

Java Memory Model

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Amdahl's law
- Introduzione alla concorrenza
- Java Memory Model
- Supporto hardware alla concorrenza
- Conclusioni



Amdahl's law

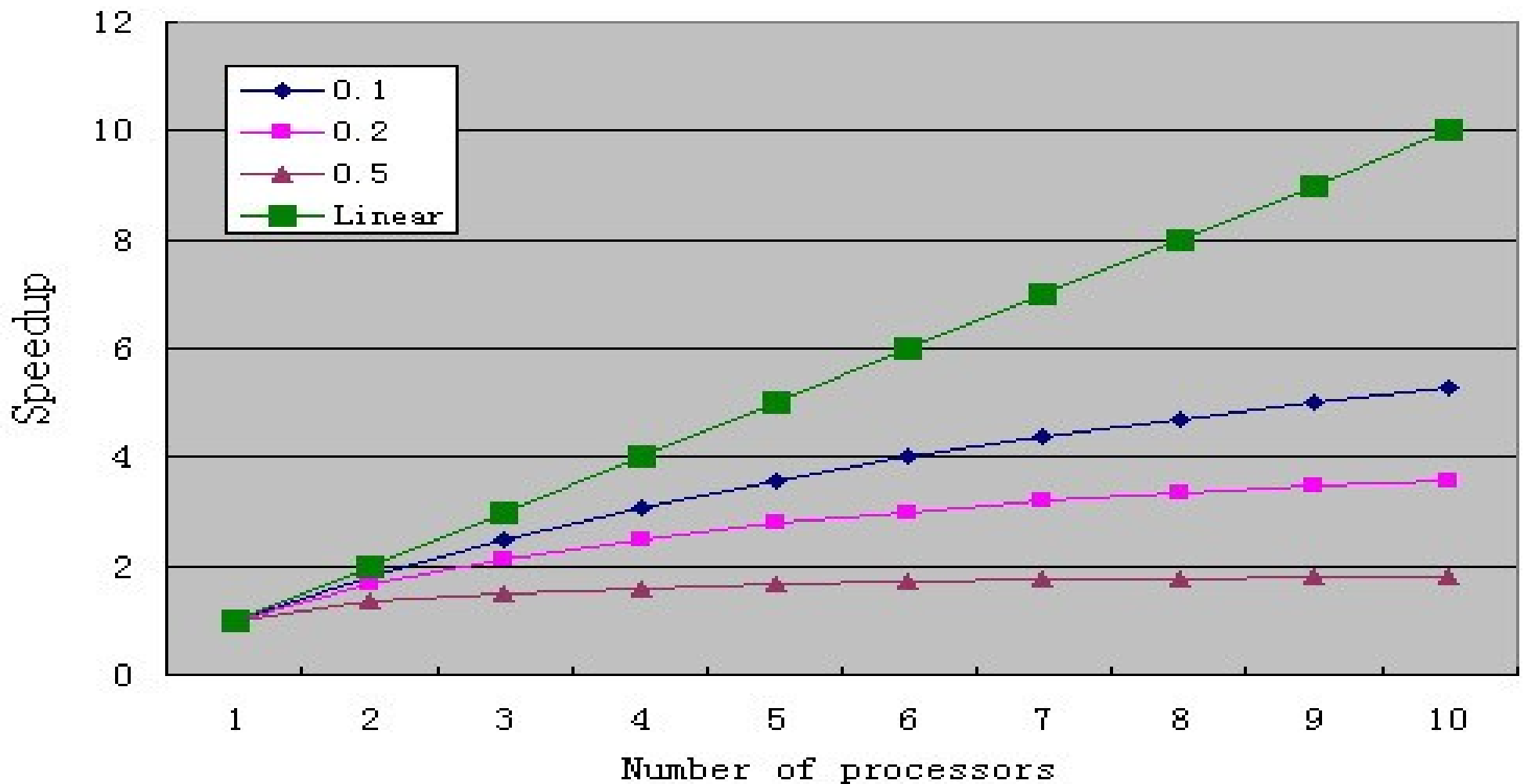
Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- “For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law. Writing code that effectively exploits multiple processors can be very challenging.”
 - *Doron Rajwan, Research Scientist, Intel Corp*



Amdahl's law

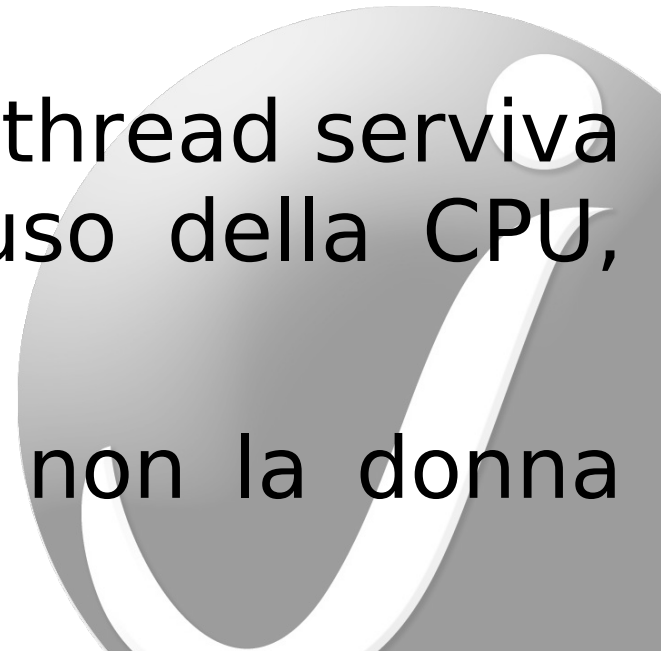
Amdahl's law:
Parallel speedup vs. Sequential fraction



più CPU, più core, più thread

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Le architetture multiprocessore sono ormai comunissime anche nel settore Desktop
- Nel 2007 la maggior parte dei desktop saranno attrezzati con CPU dual-core o quad-core
- E se prima programmare multithread serviva soprattutto per ottimizzare l'uso della CPU, ora diventerà un obbligo:
 - Serve il campo di grano, e non la donna incinta!



