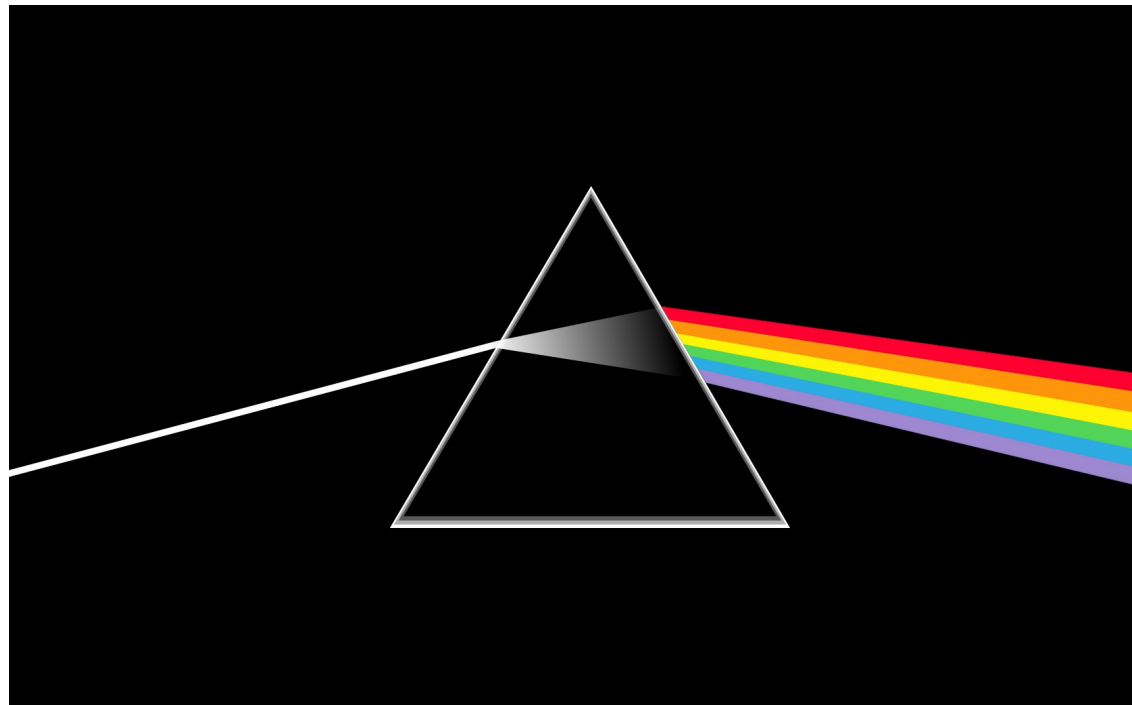


Multiprocessor and Memory Models

The dark side of ... concurrency



Uno sguardo alle architetture

Le architetture moderne hanno diversi gradi di parallelismo per supportare concorrenza ed esecuzione parallela

- Parallelismo: rendere efficiente l'esecuzione di codice eseguendo in parallelo parti di codice; spesso comportamento deterministico (es. algoritmi paralleli su matrici)
- Concorrenza: introduce modularità nel codice, sfrutta quando possibile parallelismo attraverso il multithreading, comportamento non deterministico (diverse esecuzioni possono portare a risultati diversi)

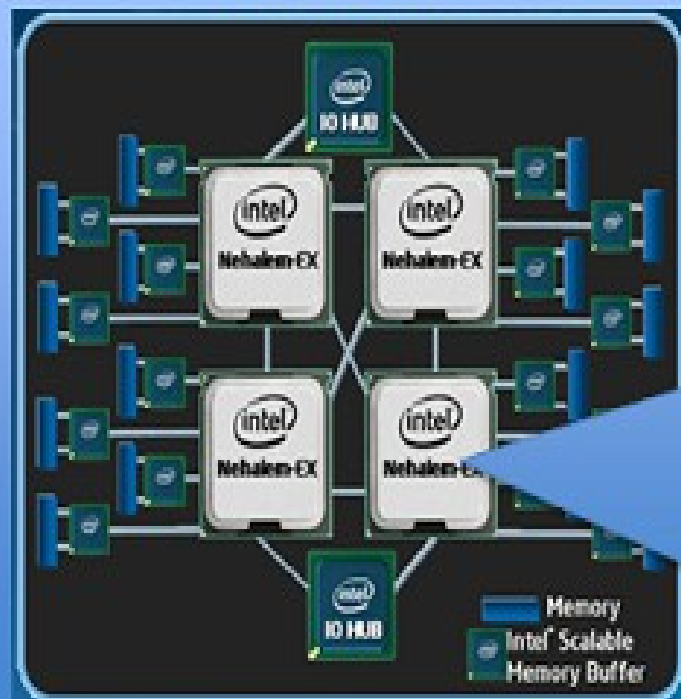
Multithreading e parallelismo

Le architetture moderne hanno diversi gradi di parallelismo per supportare concorrenza ed esecuzione parallela

