# Introduzione alla concorrenza

Non esistono SD che non siano concorrenti. La concorrenza è l’esecuzione di un programma in parti diverse (thread, processi…).

*I browser hanno 1 processo per ogni scheda aperta e poi un processo principale che gestisce tutto. Viene fatto per motivi di sicurezza, ma soprattutto per chiudere processi malati o che non funzionano.*

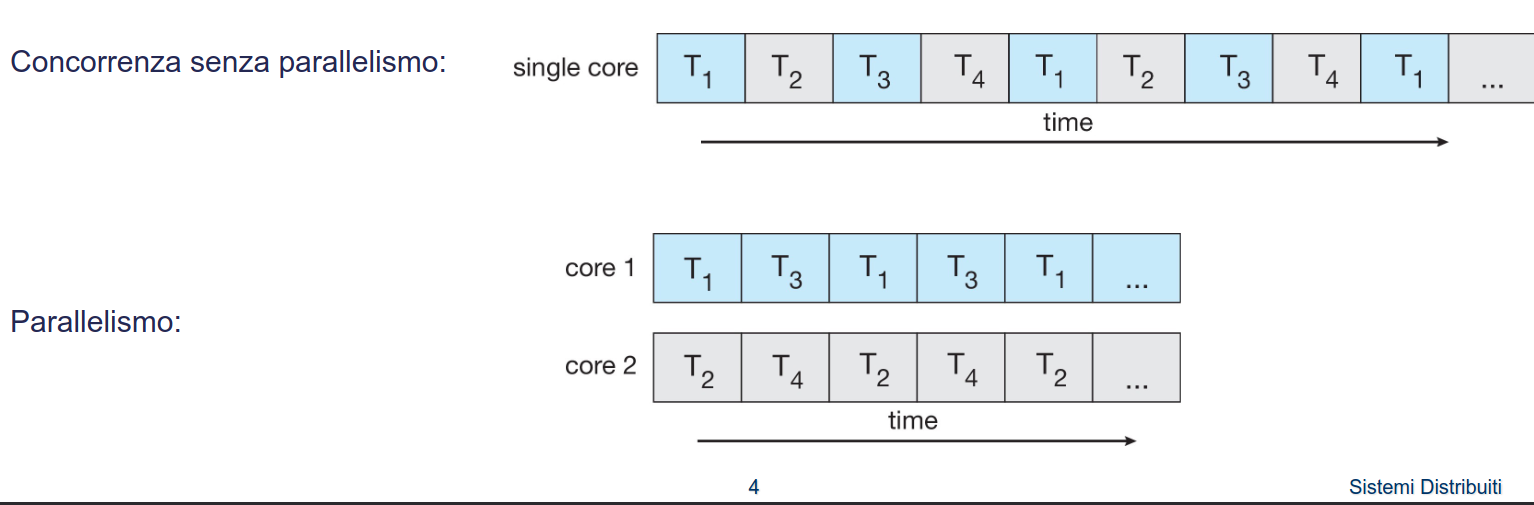
Ci sono 2 tipi di scenari possibili:

1. Contemporaneità -> stessa memoria e funzioni del SO (avviene su macchine multi-core)
2. Esecuzione su macchine distinte -> programmazione distribuita

Approcci possibili:

1. Funzionalità ambiente -> meccanismi messi a disposizione da SO
2. Funzionalità dei linguaggi di programmazione -> devono essere tali che lo sviluppatore sia capace di sfruttare le funzionalità ambiente. È importante che ci sia una forte astrazione.

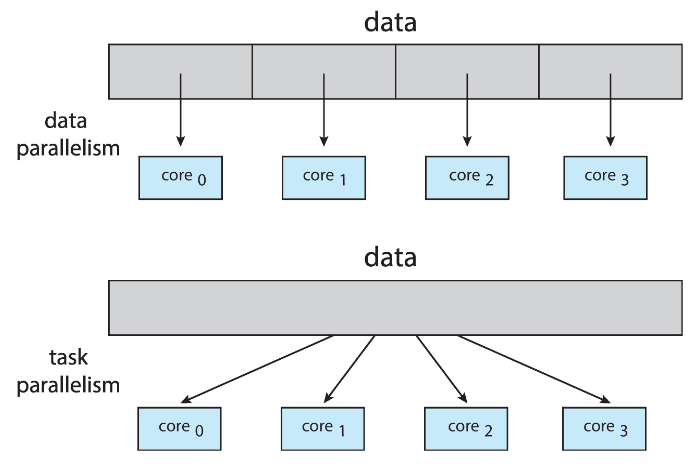
### CONCORRENZA E PARALLELISMO



La concorrenza fa progredire più di un’attività nel tempo (single-core). Mentre il parallelismo esegue più attività contemporaneamente (multi-core, multi-thread).

