termina;
- Quando *run()* ritorna, il thread è considerato terminated;
- Una volta che un thread è terminato non può essere rieseguito (**IllegalThreadStateException**);
- Non si può far partire lo stesso thread (stessa istanza) più volte.
  isAlive() può essere usato per valutare se il thread è stato fatto partire e al contempo se non sia stato terminato (ovvero, se il metodo run() non sia già terminato)

I due thread (creante e creato) saranno eseguiti in modo **concorrente** ed **indipendente**.

Attenzione: L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine globale in cui le istruzioni dei vari thread saranno eseguite effettivamente è indeterminato (*nondeterminismo*).

> ⚠ **Differenza tra start() e run()**
>
> Solo la chiamata di start() crea un nuovo thread. Si può invocare run() direttamente, ma in questo modo il metodo **verrà eseguito normalmente**, sullo stack del thread corrente, senza che un nuovo thread venga creato: non lo fate!





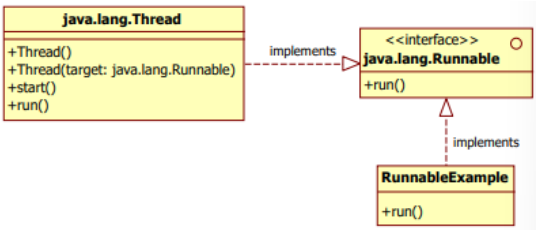

```java
public class ThreadExample extends Thread {
        public void run() {
                System.out.println("Ciao!");
        }

        public static void main(String arg[]){
                ThreadExample t1=new ThreadExample();
                t1.start();
        }
}
```

Implementazione di Runnable interface

Siccome Java non consente l'ereditarietà multipla, un modo alternativo di realizzare un thread è implementare solamente il metodo run (ovvero implementare l'interfaccia Runnable), avendo la possibilità di estendere un'altra classe base (maggior flessibilità).
La classe base Thread (il cui metodo run non fa nulla) può essere inizializzata con un oggetto Runnable, del quale utilizzerà il metodo run una volta che viene fatta partire.

Per creare thread usando l'Interfaccia *java.lang.Runnable*

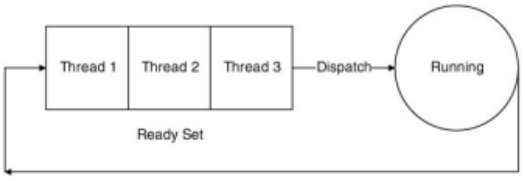
1. Definire una classe, implementazione di *Runnable*, dotata di un metodo *run()* significativo
2. Creare un'istanza di questa classe
3. Istanziare un nuovo *Thread* (cioè un'istanza della classe *Thread*) passando al costruttore l'istanza della classe che implementa Runnable
4. Chiamare il metodo **start()** sull'istanza di *Thread*




```java
public class RunnableExample implements Runnable{
        public void run() {
                System.out.println("Ciao!");
        }

        public static void main(String arg[]){
                RunnableExample re = new RunnableExample();
                Thread t1 = new Thread(re);
                // t1 contiene un riferimento a re
                t1.start();
                // Quando il metodo t1.start() viene chiamato,
                // si esegue il metodo run() fornito da re
        }
}
```

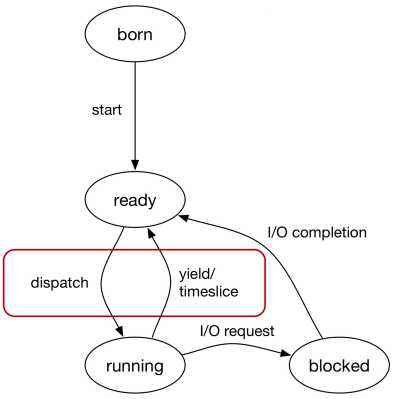
Diagramma degli stati di un thread

1. Quando invochiamo il metodo start() su un thread, il thread non viene eseguito immediatamente, ma passa nello stato di Ready.
2. Quando lo scheduler lo seleziona dalla coda, passa nello stato Running, ed esegue il metodo run() (la prima volta dall'inizio, poi da dov'era rimasto).



Come funziona esattamente lo scheduler dipende dalla specifica piattaforma in cui viene eseguita la JVM. In generale:
• Scheduling preemptive e priority based
• I thread Java hanno una priorità e il thread con la priorità più alta tra quelli ready viene schedulato per essere eseguito.

• Con il diritto di preemption lo scheduler può sottrarre la CPU al processo che la sta usando per assegnarla ad un altro processo.



yield()

yield() fornisce un meccanismo per informare lo scheduler che il thread corrente è disposto a rinunciare al suo attuale uso del processore ma vorrebbe essere schedulato di nuovo il prima possibile.
Lo "scheduler" è libero di aderire o ignorare questa informazione e, infatti, ha un comportamento variabile a seconda del sistema operativo.