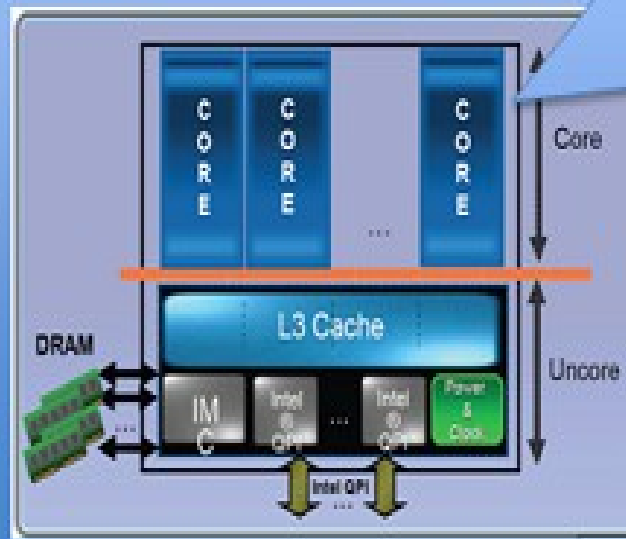
- Multicore: core multipli con diversi livelli di caching (shared e privati)
- Hyperthreading: due o più core logici in un core fisico (tecnologia Intel)
- SIMD: singola istruzione eseguita in parallelo su diverse ALU (nello stesso core)
- ILP: esecuzione in fasi di stream di istruzioni (pipelining)