Al giorno d’oggi si usa una diversa definizione:

* Parallelismo dei dati tanti core | 1 attività.
* Parallelismo delle attività tanti core | tante attività, sharing la memoria



**LEGGE DI AMDAHL**

Se aggiungo un core quanto guadagno in performance? La legge di Amdahl mi dice quanto posso accelerare al massimo la mia computazione. Il valore che da non è mai pari ad 1. *A volte più core non servono a niente. Non tutto il codice può essere parallelizzato.*

**PROGRAMMAZIONE CONCORRENTE**

Nella programmazione concorrente i thread vengono eseguiti in parallelo. Ci sono almeno 2 flussi di esecuzione in contemporanea in un’applicazione concorrente.

Perché lo facciamo? Per evitare CPU Bound, sfruttare CPU multi-core…

### PROCESSI

**MULTIPROGRAMMAZIONE E MULTITASKING**

Sistema multiprogrammato, ha in memoria più programmi e massimizza l’utilizzo della CPU. Si è potuto realizzare grazie anche alla mem. virtuale. L’approccio è limitato se un programma è CPU Bound e si prende la CPU tutto il tempo.

*Multiprogrammazione e memoria = tutte le immagini dei processi devono essere in memoria, però la memoria non è abbastanza grande. Quindi usiamo la mem. virtuale per eseguire un processo di cui l’immagine non è completamente in memoria.*

Multitasking ha risolto il problema del CPU Bound: ogni programma ha un quantum di tempo per stare con la CPU. In questo modo simula il parallelismo ed evita il problema descritto prima.

**OPERAZIONI SUI PROCESSI**

Per la creazione di un processo si usa il paradigma padre-figlio e le chiamate di sistema. Il 1° processo è systemd (pwd=1) che crea tutti gli altri processi.

Vengono implementate delle politiche speciali:

* Padre e figlio condividono tutte le risorse **||** una parte **||** nessuna.
* Padre fermo finché i figli non terminano **||** eseguono in maniera concorrente

Per la terminazione di un processo o il padre lo fa terminare, oppure è il figlio che chiede la sua terminazione al SO. Si vuole evitare che il programmatore NON usi mai il comando exit.

Motivazioni per terminare un processo: non è più utile o il padre non esiste più.

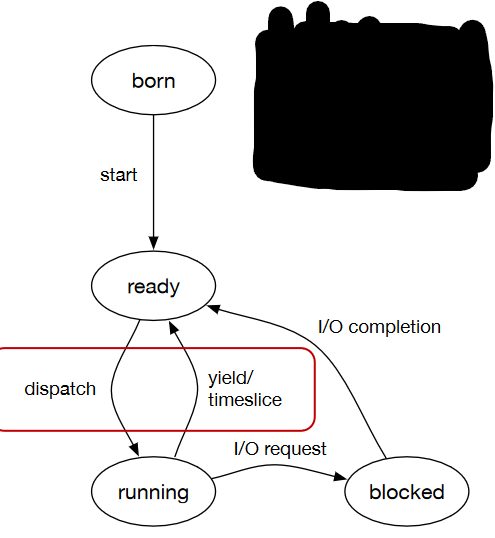
**IMPLEMENTAZIONE PROCESSO**

Un processo è composto da:

* stato registri (come il program counter)
* stato immagine del processo (ovvero il suo stack e codice…)
* stato processo (new, ready, running…)
* risorse SO attualmente in uso

Processi distinti hanno immagini distinte. Risorse SO possono essere condivise tra processi.

