Java: Thread e trabocchetti

Semplici nozioni per una civile convivenza
in ambiente multithread.

Flavio Casadei Della Chiesa
http://www.prato.linux.it/~fcasadei/
fcasadei@gmail.com

Licenza 1/

La paternità dell'opera è di Flavio Casadei Della Chiesa, il nome dell'autore deve essere riportato in forma scritta in ogni rappresentazione dell'opera.

Licenza 2/

Riferimenti

Java Concurrency In Practice

Concurrent Programming with Java

Corsi universitari che introducono la materia

Vari articoli che si trovano su internet

Miscredenze e falsi miti

Audience

- Chiunque sappia programmare in Java.
- Chiunque sia interessato alla programmazione multithread in Java.
- Chiunque abbia a che fare con Java.

Argomenti trattati

Parte 0

- Prerequisiti

Parte I

- Concetti elementari

Parte II

- Nozioni intermedie

Parte III

- Java 5 ed il suporto (aggiuntivo) alla concorrenza
- Comuni antipattern

Obiettivi

Questo non è un corso sulla programmazione concorrente o sul multithreading. Non è un corso di programmazione. E' solo un'opera mirata a sensibilizzare i programmatori Java verso problematiche tipiche della programmazione concorrente, problematiche di cui purtroppo sono completamente ignari.

Le immagini ed i contenuti

Alcuni delle immagini contenute in questo documento sono state elaborate da me in persona (non sono un artista), altre sono state recuperate da Internet senza infrangere alcun Copyright.

Alcuni contenuti ed alcune definizioni sono stati presi da varie fonti su Internet e su libri di testo, tali fonti verranno mostrare ove opportuno.

Prima di proseguire

Alcuni degli esempi e dei concetti presentati in queste slide hanno preso spunto dall'ottimo libro "*Java Concurrency In Practice*" di Brian Goetz.

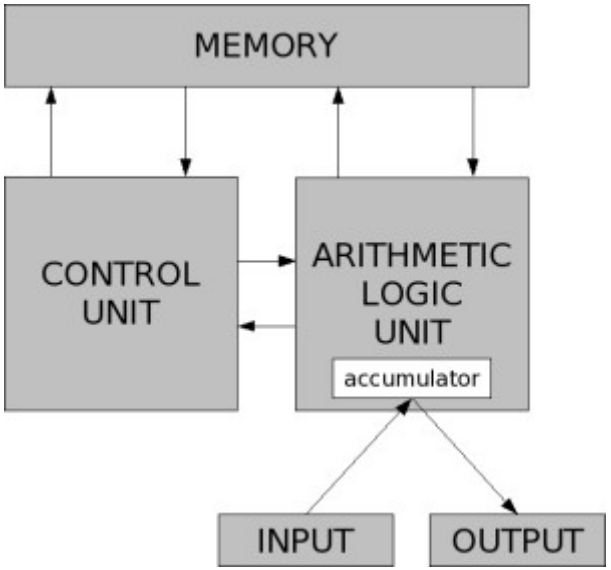
Il termine "In Practice" dovrebbe allontanare ogni sospetto di "cavolate accademiche o problemi teorici".

Durante la stesura delle slide ho più volte provato ad esporre i concetti con parole mie, molte volte non sono riuscito ad esporli in maniera chiara e sintetica. JCIP in molti punti rappresenta il massimo della chiarezza e della sintesi ed espone tutto in modo pragmatico.

Parte 0

Prerequisiti e introduzione

Macchina di Von Neumann



Componenti della macchina di Von Neumann

- CPU
  - Unità di controllo
  - Unità aritmetico–logica (ALU)
- Memoria
- Unità di Input
- Unità di Output

CPU

Compito della CPU è quello di leggere le istruzioni e i dati *dalla memoria* ed eseguire le istruzioni; il risultato della esecuzione di una istruzione dipende dal dato su cui opera e dallo stato interno della CPU stessa.

Componenti di una CPU moderna

- ALU: esegue operazioni aritmetiche e logiche
- *Program Counter*: indica l'istruzione corrente
- Registri di stato: vari flag
- (Pochi) Registri generici: piccole “memorie” sui quali opera la ALU
- Controllo: legge dati e istruzioni, esegue quest'ultime e scrive il risultato (in memoria o nei registri)

Una parolina sui registri

Il trend attuale (nei sistemi RISC) è che la ALU possa eseguire operazioni solo sui registri e **non** sulla memoria.

I registri contengono quindi gli operandi della ALU.

La memoria serve solo per immagazzinare informazioni, non per operare su di esse.

Ancora sui registri

La memoria è come un fornitissimo carrello degli attrezzi.

I registri sono il banco da lavoro.

Non si può lavorare nel carrello degli attrezzi, questi devono essere spostati sul banco per essere utilizzati.

Esempio di architettura

Verrà utilizzata come esempio di architettura di elaboratore una semplificazione dell'architettura MIPS.

Esecuzione di un'istruzione 1/

- **IF** Recupero istruzione (dalla memoria)
- **ID** Decodifica Istruzione (controllo)
- **EX** Esecuzione istruzione o calcolo indirizzo (ALU)
- **MEM** Accesso (lettura o scrittura) ad un dato in memoria
- **WB** Scrittura risultato *in un registro*

Esecuzione di un'istruzione 2/

L'esecuzione di una istruzione è quindi una sequenza di passi (*stages*) non alterabili o permutabili a causa della loro dipendenza.

I vari passi compiono azioni diverse ed operano su “dati” non in conflitto.

Esecuzione di un'istruzione 3/

Non tutte le istruzioni necessariamente accedono alla memoria, il passo MEM non fa niente ma non può essere saltato.

IF -> ID -> EX -> __ -> WB

Esecuzione di un'istruzione 4/

Non tutte le istruzioni scrivono un risultato in un registro, il passo WB quindi non fa niente ma non può essere saltato.

IF -> ID -> EX -> MEM -> __

Esecuzione di un'istruzione 5/

Esiste l'istruzione NOP (*no operation*).

IF -> ID -> __ -> __ -> __

(si, ha senso .... lo vedremo dopo)

Esempio casalingo .....

Fare il bucato

Fare il bucato 1/

- Prendere dal cesto i panni sporchi e metterli nella lavatrice
- Quando la lavatrice ha finito mettere i panni nell'asciugatore
- Quando l'asciugatore ha finito mettere i panni su un tavolo e stirarli

Fare il bucato 2/

Fare il bucato è quindi una sequenza di passi (*stages*) non alterabili o permutabili a causa della loro dipendenza.

I vari passi compiono azioni diverse ed operano su "dati" (panni) non in conflitto.

Fare il bucato 3/

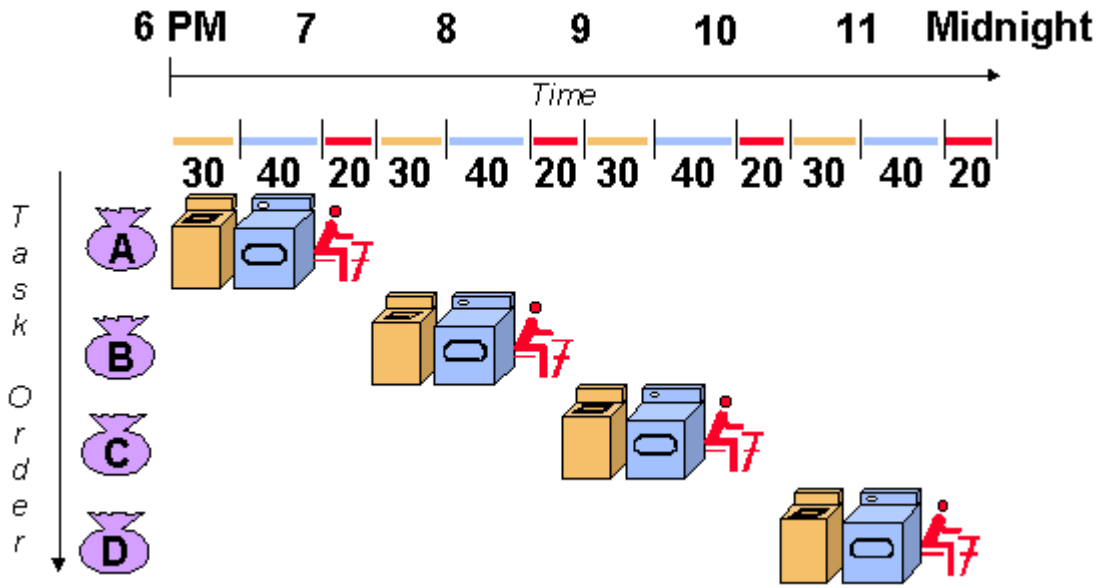
Problema: tutto il contenuto del cesto dei panni sporchi non entra nella lavatrice.

E' quindi necessario ripetere il procedimento fino ad esaurimento panni sporchi (o sfinimento fisico).

Fare il bucato 4/

Quanto ci vuole a fare tutto il bucato?

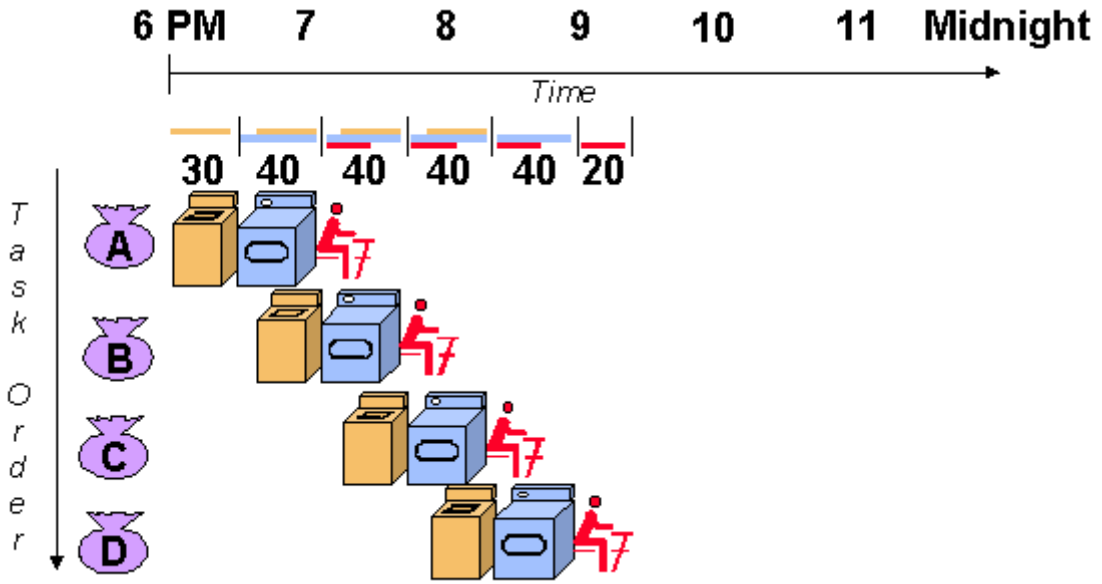
Fare il bucato 5/



Fare il bucato 6/

Esiste un modo più efficace di fare il bucato?

Fare il bucato 7/



Fare il bucato 8/

Nota: non è possibile stirare i panni mentre sono ancora nell'asciugatore.

E' tuttavia possibile stirare panni precedentemente asciugati.

Ma cosa c'entra il bucato con la CPU?

Pipeline 1/

Esecuzione normale

Pipeline 1/

Esecuzione normale

Pipeline 2/

La *pipeline* (conduttura, tubatura) è una tecnologia utilizzata dalle CPU per incrementare il numero di istruzioni processate per unità di tempo.

Pipeline 2/

La *pipeline* (conduttura, tubatura) è una tecnologia utilizzata dalle CPU per incrementare il numero di istruzioni processate per unità di tempo.

Pipeline 3/

Permette di eseguire *stage* di tipo diverso in parallelo.

Tuttavia neanche la *pipeline* può stirare i panni che sono nell'asciugatrice (*data hazard*).

Pipeline 4/

Il risultato di un'operazione aritmetico–logica o di lettura dalla memoria viene scritto nel registro opportuno solo nell'ultimo *stage* (WB) della *pipeline*.

La memoria viene letta o scritta nel penultimo *stage* (MEM).

Che problemi crea?

Vi ricordate?

- **IF** Recupero istruzione (dalla memoria)
- **ID** Decodifica Istruzione (controllo)
- **EX** Esecuzione istruzione o calcolo indirizzo (ALU)
- **MEM** Accesso (lettura o scrittura) ad un dato in memoria
- **WB** Scrittura risultato *in un registro*

Pipeline 5/

Pseudo codice

C = A + 20
D = C - 1

Pipeline 6/

La prima istruzione deve prelevare A (dalla memoria per poi scriverla in un registro: MEM e WB) per poi sommarlo a 20 (EX) e scrivere il risultato (WB) in C.

La seconda istruzione sottrae a C (scritto in un registro) il valore 1 (EX) e lo va a scrivere in D (WB).

IF ID EX(____) MEM(A) WB(A)

Pipeline 6/

La prima istruzione deve prelevare A (dalla memoria per poi scriverla in un registro: MEM e WB) per poi sommarlo a 20 (EX) e scrivere il risultato (WB) in C.

La seconda istruzione sottrae a C (scritto in un registro) il valore 1 (EX) e lo va a scrivere in D (WB).

IF ID EX(____) MEM(A) WB(A)

IF ID EX(20+A) MEM(_) WB(C)

Pipeline 6/

La prima istruzione deve prelevare A (dalla memoria per poi scriverla in un registro: MEM e WB) per poi sommarlo a 20 (EX) e scrivere il risultato (WB) in C.

La seconda istruzione sottrae a C (scritto in un registro) il valore 1 (EX) e lo va a scrivere in D (WB).

```
IF ID EX(____) MEM(A)       WB(A)
   IF  ID       EX(20+A)    MEM(_)  WB(C)
       IF       ID          EX(C-1) Fault! Data Hazard
```

Pipeline 7/

La CPU è in fault! Cosa può fare?

- Inserire una “bolla” NOP
- Qualcosa che vedremo più avanti

IF ID EX(____) MEM(A) WB(A)

Pipeline 7/

La CPU è in fault! Cosa può fare?

- Inserire una “bolla” NOP
- Qualcosa che vedremo più avanti

IF ID EX(____) MEM(A) WB(A)

IF ID EX(20+A) MEM(_) WB(C)

Pipeline 7/

La CPU è in fault! Cosa può fare?

- Inserire una “bolla” NOP
- Qualcosa che vedremo più avanti

IF ID EX(____) MEM(A) WB(A)

IF ID EX(20+A) MEM(_) WB(C)

IF ID EX(____) MEM(_) WB(_)

Pipeline 7/

La CPU è in fault! Cosa può fare?

- Inserire una “bolla” NOP
- Qualcosa che vedremo più avanti

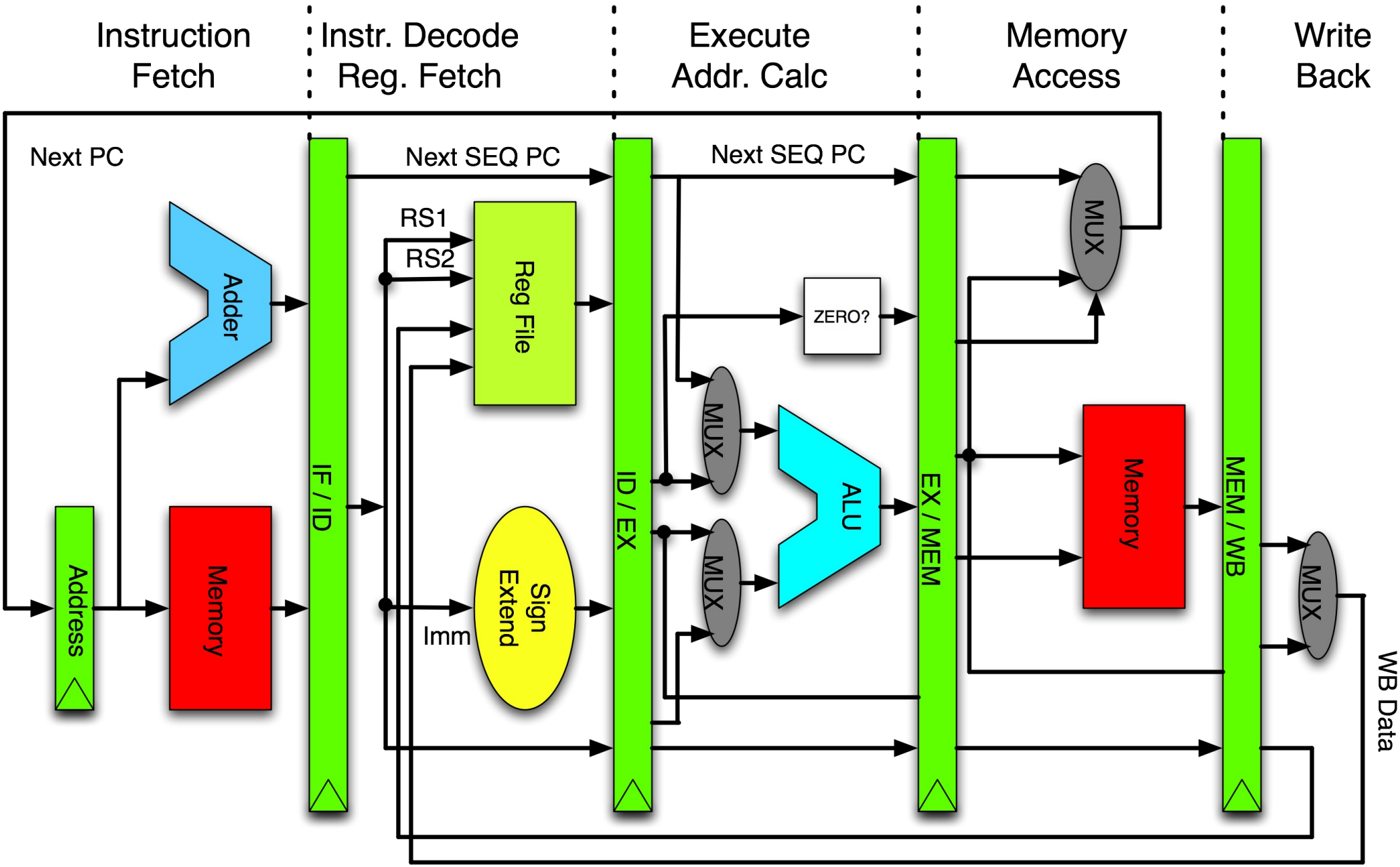
```
IF ID EX(____) MEM(A)       WB(A)
   IF  ID       EX(20+A)    MEM(_)  WB(C)
       IF       ID          EX(____) MEM(_) WB(_)
                IF          ID       EX(C-1) MEM(_) WB(D)
```

Pipeline 8/

La bolla fa andare un ciclo a vuoto ma previene un disastro.

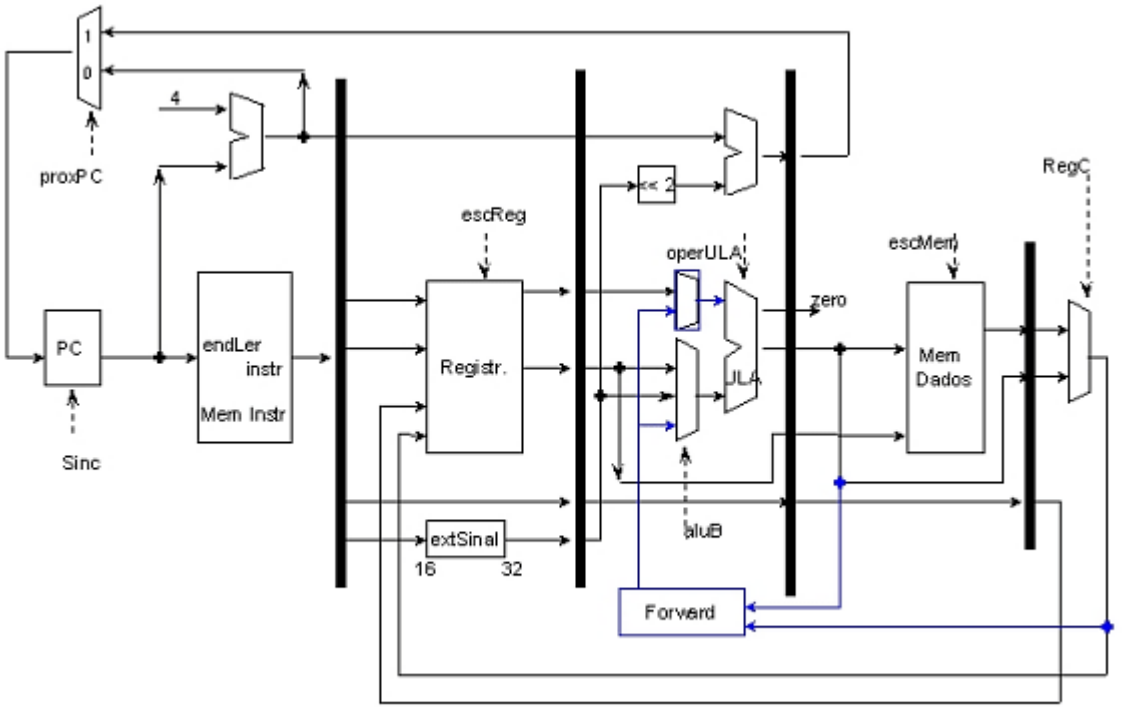
Nel nostro esempio è necessaria.

Per i più attenti ..... vi torna l'analogia con i panni che non sono stirabili se sono ancora nell'asciugatore?


Instruction Fetch
Instr. Decode Reg. Fetch
Execute Addr. Calc
Memory Access
Write Back
Next PC
Next SEQ PC
Next SEQ PC
RS1
RS2
Reg File
Adder
IF / ID
ID / EX
EX / MEM
MEM / WB
ZERO?
MUX
MUX
MUX
MUX
ALU
Memory
Memory
Address
Sign Extend
Imm
WB Data

Pipeline 9/

Il trucco della bolla funziona perché c'è un *forward* tra gli stage MEM ed EX .... altrimenti sarebbero servite **due** bolle.



Anexo 1 - (As linhas mais grossas são os registradores necessários para um pipeline de cinco estágios)

Lo statement if–then–else

Il blocco *if–then–else* potrebbe mandare a vuoto qualche ciclo di *pipeline* svuotando il processore delle istruzioni.

La decisione di quale ramo prendere avviene solo nello stadio EX.

Vengono quindi caricate (IF) e decodificate (ID) due istruzioni prima della decisione.

Queste istruzioni potrebbero non venire eseguite a causa della guardia dello *statement*.

Devono quindi essere rimosse (lavatrice a vuoto) e ne devono essere inserite altre.

Tecniche utili

Per non fare inutili cicli di *clock* esistono tecniche di *scheduling* e di predizione dei salti

- *Prefetch* (*multiple fetch*)
- *Branch prediction unit*

La memoria

Nei moderni elaboratori non esiste un'unica (grossa e lenta) memoria centrale (RAM), esistono infatti varie gerarchie di memoria (sempre più piccole e sempre più veloci).

La biblioteca

Una biblioteca ben fornita e ben organizzata dispone di vari titoli suddivisi per argomento, libri appartenenti al solito argomento stanno nel medesimo scaffale (o in zone adiacenti).

Lo studioso

Lo studioso si reca in biblioteca per consultare più libri appartenenti ad uno o più argomenti collegati.

La ricerca

- Lo studioso si posizione su di un tavolo libero
- Prende alcuni libri dallo scaffale
- Studia
- Prende altri libri
  - Se non ha più posto sul tavolo ripone i volumi meno utili sugli scaffali
  - E' probabile che consulti libri di argomenti uguali a qualcuno già presente sul tavolo

Principi di località

- Località *temporale*
  - Se viene consultato un libro sullo scaffale, è probabile che venga presto ri–consultato
- Località *spaziale*
  - Se viene preso dallo scaffale un libro di una certa materia è probabile che entro breve verranno consultati anche altri libri della stessa materia
  - *Finito lo studio di una materia nessun libro della stessa materia sarà più utile*

Il tavolo e gli scaffali

- Lo scaffale è ben fornito ma si impiega molto tempo per accedervi e per ricercare i volumi.
- Il tavolo ha poco spazio ma i libri vengono consultati velocemente.

L'uso del tavolo velocizza le operazioni.

Gerarchie di tavoli

Cosa accadrebbe se a disposizione dello studioso ci fossero gli scaffali, un banco piuttosto grosso ed il tavolo?