- A breve avremo cellulari biprocessore? :)

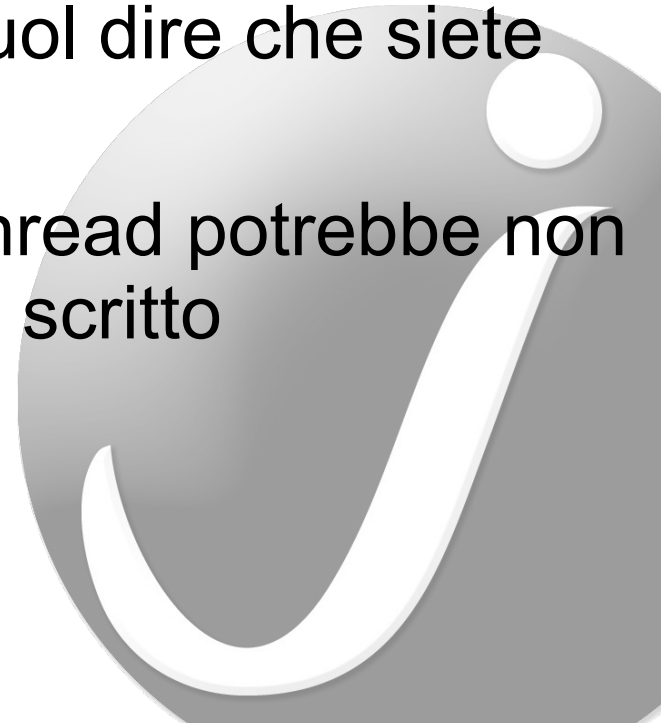
Cos'è il Java Memory Model?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Dato lo statement

risposta = 42;

- Il memory model si occupa di rispondere alla domanda “*In quali condizioni un thread che accede alla variabile **risposta** vedrà il valore 42?*”
 - Se la domanda vi sembra assurda, vuol dire che siete nella stanza giusta
 - Ci sono moltissimi motivi per cui un thread potrebbe non vedere il valore 42 DOPO che è stato scritto



Java Concurrency 101

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Prima di approfondire l'argomento JMM, ripassiamo un momento i concetti base della programmazione concorrente in Java
 - Java è stato il primo linguaggio mainstream ad includere un completo supporto alla programmazione concorrente
 - La classe **Thread**, l'interfaccia **Runnable**, i metodi **wait**, **notify** e **notifyAll** di Object, le parole chiave **synchronized** e **volatile** sono presenti fin da Java 1.0



Java concurrency 101

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- MyThread extends Thread
 - Thread t = new MyThread();
- Mythread implements Runnable
 - Thread t = new Thread(MyThread);
- t.start()
 - Invoca il metodo run in un altro contesto di esecuzione



Java concurrency 101

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- **synchronized** acquisisce il monitor dell'oggetto, se disponibile
- Se non è disponibile, attende
- Il monitor è una caratteristica della classe **Object**, per cui qualsiasi oggetto lo possiede
- Un metodo **synchronized** nelle prime JVM era di un'ordine di grandezza più lento di un metodo "normale"
- Ormai il costo dell'acquisizione del monitor *uncontended* è trascurabile



Java Concurrency in Java5

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Java5 ha portato un intero nuovo package con decine di classi (java.util.concurrent) ed una revisione del Java Memory Model e delle specifiche dei Thread
 - JSR 133
 - Thread Specification
 - JSR 166
 - Java Concurrency



Java Thread Specification

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

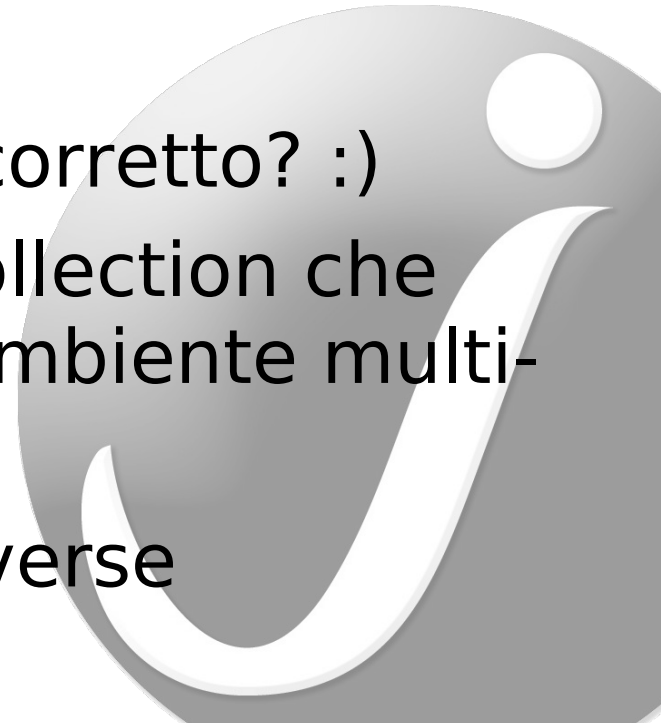
- JSR 133
 - Revisionata la gestione dei thread e del Java memory Model
- Obiettivi
 - Essere chiara e non ambigua
 - Rendere semplice lo sviluppo di applicazioni multithread corrette sotto ogni piattaforma
 - Permettere la scrittura di HIGH Performance JVM
- Le spec precedenti di fatto vietavano ottimizzazioni che le JVM però facevano