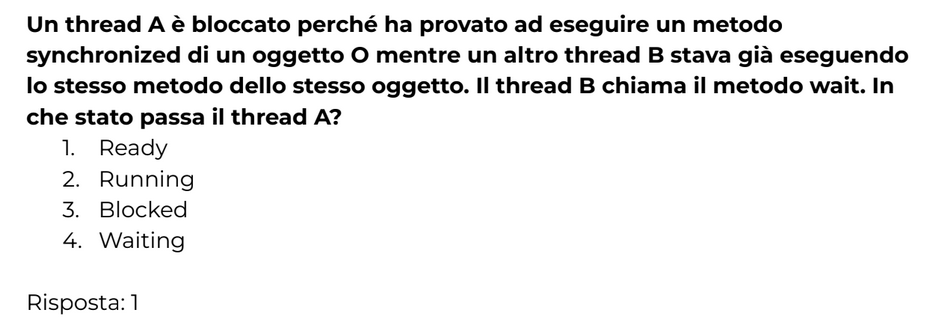
Il processo ha un'area dedicata al codice (sola lettura) e una per i dati. Come devo modificare la struttura del processo (contatori…) in modo da avere delle entità in esecuzione in contemporanea sullo stesso codice? Devono avere lo stack replicato in quanto durante le chiamate non devono modificare i parametri.

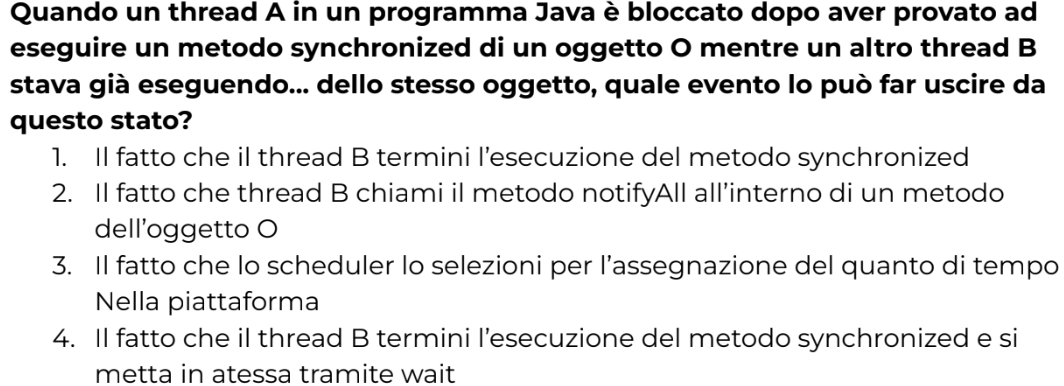


Java implementa un meccanismo di protezione di codice. Se siamo in thread diversi e vogliamo accedere alla stessa risorsa uno ha il lock (uso esclusivo della risorsa) e l’altro non ce l’ha.

**new Thread()** -> **start()** il thread passa allo stato di **ready** -> lo scheduler gli farà eseguire il metodo **run()** verrà portato allo stato di **running** -> Quando **run() ritorna**, il thread finisce nello stato di **terminato**. Ci sono inoltre 2 stati aggiuntivi:

* **Waiting** -> thread invoca il metodo wait() e quindi rilascia il lock. Un altro thread invoca: notifyAll() che sveglia tutti i thread in waiting OPPURE notify() che ne sveglia solo uno. Lo fanno andare da waiting -> ready.
* **Blocked** -> quando il thread deve eseguire un’operazione di I/O oppure quando tenta di accedere ad una risorsa il cui lock è già occupato. Per tornare allo stato ready deve aspettare la fine dell’operazione di I/O oppure il rilascio del lock della risorsa





La risposta è la numero -> (bianco) 4

Un monitor è un qualunque oggetto dotato di synchronized, di una coda di thread bloccati e una di thread in waiting.

**MULTITHREADING**

Per poter consentire l’esecuzione di diversi flussi di controllo in «parallelo» sono stati introdotti i Thread (LightWeight Process).

Questo comporta molti vantaggi: i thread condividono lo spazio di memoria del processo, creare un thread richiede meno risorse che creare un processo, context-switch più rapido, l’applicazione può eseguire un thread a parte e continuare a rispondere agli input utente, aumenta il grado di parallelismo, se un thread è in attesa di una operazione di I/O un altro thread del medesimo programma può accedere alla cpu evitando il context switch dell’intero processo.

Questo comporta anche delle sfide da affrontare: identificare delle attività da eseguire su core separati, bilanciare il carico sui thread, suddividere dei dati per massimizzare l’accesso in parallelo, stabilire la rispettiva sincronizzazione, testing e debugging.