Intel

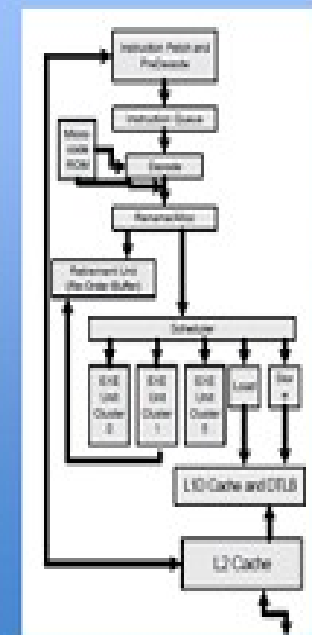
SYSTEM



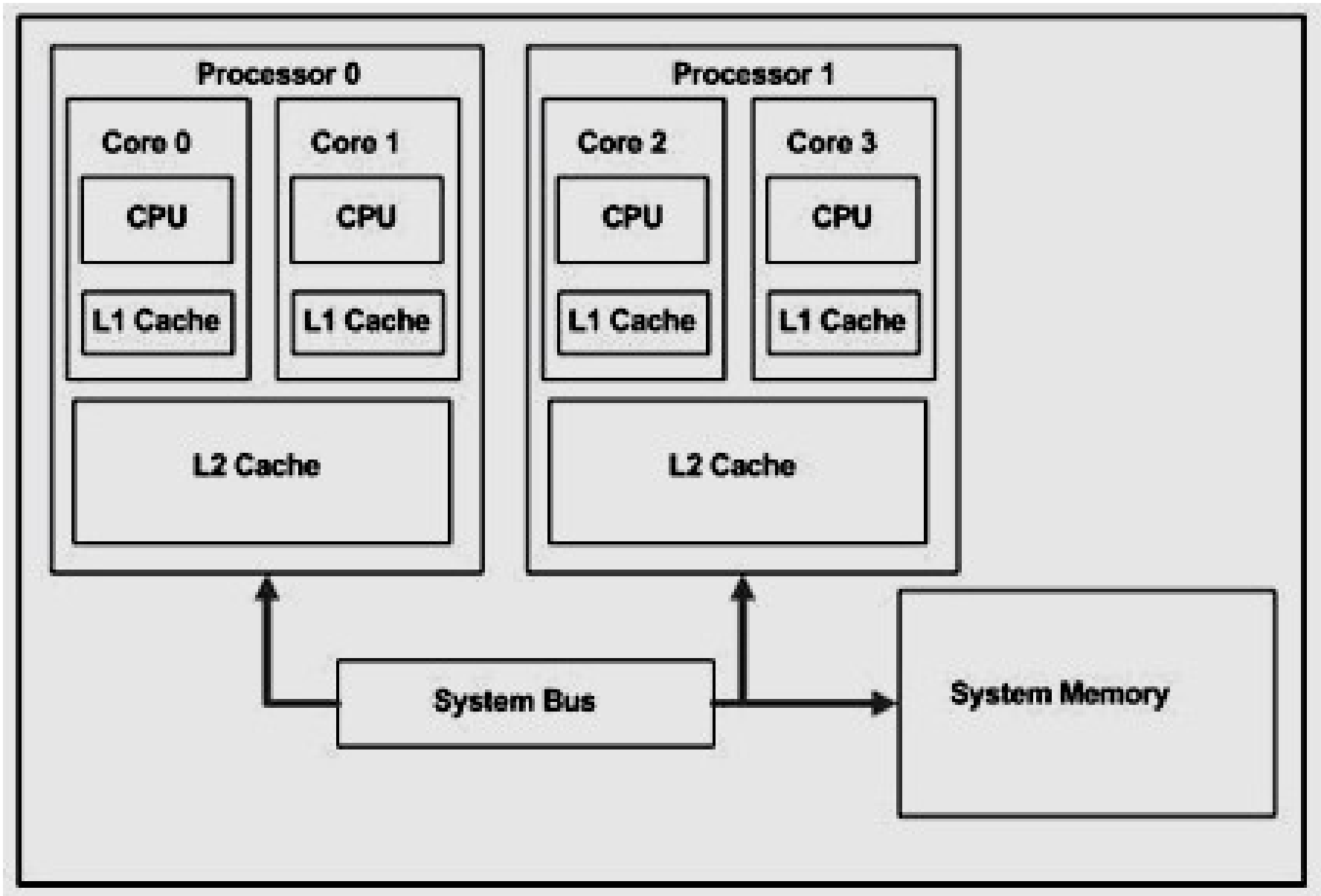
SOCKET (Processor)



CORE



Esempio



Impatto sulla programmazione?

I sistemi hanno diversi livelli di astrazione

- Linguaggio ad alto livello (es. compilatore)
- Sistema operativo (es. multithreading)
- Hardware (es. caching)

cerchiamo di capire l'impatto di ognuno di essi
nella programmazione concorrente con qualche
esempio

Single Thread

a=Flag=0;

Thread

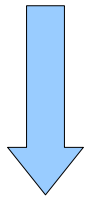
a=26;
Flag=1;

Compiler Optimizations

a=Flag=0;

Thread

a=26;
Flag=1;



Trasformazione del compilatore

Flag=1;
a=26; **OK!**

Multithread

a=Flag=0;

Thread 1

a = 26;

Flag = 1;

Thread 2

while (Flag != 1) {};

b = a;

Che valore ha b alla fine dell'esecuzione?

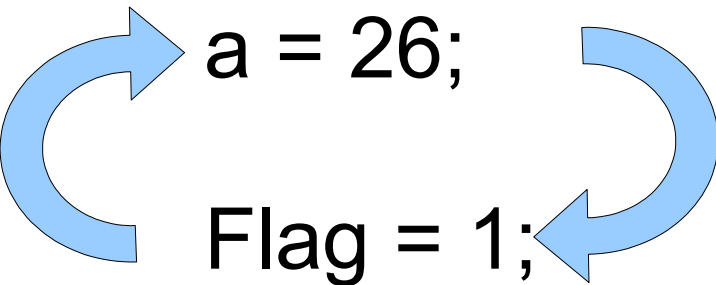
b=26 OK

Compiler Optimization

a=Flag=0;

Thread 1

Thread 2



Trasformazione
del compilatore

while (Flag != 1)
{ }; %skip

b = a;

Se il compilatore scambia le istruzioni?

Compiler Optimization

a=Flag=0;

Thread 1

Flag = 1; (1)

a = 26; (4)

Thread 2

while (Flag != 1) (2)
{ }; %skip

b = a; (3)

Alla fine b potrebbe essere = 0!