3.6.2 Rallentamento di un thread

Busy loop

Per rallentare l'esecuzione dei due Thread potremmo creare un metodo privato che cicli a vuoto allo scopo di perdere del tempo.

```
public void busyLoop() {
        // wait for 60 seconds long
        startTime = System.currenTimeMillis();
        long stopTime = startTime + 60000;
        while (System.currentTimeMillis() < stopTime) {
         // do nothing, just loop
        }
}
```

Generalmente non è una buona soluzione: si sprecano cicli di processore mentre si potrebbero impiegare in qualche attività utile. Inoltre, se ci sono molti thread, è possibile che quando si giunge al momento in cui il thread deve uscire dal ciclo, il thread in questione non abbia accesso alla CPU. Con la conseguenza che esce in ritardo rispetto al momento desiderato.

Wait

```
public class ThreadSleep extends Thread {
        public ThreadSleep(String s) {
                super(s);
        }
        public void run() {
                for (int i=0; i<5; ++i) {
                        System.out.println(getName()
                        + ": in esecuzione.");
                        try {
                        Thread.sleep(200);
```

```
                        }
                        catch (InterruptedException e) {
                        System.err.println(getName()
                        + ": interrotto");
                        break;
                        }
                }
                System.err.println(getName() + ": finito");
        }
}
```

Thread.sleep(milli)

Ferma l'esecuzione del thread per milli millisecondi, liberando la CPU (cfr. grafico di due slide precedenti). Non utilizza cicli del processore ed è un metodo statico e mette in pausa il thread corrente. Non è possibile per un thread mettere in pausa un altro thread.

Attenzione: questo metodo può lanciare una *InterruptedException*, che va catturata e gestita.

3.6.3 Cancellazione dei thread

Cancellazione == Terminazione prematura di un thread

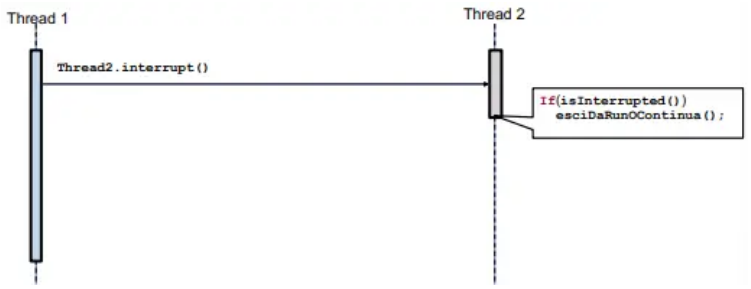
Due possibili modelli:

- **Cancellazione immediata**: il thread è terminato immediatamente
- **Cancellazione differita**: il thread controlla periodicamente se deve terminare, in modo da effettuare una terminazione ordinata (è il caso di Java)

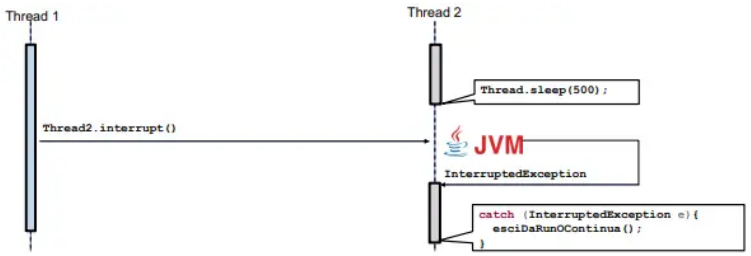
interrupt()

Il metodo *interrupt()* setta un flag di interruzione nel thread di destinazione e ritorna.

Il thread che riceve l'interruzione non viene effettivamente interrotto, quindi al ritorno di *interrupt()* non si può assumere che il thread sia stato effettivamente interrotto.
Il thread deve controllare se tale flag è settato (*isInterrupted()*) e nel caso uscire (dal *run()*).



Ad ogni chiamata di alcuni metodi come sleep, join, etc. il flag viene controllato automaticamente. Se un thread è già in pausa e nel mentre riceve un *interrupt()*, la JVM si incarica di accorgersene e di risvegliare il thread, lanciando l'eccezione *InterruptedException* e settando il flag.