Concurrency utilities

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- JSR 166
 - Le primitive di sincronizzazione di Java sono troppo di basso livello
 - Come fare un read-write lock?
 - Con le primitive è troppo semplice commettere errori gravi
 - Come fare un read-write lock corretto? :)
 - Spesso quello che serve è una collection che sappia come comportarsi in un ambiente multi-thread
 - La JSR 166 ne ha introdotte diverse



Concurrency utilities

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Executors

- *Executor*
- *ExecutorService*
- *ScheduledExecutorService*
- *Callable*
- *Future*
- *ScheduledFuture*
- *Delayed*
- *CompletionService*
- *ThreadPoolExecutor*
- *ScheduledThreadPoolExecutor*
- *AbstractExecutorService*
- *Executors*
- *FutureTask*
- *ExecutorCompletionService*

Queues

- *BlockingQueue*
- *ConcurrentLinkedQueue*
- *LinkedBlockingQueue*
- *ArrayBlockingQueue*
- *SynchronousQueue*
- *PriorityBlockingQueue*
- *DelayQueue*

Concurrent Collections

- *ConcurrentMap*
- *ConcurrentHashMap*
- *CopyOnWriteArray{List,Set}*

Synchronizers

- *CountDownLatch*
- *Semaphore*
- *Exchanger*
- *CyclicBarrier*

Timing

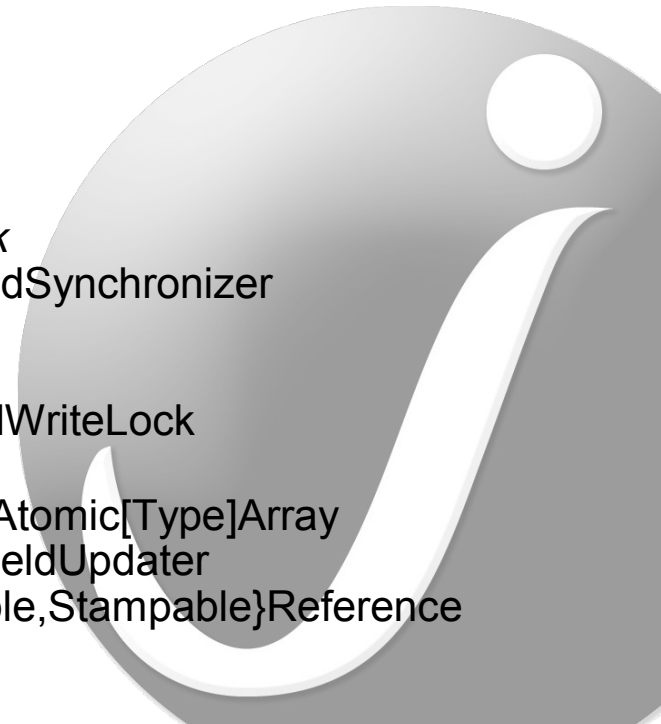
- *TimeUnit*

Locks

- *Lock*
- *Condition*
- *ReadWriteLock*
- *AbstractQueuedSynchronizer*
- *LockSupport*
- *ReentrantLock*
- *ReentrantReadWriteLock*

Atomics

- *Atomic[Type]*, *Atomic[Type]Array*
- *Atomic[Type]FieldUpdater*
- *Atomic{Markable,Stampable}Reference*



Thread safe: definizione

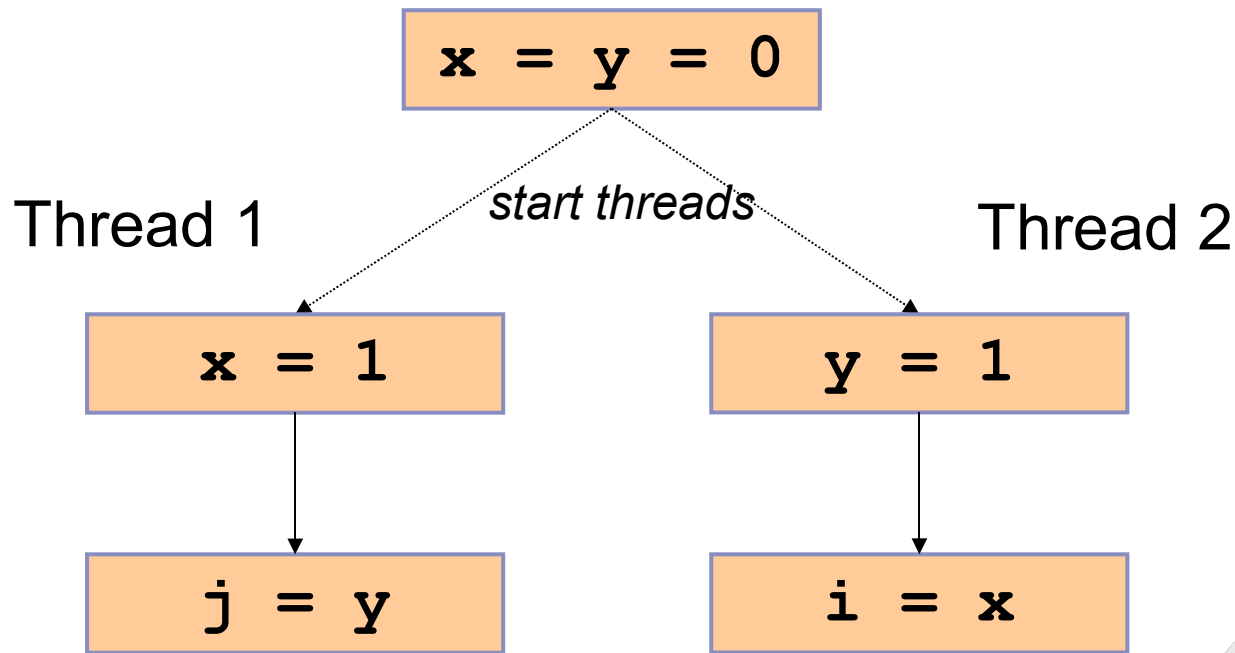
Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Una classe è thread-safe quando si comporta correttamente se acceduta da thread multipli, a prescindere dallo scheduling dei thread e senza bisogno di ulteriore sincronizzazione da parte del codice chiamante