- Si limiterebbe l'accesso agli scaffali (utilizzati prevalentemente nella fase iniziale)
- Si farebbe un poco di confusione su "*dove stanno i libri*"

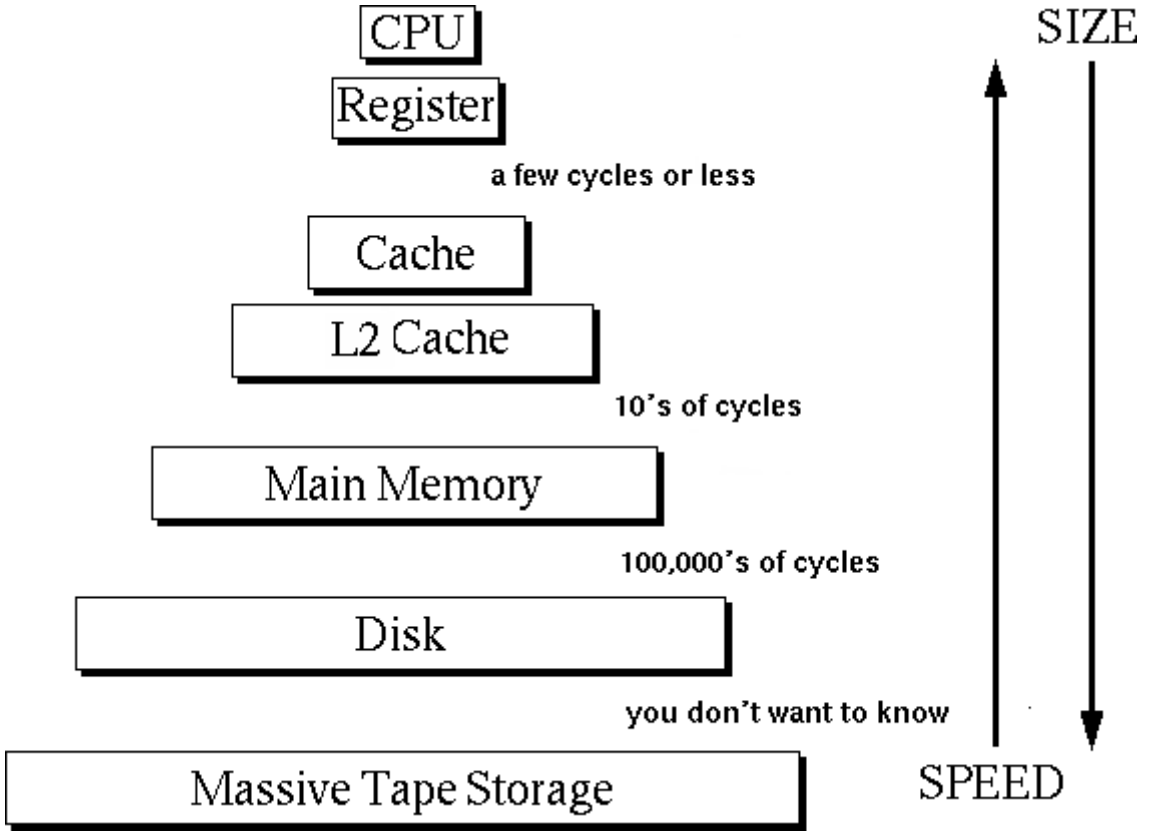
Memorie cache 1/

I principi di località spaziale e temporale e la necessità di avere (l'illusione di) una memoria veloce ed illimitata hanno portato all'adozione di memorie *cache*.

Nota: la *cache* non è proprio come il tavolo, infatti i dati non vengono tolti dalla memoria ma ne viene fatta una copia all'interno della *cache*.

Inoltre nessuno oserebbe scrivere su un libro di proprietà della biblioteca (vero?) mentre sulla memoria .....

Memorie cache 2/



Memorie cache 3/

L'utilizzo delle *cache* ha contribuito ad un innegabile aumento delle *performance* nei moderni elaboratori.

Si è tuttavia perso di vista “dove stanno i dati”

- la *cache* può non essere *coerente* con la memoria
- più applicazioni leggono e scrivono (apparentemente) in memoria
- chi è più aggiornato tra la memoria e la *cache*?

Domande?

Pausa

Cenni storici ....

Anticamente i computer non avevano il sistema operativo.

Le istruzioni venivano eseguite ***in sequenza*** dalla ***prima*** all'***ultima*** senza ***interruzioni***.

I *programmi* avevano accesso a tutte le risorse della macchina.

Nota

- ***in sequenza***
- ***prima***
- ***ultima***
- ***interruzioni***

Definizione

Sistema operativo è il programma responsabile del diretto controllo e gestione dell'hardware che costituisce un computer e delle operazioni di base. Si occupa dei processi che vengono eseguiti e della gestione degli accessi degli utenti. Compito del sistema operativo è inoltre quello di virtualizzare le risorse hardware e software nei confronti dei programmi applicativi.

Preso da wikipedia

Problemi riscontrati

Scrivere programmi che giravano sul *nudo metallo* era complicato.

Eseguire un programma alla volta era inefficiente.

L'avvento dei sistemi operativi

Esecuzione di più programmi alla volta.

Esecuzione di programmi in *processi*.

Allocazione di risorse (memoria, descrittori di file e credenziali utente) a processi.

Definizione

*Un processo è l'esecuzione di un programma (o sottoprogramma) sequenziale. Lo **stato** di un processo ad un certo istante nel tempo consiste di tutte le sue variabili **esplicite** dichiarate dal programmatore e da quelle **implicite** dettate dalla piattaforma hardware e dal sistema operativo. Via via che il processo va avanti questo trasforma il suo stato attraverso l'esecuzione di **statement**; ovvero sequenze di una o più istruzioni atomiche.*

Da "Concurrency: state models and java programming"

Riflessione

- Stato
- Implicito
- Esplicito
- Statement

Comunicazione tra processi

Il sistema operativo mette a disposizione dei meccanismi di comunicazione tra processi (IPC).

- PIPE
- Socket
- Memoria *condivisa*
- Descrittori di file
- Semafori
- .....

Esecuzione simultanea di più processi

- Utilizzo di risorse
- *Fairness*
- Convenienza

Utilizzo di risorse

Spesso i programmi devono ***attendere*** il completamento di operazioni esterne come ad esempio la lettura di dati da disco e mentre attendono non possono dare ***niente di utile***.

Conviene utilizzare questo tempo di attesa per far lavorare un *altro* programma.

Fairness (imparzialità)

Più programmi (e utenti) richiedono un utilizzo *equo* delle risorse del sistema.

Meglio adottare una fine politica di *timeslicing* che far eseguire un solo programma alla volta.

Convenienza

E' più semplice scrivere più programmi che eseguono compiti distinti e poi coordinarli che scrivere un unico *programmone* che esegue tutti i compiti,

Schedulazione dei processi

Dovendo gestire l'esecuzione di più processi il sistema operativo deve scegliere di volta in volta *quale* processo inviare alla CPU.

Se un processo schedulato rimane in attesa di una risorsa di I/O il sistema operativo deve rimuoverlo dalla CPU e mandarne in esecuzione un altro.

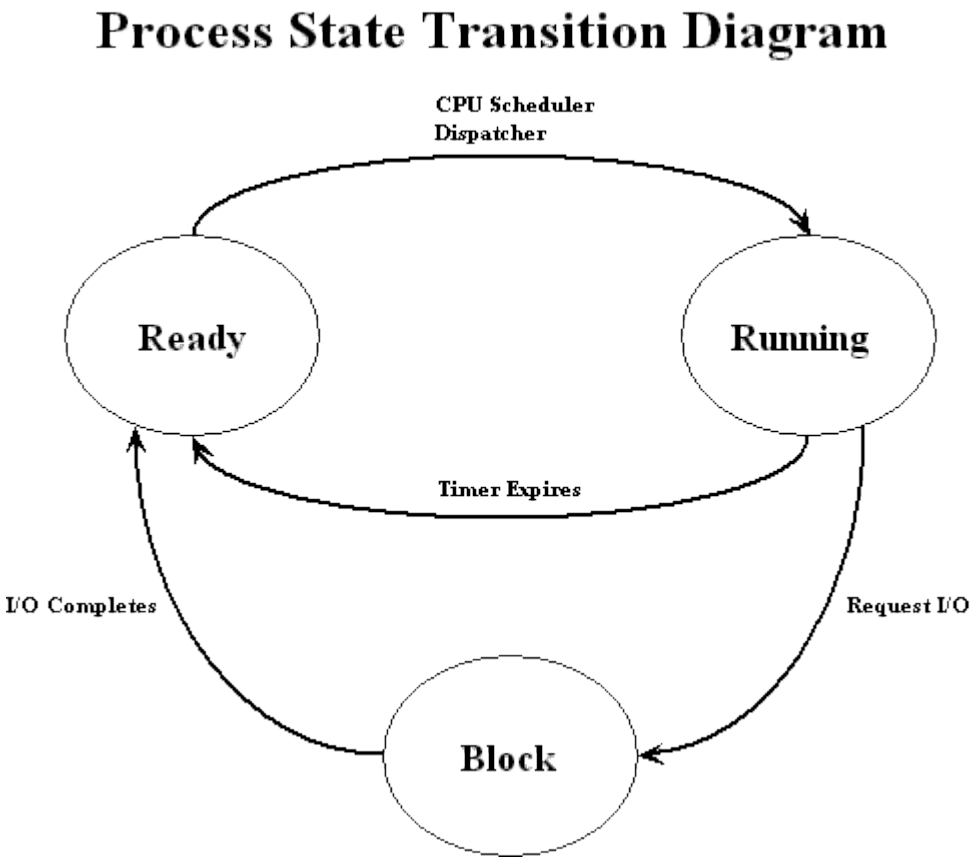
Definizione

Lo ***scheduler*** è quella componente dei sistemi operativi che si occupa di fare avanzare un processo interrompendone temporaneamente un altro, realizzando così un cambiamento di contesto (*context switch*).

Generalmente computer con un processore sono in grado di eseguire un programma per volta, quindi per poter far convivere più task è necessario usare lo ***scheduler***.

Preso da: Wikipedia

Diagramma a stati di un processo



Lo scheduling dei processi 1/

- Difficilmente si ha una politica *First Come First Served*
- I processi possono avere delle priorità
  - Procedono per primi quelli con priorità maggiore
  - Quelli con priorità minore possono rimanere bloccati per sempre (*starvation*)
    - Esistono tecniche di *invecchiamento* che aumentano la priorità dei processi
    - Non è quindi noto staticamente l'ordine di esecuzione e completamento dei processi

Lo scheduling dei processi 2/

- Lo *scheduler* può rimuovere un processo dalla CPU anche se questo non è in attesa di un I/O (*preemption*)
- Tecniche molto utilizzate di *scheduling* sono
  - **Round Robin**: a rotazione a tutti i processi attivi viene assegnato un tempo ci CPU terminato il quale il processo viene rischedulato. Implica la *preemption*.
  - **Shortest Job First**: passa per primo il processo "più corto". Può adottare la *preemption*.

Lo scheduling dei processi 3/

- Il *context-switch* è un'operazione *costosa*:
  - descrittori di file
  - grossi blocchi di memoria
  - stato del processo
  - memoria *heap*
  - .....

tutto deve essere salvato (su disco) per poi essere ripristinato quando necessario.

Pausa

Domande?

Ai bei tempi ...

Nei primi sistemi *timesharing* ogni processo era una distinta macchina di Von Neumann virtuale:

- Memoria ***privata*** (dati + istruzioni)
- Esecuzione ***sequenziale*** delle istruzioni (in accordo alla semantica del linguaggio macchina)
- Interazione col mondo esterno tramite le primitive di I/O fornite dal sistema

Sequenzialità

Per ogni istruzione eseguita era ben definita la prossima istruzione.

Anche oggi nei linguaggi moderni la situazione è simile.

Le ***specifiche*** del linguaggio hanno un concetto diverso di “cosa accade dopo” che un'istruzione viene eseguita.

La sequenzialità è naturale e intuitiva

- Alzati
- Vestiti
- Fatti un toast
- Esci di casa

La sequenzialità è un'astrazione

Ciascuno dei passi precedenti è un'astrazione di un passo più complesso.

Fatti un toast:

- Prendi il pane
- Prendi il companatico
- Accendi il tostapane
- Inserisci il toast
- ***Attendi*** che sia pronto

Asincronia 1/

L'ultimo passo concerne un certo livello di asincronia.

Mentre il toast si sta scaldando è possibile fare altro:

- Attendere
- Guardare la televisione (rimanendo consapevoli che prima o poi sarà necessario prestare attenzione al tostapane)

Asincronia 2/

I costruttori di tostapane sanno che i loro dispositivi verranno usati in maniera asincrona.

I tostapane dispongono di un segnalatore acustico che avverte quando il toast è pronto.

Asincronia 3/

Le persone efficienti trovano il giusto bilanciamento tra sequenzialità ed asincronia (guardate la televisione mentre si attende il toast).

Lo stesso si applica ai programmi.

Thread 1/

Utilizzo delle risorse, *fairness* e convenienza hanno portato alla nascita dei Thread.

- Permettono l'esecuzione di più flussi di programma all'interno di uno stesso processo.
- Condividono memoria e descrittori di file.
- Hanno tuttavia *program counter*, *stack* e variabili locali *propri*.
- Più Thread dello stesso programma possono essere eseguiti in contemporanea su diverse CPU.

Thread 2/

I Thread vengono spesso chiamati *processi leggeri* e i moderni sistemi operativi utilizzano i Thread al posto dei processi come unità base per l'esecuzione dei programmi.

Thread :-(1/

In assenza di coordinamento esplicito i Thread vengono eseguiti in maniera asincrona ed indipendente dagli altri.

Thread :–(2 /

Poiché condividono la memoria del loro processo padre, tutti i Thread appartenenti allo stesso processo hanno accesso alle solite variabili e possono allocare i dati sullo stesso *heap*!

Questo è un altro meccanismo di IPC.

Thread :-(3 /

Senza *sincronizzazione esplicita* per il coordinamento dell'accesso concorrente alle risorse un Thread può modificare un valore *mentre* un altro Thread lo sta usando.

Questo porta a risultati *imprevedibili*,

Benefici dei Thread 1/

Quando utilizzati in maniera propria i Thread riducono i costi di sviluppo e manutenzione e migliorano le *performance* degli applicativi.

Modellano in maniera semplice il comportamento degli esseri umani trasformando *task* asincroni in *task* perlopiù sequenziali.

Benefici dei Thread 2/

Utilizzo delle risorse

- I computer moderni dispongono di più CPU
- Su una sola CPU può girare un solo Thread alla volta
- In un un sistema con due CPU un programma mono Thread occupa solo il 50% delle risorse
- In un un sistema con 100 CPU un programma mono Thread occupa solo l'1% delle risorse
- Un programma mono Thread lascia inutilizzata la CPU mentre è in attesa di un evento asincrono di I/O

Benefici dei Thread 3/

Cambi di contesto non devastanti

- Copia delle variabili locali
- Copia dello stato della CPU
  - Registri
  - PC
  - .....
- *Stack*

nessuno *swap* di imponenti aree di memoria.

Problematiche dei Thread

- Sicurezza
- Vitalità
- *Performance*

Sicurezza

Condividendo lo stesso *heap* si possono generare interferenze tra Thread lettori e Thread scrittori, la condivisione senza controllo è un fatto negativo.

E' necessario un meccanismo di *sincronizzazione* tra Thread che accedono a dati condivisi.

Vitalità

Ha a che fare con questioni del tipo "*prima o poi* qualcosa di buono accade".

Un programma è *vivo* se non giunge mai in uno stato entrato nel quale non può più fare progressi.

Performance

La vitalità ha a che fare con *questioni* del tipo "*prima o poi* qualcosa di buono accade".

Ma "*prima o poi*" potrebbe essere un tempo piuttosto lungo, la *performance* vuole che le cose buone *accadano velocemente*.

Pausa

Domande?

Java 1/

Mix tra linguaggio *compilato* ed *interpretato*.

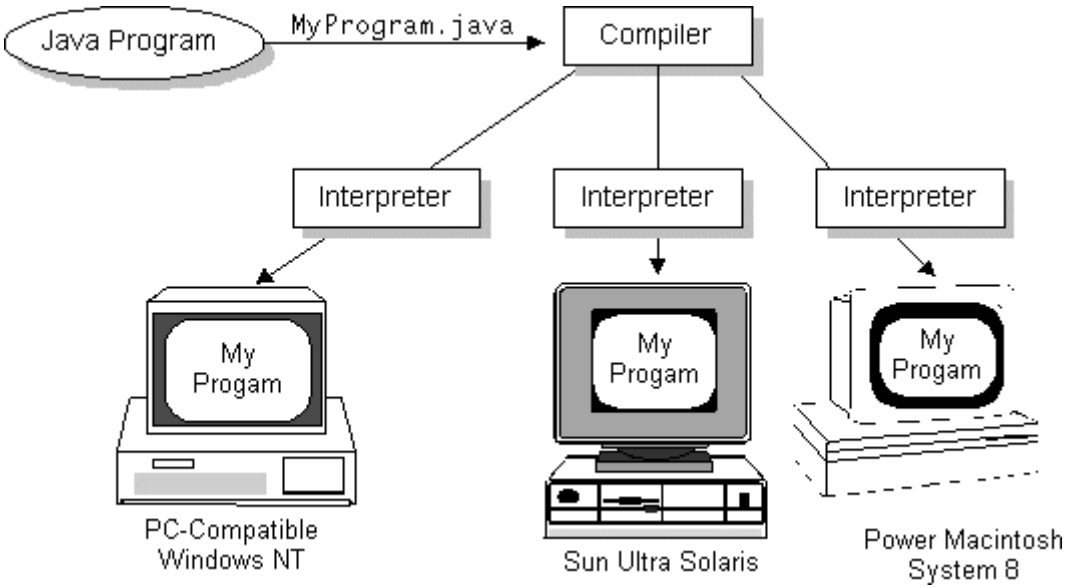
Java 2/

Il codice sorgente (.java) viene *compilato* in un file *bytecode* (o più file) che viene *interpretato* dalla *Java Virtual Machine*.

Java 3/

Il *bytecode* può essere interpretato su piattaforme diverse a patto che dispongano di una adatta JVM.

Java 4/


Java Program
MyProgram.java
Compiler
Interpreter
Interpreter
Interpreter
My Progam
My Progam
My Progam
PC-Compatible
Windows NT
Sun Ultra Solaris
Power Macintosh
System 8

Java 5/

Il fatto che Java “giri” su piattaforme differenti e mantenga su di esse lo stesso comportamento (almeno in teoria) influisce su alcuni meccanismi riguardanti i Thread, la memoria e le interazioni tra questi.

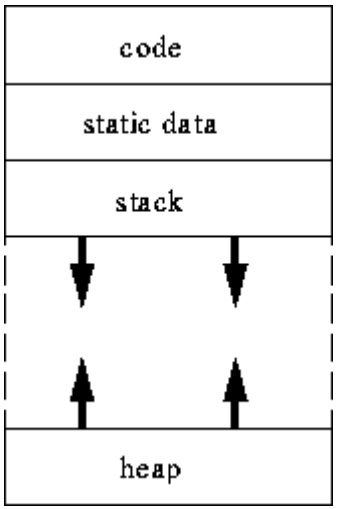
Java 6/

La JVM “simula” una piattaforma *hardware* sulla quale gira il *bytecode*, le regole della JVM possono essere assai diverse da quelle della piattaforma reale e soprattutto possono essere non intuitive.

Allocazione della memoria

Rozzamente la memoria di un elaboratore viene divisa in due parti:

- *text segment*
- *data segment*
  - *heap*
  - *stack*



Text segment

Contiene le istruzioni che devono essere eseguite.

In Java viene chiamato *Method Area*.

Stack segment 1/

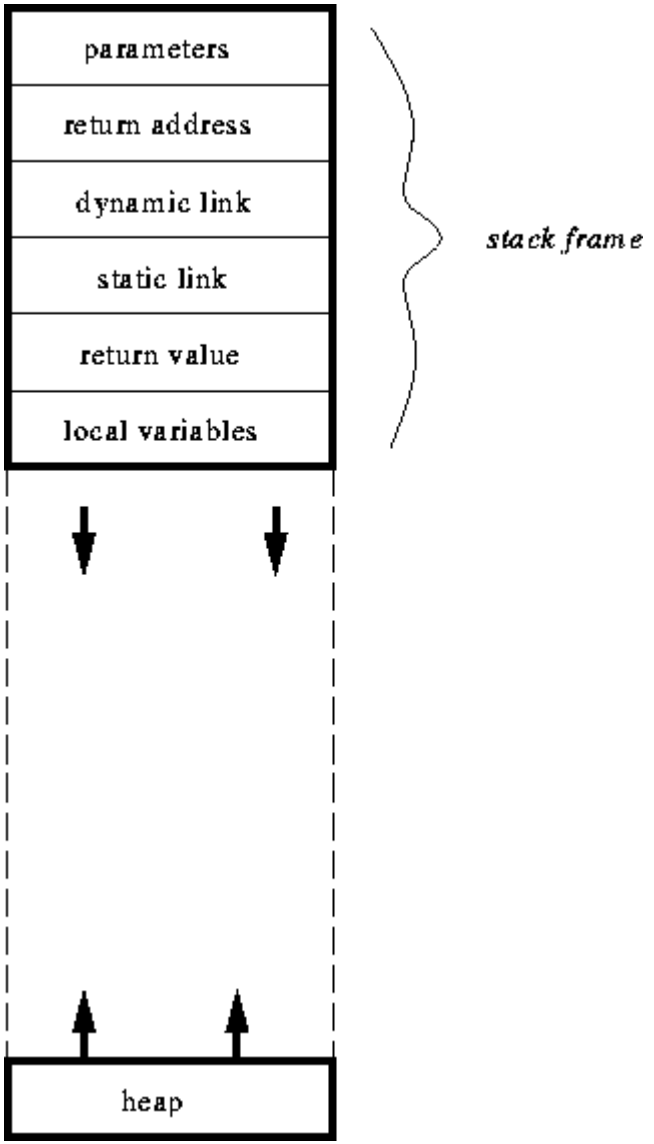
Contiene le variabili locali, risultati intermedi ed i parametri dei metodi.

Ogni Thread Java ha il suo proprio *stack*.

Stack segment 2/

Può essere *espandibile* come dimensione, è fatto come una pila.

- Se un Thread richiede uno *stack* più grande di quanto permesso viene lanciata una **StackOverflowError**
- Se l'espansione dinamica di uno *stack* "finisce" la memoria viene lanciata una **OutOfMemoryError**



Heap Segment

Ci vengono allocate le istanze di tutte le classi e gli array.

Ne esiste uno solo per tutta la JVM, è quindi condiviso da tutti i Thread.

Se viene richiesto più spazio di *heap* di quanto non sia disponibile viene lanciata una *OutOfMemoryError*.

Heap vs Stack

Variabili:

- locali per lo *stack*
- condivise per lo *heap*

Condivisione:

- un solo *heap* per tutta la JVM
- uno *stack* per ogni Thread

Heap & Stack

Attenzione: lo *stack* può contenere il *reference* ad un oggetto allocato (e quindi potenzialmente condiviso) sullo *heap*.

Stack e ricorsione

Attenzione, nella scrittura di un metodo ricorsivo è bene prediligere la versione “ricorsiva di coda” o se possibile quella iterativa.

Ogni chiamata di metodo richiede l'allocazione di una certa area dello *stack*, le chiamate ricorsive tipo *fibonacci* possono generare velocemente **StackOverflowError**.

Parte I

Concetti introduttivi

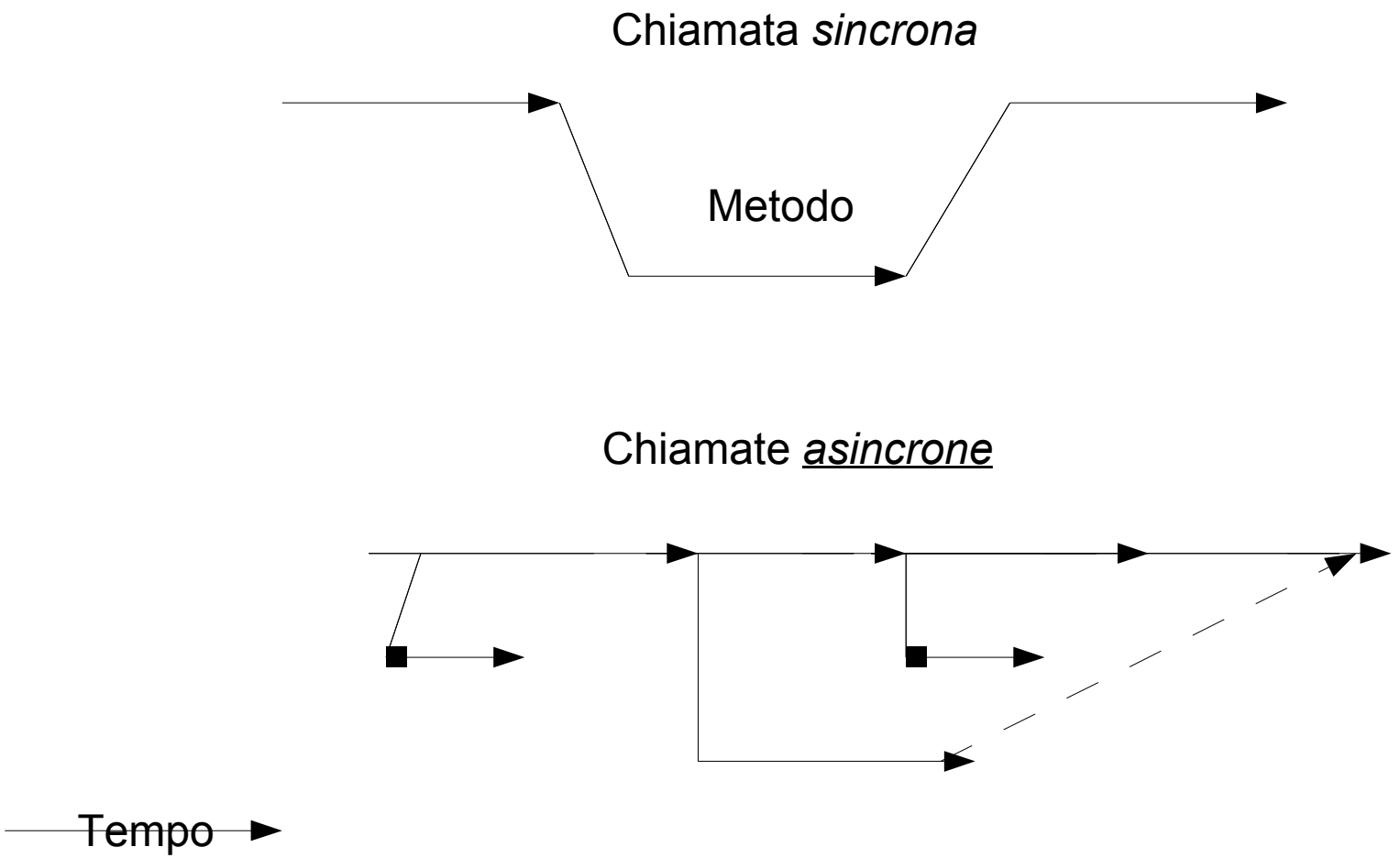
Argomenti trattati

- Thread
- Race condition e sincronizzazione
- Java Memory Model
- Framework

Eseguire task in parallelo?