Il thread che era stato interrotto intercetta l'eccezione e "dovrebbe" terminare l'esecuzione (cioè, deve gestire l'eccezione in qualche modo). Il fatto che il thread termini o meno dipende da cosa viene scritto nel codice che gestisce l'eccezione.

```
public class ThreadSleep extends Thread {
        public ThreadSleep(String s) {
                super(s);
        }
        public void run() {
                for (int i=0; i<5; ++i) {
                        System.out.println(getName() + ": in esecuzione.");
                        try {
                                Thread.sleep(200);
                        }
                        // Gestisce l'interruzione uscendo dal loop
                        // e quindi terminando
                        catch (InterruptedException e) {
                                System.err.println(getName() + ": interrotto");
                                break;
                        }
                }
                System.err.println(getName() + ": finito");
        }
}
```

Problemi con interrupt()

Il metodo interrupt() non funziona se il thread "interrotto" non esegue mai metodi di attesa (sleep, join, ...).
I thread devono controllare periodicamente il suo stato di "interruzione":

- isInterrupted() controlla il flag di interruzione senza resettarlo.
- Thread.interrupted() controlla il flag di interruzione del thread corrente e se settato lo resetta.

```
public class MyThread extends Thread {
        public void run() {
                while (condizione) {
                        if {Thread.interrupted();
                        System.err.println(getName() + ": interrotto");
                        break;
                        }
                }
        }
}
```

3.6.4 Threading implicito

Meccanismo che delega la creazione e gestione dei threads ai compilatori e alle librerie.
Permette agli sviluppatori di ragionare in termini di task da compiere, che vengono poi mappati sui thread in maniera automatica.

Thread pool

Idea 💡: viene creato un certo numero di thread, organizzati in un gruppo, che attendono di rispondere ad una richiesta di lavoro.

- Un sistema intelligente assegna ad ogni thread il lavoro da fare.
- Approccio utilizzato per la gestione del ciclo di vita delle servlet (i **servlet** sono oggetti scritti in linguaggio Java che operano all'interno di un server web oppure un server per applicazioni permettendo la creazione di applicazioni web).

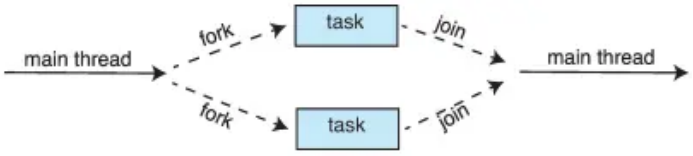
Vantaggi: Più rapido che creare il thread all'arrivo della richiesta. Il numero di threads contemporaneamente esistenti è limitato dalla dimensione del pool. Separa il task da svolgere dalla meccanica della sua creazione, permettendo diverse strategie di esecuzione (periodica, con ritardo…).

```
import java.util.concurrent.*;
class Summation implements Callable{
        private int upper;
        public Summation(int u){
                this.upper = u;
        }
        // the thread will execute this code public
        Integer call() {
                System.out.println("Thread started");
                int sum = 0;
                for (int i = 0; i <= upper; i++) {
                        sum+=i;
                }
                return Integer.valueOf(sum);
        }
}
```

```
import java.util.concurrent.*;
class Main {
        public static void main(String[] args) {
                ExecutorService pool =
                Executors.newSingleThreadExecutor();
                Future result = pool.submit(new Summation(10));
                try{
                        System.out.println("result: "+ result.get());
                }
                catch(InterruptedException | ExecutionException ie ){}
                pool.shutdown();
                System.out.println("Exiting");
        }
}
```

Oggetti generici **Future<>** permettono di rappresentare dei segnaposto nei quali i risultati dell'operazione saranno resi disponibili. Questo consente di effettuare computazione nell'attesa, sebbene poi sia necessario effettuare un'operazione di polling (richieste continue –while) per controllare che il risultato sia pronto.

Fork-join

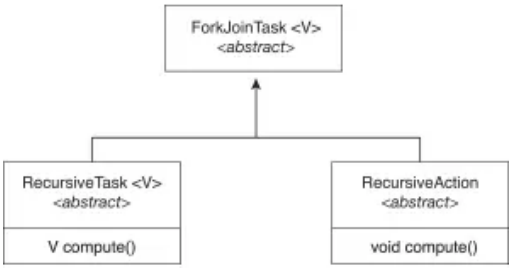


Pseudocodice

```
Task(problem)
        se il problema è abbastanza piccolo
                risolvi il problema direttamente
        altrimenti
                subtask1 = fork(new Task(sottoinsieme del problema))
                subtask2 = fork(new Task(sottoinsieme del problema))

                result1 = join(subtask1)
                result2 = join(subtask2)

                return risultati combinati
```

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];
SumTask task= new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

```
import java.util.concurrent.*;
public class SumTask extends RecursiveTask {
        static final int THRESHOLD = 1000;
        private int begin;
        private int end;
        private int array;

        public SumTask(int begin, int end, int[] array) {
                this.begin = begin;
                this.end = end;
                this.array = array;
        }

        protected Integer compute() {
                if (end - begin < THRESHOLD) {
                        int sum = 0;
                        for (int i= begin; i <= end; i++) sum += array[i];
                        return sum;
                }
                else {
                        int mid (begin + end) / 2;
                        SumTask leftTask= new SumTask (begin, mid, array);
                        SumTask rightTask= new SumTask (mid + 1, end, array);
                        leftTask.fork();
                        rightTask.fork();
                        return rightTask.join() + leftTask.join();
                }
        }
}
```