Test

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

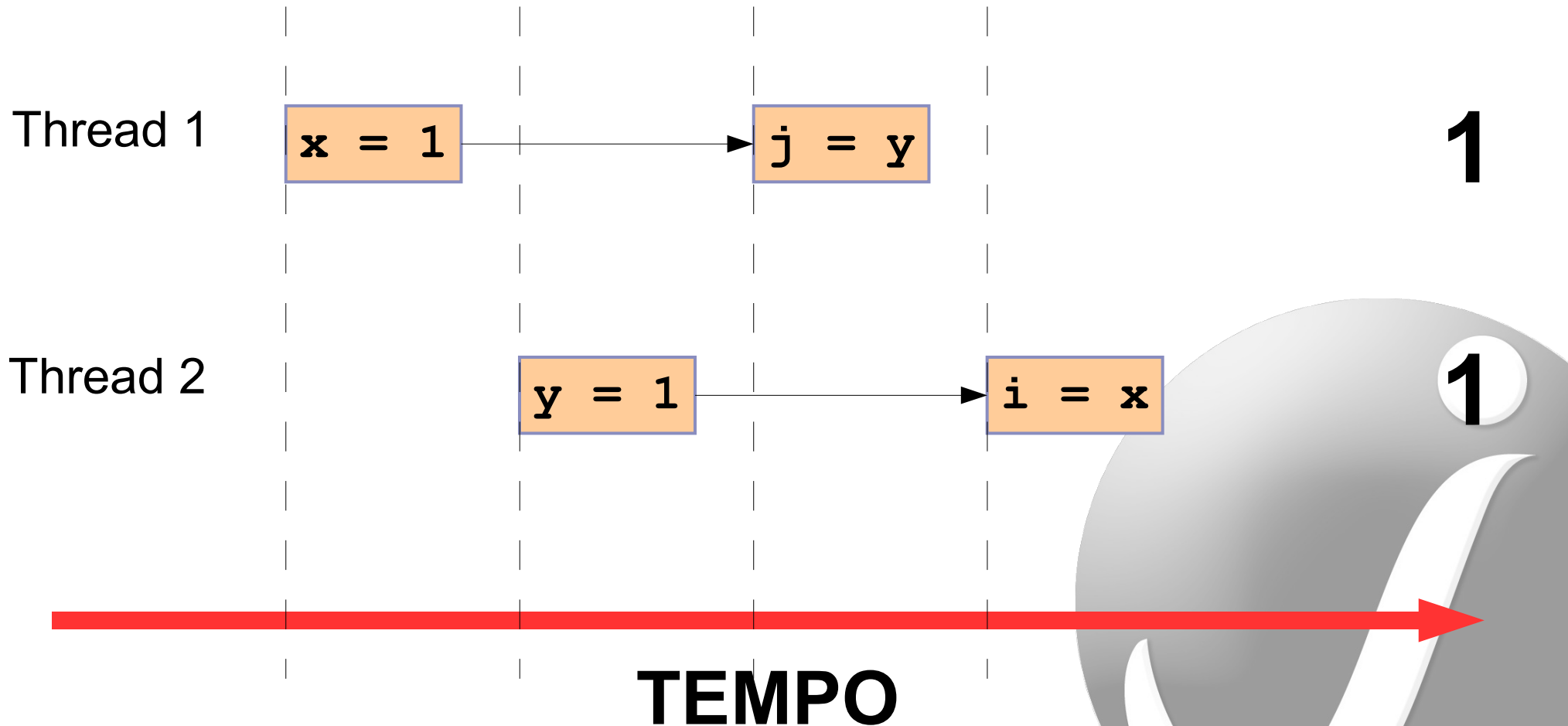


**Dopo la join dei due thread,
quale sarà l'output di i e j?**



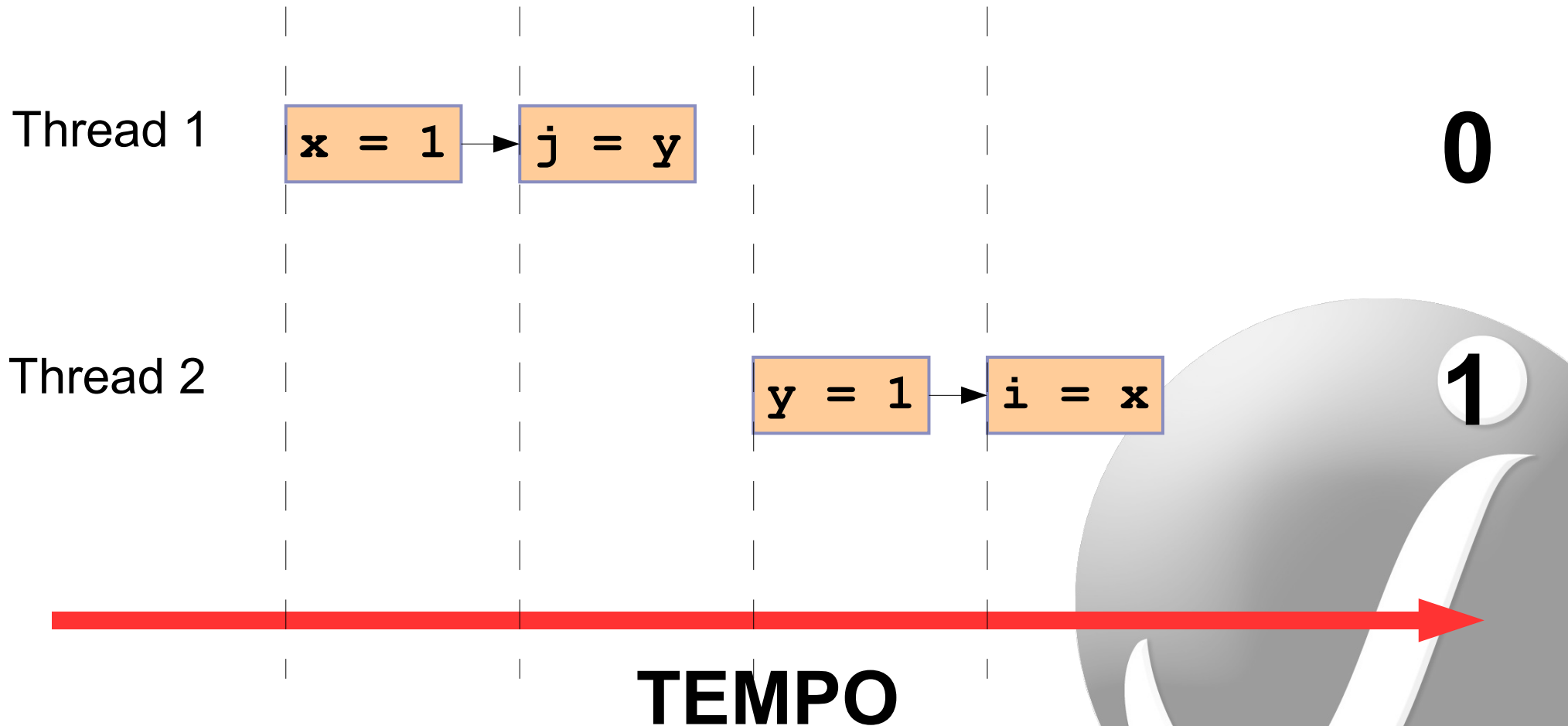
Output possibile

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License



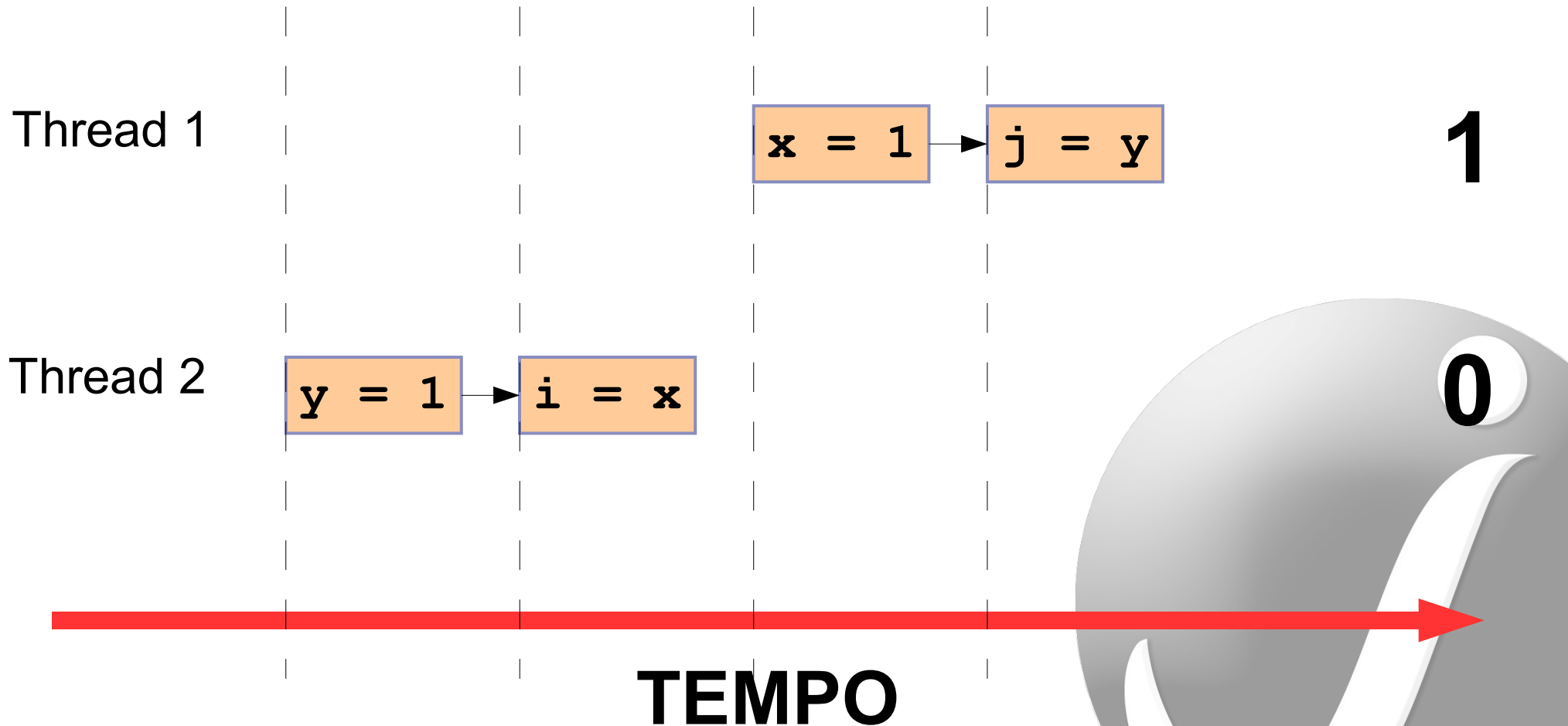
Output possibile

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License



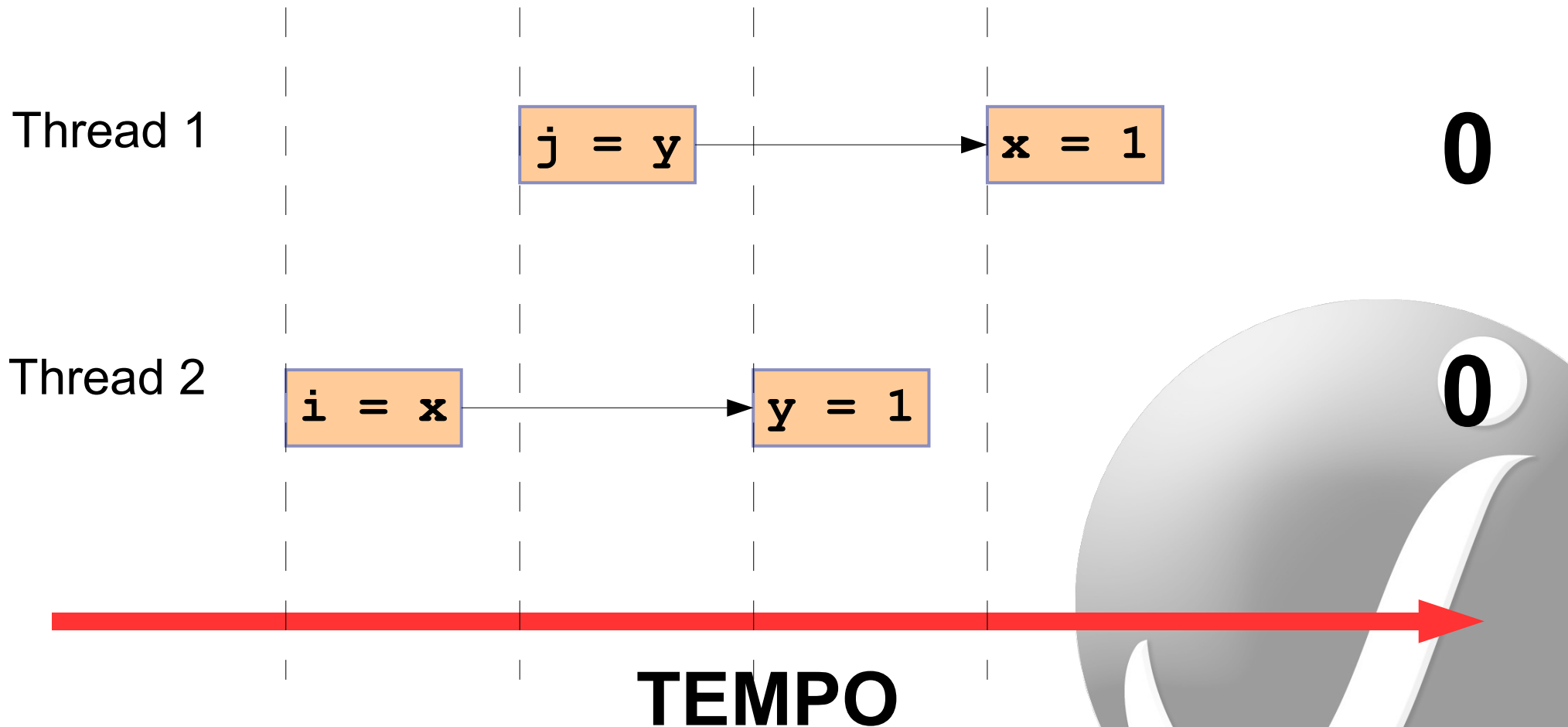
Output possibile