Si eseguono *task* in “parallelo” per migliorare, quando possibile, la performance degli applicativi; performance intesa anche come reattività del sistema.

Esempio di chiamate



Risorse usate nelle computazioni

- CPU
- Memoria
- I/O

Quando il processore è in attesa di una risposta da un I/O (o dalla memoria) può fare altre cose, ad esempio sospendendo il *task* corrente ed eseguendone un altro.

Processi e Thread

Sono unità di computazione che vengono eseguite in maniera asincrona ed indipendente.

Per eseguire una chiamata asincrona è necessario "lanciare" un processo o un Thread.

Thread Java

Estendono la classe java.lang.Thread.

Non possono estendere altre classi!

Implementano l'interfaccia java.lang.Runnable.

L'esecuzione di un Runnable avviene tramite un l'istanziazione di un Thread.

Thread java: esempi

```
public class SimpleThread  extends Thread{
        private final String myName ;
        public SimpleThread(String s){this.myName = s;}
        public void run() {
                long l = 0 ;
                while( l < 50) {
                        System.out.println("CIAO " + myName);
                        try {
                        sleep( (long) (100 * Math.random() ));
                        } catch (InterruptedException ie) {
                                return;
                        }
                        l++;
                }
        }
        public static void main(String[] args) {
                SimpleThread s = new SimpleThread("S");
                SimpleThread t = new SimpleThread("T");
                s.start();
                t.start();
                System.out.println("Partito");
        }
}
```

Runnable: esempio

```
public class SimpleRunnable implements Runnable {
	private final int foo;
	public SimpleRunnable(int id){this.foo = id;}
	public void run() {
		long l = 0 ;
		while (l < 100){
			System.out.println("Thread " + foo + " ciclo " + l);
			try {
				Thread.sleep(100);
			} catch (InterruptedException e) {
				return;
			}
			l++;
		}
	}
	public static void main(String[] args) {
		int no = (int)( Math.random() * 30  );
		Thread[] t = new Thread[no];
		int i = 0 ;
		for (i = 0 ; i < no ; i++){
			t[i] = new Thread(
					new SimpleRunnable(i)
			);
		}
		for (i = 0 ; i < no ; i++){	t[i].start();	}
	}
}
```

Thread: alcuni metodi

***start*()**: fa partire il Thread DA NON SOVRASCRIVERE

run(): codice asincrono che viene eseguito dal Thread

sleep(millis): mette a dormire il Thread

join(): attende che il Thread abbia completato il metodo *run()*

yeld(): suggerisce alla JVM di sospendere il Thread corrente se ci sono altri Thread attivi

isAlive(): indica se il Thread sta è vivo o meno

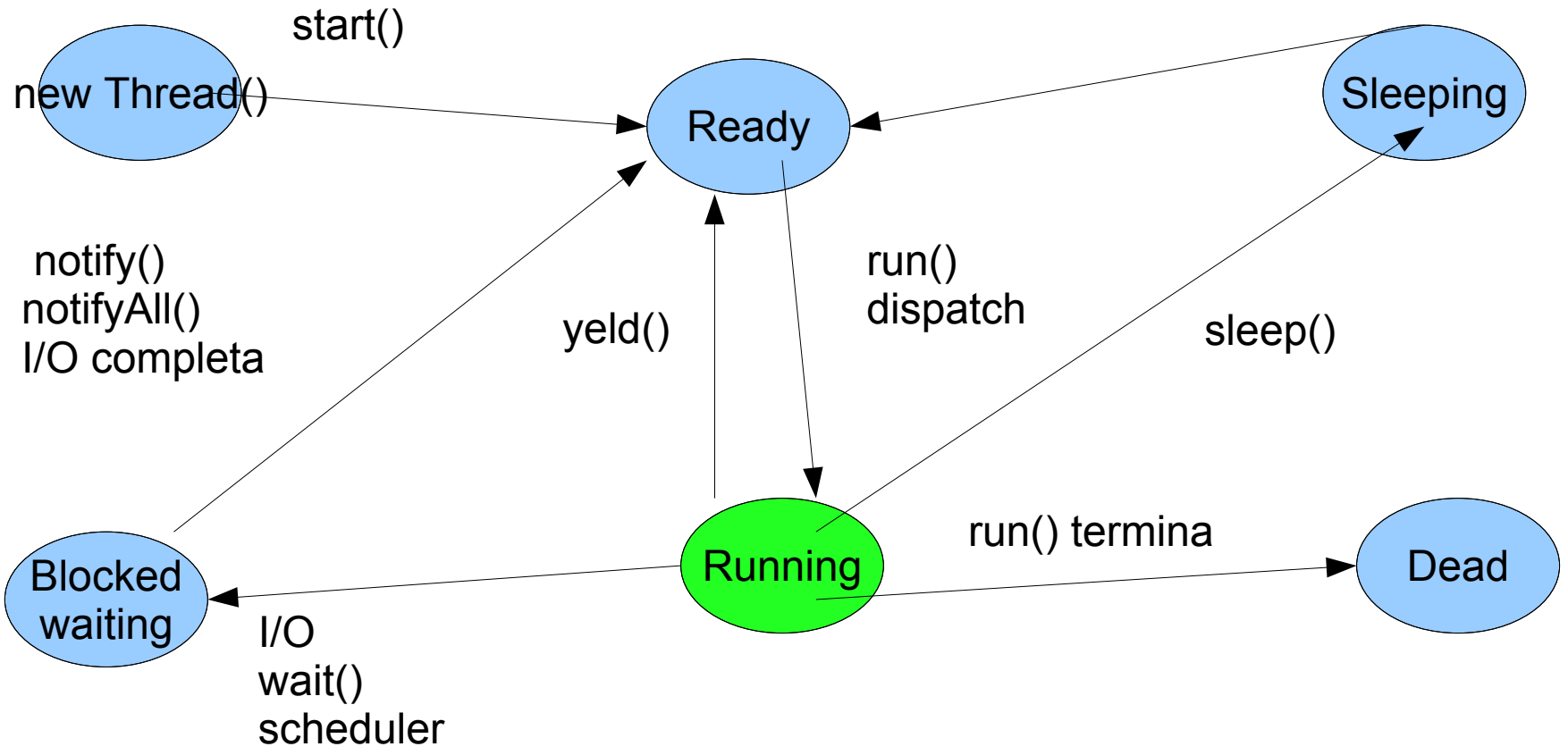
stop(), *suspend()* e *resume()* sono stati deprecati

Nota

Nonostante il Thread parta con l'invocazione del metodo *start()* il codice da eseguire deve essere scritto nel metodo *run()*.

Il primo metodo infatti invoca metodi nativi si sistema i quali quando opportuno invocheranno il metodo *run()*.

Thread: funzionamento di massima



Ancora sullo start

Il metodo start() può essere invocato una sola volta, non esistono eccezioni.

Un Thread si può trovare quindi una sola volta nello stato di partenza.

Esecuzione di più start

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

SimpleThread t = new SimpleThread("Prova");

t.start();

t.start();

}
```

Cosa accade?

Esecuzione di più start

```
public static void main(String[] args) {
SimpleThread t = new SimpleThread("Prova");
t.start();
t.start();
}
```

Cosa accade?

```
Exception in thread "main" java.lang.IllegalThreadStateException
at java.lang.Thread.start(Thread.java:589)
at jtt2007.ch1.thread.MultiStart.main(MultiStart.java:11)
```

Esecuzione di più run

E' tuttavia possibile invocare più volte il metodo *run()*.

Il metodo verrà eseguite nel Thread corrente e non in uno separato, sarà quindi puro codice sequenziale.

Esempio di run multiple

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

SimpleThread t = new SimpleThread("Prova");

t.start();

t.run();

t.run();

}
```

Gli ultimi due *run()* vengono eseguiti in sequenza dallo stesso Thread del metodo *main()*.

Funzionamento del join

Si utilizza il metodo T.join() *per* attendere il completamento del Thread T prima di proseguire.

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

  SimpleThread s = new SimpleThread("Primo");

  SimpleThread t = new SimpleThread("Secondo");

  s.start();

  t.start();

  System.out.println("Partiti");

  s.join();

  t.join();

  System.out.println("Esecuzione terminata");

}
```

Output del programma

La stringa “Esecuzione terminata” non verrà stampata fino a quando i due Thread non avranno terminato il metodo *run()*.

Esempio di utilizzo di IsAlive

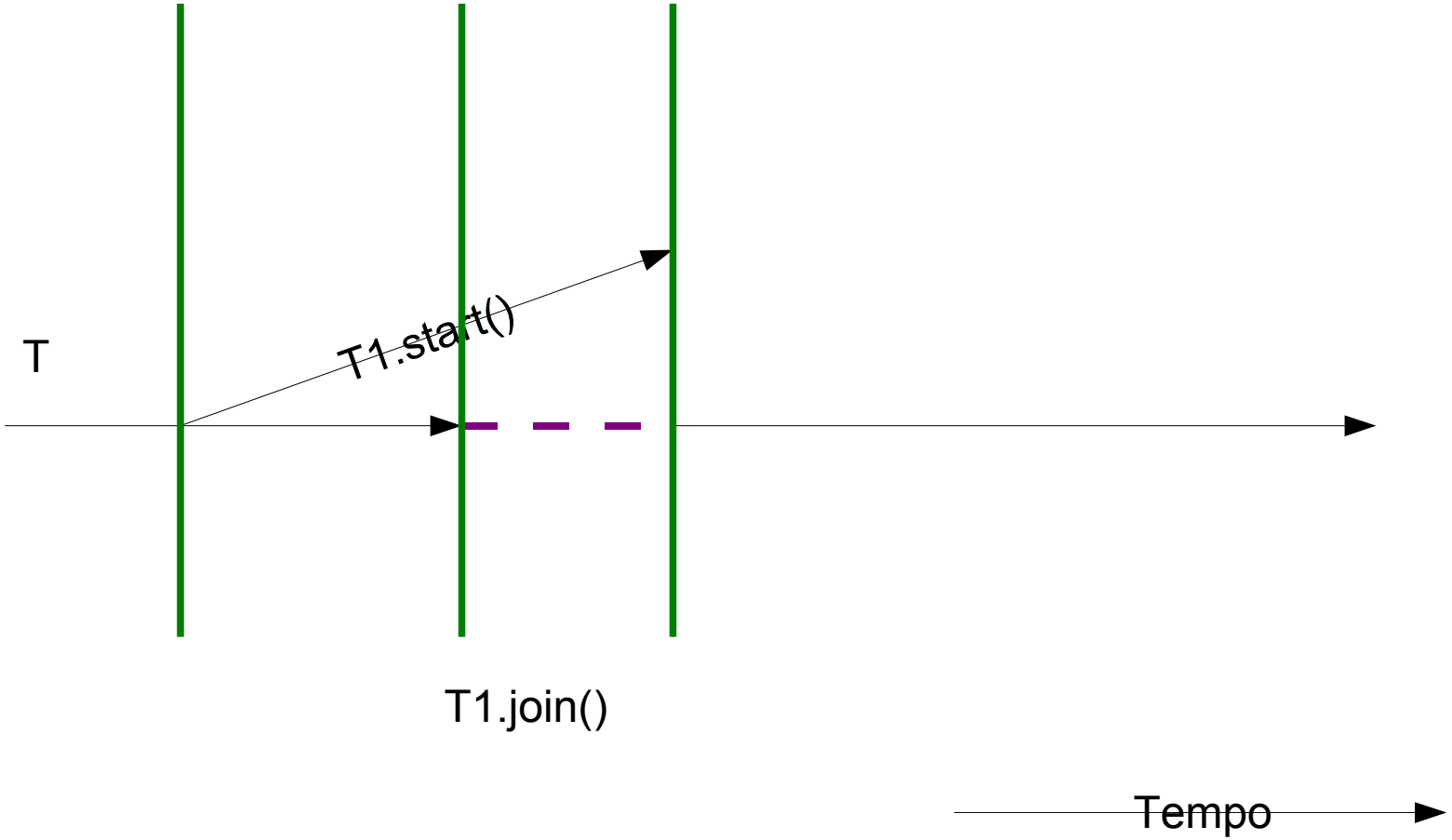
```
public static void main(String[] args) throws InterruptedException {
String nome = "MioThread";
SimpleThread s = new SimpleThread(nome);
System.out.println("Thread " + nome + " non paritto, isalive:"+ s.isAlive());
s.start();
System.out.println("Thread " + nome + " non in running, isalive:"+ s.isAlive());
s.join();
System.out.println("Thread " + nome + " terminato, isalive:"+ s.isAlive());
}
```

Start e Join 1/

I metodi *start()* e *join()* rivestono due ruoli complementari:

- sgancio dal Thread di controllo
- ricongiungimento al Thread di controllo

Start e Join 2/



Pausa

Domande?

Problema: Race Condition

Più Thread possono voler accedere contemporaneamente alla stessa area di memoria (variabile Java).

Attenzione! Le *race condition* si verificano quando uno meno se lo aspetta.

Non sono simulabili a piacimento

Problema: Race Condition

Si definiscono *race condition* tutte quelle situazioni in cui processi diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall'ordine in cui essi effettuano le loro operazioni.

Preso da: http://www.lilik.it/~mirko/gapil/gapil.html

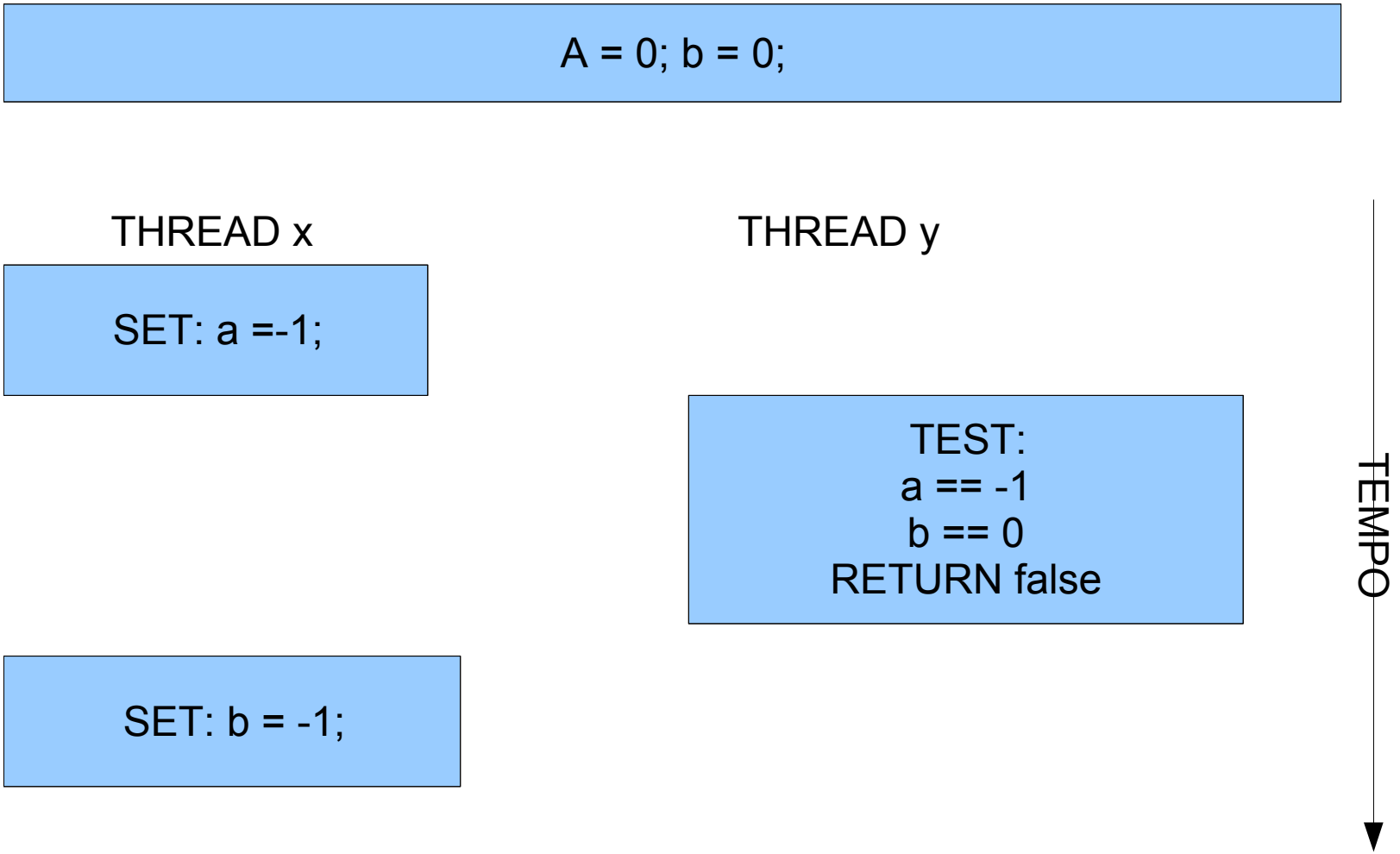
A race condition or race hazard is a flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first

Preso da: Wikipedia

Esempio di race condition

```
public class SimpleRace {
	private int a = 0 ;
	private long b = 0;
	public void set(){
		a = -1;
		b = -1;
	}
//dovrebbe sempre ritornare true (in un ambiente sequenziale)
	public boolean test() {
		if (a == 0 || (  a == -1 && b == -1))
			return true;
		else return false;
	}
}
```

Dove sta il problema?



(Una banale) Soluzione alla race condition

```
public class SimpleNoRace {
	private int a = 0 ;
	private long b = 0;
	public synchronized  void set(){
		a = -1;
		b = -1;
	}
//ritorna sempre true
	public synchronized  boolean test() {
		if (a == 0 || (  a == -1 && b == -1))
			return true;
		else return false;
	}
}
```

Synchronized???

Ogni oggetto Java ha associatoun *lock* (mutex) intrinseco.

Questo a richiesta viene acquisito da un Thread richiedente in mutua esclusione.

L'accesso al blocco synchronized è garantito esclusivamente al Thread che ne detiene il *lock* associato.

Un Thread in possesso di un *lock* può richiedere questo più volte.

Ma quale *lock* viene acquisito?????

Blocchi sincronizzati

```
....
synchronized(pippo) {
    ..........
}
```

Il *lock* acquisito è quello *intrinseco* dell'**oggetto** pippo

Metodi sincronizzati di classe

```
Public synchronized foo() {
    CODICE
}
```

È equivalente a

```
Public foo(){
    synchronized(this) {
        CODICE
    }
}
```

Metodi statici sincronizzati

```
Public class Bar {
    public synchronized static foo(){ CODICE}
}
```

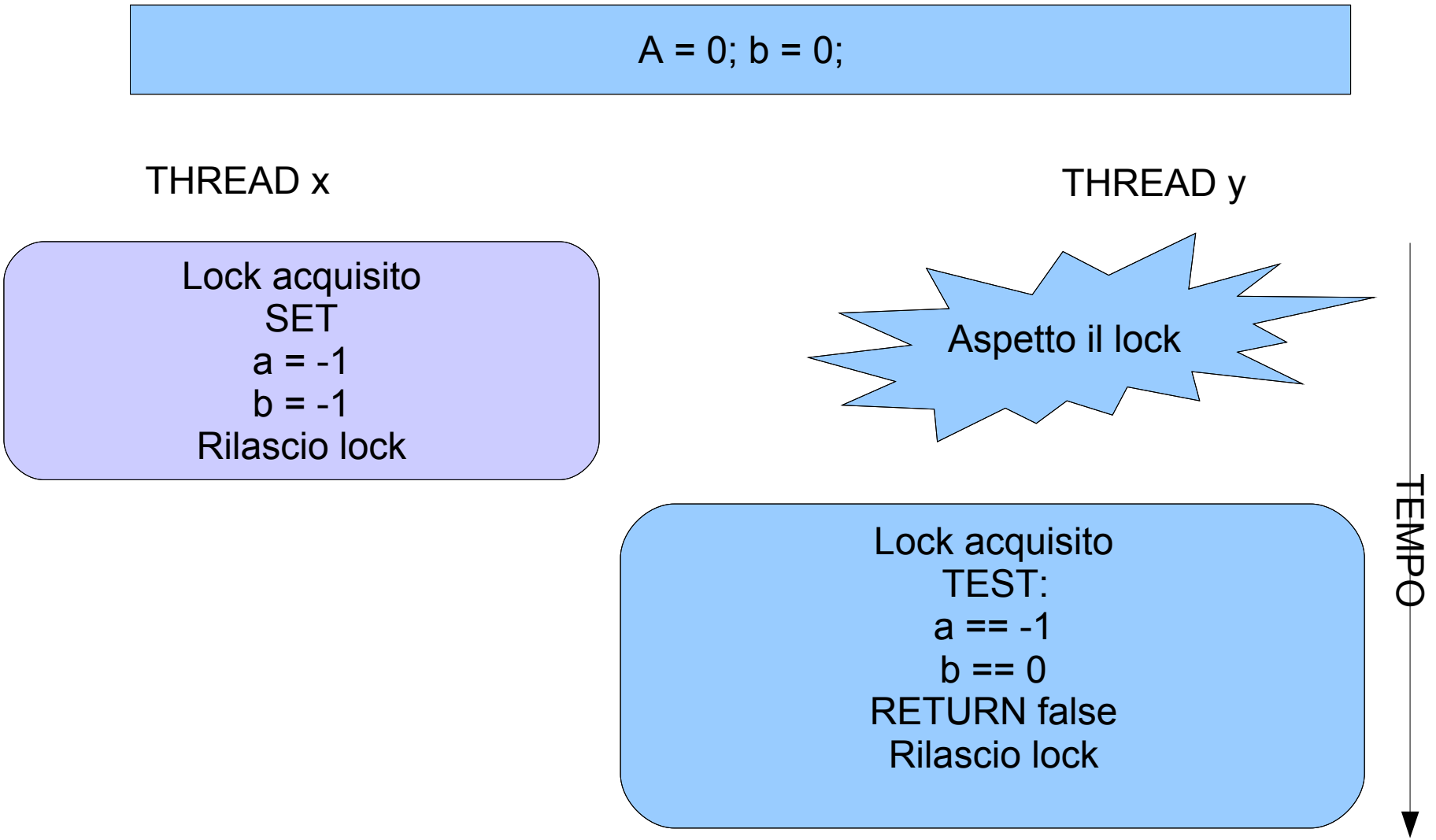
È equivalente a