Per ogni PCB (process… contiene informazioni di contesto comuni, tipo spazio memoria) c’è un TCB (informazioni di contesto di un singolo thread) che controlla un thread.

# Multithreading in Java

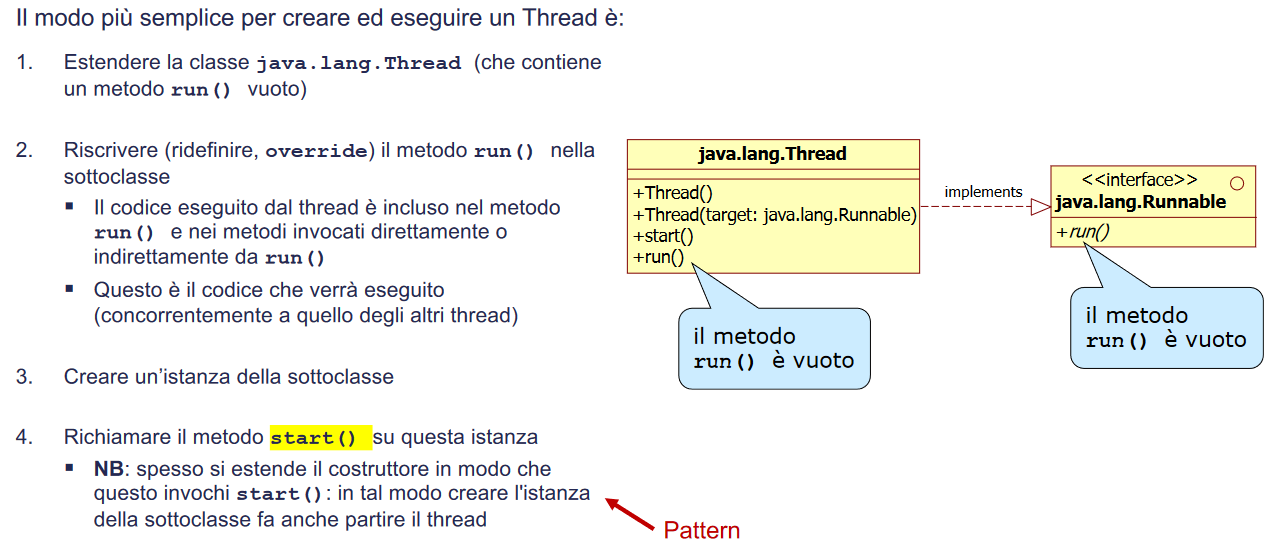
Java threads è una libreria integrata con il kernel per creare thread. JVM astrae la creazione dei thread, nasconde il livello sottostante al programmatore. Nasconde anche l'interazione con il SO per la loro esecuzione. *Bisogna evitare che il programmatore si interfacci con POSIX ect …*

Il thread main è il primo a finire.

JVM è un processo, per ogni programma Java che esegue c’è una JVM.

Ci sono 2 modi di creare un thread in java, questo perché java non ha ereditarietà multipla:

* Sottoclasse della classe standard java.lang.Thread (semplice)
* Implementazione metodo run()implementando l’interfaccia java.lang.Runnable (migliore)



Posso crearlo 1 volta sola e cancellarlo una volta sola. Per cancellarlo posso cancellarlo immediatamente oppure il thread controlla periodicamente quando deve terminare (lo fa Java).

**Thread Pool** -> Viene creato un certo numero di thread che attendono di lavorare. È più rapido che creare il thread all’arrivo della richiesta. Separa il task da svolgere dalla meccanica della sua creazione.