3.7 Casi paradigmatici

3.7.1 I filosofi a cena

Situazione:

1. Cinque filosofi sono seduti a un tavolo sul quale sono presenti cinque piatti di spaghetti e cinque forchette
2. Ogni filosofo alternativamente pensa o mangia: per mangiare, un filosofo ha bisogno di due forchette (quella alla sua sinistra e quella alla sua destra)
3. Le forchette non sono condivisibili simultaneamente (mutua esclusione)
4. Dopo aver mangiato un filosofo ripone sul tavolo le forchette usate
5. Si suppone che il piatto venga riempito da qualche entità esterna non appena diventi vuoto

Problemi:

- Deadlock: ciascuno dei filosofi ha in mano una forchetta e attende di prendere l'altra
- Starvation: almeno uno dei filosofi non riesce mai a prendere entrambe le forchette

Soluzione:

1. ***Dijkstra***: “spezza la simmetria” che causa l’attesa circolare (tutti I filosofi meno uno si comportano alla stessa maniera).
   In pratica, viene considerato l’ordine tra le forchette e si forza a prendere prima quella con identificativo minore
   Vantaggi: non richiedere un controllore centrale;
   Svantaggi: non subito chiare le conseguenze
2. ***Attesa casuale***: se un filosofo ha acquisito una forchetta e ma non è riuscito a procurarsi anche la seconda, può ipotizzare che si stia creando la condizione per un deadlock -> rilascia la forchetta in suo possesso e attende un po’ di tempo prima di riprovare a impossessarsi delle forchette.
   Questo non basta: se tutti i filosofi si comportano nello stesso modo, tutti rilasciano la forchetta destra contemporaneamente e poi tutti riprendono contemporaneamente la forchetta sinistra può crearsi una situazione di livelock.
   Perché il procedimento funzioni, occorre che l’attesa sia casuale, così diventa altamente improbabile (attenzione, non impossibile) che i filosofi cerchino di prendere una forchetta contemporaneamente.
3. ***Rimozione di hold and wait***: basta che ogni filosofo prenda (con una operazione atomica) entrambe le forchette (se disponibili), e aspetti se invece non sono disponibili. Ci vuole un mediatore/helper che osservi lo stato delle forchette e agisca di conseguenza. A questo scopo introduciamo la classe ForkPair.

3.7.2 Problema lettori/scrittori

Descrizione:

1. Abbiamo una risorsa condivisa che può essere acceduta in lettura da un più thread in contemporanea ma il cui accesso in scrittura deve invece essere esclusivo (per questioni di consistenza del dato)
2. I thread sono classificati sulla base dell’intenzione e del tipo di accesso (lettori o scrittori)
3. I thread lettori sono in numero molto maggiore rispetto agli scrittori, bisogna evitare che gli scrittori restino bloccati (starvation)

Soluzione 1:

- La **lettura** deve parzialmente sincronizzata, condizionata al fatto non ci siano scrittori attivi e che il numero di lettori sia inferiore al massimo permesso
- La **scrittura** deve essere invece completamente sincronizzata
- Nella risorsa condivisa è necessario mantenere due contatori, rispettivamente per il numero di lettori e scrittori attuali

Le invocazioni di lettura bloccano l'accesso alla risorsa database solo per il tempo necessario a verificare se ci sono uno scrittore o un lettore già attivi su di esso.
Le invocazioni in scrittura devono essere sincronizzate.
Quando la lettura/scrittura è terminata gli eventuali thread in attesa devono essere avvisati.

Problema:

- La soluzione proposta **non è Fair** (giusta) perché in caso ci sia uno sbilanciamento tra il numero di thread reader e quello dei thread writer un thread di tipo writer messo in attesa potrebbe accedere al database dopo thread reader creati più tardi di lui. I thread Reader e Writer **hanno la stessa priorità**, dovrebbero poter accedere alla risorsa con la medesima frequenza.
- Esiste anche il problema della **starvation**; se le letture arrivano in continuazione allora le scritture potrebbero rimanere sempre in attesa.

Soluzione corretta

1. La lettura deve parzialmente sincronizzata, condizionata al fatto non ci siano scrittori attivi (o in attesa) e che il numero di lettori sia inferiore al massimo permesso. Controllando il numero di scrittori in attesa si dà maggiore priorità a questi rispetto ai lettori che sono di più.
2. La scrittura deve essere invece completamente sincronizzata
3. Nella risorsa condivisa è necessario mantenere due contatori, rispettivamente per il numero di lettori e scrittori attuali