Senza Ottimizzazione Compilatore?

a=Flag=0;

Thread 1

a = 26;

Flag = 1;

Thread 2

while (Flag != 1)
{ };

b = a;

Disabilitiamo possibili riordinamenti....
cosa otteniamo?

Hardware Out-of-Order Execution

a=Flag=0;

Thread 1

a = 26;

Flag = 1;

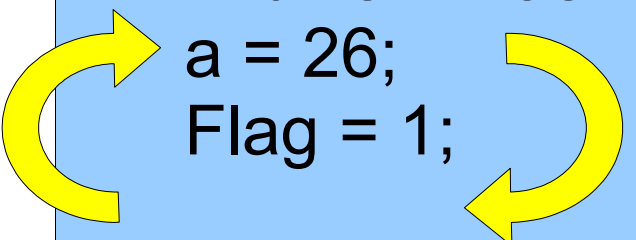
Thread 2

while (Flag != 1)
{ };

b = a;

Buffer Proc 1

a = 26;
Flag = 1;



**Riordinamento del buffer
(sequenza delle write) del
processore**

Hardware Out-of-Order Execution

a=Flag=0;

Thread 1

a = 26;

Flag = 1;

Thread 2

while (Flag != 1)
{ };

b = a;

Buffer Proc 1

Flag = 1;
a = 26;

Potrei ottenere ancora 0!!

Memory Model

Cos'è un Memory Model

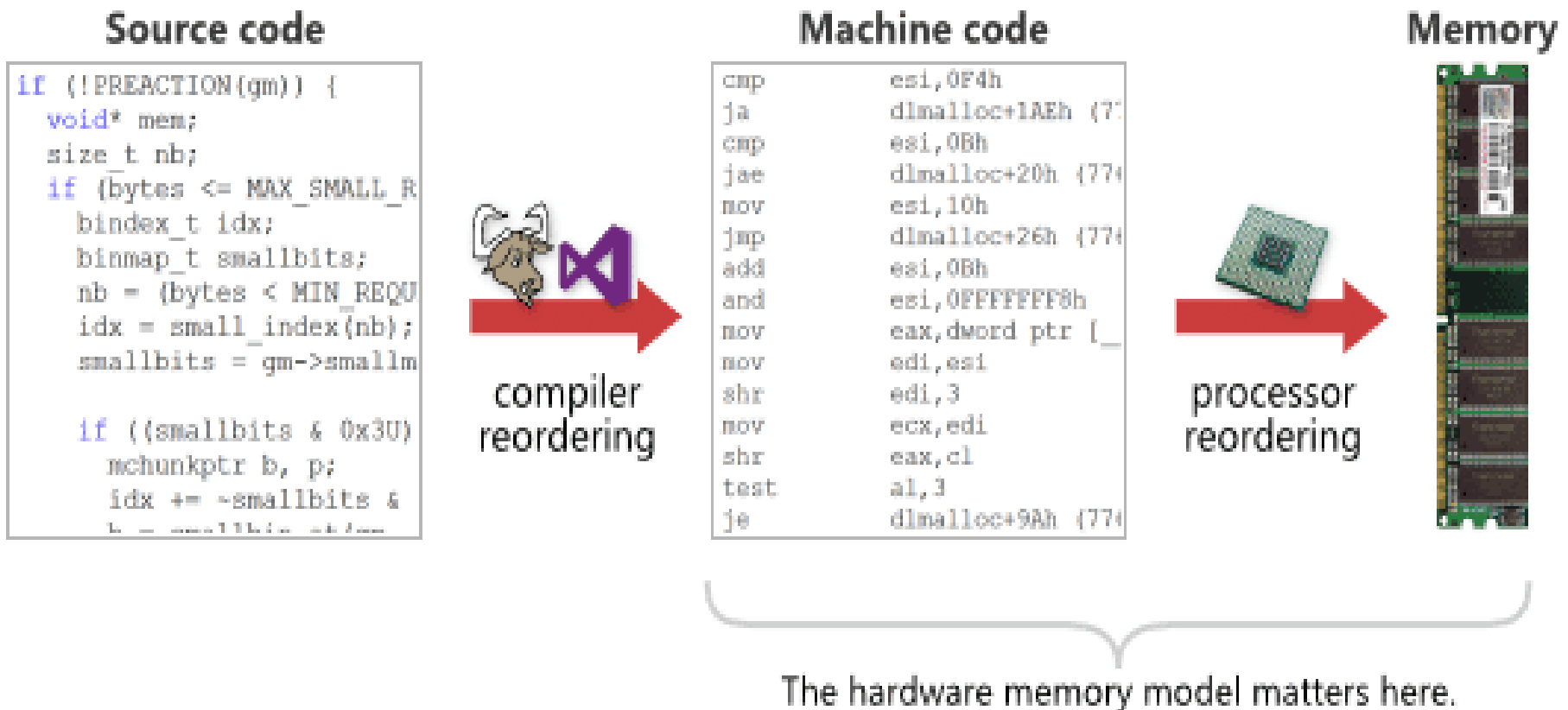
- Specifica formale di come la gestione della memoria appare al programmatore
- Elimina il gap tra il comportamento che si aspetta il programmatore e il comportamento reale del sistema