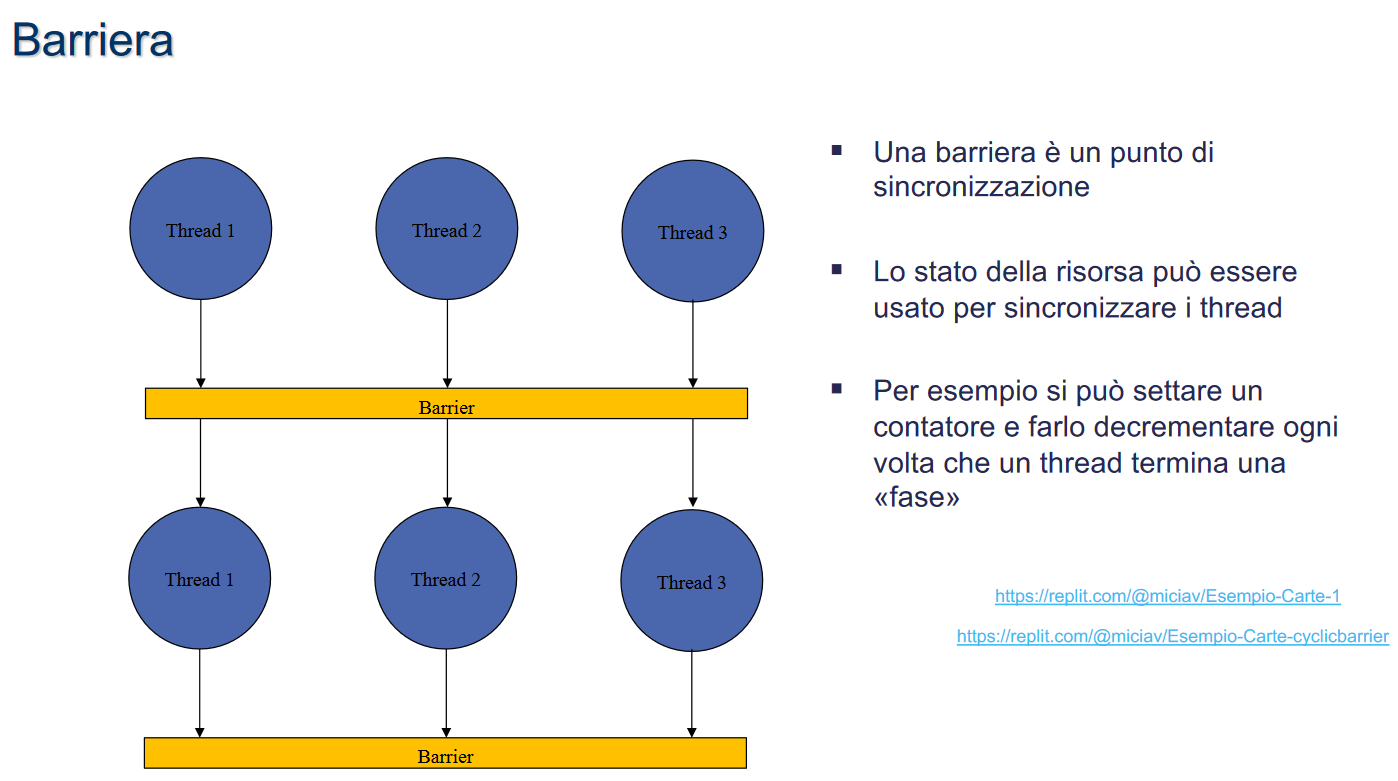
I thread se riescono ad operare nella loro fetta di tempo è OK altrimenti vengono sospesi. Se gli metto thread.sleep(40) starà nella sua fetta di tempo.

Un programma sequenziale può riprodurre costantemente lo stesso BUG. Non è così per i programmi concorrenti, dove un thread dipende da un altro in maniera non deterministica.

Servono però dei **meccanismi di sincronizzazione** per controllare l’ordine dei processi/thread. La JVM gestisce automaticamente i lock associati agli oggetti (non classi o altro). Essi vengono messi a disposizione dal SO e JVM li sfrutta a pieno:

Modello di **memoria condivisa**:

* Mutua esclusione: i dati non sono accessibili contemporaneamente a più thread
* Sincronizzazione su condizione: si sospende un thread fin quando non ci sono le condizioni per condividere le risorse, guardare l’immagine:



Modello a **scambio di messaggi**:

Tutto si riduce a “synchronize”. Java non consente al thread di accedere ad un metodo se un altro thread lo sta già usando. Evita sovrapposizioni dei thread su un singolo metodo.

In Java, la parola "synchronized" = blocchi di codice sincronizzati.

* Risorsa appartiene a tutte le istanze della classe (statica) si utilizza static synchronized
* Appartiene a uno specifico oggetto (non statica) si utilizza synchronized.

L'utilizzo inappropriato di "synchronized" può causare problemi di prestazioni o deadlock. Ci sono anche altre alternative come i lock espliciti o algoritmi non bloccanti.

**RACE CONDITIONS**

Situazioni in cui thread diversi operano su una risorsa ed il risultato viene a dipendere dall'ordine in cui essi effettuano le loro operazioni. Il problema nasce dalla competizione per l’accesso ad una risorsa condivisa.