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License



Ma anche questo!

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License



Che è successo?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

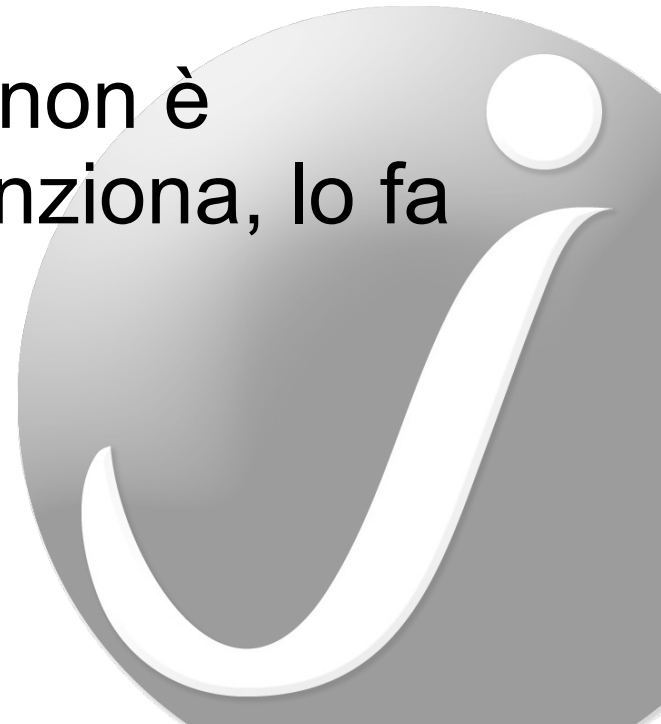
- **synchronized** non si occupa solo dell'acquisizione del monitor (Atomicità)
- I 3 aspetti di **synchronized** sono
 - **Atomicità**
 - Locking e mutua esclusione
 - **Visibilità**
 - Assicurarci che i cambiamenti fatti da un thread siano visibili ad un altro thread
 - **Ordinamento**
 - Assicurazione sull'ordine di esecuzione degli statement



Data Race

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Dal punto di vista del JMM, se in un punto qualsiasi del codice le tre condizioni di seguito sono soddisfatte c'è un data race
 - Un thread scrive una variabile
 - Un altro thread legge la variabile
 - I read e i write non sono ordinati
- Se c'è un data race, il programma non è correttamente sincronizzato. Se funziona, lo fa per sbaglio :)



Happens Before (HB)