Obiettivi

- Il memory model deve specificare:
 - L'interazione tra thread e memoria
 - Quale valore può restituire una read
 - Quando una modifica diventa visibile agli altri thread
 - Quali assunzioni si possono fare sul comportamento della memoria quando scriviamo un programma o applichiamo (il compilatore applica...) un'ottimizzazione

Perchè ci interessa?

- Il memory model influenza programmabilità, performance e portabilità di un programma



Single Thread

- Accessi in memoria eseguiti uno alla volta
- Read restituisce l'ultima write
- Riordinamenti/trasformazioni del codice:
 - Le ottimizzazioni devono rispettare le dipendenze tra dati e controllo (es. se i dati sono indipendenti posso scambiare istruzioni)
 - Le operazioni sulla memoria devono seguire l'ordine del programma

Multi Thread: Strict Consistency

- Ogni operazione in memoria viene vista nell'ordine temporale in cui viene eseguita dai diversi thread
- Praticamente impossibile da ottenere in un sistema concorrente
- Solitamente si utilizzano nozioni più deboli che non rispettano l'ordine temporale di esecuzione ma l'ordine del programma e i possibili interleaving

Strict Consistency

Ogni lettura di X restituisce l'ultimo valore scritto

Thread 1

X=1;

Thread 2

R1=X;
R2=X;

Timeline

X=1 R1=1 R2=1

OK

Strict Consistency

Ogni lettura di X restituisce l'ultimo valore scritto

Thread 1

X=1;

Thread 2

R1=X;

R2=X;

Timeline

X=1 R1=0 R2=1

OK

Strict Consistency

Ogni lettura di X restituisce l'ultimo valore scritto

Thread 1

X=1;

Thread 2

R1=X;
R2=X;

Timeline

X=1 R1=0 R2=1

NO!

Sequential Consistency

[Lamport 79]

Un multiprocessore (sistema multithreaded) è **sequentially consistent** (SC) se il risultato di ogni sua esecuzione è lo stesso di un'esecuzione sequenziale delle istruzioni di tutti i processori (thread) e le istruzioni di ogni processore (thread) sono eseguite nell'ordine del corrispondente programma

SC richiede:

program order: rispetta ordine nel programma

write atomicity: tutte le write devono apparire a tutti i thread nello stesso ordine

Sequential Consistency

Thread 1

X=1;

Thread 2

R1=X;
R2=X;

Timeline

X=1 R1=0 R2=1

OK

Sequential Consistency

Thread 1

X=1;

Thread 2

R1=X;
R2=X;

Timeline

X=1 R1=1 R2=0

NO!

Esempio: Dekker's Algorithm

Flag1=Flag2=0;

Thread 1

Flag1=1;

if (Flag2==0)
 critical section

Thread 2

Flag2=1;

if (Flag1==0)
 critical section

Esempio: Dekker's Algorithm

Flag1=Flag2=0;

Thread 1

Thread 2

Flag1=1;

Flag2=1;

if (Flag2==0)
critical section

if (Flag1==0)
critical section

Flag1 = Flag2 = 1

Esempio: Dekker's Algorithm

Flag1=Flag2=0;

Thread 1

Flag1=1;

if (Flag2==0)
critical section

Thread 2

Flag2=1;

if (Flag1==0)
critical section

Flag1 = 0 Flag2 = 1

Violazione Mutua Esc

Esempio: Dekker's Algorithm

Flag1=Flag2=0;