Supponiamo di avere due thread A e B che condividono una variabile contatore `count`. Entrambi i thread vogliono incrementare il valore di count:

1. Thread A legge il valore corrente di `count`, che è 0.

2. Thread B legge il valore corrente di `count`, che è ancora 0.

3. Thread A incrementa il valore di `count` a 1.

4. Thread B incrementa il valore di `count` a 1.

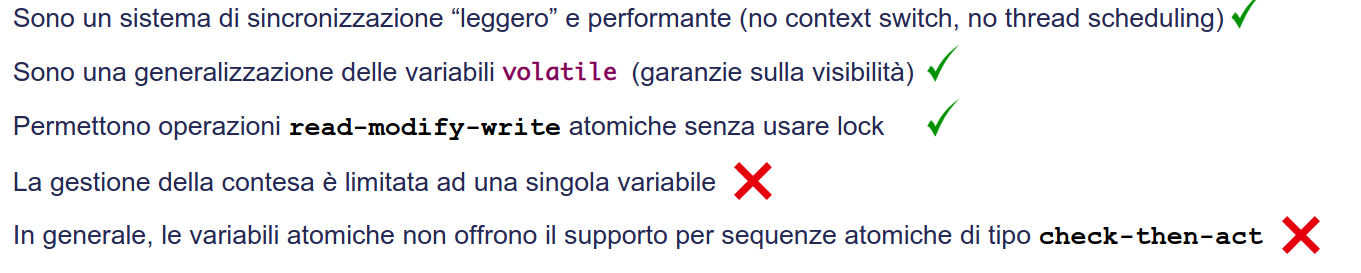
5. Thread A salva il nuovo valore di `count`, che è 1.

6. Thread B salva il nuovo valore di `count`, che è 1.

In questo caso, entrambi i thread intendevano incrementare `count` di 1, ma a causa della race condition, il risultato finale è uguale a 1 anziché 2.

Synchronized si assicura che NON si verifichino race conditions.

# Variabili Atomiche e Algoritmi ㄱ Bloccanti



Abbiamo visto come dare accesso ad una risorsa solo ad un thread alla volta (Mutua Esclusione). Esiste un altro approccio -> con le variabili atomiche e gli algoritmi non bloccanti.

Le operazioni atomiche vengono eseguite senza interruzioni:

* atomicità reale = unica istruzione non bloccante
* atomicità virtuale = ci fa credere che l’istruzione non sia interrompibile. A volte l’HW non ci permette di applicare l’atomicità reale quindi si usa questa, oppure la JVM pensa sia più efficiente usare questa.

Un’istruzione in Java di norma non si deve considerare atomica

I thread caricano in memoria locale la loro copia. Se questo avviene e c’è anche una modifica poi essa non viene propagata in tutti gli altri core. Per evitarlo basta propagare ovunque, partendo dalla copia più in basso nella gerarchia di cache.

La visibilità è stata introdotta ora perché la JVM rende volatili tutte le risorse in quel particolare metodo/blocco

**Ma perché i lock ci stanno sul cazzo?** perché costano tempo, specialmente su un linguaggio lento come Java. E poi serve context switching, può capitare la priority inversion (un thread con priorità bassa riesce ad ottenere un lock e blocca un thread con priorità più alta). Un thread poi può finire prima del suo quantum di tempo e quindi ora è il thread ad aspettare la CPU.

Approccio basato sui lock è detto pessimistico, crea una situazione rigida, un blocco CPU in qualunque momento. Se il carico di lavoro è poco poi si rischia di creare overhead inutile.

**Gli algoritmi non bloccanti usano questa filosofia** -> provare ancora piuttosto che chiedere il permesso per usare la variabile. Optimistic retrying = algoritmo non bloccante

Legge la variabile, la aggiorna, la scrive in memoria MA tutto ciò se non c’è collisione con un altro thread. Funziona solo se la variabile è accessibile su 1 solo indirizzo di memoria, gli oggetti grandi quindi non vanno bene.

Essi quindi non richiedono un lock, quindi non richiedono il blocco di un thread.

**Compare And Set** è un algoritmo non bloccante, però non è ancora lock-free. Infatti con grandi moli di dati CAS diventa inefficiente. Rendiamolo lock-free.