```
Public class Bar {
    public  static foo(){
        synchronized(Bar.class)     {
                CODICE
        }
    }
}
```

Quindi ...

D'ora in poi verrà utilizzato solo il termine “blocco”.

Esecuzione con blocco sincronizzato



Implicazioni sull'accesso esclusivo

Un blocco sincronizzato non “da noia” a blocchi non sincronizzati

Concorrentemente può essere eseguito solo un blocco sincronizzato su un determinato oggetto quindi

- Su di una classe può essere eseguito solo un metodo synchronized static alla volta.
- Solo un metodo synchronized (non static) viene eseguito in contemporanea per ogni oggetto “target”. (quello di cui viene richiesto il *lock* intrinseco)

Sinchronized (cosa?){}

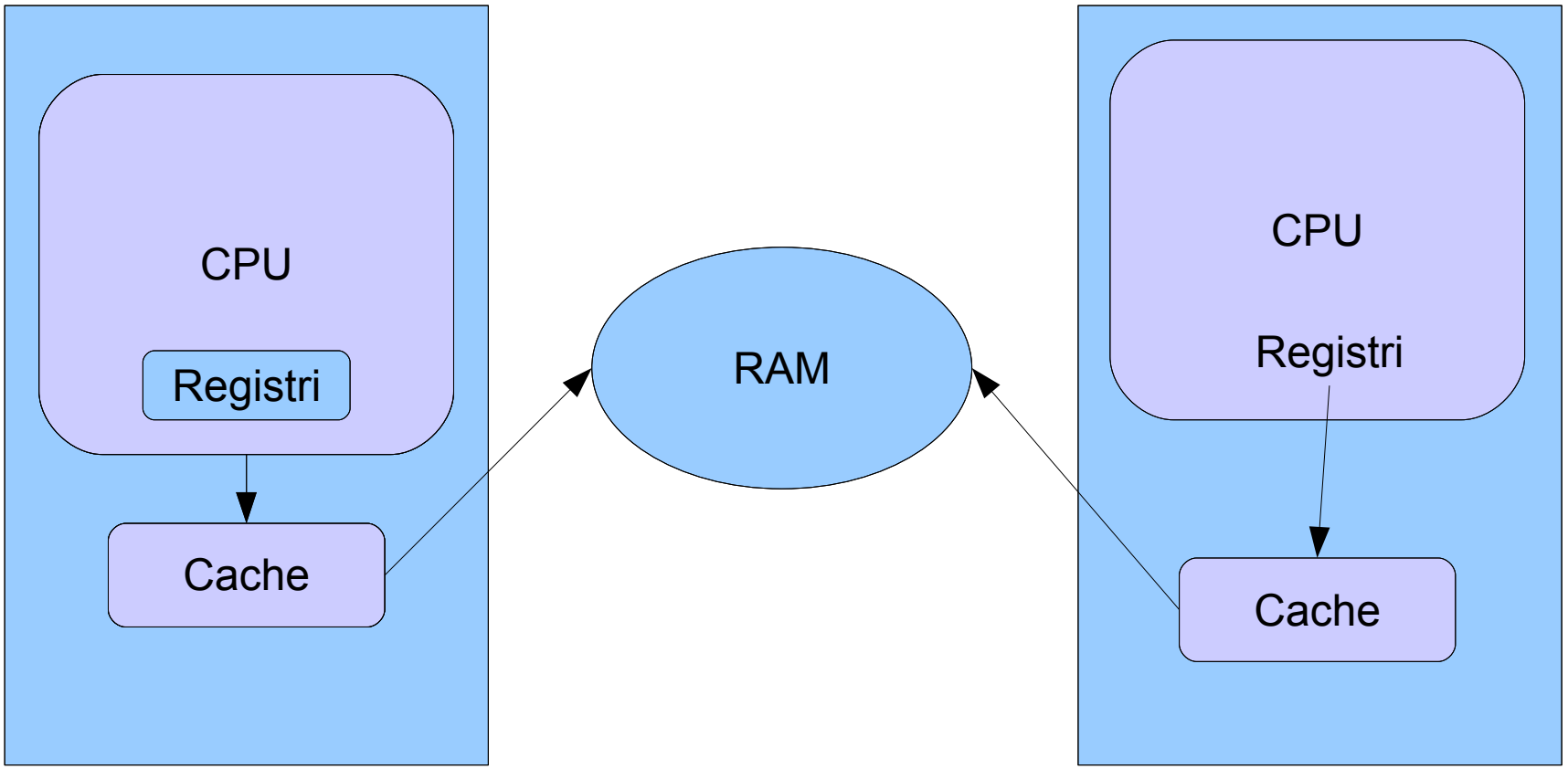
Ovvero, su quale oggetto è necessario sincronizzarsi ed in quali casi?

Non esiste una risposta universale, va visto caso per caso tuttavia è possibile applicare questa regola nei casi più semplici

- Sincronizzarsi sul this quando si vogliono proteggere variabili di istanza.
- Sincronizzarsi sull'oggetto Class della classe attuale per proteggere una variabile statica.

Hardware

Sembra familiare?



Nell moderne archiettture hardware ...

Nelle moderne architetture HW per questioni di performance non esiste una sola memoria centrale visibile dai processori della macchina. Esistono infatti gerarchie di memoria ognuna più veloce dell'altra; queste variano dalla memoria RAM (più lenta) alle *cache* (sempre più veloci) ai registri dei processori (velocissimi).

Per questioni di performance le letture–scritture avvengono prima nelle memorie più veloci e poi (dipendentemente da questioni valutabili solo a *run–time*) nelle memorie più lente fino ad arrivare alla RAM.

Thread e memoria

Nonostante i Thread condividano logicamente la stessa memoria, i dati “salvati” da un Thread non sono necessariamente subito visibili ad altri Thread in quanto potrebbero stare nei registri o nella *cache* del processore dedicato al Thread.

Ottimizzazione *aggressiva*

Le moderne macchine, i moderni compilatori e le moderne JVM potrebbero impiegare tecniche di ottimizzazione aggressiva basate su *scheduling* e riordino delle istruzioni, questo purché venga mantenuta una *within-thread as-if-serial semantic* .

Non è quindi possibile sapere staticamente (osservando il codice) quale sarà l'ordine di esecuzione dei Thread.

Inoltre certi blocchi di codice potrebbero subire delle alterazioni che effettuano delle permutazioni "buone" della sequenza delle istruzioni presenti nel codice sorgente.

Permutazioni buone?

Perché permutare le istruzioni?

Permutazione

C = A + 20

D = C – 1
E = 4
F = 9

oppure

C = A + 20
E = 4
F = 9
D = C – 1

sono sequenzialmente equivalenti?

Per quale motivo permutare?

Apparentemente non esiste alcuna ragione per scegliere tra il primo ed il secondo blocco di codice!

Una vale l'altra se eseguite in serie.

Tuttavia .....

Qualche passo indietro

Vi ricordate della pipeline?

Il primo blocco 1/

IF ID EX(____) MEM(A) WB(A)

IF ID EX(20+A) MEM(_) WB(C)

IF ID EX(C-1) *Fault! Data Hazard*

Il primo blocco 2/

Il primo blocco genererebbe un *fault* di CPU, tuttavia l'inserimento di una *bolla* previene un disastro.

Vediamo adesso il secondo blocco .....

Il secondo blocco 1/

```
IF ID EX(____) MEM(A)       WB(A)
   IF  ID       EX(20+A)    MEM(_)   WB(C)
       IF       ID          EX       MEM(E,4)  WB(_)
                IF          ID       EX        MEM(F,9)  WB(_)
                            IF       ID        EX(C-1)   MEM(_)  WB(D)
```

Il secondo blocco 2/

Nessun *fault*.

Nessuna *bolla*.

La CPU è a regime.

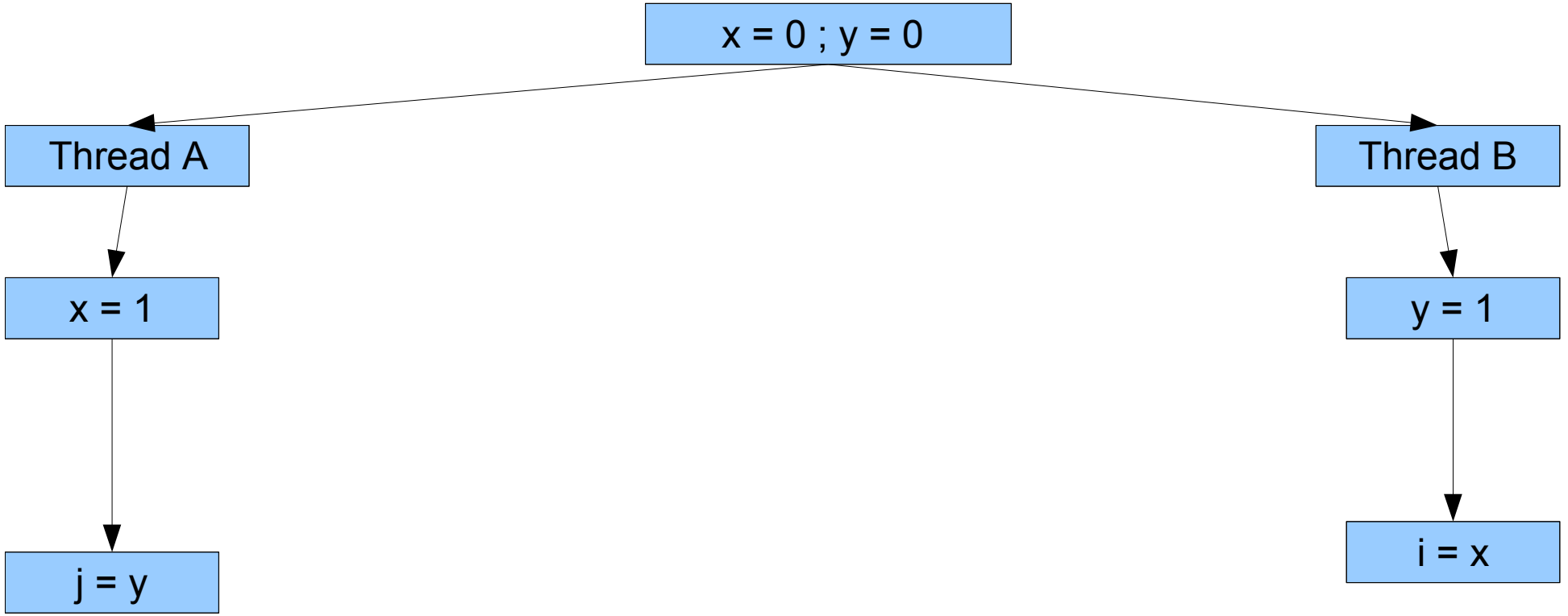
Il riordino delle istruzioni

Il riordino delle istruzioni è una nota pratica di ottimizzazione della CPU.

Viene eseguito a più livelli da entità differenti.

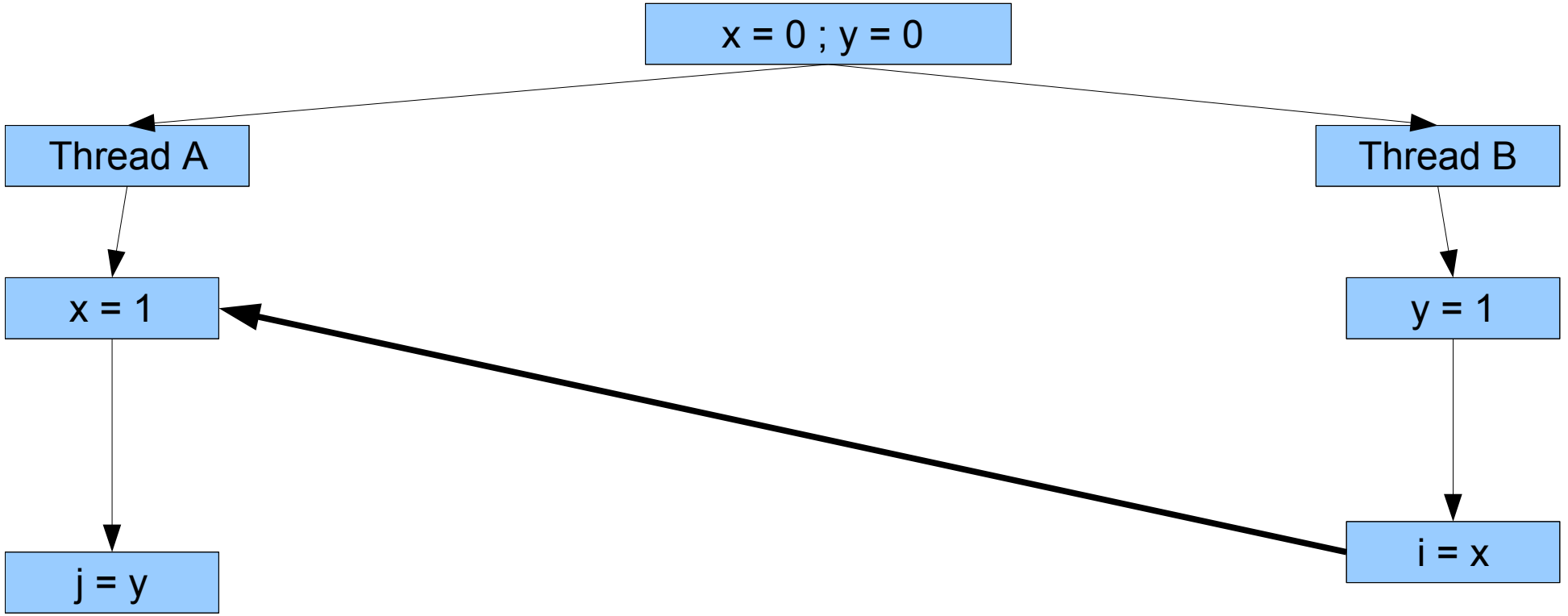
E' una pratica consolidata che accade spesso ... molto molto spesso ... anche più di quanto immaginiate.

Quiz



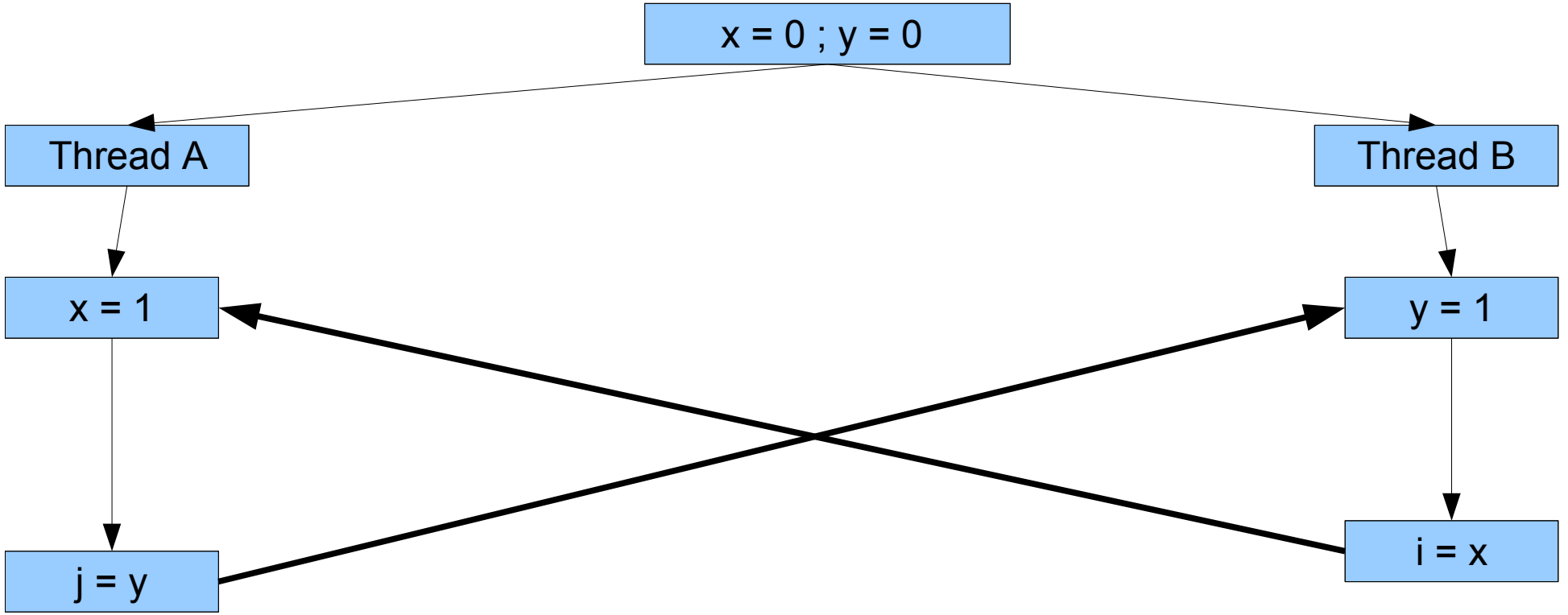
Può accadere che i = 0 e j = 0?

Quiz



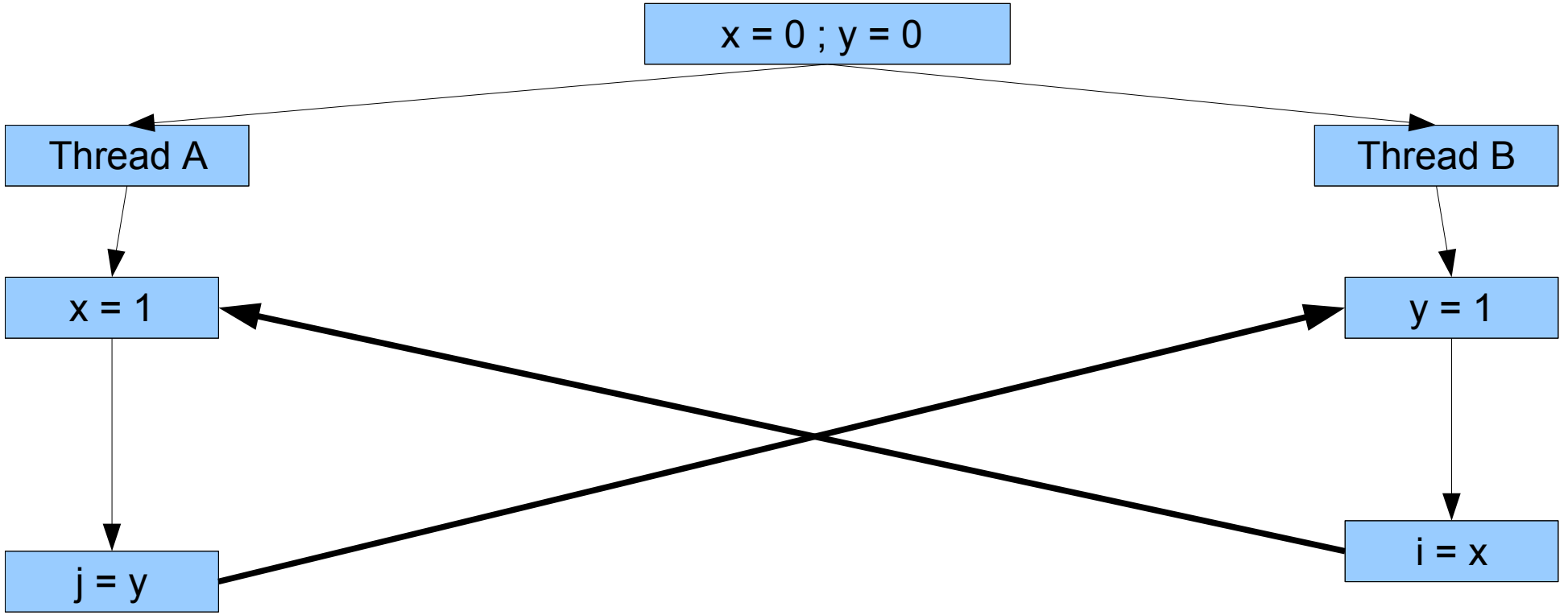
Può accadere che i = 0 e j = 0?

Quiz



Può accadere che i = 0 e j = 0?

Quiz



Può accadere che i = 0 e j = 0?

SI

E allora?

Cosa fare quindi in questi casi?

- Cercare in modo empirico di “scoprire” come si comporta il proprio ambiente? NO!
- Prendere atto del funzionamento e regolarsi di conseguenza. SI

Fortunatamente ci viene in contro il **JavaMemoryModel** dandoci alcune semplici regole da seguire.

Vale la regola del “se non è scritto esplicitamente meglio non azzardare comportamenti improvvisati”

JavaMemory che?

Java Memory Model (JMM) è quella specifica che si preoccupa di indicare come i Thread interagiscono con la memoria dell'elaboratore dichiarando relazioni d'ordinamento **parziale** tra gli eventi del tipo *happens-before*.

(A < B, B < D quindi A < D, ma se E < B niente si può dire su A < E e su E < A, dove con < si intende happens-before)

Mi dispiace per voi ma .....

... tra le regole del JMM **non** c'è quella magica che più o meno dice che

Le modifiche fatte alle "variabili" da un Thread sono istantaneamente visibili a tutti gli altri Thread, senza se e senza ma!

Paura!

Fortunatamente il programmatore non si deve preoccupare più di tanto di come agisce la propria JVM a patto che segua poche e ben definite regole.

Infatti ci viene in aiuto (per ora solo) il nostro caro amico synchronized.

Cosa garantisce un blocco synchronized?

1. Accesso in mutua esclusione al blocco sincronizzato.
2. Visibilità delle modifiche apportate dal Thread all'interno del blocco. (All'inizio del blocco viene letta la memoria centrale ed alla fine del blocco i dati ci vengono *flushati)*
3. Limitazioni sul riordino delle istruzioni presenti nel blocco. (Quelle dentro non ne escono, quelle fuori si possono “muovere”)

JMM e JVM 1 /

Vedremo una versione semplificata dell'interazione tra Thread e memoria.

Questa versione semplificata è composta da:

- *Working memory* del Thread
- *Execution engine* del Thread
- Memoria centrale dell'elaboratore
- Azioni eseguite tra di queste

JMM e JVM 2/

Ogni Thread ha associata una *Working memory* che contiene copie delle variabili che il Thread *usa* o *assegna*.

JMM e JVM 3/

La memoria centrale contiene la copia *master* di queste variabili. La memoria centrale contiene anche i *lock*, più Thread possono concorrere all'acquisizione di un solito *lock*.

JMM e JVM 4/

Il trasferimento della variabile dalla memoria centrale e l'assegnazione del Thread nella *working memory* sono azioni debolmente accoppiate.

JMM e JVM 5/

L'utilizzo di una variabile da parte di un Thread implica il trasferimento dalla *working memory* all'*execution engine*.

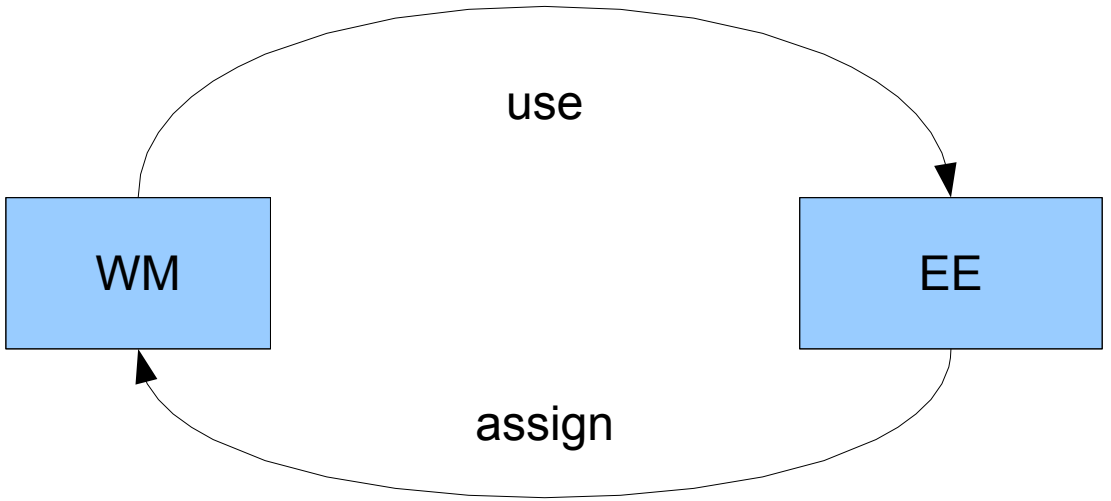
JMM e JVM 6/

Le azioni possibili sono le seguenti:

- use
- assign
- read
- load
- store
- write
- lock
- unlock

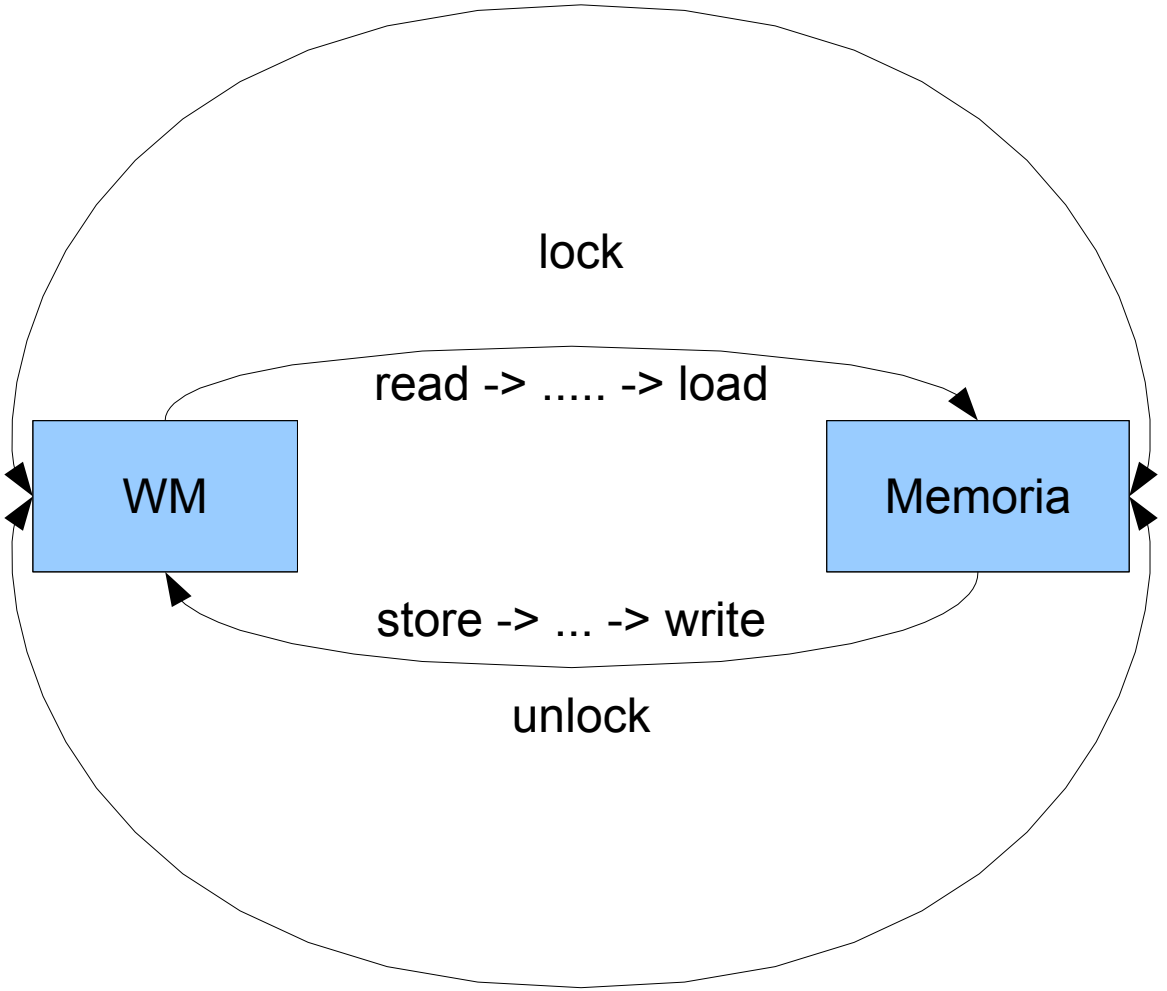
JMM e JVM 7 /

Working memory ed *Execution engine*



JMM e JVM 8/

Working memory e Memoria centrale



JMM e JVM 9/

Essendo (*read*,*load*) e (*write*,*store*) debolmente accoppiate è possibile che altre azioni avvengano nel frattempo portando ad uno stato di incoerenza dei dati (*staleness*).

Ordine di esecuzione e consistenza 1/

- Le azioni eseguite da un Thread sono totalmente ordinate (per ogni coppia di azioni una precede l'altra).
- Le azioni eseguite dalla memoria centrale su di una variabile sono totalmente ordinate.
- Le azioni eseguite dalla memoria centrale su di un lock sono totalmente ordinate.
- Un'azione né precede né segue se stessa.

Ordine di esecuzione e consistenza 2/

I Thread non dialogano direttamente tra di loro ma tramite la memoria centrale.

- Ogni azione *lock* è eseguita congiuntamente dalla memoria centrale e da un Thread.
- Ogni azione *load* è univocamente accoppiata con una *read* e la *read* precede la *load*.
- Ogni azione *store* è univocamente associata ad una azione *write* e la *write* segue la *store*.
- SI! Manca una regolina magica ...

Sincronizzazione leggera

Esiste una sincronizzazione (di memoria) leggera per singole variabili. Dichiarando una variabile come volatile questa viene sempre (ed immediatamente) letta e scritta da e su la memoria centrale.

SI! Per le variabili volatile è presente la regolina magica.

Verba volant, scripta manent

Tuttavia la *volatilità* di una variabile garantisce solo che operazioni “atomiche” (lettura e scrittura) siano corrette.

Il ++ ed il -- non sono atomici

Quando utilizzare la keyword volatile?

- La variabile non obbedisce ad alcuna invariante rispetto ad altre variabili.
- Le scritture sulla variabile non dipendono dal suo valore corrente.
- Nessun Thread scrive valori illegali nella variabile.
- Le azioni compiute dai lettori della variabile non dipendono dai valori di altre variabili.

Consiglio

Utilizzate la *keyword* volatile quando la variabile funge da contatore *strettamente monotono* (e limitato), ed è interesse avvertire solo il *superamento* di una soglia.

Se la variabile i è monotona crescente usare