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Il Java Memory Model definisce una relazione di ordinamento parziale chiamata “Happens Before” (HB) sulle azioni di un programma
 - Una relazione di ordinamento parziale è algebricamente una relazione antisimmetrica, riflessiva e transitiva.
 - A differenza di una relazione di ordinamento totale, è possibile che alcuni elementi non siano in relazione con altri.
 - Java è meglio di C#. Smalltalk è meglio di Python. Fra C# e Python non ho preferenze



Happens Before (HB)

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

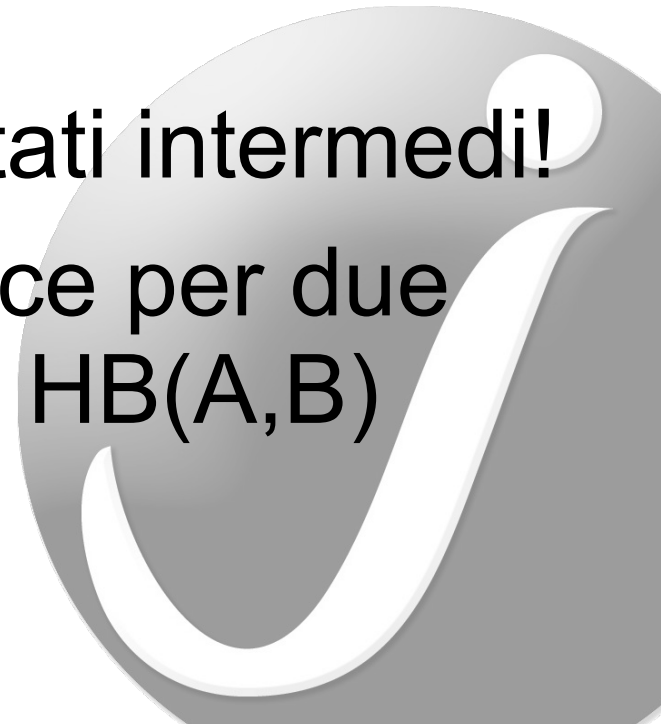
- La JVM **garantisce** che una azione A sarà visibile ad una azione B se c'è una relazione di Happens Before:
 - $HB(A,B)$
- Ci sono diverse regole per l'HB
 - Program Order
 - Finalizer rule
 - Synchronization rule (lock, volatile, Thread start, Thread termination, Interruption)
 - Transitivity: se $HB(x,y)$ e $HB(y,z) \rightarrow HB(x,z)$



Happens Before (HB)?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

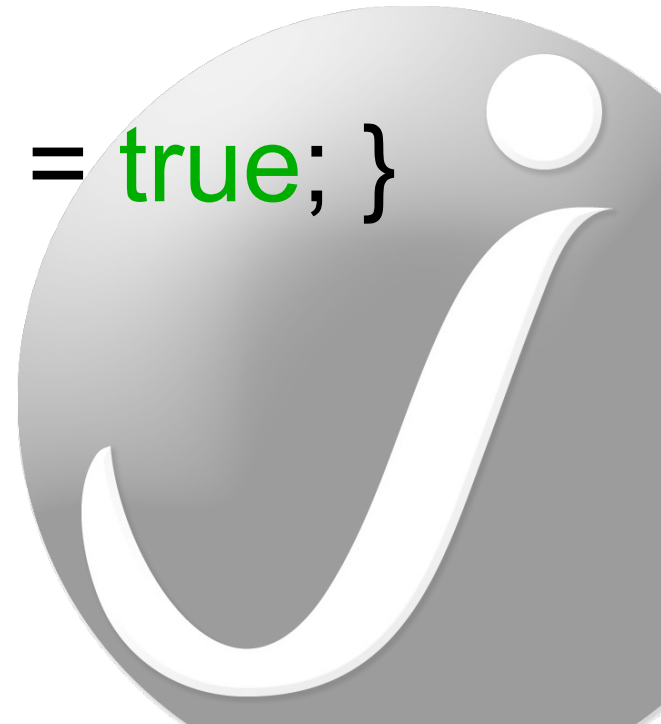
- Cosa vuol dire in pratica?
- Se non c'è una relazione $HB(A,B)$, i compilatori possono ottimizzare e riordinare gli statement come meglio credono, lasciando invariato il risultato finale
 - Ma anche modificando i risultati intermedi!
- Quindi per eliminare un data-race per due azioni A e B, si deve forzare un $HB(A,B)$



Interruzione di un thread

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
class MyThread extends Thread {  
    public void run() {  
        while (!done) { // fai qualcosa ... }  
    }  
    public void setDone() { done = true; }  
    private boolean done;  
}
```



Interruzione di un thread

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Se un Thread mette la variabile done a true
- Chi garantisce che un altro Thread veda il valore di done?
- Se le due azioni (lettura e scrittura) non hanno una relazione di Happens Before, quindi non c'è nessuna garanzia che ciò avvenga mai
- E' dunque teoricamente possibile – seppur improbabile - che il Thread non esca mai, soprattutto in un'architettura SMP!



Interruzione di un thread

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Affinchè il codice sia corretto, bisogna instaurare una relazione di HB fra la lettura e la scrittura, evitando così il data-race
 - Usando un lock esplicito (synchronized o JSR 166)
 - Usando la keyword volatile
 - Usando una variabile atomica
 - `java.util.concurrent.atomic`



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

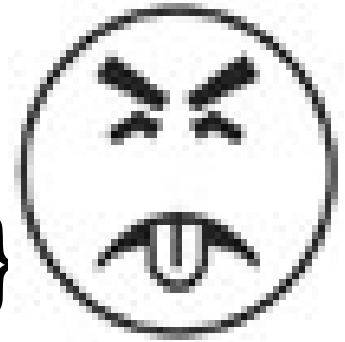
```
class MyThread extends Thread {  
    public void run() {  
        while (!done) { // fai qualcosa ... }  
    }  
    public synchronized void setDone()  
    { done = true; }  
    private boolean done;  
}
```



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
class MyThread extends Thread {  
    public void run() {  
        while (!done) { // fai qualcosa ... }  
    }  
    public synchronized void setDone()  
    { done = true; }  
    private boolean done;  
}
```