Thread 1

Flag1=1;



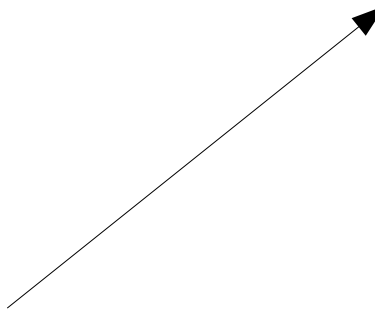
if (Flag2==0)
critical section

Thread 2

Flag2=1;

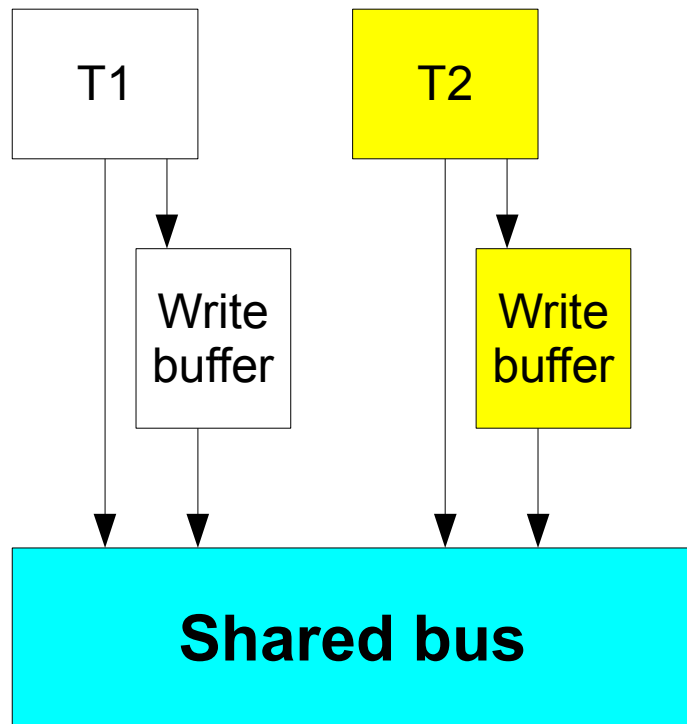


if (Flag1==0)
critical section



NO sequential consistency → Possibili violazioni mutua esclusione

Esempio di hardware violation to SC: Write buffers + read bypassing



Flag1=Flag2=0	
Thread T1	Thread T2
Flag1=1	Flag2=1
if (Flag2==0) critical section	if (Flag1==0) critical section

(1) read Flag2=0 (T1)

[bypass Flag1=1 messa nel buffer di T1]

(2) read Flag1=0 (T2)

[bypass Flag2=1 messa nel buffer di T2]

(3) write Flag1 (T1)

(4) write Flag2 (T2)

**Alla fine otteniamo violazione
della mutua esclusione**

Esempio di hardware violation to SC: Violation to Write Atomicity

A=B=tmp=0

Thread T1

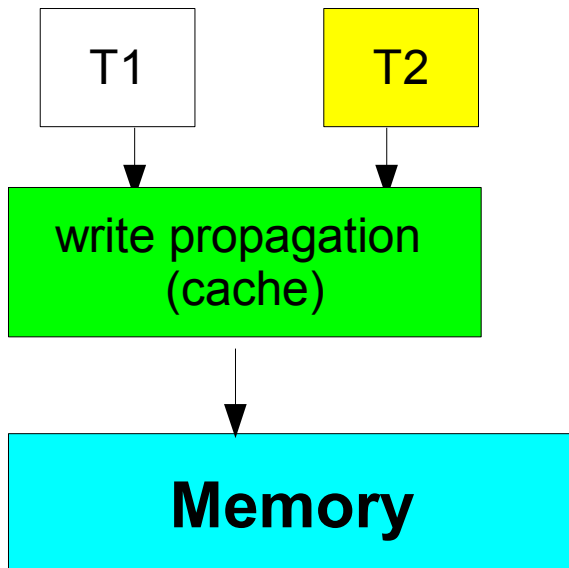
A=1

Thread T2

if (A==1)
B=1

Thread T3

if (B==1)
tmp=A



(1) write A,1 (T1)

(2) write B,1 (T2)

(3) read B,1 (T3)

(4) read A,0 (T3)

Modifica di B propagata prima di quella di A
Alla fine otteniamo tmp=0

Sequential Consistency e Cache

- Per processori con cache locali occorre utilizzare un protocollo di consistenza tra memoria e linee di cache associate ad una certa locazione
- Ad esempio in caso di aggiornamento di una certa locazione di memoria:
 - Ciclo di invalidazione o di aggiornamento di tutte le linee associate ad L!