```
while(i <5) nulla()
```

al posto di

```
while (i !=5) nulla()
```

Utilizzo corretto del volatile

```
public class ServerVolatile {
	private volatile boolean stop = false;
	public void shutdown(){stop = true;}
	public  void go(){
		while (!stop){
			new Thread( new Runnable() {
				public void run() {
					// codice
				} }).start();
			// altro codice .....
		}
	}
}
```

Break

NOTA

Tutti problemi precedentemente elencati riguardano esclusivamente variabili di istanza e di classe.

I parametri locali dei metodi non ne sono soggetti in quanto vengono allocati un uno spazio di memoria diverso e non vengono *sharati* dai Thread.

NOTA: se ad un parametro locale si passa un reference ad un oggetto condiviso il discorso cambia .... ;–)

Definizione

- *Una classe è Thread-safe se si comporta correttamente quando acceduta da più Thread a prescindere dello scheduling o dall'esecuzione interlacciata di questi all'interno dell'ambiente di esecuzione.*
- *Le classi Thread-safe incapsulano ogni sincronizzazione necessaria cosicché i suoi client non debbano fornire di proprie.*
- *.....*

Riflessione ....

Un programma con un singolo Thread è ovviamente Thread–safe, chi utilizza programmi monothread (batch e simili) può stare tranquillo in quanto il JMM garantisce una semantica *as–if–serial*.

Chi non scrive programmi che "*lanciano*" Thread allora può stare tranquillo?

.....Riflessione ....

NO

I Thread sono ovunque

- HttpServlet: una sola istanza per ogni classe.
- Actions di Jakarta Struts: una sola istanza per ogni classe.
- Vari Framework.
- Variabili static.

Attenzione

Il Framework che utilizzate potrebbe fornirvi oggetti Java condivisi da più Thread, controllate quindi la documentazione per ottenere informazioni.

Purtroppo non sempre la questione della concorrenza viene trattata in modo adeguato.

Attenzione

I Framework introducono la concorrenza nei vostri applicativi chiamando i componenti degli applicativi dai Thread del Framework.

I componenti applicativi inevitabilmente accedono allo ***stato*** dell'applicativo.

Si richiede quindi che ***tutto*** il flusso del codice sia Thread-safe.

Errore ricorrente

```
public class UnsafeServlet extends HttpServlet{
      private static final long serialVersionUID = 1L;
      private String utente ;
      private String foo;
      private Integer bar;
      public void service(HttpServletRequest req, HttpServletResponse resp)
throws IOException
            , ServletException{
            //codice
            utente = req.getRemoteUser();
            foo = req.getParameter("foo");
            try { bar = (Integer)req.getAttribute("bar");} catch(Throwable t){}
            // codice che usa utente foo e bar .... magari passandoli ad altre classi
      }
}
```

Banale soluzione

```
public class SafeServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
	public void service(HttpServletRequest req, HttpServletResponse resp) throws
IOException
	, ServletException{
		String utente ;
		String foo;
		Integer bar = null;
		//codice
		utente = req.getRemoteUser();
		foo = req.getParameter("foo");
		try { bar = (Integer)req.getAttribute("bar");} catch(Throwable t){}
		// codice che usa utente foo e bar .... magari passandoli ad altre classi
	}
}
```

Domanda da porsi

Quanto un Framework vi *fornisce* un oggetto quale è la prima e più importante domanda da porsi?

Suggerimento: pensate a che cosa si applica il synchronized.

Risposta:

Quante istanze di tale oggetto vengono create e mantenute vive dal Framework?

Se la risposta è “un'istanza per chiamata” potete (con le dovute precauzioni) tirare un sospiro di sollievo.

Se la risposta è “un'unica istanza” .....

Riflettiamo

Oggetti *stateless* o *immutabili* sono automaticamente(*) Thread–safe e possono essere quindi condivisi senza problemi tra più Thread.

Un oggetto è *statefull* se ha stato (variabili di istanza rispetto alle quali rispetta una certa invariante), è *immutabile* se tutte le sue variabili di istanza vengono inizializzate nel costruttore e non esiste modo per modificarle.

Se un oggetto non ha stato, se il suo stato viene ***correttamente*** impostato nel costruttore e non più mutato, allora si ha la certezza di avere un oggetto forzatamente Thread–safe.

Normalmente .....

Spesso, anche per mezzo dei moderni IDE, siamo tentati dall'aggiungere un *getter* ed un *setter* ad ogni campo dei nostri oggetti.

Questa in alcuni casi si può rivelare una brutta pratica in quanto non è sempre logico disporre di un modificatore.

Esempio di oggetto mutabile

```
public class MutablePoint {
        private float x;
        private float y;
        private float z;
        public MutablePoint(float x, float y, float z) {
                super();
                this.x = x;
                this.y = y;
                this.z = z;
        }
        public MutablePoint(){}
        public float getX() {return x;}
        public void setX(float x) {this.x = x;}
        public float getY() {return y;}
        public void setY(float y) {this.y = y;}
        public float getZ() {return z;}
        public void setZ(float z) {this.z = z;}
}
```

La versione immutabile

```
public class ImmutablePoint {
      private final float x;
      private final float y;
      private final float z;
      public ImmutablePoint(final float x,
                  final float y,
                  final float z) {
            this.x = x;
            this.y = y;
            this.z = z;
      }
      public float getX() {return x;}
      public float getY() {return y;}
      public float getZ() {return z;}
}
```

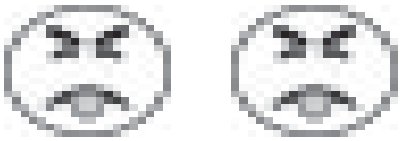
Costruttori?

I costruttori contrariamente a quanto si pensa non vengono eseguiti in *perfetta armonia*, è quindi possibile che un Thread entri in possesso di un oggetto non ancora “perfettamente” costruito.

Esempio

```
public class FinalFieldExample {
      final int x;
      int y;
      static FinalFieldExample f;
      public FinalFieldExample(){
            x = 3;
            y = 4;
      }
      public static void writer() { f = newFinalFieldExample();}

      public static void reader() {
            if (f != null){
                  int i = f.x; //final
                  int j = f.y; //non final
            }
      }
}
```



Sembra tutto ok ...

Il problema sta nel fatto che il Thread che invoca il metodo *reader*(), nonostante controlli che f non sia null (costruttore eseguito), potrebbe non vedere il valore "corretto" per y ovvero 4.

Questo non è vero per x in quanto JMM garantisce che la visibilità dei campi final *happens-before* qualsiasi (altro) Thread possa entrare in possesso del reference dell'oggetto che li contiene.

Soluzione finale

E' quindi chiaro che neanche la costruzione di un oggetto è “sicura”. Non lasciate “fuggire” un oggetto fino a quando non è ***correttamente*** costruito.

Se si deve costruire un oggetto immutabile è necessario dichiarare tutti i suoi campi come final.

Long e double

Scritture e letture su campi di tipo long o double non sono garantite essere atomiche, le JVM sono libere di spezzare le letture e le scritture in 2 (ragionevolmente!) istruzioni distinte.

Si consiglia quindi di dichiarare tali tipo di variabili come volatile oppure di gestire la sincronizzazione *ad-hoc*.

Ultima riflessione ...

..... prima di cambiare argomento

La *triade* della morte

Race Condition (data race)

Riordino delle istruzioni

Visibilità delle modifiche effettuate da un Thread

Ovviamente sono in ordine alfabetico

Pausa

Domande?

Parte II

Problematiche ricorrenti e soluzioni pratiche

Argomenti trattati

- Deadlock e starvation
- Wait/Notify
- Latch
- ReadWriteLock
- Variabili condition

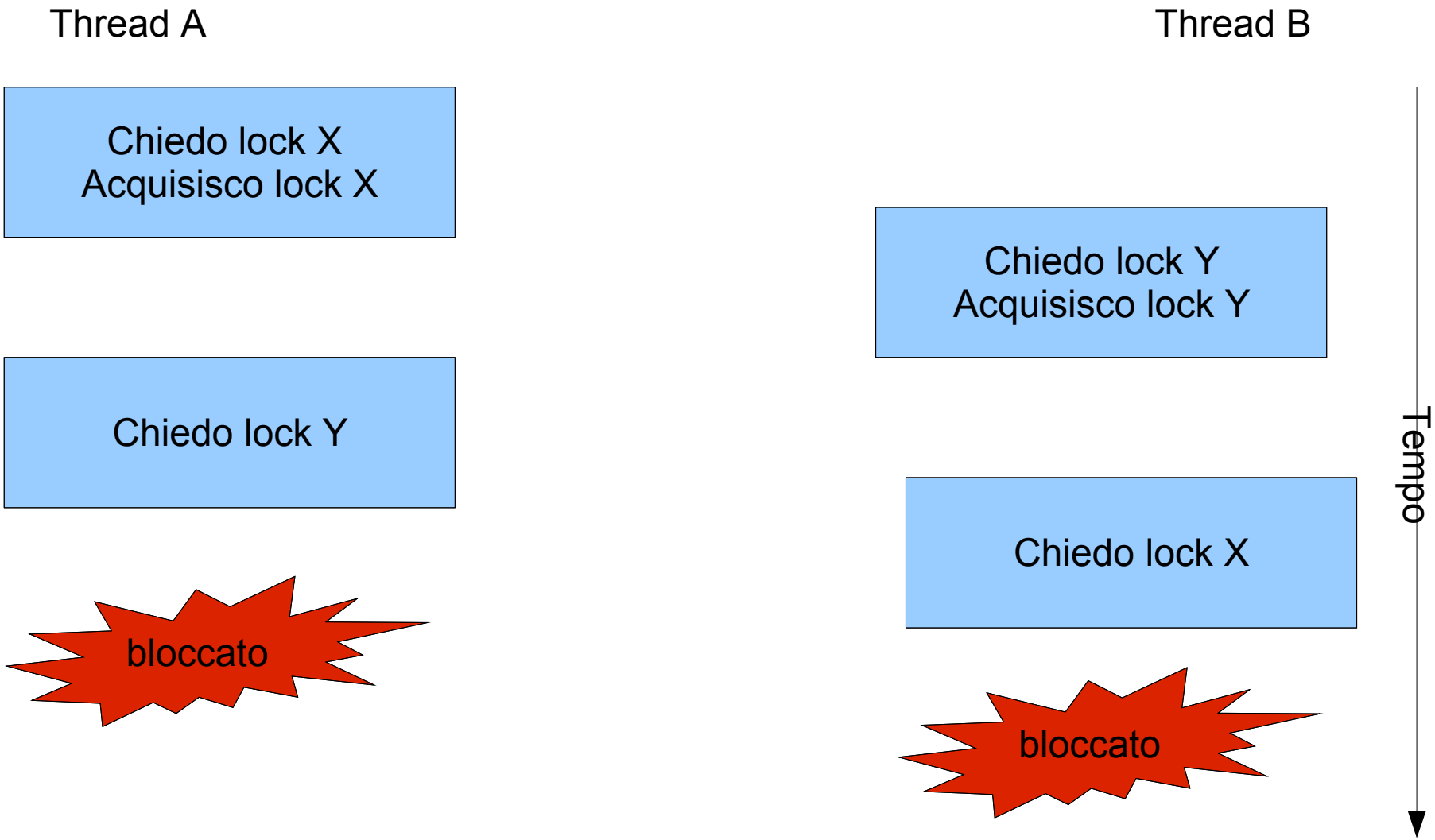
Vitalità: Deadlock e Starvation

Starvation: un Thread in attesa di una risorsa potrebbe non ottenerla mai e rimanere bloccato. (altri Thread passano avanti)

Deadlock: due Thread potrebbero *mutuamente* richiedere una risorsa in possesso dell'altro per andare avanti.Tutti i Thread si bloccano.

- Esiste una versione che “blocca tutto” con un solo Thread

Deadlock!


Thread A
Thread B
Chiedo lock X
Acquisisco lock X
Chiedo lock Y
Acquisisco lock Y
Chiedo lock Y
Chiedo lock X
bloccato
bloccato
Tempo

Prevenire il deadlock

In generale quando c'è più di un *lock* da acquisire è necessario documentare bene l'ordine di acquisizione forzandone uno *ragionevole*.

Nel caso precedente si può imporre nella documentazione di acquisire prima il *lock* X e poi quello Y.

Tipico deadlock

Thread X: acquisizione A ->..... -> B

Thread Y: acquisizione B -> ..... -> A

Dove sta il deadlock?

Verranno mostrate due classi Java, cercate di capire dove è il *deadlock*

Taxi

```
public class Taxi {
	private final Dispatcher dispatcher;
	private Point location,destination;
	public Taxi(final Dispatcher dispatcher) {
		this.dispatcher = dispatcher;
	}
	public synchronized Point getLocation(){
		return location;
	}
	public synchronized void setLocation(Point location){
		this.location= location;
		if(location.equals(destination)){
			dispatcher.notifyAvailable(this);
		}
	}
}
```

Dispatcher (la centrale)

```
public class Dispatcher {
	private final Set<Taxi> taxis = new HashSet<Taxi>(// ....
			);
	private final Set<Taxi> availableTaxis= new HashSet<Taxi>(//----
			);
	public synchronized void notifyAvailable(Taxi taxi) {
		availableTaxis.add(taxi);
	}
	public synchronized Image getImage(){
		//scritti a caso!
		Image retval = new Image();
		for (Taxi t: taxis) {
			retval.drawPoint(t.getLocation());
		}
		return retval;
	}
}
```

Vediamole appaiate

```
public class Taxi {

private final Dispatcher dispatcher;

private Point location,destination;

public Taxi(final Dispatcher dispatcher) {

this.dispatcher = dispatcher; }

public synchronized Point getLocation(){

return location; }

public synchronized void setLocation(Point location){

this.location= location;

if(location.equals(destination)){

dispatcher.notifyAvailable(this);

}

}

}
```

```
public class Dispatcher {

private final Set<Taxi> taxis = new HashSet<Taxi>(// ....

);

private final Set<Taxi> availableTaxis= new
HashSet<Taxi>(//----);

public synchronized void notifyAvailable(Taxi taxi) {

availableTaxis.add(taxi); }

public synchronized Image getImage(){

//scritti a caso!

Image retval = new Image();

for (Taxi t: taxis) {

retval.drawPoint(t.getLocation());

}

return retval;

}

}
```

Deadlock?

TAXI x

SetLocation
acquisisco lock this
location = destination

Chiedo lock dispatcher per
eseguire notifyAvailable

Dispatcher

GetImage
chiedo ed acquisisco lock Dispatcher
ciclo sui taxi
recupero taxi x
chiedo il lock per taxi x al fine di
eseguire getLocation

Soluzione?

1) Limitare la sincronizzazione allo stretto necessario.
2) All'interno di *getImage* “copiare deep” il vettore dei taxi in un blocco synchronized(this) ed operare sulle copie.

Safe Taxi

```
public  void setLocation(Point location) {
        boolean reachedDestination;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
        if (reachedDestination)
            dispatcher.notifyAvailable(this);
    }
```

Safe Dispatcher

```
public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
```

In generale

Non chiamare metodi *alieni* con un *lock* acquisito in quanto non sempre si sa a cosa si va incontro; effettuare le cosiddette *open calls*.

Azioni composte (compound actions)

Nonostante alcune classi Thread–safe implementino metodi accessori e mutatori atomici spesso l'esecuzione di un'istruzione composta, nonostante non renda “sporco” lo stato dell'oggetto, potrebbe ottenere effetti indesiderati.

Un esempio concreto

La classe Vector è Thread–safe nel senso che “rende visibili” e ***correttamente*** pubblicati gli oggetti in essa inseriti a tutti i Thread che tentano di accedervi.

Inoltre i vari *add*(), *get*() e *remove*() sono atomici. Non è quindi possibile portare il Vector in uno stato “inconsistente”.

E allora?

E' tuttavia possibile che due Thread accedano *concorrentemente* al solito vettore, un Thread prova a togliere l'ultimo elemento mentre un altro prova a recuperarlo.

Work stealing

La struttura interna del Vector rimane integra, i dati sono coerenti e non “sporchi” ma il povero Thread “lettore” si becca un **ArrayIndexOutOfBoundsException**

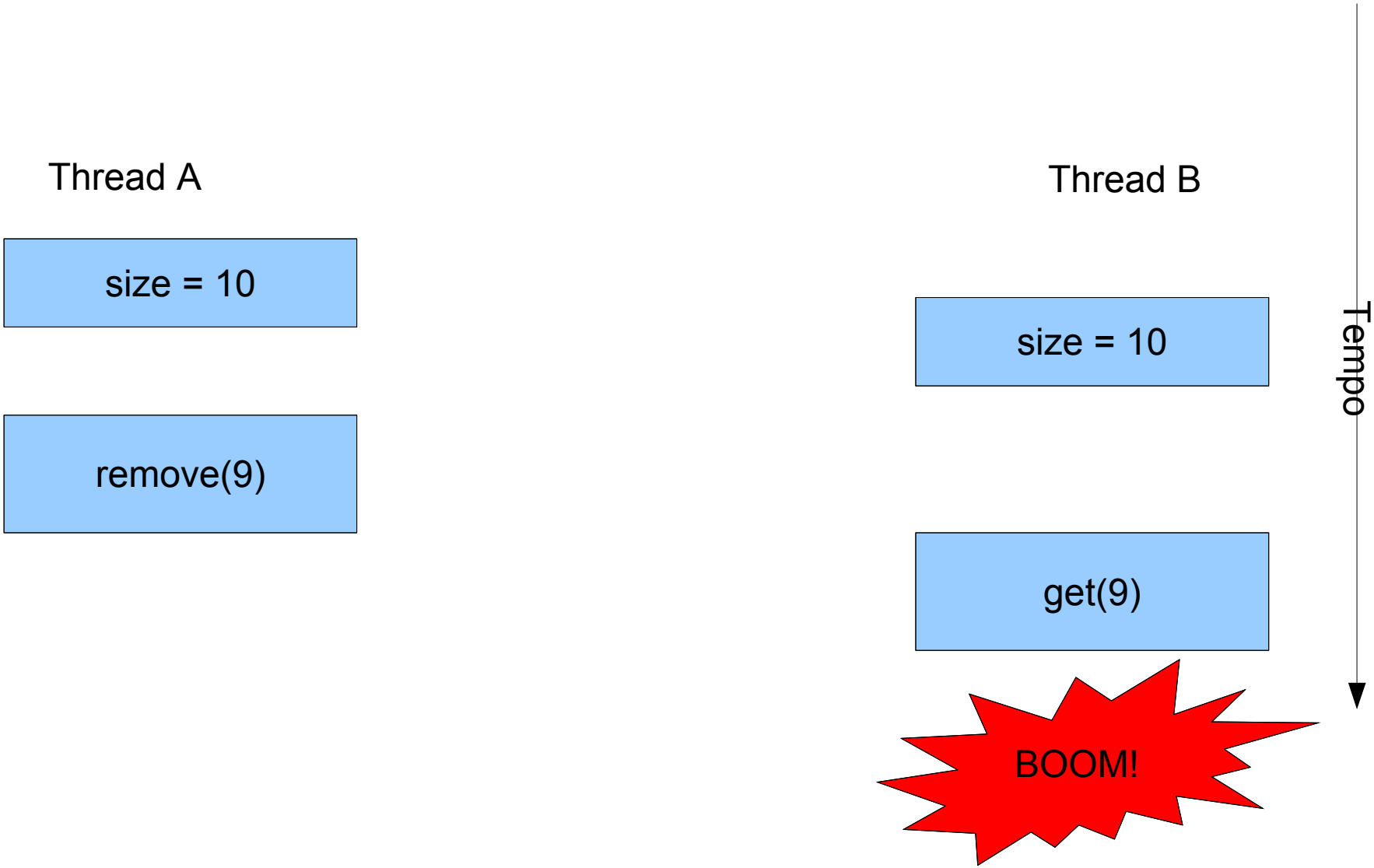
(un altro Thread gli “ruba” il lavoro)

Vediamo il codice

```
public static Object getLast(Vector v){
    int last = v.size() -1;
    return v.get(last);
}

public static void deleteLast(Vector v){
    int last = v.size() -1;
    v.remove(last);
}
```

Esecuzione temporale



Cosa fare?

La classe Vector come molte altre classi *sincronizzate* permette in modo sicuro il *client–side–locking* ovvero la possibilità di far sincronizzare le operazioni dal chiamante.

Ciò non è in conflitto con la regola di eseguire solo *open calls* in quanto il *c.s.l.* è documentato.

Quindi ...

In pratica Vector e simili “sincronizzano” le operazioni sull'argomento implicito dei metodi (this). Sincronizzare il chiamante su tale oggetto permette quindi di eseguire operazioni composte come se fossero atomiche.

Esempio

```
public static Object getLast(Vector list) {
      synchronized (list) {
          int lastIndex = list.size() - 1;
          return list.get(lastIndex);
      }
  }
  public static void deleteLast(Vector list) {
      synchronized (list) {
          int lastIndex = list.size() - 1;
          list.remove(lastIndex);
      }
  }
```

Break

Performance 1 /

Generalmente si creano applicazioni multithread per questioni di *performance*, non si deve quindi generare codice con scadenti prestazioni dipendenti da una cattiva gestione della sincronizzazione.

Performance 2/

Tuttavia non si deve neanche eliminare la sincronizzazione in virtù di una *miglior* velocità del sistema.

First make it *work*, then make it fast!

Performance 3/

In generale una cattiva performance dovuta ad una scarsa scalabilità del sistema è spesso causata da un utilizzo troppo “largo” della sincronizzazione.

Alcune migliorie possono essere introdotto con un utilizzo più ragionevole del synchronized.

A partire da Java5 sono disponibili nuove classi e costrutti per un utilizzo più veloce e semplice dei Thread.

Blocchi sincronizzati troppo grossi

Se una classe deve proteggere un campo privato da accessi concorrenti, al di là del puro accessore/mutatore, non conviene sincronizzare tutto il metodo che accede a tale campo, invece è bene capire quale è la *regione critica* e gestire lì la sincronizzazione.

Esempio

```
...
private Datum sensibleDatum = ....

public synchronized double calculateRatio(int base) {
        double d = ..... base .....;
        // codice inerme .. d= d*base + .....
        //sezione critica
        ..
        d =  d * sensibleDatum.getAndSetSomething() + 1/base;
        ..
        // fine sezione critica

        return d;
}
```

Versione “più performante”

```
...
private Datum sensibleDatum = ....

public  double calculateRatio(int base) {
        double d = ..... base .....;
        // codice inerme .. d= d*base + .....
        //sezione critica
        synchronized(this) {
        ..
        d =  d * sensibleDatum.getAndSetSomething() + 1/base;
        ..
        // fine sezione critica
        }
        return d;
}
```

Performance: Un solo *lock* protegge parti diverse :-/

Quando si devono proteggere da accessi concorrenti variabili che non “entrano in contatto” tra di loro non è buona norma utilizzare un solo *lock* in quando la scalabilità ne risente.

Può essere un valido esempio?

```
public class LockComplessoDL {

private long mele,pere;

private long auto,camion;

private final Object lock_frutta = new Object();

private final Object lock_veicoli = new Object();

public void usaMele(){synchronized(lock_frutta) { /* usa mele */ }};

public void usaPere(){synchronized(lock_frutta) { /* usa pere */ }};

public void usaMeleEPere(){synchronized(lock_frutta) { /* usa entrambe */ }};

public void usaAuto(){synchronized(lock_veicoli){/* usa auto */  }};

public void usaCamion(){synchronized(lock_veicoli){/* usa camion */  }};

public void usaAutoECamion(){synchronized(lock_veicoli){ /* usa entrambi */ }};

public void usaTutto() { synchronized (lock_frutta) { synchronized (lock_veicoli) {

/* usa la frutta ed i veicoli */ } } }

public void usaAltro() { synchronized (lock_veicoli) { synchronized (lock_frutta) {

/* usa ... non specificato */ } } }

}
```

NO!

L'esempio precedente è soggetto a Deadlock.

Versione corretta

```
public class LockComplessoNODL {

private long mele,pere;

private long auto,camion;

private final Object lock_frutta = new Object();

private final Object lock_veicoli = new Object();

public void usaMele(){synchronized(lock_frutta) { /* usa mele */ }};

public void usaPere(){synchronized(lock_frutta) { /* usa pere */ }};

public void usaMeleEPere(){synchronized(lock_frutta) { /* usa entrambe */ }};

public void usaAuto(){synchronized(lock_veicoli){/* usa auto */  }};

public void usaCamion(){synchronized(lock_veicoli){/* usa camion */  }};

public void usaAutoECamion(){synchronized(lock_veicoli){ /* usa entrambi */ }};

public void usaTutto() { synchronized (lock_frutta) { synchronized (lock_veicoli) {

/* usa la frutta ed i veicoli */ } } }

public void usaAltro() { synchronized (lock_frutta) { synchronized (lock_veicoli) {

/* usa ... non specificato */ } } }

}
```

Un esempio concreto

Creiamo una semplice classe HT (*HashTable*) che gestisce la concorrenza in modo *efficace*.

La mutua esclusione infatti non verrà applicata sincronizzando tutti i metodi su this.

La nostra Hash Table 1/

La nostra HT disporrà di n slot contenenti liste di elementi.

(Gli elementi non sono quindi buttati in un calderone unico)

La nostra Hash Table 2/

Ogni elemento andrà in uno slot *ben definito*.

La nostra Hash Table 3/

Identificato lo slot relativo ad un elemento, le operazioni avverranno sincronizzate sullo slot e non sull'intera HT.

La nostra Hash Table 4/

Come associo un elemento ad uno slot?

Ogni oggetto Java ha associato un numerino chiamato *hashcode* che identifica (in teorie) univocamente tutti gli oggetti Java.

Un elemento verrà inserito nello slot k se il suo *hashcode* **modulo** n è k.

La nostra Hash Table 5/

Esempio ...