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Ovviamente no
- Sincronizzare solo le scritture non instaura una relazione di HB fra le letture e le scritture
- La lock rule dice infatti che un *unlock* Happens Before un *lock* sullo stesso monitor
- Per evitare il data-race, bisogna sincronizzare anche le letture!
 - o usare volatile che fa l'uno e l'altro



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
class MyThread extends Thread {  
    public void run() {  
        while (!done()) { // fai qualcosa ... }  
    }  
    public synchronized void setDone() {  
        done = true; }  
    private synchronized void done() {  
        return done; }  
    private boolean done;  
}
```



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
class MyThread extends Thread {  
    public void run() {  
        while (!done()) { // fai qualcosa ...  
        }  
    }  
    public synchronized void setDone() {  
        done = true; }  
    private synchronized void done() {  
        return done; }  
    private boolean done;  
}
```



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

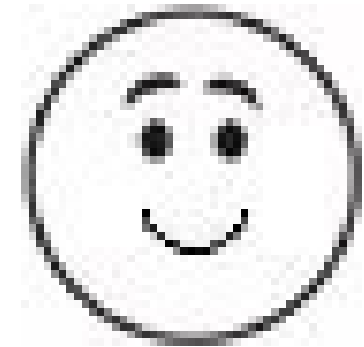
- Sì, la variabile è correttamente sincronizzata
- Si può però usare **volatile** che garantisce la stessa semantica per la visibilità e l'ordering
- Ma, attenzione, non per l'atomicità!
- Per un semplice test di uscita da un Thread, **volatile** è perfetta



Test: così funziona?

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
class MyThread extends Thread {  
    public void run() {  
        while (!done) { // fai qualcosa ... }  
    }  
    public void setDone() { done = true; }  
  
    private volatile boolean done;  
}
```



CAS: supporto hardware

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- CAS
 - **Compare and Swap**
 - E' un istruzione Assembler comune nelle nuove CPU (Intel, AMD, Sparc)
 - PowerPc usa due istruzioni: *load-linked* e *store-conditional*
- E' uno dei motivi per cui oggi è possibile ottenere altissime performance nella programmazione concorrente
- E' una sorta di *optimistic-locking* a livello hardware



CAS: supporto hardware

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- CAS(Address, Expected, New)
 - Address = Indirizzo di memoria
 - Expected = Il valore atteso, ossia il vecchio valore
 - New = Il nuovo valore
- CAS(123456, 42, 127)
 - Se all'indirizzo 123456 c'è 42, allora scriverà 127
 - Altrimenti non fa nulla
- CAS ritorna il valore alla locazione Address, in entrambi i casi



CAS e JVM

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

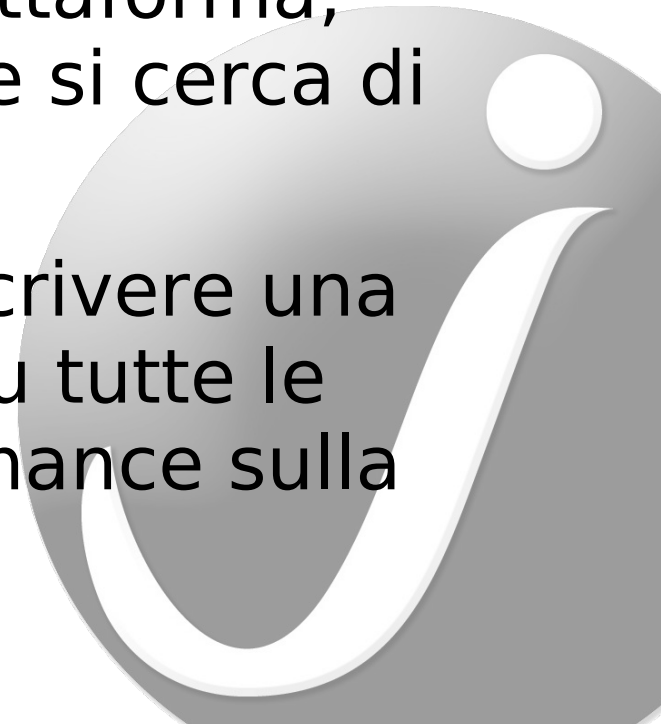
- Supporto a basso livello per int, long e object references (la JVM usa uno *spinlock* su CPU senza CAS)
- Atomic class (*AtomicInteger*, *AtomicLong*, ecc. del package *java.util.concurrent.atomic*)
- La maggioranza delle classi in *java.util.concurrent* usano al loro interno le variabili atomiche al posto dei classici lock (*synchronized*)
- Quindi può succedere, da Java5 in poi e su CPU di nuova generazione, di usare delle classi che sono multithreaded e thread-safe ma che non usano *synchronized*



Conclusioni

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Con i nuovi processori multi core scrivere applicazioni multithread sarà sempre più importante e comune
- La programmazione multithread è però in assoluto la più difficile. In Java si ha un supporto come in nessuna altra piattaforma, ma la difficoltà resta. Soprattutto se si cerca di fare i furbi!
- Con Java 5 è finalmente possibile scrivere una applicazione multithread corretta su tutte le piattaforme ed ad altissime performance sulla maggior parte di esse



Riferimenti

Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- JSR 133
 - www.cs.umd.edu/~pugh/java/memoryModel/
- JSR 166
 - <http://g.oswego.edu/dl/concurrency-interest/>
- Java Concurrency in practice
 - Brian Goetz
- Concurrent programming in Java
 - Doug Lea