Quello che cambia è che la variabile è volatile, quindi quando cambia valore cambia per tutti i thread. Questo implica meno rotture di balle per aggiornare i valori. C’è il rischio di starvation, però è molto basso.

L’uso di sincronizzazione a vuoto costa tantissimo. Mentre l'algoritmo non bloccante è il più efficiente ed efficace.

Tanti thread di fila = senza lock

Tanti thread con tanto tempo tra un’istruzione e l’altra = poca probabilità di collisione tra i thread, è meglio implementare quello con il lock (CAS)

**AGGIORNAMENTI OGGETTI COMPLESSI**

Abbiamo 2 thread, con 1 oggetto che ha 2 campi. Il valore di riferimento è 100. Il 1° thread crea un nuovo oggetto e sostituisce quello vecchio, con un nuovo riferimento (204) e così fa anche il 2° thread.

thread1 usa CAS per aggiornare il riferimento all’oggetto, con quello del suo nuovo oggetto. Il riferimento è 204.

il thread2 è in ritardo quindi si fotte. C'è una collisione, deve ripetere il meccanismo sperando che thread1 non sia ancora più veloce. Il riferimento è ora 027.

QUESTO MI OBBLIGA A CREARE OGGETTI (JAVA) NUOVI OGNI VOLTA. Risolvo in 2 modi:

* Oggetti immutabili, sono oggetti che non possono cambiare il loro riferimento.
* Garbage Collector è un thread, quello con la priorità più bassa

# Problematiche di Liveness

**Safety** (nessuno stato incoerente progredisce) e **liveness** (c’è sempre progressione, almeno un processo riesce a progredire e tutti i thread sono liberi da starvation).

libero da starvation = libero da deadlock.

**DEADLOCK =** tutte le risorse soffrono di starvation. Ci sono 4 condizioni necessarie affinchè si verifichi un deadlock.

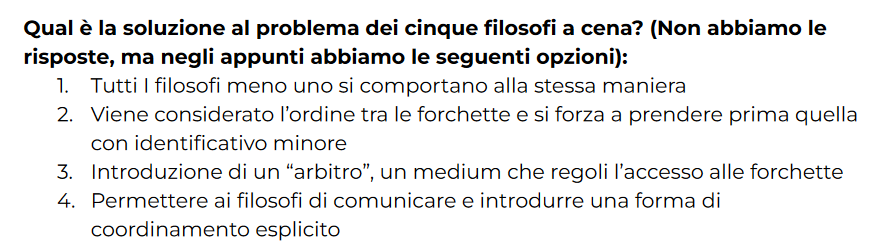
* Un risorsa può essere usata solo da 1 thread alla volta (**mutua esclusione**)
* Non esiste un'entità terza che impone di rilasciare la risorsa (**non preemptive**)
* Un thread può chiedere l’accesso ad altre risorse dopo che ne ha bloccata già una (**Hold and Wait**)
* Il ciclo creatosi non termina mai perché nessuno rilascia l’accesso alla risorsa per l’altro thread (**Attesa circolare**)

La condizione sufficiente ma non necessaria perché avvenga un deadlock è se esiste nel sistema una sola istanza per ogni tipo di risorsa.

**LIVELOCK =** Non c’è un blocco ma non si riesce comunque a progredire. Per esempio A e B possono effettuare un HANDSHAKE solo se hanno lo stesso valore. Però A=0 e B=1, quindi facciamo A++ e B++. Però non cambia niente, non riusciranno mai a comunicare.

Per risolvere usiamo un tempo randomico, dove magari A++ viene eseguito subito mentre B aspetta un attimo. Però ora A e B sono uguali e posso fare l’handshake.

Problemi/Risoluzione di Concorrenza: deadlock, livelock, liveness, safety, starvation



**PROBLEMA LETTORI/SCRITTORI**

Un thread vuole scrivere su una risorsa e l’altro vuole fare l’accesso. Molto simile al problema delle transazioni nei DBMS.

Facciamo leggere in contemporanea a tot. lettori perché tanto devono solo leggerla. Mentre ai scrittori l’accesso è a mutua esclusione (sincronizzazione), altrimenti il lettore potrebbe leggere una risorsa non più valida.

I lettori potrebbero essere infiniti, questo potrebbe causare starvation. Si risolve implementando una semi-sincronizzazione dando ogni tot. la palla agli scrittori.

Problema Lettori-Scrittori in Java? Consistenza del dato