```
public void put(Object o){
 int i = o.hashCode();
 i = i %nslots;
 synchronized (slots[i]) {
  slots[i].add(o);
 }
}
```

E' possibile usare altro al posto del synchronized?

In Java esiste un altro modo per far *aspettare* ad un oggetto il verificarsi di un evento.

La classe Object dispone di due metodi (tre per la verità) ovvero *wait*() e *notify*(). (*notifyAll*())

Questi si applicano su un oggetto che funge da *monitor* (viene anche chiamato Condition Queue, Condition Variable o Condition) e servono per bloccare/sbloccare un Thread in attesa del verificarsi di una condizione.

Wait e notify. Quando?

- *wait*(): un Thread ha bisogno del verificarsi di una certa condizione per procedere.
- *notify*(): un altro Thread provoca il verificarsi di una certa condizione e lo notifica.

E' quindi un meccanismo di IPC

Un *semplice* lock :-/

```
public class SimpleLock {
private boolean free = true;

public void lock() throws InterruptedException {
 synchronized (this) {
  while(!free) {
   wait();
  }
 free = false;
 }
}

public void release() throws InterruptedException   {
 synchronized (this) {
  free = true;
  notifyAll();
  }
 }
}
```

In pratica cosa fanno?

wait() mette il Thread corrente in attesa del verificarsi di una notifica sull'oggetto su cui è stato chiamato. (nell'esempio this).

notify() esegue tale notifica ad un Thread solo. (quale???)

notifyAll() notifica tutti i Thread in attesa sull'oggetto su cui è stato invocato; genera molto *Context–Switch.*

Esplicitiamo meglio il meccanismo.... :-/

```
public class AnotherSimpleLock {
 private final Object monitor = new Object();
 private boolean free = true;
 public void lock() throws InterruptedException {
  synchronized (monitor) {
   while(!free) {
    monitor.wait();
   }
   free = false;
  }
 }

 public void release() throws InterruptedException   {
  synchronized (monitor) {
   free = true;
   monitor.notifyAll();
  }
 }

}
```

Commentiamo il codice

Tutti i meccanismi vengono eseguiti su un oggetto chiamato monitor ed è una variabile di istanza final.

Il controllo sul *lock* libero viene fatto in un blocco sincronizzato (su monitor) e dentro un ciclo *while.*

Il rilascio del *lock* viene effettuato in un blocco sincronizzato (su monitor). Il rilascio libera il *lock* ed esegue una notifica.

Come si usa?

```
public class ExampleOfSimpleLock {
 private final SimpleLock sl = new SimpleLock();
 private final MutableSensibleData msd = new
MutableSensibleData();
 public void readData() throws InterruptedException {
  sl.lock();
  try { msd.readsomething(); }finally {sl.release();} }

 public void writeData() throws InterruptedException {
  sl.lock();
  try { msd.writesomething(); }finally {sl.release();} }

 // -----
}
```

Osservazioni

Il *lock* viene acquisito e subito dopo inizia un blocco try/finally che lo rilascia. (Non fare in nessun altro modo!)

Il *lock* deve essere utilizzato con *ferrea disciplina*: prima si acquisisce e poi si rilascia (nel solito metodo) altrimenti ci possono essere anomalie.

Nota

I *lock* precedenti si fidano troppo del chiamante.

E' possibile che il metodo release venga chiamato da un Thread che non ha acquisito il *lock*.

Versione “malfidata”

```
public class LockMalfidato {
    private Thread possessore = null;
    public synchronized void lock() throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        while (possessore != null)
            wait();
        possessore = Thread.currentThread();
    }
    public synchronized void release() {
        if (possessore == Thread.currentThread()) {
            possessore = null;
            notify();
        } else
           throw new IllegalStateException(
               "non ho chiamato io il metodo!”);
    }
}
```

Al ristorante

Il pattern *lavapiatti/cameriere* rozzamente funziona in questo modo:

- Un lavapiatti lava dei piatti e li posiziona su un carrello di dimensione limitata.
- Più camerieri recuperano un piatto alla volta dal carrello (se non vuoto) e lo portano ai clienti autonomamente.
- Il lavapiatti si blocca finché il carrello è pieno, i camerieri finché è vuoto.

Produttore consumatore

Il pattern *producer/consumer* rozzamente funziona in questo modo:

- Un produttore produce “oggetti” e li inserisce in una coda a spazio limitato.
- Più consumatori recuperano un oggetto alla volta dalla coda (se non vuota) e lo processano autonomamente.
- Il produttore si blocca finché la coda è piena, i consumatori finché la coda è vuota.

Un rozzo esempio del pattern

```
public class RawProdCons {
     private final Coda c = new Coda(100);

     public void produci() throws InterruptedException {
          synchronized(this){
               while (c.isfull()) {
                    wait();
               }
               c.inserisci(Math.random());
               notifyAll();
          }
     }

     public void consuma() throws InterruptedException{
          synchronized (this) {
               while(c.isEmpty()){
                    wait();
               }
               Object o = c.rimuovi();
               //o.qualcosa();
               System.out.print(o);
               notifyAll();
          }
     }
}
```

Riflessione

- Non usare in produzione!
- Ancora *wait()* e *notify()* sono in un blocco sincronizzato.
- Notate il *notifyAll().*
- I due metodi controllano *condition* diverse.

Attenzione

Invocare l'attesa o la notifica su di un oggetto del quale non si possiede il *lock* intrinseco genera un'eccezione di Runtime!

Uomo avvisato mezzo salvato. *(Donna avvisata mezza salvata)*

Come funziona

- La notifica o l'attesa sulla coda di wait di un oggetto ne *richiedono* l'acquisizione del *lock* intrinseco.
- Entrato nello stato wait il Thread rilascia il *lock* acquisito.
- Una volta *risvegliato* DEVE ri–acquisire il *lock;* questo gli può anche essere rubato da un altro Thread.
- Se si esegue una notify() e si sveglia un Thread che controllava un'altra Condition si ha un potenziale **deadlock**!

Break

Aspettare il completamento di un evento asincrono

E' spesso necessario mettere dei punti di controllo nel programma che fungono da barriera: “si può proseguire solo se si sono verificati N eventi”.

Ad esempio: si può eseguire questo metodo solo dopo che la Servlet *pippo* ha caricato tutte le “proprietà”.

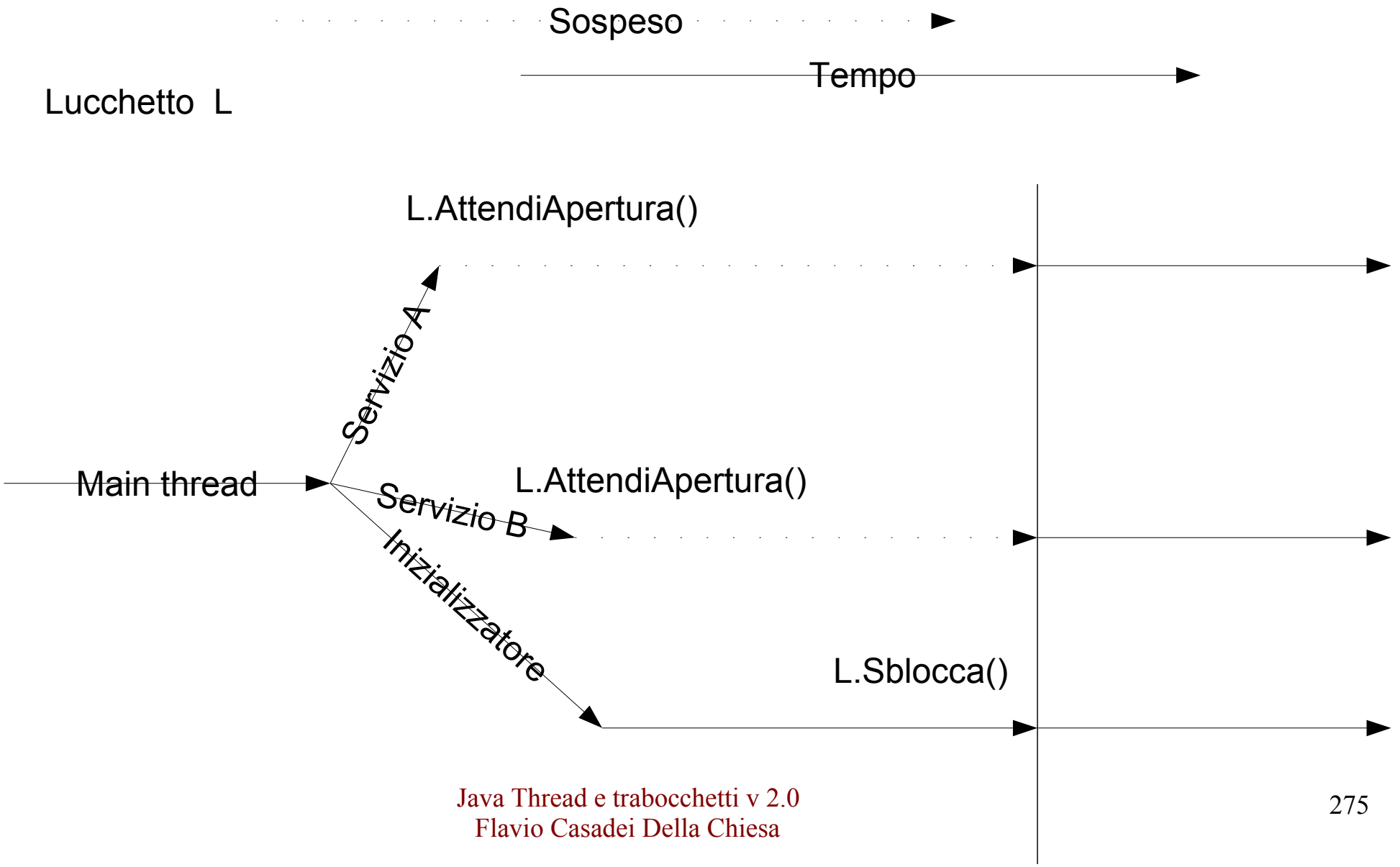
Esempio

Nell'esempio successivo si assume che un oggetto chiamato lucchetto possieda due metodi

- Sblocca()
- AttendiApertura()

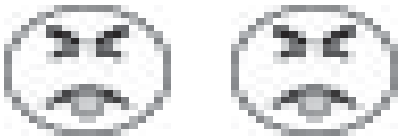
se il lucchetto è sbloccato il metodo AttendiApertura() ritorna immediatamente, altrimenti resta in attesa

Esecuzione temporale



Lucchetto ammazza-cpu

```
public class CPUKillerLatch {
    private volatile  int nEvents;
    public CPUKillerLatch(int n){ this.nEvents = n;}
    public void aspetta() {
        while (nEvents >0){}
    }
    public void esegui(){
        synchronized (this) {
            if (nEvents >0) nEvents --;
        }
    }
}
```



Lucchetto un pochino cpukiller! :-/

```
public class NotSoCPUKillerLatch {
    private volatile  int nEvents;
    public NotSoCPUKillerLatch(int n){ this.nEvents = n;}
    public void aspetta() throws InterruptedException {
        while (nEvents >0){
                Thread.sleep(1000);
        }
    }
    public void esegui(){
        synchronized (this) {
            if (nEvents >0) nEvents --;
        }
    }
}
```

Lucchetto vecchia maniera

```
public class OldFashionedLatch {
 private   int nEvents;
 public OldFashionedLatch(int n){ this.nEvents = n;}
 public void aspetta() throws InterruptedException {
  synchronized (this) {
   while (nEvents >0){
    wait();
   }
  }
 }

 public void esegui(){
  synchronized (this) {
   if (nEvents >0)  {
    nEvents--;
    if(nEvents ==0) {
     notifyAll();
    }
   }
  }
 }
}
```

Analizziamo i tre codici

Il CPUkiller farà la *felicità* di chi sostiene che *syncronizzare* il codice "rallenta": solo un blocco è synchronized :–P :–P

La versione "non proprio cpukiller" ne è un evoluzione, invece di ammazzare la CPU con cicli vuoti (in)utili solo per levare di mezzo un synchronized il metodo mette a dormire il Thread corrente.

La versione *oldfashioned* è un semplice esempio di latch *pre–java5.*

Da notare

Nei primi due codici la variabile *nEvents* ha il modificatore volatile.

In tutti i casi il test per “mettere a nanna” un Thread viene fatto in un ciclo while. (guai a fare altrimenti)

L'ultimo codice utilizza la *notifyAll()*. (guai a fare altrimenti, *Pericolo Will Robinson, Pericolo*)

Sul ciclo while

Non eseguire il test del verificarsi della *Condition* all'interno di un ciclo (*while* o altro) ma usando un banale *if* può portare a situazioni spiacevoli.

Ad esempio prima di riacquisire il mutex sul quale si è effettuato il wait, un altro Thread potrebbe modificare la *Condition* non rendendola più valida.

Attenti allo *spurious-wakeup*

Rischio *intrinseco* di Deadlock?

Non c'è un rischio di *deadlock* in quanto l'invocazione di *Pippo.wait()* fa si che il Thread corrente venga sospeso e rilasci il *lock* dell'oggetto **Pippo** in maniera atomica.

Appena "svegliato" il Thread dovrà ri–acquisire tale *lock* prima di andare avanti. (Questo meccanismo è nascosto al programmatore)

Break

Facciamo qualche passo indietro

Verrà fatta luce su ciò che sta dietro al modello *wait/notify*.

Dipendenza dallo stato

Esistono molte classi *state-dependent* ovvero che possiedono operazioni con *state-based-preconditions*.

Ad esempio non è possibile rimuovere un elemento da una coda vuota o inserirne uno in una coda piena, le precondizioni sono rispettivamente *coda non vuota* e *coda non piena*.

(Per stato come al solito si intende una serie di variabili possedute dall'oggetto che di norma rispondono ad una certa invariante)

Implicazioni sui programmi sequenziali

Se si invoca un metodo in un programma con un singolo Thread e fallisce una precondizione del tipo “il *conpool* non è vuoto” , questa non diventerà mai vera; classi in programmi sequenziali possono quindi *lanciare un fallimento* quando non si verifica una precondizione.

Implicazioni sui programmi multithread

Il discorso precedente non è vero per i programmi multithread, se fallisce una precondizione basata sullo stato non è detto che questa sia una situazione terminale. In un connection–pool per esempio un Thread potrebbe rilasciare una connessione precedentemente acquisita.

Come comportarsi?

- Fallire. (esempio: lanciare un'eccezione)
- Aspettare.

E' necessario capire se la precondizione è un caso eccezionale o meno: la coda vuota è un caso eccezionale quanto lo è il colore rosso per il semaforo, né più né meno!

La decisione dipende dal contesto.

Quale comportamento scegliere?

E' necessario capire se la precondizione è un caso eccezionale o meno: la coda vuota è un caso eccezionale quanto lo è il colore rosso per il semaforo, né più né meno!

La decisione dipende dal contesto.

Svantaggi della segnalazione di un errore

- Non sempre l'errore viene segnalato a seguito di un caso eccezionale. (i semafori spesso sono rossi)
- Obbliga il chiamante a gestire la situazione.

Metodi bloccanti

Se non si sceglie la strada della segnalazione di un errore allora si deve scegliere quella dei metodi *bloccanti,* ovvero metodi che fermano il chiamante fino al verificarsi di una certa condizione.

(Più avanti vedremo che esistono versioni "temporizzate")

Dormo o sto sveglio?

Per mettere in attesa il chiamante si può adottare una tecnica nota come *busy waiting*, oppure è possibile mettere per un certo periodo di tempo in *sleep* il Thread associato.

Il *busy waiting* è stato mostrato nel CPUKillerLatch, il suo scopo è quello di ripartire “prima possibile”, spesso a scapito della CPU. D'altra parte se il periodo di *sleep* è troppo elevato si rischia di far ripartire troppo tardi il chiamante.

Condition Queue

Le Condition Queue (CQ, variabili Condition o semplicemente Condition) sono code che non contengono dati applicativi, ma Thread in attesa del verificarsi di una Condizione.

Eseguire *pippo.wait()* mette il Thread corrente nella coda di wait (Condition queue) dell'oggetto **pippo**.

Una successiva chiamata a *pippo.notifyAll()* risveglia tutti i Thread presenti nella coda.

Esempio casalingo

Le CQ sono come la campanella che avvisa che il toast è pronto. Se siete in *attesa* della *notifica* verrete prontamente avvertiti dal segnale acustico.

Se siete usciti di casa potreste aver perso la notifica oppure no.

Comunque sia è possibile osservare lo stato della macchinetta e decidere se restare ad aspettare la campanella o mangiare il toast.

Nota

Abbiamo già visto che un oggetto Java può fungere da *lock*, può anche funzionare da CQ: *wait()*, *notify()* e *notifyAll()* costituiscono le API per le CQ *intrinseche*.

Per la precisione ...

Un oggetto che funge da *lock* ed ha a disposizione delle *Condition Queues* viene chiamato *Monitor*.

Ogni oggetto Java è quindi un *Monitor*.

Un po di chiarezza

Per chiamare uno dei tre metodi delle CQ su di un oggetto X, è necessario possedere il *lock intrinseco* di X.

Questo perché il meccanismo per eseguire metodi con notifica ed attesa basati sullo stato sono strettamente legati al meccanismo per garantire una corretta (senza se e senza ma) consistenza di tale stato.

Utilizzo delle Condition queues

Le CQ permettono di realizzare efficaci algoritmi *state-dependent*, ma possono essere facilmente utilizzate in modo scorretto in quanto tutta una serie di regole non sono forzate dal compilatore o dalla piattaforma.

Se non si usa *X.wait()* in un blocco synchronized(X) il codice compila ugualmente, ma a *runtime* viene lanciata una IllegalMonitorStateException.

Attenzione!

Se non si usa *X.wait()* in un blocco synchronized(X) il codice compila ugualmente, ma a *runtime* viene lanciata una **IllegalMonitorStateException.**

Il condition *predicate (CP)*

La chiave per un buon utilizzo delle CQ è identificare correttamente quale è il *condition predicate*, ovvero la condizione per l'attesa della quale un Thread viene messo in wait.

Il suo utilizzo non viene menzionato né nelle API né nelle specifiche del linguaggio.

Il CP è quella precondizione che rende un'operazione *state-dependent*.

Esempi

- Per un'operazione *get* di una coda il predicato è “la coda non è vuota”
- Per un operazione *put* il predicato è “la coda non è piena”

Questi predicati sono espressioni booleane costruite con lo stato della classe.

Per esempio una coda limitata potrebbe avere due variabili di stato *maxSize* e *numElements* che permettono di comporre i due predicati sopra elencati.

Buona norma

Documentare sempre il *condition predicate* associato ad una *condition queue* e le operazioni di wait su di esso.

Menage a trois

C'è un'importante relazione a tre tra *locking*, il metodo *wait()* ed il CP

- Il CP coinvolge variabili di stato.
- Le variabili di stato devono essere protette da un *lock*.
- Prima di testare il CP (per eseguire una *wait()*) è necessario possedere tale *lock*.

Il *lock* e la CQ devono essere lo stesso oggetto.

Problema *intrinseco*

- Una singola *intrinsic condition queue* può essere usata con più di un *condition predicate*.
- Un solo oggetto funge sia da *lock* che da "*Condition*".

Intrinseco?

Si, vedremo più tardi che esiste anche una versione *esplicita*. Per questo ho abusato del termine.

Allarmi multipli

Un elettrodomestico combinato potrebbe avere una singola campanella per avvertire che il toast è pronto o che il caffè è caldo.

Se un thread in attesa di un CP su un oggetto X viene svegliato da una *notify*()* non è detto che si sia verificato il predicato per cui è rimasto in attesa. Potrebbe essere il toast o il caffè.

Utili regole (imposizioni!)

- Avere sempre un CP da testare basato sullo stato dell'oggetto.
- Sempre testare tale predicato prima di chiamare una *wait()* e dopo l'uscita dal blocco che la contiene.
- .. quindi ... chiamarlo sempre in un loop che controlla il predicato.
- Assicurarsi che lo stato del predicato sia guardato da un *lock.*
- Ottenere il *lock* che protegge lo stato mentre si chiamano *wait()*, *notify()* e *notifyAll().*
- Non rilasciare il *lock* dopo aver testato il predicato se poi si deve agire su di esso. (notify!)

Forma canonica (usatela!)

```
void metodoStateDependent() throws
InterruptedException {
    synchronized(myLock) {
        while(!conditionPredicate()) {
            myLock.wait();
        }
    }
}
```

Notifica

Ogni volta che si entra in wait su una condizione è necessario assicurarsi che qualcun'altro esegua una notifica ogni volta che il predicato diventa vero.

(Vi ricordate i problemi di vitalità? ... “qualcosa di buono prima o poi accade” .....)

Quando usare notify al posto di notifyAll

- **Uniform waiters**: solo un predicato è associato alla CQ e tutti i Thread coinvolti seguono la stessa logica.

..... E .....

- **One-in, One-out**: Una notifica su una *condition* abilita *al più* un Thread a procedere oltre.

Break

Il synchronized rallenta tutto! Quindi non lo uso!

- Che un codice multithread de-sincronizzato "viaggi" (non si sa bene verso quale meta) più velocemente di un codice correttamente sincronizzato, beh, ci sono pochi dubbi. *Ma anche del codice al quale viene tolto un (utile) ciclo while va più veloce del (corretto) codice che contiene il ciclo .....*
- Tuttavia ci sono casi in cui bloccare indistintamente tutti i Thread può essere un vero e proprio collo di bottiglia.

Che fare?

Fino ad ora abbiamo “fermato” tutti i Thread in attesa di una risorsa senza sapere cosa questi dovessero fare con essa.

Prendiamo un esempio “reale”: sul filesystem quando si *locka* un file è possibile distinguere tra le operazioni di lettura e di scrittura; perché non fare una cosa simile con i Thread?

N Thread potrebbero accedere “in lettura” ad una risorsa a patto che non vi sia nessun Thread “scrittore” pendente o in corso.

Un ReadWriteLock “rozzo”

```
public class LockLetturaScrittura {
	private int nlettori = 0;//vale -1 quando c'e' uno scrittore
	private int nscrittori = 0;
	private Object lock = new Object();
	public void getLockLettura() {
		synchronized(lock)		{
			/**
			 * Se c'e' uno scrittore in attesa  (nscrittori != 0)
			 * o se e' in corso un lock di scrittura (nlettori == -1)
			 * metto a dormire il thread
			 */
			try	{
				while((nlettori == -1) || (nscrittori != 0)){
					lock.wait();
				}
			}
			catch(java.lang.InterruptedException e){}
			nlettori++;
		}
	}
```