Code Optimization che viola SC

Thread T1

```
for(i=0;i<10;i++)  
    *a = i;
```

Thread T2

```
while (true)  
    b = *a;
```

Non si può sposare il caricamento di a fuori dal ciclo in T2

Code Optimization che viola SC

Thread T1

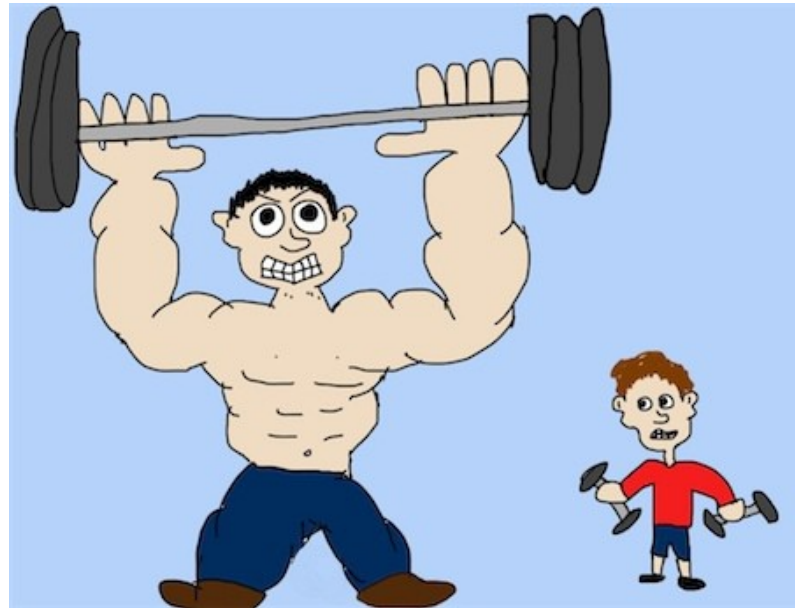
```
a=6;  
Flag=1;
```

Thread T2

```
c = a - 1;  
while (Flag == 0) {}  
b = a - 1;
```

La sottoespressione comune $a-1$ non si può eliminare dal secondo assegnamento

Strong vs Weak Memory Models



WEAK

STRONG

Really weak



Weak with
data dependency
ordering



Usually strong
(implicit acquire/
release & TSO, usually)



Sequentially
consistent

DEC Alpha



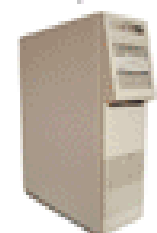
ARM



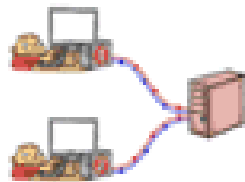
x86/64



dual 386 (circa 1989)



C/C++11
low-level atomics



Source control
analogy

PowerPC



SPARC TSO



Java volatile

C/C++11
default atomics

Or, run on
a single core
without optimization

ARM:a data access that misses in the cache can be overtaken by other data accesses that hit (or miss) in the cache, as long as there are no data dependencies between them.

Relaxed Memory Models

Rilassiamo i requisiti per avere SC

- Program order (locazioni diverse)
 - Si ammettono reorder es. read to write
- Write atomicity (stessa locazione)
 - Read other write early
 - Read own write early (leggi dal write buffer)
- Es. Total Store Order (TSO)=RtoW+RownWE

Weak (Relaxed) Memory Models

- Hardware
 - Write buffer+read bypassing
 - Cache locali ai processori

x86 (Strong Memory: ogni operazione acquire/release)

ARM, PowerPC (Weak Memory with Data Dependency)

Come si lavora con Weak Memory?

- Costrutti di sincronizzazione
 - volatile in Java (simula Sequential Consistency)
 - synchronize
 - librerie atomiche
- Garanzie compilatore/hardware:
 - Memory barrier: assicura che tutte le read successive (alla memory barrier) verranno eseguite dopo il completamento delle write precedenti