```
        public void getLockScrittura()      {
                synchronized(lock)          {
                        nscrittori++;
                        try                 {
                                while(nlettori != 0){lock.wait();}
                        }
                        catch(java.lang.InterruptedException e){}
                        nscrittori--;
                        nlettori = -1;
                }
        }
        public void rilasciaLock()   {
                synchronized(lock)          {
                        // se non ce ne erano allora non faccio niente
                        if(nlettori == 0)       return;
                        // se c'erano letture pendenti setto a zero
                        if(nlettori == -1) {nlettori = 0;}
                        else {
                                //altrimenti( almeno un lettore bloccato) decremento di uno
                                nlettori--;
                        }
                        lock.notifyAll();
                }
        }
}
```

Commenti

E' codice puramente didattico e non testato!

Più lettori accedono *concorrentemente* con un piccolo sovraccarico.

Se ci sono scrittori in attesa i lettori pendenti si bloccano.

Uno scrittore “passa” il *lock* se non ci sono né altri scrittori, né lettori attivi.

Non è noto staticamente chi “passa” prima e chi “passa” dopo.

A cosa serve tutto questo?

In generale i ReadWriteLock, i Latch e simili sono meccanismi di sincronizzazione e di comunicazione tra processi.

Sono un utilissimo strumento che non viene quasi mai utilizzato.

Performance?

I ReadWriteLock sono un ottimo meccanismo che migliora le performance di un classico “blocco sincronizzato”.

Se si ha paura del synchronized suggerisco quando possibile una tecnica di *preloading* (con la dovuta “*sincronizzazione*”) al fine di evitare successive chiamate sincronizzate.

RI–Performance

Se un oggetto è immutabile e **perfettamente** costruito non è necessario eseguire su di esso alcuna sincronizzazione.

Utilizzare insiemi di oggetti immutabili e perfettamente inizializzati senza gestire la sincronizzazione degli accessi è possibile e sicuro a patto che venga effettuato un *preloading* sicuro una sola volta e che il codice che gestisce il *preloading* sia sicuro. (Esistono comunque altre tecniche .....)

Inizializzazione

- L'inizializzazione di qualsiasi cosa non è *automagicamente* sicura.
- Esiste un (solo?) *automagico* inizializzatore (Thread-safe e *only-once*) in Java, ma non viene quasi mai utilizzato.
- Esiste una keyword che *automagicamente* garantisce (potenziale) massima performance e limitata (se l'oggetto è mutabile) Thread-safety. ... quale è? ....

Don't try this at home!

Usare "*antipattern*" della concorrenza che (forse) funzionano con "*1000 se e 1000 ma*" ovviamente non documentati.

Usare questi al posto di vera immutabilità e corretta inizializzazione per non sincronizzare l'accesso ad alcune risorse.

Rammarico

Spesso una piccola keyword di 5 lettere .....

Pausa

Domande?

Parte II/b

Problematiche ricorrenti

Argomenti trattati

- InterruptedException
- ThreadLocal
- ThreadPool e Work Queues
- Algoritmi non bloccanti

Gestire l'InterruptedException

Molti metodi come *sleep*() e *wait*() lanciano l'InterruptedException. Questa non può essere semplicemente ignorata (checked Exception), cosa ci si deve fare allora?

- Buttarla giù?
- Rilanciarla?
- Notificare l'accaduto? (a chi?)

Non *trangugiate* l'InterruptedException

A meno che non sia l'ultima istruzione eseguita dal Thread!

Rilanciare l'InterruptedException

Semplice ma non sempre possibile (interfacce, ... metodo *run()*, ...)

Metodi bloccanti 1/

Quando un metodo lancia l'InterruptedException significa che è un **metodo bloccante** e che, se chiesto con gentilezza, proverà a sbloccarsi ed a fare un e*arly return*.

Un metodo bloccante è diverso da un metodo che impiega molto tempo per essere eseguito.

Metodi bloccanti 2/

Il tempo di completamento di un metodo ordinario dipende da:

- Quantità del lavoro
- Risorse di calcolo a disposizione

Il tempo di completamento di un metodo bloccante dipende da:

- eventi esterni (I/O, ...)
- altri Thread (*lock*, *wait* ecc .....)

Metodi bloccanti 3/

Poiché i metodi bloccanti possono durare all'infinito (se restano in attesa di un evento che mai accadrà) è necessario che questi siano in qualche modo *cancellabili*.

Metodi bloccanti 4/

Quanto un metodo lancia l'**InterruptedException** sta dicendo che se il Thread che esegue il metodo viene interrotto, questo proverà a smettere di fare quello che stava facendo e ad eseguire una *early return* rilanciando l'eccezione.

Interruzione del Thread 1/

Ogni Thread ha associata una proprietà booleana che rappresenta il suo *interruption status* che inizialmente vale false.

Interruzione del Thread 2/

Quando un Thread viene interrotto da un altro tramite la chiamata Thread.*interrupt()* accade una delle seguenti cose:

- Nel caso di *wait()*, *sleep()* o *join()*, il metodo si sblocca e rilancia l'InterruptedException.
- Altrimenti viene solamente settato l'*interruption status*.

Interruzione del Thread 3/

Il codice del Thread interrotto può effettuare un *polling* sullo stato e decidere poi cosa fare.

Il controllo viene effettuato tramite il metodo *isInterrupted()*.

Lo stato di interruzione può essere controllato e riazzerato col metodo (che nome poco azzeccato) *interrupted()*.

L'interruzione è un meccanismo cooperativo! 1/

Se un Thread viene interrotto questo non necessariamente deve fermarsi.

L'interruzione è un modo gentile di chiedere ad un Thread di smettere di fare quello che sta facendo.

I metodi tipo *sleep()* e *wait()* prendono queste richieste molto sul serio.

Altri metodi possono effettuare un *polling*. Non reagire alle richieste di interruzione può compromettere la responsività.

L'interruzione è un meccanismo cooperativo! 2/

Il maggior beneficio del meccanismo cooperativo è la possibilità di applicare una politica più flessibile per la cancellazione dell'attività.

Raramente vogliamo che un'attività venga interrotta immediatamente: le strutture dati potrebbero rimanere “sporche”.

L'interruzione cooperativa permette la realizzazione di fini politiche di ripulitura .....

Gestire l'eccezione

Se un metodo lancia l'**InterruptedException** allora è un metodo bloccante.

Se un altro metodo chiama un metodo bloccante allora anche questo è un metodo bloccante! Deve quindi gestire l'**InterruptedException**.

Rilanciare l'eccezione 1 /

```
public class CodaDiTask {
 private final CodaBloccante coda = new CodaBloccante();

 public void aggiungiTask(Task t) throws InterruptedException{
  coda.put(t);
 }

 public Task prendiTask() throws InterruptedException{
  return coda.get();
 }
}
```

Rilanciare l'eccezione 2/

Non sempre è possibile.

Cosa fare?

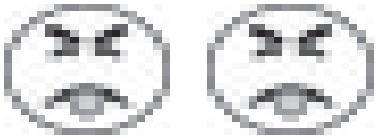
- Buttare giù l'eccezione? :-P
- Notificare l'accaduto

Impostare lo stato di interruzione del Thread

```
public class Esecutore implements Runnable{
 public final CodaDiTask queue ;
 public Esecutore(CodaDiTask c){
  queue = c;
 }
 public void run() {
  try {
   while(true){
    Task t = queue.prendiTask();
    t.esegui();
   }
  } catch(InterruptedException ie){
   Thread.currentThread().interrupt();
  }
 }
}
```

Non tracannatevi l'eccezione!

```
public class EsecutoreMalvagio implements Runnable{
 public final CodaDiTask queue ;
 public EsecutoreMalvagio(CodaDiTask c){
  queue = c;
 }
 public void run() {
  try {
   while(true){
    Task t = queue.prendiTask();
    t.esegui();
   }
  } catch(InterruptedException swallowed){}
   faiAltreAzioni();
 }
 // .....
}
```



Implementare task non interrompibili

```
public Task prendiTask(CodaDiTask c){
 boolean interrupted = false;
 try {
  while(true){
   try {
    return c.prendiTask();
    } catch (InterruptedException e) {
    interrupted = true;
   }
  }
 } finally {
  if (interrupted){
  Thread.currentThread().interrupt();
 }
 }
}
```

ThreadLocal 1/

Ogni Thread Java, oltre che ad un proprio Stack, dispone di un'area privata nella quale memorizzare variabili locali.

ThreadLocal 2/

La classe java.lang.ThreadLocal permette di definire variabili locali al Thread.

L'utilizzo tipico è come variabile static private in classi che vogliono associare informazioni ad un Thread (userID, transactionID, ...)

ThreadLocal 3/

Vediamone i metodi

- *get()* recupera il valore corrente
- *initialValue()* setta il valore iniziale
- *remove()* pulisce il ThreadLocal (*leakage*)
- *set(**T**)* setta il valore nel ThreadLocal

ThreadLocal 4/

```
public class SerialNum {
private static int nextSerialNum = 0;
    private static ThreadLocal<Integer> serialNum =
            new ThreadLocal<Integer>() {
        protected synchronized Integer  initialValue() {
            return new Integer(nextSerialNum++);
        }
    };

    public static int get() {
        return serialNum.get();
    }
}
```

Diventare atomici 1/

L'approccio standard per accedere in modo sicuro a risorse condivise è il *locking*.

Problemi derivanti:

- Scalabilità.
- Thread bloccati.
- Tempo di tenuta del *lock* può essere elevato.

Annuncio di servizio: l'acquisizione di un *lock* non conteso non è più costosa sulle moderne JVM.

Diventare atomici 2/

Le variabili volatile possono essere d'aiuto solo in alcuni casi

- Implicano una forte coerenza con la memoria centrale
- Non possono essere utilizzate con operazioni non atomiche

Diventare atomici 3/

```
public class SyncCounter {
 private int value;
 public synchronized int getValue() {return value;}
 public synchronized int increment(){ return ++value;}
 public synchronized int decrement() {return --value;}
}
```

Diventare atomici 4/

Anche un semplice contatore deve essere realizzato con il *locking*.

Figuriamoci qualcosa di più complicato .....

Diventare atomici 5/

Supponiamo che esista un metodo **CompareAndSwap**

"Io penso che nella locazione di memoria V ci sia il valore A, se è così sovrascrivilo con B. In ogni caso fammi sapere quale era il valore di V"

.... supponiamo che esegua questo in modo atomico e senza alcun *locking*!

Diventare atomici 5/

curr = **CompareAndSwap**(V,A,B)

- **V** locazione di memoria o variabile
- **A** valore atteso
- **B** valore che può sostituire A
- **curr** quello che è presente in V (può essere A)

Diventare atomici 6/

```
public class CasSimulato {
 private int value;
 public synchronized int getValue() {return value;}
 public synchronized int compareAndSwap(int atteso, int nuovo){
  int vecchio = value;
  if(vecchio == atteso)
   value = nuovo;
  return vecchio;
 }
}
```

Diventare atomici 7/

```
public class ContatoreCAS {
 private final CasSimulato valore = new CasSimulato();
 public int getValore(){return valore.getValue();}
 public int increment(){
  int vecchio = valore.getValue();
  while (valore.compareAndSwap(vecchio, vecchio +1) != vecchio){
   vecchio = valore.getValue();
  }
  return vecchio +1;
 }
}
```

Diventare atomici 8/

Purtroppo il CAS utilizza il *locking* in tutti i suoi metodi e Java non mette a disposizione un CAS veramente atomico e senza *locking* ...

Diventare atomici 9/

..... oppure si?

Work Queue 1/

Come implementare un meccanismo *client/server*?

Work Queue 2/

```
public class SingleThreadWebServer {

    public static void main(String[] args) throws
IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
            }
    }
}
```

Work Queue 3/

Un solo Thread gestisce sequenzialmente tutte le richieste!

Limitazione della responsività del sistema.

Work Queue 4/

```
public class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```

Work Queue 5/

Un Thread per richiesta.

Può affogare la macchina.

Work Queue 6/

```
public class WorkQueue {
 private final int nThreads;
 private final PoolWorker[] threads;
 private final LinkedList queue;
 public WorkQueue(int nThreads){
  this.nThreads = nThreads;
  queue = new LinkedList();
  threads = new PoolWorker[nThreads];
  for (int i=0; i<nThreads; i++) {
   threads[i] = new PoolWorker();
   threads[i].start();
  }
 }
 public void execute(Runnable r) {
  synchronized(queue) {
   queue.addLast(r);
   queue.notify();
  }
 }
```

Work Queue 7/

```
private class PoolWorker extends Thread {
 public void run() {
 Runnable r;
 while (true) {
  synchronized(queue) {
   while (queue.isEmpty()) {
    try {
     queue.wait();
    }
    catch (InterruptedException ignored){}

   }
   r = (Runnable) queue.removeFirst();
  }
  try {
   r.run();
  }
  catch (RuntimeException e) {}
 }
}
}
```

Work Queue 8/ :-)

Più Thread (sempre vivi) si dividono le richieste (*leader followers*).

Aumenta della responsività del sistema.

Work Queue 9/ :-(

- Deadlock (intrinseco in tutte le applicazioni multihtread)
- Trashing delle risorse (pool troppo grossi)
- Leakage
- Sovraccarico

Parte III

Ci siamo quasi

Argomenti trattati

- Nuove API per vecchi problemi
- Alcuni comuni *antipattern*

Le API per la concorrenza

Java è un linguaggio che *nativamente* supporta la concorrenza, non mi riferisco solo al tanto *amato/odiato* synchronized ma quanto al modello *wait/notify* che usato congiuntamente alla sincronizzazione permette la costruzione di primitive di concorrenza presenti in altri linguaggi solo come “moduli esterni”. (posix threads, ...)

Fino alla 1.4 inclusa

La non esistenza di una API standard ha fatto si che ne nascessero di ottime come quella della S.U.N.Y. Oswego (di Doug Lea). Quindi o si usavano le primitive Java, o si usava una API esterna o ci si cimentava nella scrittura di costrutti comuni per la concorrenza. (latch, lettori/scrittori, ...)

Java5

A partire da java5 (1.5 o Tiger) è disponibile una API apposita per la concorrenza **java.util.concurrent API** che sfrutta aggiunte apportate alla JVM che rendono più sicura e performante la programmazione in ambiente multithread.

Molti degli esempi riportati nelle *slide* precedenti sono presenti come interfacce e classi del package **java.util.concurrent**

Strutture dati

Le vecchie versioni di Java mettevano a disposizione alcune collezioni "*sincronizzate*" come HashTable, Vector e Stack.

Java5 mette a disposizione versioni più performanti come ConcurrentHashMap e ConcurrentLinkedQueue.

Si raccomanda quindi di sostituire le vecchie classi con le nuove.

Nuovi costrutti

Oltre alle strutture dati java5 mette a disposizione nuovi costrutti per la concorrenza che sono versioni più performanti (basate su aggiunte alla JVM) di quelli “vecchi”. Esistono infatti *lock espliciti* e classi *Condition esplicite*.

Panico! Devo imparare tutto da capo

Fortunatamente la semantica dei nuovi costrutti è molto simile a quella dei vecchi.

Ed il JMM?

Il JMM mantiene la stessa semantica dei vecchi costrutti. Non è presente neanche questa volta la regola che dice che *ogni Thread vede istantaneamente tutte le modifiche fatte dagli altri Thread senza se e senza ma*.

Per fortuna!

Code bloccanti

Java5 nel pacchetto *concurrent* mette a disposizione le *BlockingQueue*, il funzionamento è il seguente

- Se la coda è piena, tutti i Thread in attesa di inserire elementi restano bloccati fino a quando non si libera una posizione.
- Se la coda è vuota, tutti i Thread in attesa di recuperare elementi restano bloccati fino a che non viene inserito un elemento.

La *BlockingQueue* lancia l'**InterruptedException**!

Produttore / consumatore

Come implementare il pattern producer/consumer con una BlockingQueue?

Potrebbe essere più semplice del previsto.

Sincronizzatori
AbstractQueueSynchronizer

Un sincronizzatore è qualsiasi oggetto che, basandosi sul suo stato interno, coordina il controllo di flusso dei Thread.

Tutti i sincronizzatori condividono certe proprietà:

- Incapsulano informazioni di stato che determinano se un insieme di Thread che "arrivano" al sincronizzatore possono passare oltre oppure devono attendere.
- Forniscono metodi per manipolare lo stato.
- Forniscono metodi efficienti per aspettare che il sincronizzatore arrivi nello stato desiderato.

Alcuni sincronizzatori

- Latch
- Semafori
- Barriere
- *Lock* espliciti
- *Condition* esplicite

Latch

Un Latch (lo abbiamo visto prima) è un sincronizzatore in grado di ritardare l'avanzamento di Thread fino a che esso non raggiunge il suo stato *terminale*.

Funziona come un cancello: fino a quando non si arriva allo stato terminale (cancello *sempre* aperto) non si passa. Di norma si utilizza quando è necessario che alcune attività aspettino il verificarsi di altre attività di tipo "*only-once*".

Esempi di utilizzo del Latch (quando)

- Assicurarsi che una computazione non vada avanti fino a quanto tutte le sue risorse non sono state inizializzate.
- Assicurarsi che un servizio non parta fino a quando non siano partiti tutti i servizi da cui esso dipende.
- Aspettare fino a quanto tutte le parti coinvolte non sono pronte per procedere.

NOTA: il Latch è strettamente *monotono*

In pratica

La classe *java.util.concurrent.CountDownLatch* implementa il comportamento del classico Latch.

Nel costruttore richiede un numero intero (maggiore di ZERO!) ed è il numero di “scatti” che deve effettuare il Latch prima di essere sbloccato.

Esempio di utilizzo (come)

```
public class WaitForME {
 static Logger l =
Logger.getLogger(WaitForME.class.toString());
 private final CountDownLatch cdl = new CountDownLatch(1);
 public void init() throws InterruptedException {
 //computazione lunga tipo
  l.severe("parte init");
  try {
   Thread.sleep(10000);
  } //non dovrebbe stare qua, dipende dal contesto ...
  finally {
   l.severe("inizializzato! o nel bene o nel male");
   cdl.countDown();}
  }
 public void faiQualcosa() throws InterruptedException {
  l.severe("inizio");
  cdl.await();
  l.severe("finito");
 }
}
```

Semafori

I semafori sono usati per controllare il numero di attività che possono accedere ad una certa risorsa o eseguire una certa azione nello stesso momento.

Un semaforo gestisce un insieme di *permessi* il cui numero viene definito a tempo di costruzione. I Thread possono *acquisire* permessi fino a quando ne ce sono di disponibili, oppure possono *rilasciarli* (restituendoli al semaforo) quando hanno completato la computazione.

Un Thread che tenta di acquisire un *permesso* resta in attesa fino a quando non lo riceve.

Per maggiori info: ***java.util.concurrent.Semaphore***

Quando?

- Pool di risorse. (solo un certo numero alla volta è disponibile)
- Simulare un *lock* esclusivo gestendo un solo *permesso* alla volta.

Nota

In generale sono disponibili versioni “temporizzate” dei metodi che mettono un Thread “in attesa” di una notifica.

Se nel tempo passato come parametro al metodo non si ottiene una notifica, il metodo termina restituendo *false*.

Barriere

Le barriere sono simili ai Latch in quanto bloccano un gruppo di Thread fino a quando non si verifica una condizione. La differenza è che per andare avanti, *tutti* i Thread devono essere arrivati alla barriera.

"*Ci vediamo in piazza alle 19:00, aspettiamo fino a quando non siamo tutti arrivati e poi decidiamo dove andare .....*"

java.util.concurrent.CyclicBarrier

Permette ad un certo numero di partecipanti di incontrarsi in un *luogo prestabilito (barrier point)* ripetutamente (non è quindi monouso come il Latch) ed è utile per punti di congiunzione in computazioni parallele.

Lock espliciti

Un lock è un sincronizzatore utile per controllare l'accesso ad una risorsa condivisa da più Thread. Di norma un *lock* fornisce accesso in mutua esclusione alla risorsa, solo un Thread alla volta può acquisire il *lock* e quindi accedere alla risorsa, ovviamente solo se prima si forza la sua acquisizione.

Alcuni tipi di *lock* potrebbero garantire, sotto opportune condizioni, l'accesso concorrente alla risorsa condivisa. (I ReadWriteLock)

Ma non c'era il synchronized?

Diversamente dai *lock intrinsechi*, i *lock espliciti* offrono un' acquisizione che può essere *temporizzata*, *condizionale* e *interrompibile*, le operazioni di *lock()* ed *unlock()* sono esplicite e *possono* essere eseguite in blocchi differenti.

Utilizzano la stessa semantica di visibilità della memoria del *lock* intrinseco, ma possono usare una diversa e selezionabile politica di acquisizione e rilascio.

Generalmente hanno performance migliori.

java.util.concurrent.locks.ReentrantLock

Il ReentrantLock garantisce la stessa semantica del blocco synchronized, sia in acquisizione che in rilascio.

(Importante: se si possiede un *lock* X e si prova a riacquisire il *lock* X non si ha deadlock)

Un vecchio esempio

```
import java.util.concurrent.locks.ReentrantLock
import it.linux.prato.esempi.thread.MutableSensibleData;
public class ExampleOfSimpleLock {
	private final ReentrantLock sl = new ReentrantLock();
	private final MutableSensibleData msd = new MutableSensibleData();
	public void readData() throws InterruptedException {
		sl.lock();
		try {msd.readsomething();
		}finally {sl.unlock();}
	}
	public void writeData() throws InterruptedException {
		sl.lock();
		try {			msd.writesomething();
		}finally {sl.unlock();}
	}
	public void foo(){
		try {
			sl.lock();
			msd.readsomething();
			msd.writesomething();
		} finally {try{sl.unlock();}
	}
}
```

Come suggerimento

Non usare una politica di utilizzo del *lock* esplicito diversa da

- Acquisizione
- Immediato blocco try
- CODICE
- Immediato blocco finally contenente il rilascio del lock

La chiarezza del codice e questo schema sono garanzia di successo!

Quale scegliere

- synchronized ha un throughput minore dei *lock* espliciti in java5.
- synchronized è più noto.
- synchronized è indubbiamente meno pericoloso del lock esplicito.
- synchronized è meno flessibile.
- synchronized non ha una "*tryLock*".
- entrambi si possono "piantare" se si acquisiscono e poi si mette "a nanna" il Thread che li possiede.
- Il *lock* esplicito può avere una politica *unfair* (default) o *fair.*
- E' possibile scoprire deadlock se si usa il synchronized osservando l'output di kill –3 PIDJVM. (se i vostri sistemisti hanno qualcosa da ridire licenziateli!)

Esempio

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s has bowed to me!%n",
                    this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!%n",
                    this.name, bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Esempio di kill –3

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0809aa04 (object 0x88b420f0, a Deadlock$Friend),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0809a9c4 (object 0x88b42100, a Deadlock$Friend),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at Deadlock$Friend.bowBack(Deadlock.java:16)
        - waiting to lock <0x88b420f0> (a Deadlock$Friend)
        at Deadlock$Friend.bow(Deadlock.java:13)
        - locked <0x88b42100> (a Deadlock$Friend)
        at Deadlock$2.run(Deadlock.java:28)
        at java.lang.Thread.run(Thread.java:595)
"Thread-0":
        at Deadlock$Friend.bowBack(Deadlock.java:16)
        - waiting to lock <0x88b42100> (a Deadlock$Friend)
        at Deadlock$Friend.bow(Deadlock.java:13)
        - locked <0x88b420f0> (a Deadlock$Friend)
        at Deadlock$1.run(Deadlock.java:25)
        at java.lang.Thread.run(Thread.java:595)

Found 1 deadlock.
```

ReadWriteLock

La mutua esclusione a volte è una disciplina più forte della semplice integrità dei dati.

E' quindi una (robusta) tecnica conservativa che giustamente previene sovrapposizioni del tipo scrittore/scrittore e scrittore/lettore.

Tuttavia previene anche le sovrapposizioni lettore/lettore.

Caso ricorrente

Per dati che vengono prevalentemente letti e raramente scritti la mutua esclusione limita (ma non rende insicura) la concorrenza, su sistemi multiprocessore questo quindi limita il *throughput*.

In questi casi ci vengono in aiuto i ReadWriteLock che permettono sovrapposizioni tra lettori, ma non tra scrittori o tra lettori e scrittori.

Performance

In generale, in una situazione molti lettori/pochi scrittori i ReadWriteLock garantiscono una performance ottimale e piena sicurezza.

L'utilizzo va comunque monitorato e profilato per vedere se sono una scelta adatta. In casi estremi, a causa di una complessità maggiore, può risultare più performante la rude mutua esclusione.

java.util.concurrent.locks.ReentrantReadWriteLock

- E' rientrante. (lo dice anche il nome)
- Può funzionare in modalità *unfair* o *fair.*
- Espone due *Lock*, il *readLock* ed il *writeLock.*
- Permette il downgrading: chi possiede il *writeLock* può acquisire il *readLock* senza rilasciare il primo e senza generare deadlock.
- Non esiste l'upgrading: chi detiene il *readLock* non può acquisire il *writeLock* senza rilascio, pena il deadlock.

Esempio di utilizzo

```
public class ExampleOfRRWL {
 private final Lock rl;
 private final Lock wl;
 private final MutableSensibleData msd = new MutableSensibleData();
 public ExampleOfRRWL() {
  ReentrantReadWriteLock rrwl = new ReentrantReadWriteLock();
  rl = rrwl.readLock(); wl = rrwl.writeLock();
 }
 public void readData()  {
  rl.lock();
  try { msd.readsomething();}
  finally {rl.unlock();}
 }
 public void writeData()  {
  wl.lock();
  try {msd.writesomething();}
  finally {wl.unlock();}
 }
}
```

La novella dello stento

Devo ricordarvi del try/finally?

Spero di no :–)

java.util.concurrent.locks.Condition

Cosa saranno mai?

Esempio .....

```
public class ConditionBoundedBuffer <T> {
    protected final Lock lock = new ReentrantLock();
    // CONDITION PREDICATE: notFull (count < items.length)
    private final Condition notFull = lock.newCondition();
    // CONDITION PREDICATE: notEmpty (count > 0)
    private final Condition notEmpty = lock.newCondition();
    private static final int BUFFER_SIZE = 100;
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    private int tail, head, count;

   ..........................................
```

.....

```
// BLOCCATO-FINO-A-CHE: notFull
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[tail] = x;
            if (++tail == items.length)
                tail = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
.....
```

```
  // BLOCCATO-FINO-A-CHE: notEmpty
    public T take() throws
InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

Note

Lasciate stare l'implementazione ...

Le *Condition* vengono “generate” dal solito *Lock.* (*newCondition()*)

Lo stato è protetto dal solito *Lock* che ha generato le Condition.

Le notifiche di una *Condition* non vengono inviate a chi aspetta l'altra *Condition.*

Mmmmmm

Vale sempre il menage à trois.

Lock, Latch, Condition, ...

Quale metodo devo richiamare per mettermi in attesa?

Risposta

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa await()

Con la "a" di fronte

Meglio una in più che una in meno!

In generale per le nuove API

- Acquisizione *lock* condizionata. (*tryLock()*)
- “Acquisizione” temporizzata. (se timeout allora ritorna false ed esco)
- Politica *unfair* o *fair.*

Parte III/b

Comuni *Antipattern*

Monito

Tra il Giugno 1985 ed il Gennaio 1987 una macchina computerizzata per la radio terapia chiamata Therac–25 fu causa di sei incidenti noti che causarono morte e lesioni gravi. Sebbene la maggioranza degli incidenti siano stati causati prevalentemente dall'interazione errata delle complesse componenti, la programmazione concorrente ha giocato un ruolo importante in questi sei incidenti. Race condition tra le differenti attività delle componenti hanno prodotto output di controllo errati.

Inoltre, la natura sporadica di questo tipo di eventi ha contribuito al ritardo nel rilevamento del problema.

I progettisti del software del Therac–25 sono risultati essere totalmente inconsapevoli dei principi e delle pratiche della programmazione concorrente.

Da "Concurrency: state models & Java programming"

Curiosando

Curiosando nelle comunità che trattano di questioni legate al multithreading ed alla concorrenza possiamo imbatterci in due termini

- *Antipattern*
- Broken Idiom

Cosa indicano?

Non vogliono essere definizioni formali o riconosciute universalmente, la mia opinione è che un *antipattern* sia un modo di risolvere un problema tramite *cattive regole* o schemi, un *broker idiom* invece è un *antipattern* basato su schemi, linguaggi o “idiomi” insicuri!

In pratica?

A discapito del loro vero significato, un *antipattern* e *broken idiom* sono tecniche da evitare!

Antipattern

D'ora in poi verrà utilizzato solo il termine Antipattern.

Vediamone qualche esempio all'opera!

Antipattern 1: Nessuna sincronizzazione!

```
public class NoSyncAP {
 private MutableSensibleData msd ;
 private HashMap map;
 public NoSyncAP(){
  msd = new MutableSensibleData();
 }
 public MutableSensibleData getMsd() {return msd;}
 public void setMsd(MutableSensibleData msd) {this.msd = msd;}
 public Object get(Object key) {return map.get(key);}
 public Object put(Object key, Object value) {return map.put(
    key, value);}
 public HashMap getMap() {return map;}
 public void setMap(HashMap map) {this.map = map;}
}
```

Nota

Se l'oggetto è utilizzato in un vero ambiente monothread non ci sono problemi.

Niente è protetto da un *lock*.

Non è neppure documentato come fare ad usare un minimo di *client–side–locking*.

C'è un bug! :–) .. magari il creatore della classe (non) ha documentato il fatto che è necessario chiamare una *setMap*() dopo aver costruito l'oggetto. (Non è detto che le due istruzioni vengano eseguite sequenzialmente ed atomicamente)

Antipattern 2: Escaping di oggetti mutabili

In **NoSyncAP** si nota che l'HashMap interna è visibile all'esterno e potenzialmente può “fuggire”.

Problemi derivanti dall'escaping

Se l'oggetto è mutabile qualunque Thread è in grado di alterarne lo stato. (potrebbe non rispettare più le invarianti poste dall'oggetto che lo conteneva)

Se l'oggetto non è Thread–safe può succedere di tutto. (anche se l'oggetto che lo conteneva gestiva correttamente la sincronizzazione)

L'oggetto contenitore e l'oggetto che entra in possesso dell'oggetto fuggito possono non utilizzare il solito protocollo di sincronizzazione.

Soluzioni (quando possibili)

- Non far fuggire oggetti mutabili o contenenti oggetti mutabili.
- Far fuggire solo copie dell'oggetto mutabile.
- Se l'oggetto mutabile deve essere condiviso allora condividerne una versione sincronizzata.
- Al limite documentare bene cosa è permesso fare con l'oggetto fuggito.

Realizzare versioni atomiche

- Tramite *delegation/wrapper/decorator.*
- Con opportune classi già predisposte. (Collections.synchronizedMap)
- Per variabili “singole” ci aiutano le classi *java.util.concurrent.atomic.Atomic*.*
- *Rendendo lo stato privato (già buon norma OOP) e gestendo il tutto internamente.*

Decorator/Wrapper/Adapter: classe non sincronizzata

```
public class MutableSensibleData {
    //stato mutabile .....
    public void readsomething() {
        //   legge lo stato     .......
    }

    public void writesomething() {
        // scrive lo stato .....
    }

    public void putIfAbsent(Object o) {
        // if (stato.contains(o)) stato += o
    }
}
```

Versione *wrappata* e sincronizzata :-/

```
public class SyncMutaleSensibleData extends MutableSensibleData {
 private final MutableSensibleData delegate;
 public SyncMutaleSensibleData(MutableSensibleData m){
  this.delegate = m;
 }
 public void putIfAbsent(Object o) {
  synchronized (this) {delegate.putIfAbsent(o);}
 }
 public void readsomething() {
  synchronized (this) {delegate.readsomething();}
 }
 public void writesomething() {
 synchronized (this) {delegate.writesomething();    }
 }
}
```

Note

- Viene effettuato il *client-side-locking* sull'oggetto contenitore.
- Adesso i metodi sono atomici.
- Se si aggiungono altri metodi è semplice gestire la sincronizzazione dall'esterno. (documentazione!)
- Può non essere la soluzione ideale se la classe delegata (MutableSensibleData) è condivisa da più Thread.
- E' possibile un altro tipo di *client-side-locking.*

Versione alternativa

```
public class AltSyncMutableSensibleData extends MutableSensibleData {
 private final MutableSensibleData delegate;
 public AltSyncMutableSensibleData(MutableSensibleData m){
  this.delegate = m;
 }

 public void readsomething() {
  synchronized (delegate) {delegate.readsomething();}
 }
 public void writesomething() {
  synchronized (delegate) {delegate.writesomething();  }
 }
}
```

Forma canonica

```
class GoodListHelper <E> {
    public final List<E> list =
      Collections.synchronizedList(new ArrayList<E>());
    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
    ..........................
}
```

La sincronizzazione dell'ArrayList non è sufficiente

In generale

- Capire quale tipo di sincronizzazione è più adatta.
- E' possibile utilizzare entrambe le versioni sincronizzate al posto della classe base. (ereditarietà)
- Preferire la programmazione per *interfacce* a quella per classi. (Anche in ambienti non concorrenti)

Antipattern 3: forzare lo stato in un oggetto

Se traslo un punto tramite una *trasformazione affine* (il termine non deve incutere paura!) ottengo *un altro punto*, non il solito punto con le coordinate cambiate.

Una data + 10 giorni "*non fa*" la stessa data traslata ma un'altra data.

Come rimediare

- Rendere il punto immutabile.
- Far sì che l'operazione di traslazione restituisca un nuovo punto.
- Non inserire lo stato in classi di utilità prettamente procedurali.
- Non creare nuovi tipi (mutabili!) quando possibile, preferire invece classi di utilità che operano su di esse.

Versione statefull

```
public class MutablePointAP {
 private float x;
 private float y;
 private float z;
 public MutablePointAP(float x, float y, float z) {
  this.x = x;
  this.y = y;
  this.z = z;
 }
 public MutablePointAP(){}
 public float getX() {return x;}
 public void setX(float x) {this.x = x;}
 public float getY() {return y;}
 public void setY(float y) {this.y = y;}
 public float getZ() {return z;}
 public void setZ(float z) {this.z = z;}
 public void translate(float dx, float dy, float dz){
  x +=dx; y +=dy ; z+=dz;
 }
}
```

Versione immutabile :-)

```
public class ImmutablePoint {
 private final float x;
 private final float y;
 private final float z;
 public ImmutablePoint(final float x,
  final float y,
  final float z) {
   this.x = x;
   this.y = y;
   this.z = z;
 }
 public float getX() {return x;}
 public float getY() {return y;}
 public float getZ() {return z;}
 public ImmutablePoint translate(float dx, float dy, float dz) {
  return new  ImmutablePoint(x+dx, y+dy, z+dz);
 }
}
```

Versione alternativa :-)

```
public class AffineGeometry {
 public static final ImmutablePoint translate(ImmutablePoint p,
  float dx,   float dy, float dz){
   return new ImmutablePoint(
    p.getX() + dx,
    p.getY() + dy,
    p.getZ() + dz
   );
 }
}
```

Altra forzatura

Inserire variabili di istanza in una Servlet (a parte logger di classe, contatori e poco altro) è un forzato inserimento dello stato. Spesso si inserisce come stato un insieme di variabili riguardanti la richiesta *corrente* che arriva alla *Servlet*.

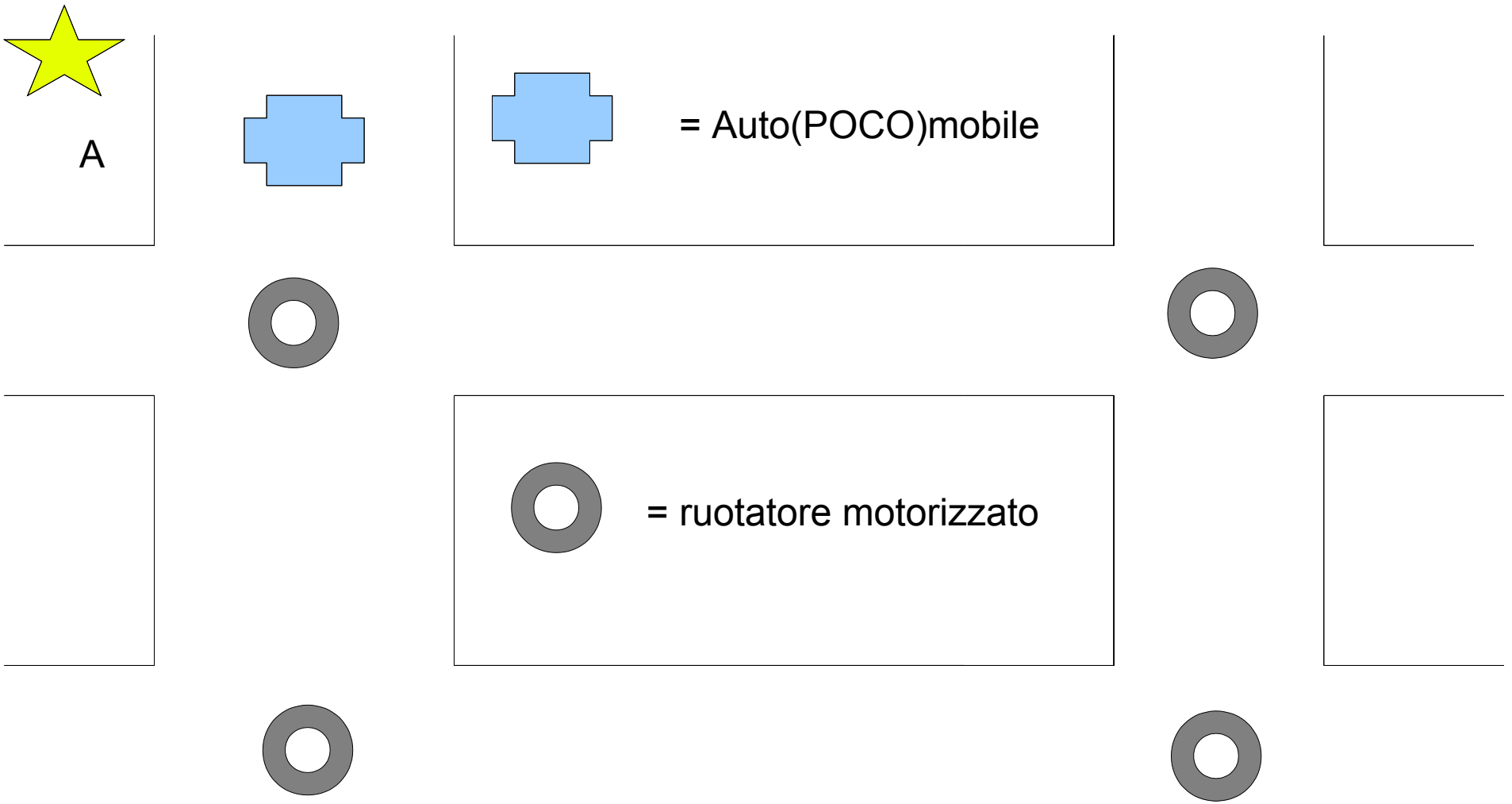
Riflettiamo: in un ambiente concorrente quale è la richiesta *corrente*?

Antipatern 4: Cercare rimedi non risolutivi

Nell'esempio della *Servlet* un rimedio non risolutivo è quello di far accedere in mutua esclusione allo stato della *Servlet* (*service()* sincronizzata).

Tuttavia la soluzione è togliere le variabili di stato in quanto lo stato non appartiene alla *Servlet* ma alla singola richiesta.

Se la macchina non curva ...



E allora?

Se l'automobile non curva non si devono realizzare strade “modello Manhattan” ed inserire ruotatori motorizzati ad ogni incrocio.

Beh, in questo modo si riesce comunque ad arrivare da qualsiasi luogo A a qualsiasi luogo B.

La soluzione tuttavia è *dotare le automobili dello sterzo!*

Antipattern 5: *Unsafe publication*

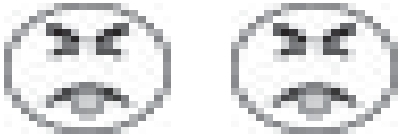
Pubblicare significa “rendere visibile” all'esterno.

Farlo in modo insicuro spesso vuol dire esporre un oggetto non perfettamente costruito.

Esempio *crudo* di pubblicazione

```
class Segreti {
     public static Set<Segreto> knownSecrets;
     public void inizializza() {
       knownSecrets = new HashSet<Segreto>();
     }
}
```

Toccata e fuga



```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) { doSomething(e); }
        });
    }
    void doSomething(Event e) {
     ..........................................
    }
.....................
}
```

Problema?

Si, nell'esempio “toccata e fuga” si lascia *potenzialmente* fuggire il riferimento a *this* prima di una effettiva e completa inizializzazione.

Una variante di questo *Antipattern* è quella in cui si fanno *esplicitamente* partire Thread nel costruttore e si passa loro qualche variabile di istanza (potenzialmente) non completamente costruita.

Soluzione possibile

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);   }   };
     }
    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

Piccolo ma *letale*!

```
public class StuffIntoPublic {
    public Holder holder;
    public void initialize() {
        holder = new Holder(42);
    }
}
```

Senza adeguata sincronizzazione questo frammento di codice può essere letale!

:-/

```
public class Holder {
    private int n;
    public Holder(int n) {
        this.n = n;
    }
    public void assertSanity() {
        if (n != n)
            throw new AssertionError("FAULT");
    }
}
```

Può accadere che *assertSanity* fallisca!

Soluzione

```
public class Holder {
    private final int n;

    public Holder(int n) {
        this.n = n;
    }

    // non fallisce MAI!
    public void assertSanity() {
        if (n != n)
            throw new AssertionError("FAULT");
    }
}
```

Consigli

Per pubblicare in modo sicuro un oggetto, sia il suo *reference* che il suo stato devono essere visibili a tutti i Thread nello stesso istante. Un oggetto propriamente costruito può essere pubblicato in modo sicuro

- Utilizzando un inizializzatore statico.
- Memorizzando il suo *reference* in un campo final.
- Memorizzando il suo *reference* in un campo volatile o in un AtomicReference.
- Memorizzando il suo *reference* in una variabile protetta da un *lock.*

NON ESISTONO ALTRE SOLUZIONI

Antipattern 6: Guardare dalla parte sbagliata

Se ad un incrocio due automobili passano col verde e "picchiano" tra di loro non è detto che abbia fallito il semaforo.

Chiedete ad entrambi gli automobilisti quale semaforo hanno guardato!

Ognuno deve guardare il semaforo giusto, solo uno è verde!

Esempio: classe contenitore

```
public class SynchedSensibleData<K,V> {
 private final Map<K,V> m = new HashMap<K,V>();
 public synchronized V get(K key){
  return m.get(key);
 }
 public synchronized void put(K key, V value) {
  m.put(key,value);
 }
 public synchronized boolean containsKey(K k) {
  return m.containsKey(k);
 }
}
```

Cattivo esempio di put if absent

```
public class BadExamplePIA<K,V> {
 private final SynchedSensibleData<K,V> delegate;
 public BadExamplePIA(SynchedSensibleData<K,V> m){
  this.delegate = m;
 }

 public synchronized void putIfAbsent( K k, V v) {
  if(!delegate.containsKey( k )){
   delegate.put(k,v);
  }
 }
}
```

Problema

- E' scorretto il *client-side-locking.*
- La SynchedSensibleData permette il *client-side-locking* ma la classe che implementa il *put if absent* esegue il *locking* sull'oggetto sbagliato.

Soluzione?

```
public class ExamplePIA<K,V> {
 private final SynchedSensibleData<K,V> delegate;
 public ExamplePIA(SynchedSensibleData<K,V> m){
  this.delegate = m;
 }

 public  void putIfAbsent( K k, V v) {
  synchronized (delegate) {
   if(!delegate.containsKey( k )){
    delegate.put(k,v);
   }
  }
 }
}
```

Antipatter 7: ?????

Non uso la keyword final in quanto “modifico l'oggetto associato”

Ignoranza pura?

Se per modificare si intende variare il *reference* allora non ci sono dubbi che la *keyword* final non possa essere utilizzata.

Tuttavia .....

Ignoranza insindacabile

FINAL (Java) == CONST (C++)

NO!

Riflettiamo

- In C++ (idiosincrasie a parte) **const** vuol dire "*stato non modificabile*".
- In Java final vuol dire "*reference non modificabile*".
- Non esiste **const** in java. Solo le classi immutabili sono ***const***–anti, e solo quelle con tutto lo stato final sono thread–safe *senza se e senza ma.*
- final implica anche molte altre cose riguardanti la visibilità della memoria (questo non viene mai preso in considerazione .... la piccola keyword di 5 caratteri .....)

Antipattern 8: Unsafe (lazy) instantiation

La *Lazy instantiation* è una tecnica mirata a ritardare il più possibile la creazione di un oggetto *costoso* da costruire.

Oltre ad essere spesso inutile in Java può anche risultare dannosa se utilizzata senza la dovuta sincronizzazione, è figlia dell' *unsafe publication*.

Schema della Lazy Instantiation

```
private MutableSensibleData instance;

public void doSomething() {
      .....
      if (instance == null) {
            instance = new MutableSensibleData();
      }
      .....
}
```

In un ambiente a thread singolo ...

In un vero ambiente monothread non ci sono problemi, anzi una sorta di *Lazy Instantiation* “ragionata” potrebbe anche essere utile. (basta non *copia&incollare* tre righe dappertutto)

Lazy Instantiation e falsi Singleton

```
public class UnsafeLazyInstatniation {
private static Resource resource;
    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
    // ---
}
```

In generale

Si nota uno scorretto utilizzo della tecnica *check-then-act*, in pratica non vi è alcuna garanzia che la risorsa venga istanziata una volta sola.

Vi ricordate la triade della morte? Qua sopra ho fatto riferimento solo alla *race condition*, ma anche le due altre componenti sono presenti, molto presenti.

... lo vedremo dopo ...

Soluzioncina ina ina .....

```
public class SafeLazyInitialization {
 private static Resource resource;
 public synchronized static Resource getInstance() {
  if (resource == null)
   resource = new Resource();
  return resource;
 }
}
```

Commenti

- Funziona.
- E' sicura.
- Richiede però una sincronizzazione ad ogni invocazione del metodo di istanziazione.
- Finalmente i Singleton sono degni di tale nome. (*Classloader* permettendo!)

I nemici dei miei avversari sono miei amici

Chi è in nemico della Lazy Instantiation?

Il *nemico* della Lazy Instantiation

```
public class EagerInstantiation {
 private static  Resource resource = new Resource();
 public static Resource getResource() {
  return resource;
 }
}
```

Commenti

- Nessuna sincronizzazione.
- Funziona!
- Questa tecnica combinata con il *Lazy Class Loading* di Java permette di simulare una *Lazy Instantiation.*
- Vi ricordate l'inizializzatore automagico? .... qua viene utilizzato!

Inizializzatori statici (automagici)

Gli inizializzatori statici vengono eseguiti dalla JVM a tempo di inizializzazione, dopo che la classe viene caricata ma prima che la classe sia utilizzata da altri Thread.

Domanda

Vi serve sul serio una inizializzazione/istanziazione Lazy?

Lazy instantiation class holder idiom

```
public class ResourceFactory {
 private static class ResourceHolder {
  public static Resource resource = new Resource();
 }

 public static Resource getResource() {
  return ResourceFactory.ResourceHolder.resource;
 }

}
```

Commenti

- E' molto Lazy.
- Non usa la sincronizzazione.
- Utilizza pesantemente l'inizializzazione statica.

Antipattern 9: forzatura della concorrenza

Un javabean “POJO” non deve essere necessariamente Thread safe.

Oggetti associati ad una singola “richiesta” non dovrebbero essere Thread–safe.

Logger contenenti dati dell'utente attualmente collegato non dovrebbero essere Thread–safe.

Ma ?

Dire che un oggetto non dovrebbe essere Thread-safe significa che non e' il tipo di oggetto che deve essere *sharato*, o meglio e' un oggetto le cui istanze non dovrebbero essere condivise.

Non partite in quinta sincronizzando internamente i POJO, se fosse necessario createne versioni decorate oppure rendete gli accessi atomici ed immediati (variabile volatile).

Ma se uso troppy synchronized?

- Potenziale *deadlock*
- Inutile spreco di risorse
- .....

NMP 1 /

```
public class NMP1 {
 class Inner {
  protected boolean cond = false;
  public synchronized void await () throws InterruptedException   {
   while (!cond){
    wait ();
   }
  }
  public synchronized void signal (boolean c)   {
   cond = c;
   notifyAll ();
  }
 }
 class Outer {
  protected Inner inner_ = new Inner (); // nota! non √¨ accessibile
all'esterno
  public synchronized void process () throws InterruptedException   {
   inner_.await ();
  }
  public synchronized void set (boolean c)   {
   inner_.signal (c);
  }
 }
}
```

NMP 2 /

```
public class Outer {
 protected Nested _nested = new Nested (); // nota! non √¨ accessibile
all'esterno
 public synchronized void process () throws InterruptedException   {
_ nested.await ();
 }
 public synchronized void set (boolean c)   {
_ nested.signal (c);
 }
 private class Nested {
  protected boolean cond = false;
  public synchronized void await () throws InterruptedException   {
   while (!cond) {
    wait (); // .....
   }
  }
 public synchronized void signal (boolean c)   {
  cond = c;
  notifyAll ();
  }
 }
}
```

Ma non ci sono troppi sinchronized?

Beh ... si ma quale leviamo?

Quello interno? No!

Quello esterno? ... possibile

Beh ... lasciamoli entrambi

Nested Monitor Problem

Non utilizzare entrambi i lock!

Solo quello del monitor viene rilasciato durante il wait()
quello esterno viene “portato dietro” dal Thread che viene messo a nanna.

NMS 1 /

```
public class NestedMonitorSolution {
 private final Nested _nested = new Nested (); // nota! non √¨ accessibile
all'esterno
 public  void process () throws InterruptedException   {
_ nested.await (); // nessun lock acquisito
 }
 public  void set (boolean c)   {
_ nested.signal (c); // nessun lock acquisito
 }
 private class Nested {
  protected boolean cond = false;
  public synchronized void await () throws InterruptedException   {
  while (!cond)
  { wait (); } // .....
  }
 public synchronized void signal (boolean c)   {
  cond = c;
  notifyAll ();
  }
 }
}
```

Antipattern 10: il re dei re

Eccolo, lui! Il più infame e subdolo *Antipattern* della programmazione multithread in Java.

Per anni osannato come ottimo esempio di una ottimale gestione della concorrenza ....

... studi più approfonditi è risultato essere un letale *broken idiom* ...

Signore e Signori
Ecco a voi il peggiore di tutti i mali

Double checked locking idiom

```
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }


}
```

Monito

E' stato creato per *eliminare* un synchronized.

E' stato ideato per questioni di *performance.*

E' stato ideato "*ragionando troppo*" su come *eliminare* una sincronizzazione.

Quindi

Non pensate troppo su come eliminare la sincronizzazione, il JMM è subdolo.

First make it work, then make it fast !

Problemi!

A prima vista sembrerebbe che il DCL risolva il problema dei finti Singleton utilizzati nella *Lazy Instantiation*. Su questo si sono basati i sostenitori del DCL.

Tuttavia i problemi letali sono TRE, ovvero la triade della morte.

Tuttavia ...

```
public class UnsafeLazyInstatniation {
private static Resource resource;
    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
    // ---
}
```

Non è preso in considerazione il corretto ordinamento *happens-before*

JMM

Inizializzare un oggetto comporta la scrittura di alcune variabili (stato dell'oggetto), pubblicare un oggetto riguarda la scrittura di altre variabili (il *reference*).

Se non si assicura che *pubblicare l'oggetto* < un Thread possa *leggerne il reference* la scrittura del reference può essere riordinata con le scritture dello stato dell'oggetto.

Problema

In tale caso un Thread può vedere un valore aggiornato per il *reference*, ma un valore non aggiornato per alcune delle variabili che compongono lo stato dell'oggetto.

Si ottiene quindi il *reference* ad un oggetto parzialmente costruito.

Quindi

Non “giocate” troppo con l'ordine degli eventi, l'eliminazione della sincronizzazione non è un gioco da ragazzi e dipende spesso da fattori esterni all'oggetto di cui si vuole acquisire il *reference*.

Molti esperti Java hanno consigliato l'utilizzo del DCL, dopo un po di tempo e dopo molti ragionamenti si sono accorti dell'errore.

Non commettete il solito sbaglio. (*Don't try this at home*)

Concludendo ....

Siamo alla fine .....

Consigli

- Non buttatevi nell'ottimizzazione preventiva.
- Studiate il JavaMemoryModel.
- Utilizzate sempre forme canoniche.

Non abusate di questo ....

Sotto ***ben documentate*** assunzioni il DCL può anche funzionare correttamente ... anche se spesso queste assunzioni sono tali da *rendere inutile l'utilizzo* di un DCL. (L'inizializzazione statica basta ed avanza)

Non abusate delle buone pratiche

Inizializzare variabili con un inizializzatore statico, se fatto tramite una variabile public final può avere anche effetti collaterali ....

L'inizializzatore statico di una variabile (PRIVATA) è sufficiente!

Alla fine

Domande?

Grazie per l'attenzione!